Uma infra-estrutura de suporte à evolução para repositórios de componentes

Este exemplar corresponde à redação da Dissertação apresentada para a Banca Examinadora antes da defesa da Dissertação.

Campinas, 2 de Março de 2007.

Profa. Dra. Cecília Mary Fischer Rubira
(Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Instituto de Computação
Universidade Estadual de Campinas

Uma infra-estrutura de suporte à evolução para repositórios de componentes

Leonardo Pondian Tizzei[1]

2 de Março de 2007

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)
- Prof. Dr. Marcos Lordello Chaim
  Escola de Artes, Ciências e Humanidades — Universidade de São Paulo
- Prof. Dr. Luiz Eduardo Buzato
  Instituto de Computação — UNICAMP
- Profa. Dra. Ariadne Maria Brito Rizzoni Carvalho (suplente)
  Instituto de Computação — UNICAMP

[1]**Apoio do CNPq - processo no. 131817/2005-1**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecária: Miriam Cristina Alves – CRB8a / 5094

Tizzei, Leonardo Pondian

T546i Uma infra-estrutura de suporte à evolução para repositórios de componentes / Leonardo Pondian Tizzei -- Campinas, [S.P. :s.n.], 2007.

Orientadora: Cecília Mary Fischer Rubira

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Software (Evolução). 2. Software – Reutilização. 3. Software – Desenvolvimento. 4. Componentes de software. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

(mca/imecc)

Título em inglês: An infrastructure to support evolution in component repositories

Palavras-chave em inglês (Keywords): 1. Software (Evolution). 2. Software – Reuse. 3. Software – Development. 4. Software components.

Área de concentração: Engenharia de software

Titulação: Mestre em Ciência da Computação

Banca examinadora: Profa. Dra. Cecília Mary Fischer Rubira (IC-Unicamp)
Prof. Dr. Marcos Lordello Chaim (EACH-USP)
Prof. Dr. Luiz Eduardo Buzato (IC-Unicamp)

Data da defesa: 02/03/2007

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 02 de março de 2007, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Marcos Lordello Chaim
USP – Leste.

Prof. Dr. Luiz Eduardo Buzato
IC / UNICAMP.

Profa. Dra. Cecília Mary Fischer Rubira
IC – UNICAMP.

Resumo

No contexto do Desenvolvimento Baseado em Componentes(DBC), o elo de ligação entre desenvolvedores e consumidores de componentes de software são os repositórios de componentes, onde eles são adicionados por seus desenvolvedores e recuperados pelos integradores de sistemas. Considerando um cenário de evolução, onde componentes são modificados, são imprescindíveis modelos de evolução de componentes para definir uma abordagem sistemática de mudanças. Elas podem ocorrer, por exemplo, na especificação ou implementação de um componente e o repositório de componentes deve dar apoio à evolução uma vez que ela é parte inerente do processo de desenvolvimento de software. Contudo, assim como o software, modelos de evolução são modificados para acompanhar o desenvolvimento tecnológico e as necessidades de seus usuários. Além disso, diferentes produtores de componentes podem possuir modelos distintos.

Este trabalho apresenta uma infra-estrutura de suporte à evolução em um repositório de componentes, para apoiar serviços de DBC, como a conversão de componentes para outros modelos de implementação, extração de metainformação de código-fonte de componentes, evolução de componentes e verificação de modelos de implementação de componentes. No caso particular desta dissertação, o repositório de componentes escolhido é o Rigel, que foi estendido para dar suporte à evolução de componentes. O repositório Rigel oferece as condições necessárias para adoção de um modelo de evolução de componentes, através da utilização de um padrão de metadados de componentes chamado RAS.

Ferramentas de software ajudam a aumentar a produtividade de desenvolvedores e evitar tarefas repetitivas. Além disso, atividades de modificação são sujeitas a erros humanos. Portanto, foram desenvolvidas quatro ferramentas para dar suporte ao modelo de evolução SACE e ao modelo de implementação de componentes COSMOS. As ferramentas foram construídas com base em um *framework* de componentes baseado em regras que usa um motor de inferência chamado Drools. Este *framework* de componentes externaliza as regras da aplicação, garatindo uma maior modificabilidade, característica que é essencial para que as ferramentas possam evoluir em conformidade com os modelos apóiam.

palavras-chave: Repositório de componentes, evolução de componentes, COSMOS, reutilização de software, modelos de implementação, regras.

Abstract

In the Component-Based Development (CBD), the link between software component developers and consumers are component repositories where software components are checked-in by their developers and checked-out by their systems integrators. Considering an evolution scenario, when components are modified, it is essential component evolution models in order to define a systematic approach to component changes. These modifications can occur, for instance, in the component specification and implementation, and component repositories should support these kind of evolutions. However, component evolution models themselves can be modified to follow technological development and different user's needs. Furthermore, different component producers may also have distinct models.

This work presents an infrastructure to support evolution in component repositories, to support various CBD services, such as component conversion to other implementation models, extraction of metainformation from the source code of components, component evolution and check implementation models. Particularly, in this work, the component repository chosen was Rigel (developed at IC-UNICAMP), which was extended to support component evolution. The Rigel repository provides necessary conditions to adopt a component evolution model, by using an extensible component metadata pattern called RAS specification.

Software tools help to increase developer's productivity and to avoid repetitive tasks. Furthermore, human modification activities are error prone. Therefore, we have developed four tools to support the SACE evolution model and the COSMOS component implementation model. These tools were developed using a rule-based framework which is based on an inference engine called Drools. This component framework is important to externalize the application rules, to guarantee a greater modifiability, which is essential for the evolution of the tools in conformity with the models they support.

key words: Component repository, component evolution, COSMOS, software reuse, implementation models, rules.

Agradecimentos

Primeiramente agradeço a Deus porque sem ele nada disso seria possível.

Agradeço à minha família (meu pai Agostinho, minha mãe Inês e minhas irmãs Raquel e Paula) pelo amor incondicional durante toda minha vida e pela compreensão nos momentos de dificuldade. Amo vocês!

Agradeço minha orientadora Cecília M.F. Rubira por pacientemente me ensinar a pesquisar e escrever textos científicos e por compartilhar parte de sua experiência e conhecimentos.

Agradeço aos meus amigos na pós-graduação do IC: Bittencourt, Neumar, Carlos, Cláudio, Sheila, Ivan, Paulo Astério, Helder, Janaína, Fernando, Ana Elisa e, especialmente, Leonel, Patrick, Tiago e Tomita que muito me ajudaram com minhas dúvidas, participaram de calorosas discussões, consolaram meus desabafos e me acompanharam em descontraídos bate-papos rumo à sala de café.

Agradeço aos funcionários do IC que sempre me atenderam pronta e amigavelmente.

Agradeço ao IC e a Unicamp que me acolheram durante todos esses anos e me proporcionaram não somente aprender novas coisas, mas também fazer novos amigos.

"*Pedras no caminho? Guardo todas... um dia vou construir um castelo!*" - **Fernando Pessoa** em **Palco da Vida**

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Neste capítulo são apresentados o contexto, a motivação, o problema e uma breve solução deste trabalho. Também são mostrados os trabalhos relacionados e, por fim, como este documento está organizado.

1.1 Contexto

Uma das principais metas do Desenvolvimento Baseado em Componentes (DBC) é reduzir o custo e tempo de desenvolvimento de software [59]. O **DBC** é uma técnica de desenvolvimento de software que se baseia na construção rápida de sistemas a partir de componentes pré-fabricados [19]. A reutilização de componentes pré-fabricados amortiza os gastos com seu desenvolvimento e, potencialmente, aumenta qualidade dos sistemas através do uso de componentes previamente testados ou certificados. Um **componente de software** é uma unidade de composição com interfaces e dependências de contexto explicitamente especificadas, que pode ser fornecido isoladamente para integrar sistemas de software desenvolvidos por terceiros. Uma **interface** identifica um ponto de interação entre um componente e o seu ambiente [39]. Um componente possui **interfaces providas**, através das quais ele declara os serviços oferecidos ao ambiente e **interfaces requeridas**, pelas quais ele declara os serviços do ambiente dos quais depende para funcionar [22]. A Figura 1.1 mostra um exemplo de um componente `ComponenteA`, que possui duas interfaces: uma interface provida chamada `IProvida` e uma interface requerida chamada `IRequerida`.

A **especificação de um componente**, também conhecida como componente abstrato, é uma abstração que define o comportamento observável externamente de um componente de software, de forma independente de qualquer implementação. Uma especificação abstrata pode ser materializada em diferentes implementações. Além disso, ela pode ser utilizada como elemento arquitetural para composição de diferentes arquite-

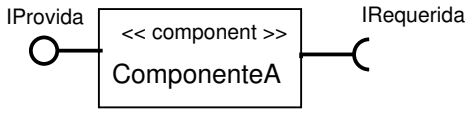

Figura 1.1: Exemplo de um componente de software.

turas de software.

Uma **implementação de componente** é um módulo executável resultante de um processo de refinamento e tradução de uma especificação de componente. Uma implementação de componente pode ser utilizada como item de configuração para compor diferentes sistemas de software executáveis. A implementação também é conhecida como componente concreto e possui duas subdivisões: componente elementar ou componente composto. Um **componente elementar** é uma implementação atômica de um componente de software e um **componente composto** é uma implementação de componente estruturada internamente como um conjunto de subcomponentes a serem providos separadamente. Uma implementação de componente representa um componente em tempo de execução, ou seja, uma instância de sua especificação. Essa instância deve se comportar como descrito em sua especificação e pode existir mais de uma instância distinta para uma mesma especificação [22].

A decomposição de sistemas em componentes menores e a separação entre implementação do componente e sua especificação permite que desenvolvedores de componentes e integradores de sistemas trabalhem de forma independente um do outro. O elo de ligação entre desenvolvedores e consumidores de componentes de software são os **repositórios de componentes** [55, 4, 5, 12], onde eles são adicionados por seus desenvolvedores e recuperados pelos integradores de sistemas. Assim, umas das responsabilidades de um repositório de componentes é oferecer serviços confiáveis e eficientes de adição e recuperação de componentes.

Um componente de software implementa uma funcionalidade, entretanto a maneira como os componentes interagem entre si é objeto de estudo da arquitetura de software. Sistemas de software complexos são decompostos em sub-sistemas menores com o objetivo de lidar com a complexidade e o tamanho de sistemas de software. Assim, a **arquitetura de software** de um sistema define uma estrutura de alto nível do mesmo em termos de elementos arquiteturais, abstraindo detalhes de sua implementação [23, 39]. Os **elementos arquiteturais** representam uma parte do sistema que é responsável por um determinado comportamento ou propriedade desse sistema, provendo um conjunto de serviços relacionados. Esses elementos podem ser componentes e conectores arquiteturais. Um **componente arquitetural** é responsável pela funcionalidade de um sistema de software. Componentes arquiteturais podem ser definidos com diferentes granularidades,

desde um componente que provê apenas uma funcionalidade básica até um subsistema complexo responsável por um amplo conjunto de funcionalidades. Um **conector arquitetural** tem como principal responsabilidade mediar a interação entre outros elementos arquiteturais. Um conector pode ser responsável também por uma parcela dos aspectos de qualidade do sistema, tais como distribuição e segurança. Os elementos arquiteturais possuem **pontos de conexão** que definem formas de interação entre um elemento e o seu ambiente [39]. Esses pontos de conexão podem ser chamados de **interfaces** ou **portas**. Uma determinada organização de componentes e conectores arquiteturais interligados entre si em um sistema é denominada **configuração arquitetural**. A Figura 1.2 mostra dois componentes arquiteturais, CompA e CompB ligados por um conector ConnAB. Essa organização constitui uma configuração arquitetural.

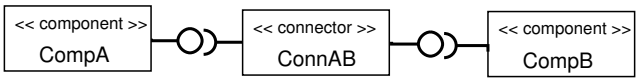

Figura 1.2: Exemplo de uma configuração arquitetural

O **modelo de implementação de componentes COSMOS** [43] une os conceitos de DBC e arquitetura de software, pois tem como objetivo garantir a conformidade entre uma arquitetura de software e o código-fonte da sua implementação, através da materialização dos elementos arquiteturais (componentes e conectores) em módulos independentes. A principal preocupação desse modelo está no projeto dos componentes, conectores e na interação entre eles. Com essa finalidade, foram criados três sub-modelos: (i) um modelo de especificação que define a visão externa do componente, (ii) um modelo que define como deve ser a implementação do componente e (iii) um modelo de conectores que define com ocorre as interações entre os componentes através dos conectores. Essa separação explícita entre especificação e implementação, permite que o componente tenha uma parte acessível externamente (a especificação) e uma parte inacessível de fora do componente (a implementação). Essa separação assegura que as funcionalidades de um componente possam ser acessadas unicamente através de suas interfaces públicas, uma vez que os elementos que compõem a implementação do componente têm visibilidade interna ao componente. Além disso, a técnica de estruturação dos componentes usada no modelo de implementação COSMOS na qual os elementos arquiteturais são materializados para código-fonte, pode ser adaptada para diferentes plataformas de componentes, por exemplo, Java EE [9] e .NET [13].

A arquitetura de software fornece uma abstração do sistema que auxilia na reutilização de componentes. A construção da arquitetura de componentes é uma das atividades apoi-

adas por ambientes integrados de desenvolvimento de software (IDE[1]). Um ambiente integrado oferece um conjunto de ferramentas que auxiliam os produtores de componentes e integradores de sistemas nas suas atividades de desenvolvimento e manutenção de software (componentes e sistemas). O ambiente Bellatrix [63] é constituído por um conjunto de *plug-ins* do ambiente Eclipse [36]. Ele apóia as fases de especificação e implementação de componentes, incluindo a construção da arquitetura de componentes e geração automática de esqueletos de componentes que compõem essa arquitetura. Esses componentes gerados automaticamente adotam o modelo de implementação de componentes COSMOS. O ambiente Bellatrix se acopla ao repositório de componentes por intermédio de um Gerenciador de Serviços do Repositório de Componentes (GSRC), que além de fornecer acesso aos serviços básicos de adição e recuperação de componentes, provê serviços adicionais como: (i) a aplicação de regras de evolução nos componentes modificados, (ii) extração de metainformação dos componentes, (iii) conversão de componentes para o modelo de implementação COSMOS e (iv) verificação de componentes em relação ao modelo de implementação COSMOS.

A Figura 1.3 mostra um exemplo de um ambiente de DBC baseado no trabalho de Beneken [24]. Os modelos de processos orientados a componentes estruturam o desenvolvimento, definindo tarefas e resultados para essas tarefas. O desenvolvimento de uma arquitetura de componentes é apoiado pelo ambiente Bellatrix, serviços adicionais são apoiados pelo GSRC e a adição e recuperação de componentes é apoiada pelo repositório Rigel [55], desenvolvido por Pinho no IC-UNICAMP. O ambiente Bellatrix, GSRC e o repositório Rigel são as ferramentas que são usadas neste trabalho especificamente.

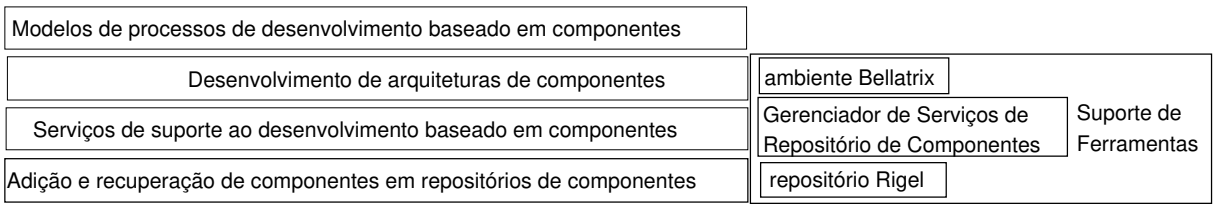

Figura 1.3: Um ambiente de apoio ao DBC

A integração do ambiente Bellatrix com o repositório de componentes Rigel ocorre por intermédio da camada do GSRC. Um exemplo é o repositório Rigel que provê serviços básicos como adição, busca e recuperação de componentes. Esses serviços podem ser acessados por meio de uma interface *web* ou através da integração entre o repositório Rigel, o ambiente Bellatrix e o GSRC. A adição de um componente faz uma cópia do componente no repositório Rigel, a busca permite ao usuário localizar o componente que

[1]do inglês *Integrated Development Environment*

deseja e a recuperação copia o componente na área de trabalho do usuário. O GSRC também pode apoiar outros serviços, como por exemplo a manipulação de modelos de implementação, suporte à evolução, certificação e segurança, que podem não fazer parte do escopo do repositório de componente e nem da IDE [33].

Para ser reutilizado, um componente deve: (i) estar disponível, (ii) ser possível de encontrar e (iii) ser possível de entender [37]. O ***Reusable Asset Specification*** (RAS) [54] padroniza a metainformação facilitando o entendimento e busca pelo componente. Ele é usado para descrever bens[2], que são soluções de software para um problema em um dado contexto. Por exemplo, no contexto de DBC, componentes abstratos, componentes concretos, interfaces e configurações arquiteturais são bens. Repositórios de componentes do mercado como Flashline [5], DAM [4] e LogicLibrary [12], vendedores de componentes como o ComponentSource [3] e repositórios acadêmicos de componentes como o Rigel [55], utilizam a especificação RAS para descrever as metainformações sobre os bens nele adicionados. A especificação RAS pode ser dividida em *Core RAS* e *Profiles*. O *Core RAS* descreve os elementos básicos de uma especificação, como o nome do bem, sua utilidade, sua classificação entre outras características. Os *Profiles* descrevem as extensões dessas características.

1.2 Motivação e Problema

É consenso na literatura que um software com tempo de vida longo deve evoluir para implementar novas funcionalidades, corrigir falhas ou adaptar-se a novos ambientes [27, 29, 25, 47]. Assim, componentes de software com um tempo de vida longo sofrerão modificações para serem reutilizados. Por isso, além de modelos de implementação de componentes como o COSMOS, são imprescindíveis **modelos de evolução de componentes** para definir uma abordagem sistemática para a evolução de componentes de software reutilizáveis. Por exemplo, um modelo de evolução de componentes proposto por Lobo *et al.* [49], chamado de *Systematic Approach for Component Evolution* (SACE), utiliza regras de evolução de componentes e um modelo de versionamento para aumentar a substitubilidade de componentes de software que evoluíram. Por exemplo, a adição de uma interface provida a um componente é um exemplo de evolução do componente. As **regras de evolução de componentes** são baseadas em alterações que podem ser aplicadas em determinados atributos de um componente. Além da adição, também podem ser realizadas modificações e subtrações em alguns atributos dos componentes. O **modelo de versionamento** determina o novo número de versão do componente de acordo com o nível de impacto da alteração sofrida por ele. Os níveis de impacto podem ser alto,

[2]do inglês *assets*

médio e baixo. A Figura 1.4 mostra um componente CompA e o arquivo com os metadados correspondentes a ele no formato do RAS. Na figura, dentre muitas metainformações sobre o componente, apenas o número de versão dele está ressaltado, que neste caso é 1.0.1. Um exemplo de evolução de componente é a adição de uma interface provida IB ao componente `CompA`. Essa alteração resulta em um nível de impacto médio, de acordo com o modelo de evolução SACE. Portanto, o arquivo de metadados que descreve este componente possuía o número de versão 1.0.1 e agora possui número de versão 1.1.1 (mais detalhes do modelo de versionamento serão apresentados na seção 2.2).

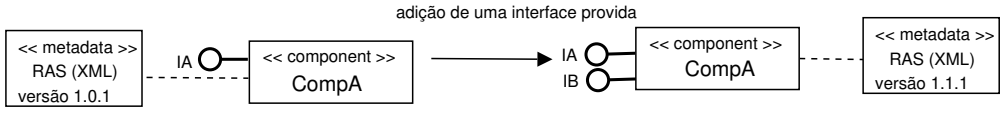

Figura 1.4: Exemplo de evolução de um componente: adição da interface provida IB

Modelos de apoio a componentes, como o modelo de implementação COSMOS e o modelo de evolução SACE, podem auxiliar produtores e consumidores de componentes na construção de um sistema baseado em componentes. Um dos objetivos destes modelos é garantir ou melhorar os atributos de qualidade de um componente. O grau de complexidade destes modelos pode divergir, sendo, em alguns casos, necessária a ajuda de uma infra-estrutura que ofereça serviços implementados por ferramentas para auxiliar os usuários desses modelos.

Além disso, os modelos de evolução e implementação adotados por uma empresa podem ser distintos aos de outra. Portanto, a infra-estrutura dos serviços deve ser flexível o suficiente para dar suporte a ferramentas que utilizam modelos distintos.

Outra característica desses modelos é que eles podem evoluir ao longo do tempo para adaptar-se às novas necessidades de seus usuários. Deste modo, as ferramentas que os apóiam devem evoluir para não se tornarem obsoletas e, por isso, a modificabilidade é um atributo de qualidade desejável para essas ferramentas.

As ferramentas que apóiam esses modelos são administradas pelo Gerenciador de Serviços do Repositório de Componentes (mostrado na Figura 1.3) e podem ser divididas em três categorias [33]:

- **ferramentas de desenvolvimento**, usadas por produtores de componentes
- **ferramentas de integração**, usadas por consumidores de componentes
- **ferramentas de gerenciamento**, usadas para administrar os serviços do repositório

1.3 Solução Proposta

Neste trabalho é proposta uma infra-estrutura de suporte à evolução em um repositório de componentes. Um dos principais objetivos deste trabalho é que seja possível apoiar a evolução de componentes adicionados ao repositório Rigel [55]. Para isso, pequenas alterações no repositório Rigel foram realizadas e além da criação novos *profiles* do RAS para apoiar a evolução de componentes de acordo com o modelo de evolução SACE. Essas alterações possibilitam que o Rigel ofereça suporte à evolução de componentes.

O apoio à evolução é realizado através de um conjunto de ferramentas que auxiliam o usuário não somente a utilizar os modelos COSMOS e SACE, mas também a converter componentes não-COSMOS em COSMOS e extrair metainformação necessária dos componente para dar suporte à evolução de componentes. Ou seja, a adequação aos dois modelos também é apoiada pelas ferramentas.

A Figura 1.5 mostra a visão geral da arquitetura do sistema, dividida em camadas, onde a camada do Gerenciador de Serviços do Repositório de Componentes (GSRC) cria uma abstração entre as camadas do Bellatrix e do Rigel, aumentando a modificabilidade do sistema. Assim, torna-se mais fácil usar o Rigel com outra IDE e o Bellatrix com outro repositório do componentes. O **GSRC** foi criado para administrar as ferramentas de uma forma que a adição e remoção de ferramentas ocorresse de maneira simples, semelhante ao mecanismo de *plug-ins* do Eclipse. Isso facilita a adaptação a novos modelos e a criação de serviços para utilizar o repositório de componentes.

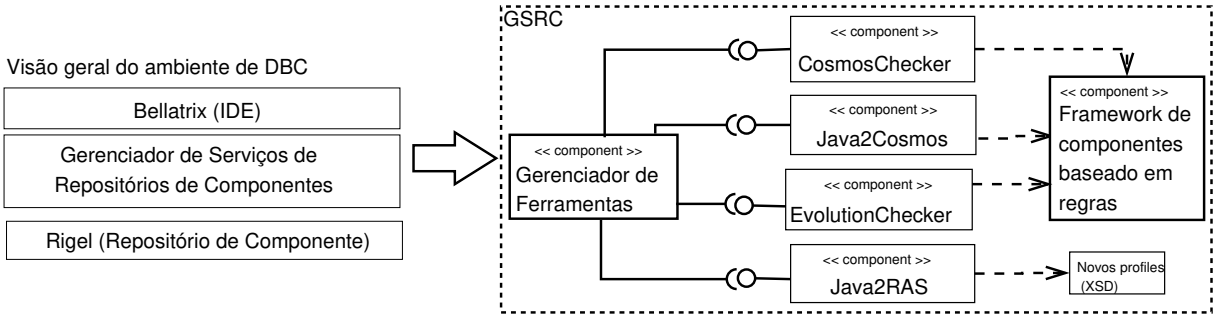

Figura 1.5: Visão geral de um ambiente de DBC. Em detalhe, uma visão interna da camada do Gerenciador de Serviços do Repositório de Componentes.

Ainda na Figura 1.5, a visão interna da camada do GSRC mostra o componente `Gerenciador de ferramentas` e quatro ferramentas: o CosmosChecker, o Java2Cosmos, o Java2RAS e o EvolutionChecker. O **Gerenciador de Ferramentas** administra as ferramentas, tratando-as como *plug-ins*, facilitando a adição e remoção de ferramentas. O **CosmosChecker** é um verificador das restrições do modelo de implementação COSMOS. Ele avalia se um componente escrito na linguagem Java satisfaz as restrições do modelo

de implementação COSMOS. O **Java2Cosmos** converte um componente escrito em Java para o modelo de implementação COSMOS. Essa conversão não altera o código-fonte do componente, apenas cria um invólucro ao redor do componente. Este invólucro representa as classes de infra-estrutura do modelo de implementação COSMOS. O **Java2RAS** extrai metainformações de componentes escritos em Java e com elas preenche o arquivo de descrição do componente de acordo com o *profile* escolhido pelo usuário. Por fim, o **EvolutionChecker** automatiza o modelo de evolução SACE, analisando o metadado de dois componentes descritos no formato RAS e calculando o nível de impacto que a substituição da nova versão do componente causará na arquitetura.

As ferramentas, com exceção do Java2RAS, foram desenvolvidas utilizando um motor de inferência chamado Drools [11], para torná-las mais modificáveis, já que o modelos podem evoluir. Desta forma, torna-se mais fácil a adaptação da ferramenta para o modelo que o desenvolvedor está utilizando, porque o modelo é implementado através de regras, que por sua vez são fáceis de alterar.

A solução desta proposta tem como principais contribuições:

1. um *framework* de componentes para desenvolvimento de ferramentas baseadas em regras usando o motor de inferência Drools.

2. construção das ferramentas de apoio ao modelo de evolução SACE através de preenchimento semi-automatizado dos metadados (Java2RAS) e aplicação das regras de evolução EvolutionChecker).

3. construção das ferramentas de apoio ao modelo de implementação COSMOS através de um conversor (Java2Cosmos) e um verificador (CosmosChecker)

4. a extensão do padrão RAS através da definição de *profiles* para dar suporte a um modelo de evolução, no caso, o SACE [49].

5. a criação de um gerenciador de serviços de repositórios de componentes modificável.

1.4 Trabalhos Relacionados

1.4.1 O repositório de bens *DigitalAssets Manager*

O repositório de bens **DigitalAssets Manager**(DAM) oferece integração com ambientes de desenvolvimento além de dispor de sofisticados mecanismos de busca e compartilhamento de bens [20]. O DAM utiliza o padrão RAS(ver seção 1.1) para descrever seus bens. Diferente de muitos repositórios acadêmicos, o DAM fornece informações interessantes não somente para desenvolvedores e integradores de componentes como também

para os gerentes de desenvolvimento. Por exemplo, uma de suas métricas de reutilização de componentes avalia quanto dinheiro foi economizado com a reutilização de componentes.

As funcionalidades de busca e compartilhamento do DAM são providas com o auxílio da **RCCS**(Rede de Compartilhamento de Componentes Software) [53]. Trata-se de uma rede *Peer-to-Peer*(P2P) baseada em padrões abertos de comunicação, arquitetura e modelagem de componentes visando facilitar o surgimento de novas implementações e participações na rede. Como a rede é P2P ela independe de uma entidade central, o que aumenta sua disponibilidade. O protocolo de comunicação baseou-se no RAS para definir os tipos de dados que serão transmitidos na rede. Dessa forma, repositórios que utilizam o padrão RAS são aptos a utilizar a RCCS. Entretanto, o DAM não oferece facilidades para desenvolver ferramentas de apoio para integradores e construtores de componentes.

1.4.2 O *framework* arquitetural ComponentForge

O **ComponentForge** é um *framework* arquitetural proposto por Oliveira *et al* [33] para apoiar o DBC distribuído. Diferente do DAM, o *framework* adota uma arquitetura orientada a serviços cujo o conjunto oferece suporte à adição, busca, certificação e negociação de componentes. O compartilhamento dos componentes é feito através de um esquema de nomes hierárquico similar ao DNS (*Domain Name Service*). Os nomes são organizados em uma árvore hierárquica onde os nós intermediários são zonas ou domínios e as folhas são os bens[3]. As zonas representam os produtores desses bens. Elas são subdivididas em domínios que facilitam o gerenciamento dos bens.

O ComponentForge utiliza uma especificação para descrever componentes baseada em XML chamada **X-ARM** (*XML-Asset Representation Model*) [34]. O X-ARM descreve características dos componentes como nome, desenvolvedor, versão entre outras. O modelo também descreve o domínio no qual o componente é usado, por exemplo, engenharia civil. Além de componentes, existem descrições de interfaces e recursos (esquemas XML, licenças, descrição de processos de desenvolvimento e outros). Entre as principais características do X-ARM estão a identificação única dos componentes através de uma abordagem similar ao DNS, com domínios controlados por instituições (empresas ou universidades) que determinam a política de acesso nesses domínios. Por exemplo, quem pode e quem não pode acessar os componentes desse domínio. Outra característica do X-ARM é o controle de visibilidade que determina o que pode e o que não pode ser indexado por máquinas de busca. O X-ARM também apóia alguns processos de DBC, classificação de bens e modelos de certificações

O controle de versões do ComponentForge permite a criação de linhas de versionamento

[3]Para Oliveira, bens são artefatos de software, como código executável e fonte

(*branches*) distintas para um mesmo bem. Entretanto, não é definido um modelo de versionamento e nem regras de evolução para os bens.

1.4.3 O ambiente MAE

Em contraste com o ComponentForge e o DAM, o ambiente **MAE**(*Managing Architectural Evolution*) [56] é utilizado para gerenciamento da evolução da arquitetura de sistemas e não de componentes. O modelo do sistema arquitetural é o ponto central do MAE. Este modelo de sistema arquitetural integra conceitos de gerência de configuração de sistemas (GCS) e de arquitetura de software. O resultado desta integração é o suporte à evolução, à variações e à opcionalidades dos elementos da arquitetura. Contudo, o MAE não possui algumas características comuns aos repositórios de bens como pesquisa e integração com ambiente de desenvolvimento.

1.4.4 O ambiente ArchEvol

O ambiente **ArchEvol** [52], ao contrário do MAE, integra um ambiente de desenvolvimento de componentes (Eclipse [36]), um ambiente para gerenciar arquiteturas (ArchStudio [61]) e um gerenciador de versões (Subversion [15]). Desta forma, o ArchEvol apóia o versionamento centrado na arquitetura. O ArchEvol provê uma série de facilidades para o versionamento da arquitetura em conformidade com a implementação. Ele estende uma linguagem de descrição arquitetural baseada em XML chamada xADL [31] para mapear a arquitetura com os componentes garantindo a conformidade entre eles. Ao contrário do MAE que prioriza o versionamento da arquitetura, o ArchEvol apóia a evolução da implementação e o mapeamento entre a implementação e a arquitetura. O ambiente não define regras para evoluir a arquitetura ou os componentes e nem um modelo de versionamento.

1.4.5 O gerador XDoclet

XDoclet [16] é um projeto de código aberto para geração automatizada de código-fonte e arquivos XML. Ele permite a programação orientada a atributo, ou seja, através de *tags* especiais inseridas no código, novas classes, interfaces ou XML descritores podem ser gerados aumentando a produtividade do programador. Para realizar essa geração automática, é necessário que o desenvolvedor crie ou altere um arquivo *build.xml* usado para compilar projetos através do Ant [1]. Depois basta compilar o código através do Ant e os arquivos serão gerados automaticamente.

1.4.6 O verificador REV-SA

REV-SA [28] é uma ferramenta para verificar a consistência entre projeto e implementação. A ferramenta é dividida em dois módulos: *XMI generator* e *XMI comparator*. O primeiro extrai informações do código compilado através de reflexão. Ele extrai informação de todas as classes relacionadas a uma determinada classe de entrada e cria um diagrama de classes em XMI. O *XMI comparator* compara o XMI gerado pelo *XMI generator* com o XMI gerado por alguma ferramenta CASE para representar o diagrama de classes. O resultado da comparação mostra se o desenvolvedor seguiu ou não o projeto e, caso não, quais as diferenças entre o que foi projetado e o que foi implementado.

1.4.7 Java2XML: tradutor de código-fonte Java para XML

Badros [21] propõem uma linguagem de marcação para Java em XML, chamada **JavaML**. Assim, a JavaML seria uma forma de estruturar informações sobre código-fonte em Java, como métodos e atributos, em XML. Isso facilita a busca de informações dentro de um código-fonte Java sem precisar fazer *parse* de texto ou utilizar um árvore de sintaxe abstrata. Para isso, Badros definiu um esquema de XML (XSD) para seu modelo e desenvolveu uma ferramenta que extrai informações de código em Java e as armazenava em XML.

Uma ferramenta chamada Java2XML foi desenvolvida para automatizar a extração de metainformação do código-fonte e armazená-la em arquivo XML.

1.4.8 Resumo dos trabalhos relacionados

A Figura 1.6 mostra de forma sintética qual é o relacionamento entre os trabalhos relacionados e as contribuições deste trabalho, listadas na seção 1.3. Os novos *profiles* adicionados ao Rigel (contribuição 1) estão relacionados com o DAM, ComponentForge, MAE e ArchEvol. O ComponentForge também está relacionado ao Gerenciador de Serviços (contribuição 2). Os ambientes MAE e ArchEvol, assim como o EvolutionChecker (contribuição 5), apóiam à evolução, contudo, de forma diferente. As ferramentas Xdoclet e REV-SA estão relacionadas ao Java2Cosmos e CosmosChecker, respectivamente (contribuição 4). Java2 XML assim como o Java2RAS, extrai metainformação de código-fonte e armazena de forma estruturada em XML (contribuição 5).

1.5 Organização deste documento

Os capítulos estão organizados da seguinte forma:

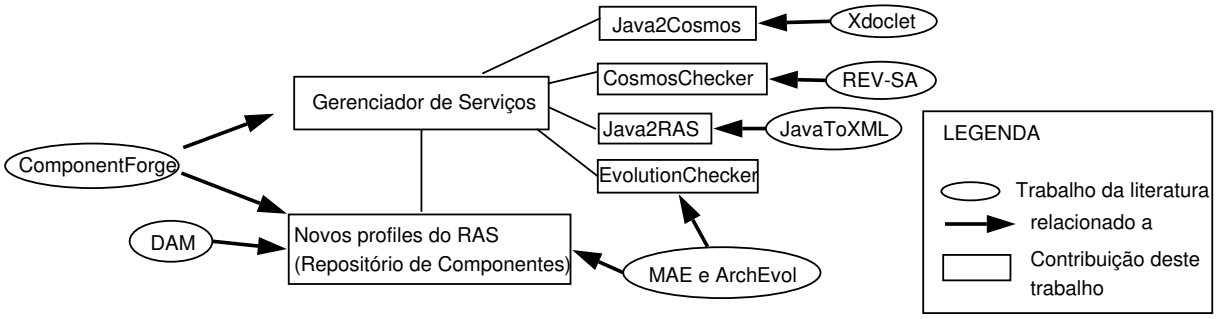

Figura 1.6: Trabalhos relacionados às contribuições da solução proposta neste trabalho

- **Capítulo 2 - Metadados para apoiar a evolução de componentes e a ferramenta Java2RAS** - Neste capítulo a extensão do *profile* do RAS para oferecer suporte à evolução é descrita em detalhes. Esses novos *profiles* são essenciais para fornecer as características do componente avaliadas pelo modelo de evolução SACE. Também é apresentada a ferramenta Java2RAS, para extrair metadados de código-fonte e armazenar no formato padrão do RAS. Alguns conceitos necessários para o entendimento do capítulo como o padrão RAS, o modelo de evolução SACE e o repositório Rigel são apresentados no começo do capítulo.

- **Capítulo 3 - Ferramentas baseadas em regras: EvolutionChecker, CosmosChecker e Java2Cosmos** - Neste capítulo são apresentados o *framework* para ferramentas baseadas em regras e as ferramentas que utilizam esse *framework*. Também são descritos o motor de inferência Drools e modelo de implementação COSMOS, para melhor entendimento das ferramentas que os utilizam. Por fim, é mostrado o Gerenciador de Serviços do Repositório de Componentes (GSRC) que administra as ferramentas.

- **Capítulo 4 - Estudos de Caso** - Este capítulo descreve dois estudos de caso e duas avaliações com as ferramentas. O primeiro estudo de caso mostra um cenário hipotético mas plausível que foi criado de forma que todas as ferramentas se interligassem e assim, pudessem ser avaliadas. No segundo estudo de caso, um sistema bancário real é usado para avaliar o CosmosChecker. Duas avaliações práticas também são descritas: uma usa componentes reais para estimar métricas de uso do Java2Cosmos e outra discute a compatibilidade entre o CosmosChecker e o Bellatrix. Por fim, é feita uma avaliação das ferramentas.

- **Capítulo 5 - Conclusões e Trabalhos Futuros** - Neste capítulo são retomados os problemas e motivações deste trabalho e apresentadas as contribuições. Também são apresentados alguns trabalhos futuros e limitações deste trabalho.

Capítulo 2

Metadados para apoiar a evolução de componentes e a ferramenta Java2RAS

Este capítulo apresenta o padrão RAS, para descrição de bens reutilizáveis. Os novos *profiles* foram criados estendendo o RAS. Esses *profiles*, que descrevem os atributos dos componentes usados pelo modelo de evolução SACE, são detalhados neste capítulo. O repositório Rigel [55] também é brevemente apresentado. Em seguida, são descritas as características que adicionadas ao Rigel permitiram que oferecesse suporte aos novos *profiles*. Por fim, é apresentada uma ferramenta para extrair metainformações de componentes chamada Java2RAS.

2.1 *RAS: Reusable Asset Specification*

O padrão RAS (*Reusable Asset Specification*), citado na seção 1.1, é usado para descrever bens reutilizáveis. O RAS utiliza a tecnologia XML para armazenar seus dados, o que facilita sua extensão uma vez que um arquivo XML é facilmente extensível.

No contexto deste trabalho, lidamos freqüentemente com bens, componentes e artefatos. Artefato, segundo o RAS, é qualquer produto criado no ciclo de desenvolvimento de um software, por exemplo, código-fonte, casos de uso, diagrama de classes entre outros. A Figura 2.1 mostra a relação entre componentes, bem e artefato. Um Componente, seja ele abstrato ou concreto, é um Bem. Mas o contrário nem sempre é verdade. Podem existir bens que são definições de interface ou configurações arquiteturais. Esses bens são materializados em arquivos, que por sua vez representam um ou mais Artefatos. Ou seja, um bem possui um ou mais artefatos. Por exemplo, o código-fonte pode ser um de vários artefatos de um componente concreto.

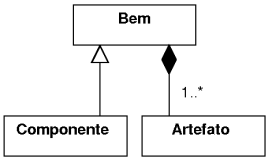

Figura 2.1: Relação entre Bem, Componente e Artefato

A especificação RAS pode ser dividida em *Core RAS* e *Profiles*. O *Core RAS* descreve os elementos básicos de uma especificação e os *profiles* descrevem as extensões desses elementos. O *Core RAS* é uma descrição abstrata que é materializada pelo `Asset`. O `Asset` é um elemento que descreve o bem e tem como atributos o nome, id, status, versão entre outros.

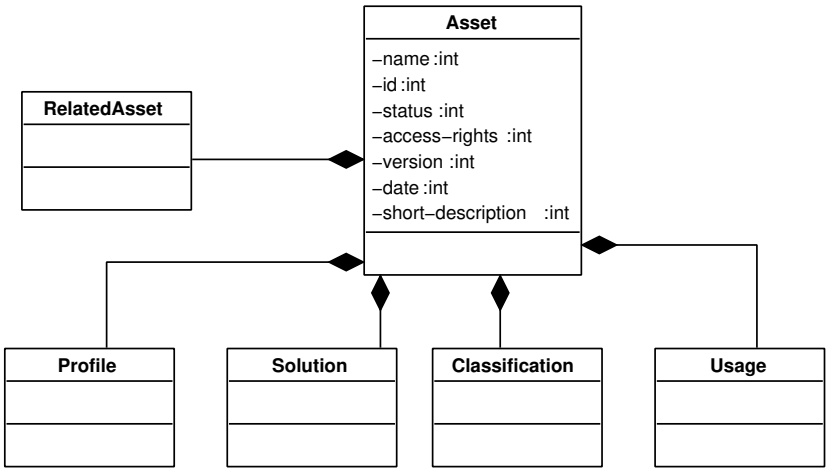

Figura 2.2: Parte do Modelo RAS

A Figura 2.2 mostra o elemento `Asset`, seus atributos e seus sub-elementos: `RelatedAsset, profile, Solution, Classification` e `Usage`. `RelatedAsset` descreve os bens relacionados; `profile` apresenta algumas características do `profile` usado por este bem; `Classification` oferece uma descrição mas detalhada sobre o tipo do bem, por exemplo, se é um bem utilizado na área médica; `Usage` mostra em qual contexto este bem deve ser usado e; `Solution` detalha a solução. Apenas os elementos dos dois níveis mais altos

estão representados na figura 2.2 e os atributos dos sub-elementos de `Asset` também não foram representados para tornar mais fácil o entendimento da figura.

A Figura 2.3 mostra o relacionamento entre os conceitos do RAS. O *Core RAS* que é materializado pelo *Default Profile*, que por sua vez é estendido pelos *profiles*. Dois *profiles* aparecem na especificação do RAS, o *Default Component profile* e o *Default WebServices profile*. Entretanto, outros *profiles* podem ser criados se algum conceito adicional sobre um bem deve ser especificado.

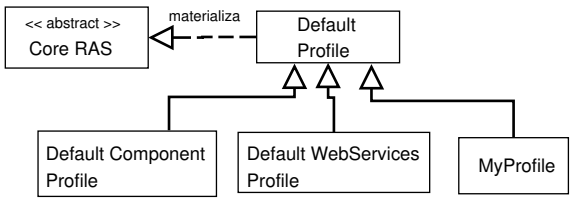

Figura 2.3: Core RAS e os *Profiles*

2.2 Modelo de evolução de componentes SACE

O **modelo de evolução** proposto por Lobo [49] (SACE) define uma abordagem sistemática para evolução de componentes de software reutilizáveis. O principal propósito deste modelo é melhorar a substitubilidade e a capacidade de evolução de componentes. Isto é realizado através da adoção de um modelo de versionamento e de regras de evolução.

As **regras de evolução** são fundamentadas em alterações sobre os componentes: adição, modificação ou subtração de determinados atributos de componentes, tais como: conjunto de interfaces, tipo de uma interface provida, plataforma alvo, código-fonte, etc. Para cada alteração existe um grau de impacto resultante que pode ser alto, médio ou baixo. Por exemplo, a modificação da plataforma alvo de um componente abstrato tem impacto alto no versionamento do componente.

Estas regras de evolução devem ser aplicadas em componentes abstratos ou componentes concretos (ver seção 1.1). O resultado é uma avaliação do impacto da mudança a ser realizada, e rejeição de modificações que são incompatíveis com o modelo evolução de componentes.

A Tabela 2.1 mostra parte da tabela de nível de impacto do componente abstrato apresentada por Lobo. Nela podemos ver que apenas três alterações possíveis em um componentes: modificação, adição e subtração. De acordo com cada atributo de um componente, temos o nível de impacto para cada uma das operações. Por exemplo, um dos

atributos do componente abstrato é o Contrato de Sincronização. Para uma modificação no contrato de sincronização o nível de impacto será alto. Uma adição tem um nível de impacto médio e, por fim, não é possível subtrair um contrato de sincronização.

Tabela 2.1: Nível de Impacto das Operações de Mudança

	Operações		
Atributos	**Modificação**	**Adição**	**Subtração**
Contrato de Sincronização	Alto	Médio	N/A
Contrato de Qualidade de Serviços	Médio	Médio	N/A

O **modelo de versionamento** determina como o impacto será refletido no número de versão do componente. A Figura 2.4 mostra exemplos de versionamento de acordo com o impacto das mudanças nos componentes. O número de versão dos componentes é subdividido em três elementos: alto, baixo e *update*. Cada elemento representa impactos de um determinado nível.

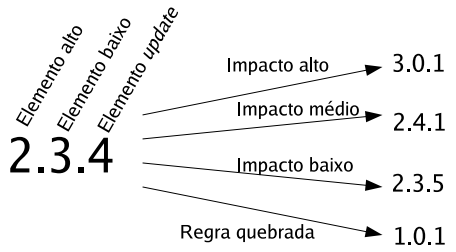

Figura 2.4: Exemplo de aplicação do modelo de versionamento. Um determinado número de versão é alterado de acordo com o nível de impacto que o componente sofreu.

Por exemplo, se um componente cujo número de versão é 2.3.4 sofrer uma alteração de impacto alto, seu número de versão passa a ser 3.0.1. Ou seja, o elemento alto é incrementado, o elemento baixo passa a ser 0 e o elemento *update* torna-se 1. Modificações de impacto médio incrementam o elemento baixo, o elemento *update* passa a ser 1 e o elemento alto permanece inalterado. Modificações de impacto baixo incrementam o elemento *update* e os demais elementos permanecem inalterados. Se uma regra for quebrada, uma nova família deve ser criada a partir da versão base de 1.0.1.

2.3 O repositório de bens Rigel com suporte à evolução de componentes

O Rigel [55] é um repositório de bens de software reutilizáveis que oferece apoio a um processo de DBC. Ele possui as características básicas de um repositório como adição, recuperação e busca de bens. O Rigel possui uma interface *web*, mas sua arquitetura permite integrá-lo com IDEs. Ele utiliza o padrão RAS para obter metainformações sobre os bens nele armazenados e para realizar um versionamento simples e manual dos bens. Contudo, o *profile* usado pelo Rigel, o *Default Component profile* (ver seção 2.1), não oferece metainformações suficientes para dar suporte ao modelo SACE. O Rigel ainda indexa os metadados do RAS para realizar a busca de bens.

2.3.1 Requisitos necessários para apoiar evolução

O Rigel utiliza o *Default Component profile* do RAS, que não possui todas as características dos componentes requeridas pelo modelo SACE. Portanto, novos *profiles* para dar suporte ao modelo SACE devem ser criados.

Além disso, esses novos *profiles* devem continuar condizentes ao modelo RAS, para que a interoperabilidade do Rigel não seja alterada.

Contudo, se o modelo SACE for alterado, os novos *profiles* podem ficar obsoletos e outros *profiles* deverão ser criados para dar suporte ao modelo. Essa troca deve exigir pouco esforço de programação.

2.3.2 Cenário de uso

A Figura 2.5 mostra um possível cenário de uso do Rigel com suporte a evolução de componentes. Um componente `CompA` e seu metadado são recuperados (*check-out*) do repositório de bens. O desenvolvedor que recuperou este componente adiciona uma interface provida `IB` ao componente. Essa mudança é refletida no metadado do novo componente, que antes possuía o número de versão 2.3.4 e agora possui número de versão 2.4.1 (ver seção 2.2 para mais detalhes sobre o modelo de versionamento). O componente evoluído pode então ser reinserido (*check-in*) no Rigel, onde ambas as versões coexistirão.

2.3.3 Visão geral da arquitetura do Rigel com suporte à evolução

A Figura 2.6 mostra a arquitetura de componentes do Rigel. O componente *Retrieve-AndEdit* 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

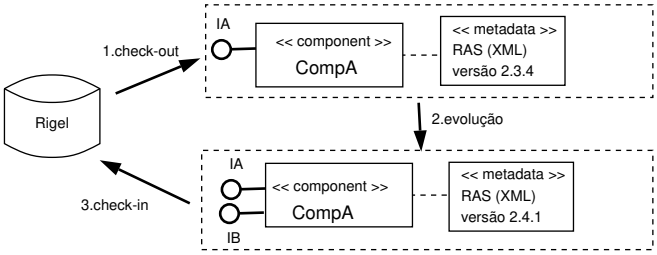

Figura 2.5: Exemplo de cenário de uso do Rigel com suporte a evolução

persistir metadados do bem e artefatos, utilizando o *FileMgr* e o *CVSMgr* para realizar a persistência. O *AssetEdit* também utiliza o componente *RASModel*. O *RepositorySearch* formata os dados e envia requisições de busca para o componente *SearchEngine*. Ao indexar os metadados, o *SearchEngine* utiliza o *RASModel* afim de extrair as informações a serem indexadas dos arquivos RAS. O componente *RASModel* está destacado dos demais componentes para indicar que este foi o único componente alterado para dar suporte aos novos *profiles* do RAS. Este componente lê os arquivos XSD que descrevem os *profiles* para fazer a conversão de arquivos no formato XSD para o modelo de objetos que representa os metadados. Esse modelo de objetos é definido pelos *profiles*.

2.4 Extensões do RAS

Como já foi dito na seção 2.3, o repositório Rigel utilizava somente o *Default Component profile*, que não oferece suporte ao modelo SACE. Para dar suporte ao modelo SACE, foram criados os *profiles* do Componente Abstrato e do Componente Concreto. Outros dois *profiles* também foram criados com o intuito de facilitar a reutilização do bem mas não apóiam nenhum modelo de evolução : Definição de Interface e Configuração. Ou seja, interfaces e configurações arquiteturais também podem ser adicionadas e recuperadas do Rigel, todavia não possuem um modelo de evolução para elas. A Figura 2.7 mostra os *profiles* criados e a relação entre *profiles* e *Core RAS*.

O *profile* de **Definição de Interface** descreve com detalhes as características de uma interface, como seus métodos, atributos, pré e pós-condições. O *profile* do **Componente Abstrato** informa sobre as características visíveis externamente, como interfaces providas e requeridas, os contratos de sincronização e qualidade de serviço e dependência de contexto. Já o *profile* do **Componente Concreto** foca nas características de implementação do componente como, por exemplo, se ele é elementar ou composto e qual componente abstrato ele implementa. Por fim, o *profile* de **Configuração** mapeia os componentes abstratos de uma arquitetura de software para componentes concretos. Os novos *profiles* e todos seus elementos e atributos são descritos mais detalhadamente no Apêndice A.

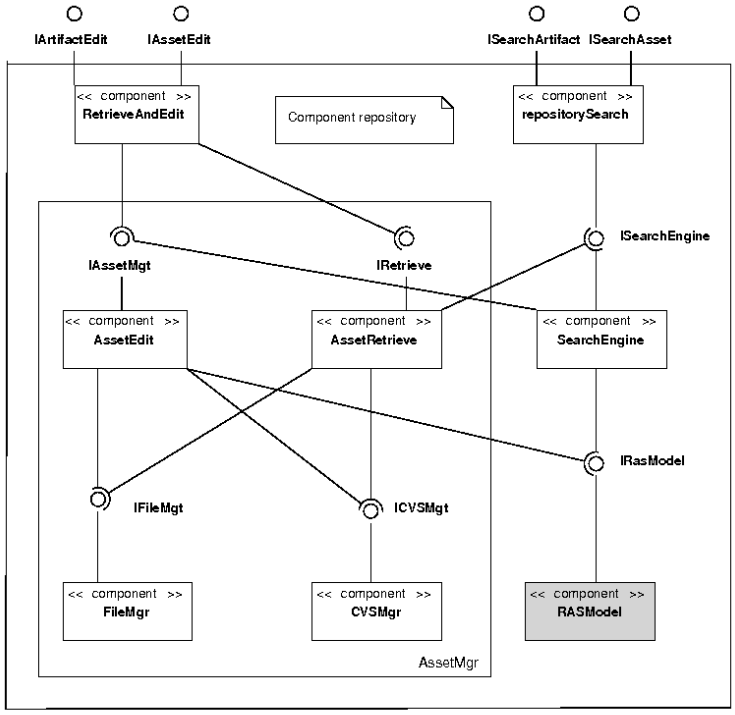

Figura 2.6: Visão Interna da Arquitetura do Repositório de Componentes

Os *profiles* do RAS são materializados em XML de acordo com um formato padrão (ver seção 2.1). Com a criação dos novos *profiles* é possível adicionar novos elementos. A listagem da Figura 2.8 mostra um exemplo de parte de um arquivo de metadados de um componente abstrato. Ou seja, o arquivo em XML está de acordo com o *profile* do Componente Abstrato em XSD. Apenas parte do arquivo é mostrada, contudo, todos os elementos deste e dos demais *profiles* são descritos em detalhes no apêndice A. Na linha 2 está o elemento `abstractcomponent`, que possui os atributos `targetplatform` cujo valor é `java` e `name` cujo valor é `Petstore`. A linha 5 mostra que este componente abstrato possui uma interface provida, normal (não é *deprecated*) cujo nome é `IManager`.

2.5 A ferramenta Java2RAS

A ferramenta que será apresentada nesta seção, o Java2RAS, tem como principal objetivo extrair metadados de um bem e criar um arquivo RAS com esses metadados. De todas as ferramentas desenvolvidas neste trabalho, esta é a única que não utiliza o *framework*

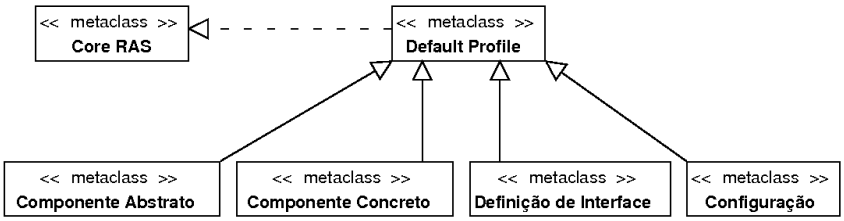

Figura 2.7: Novos *profiles* criados

```
1 ...
2 <abstractcomponent targetplatform="java" name="Petstore">
3     <externalproperty>
4         <contextdependency>
5             <interface state="normal" name="IManager" direction="provided">
6                 <relatedasset />
7             </interface>
8 ...
```

Figura 2.8: Parte de um arquivo de metadados que utiliza o *profile* do Componente Abstrato

de componentes para desenvolvimento de ferramentas baseadas em regras, apresentado na seção 3.1.

2.5.1 Requisitos

A ferramenta Java2RAS deve auxiliar o desenvolvedor a preencher as metainformações sobre bens, mais especificamente, sobre os bens representados pelos novos *profiles*, ou seja, componente abstrato, componente concreto, definição de interface e configuração. Essa extração automática de metainformação auxilia os usuários de repositórios de bens que adotam o RAS.

Cenário de uso

Um cenário de uso possível, ilustrado na Figura 2.9, ocorre quando um componente foi desenvolvido e será adicionado a um repositório de componentes que utiliza o padrão RAS, como Flashline, DAM ou LogicLibrary [5, 4, 12]. O desenvolvedor passa o componente como valor de entrada do Java2RAS e define qual é o *profile* do componente (passo 1). O Java2RAS extrai os metadados gera um arquivo de metadados de acordo com o *profile*

especificado (passo 2). Em seguida, o componente e seu arquivo de metadados podem ser adicionados no repositório (passo 3).

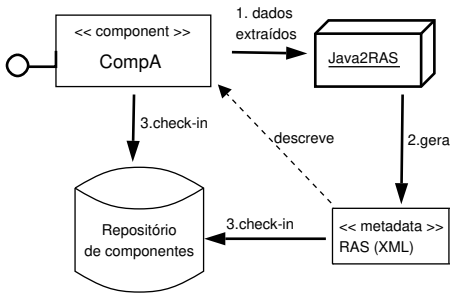

Figura 2.9: Cenário de uso do Java2RAS

2.5.2 Modelagem do Java2RAS

Para extrair metainformações dos componentes o Java2RAS utiliza uma biblioteca escrita em Java chamada Java2XML [21] (ver seção 1.4.7). Essa biblioteca extrai metainformação de código-fonte Java e guarda essas informações em um arquivo XML. O RAS também é materializado em um arquivo XML, contudo, o esquema do RAS (XSD) é diferente do utilizado pelo Java2XML. Para transformar o arquivo gerado pelo Java2XML em um compatível com o RAS, foi utilizada a *Extensible Stylesheet Language Transformations* (XSLT). A XSLT possui um tipo específico de processador de *templates* projetado para transformar documentos em XML em outros documentos em XML. Desta forma, é possível transformar o arquivo gerado pelo Java2XML em RAS. Para realizar essa transformação, foi necessário criar quatro arquivos *xslt*, um para cada *profile*. O XSLT possui algumas limitações para manipular os dados, por isso, implementar essas transformações não é trivial. A Figura 2.10 mostra o projeto do Java2RAS, que é composto por apenas um componente, o `Java2RASMgr`, e as duas bibliotecas que ele utiliza, `harsh.java2xml` (Java2XML) e `org.apache.xalan.xslt` (XSLT).

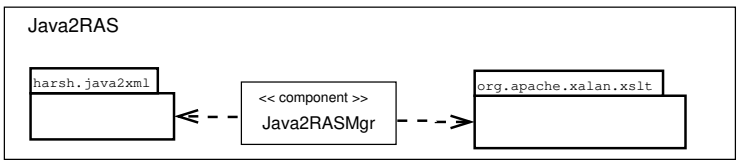

Figura 2.10: Projeto do Java2RAS

A Figura 2.11 ilustra como ocorrem as transformações. Nela vemos o metadado sobre o componente gerado pelo Java2XML que é a fonte de informação para as transformações.

Cada um dos quatro *profiles* apoiados pelo Java2RAS possui um arquivo no formato XSLT que faz o mapeamento da transformação. Esse arquivo é um dos parâmetros passados para a biblioteca que realiza a transformação. O resultado da transformação é um arquivo XML de acordo com o especificado no arquivo XSLT passado como parâmetro.

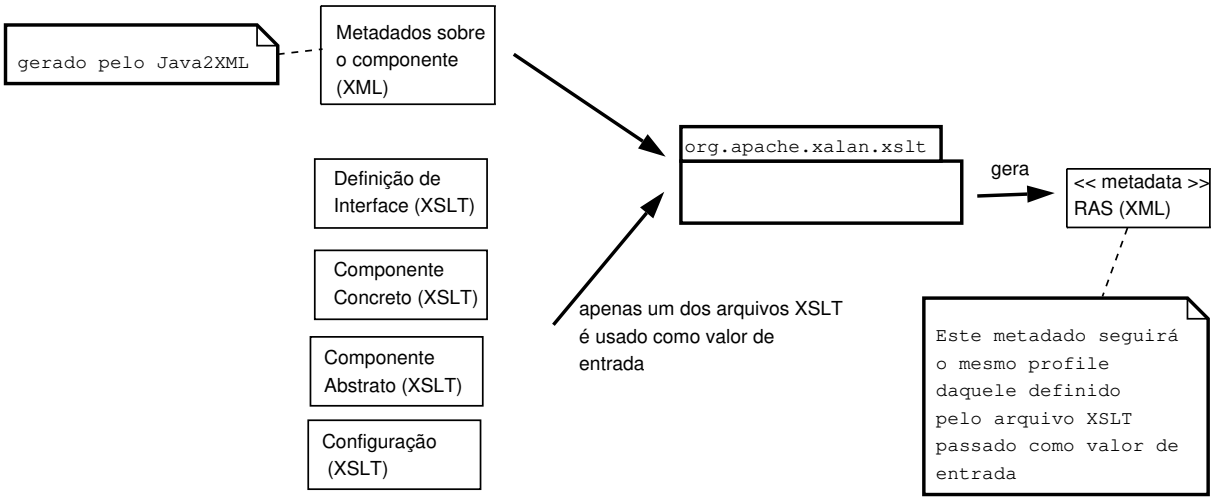

Figura 2.11: Transformações realizadas pelo Java2RAS

A transformação é possível porque existe o esquema XML (XSD) da fonte e do destino das transformações. Em outras palavras, existe um XSD para os arquivos gerados pelo Java2XML e um XSD para cada um do *profiles*.

A Tabela 2.5.2 mostra uma estimativa inferior da quantidade de campos obrigatórios preenchidos automaticamente pelo Java2RAS. Essa estimativa não é igual para todos de bens de uma mesma categoria. Por exemplo, dependendo do número de interfaces de um componente abstrato o número de campos preenchidos aumenta. Os dados apresentados na tabela são os mínimos para cada *profile*. Entretanto, se considerarmos não somente os campos obrigatórios mas todos os campos disponíveis, esses números cairão drasticamente. Isso se deve à particularidade que cada bem pode ter em um determinado contexto. Todos os *profiles* possuem campos mais subjetivos para classificar o bem, determinar seu uso, descrever quais são os bens relacionados entre outros. Esses campos são mais difíceis de serem automatizados.

Os resultados da tabela eram esperados, já que os bens com comportamento externo mais visíveis permitiram uma melhor extração dos dados. Pouco se pode afirmar sobre uma configuração analisando somente um componente. Mas muito sobre a interface podemos descobrir analisando-a.

A Figura 2.12 mostra a interface gráfica do Java2RAS, na qual vemos os campos do `template` (outro nome para o arquivo de transformação XSLT), o diretório do componente

Tabela 2.2: Estimativa da porcentagem de campos obrigatórios dos *profiles* preenchidos automaticamente pelo Java2RAS

profile	**Porcentagem de campos obrigatórios preenchidos**
Definição de Interface	79
Componente Abstrato	70
Componente Concreto	45
Configuração	44

(`component folder`) e o nome do novo arquivo que possuirá as metainformações sobre o bem (`destination`).

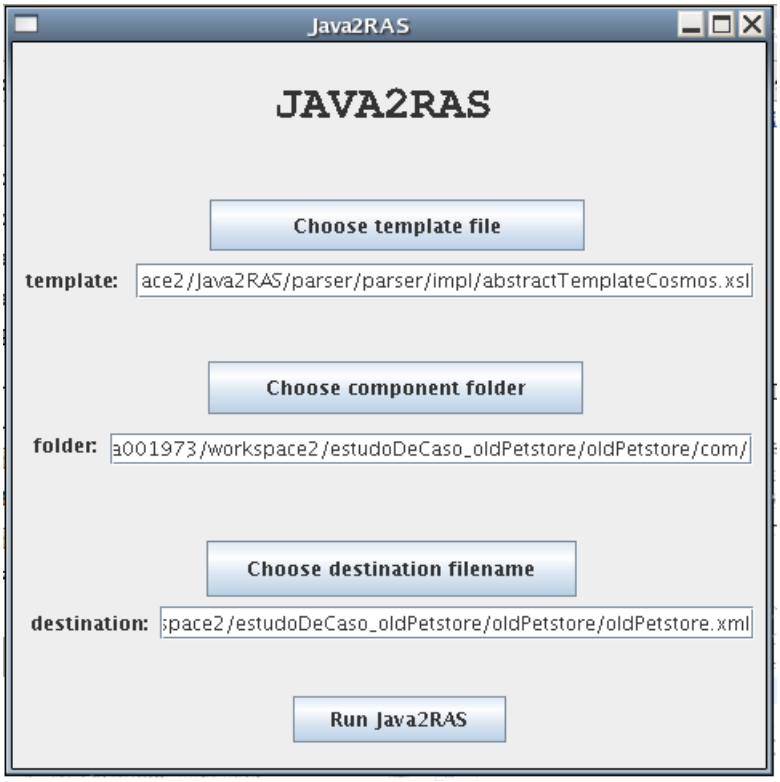

Figura 2.12: Interface gráfica do Java2RAS

2.5.3 Limitações

Os *profiles* criados são bastante extensos e muitas características sobre os bens puderam ser preenchidas. Entretanto, apenas as metainformações relacionadas ao código-fonte foram preenchidas, uma vez que só ele é analisado para preencher os metadados. Características mais subjetivas e outras que não são diretamente descritas no código-fonte

precisam ser preenchidas manualmente. Por exemplo, é possível extrair o nome de uma interface mas não sua pré e pós-condições. Por isso, apenas parte do trabalho do desenvolvedor foi automatizado.

Outra limitação é que somente código-fonte de componentes escritos em Java são aceitos.

2.6 Resumo

No começo deste capítulo foi detalhado o padrão RAS, o modelo SACE e o repositório Rigel, que não são contribuições deste trabalho, mas são importantes entendimento delas. O padrão RAS da OMG descreve bens reutilizáveis, possibilitando que eles sejam achados e entendidos mais facilmente. O RAS é extensível e permite que novas extensões, os *profiles*, sejam criadas a critério do desenvolvedor.

O modelo de evolução de componentes (modelo de evolução SACE) é apoiado pelas ferramentas EvolutionChecker (seção 3.2) e Java2RAS. O modelo de evolução SACE define regras de evolução para determinados atributos de componentes e um modelo de versionamento que determina o impacto que uma alteração no componente terá na arquitetura.

Também foi apresentado o repositório Rigel, no qual este trabalho se baseou para criar a infra-estrutura de suporte à evolução.

As principais novas contribuições apresentadas neste capítulo foram os novos *profiles* do RAS e a ferramenta Java2RAS. Os novos *profiles* são: Componente Abstrato, Componente Concreto, Definição de Interface e Configuração. Contudo, somente os *profiles* Componente Abstrato e Componente Concreto auxiliam o modelo de evolução SACE, pois este somente oferece suporte à evolução de componentes e não de interfaces ou configurações. Eles são descritos em mais detalhes no apêndice A.

Também é apresentado o Java2RAS, cuja função é extrair metainformações de código-fonte dos componentes para preencher os arquivos de metadados no formato do RAS. Somente código-fonte de componente escrito em Java é aceito. Essa ferramenta é complementar ao EvolutionChecker, descrito na seção 3.2.

Capítulo 3

Ferramentas baseadas em regras: EvolutionChecker, CosmosChecker e Java2Cosmos

Neste capítulo é apresentado um *framework* de componentes para desenvolvimento de ferramentas baseadas em regras, que utiliza um motor de inferência chamado Drools (visto na seção 3.1.1) para externalizar a parte modificável de sua implementação, ou seja, as regras. Assim, o objetivo é obter um *framework* de componentes no qual os desenvolvedores criem as regras de acordo com a finalidade da aplicação.

Em seguida são apresentadas as ferramentas que utilizam esse *framework*: EvolutionChecker, CosmosChecker e Java2Cosmos. A seção 3.3.1 detalha o modelo de implementação de componentes COSMOS [43], usado pelo CosmosChecker e pelo Java2Cosmos.

Por fim, é apresentado o Gerenciador de Serviços de Repositório de Componentes (GSRC), que administra as ferramentas EvolutionChecker, CosmosChecker, Java2Cosmos e Java2RAS.

3.1 *Framework* de componentes para desenvolvimento de ferramentas baseadas em regras

3.1.1 Motor de inferência Drools

Um **motor de inferência** é um software projetado para executar regras de forma otimizada. Ele simula a capacidade humana de chegar a uma decisão através de um raciocínio lógico. Um motor de inferência possui uma série de regras que são comparadas com fatos e cada uma delas pode ou não ter uma condição para ser executada. Por exemplo, o motor

de inferência compara se uma interface provida não foi removida de uma especificação de componente. Neste caso, o fato é a especificação de componente e a regra é verificar se uma interface não foi removida. A condição para execução de uma regra poderia ser, por exemplo, que a especificação do componente fosse válida.

Outra característica de um motor de inferência é a separação entre as partes do software com maior probabilidade de mudança daquelas que tendem a permanecer imutáveis. Essa característica, segundo Arsanjani [18], é um dos meios de obtermos sistemas mais modificáveis. Além disso, o arquitetura de um motor de inferência assemelha-se com o estilo *Máquina Virtual* que possui como atributos de qualidade a flexibilidade e a portabilidade [23].

O **motor de inferência Drools** é um projeto de código-fonte aberto que está associado ao JBoss [11]. O Drools é baseado em linguagens declarativas, suas regras são descritas em blocos *if/then* e pode não haver dependências entre elas. É possível também estabelecer uma seqüência na qual as regras serão executadas.

A Figura 3.1 mostra a visão geral do Drools. Nela vemos que o Drools possui dois tipos de memória: a Memória de produção e a Memória de trabalho. A Memória de produção é o nome dado ao local onde são armazenadas as regras que serão executadas. A Memória de trabalho comporta-se como um banco de dados global de símbolos que representam os fatos. Ambas são comparadas no Parelhador[1] de Padrões, que pode ou não estar ligado a alguma Agenda, que por sua vez determina a ordem de execução das regras.

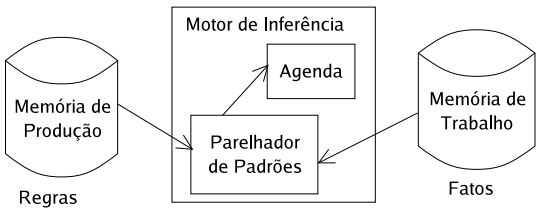

Figura 3.1: Visão geral do Drools

O motor de inferência Drools possui um **arquivo de regras** que contém as regras e condições para executá-las. O arquivo de regras é processado pelo motor de inferência no qual ocorrem as comparações entre os fatos e as condições das regras. Além da linguagem própria do Drools, cuja extensão é DRL, as regras podem ser implementadas em três linguagens de programação diferentes: Java [8], Groovy [6] e Python [14].

A linguagem usada para descrever o arquivos de regras do Drools é bastante amigável e permite que usuários com pouco conhecimento dela façam alterações de forma correta.

[1] do inglês *matcher*

```
1 #arquivo de regras
2 package com.sample;
3
4 import java.util.List;
5 import com.sample.Cheese;
6
7 global List cheeses;
8
9 rule "A Cheesey Rule"
10     when
11         cheese : Cheese( type == "stilton" )
12     then
13         cheeses.add( cheese );
14 end
```

Figura 3.2: Exemplo de uma regra do Drools

O arquivo de regras da listagem da Figura 3.2 mostra uma regra na qual somente queijos do tipo 'stilton' são adicionados à lista de queijos. Na linha 2, é declarado o pacote do arquivo de regras. Nas linhas 4 e 5, as classes usadas pela regra são importadas. A lista 'cheeses' é declarada na linha 7. A regra começa na linha 9, com a declaração de seu nome. Nas linhas 10 e 11 aparece a condição para executar a regra, que neste exemplo é que o queijo seja do tipo 'stilton'. Se a regra for satisfeita, as linhas 12 e 13 são executadas e o queijo é adicionado.

O motor de inferência Drools possui um ambiente de desenvolvimento que auxilia seus usuários na criação e edição de regras. Um *plug-in* do Eclipse [36] foi criado para dar suporte à edição de regras. Esse ambiente facilita a utilização do Drools pelo usuário pois integra a edição e execução das regras em um único lugar.

A Figura 3.3 mostra o editor do Eclipse usado para editar as regras do Drools. Notem que as palavras reservadas são destacadas para facilitar a edição das regras.

Ao contrário do *Jess* [38], outro motor de inferência para Java que também usa o algoritmo Rete [35] para processar as regras, o Drools usa uma linguagem de mais alto nível para descrever as regras. Isso facilita sua utilização por usuários que não são *experts* no assunto. O motor de inferência *Jess* possui uma linguagem para especificar regras mais parecida com linguagens típicas de inteligência artificial como Lisp e Prolog.

3.1.2 Requisitos do *framework* de componentes

Mudanças nas decisões de projeto tendem a impactar adversamente a arquitetura de software. Cientes disso, arquitetos preocupam-se em criar arquiteturas que possuem boa manutenibilidade. Segundo Arsanjani *et. al.* [18], o segredo para atingir esse resul-

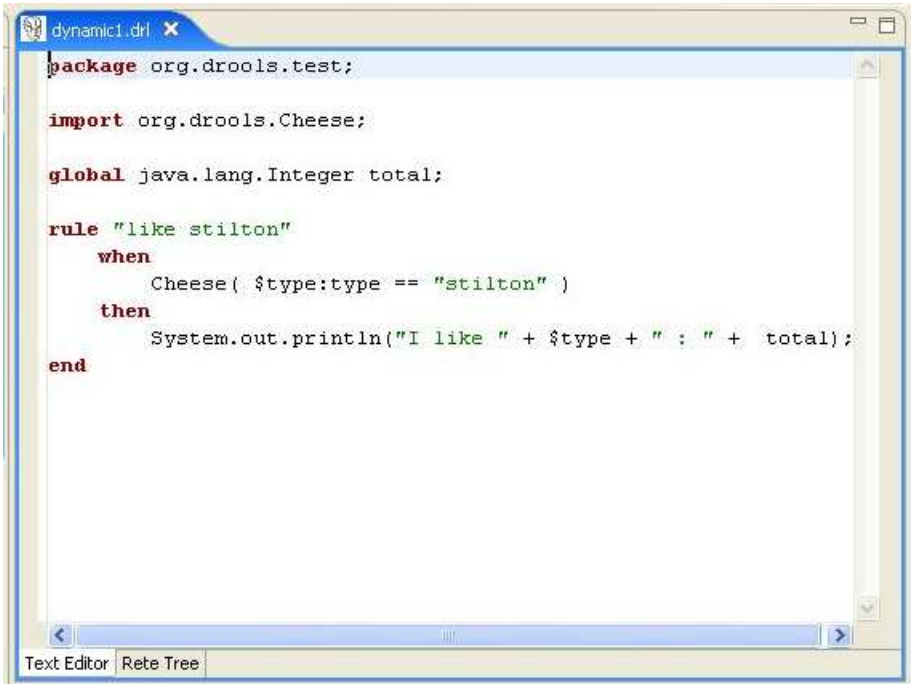

Figura 3.3: Editor de regras do Drools (fonte: [11])

tado é separação entre as parte mais estáveis do sistema das mais instáveis. Um *framework* ajusta-se bem à essa idéia. Um ***framework*** é um projeto reutilizável para um (sub)sistema de software, que é expresso através de um conjunto de classes e o modo como suas instâncias colaboram para um tipo específico de software [44].

Assim, é importante que as regras do *framework* sejam externalizadas, pois elas são a parte instável da ferramenta. As regras ainda devem ser fáceis de serem alteradas, uma vez que essa pode ser uma atividade constante.

3.1.3 Modelagem do *framework* de componentes

A Figura 3.4 mostra o projeto de um *framework* de componentes para desenvolvimento de ferramentas baseadas em regras. O componente `DroolsManager` oferece uma interface `IDroolsMgt` que permite outros componentes executar as regras por ele especificadas. O componente `Gerenciador da Ferramenta` é responsável por obter os valores de entrada e tomar decisões com os valores obtidos pelas regras. O componente `DroolsMgr` funciona como uma adaptador para utilizar a biblioteca `org.drools.ide`. É através do `DroolsMgr` que a biblioteca é configurada e os parâmetros necessários para a execução das regras são passados. Após serem aplicadas, as regras retornam o resultado obtido para a biblioteca. Qualquer tipo de alteração nas regras não afeta os demais componentes. Essa carac-

terística facilita o trabalho do desenvolvedor porque as regras podem ser desenvolvidas incrementalmente e não necessariamente todas de uma só vez. Os ponto flexíveis do *framework* de componentes são conhecidos como *hotspots* e os demais compõem o *kernel* (ou `frozenspot`) do *framework* [32].

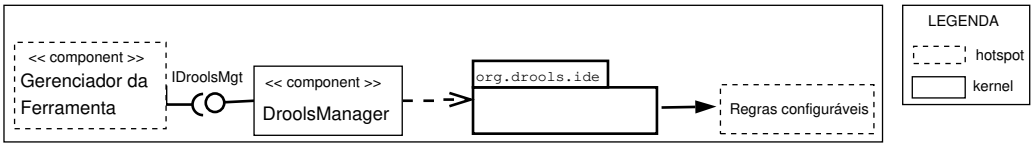

Figura 3.4: Exemplo de projeto que utiliza o *framework* de componentes. As figuras em pontilhado são os *hotspots* do *framework* e as com linhas inteiras são o *kernel* do *framework*.

Três diferentes ferramentas, com funcionalidades distintas, utilizam o mesmo *framework* de componentes pois as três são baseadas em regras e a maior diferença entre elas está nas regras que elas executam. Regras essas que podem ser editadas através de um *plug-in* para Eclipse do Drools que auxilia o trabalho do programador (a seção 3.1.1 descreve em detalhes a utilização das regras).

A Figura 3.5 apresenta a interface `IDroolsMgt` usada para executar as regras. Ela possui um único método `fireRules` para executar as regras. Os parâmetros necessários são uma `String` que aponta para o arquivo de regras que será executado e um vetor de objetos que serão os fatos relativos a esse conjunto de regras.

Figura 3.5: Interface IDroolsMgt

O diagrama de colaboração mostrado na Figura 3.6 detalha um pouco mais sobre o funcionamento do *framework* no que diz respeito a interação entre seus elementos. Para executar as regras, o `Gerenciador de Ferramentas` precisa chamar o `DroolMgr` (*1:fireRules*) que intermedia o relacionamento com a biblioteca `org.drools.ide`. Antes de executar as regras (*1.3:fireRules*), a biblioteca precisa ser configurada (*1.1:setup*) e os fatos precisam ser inseridos na memória de trabalho (*1.2:assertFact*). Após esses passos, a biblioteca executa as `regras configuráveis` (*1.3.1:fire*).

Para obter medidas de quanto o *framework* de componentes auxilia o desenvolvedor, foram analisados alguns dados referentes às implementações das ferramentas que utilizam o *framework*: EvolutionChecker (seção 3.2), CosmosChecker (seção 3.3) e Java2Cosmos (seção 3.4). O objetivo desta análise é estimar quanto de esforço é economizado ao utilizar o *framework* de componentes para desenvolver uma ferramenta baseada em regras.

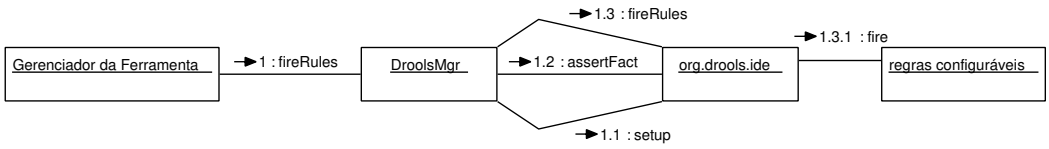

Figura 3.6: Diagrama de colaboração do *framework*

A Tabela 3.1 mostra o número de linhas de código[2] (LOC[3]) escritas para cada ferramenta e qual é a porcentagem dessas linhas que correspondem ao *framework* de componentes (*frozenspots*) e quanto corresponde a instância da ferramenta (*hotspots*). Os dados mostram que a reutilização de código oferecido pelo *framework* ficou aquém do esperado. Nas ferramentas EvolutionChecker e CosmosChecker o código do *framework* corresponde a apenas 14% do código total da ferramenta. O reaproveitamento é ainda pior na ferramenta Java2Cosmos, onde apenas 4% do código é reutilizado.

Tabela 3.1: Número total de LOC de cada ferramenta e porcentagem de LOC correspondente ao *framework* e suas instâncias

Ferramenta	**Total de LOC**	**Porcentagem de LOC do Framework**	**LOC da Instância**
EvolutionChecker	1854	14%	86%
CosmosChecker	1568	14%	86%
Java2Cosmos	7357	4%	96%

O principal motivo para os resultados ficarem aquém do esperado foi a desconsideração do código-fonte da biblioteca Drools, que é a base do *framework* e responsável pela maior parte do processamento. A Tabela 3.2 mostra os dados sobre as ferramentas incorporando o código-fonte do motor de inferência Drools. Segundo a tabela, para desenvolver o EvolutionChecker e o CosmosChecker, apenas 1% do código-fonte precisa ser implementado. Para implementar o Java2Cosmos usando o *framework* de componentes, somente 4% do código precisa ser desenvolvido. Esses resultados, por sua vez, superestimam o *framework* de componentes, dado que grande parte do código-fonte das bibliotecas não é utilizado pelo *framework*.

Portanto, é difícil obter uma boa estimativa de qual é a porcentagem do código-fonte de cada ferramenta que é economizado pela utilização do *framework*, uma vez que este baseia-se na complexa biblioteca do motor de inferência Drools. O número de linhas de código reaproveitadas do motor de inferência Drools não é ignorável e, ao mesmo tempo,

[2]Não foram contadas linhas em branco ou comentários
[3]Acrônimo do inglês para Linhas de Código

Tabela 3.2: Número total de LOC de cada ferramenta e porcentagem de LOC correspondente ao *framework* e suas instâncias.

Ferramenta	Total de LOC	Porcentagem de LOC do Framework	LOC da Instância
EvolutionChecker	179043	99%	1%
CosmosChecker	184821	99%	1%
Java2Cosmos	200786	96%	4%

não pode-se considerar o reaproveitamento de todas as funcionalidades oferecidas pelo motor de inferência Drools, já que apenas parte delas é usada.

3.1.4 Limitações do *framework*

O motor de inferência Drools usado pelo *framework* de componentes impõem que as regras sejam implementadas por uma das linguagens que o Drools oferece suporte: Java, Groovy ou Python. Outra limitação diz respeito ao desempenho do Drools. Como ele testa todas as possíveis combinações de fatos com as variáveis das condições, um determinado conjunto de regras muito grande no qual muitas são possíveis combinações terá o desempenho prejudicado.

3.2 A ferramenta EvolutionChecker

A ferramenta EvolutionChecker oferece suporte ao modelo de evolução SACE proposto por Lobo [49] (ver seção 2.2) através da utilização dos novos *profiles* do RAS, apresentados na seção 2.4. Os novos *profiles* do RAS descrevem as características dos componentes (abstratos e concretos) que são analisadas pelas regras de evolução e o resultado dessa análise determina o nível de impacto das alterações.

3.2.1 Requisitos do EvolutionChecker

Os requisitos funcionais do EvolutionChecker são oferecer suporte a um modelo de evolução e versionar componentes automaticamente. Como o modelo pode ser alterado, é interessante que a ferramenta seja modificável. Outro atributo de qualidade desejável é a usabilidade.

Cenário de uso

A Figura 3.7 mostra um possível cenário de uso do EvolutionChecker. Para facilitar o entendimento, dividimos o cenário em quatro fases. Na fase 1, suponha que um `CompA` é

recuperado (*check-out*) do repositório de componentes junto com seu arquivo de metadados no formato RAS. Dentre os vários metadados contidos no arquivo, apenas o número de versão do componente, 1.0.1, é destacado. Na fase seguinte, o `CompA`, assim como seu metadado correspondente, evolui para prover uma interface `IB`. Entretanto, o metadado do novo componente não está completo até que o impacto das alterações (no caso, uma adição de interface provida) seja analisado. Os metadados do componente original e do evoluído junto com as regras de evolução em um formato próprio do Drools (DRL) são passados como valores de entrada para o EvolutionChecker, na fase 3. Na fase 4, o EvolutionChecker analisa as alterações e calcula o novo número de versão, que completa o metadado do novo componente. Neste caso específico, a adição de uma interface provida tem impacto médio e o novo número de versão é 1.1.1 (a seção 2.2 explica com maiores detalhes o modelo de versionamento).

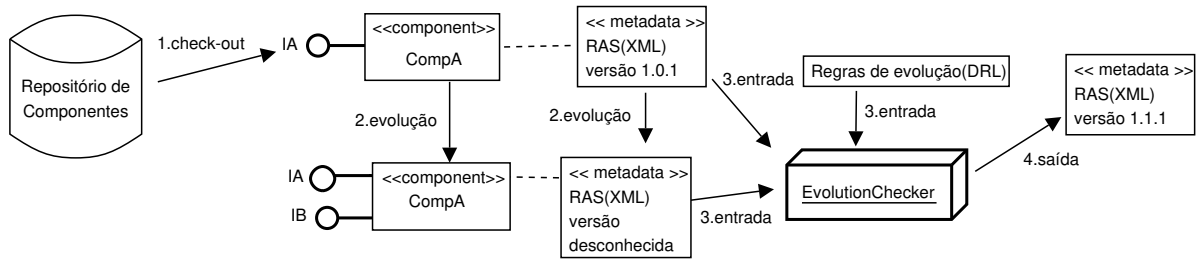

Figura 3.7: Exemplo de uso do EvolutionChecker

3.2.2 Modelagem da ferramenta EvolutionChecker

O EvolutionChecker utiliza o *framework* de componentes apresentado na seção 3.1. A Figura 3.8 é uma instanciação da figura 3.4 e mostra o projeto do EvolutionChecker. O componente `EvolutionMgr` envia para o `DroolsManager` qual é o arquivo de regras e os fatos que, neste caso, são os metadados dos componentes no formato RAS das duas versões de um componente. O `DroolsManager` prepara a biblioteca, faz uma asserção[4] dos fatos e dispara as `regras de evolução`.

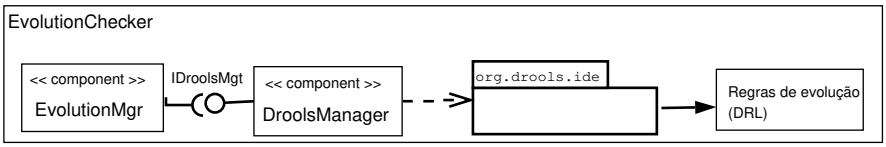

Figura 3.8: Projeto do EvolutionChecker

[4]do inglês *assertion*

Após mostrar em detalhes a parte estática dos componentes, será mostrado um exemplo de execução do componente. Por motivos de clareza, apenas uma regra, `TargetPlatform`, é disparada neste exemplo.

Foi construído um diagrama de seqüência mostrado na Figura 3.9 a partir da chamada do componente `DroolsManager` pelo `EvolutionMgr`. O diagrama começa com a classe `Facade_IDroolsMgt` que implementa a interface provida `IDroolsMgt`. A classe `DroolsMgr` (note que estamos falando da classe `DroolsMgr` e não do componente `DroolsManager`) é responsável por definir qual é o arquivo de regras (passo 1), quais são os fatos (passo 2) e quando as regras serão disparadas (passo 3). As regras de evolução do modelo SACE estão descritas no arquivo `EvolutionRules.drl` e os fatos são os metadados dos componentes, antes e depois da evolução. Após disparar as regras (passo 4) o controle da execução passa para o motor de inferência Drools. Ele tentará executar todas as regras usando combinando os fatos com as condições de execução das regras (passo 5). Quando uma condição for satisfeita, a regra é disparada (passo 6). Na figura, a regra `TargetPlatform` foi disparada. Ela processa os metadados para verificar se alguma alteração foi feita. Caso sim, a classe realiza uma busca na tabela contida no arquivo `changeImpact.xml` para descobrir o impacto causado pela mudança. Por fim, retorna o valor desse impacto (passo 7). Esse valor é armazenado utilizando a classe `RulesManager` (passo 8). Os passos 5, 6, 7 e 8 do diagrama de seqüência são repetidos proporcionalmente ao número de regras. Com o fim das regras, o controle da execução volta para o `DroolsMgr` (passo 9) e ele recupera o resultado final armazenado em `RulesManager` (passo 10 e 11), para depois repassar esse valor para `Facade_IDroolsMgt` (passo 12).

A Figura 3.10 mostra a interface gráfica do EvolutionChecker. Trata-se de uma interface simples construída com Java Swing [10]. O objetivo desta interface é apenas ajudar o desenvolvedor a executar o EvolutionChecker. Ela não auxilia na criação do arquivo de regras, uma vez que o Drools já possui um editor com essa finalidade.

Na figura também é possível ver parte da saída do EvolutionChecker na área de texto. O nome de cada regra aparece seguido do impacto que teve a aplicação dessa regra. Por exemplo, a modificação do estado de manutenção de uma interface (*maintenance state of an interface*) de componente abstrato resulta num impacto de nível alto (*high*). Na última linha da saída o EvolutionChecker calcula qual é o maior impacto resultante da aplicação de todas as regras e determina o novo número de versão do componente alterado.

A Tabela 3.3 mostra os cinco níveis de impacto e um exemplo de alteração nos componentes que podem resultar nesse impacto. Este impacto por sua vez determinará o número de versão do componente alterado.

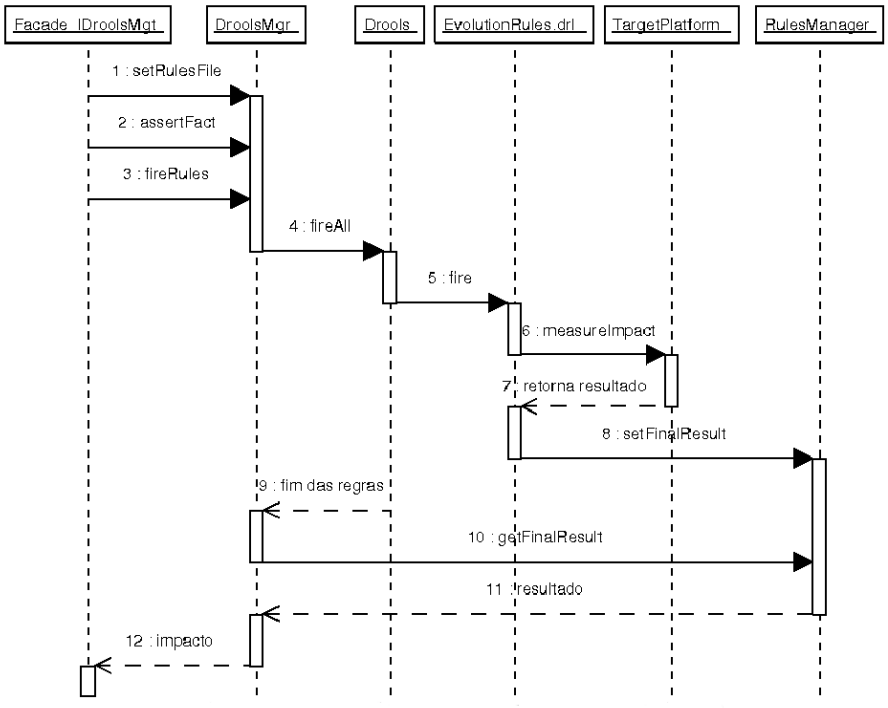

Figura 3.9: Diagrama de seqüência da ativação do EvolutionChecker

Tabela 3.3: Exemplos dos Níveis de impacto causados por alterações

Nível de impacto	**Exemplo**
Alto	Adição de interface requerida a um componente abstrato
Médio	Adição de interface provida a um componente abstrato
Baixo	Mudanças no código-fonte de um componente concreto
Insignificante	Mudanças nas pré-condições de uma interface
Regra quebrada	Subtração de um contrato de sincronização

3.2.3 Limitações

O EvolutionChecker limita a análise dos componentes aos metadados descritos no formato RAS. Nenhuma metainformação é obtida através do próprio componente. Por isso, inconsistências nos metadados geram inconsistências no resultado do EvolutionChecker.

Como o EvolutionChecker utiliza o *framework* de componentes para desenvolvimento de ferramentas baseadas em regras (ver seção 3.1), todas as vantagens e limitações do *framework* também pertencem ao EvolutionChecker.

Algumas regras como, por exemplo, a que compara os códigos-fonte de dois componentes concretos dependem da acuidade das informações fornecidas pelo desenvolvedor. Por isso, esforços foram feitos para automatizar a extração de informação dos componentes.

Figura 3.10: Interface gráfica do EvolutionChecker

Na seção 2.5 mostra uma ferramenta extrair metainformação de código-fonte escrito em Java e cria um metadado.

3.3 A ferramenta CosmosChecker

A adição de componentes ao repositório de componentes é uma das atividades mais críticas do DBC, pois um componente defeituoso pode ser reutilizado diversas vezes infectando outros sistemas. Modelos de certificação como o proposto por Álvaro *et al.* [17] procuram garantir aos usuários do repositório a qualidade dos componentes nele depositados. Modelos de estruturação ou de codificação de componentes também agregam qualidade aos componentes. Para garantir compatibilidade ao modelo de implementação COSMOS, foi desenvolvida a ferramenta CosmosChecker, que analisa o componente para determinar se ele é ou não um componente COSMOS.

3.3.1 O modelo de implementação de componentes COSMOS

O **COSMOS** é um modelo para estruturação de componentes que faz o mapeamento de arquiteturas baseadas em componentes para linguagens de programação, garantindo conformidade entre arquitetura de software e o código [43]. O modelo COSMOS também oferece diretrizes para criar componentes de software reutilizáveis e adaptáveis.

A Figura 3.11 mostra um exemplo de um componente COSMOS. Internamente o componente é dividido em duas partes: a especificação e a implementação. Na Figura 3.11 podemos ver dois pacotes que representam essa divisão. São eles o `compA.spec` e o `compA.impl`.

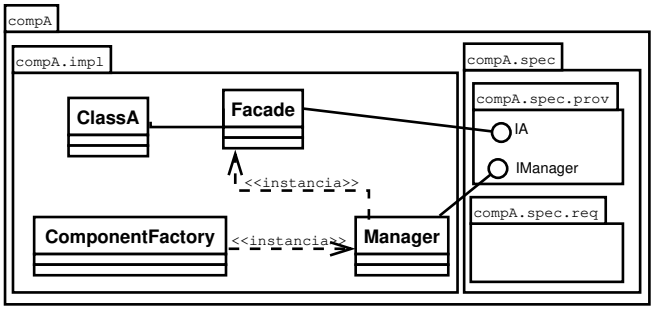

Figura 3.11: Exemplo de um componente COSMOS

A **especificação** é a visão externa do componente que também é subdividida em duas partes, uma que especifica os serviços oferecidos e outra que explicita as dependências. Na exemplo da figura, as interfaces `IA` e `IManager` representam os serviços providos pelo componente. `IManager` é uma interface que faz parte das classes de infra-estrutura do COSMOS e `IA` é um exemplo de uma interface que oferece um serviço. Como esses são os serviços oferecidos pelo componente, eles ficam no pacote `compA.spec.prov`. As dependências deste componente são representadas pelas interfaces do pacote `compA.spec.req`. Nem todo componente possui dependências externas.

A **implementação** possui uma infra-estrutura para instanciar o componente e implementar os serviços oferecidos pelas interfaces. A classe `ComponentFactory` é a única classe visível externamente pois ela é responsável por começar a instanciação do componente. Através da `ComponentFactory` a classe `Manager` é instanciada e por ela pode-se obter as interfaces do componente. As interfaces providas são implementadas por `Facades` que, com auxílio de outras classes, implementam os serviços das interfaces.

A descrição detalhada do modelo de implementação COSMOS é importante para o entendimento das ferramentas CosmosChecker e Java2Cosmos.

3.3.2 Requisitos do CosmosChecker

O CosmosChecker deve verificar se um componente satisfaz ou não as restrições do modelo COSMOS. Essa verificação pode ser feita com dois diferentes valores de entrada: o diretório onde estão as classes do componente ou um arquivo *jar*. Além disso, se um componente não satisfaz o modelo COSMOS, mostrar qual(is) restrição(ões) não foi(ram) satisfeita(s).

Assim como o modelo SACE, o COSMOS pode evoluir ao longo do tempo e, portanto, é importante que a ferramenta seja planejada para acompanhar essa evolução. Isso significa que a modificabilidade é um atributo de qualidade desejado e, por isso, o *framework* de componentes para ferramentas baseadas em regras é usado (seção 3.1). A usabilidade também é importante uma vez que ela auxiliará os desenvolvedores não somente a usarem a ferramenta, como também alterá-la conforme seus interesses.

Cenário de uso

A Figura 3.12 ilustra um cenário de uso do CosmosChecker dividido em quatro etapas. Na primeira etapa o desenvolvedor recupera (*check-out*) um componente CompA do Repositório de componentes. O CompA é então alterado para atender novos requisitos ou corrigir uma falha. Após ser devidamente testado, a nova versão do componente pode ser depositada no Repositório de componentes. Contudo, suponha que a empresa desenvolva somente componentes COSMOS. Antes do novo componente ser depositado, deve-se verificar se ele satisfaz as restrições do modelo COSMOS, o que corresponde à terceira etapa. Por fim, o componente pode ser depositado (*check-in*).

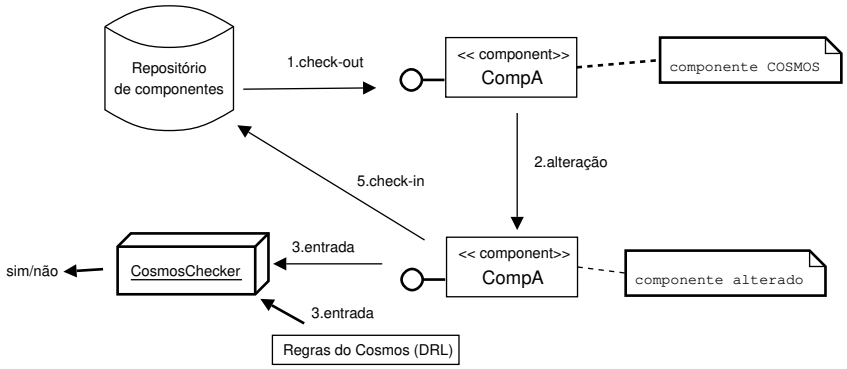

Figura 3.12: Exemplo de uso do CosmosChecker

3.3.3 Modelagem da ferramenta CosmosChecker

A Figura 3.13 mostra o projeto do CosmosChecker, que é uma instância do *framework* de componentes mostrado na Figura 3.4. O modelo COSMOS foi inicialmente proposto por M.S. Júnior [43] em 2003 mas, assim como software evolui, um modelo de implementação também deve evoluir. Ou seja, o modelo não é estático e definitivo. Portanto, é esperado que as restrições do modelo sejam facilmente alteradas caso o modelo evolua. Essa é a motivação para utilizar o *framework* de componentes para ferramentas baseadas em regras. As regras que verificam se um componente é ou não COSMOS podem ser alteradas sem muito esforço de programação. Além disso, o motor de inferência usado pelo *framework* de componentes, o Drools, possui um editor de regras amigável para o desenvolvedor.

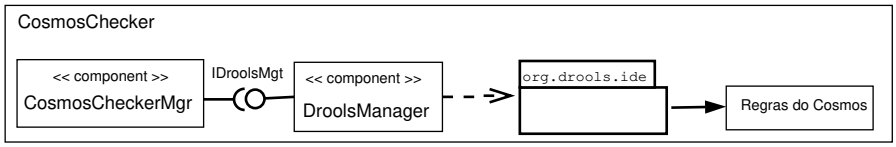

Figura 3.13: Projeto da ferramenta CosmosChecker

Uma pergunta importante norteou o desenvolvimento do CosmosChecker: quais são os requisitos mínimos para um componente poder ser considerado COSMOS? Apesar de listar diretrizes do modelo COSMOS, Silva não responde essa questão. Outro ponto problemático é que o modelo vêm sofrendo evoluções desde sua primeira publicação (vide artigo de Gayard [40]). Essas evoluções ocorrem de acordo com o interesse das empresas e grupos de pesquisa que utilizam o COSMOS, não havendo, necessariamente, um consenso sobre elas. Isso reforça a necessidade de criar uma ferramenta que seja modificável para verificar as restrições do modelo COSMOS, uma vez que essas restrições não são fixas. As regras implementadas pelo CosmosChecker seguem as diretrizes apresentadas por Silva [43].

O CosmosChecker recebe três valores de entrada: (1) o arquivo de regras do COSMOS;(2) o pacote do componente e;(3) o diretório das classes ou o arquivo *jar*. O arquivo de regras do COSMOS lista as regras que analisam o componente para saber se ele satisfaz as restrições do COSMOS. Um exemplo de regra é a divisão do componente nos subpacote *spec.prov*, *spec.req* e *impl*. Além disso, o CosmosChecker utiliza reflexão para saber se a lista de interfaces providas obtida através da chamada do método *listProvidedInterfaces()* corresponde às interfaces implementadas pelas classes.

A Figura 3.14 ilustra a interface gráfica do CosmosChecker foi construída utilizando Java Swing [10]. A interface gráfica, apesar de bastante simples, auxilia o entendimento da ferramenta.

A figura mostra um exemplo de uma execução. O usuário escolheu o arquivo de regras

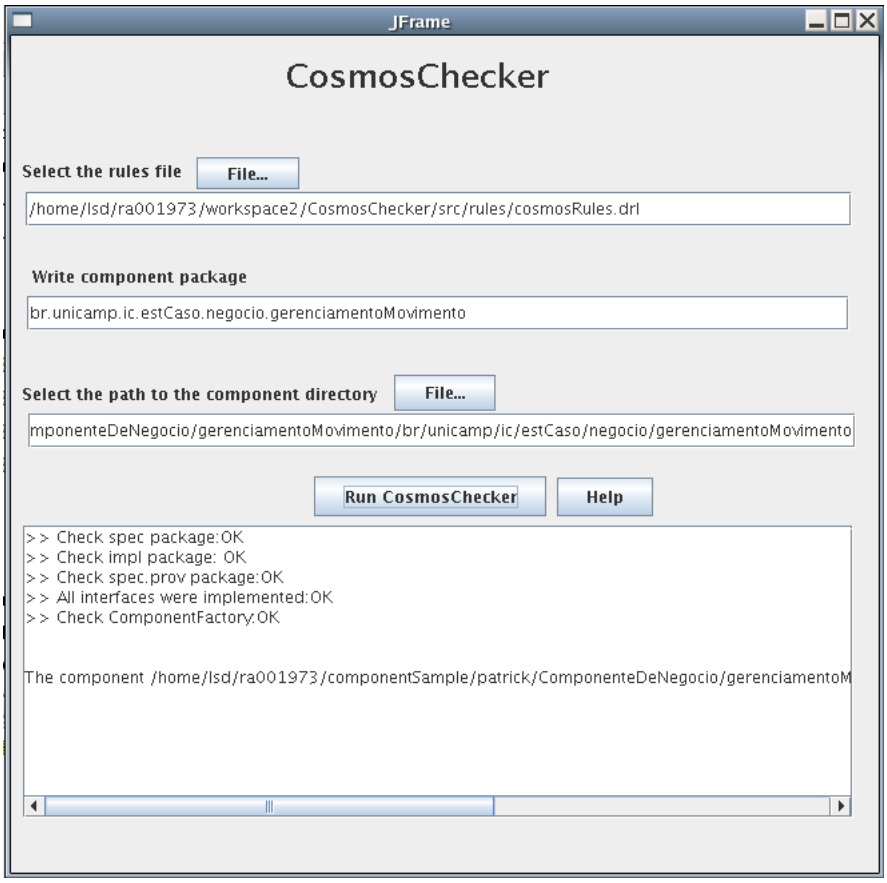

Figura 3.14: Interface gráfica do CosmosChecker

que foi executado, digitou o pacote deste componente e escolheu o diretório do componente que ele queria verificar. O resultado da execução é mostrado na área de texto. Ao lado de cada regra, aparece a confirmação (*true*) ou não (*false*) da regra. Se todas as regras tiveram resultado positivo então o componente é COSMOS. Caso contrário, o componente não é COSMOS.

3.3.4 Limitações

O uso de reflexão para verificar se um componente é COSMOS, apesar de enriquecer a verificação prejudica o desempenho da aplicação como aponta Sosnoski [57].

Outra limitação é que COSMOS pode ser implementado com outras linguagens de programação orientadas a objeto além de Java, contudo o CosmosChecker só verifica componentes Java.

3.4 A ferramenta Java2Cosmos

Modelos de estruturação e padrões de projeto são utilizados para agregar qualidade ao software. O modelo de implementação COSMOS utiliza padrões de projeto para construir componentes adaptáveis e reutilizáveis [43]. Ferramentas como BeautyJ e Jalopy [2, 42] auxiliam os desenvolvedores a transformar o código-fonte de forma automatizada. A ferramenta Java2Cosmos não altera o código-fonte do componente, dado que nem sempre este é disponível. Ela apenas cria um invólucro sobre o componente, adicionando as classes e interfaces que fazem parte da infra-estrutura do COSMOS (ver seção 3.3.1).

3.4.1 Requisitos do Java2Cosmos

O único requisito funcional do Java2Cosmos é converter componentes caixa-preta[5] ou não em componentes COSMOS. A dificuldade está em converter diferentes tipos de implementações no modelo COSMOS. Como o modelo COSMOS pode evoluir, foi utilizado o *framework* de componentes para desenvolvimento de ferramentas baseadas em regra apresentado na seção 3.1. Assim, o Java2Cosmos pode acompanhar a evolução do modelo com pouco esforço de programação. Para que esse esforço seja pequeno, também é necessário que a ferramenta possua uma boa usabilidade.

Cenário de uso

A Figura 3.15 apresenta um exemplo de cenário de uso do Java2Cosmos, com apenas duas fases. Na primeira, um `CompA` é passado como entrada para o Java2Cosmos junto com as `Regras de conversão`, que estão no formato do Drools (DRL). O resultado da segunda fase é um novo componente `CompA` que satisfaz as restrições do modelo COSMOS. O novo componente possui uma interface a mais chamada `IManager` que faz parte da infra-estrutura do COSMOS.

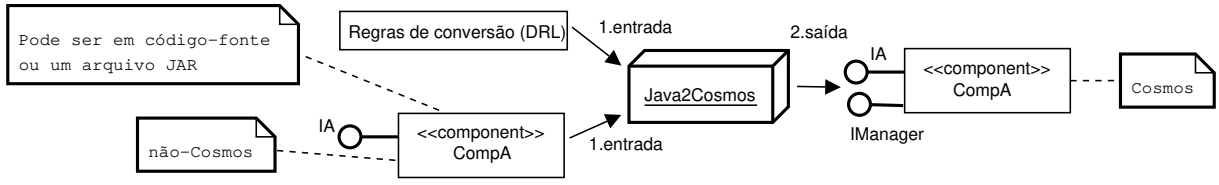

Figura 3.15: Exemplo de uso do Java2COSMOS

[5] cujas estruturas internas não são conhecidas

3.4.2 Modelagem da ferramenta Java2Cosmos

A Figura 3.16 é uma instância da figura 3.4 e mostra o projeto do Java2Cosmos. Para converter um componente para COSMOS, a ferramenta precisa saber metainformações sobre o componente a ser convertido, por exemplo: quais são as interfaces, quem implementa essas interfaces, quais são seus métodos entre outras metainformações. Isso é obtido através de duas formas, de acordo com o tipo de dado que é passado como valor de entrada para a ferramenta. Se for um arquivo *jar*, a ferramenta não tem acesso ao código-fonte do componente e, portanto, as metainformações são obtidas através de reflexão computacional. As classes e suas dependências são carregadas pelo Java2Cosmos, possibilitando extrair as metainformações necessárias para a conversão.

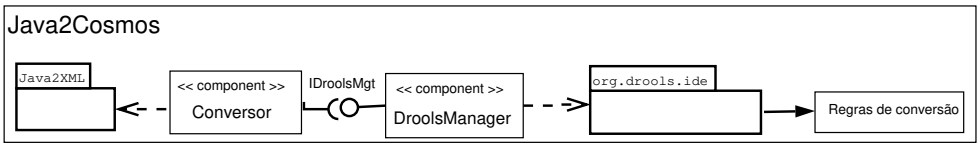

Figura 3.16: Projeto do Java2COSMOS

Todavia, se a entrada for o código-fonte do componente ao invés de um arquivo *jar*, o Java2Cosmos adota uma estratégia diferente: para extrair metainformação do código-fonte ele utiliza uma biblioteca chamada Java2XML [21] (ver seção 1.4.7). Essa biblioteca extrai metainformações sobre código-fonte escrito Java e converte para XML. Informações como nome da classe, assinatura dos métodos entre outras são estruturadas em XML, de acordo com um XSD pré-definido. O código-fonte analisado não precisa ser compilável.

Essas duas formas distintas de executar o Java2Cosmos, com arquivo *jar* ou código-fonte, são chamadas de modos de execução. Em ambos os modos, a conversão não é totalmente automatizada, pois pequenas adaptações podem eventualmente ser necessárias no componente convertido e serão mostradas com mais detalhes na seção 3.4.3.

Outra vantagem da utilização da ferramenta é que Java2Cosmos auxilia os usuários do ambiente Bellatrix. O Bellatrix gera automaticamente esqueletos de código-fonte que seguem o modelo COSMOS. Esses componentes formados por esses esqueletos são gerados a partir de uma arquitetura de componentes especificada no Bellatrix. Portanto, se um arquiteto de software que utiliza esse ambiente deseja reutilizar um componente Java que não é COSMOS, ele pode utilizar o Java2Cosmos para converter para COSMOS o componente, o que facilitaria sua integração na arquitetura.

A Figura 3.17 mostra a interface gráfica do Java2Cosmos. Os três campos superiores, regras (*rules*), destino (*destination*) e pacote (*package*) são de uso comum tanto do modo de execução com arquivo *jar* quanto com código-fonte. Existe a opção de escolha por um dos modos de execução que determina quais campos ficam habilitados e quais não.

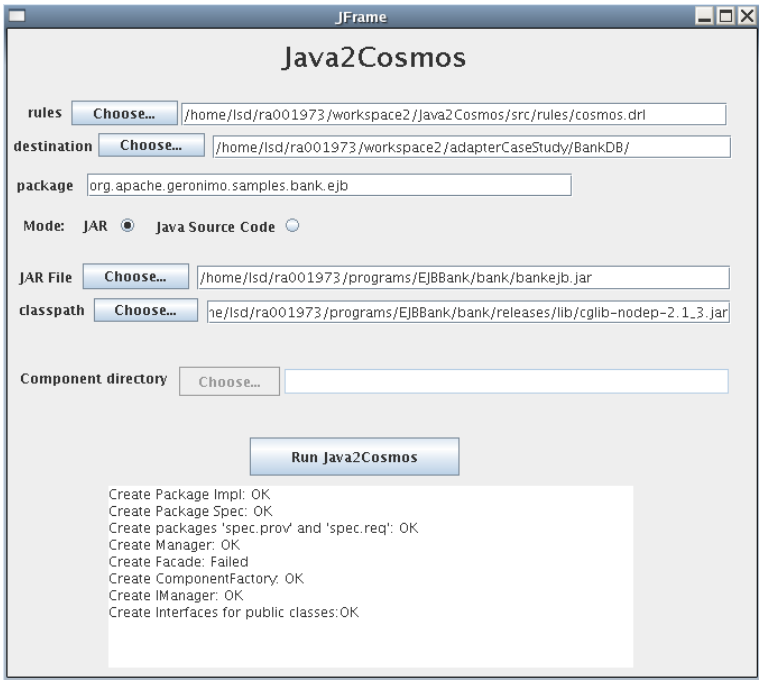

Figura 3.17: Interface gráfica do Java2COSMOS

A figura ainda mostra o modo de execução com arquivo *jar* no qual os campos arquivo *jar* (*jar file*) e *classpath* ficam habilitados e o campo diretório do componente (*component directory*) fica desabilitado. Preenchido os campos corretamente, os usuários podem executar o programa pressionando o botão *Run Java2Cosmos*. Na área de texto abaixo do botão aparece o resultado da conversão. Cada uma das regras é executada e seu resultado é mostrado ao lado: *OK* para bem sucedida e *Failed* caso contrário.

3.4.3 Limitações

Uma limitação da ferramenta gerou erros de sintaxe, mais especificamente, construtores não-visíveis pelas classes Facade que os utilizam. Isso ocorreu por dois motivos: o primeiro motivo é que as instâncias de algumas classes são obtidas através de outras classes (as fábricas) e essa indireção não é compreendida pela ferramenta. O segundo motivo é que o construtor de algumas classes possui visibilidade de pacote e pode ocorrer de o Facade não estar nesse mesmo pacote, portanto, o construtor torna-se inacessível para o Facade. O exemplo a seguir ajuda a entender o problema. A listagem abaixo mostra como obter um objeto do tipo `IManager` de um componente COSMOS.

```
1 public void m1(){
2     IManager manager = ComponentFactory.createInstance()
3     ...
```

```
4 }
```

Contudo, o método gerado automaticamente faria algo parecido com a listagem abaixo, o que é incorreto.

```
1 public void m1(){
2     IManager manager = new Manager();
3     ...
4 }
```

Outra limitação do Java2Cosmos é que classes abstratas que implementam interfaces podem induzi-lo a gerar código-fonte incorreto. O Java2Cosmos criará um Facade para essas classes e tentará instanciá-las, o que não é permitido pelo compilador de Java.

Arquivos *jar* que possuem classes não-compiláveis geram exceções uma vez que o Java2Cosmos usa reflexão para extrair metainformação do componente.

3.5 Gerenciador de Serviços de Repositório de Componentes (GSRC)

Nesta seção é apresentado um Gerenciador de Serviços do Repositório de Componentes (apresentado na figura 1.5) que permite ao usuário a fácil adição e remoção de serviços que ele julga importante e adequado para seu contexto. O GSRC propicia uma camada de abstração entre o repositório de componentes e um IDE o que aumenta a modificabilidade da arquitetura do sistema. Os serviços são implementados pelas ferramentas Evolution-Checker, Java2RAS, CosmosChecker e Java2Cosmos, que fazem o papel de *plug-ins*, para facilitar a adição e remoção de serviços de acordo com a necessidade do desenvolvedor.

3.5.1 Requisitos do GSRC

Como o GSRC intenciona administrar ferramentas, deve ser possível adicioná-las e removelas com facilidade. Também deve ser possível utilizar os serviços dessas ferramentas. Todavia, para isso é necessário que elas sejam listadas e que detalhes de seus serviços sejam especificados.

As ferramentas administradas pelo GSRC são adaptadas ao repositório Rigel. Diferentes repositórios demandam diferentes ferramentas. A adição e remoção não deve ser custosa, facilitando a adesão de novos serviços e a rejeição dos obsoletos.

Cenário de uso

Um possível cenário de uso seria a adição e utilização de um serviço. Esse serviço é implementado por uma ferramenta, cuja adição ao GSRC deve ser um processo que exige pouco esforço. Para adicionar a ferramenta, o desenvolvedor precisa criar um subdiretório onde colocará a implementação da ferramenta. Também é necessário que o desenvolvedor crie um arquivo de descrição dela, que permitirá o GSRC obter algumas metainformações sobre o nova ferramenta automaticamente.

Para utilizar o novo serviço, o desenvolvedor solicita ao GSRC uma lista dos serviços disponíveis. Essa lista é dispensável caso o desenvolvedor conheça detalhadamente o serviço que deseja usar. Desta forma, o desenvolvedor pode utilizar o novo serviço.

3.5.2 Projeto do GSRC

A Figura 1.5 mostra uma visão geral de um ambiente de DBC. O ambiente Bellatrix é uma IDE que interage com o GSRC, cujo objetivo é criar uma camada de abstração entre IDE e repositório para aumentar a modificabilidade do sistema e gerenciar a ferramentas. A figura ainda mostra em detalhes a camada do GSRC na qual podemos ver as ferramentas desenvolvidas.

Na visão interna da camada do GSRC, podemos ver como o `Gerenciador de Ferramentas` se relaciona com as ferramentas. As interfaces das ferramentas são acessadas pelo `Gerenciador de Ferramentas` que repassa os serviços para a camada superior. De certa forma, o `Gerenciador de Ferramentas` funciona como um conector. Contudo, não existe uma dependência entre o `Gerenciador de Ferramentas` e as ferramentas, o que facilita a remoção dessas caso necessário.

Por que não deixar as ferramentas soltas, ou seja, sem um gerenciador para controlá-las? Porque ao usarmos o GSRC podemos estabelecer políticas para os serviços implementados pelas ferramentas. Por exemplo, suponha que um determinado repositório de componentes só aceite componentes COSMOS. Poderíamos usar o verificador de componentes COSMOS implicitamente, só permitindo que componentes que satisfaçam as condições sejam aceitos.

Outro ponto positivo é que a camada extra criada pelo GSRC entre o repositório de componentes e as IDEs que os utilizam resulta em um menor acoplamento entre eles. Esse baixo acoplamento aumenta a modificabilidade do sistema, ou seja, permite que o usuário mude de IDE ou de repositório com maior facilidade.

Contudo, a presença desta camada extra não deve impedir a integração entre um repositório de componentes e um IDE. Esta integração pode ser implementada através de uma ferramenta do GSRC que funcionaria como conector entre as duas aplicações. Uma avaliação mais detalhada desta arquitetura foi realizada no Apêndice B.

3.5.3 Implementação do GSRC

Um dos atributos de qualidade importantes para o GSRC é a modificabilidade, dado que a adição e remoção de ferramentas deve exigir pouco esforço do desenvolvedor. O projeto possui influência direta nos atributos de qualidade e, assim, é uma parte importante do desenvolvimento para atingir a modificabilidade desejada.

Uma influência forte no projeto final do GSRC foi o Eclipse [36], que possui mecanismos para adicionar e remover *plug-ins* com facilidade. O Eclipse possui um diretório chamado `plugins`, dentro do qual são colocados os *plug-ins* que adicionam novas funcionalidades àquelas providas pelo núcleo do Eclipse. Cada *plug-in* possui um arquivo que o descreve, chamado de *manifest file*. O *manifest file* descreve o nome do *plug-in*, sua versão, desenvolvedores, quais outros *plug-ins* que ele estende entre outras informações. Além disso, o *manifest file* possui um nome padrão que é `plugin.xml`.

O projeto do GSRC possui algumas características do Eclipse, como o diretório `plugin` e o arquivo de descrição da ferramenta. Uma restrição imposta no projeto é que todas as ferramentas devem ser componentes COSMOS. O motivo desta restrição é que o modelo COSMOS proporciona uma fácil instanciação dos componentes, sem que para isso seja necessário saber muitos detalhes sobre sua estrutura interna. Assim, através de reflexão é possível listar as interfaces providas de todos os *plug-ins*. Essa lista de interfaces possibilita que o usuário utilize uma interface, sem necessariamente saber quais são as ferramentas disponíveis.

Outra característica das ferramentas é que elas devem estar no formato *jar*, dentro de um subdiretório do diretório `plugin`.

A Figura 3.18 mostra o projeto simplificado do GSRC. As classes `ComponentFactory`, `Manager` e a interface `IManager` do projeto foram omitidas para simplificar a figura. No pacote `repositoryServiceMgr.impl`, as classes `XPathHelper` e `JarClassLoader` ajudam a lidar com arquivos XML e *jar*, respectivamente. `XPathHelper` extrai informações do arquivo de descrição do *plug-in* e `JarClassLoader` é um ClassLoader próprio da aplicação para extrair informações do arquivos *jar*.

O diretório `plugin` (sem hífen) contém os subdiretórios de todas as ferramentas. Esses subdiretórios possuem a implementação da ferramenta em formato *jar*, junto com um arquivo de descrição com nome de `repositoryService.xml`. Os arquivos de descrição de cada ferramenta contém informações que são necessárias para sua instanciação.

Na Figura 3.19 temos a estrutura de diretórios do GSRC com a ferramenta `evolution-Checker` adicionado. Criado o diretório, o usuário deve criar o arquivo de descrição do serviço. Esse arquivo segue um esquema já definido pelo GSRC. Ele deve conter o nome, pacote, id, caminho do arquivo JAR , descrição do *plug-in* e o número de versão do *plug-in* e seu nome é, obrigatoriamente, `repositoryService.xml` para facilitar a automatização. A listagem mostrada na Figura 3.20 mostra um exemplo deste arquivo.

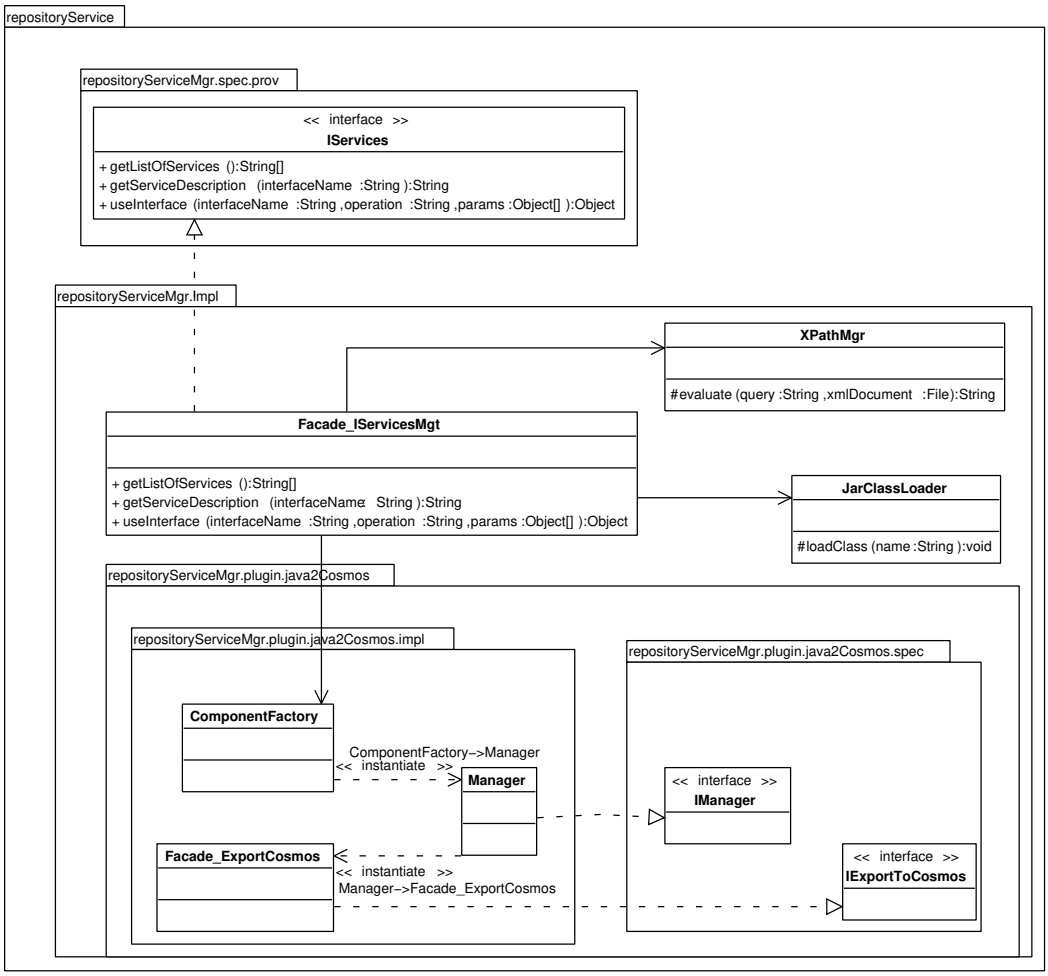

Figura 3.18: Projeto simplificado do GSRC. Algumas classes de infra-estrutura do COSMOS foram omitidas por simplificação.

Com o diretório e os arquivos criados adequadamente, a adição de um novo serviço está completa. Para usá-lo, o usuário deve utilizar a interface `IServices` (ver Figura 3.18) para listar os serviços disponíveis e depois escolher um deles. Esses serviços são oferecidos listando as interfaces providas pelas ferramentas e seus métodos.

A listagem da Figura 3.20 mostra um exemplo de um arquivo de descrição da ferramenta, que neste caso é o EvolutionChecker. Entre as linhas 2 e 8, são descritos os seguintes atributos da ferramenta: id, nome (*name*), caminho dentro da estrutura de diretórios do sistema (*path*), versão (*version*), localização do esquema (*xsi:schemaLocation*), pacote (*package*) e descrição da ferramenta (*description*). As demais informações não são importantes no escopo deste trabalho.

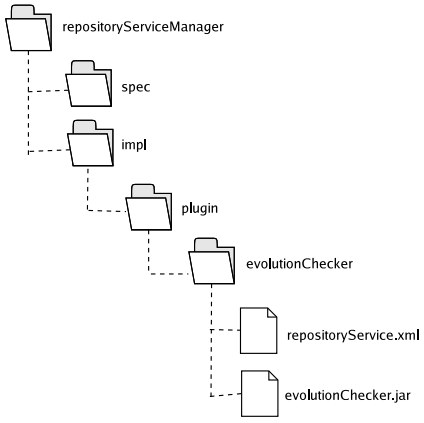

Figura 3.19: Estrutura de diretórios do GSRC.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <tns:repositoryService id="evolutionChecker" name="evolutionChecker"
3     path="/home/lsd/ra001973/workspace2/RepositoryServiceManager/src/repositoryService/
          impl/"
4     version="1.0.1" xmlns:tns="http://www.example.org/serviceDescriptor"
5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6     xsi:schemaLocation="http://www.example.org/serviceDescriptor ../../serviceDescriptor.
          xsd "
7     package="evolutionChecker" description="Este plug-in permite ao usuário verificar se
          as
8 regras de evolução estão sendo seguidas e determinar qual é o próximos número de
      versão do componente.">
9 </tns:repositoryService>
```

Figura 3.20: Exemplo de um arquivo de descrição de *plug-in*

3.5.4 Limitações

A exigência das ferramentas serem desenvolvidas em Java e adotarem o modelo COSMOS são limitações do GSRC. O uso de reflexão também é uma limitação sob o ponto de vista do desempenho [57]. Outra limitação é a ausência de uma interface gráfica para guiar o desenvolvedor na adição e remoção de ferramentas. A edição do arquivo de descrição de ferramentas poderia ser auxiliada por uma ferramenta melhorando a usabilidade do GSRC, por exemplo.

3.6 Resumo

Inicialmente foi apresentado o *framework* de componentes para desenvolver ferramentas baseadas em regras. Esse *framework* melhora a modificabilidade das ferramentas que o utilizam, o que é importante para acompanhar a evolução dos modelos apoiados pelas ferramentas.

Em seguida, foi apresentada a ferramenta EvolutionChecker, para aplicar o modelo SACE em especificações de componentes. O EvolutionChecker utiliza os arquivos de metadados do RAS para determinar o nível de impacto que a substituição de um componentes causa na arquitetura.

Também foi detalhado o modelo de implementação de componentes COSMOS, que apesar de não ser uma contribuição deste trabalho, é importante para entender as ferramentas CosmosChecker e Java2Cosmos.

A ferramenta CosmosChecker verifica se um componente satisfaz as restrições do modelo COSMOS. O componente deve ser escrito em Java e o CosmosChecker analisa o binário do componente e sua estrutura de diretório para determinar se um componente é ou não COSMOS.

Para complementar o uso do CosmosChecker foi desenvolvido o Java2Cosmos, uma ferramenta que converte componentes escritos em Java para o modelo COSMOS. Os componentes convertidos podem ser um conjunto de códigos-fonte ou um arquivo *jar*.

Por fim, foi apresentado uma ferramenta para administrar essas ferramentas e outras que eventualmente sejam desenvolvidas. A ferramenta é o Gerenciador de Serviços do Repositório de Componentes (GSRC). Ele utiliza mecanismos de *plug-ins* para facilitar a adição e remoção de ferramentas, aumentando a modificabilidade do sistema.

Capítulo 4

Estudos de Caso

Neste capítulo apresentamos dois estudos de caso e duas avaliações práticas das ferramentas. O estudo de caso do cenário integrado de uso das ferramentas (seção 4.1) analisa o uso conjunto da ferramentas Java2Cosmos, EvolutionChecker, Java2RAS e CosmosChecker. O estudo de caso do sistema bancário (seção 4.2) analisa a ferramenta CosmosChecker. A avaliação prática do Java2Cosmos (seção 4.3.1) mostra métricas relativas ao uso da ferramenta em componentes reais. A avaliação prática do CosmosChecker (seção 4.3.2) analisa a compatibilidade da ferramenta em relação ao ambiente Bellatrix. Por fim, na seção 4.4, os resultados são analisados.

4.1 Estudo de Caso 1 - Cenário integrado de uso das quatro ferramentas

4.1.1 Planejamento do estudo de caso

Para avaliarmos as ferramentas desenvolvidas neste trabalho, ao invés de avaliar cada uma delas isoladamente, criamos um cenário plausível que as interligasse. Com isso, além de avaliá-las podemos mostrar que elas fazem parte de um mesmo contexto.

A Figura 4.1 mostra o planejamento deste estudo de caso. As setas e os números auxiliam a entender a ordem dos acontecimentos e o resultado deles. (1.conversão) Escolhemos um componente que não é COSMOS e convertemos esse componente. (2.extração) Após convertido extraímos os metadados deste componente com o Java2RAS e criamos um arquivo de descrição no formato RAS para o componente. (3.evolução) Então evoluímos o componente e também seu metadado, para que este continue consistente. (4.verificação) Em seguida, avaliamos através do CosmosChecker se o componente evoluído continua sendo um componente COSMOS ou não. (5.análise de impacto) Por fim, a analisamos o

nível de impacto das alterações usando o EvolutionChecker e, com isso, preenchemos o número de versão do novo componente.

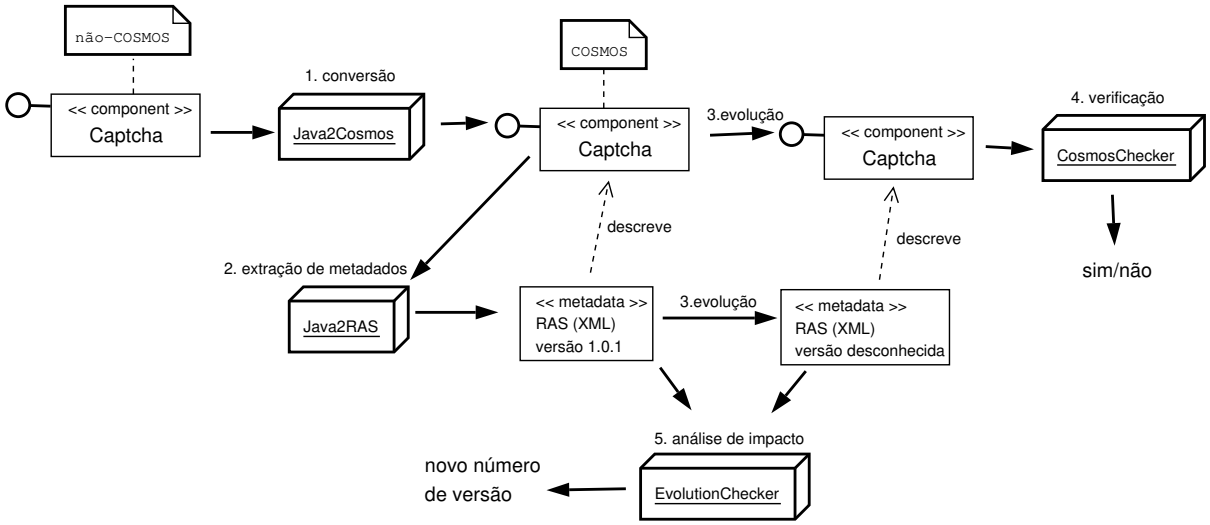

Figura 4.1: Planejamento do estudo de caso

4.1.2 Execução do estudo de caso

Primeiro, escolhemos a aplicação Petstore [7] versão 2.0 EA2[1] que ilustra como usar a plataforma Java EE 5 e outras tecnologias Java. Adotamos como componente o pacote chamado Captcha que possui cinco classes Java. Esse novo componente chamaremos de Captcha e ele não satisfaz as restrições do modelo de implementação COSMOS. Na primeira etapa do estudo de caso, convertemos o componente Captcha para COSMOS, usando o Java2Cosmos. O resultado é um componente que satisfaz as restrições do modelo de implementação COSMOS (ver seção 3.3.3 para discussão sobre as restrições do modelo de implementação COSMOS).

A Tabela 4.1 mostra algumas métricas relacionadas à conversão do componente Captcha em COSMOS. Foram criadas seis interfaces providas e sete classes totalizando 156 linhas de código-fonte criadas[2]. Cinco das seis interfaces foram criadas em decorrência da regra que cria uma interface provida para cada classe pública, já que um componente deve explicitar seus serviços através de interfaces providas. A sexta interface é a `IManager`, que pertence a infra-estrutura do COSMOS. Para cada interface criada, uma

[1]Na verdade, o nome completo da aplicação é *AJAX-enabled Web 2.0 Java Pet Store Reference Application for Java EE 5, 2.0 Early Access distribution*, mas para simplificar chamamos apenas de Petstore versão 2.0 EA2

[2]Não foram contadas linhas em branco ou comentários

classe `Facade` que implementa essa interface também foi criada. Além delas, também tem a classe `Manager`, que implementa `IManager`, e `ComponentFactory`. Ambas fazem parte da infra-estrutura do COSMOS.

Tabela 4.1: Dados sobre a conversão

Métricas	**Antes da conversão**	**Depois da conversão**
número de classes	5	12
número de interfaces	0	6
número de linhas de código	151	307

Na segunda etapa, os metadados do componente são extraídos usando o Java2RAS. A listagem a seguir mostra parte do XML gerado pelo Java2RAS com os metadados sobre o componente. Este XML segue o *profile* do Componente Abstrato e ele foi gerado corretamente, ou seja, de acordo com as especificações do XML. No trecho do arquivo mostrado, vemos os elementos (*tags*) *contextdependency* (linha 2) que é uma característica do componente abstrato e o elemento *interface* (linha 3) que mostra alguns metadados extraídos de uma das interfaces deste componente abstrato. Os atributos são o estado (*state*), o nome (*name*) e a direção (direction) da interface. Na linha 4 temos um elemento vazio chamado *relatedasset*, que é usado para relacionar a interface ao bem que a descreve.

```
1 <!-- trecho do XML gerado automaticamente pelo Java2RAS -->
2 <contextdependency>
3     <interface state="normal" name="IManager" direction="provided">
4         <relatedasset />
5     </interface>
6     ...
7 </contextdependency>
8 ...
```

Como descrito na seção 2.5.3, o Java2RAS possui algumas limitações e, por isso, parte dos metadados deve ser preenchida pelo desenvolvedor. Mais especificamente, no caso do componente abstrato, os campos nome e id que são obrigatórios precisam ser preenchidos manualmente para que o XML tornar-se válido. O preenchimento desses campos é necessário para obter um XML válido.

Na etapa seguinte, o componente foi evoluído usando a nova versão do Petstore disponibilizada na página da Sun, a versão 2.0 EA3[3]. As classes antigas foram substituídas pelas novas, sem fazer qualquer modificação no código-fonte. Uma classe pública foi removida, a CaptchaServlet, e dois métodos foram adicionados à classe SimpleCaptcha. Todavia, algumas modificações foram necessárias nas classes e interfaces geradas automaticamente

[3] *AJAX-enabled Web 2.0 Java Pet Store Reference Application for Java EE 5, Early Access version3.0 distribution*

pelo Java2Cosmos na etapa 1 para refletir a evolução do componente. Nesse caso, a interface que oferecia os serviços implementados por CaptchaServlet tornou-se obsoleta[4] e dois métodos foram adicionados à interface que oferece os serviços de SimpleCaptcha. Todas essas alterações foram refletidas nos metadados de seus componentes.

Após as alterações, existe a necessidade de comprovar se o componente que fora convertido para COSMOS continua satisfazendo as regras do modelo. Para isso, foi utilizado o CosmosChecker. A listagem a seguir mostra a saída obtida com verificação do componente. As linhas 1,2 e 3 mostram o resultado das regras que verificam a separação explícita entre especificação e implementação. Na linha 4 aparece o resultado da regra que averigua se todas as interfaces foram implementadas e listadas para uso. A linha 5 mostra a regra que verifica se existe a classe `ComponentFactory`, usada para instanciar o componente. Por fim, a linha 6 mostra que o componente apesar de alterado, continua satisfazendo as restrições do modelo COSMOS.

```
1 >> Check spec package:OK
2 >> Check impl package: OK
3 >> Check spec.prov package:OK
4 >> All interfaces were implemented:OK
5 >> Check ComponentFactory:OK
6 The component /home/lsd/ra001973/workspace2/estudoDeCaso_newPetstore/
    newPetstore/com/sun/javaee/blueprints/petstore/captcha is COSMOS
```

Após evoluir o componente e verificar que ele continua condizente com as restrições do modelo COSMOS, é necessário determinar através do EvolutionChecker seu impacto na arquitetura e conseqüentemente determinar seu número de versão. O EvolutionChecker analisa os metadados do componente, o da versão antiga e da atual, e aplica as regras de evolução. Neste caso, o que ocorreu foi que a interface que oferecia os serviços de CaptchaServlet tornou-se obsoleta, ou seja, mudou seu estado de manutenção (ver apêndice A).

A seguir está a listagem que apresenta o resultado da análise dos dois metadados. Como era esperado, somente entre as linhas 23 e 25 ocorreu uma modificação no estado de manutenção da interface, que tem impacto alto na arquitetura. O reflexo deste impacto é mostrado nas linhas 43 e 44, onde o componente muda do número de versão 1.0.1 para 2.0.1 (ver seção 2.2 para mais detalhes sobre o modelo de versionamento).

```
1 starting evolution checker...
2 Rules /home/lsd/ra001973/workspace2/EvolutionChecker/src/rules/fenixRule.
    drl were successfully defined
3 >>fire type of required interface rule...
4 Operation performed: NOTHING
5 Impact level: INSIGNIFICANT
```

[4]do inglês *deprecated*

```
6
7 >>fire type of provided interface rule...
8 Operation performed: NOTHING
9 Impact level: INSIGNIFICANT
10
11 >>fire set Of Interfaces of a Port rule...
12 Operation performed: NOTHING
13 Impact level: INSIGNIFICANT
14
15 >>fire set of ports rule...
16 Operation performed: NOTHING
17 Impact level: INSIGNIFICANT
18
19 >>fire set of interfaces rule...
20 Operation performed: MODIFICATION
21 Impact level: INSIGNIFICANT
22
23 >>fire maintenance state of a interface rule...
24 Operation performed: MODIFICATION
25 Impact level: HIGH
26
27 >>fire maintenance state of a port rule...
28 Operation performed: NOTHING
29 Impact level: INSIGNIFICANT
30
31 >>fire Target Platform rule...
32 Operation performed: NOTHING
33 Impact level: INSIGNIFICANT
34
35 >>fire Synchronization contract rule...
36 Operation performed: NOTHING
37 Impact level: INSIGNIFICANT
38
39 >>fire Quality of Service rule...
40 Operation performed: NOTHING
41 Impact level: INSIGNIFICANT
42
43 The old version was 1.0.1
44 The new version is 2.0.1
45 execution time:9.014 seconds
```

4.2 Estudo de caso 2 - Um sistema bancário baseado em componentes

4.2.1 Planejamento do estudo de caso

Para avaliar a ferramenta CosmosChecker, utilizamos componentes COSMOS desenvolvidos para um sistema bancário real baseado em componentes. Esse sistema bancário possui seis funcionalidades:

- Requisição de talões de cheques
- Entrega de talões de cheques
- Sustação de cheques
- Captura de cheque para compensação
- Cancelamento do contrato da conta
- Cadastramento de limite adicional

Esse sistema foi implementado para avaliar um método de modelagem de exceções em DBC, apresentado por Brito [30]. Por ser um sistema real inteiramente desenvolvido em COSMOS, este sistema foi escolhido para este estudo de caso. Um dos desenvolvedores do sistema aplicou o CosmosChecker em todos os 21 componentes COSMOS do sistema. Ao final do estudo de caso, o desenvolvedor preencheu um questionário onde avaliou a usabilidade da ferramenta (ver apêndice C). Desta forma, foi avaliado não somente os resultados da ferramenta como também sua usabilidade.

4.2.2 Execução do estudo de caso

O desenvolvedor dos componentes COSMOS foi brevemente instruído sobre a utilização do CosmosChecker e como seria a avaliação da ferramenta. Ele executou o CosmosChecker para cada um dos 21 componentes e marcou o tempo de duração do estudo de caso e gravou o resultado das execuções em um arquivo.

A Tabela 4.2 resume os resultados do estudo de caso. Todos componentes analisados passaram por todas as regras de verificação do modelo COSMOS e, portanto, a ferramenta acertou todas as avaliações. O tempo total do estudo de caso, excluindo as instruções iniciais, foi de 22 minutos. Um defeito foi encontrado e já corrigido. Por fim, duas sugestões de novas funcionalidade e duas de melhorias na usabilidade foram feitas.

Tabela 4.2: Resultados do estudo de caso do CosmosChecker

Porcentagem de acertos	100%
Tempo total do estudo de caso (min)	22
Defeitos encontrados	1
Sugestões de novas funcionalidades	2
Sugestões de melhorias na usabilidade	2

4.3 Outras avaliações práticas

4.3.1 Métricas do Java2Cosmos usando componentes reais

Foram realizadas avaliações práticas para avaliar a ferramenta Java2Cosmos usando somente arquivos *jar*. As métricas que escolhemos para avaliar foram as seguintes: o tamanho do arquivo *jar*, que pode indicar que estamos lidando com diferentes granularidades de componentes; o tempo de execução médio, para estimar o tempo que o usuário terá que esperar pela conversão; o número de classes e linhas de código geradas automaticamente, que representa o trabalho poupado pelo desenvolvedor, e o número de erros de sintaxe, o que mostra se a conversão foi bem sucedida ou não. O computador usado para coletar os dados foi um AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ com 1.0 GB de memória. Os passos foram os seguintes: (1) foram selecionados os cinco componentes mais vistos na categoria Java na página do ComponentSource (CS)[5]. Escolhemos componentes produzidos em diferentes empresas. Também escolhemos os quatro[6] arquivos *jar* mais relevantes do SourceForge (SF)[7]; (2) convertemos cada arquivo *jar* três vezes para estimar mais precisamente o tempo médio de execução; (3) compilamos o código gerado para encontrar os erros de sintaxe.

Os dados coletados são apresentados na Tabela 4.3.

4.3.2 Compatibilidade entre a ferramenta CosmosChecker e o gerador de código do Bellatrix

Como já foi apresentado na seção 1.1, o Bellatrix é um IDE que, entre outras funcionalidades, gera esqueleto de código-fonte escrito em Java compatível com o modelo COSMOS. Para avaliar a compatibilidade entre o gerador do Bellatrix e o CosmosChecker, seis componentes gerados a partir do Bellatrix foram analisados pelo CosmosChecker. O esqueleto do código-fonte foi gerado automaticamente pelo Bellatrix e o restante do código foi implementado por um desenvolvedor.

[5]O ComponentSource é um dos principais vendedores de componentes do mercado [http://www.componentsource.com]

[6]Inicialmente escolhemos cinco componente, mas um deles era incompilável e por isso foi descartado

[7]http://sourceforge.net

Tabela 4.3: Métricas extraídas dos componentes em formato de arquivo *jar*

Arquivo jar	**Tamanho** (kb)	**Tempo Médio de execução(s)**	**Número de classes geradas**	**Número de linhas de código geradas**	**Número de erros**
mcbeans (CS)	285	24.22	147	4018	0
mcharts (CS)	931	28.92	897	34475	0
mdemo (CS)	1528	25.24	701	10139	0
SMPSMVDemo (CS)	388	-	-	-	1
djt (CS)	248	11.31	187	13555	2
CWEvolver (SF)	85	6.73	73	2322	13
JarUpdator (SF)	44	5.35	2	505	0
Jenia4faces (SF)	750	10.75	223	10825	14
One-jar-boot (SF)	27	5.25	9	262	0
Média	476	14.72	280	9513	3.33

Os seis componentes fazem parte de um sistema de cadastro de alunos universitários. Esse sistema deve possibilitar que os alunos se cadastrem em matérias de um determinado semestre pela internet. Além disso, o sistema deve manter um histórico das disciplinas cursadas pelos alunos. Trata-se de um sistema real que ainda está em desenvolvimento e é descrito em detalhes no trabalho de Moronte [51].

O resultado da avaliação é que os seis componentes satisfazem as restrições do modelo COSMOS implementadas pelo CosmosChecker.

4.4 Avaliação Geral dos resultados

4.4.1 Java2RAS

Na estudo de caso do cenário integrado das ferramentas (seção 4.1), os metadados extraídos pelo RAS preencheram 88% dos campos obrigatórios do *profile* de Componente Abstrato, o que é satisfatório. O arquivo XML gerado é válido, ou seja, não viola nenhuma das especificações do XML e corresponde corretamente ao esquema do Componente Abstrato. Os campos obrigatórios que foram preenchidos manualmente foram: o nome do componente abstrato e seu ID. O nome do componente não necessariamente aparece em seu código-fonte. O ID poderia até ser gerado automaticamente, entretanto, existem repositórios que possuem políticas próprias para definir um ID, por exemplo, IDs com 16 caracteres alfanuméricos. Por isso, optou-se por deixar este campo vazio.

4.4.2 EvolutionChecker

No estudo de caso da seção 4.1, o EvolutionChecker identificou a alteração do estado de uma interface e classificou essa alteração com um impacto alto na arquitetura onde

o componente está inserido. Essa avaliação ocorre em conformidade com o modelo de evolução de Lobo [49]. Visto que o impacto é alto, o número de versão do componente abstrato passou de 1.0.1 para 2.0.1, como esperado.

4.4.3 CosmosChecker

No estudo de caso do cenário integrado das ferramentas (seção 4.1), a alteração realizada no componente, a de mudar o estado de uma interface de normal para obsoleta, não afeta nenhuma das diretrizes do COSMOS. Portanto, era esperado que o resultado do CosmosChecker fosse positivo para avaliar o componente como COSMOS.

No estudo de caso do sistema bancário (seção 4.2), o CosmosChecker corretamente classificou todos os componentes como COSMOS. O pouco tempo necessário para avaliar os componentes é um indicativo de que a ferramenta é simples e prática. Ainda nesse estudo de caso foi identificado um pequeno defeito na ferramenta, que já foi corrigido. Por fim, o usuário sugeriu mudanças nas funcionalidades (mais duas regras para melhor identificar um componente COSMOS) e sugestões de usabilidade.

Em relação à avaliação de compatibilidade com o Bellatrix (seção 4.3.2), o resultado foi satisfatório pois todos os componentes foram classificados como COSMOS.

4.4.4 Java2Cosmos

O Java2Cosmos converteu corretamente o componente Captcha, no estudo de caso da seção 4.1. Substituímos o original pelas duas versões do componente convertido e a aplicação Petstore funcionou normalmente com ambas as versões. Para utilizar esse componente, antes foi necessário compilá-los e nenhum erro foi encontrado. As únicas adaptações necessárias no código-fonte original foram nas classes que utilizam o componente Captcha. A seguir, um exemplo dessas modificações. Na linha 1, comentada, aparece o código-fonte original. Nele, o acesso às classes de implementação era direto. A partir da linha 2 está o código-fonte adaptado, que utiliza da infra-estrutura oferecida pelo COSMOS para acessar as classes de implementação de forma indireta.

```
1 //SimpleCaptcha captcha = new SimpleCaptcha();
2 IManager imanager = ComponentFactory.createInstance();
3 Icom_sun_javaee_blueprints_petstore_captcha_SimpleCaptcha captcha = (
    Icom_sun_javaee_blueprints_petstore_captcha_SimpleCaptcha) imanager.
    getProvidedInterface("
    Icom_sun_javaee_blueprints_petstore_captcha_SimpleCaptcha");
```

Na avaliação com componentes reais (seção 4.3.1), a segunda coluna da Tabela 4.3 mostra uma grande variedade de tamanhos de arquivos, o que significa que diferentes granularidades de componentes foram escolhidas. Os resultados obtidos com o tempo de

execução foram bons se considerarmos o trabalho poupado pelo desenvolvedor. Linhas em branco e comentários não foram contados na medição de linhas de código geradas.

Os resultados dessa avaliação evidenciaram também algumas limitações da ferramenta. Classes aninhadas encontradas nos arquivos *jar* não são tratadas pela ferramenta, gerando código-fonte não-compilável que resultam em 47% dos erros de sintaxe. Isso porque as classes aninhadas exigem uma instanciação um pouco mais complexa que as classes normais e o Java2Cosmos não lida com essa complexidade extra. Outro problema de sintaxe foram os construtores não-visíveis pelas classes Facade que os utilizam. Isso ocorreu por dois motivos: o primeiro motivo é que as instâncias de algumas classes são obtidas através de outras classes (as fábricas) e essa indireção não é compreendida pela ferramenta. O segundo motivo, é que o construtor de algumas classes possui visibilidade de pacote e pode ocorrer de o Facade não estar nesse mesmo pacote, portanto, o construtor torna-se inacessível para o Facade. Essa limitação correspondeu a 50% dos erros de sintaxe. Por fim, algumas dependências não foram resolvidas e resultaram em 3% dos erros.

Apesar dessas limitações, apenas três tipos de erros de sintaxe foram detectados, entre mais de 2000 classes geradas automaticamente, e os erros de sintaxe gerados pelas classes aninhadas podem ser resolvidos com pouco esforço de programação.

Uma limitação desta avaliação é que algumas características importantes como usabilidade e manutenibilidade não foram avaliadas. Outra limitação é que os testes foram realizados apenas com arquivos *jar* e não com código-fonte.

4.5 Resumo

Neste capítulo foram apresentados dois estudos de caso e duas avaliações práticas das ferramentas. O primeiro estudo de caso criou um cenário hipotético de uso integrado de todas as ferramentas. Foi utilizado uma aplicação real, a Petstore 2.0, para avaliar o uso conjunto das ferramentas.

O segundo estudo de caso utilizou um sistema bancário para avaliar o CosmosChecker. O sistema bancário foi todo desenvolvido em Java e todos seus componentes eram COSMOS. O desenvolvedor deste sistema utilizou o CosmosChecker para determinar se os componentes realmente eram COSMOS e também avaliou a usabilidade da ferramenta.

A primeira avaliação prática analisou algumas métricas de uso do Java2Cosmos. Foram convertidos componentes reais recuperados do ComponentSource e do SourceForge para obter as métricas.

A segunda avaliação prática analisa a compatibilidade entre o CosmosChecker e o ambiente de desenvolvimento Bellatrix. Os componentes gerados pelo Bellatrix foram analisados pelo CosmosChecker, para determinar se eram ou não COSMOS.

Capítulo 5

Conclusões e Trabalhos Futuros

Neste capítulo, este documento é encerrado apresentando as conclusões, as contribuições e os trabalhos futuros.

5.1 Conclusões

O apoio aos modelos de evolução SACE e de implementação de componentes COSMOS, é uma atividade sujeita à erros humanos. Por isso, são necessárias ferramentas para automatizar o suporte a esses modelos. Contudo, os modelos podem evoluir para acompanhar novas tecnologias ou necessidades de seus usuários. Essa evolução dos modelos por sua vez pode implicar na evolução das ferramentas que apóiam esses modelos. Portanto, é desejável que as ferramentas de apoio aos modelos sejam facilmente modificáveis.

Para apoiar a evolução das ferramentas baseadas em regras, este trabalho propõe um *framework* de componentes baseado no motor de inferência Drools, que externaliza as regras tornando a ferramenta mais modificável. Com a utilização deste *framework* de componentes, foram criadas quatro ferramentas: Java2RAS, EvolutionChecker, CosmosChecker e Java2Cosmos. As duas primeiras apóiam o modelo de evolução SACE e as duas últimas apóiam o modelo de implementação COSMOS. Desta forma, as ferramentas aumentam a produtividade do desenvolvedor e reduzem as falhas humanas. Além disso, elas são modificáveis e, portanto, podem acompanhar a evolução dos modelos.

Para dar suporte ao modelo de evolução SACE, foi necessário criar novos *profiles* do RAS. Esses *profiles* são usados pelo repositório de bens Rigel que, com as contribuições deste trabalho, oferece suporte à evolução de componentes.

Outro problema no contexto de desenvolvimento de software é que diferentes empresas podem usar diferentes ferramentas para dar suporte aos modelos de evolução e implementação usados por ela. Para facilitar a administração dessas ferramentas foi criado o GSRC, que utiliza mecanismos de *plug-ins* semelhantes aos do Eclipse para facilitar

a adição e remoção de ferramentas.

Este conjunto de soluções forma uma **infra-estrutura para apoiar a evolução em um repositório de componentes**. Os estudos de caso e avaliações visaram mostrar a utilidade das ferramentas, bem como alguns requisitos funcionais e atributos de qualidade, dentro de um contexto de DBC. Contudo, são necessários estudos mais rigorosos, dentro de um processo de desenvolvimento de uma empresa, por exemplo. Esses estudos ajudariam a identificar as vantagens e desvantagens do uso de toda a infra-estrutura.

5.2 Contribuições

Como contribuições deste trabalho ressaltamos:

- **O *Framework* para desenvolvimento de ferramentas baseadas em regras -** O *framework* apóia a evolução das ferramentas, que por sua vez apóiam os modelos. Como os modelos evoluem, as ferramentas devem também evoluir.

- **O Gerenciador de Serviços do Repositório de Componentes (GSRC)-** o GSRC oferece mecanismos para facilitar a adição e remoção de ferramentas que implementam os serviços do repositório de componentes. Como esses serviços podem variar bastante de empresa para empresa, o GSRC ajuda a lidar com essa variedade.

- **Automatização do modelo de evolução -** As ferramentas Java2RAS e EvolutionChecker contribuem para a adoção e utilização do modelo de evolução proposto por Lobo. A ferramenta Java2RAS auxilia o desenvolvedor na criação dos metadados sobre os bens enquanto o EvolutionChecker apóia a aplicação das regras de evolução e do modelo de versionamento.

- **Automatização do modelo COSMOS -** A ferramenta Java2Cosmos promove a evolução da implementação de um componente, convertendo-o ao modelo COSMOS. Já o CosmosChecker atua no outro sentido, o de contribuir para que componentes continuem em conformidade com o modelo COSMOS.

- **A criação dos novos profiles do RAS** - Os novos *profiles* são essenciais para fornecer os metadados necessários ao modelo de evolução de componentes. Além disso, eles detalham a descrição dos bens do repositório Rigel, contribuindo para reutilização de componentes abstratos e concretos, definições de interfaces e configurações arquiteturais. Segundo Frakes [37], o entendimento é um dos fatores que estimulam o reuso.

5.2.1 Publicações

- Leonardo P. Tizzei, Helder Pinho, Paulo A. Guerra e Cecília M.F. Rubira. *Um repositório de componentes com suporte a evolução centrada na arquitetura de software.* Workshop de Desenvolvimento Baseado em Componente - Juiz de Fora, MG - Brasil, 2005.

- Leonardo P. Tizzei, Paulo A. Guerra e Cecília M.F. Rubira. *Ferramentas para automatizar a evolução e reutilização de componentes em um repositório de componentes.* Simpósio Brasileiro de Linguagens de Programação - Natal, RN - Brasil, 2007. (*Submetido para publicação*)

- Leonardo P. Tizzei e Cecília M.F. Rubira. *EvolutionChecker: Uma Ferramenta para apoiar a evolução de componentes* - Relatório Técnico - Instituto de Computação - UNICAMP, 2007 (*a ser publicado*).

- O código-fonte das ferramentas será disponibilizado a todos através da Internet futuramente. Para isso, algumas melhorias citadas na seção 5.3 devem ser implementadas. A licença para utilizar as ferramentas também deve ser discutida.

5.3 Trabalhos futuros

Com respeito à toda a infra-estrutura de apoio à evolução, ou seja, novos *profiles* do RAS, o GSRC e as ferramentas:

- um estudo de caso real, realizado em uma empresa onde a toda a infra-estrutura fosse usada e avaliada pelos funcionários dessa empresa seria um bom parâmetro sobre as vantagens e desvantagens do trabalho proposto.

- o trabalho de Kotonya e Hutchinson [46] oferece uma abordagem complementar ao modelo de evolução proposto por Lobo, uma vez que eles analisam a propagação da evolução. A análise feita por Lobo se restringe ao impacto que o novo componente terá na arquitetura, mas não na propagação deste impacto.

- a integração entre Bellatrix, infra-estrutura de apoio à evolução e Rigel é um trabalho conjunto do grupo de pesquisa onde este trabalho foi desenvolvido.

- integração entre Rigel, infra-estrutura de apoio à evolução e a Rede de Compartilhamento de Componentes de Software (RCCS) [53] propiciaria uma oportunidade ímpar para avaliar o Rigel e o GSRC sob o ponto de vista dos consumidores de componentes.

Com respeito aos novos *profiles* do RAS:

- O uso de ontologias para descrever componentes e/ou ajudar na recuperação desses é bastante difundida [26, 58, 41, 64, 60]. A adoção de ontologias para esses fins é uma hipótese em aberto, que poderiam exigir mudanças no projeto do Rigel.

Com respeito às ferramentas:

- as interfaces gráficas das ferramentas são protótipos que visam melhorar a usabilidade da ferramenta, mas ainda estão longe do ideal. Por exemplo, é necessária a criação de um menu de ajuda.

- a usabilidade das ferramentas não foi avaliada e deve ser avaliada dentro de um estudo de caso envolvendo usuários que não conhecem as ferramentas.

Com respeito ao EvolutionChecker:

- a evolução da configuração arquitetural e da interface não são consideradas no modelo de evolução e, consequentemente, nem no EvolutionChecker. A evolução da arquitetura é um campo com trabalhos recentes [56, 52] assim como a evolução da interface [48], cujos resultados podem ser adotados ou adaptados para o contexto deste trabalho.

Com respeito ao Java2Cosmos:

- Avaliações preliminares do Java2Cosmos mostraram que nem todas as estruturas de Java são corretamente convertidas. Por exemplo, classes aninhadas geram código-fonte incorreto. Eliminar essas limitações ampliaria o número de estruturas corretamente convertidas.

Com respeito ao CosmosChecker:

- Implementar as sugestões dadas no estudo de caso da seção 4.2. Foram dadas duas sugestões de melhorias na usabilidade e duas de novas funcionalidades.

Apêndice A

Novos *Profiles* do RAS

A.1 Extensões do RAS

O RAS é dividido em *Core RAS* e *Profiles*. O *Core RAS* descreve os elementos básicos de uma especificação e os *Profiles* descrevem as extensões desses elementos. Todavia, o *Profile* não pode alterar a definição ou a semântica definida no *Core RAS*. Os *Profiles* podem ser estendidos ou criados para melhor se adequar a um determinado modelo. Com essa intenção, *Default Component Profile* (um *Profile* já especificado pelo RAS) serviu de base para a criação de novos quatro *Profiles*: *Componente Abstrato*,*Componente Concreto*, *Definição de Interface* e *Configuração*. Para isso, deixamos de seguir o *Default Component Profile*, mas continuamos seguindo o *Default Profile* que por sua vez implementa o *Core RAS*. A seguir, são apresentados o *Profile de Definição de Interface*, o *Profile do Componente Abstrato*, o *Profile de Componente Concreto* e o *Profile de Configuração*.

Apenas os novos elementos dos *profiles* são descritos neste capítulo. Os demais elementos são descritos na especificação do RAS [54]

A.1.1 *Profile* da Definição de Interface

Breve descrição

O ***Profile* da Definição de Interface**, apresentado na figura A.1, descreve um tipo de interface. Poucas alterações foram realizadas sobre o *Default Component Profile* descrito pelo RAS. Um novo elemento foi criado chamado **InterfaceDefinition** (Definição de Interface) para caracterizar a interface. A motivação para criarmos este *Profile* é permitir a reutilização de definições de interfaces e junto com elas os artefatos que ajudam a especificá-la, como casos de uso, modelo de informação entre outros.

Novos elementos

InterfaceDefinition
A classe *InterfaceDefinition* (Definição de Interface) define o tipo da interface. O atributo *name* define um nome único para o tipo de interface. Já *description* oferece um breve descrição sobre o tipo da interface e *development_state* classifica o tipo de interface em três possíveis estados: *normal, deprecated* e *removed*. Esse atributo nos permitirá remover uma interface, que por alguma razão se tornou inútil, de forma que os componentes que usam aquela interface sejam avisados que ela será eliminada e possam ser preparados para isso. Por fim, quando um *InterfaceDefinition* estende outra *InterfaceDefinition* é representado através da ligação entre *InterfaceDefinition* e *RelatedAsset* (Bem relacionado), que nesse caso seria a *InterfaceDefinition* que foi estendida.

A.1.2 *Profile* do Componente Abstrato

Breve descrição

O *Profile* do Componente Abstrato mostrado na Figura A.2 visa prover o usuário com informações sobre a especificação do componente, suas dependências e contratos. Este *Profile* relaciona-se com o *Profile* de Definição de Interface, umas vez que a especificação de um componente possui pelo menos uma interface.

Novos elementos

AbstractComponent: A classe *AbstractComponent* (Componente Abstrato) tem um identificador único, o atributo *name*, dois atributos para representar os contratos especificados por aquela interface. Qualidade de serviço e sincronização são representados, respectivamente, por *quality_of_service_contract* e *synchronization_contract*. Por fim, *target-platform* especifica a plataforma para a qual o componente foi desenvolvido.

ExternalProperty:representa as propriedades externas do sistema, que são as interfaces e portas. Ou seja, a parte visível externamente.

ContextDependency: As dependências de contexto são representadas em *Context-Dependency*. Todas as interfaces, providas e requeridas, são dependências de contexto.

Interface: A classe *Interface* representa as interfaces do componente abstrato. Ela possui um *name* (nome), que é um identificador único de interface, *direction* (direção) que informa se ela é provida (*provided*) ou requerida (*required*) e *interfaceDefinitionName* (nome da definição de interface) que especifica o seu tipo. Um detalhe importante é que a classe *Interface* está ligada com *RelatedAsset*, para representar a ligação entre a *Interface*

e sua *InterfaceDefinition*.

Port: A classe *Port* (porta) representa um conjunto de interfaces e possuem apenas um identificador único: *name*.

A.1.3 *Profile* do Componente Concreto

Breve descrição

A Figura A.3 mostra os elementos do *profile* do Componente Concreto. Neste *profile* utilizamos duas formas de representar um componente concreto: o *CompositeComponent* (componente composto) e *ElementaryComponent* (componente elementar). Como possui características de fase de Projeto e da fase de Implementação, *CompositeComponent* compõem tanto *Design* quanto *Implementation* (Implementação).

Este *Profile* relaciona-se com o Componente Abstrato, pois todo componente concreto implementa um componente abstrato.

Novos elementos

Implementation: A classe **Implementation** contém outras classes que determinam a implementação do componente concreto descrito. Além disso, ela possui duas ligações com **RelatedAsset**: uma de dependência e outra de realização. A dependência ocorre quando o bem é um componente elementar que depende de outras bibliotecas e/ou ferramentas. A realização resulta do fato que todo componente concreto implementa um componente abstrato, identificado pelo bem relacionado.

ElementaryComponent: A classe **ElementaryComponent** representa um componente elementar, ou seja, um componente concreto sem subcomponentes. O código do componente é representado por **Artifact** (artefato). Esse componente concreto implementa um componente abstrato, representado como **RelatedAsset**. Um componente concreto depende de **External Libraries** (bibliotecas externas) e **Tools** (ferramentas), que também são **RelatedAsset**. **CompositeComponent** representa um componente concreto composto que possui uma arquitetura interna e um código interno, representados pelas classes **ComponentBasedView** e **Artifact**, respectivamente.

ComponentBasedView (Visão baseada em componente) representa a arquitetura interna do componente concreto e composto por **InterfaceConnection** (Conexão de interface), **ServiceConnection** (Conexão de serviços) e **Subcomponents** (subcomponentes).

Subcomponent: Os subcomponentes são representados pela classe **Subcomponents**. Um detalhe importante é que esta classe possui como atributo um **id** que usado para distinguir dois subcomponentes de um mesmo tipo de componente abstrato. Além disso, cada subcomponente está relacionado com seu tipo através do relacionamento entre **Subcomponents** e um **RelatedAsset**, que nesse caso é um componente abstrato. Outro detalhe importante é a relação entre as conexões e os subcomponentes, que indicam qual componente abstrato implementa uma determinada Interface.

InterfaceConnection: Esta classe representa as conexões entre interfaces. Entretanto, pode existir mais de uma interface com um determinado nome dentro de uma arquitetura interna. Por isso, precisamos distinguir as duas interfaces de mesmo nome, para saber exatamente quais são os dois componentes ligados pela conexão. No RAS, a representação de um **interface connection** ocorre da seguinte forma:

```
[absCompID]:[subCompID]:[interfaceName]::[absCompID]:[subCompID]:[interfaceName]
```

Onde `absCompID` é o identificador único do componente abstrato, `subCompID` é o identificador do subcomponente abstrato dentro de uma arquitetura interna e `interfaceName` é o nome da interface. Assim, é possível distinguir dois componentes abstratos de um mesmo tipo usados na arquitetura, já que eles possuem `subCompIDs` distintos. Também possível distinguir dois componentes abstratos de um mesmo tipo mas de versões diferentes, uma vez que ele possuem `absCompIDs` diferentes. Por fim, como cada interface de um componente abstrato tem um nome diferente, não haverá duas **interface connections** iguais.

A.1.4 *Profile* da Configuração

Breve Descrição

O *Profile* de Configuração, mostrado na Figura A.4, permite ao usuário versionar uma configuração concreta. A configuração é a materialização de uma arquitetura especificada pelo componente concreto composto. Essa arquitetura especifica o relacionamento entre os componentes abstratos. Cabe a configuração indicar qual é o componente concreto que implementa um determinado componente abstrato.

Novos Elementos

Configuration é o elemento que identifica uma configuração. Ele possui um atributo **id** para caracterizar unicamente uma configuração. A configuração é a materialização de

uma arquitetura, que pode ser representada como um bem relacionado. Isso justifica o relacionamento com **RelatedAsset**.

ConcreteInstance (instância concreta) é o elemento que liga um subcomponente de um componente concreto composto a um componente concreto. Este elemento relaciona-se com um componente abstrato.

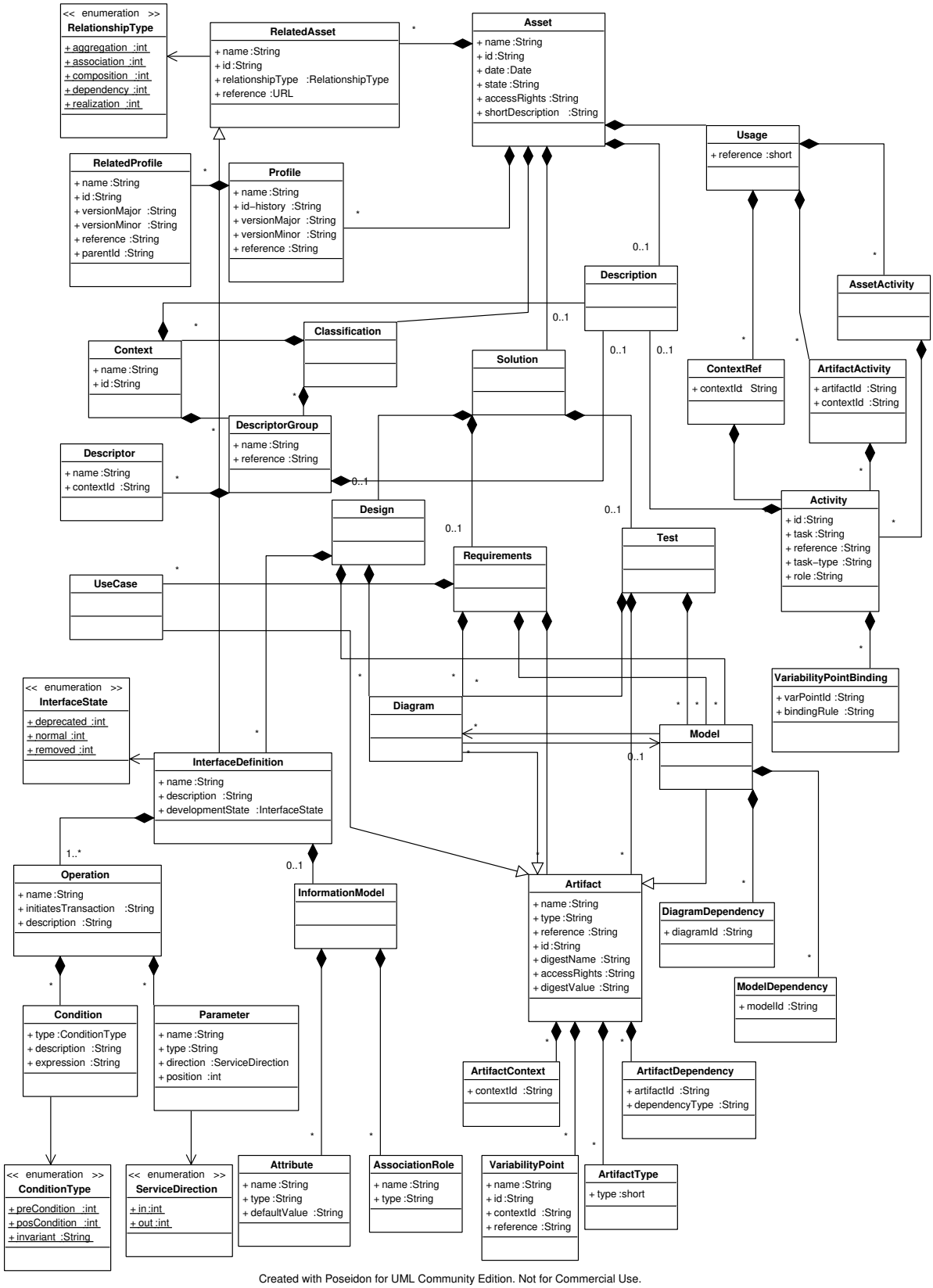

Figura A.1: Profile de Definição de Interface

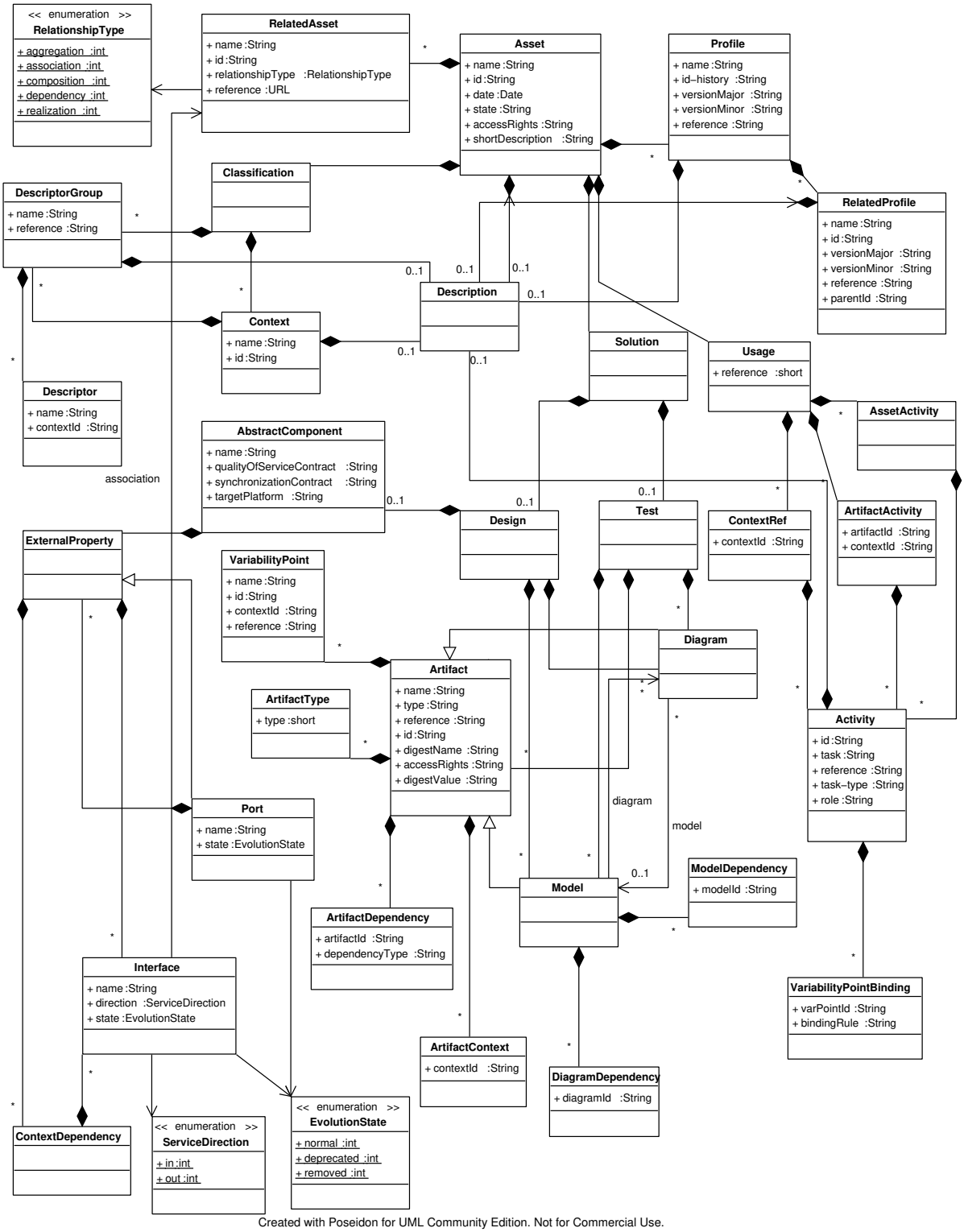

Figura A.2: Profile do Componente Abstrato

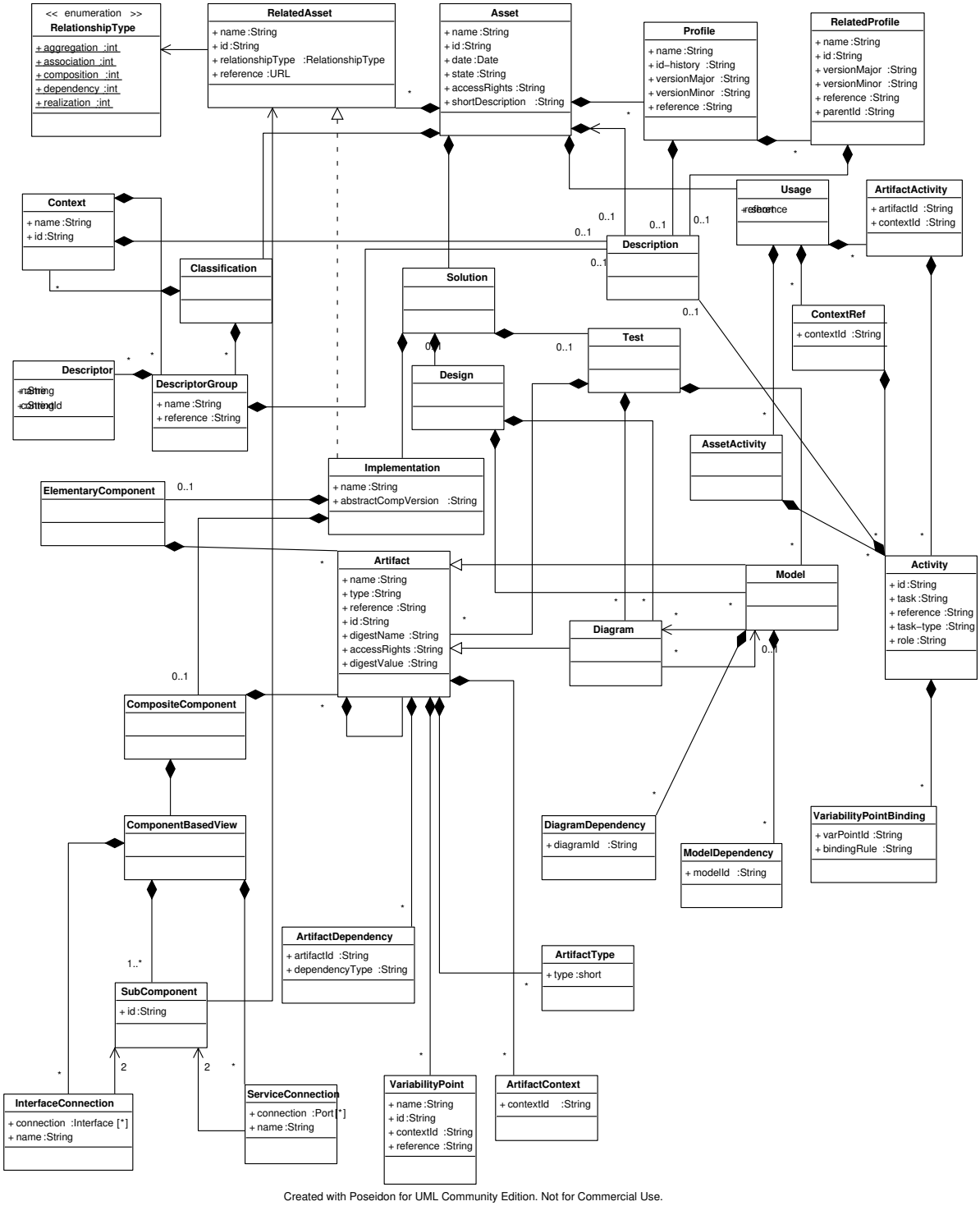

Figura A.3: Profile do Componente Concreto

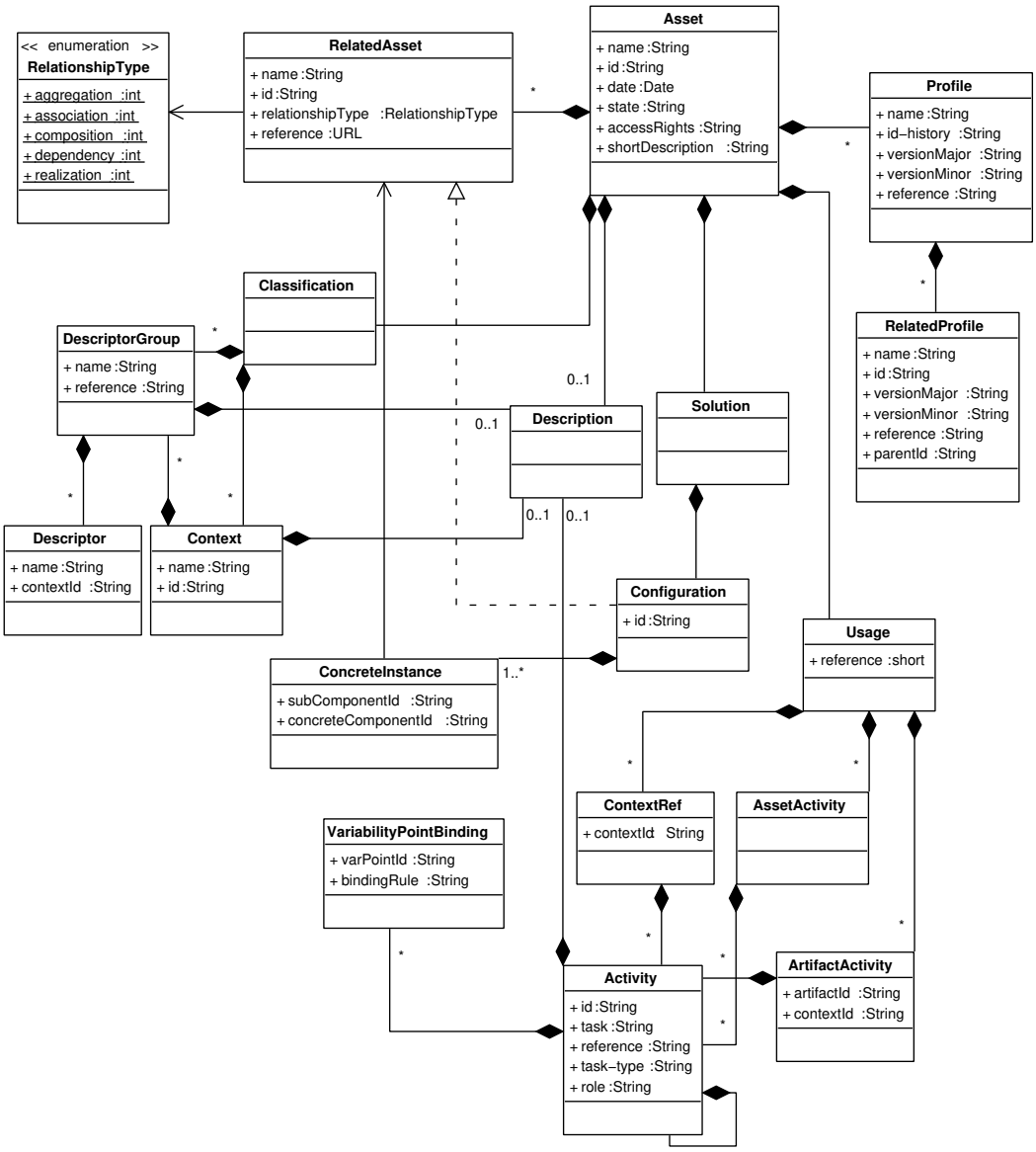

Figura A.4: Profile da Configuração

Apêndice B

Análise da arquitetura do sistema através do SAAMER

B.1 Avaliação da arquitetura do geral do sistema com o uso SAAMER

B.1.1 Introdução ao método SAAMER

O SAAMER (*Software Architecture Analysis Method for Evolution and Reusability*) [50] é uma extensão do SAAM [45] sob o ponto de vista da evolutibilidade e da reusabilidade. O SAAMER avalia mais precisamente esses atributos de qualidade em uma arquitetura de software. Assim como o SAAM, no SAAMER os cenários são utilizados para descrever uma funcionalidade que o sistema deve conter ou onde ele pode mudar no decorrer do tempo.

Neste capítulo, a arquitetura geral do sistema será avaliada seguindo os passos proposto pelo método SAAMER. O repositório Rigel, apesar de não fazer parte diretamente deste trabalho, também é usado na avaliação uma vez que a infra-estrutura para dar suporte à evolução foi desenvolvida para apoiar a sua utilização.

B.1.2 Objetivos

A Tabela B.1 mostra o relacionamento entre os *stakeholders*[1]. Para cada objetivo do *stakeholder*, o arquiteto define quais serão os objetivos arquiteturais do sistema. Esses objetivos possuem atributos de qualidade associados a eles. A partir desses objetivos é que serão criados os cenários.

[1] os interessados no sistema, como desenvolvedores, arquitetos, compradores, usuários etc...

Tabela B.1: Relacionamento entre os objetivos dos *stakeholders*, arquitetos de software e atributos de qualidade

Objetivos do *Stakeholder*	**Objetivos Arquiteturais**	**Atributos de qualidade**
Permitir a adição e remoção de novas ferramentas com facilidade	Prover pontos para extensão do gerenciador de serviços e diminuir a dependência entre os componentes	Modificabilidade
Permitir a utilização do repositório de componentes para pesquisa, adição, obtenção e remoção de componentes	Prover suporte para diferentes modelos de componentes	Portabilidade, usabilidade
Permitir que novos modelos de componentes sejam utilizados	Evitar acoplar fortemente os componentes relacionados com modelos e controle para facilitar a adoção de novos modelos	Manutenibilidade, modificabilidade
Permitir alterar as ferramentas de suporte ao repositório	Utilizar padrões de projeto adequados para facilitar a alteração	Modificabilidade

B.1.3 Fluxo funcional

Nesta seção, apenas os cenários que representam **funcionalidades** do sistema estão representados através de fluxo funcional. A Figura B.1 é a base para a construção dos fluxos.

Inserção/Remoção de componentes

1. IDE/Web
2. GSRC
3. RetrieveAndEdit
4. AssetMgr
5. CVS

Recuperação de componentes

1. IDE/Web
2. GSRC
3. RetrieveAndEdit
4. AssetMgr
5. CVS
6. Sistema de arquivos

Pesquisa de Componentes

1. IDE/Web
2. GSRC
3. RepositorySearch
4. SearchEngine
5. RASModel
6. RASProfiles

Verificação de um componente para saber se ele satisfaz as regras de evolução (Utilização de um *plug-in*)

1. IDE/Web
2. GSRC
3. EvolutionChecker (ferramenta)

Adição de uma ferramenta

1. GSRC

Remoção de uma ferramenta

1. GSRC

B.1.4 Visão estruturada

A Figura B.1 mostra como estão relacionados os diversos componentes do sistema. Os componentes estão caracterizados de acordo com seu papel no funcionamento do sistema (ver legenda da figura). O repositório de componentes mostrado na figura é o Rigel [62].

B.1.5 Mapeamento de funcionalidades e componentes

A Tabela B.2 mostra quais componentes estão relacionados a uma determinada funcionalidade. Isso é importante para definir quais componentes poderão ser alterados caso uma funcionalidade seja alterada.

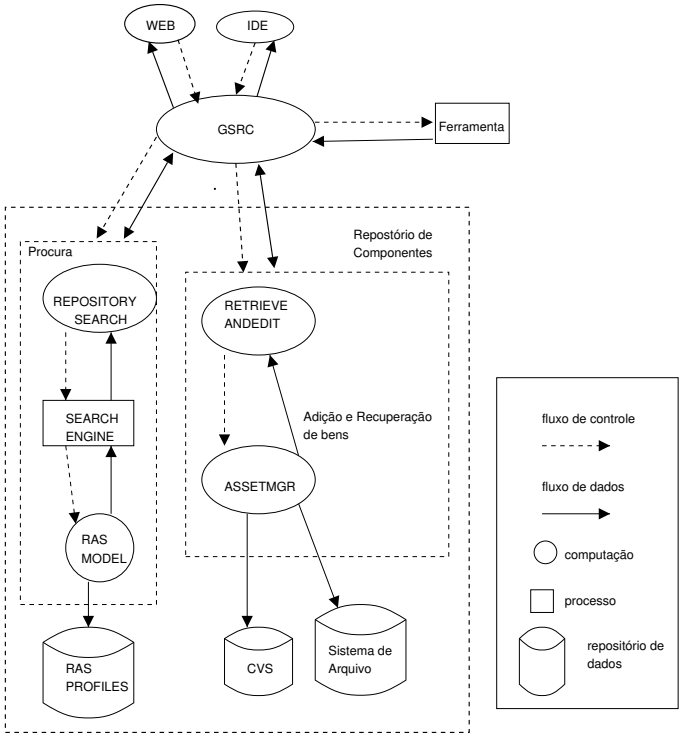

Figura B.1: Visão estruturada da arquitetura do sistema

B.1.6 Identificação dos estilos

O estilo adotado foi o **estilo em camadas**, como é mostrado na Figura 10A por facilitar a alteração dos componentes gerando um pequeno impacto nos componentes de outras camadas. A Figura 10B mostra uma visão geral da arquitetura.

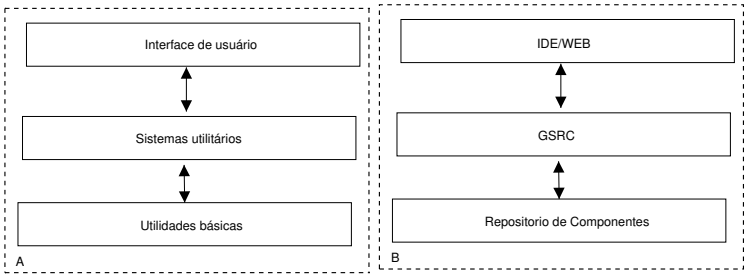

Figura B.2: Na Figura 10A, temos que o estilo arquitetural usado pelo sistema. A Figura 10B, mostra um visão de alto nível do sistema todo.

Tabela B.2: Mapeamento entre funcionalidades e componentes

Funcionalidades	Componentes
Inserção de componentes	IDE, Web, GSRC, RetrieveAndEdit AssetMgr, CVS, FileSystem
Busca de componentes	IDE, Web, GSRC, RepositorySearch SearchEngine, RASModel, RASProfile
Recuperação de componentes	IDE, Web, GSRC RetrieveAndEdit, AssetMgr, FileSystem, CVS
Usar ferramenta	IDE, Web, GSRC, EvolutionChecker
Adicionar ferramenta	GSRC
Remover ferramenta	GSRC

B.1.7 Cenários

Antes de descrevermos os cenários, é interessante observar que os cenários que possuem funcionalidades muito parecidas foram agrupados.

Como esta avaliação está sendo feita sob o ponto de vista da evolutibilidade, cenários que não exigem alterações na arquitetura são considerados *cenários diretos*. *Cenários indiretos* são os que demandam alterações para serem realizados completamente.

A Tabela B.3 mostra os cenários e o custo estimado da alteração relativa ao cenário.

Tabela B.3: Custo de modificação de cada cenário

Cenário	**Alterações na arquitetura**
Adição/Remoção de componentes	-
Recuperação de componentes	-
Pesquisa de componentes	-
Utilização de uma ferramenta	-
Adição de uma ferramenta	É necessário criar um subdiretório do diretório plugin e criar um arquivo de descrição do plugin **Custo de modificação: baixo**
Remoção de uma ferramenta	Remove-se o diretório onde a ferramenta está contido. **Custo de modificação: baixo**
Alteração do modelo RAS para um outro modelo de metadados de componentes	Se o usuário desejar utilizar outro tipo de metadados ao invés de estender o RAS, ele precisaria trocar os esquemas do *RAS Profile* além de pequenas mudanças na manipulação dos dados do *RAS Model* **Custo de modificação: médio**
Alteração do repositório de componentes ou de IDE	Seria necessário substituir a ferramenta responsável pela integração entre IDE e repositório de componentes e seus conectores. **Custo de modificação: alto**

B.1.8 Conclusões

A arquitetura do sistema mostrou uma boa modificabilidade. A criação de uma camada de abstração entre repositório de componentes e IDEs contribui para desacoplar os componentes e aumentar a modificabilidade. Todavia, a performance do sistema pode piorar com essa indireção a mais.

Apêndice C

Questionário de avaliação do CosmosChecker

O questionário abaixo foi usado no estudo de caso do sistema bancário (seção 4.2). Ele foi baseado no questionário de Tomita [63] e almeja avaliar a ferramenta de forma qualitativa.

C.1 Questionário

1. Você tem sugestões de melhoria para problemas de usabilidade? (ou seja, pontos na interface com o usuário que são difíceis de entender ou confusos para se utilizar?)

2. Você tem sugestões de melhoria para as funcionalidades? Faltou alguma funcionalidade importante? Qual?

3. Ocorreram erros (*bugs*) no uso da ferramenta? Por favor, descreva-os para que possamos corrigi-los:

Bibliografia

[1] Ant. [http://ant.apache.org/] Acessado em Janeiro de 2007.

[2] Beautyj. [http://beautyj.berlios.de/] Acessado em Janeiro de 2007.

[3] Componentsource. [http://www.componentsource.com] Acessado em Janeiro de 2007.

[4] Digital assets manager (cit). [http://www.digitalassetsmanager.com.br] Acessado em Janeiro de 2007.

[5] Flashline. [http://www.flashline.com] Acessado em Janeiro de 2007.

[6] Groovy. [http://groovy.codehaus.org/] Acessado em Janeiro de 2007.

[7] Java pet store. [http://java.sun.com/developer/releases/petstore/] Acessado em dezembro de 2006.

[8] Java platform - standard edition 6 - api specification. [http://java.sun.com/javase/6/docs/api/index.html] Acessado em Janeiro de 2007.

[9] Java platform, enterprise edition. [java.sun.com/javaee/] Acessado em Janeiro de 2007.

[10] *Java Swing*. [http://java.sun.com/docs/books/tutorial/uiswing/] Acessado em Janeiro de 2007.

[11] Jboss drools. [http://www.jboss.com/products/rules] Acessado em Janeiro de 2007.

[12] Logiclibrary. [http://www.logiclibrary.com] Acessado em Janeiro de 2007.

[13] .net framework. [msdn.microsoft.com/netframework/] Acessado em Janeiro de 2007.

[14] Python. [http://www.python.org/] Acessado em Janeiro de 2007.

[15] Subversion. [http://subversion.tigris.org/] Acessado em Dezembro de 2006.

[16] Xdoclet. [http://xdoclet.sourceforge.net/xdoclet/index.html] Acessado em Janeiro de 2007.

[17] Alexandre Alvaro, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. A component quality model for component quality assurance. In *Workshop de Desenvolvimento Baseado em Componentes*, 2005.

[18] Ali Arsanjani, Hussein Zedan, and James Alpigni. Externalizing component manners to achieve greater maintainability through a highly re-configurable architectural style. *International Conference on Software Maintenance*, 2002.

[19] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute at Carnegie-Mellon University, April 2000.

[20] Kleber R. Bacili and Marcilio Oliveira. Digitalassets manager,: sharing and managing software development assets. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 700–701, New York, NY, USA, 2006. ACM Press.

[21] Greg J. Badros. Javaml: a markup language for java source code. In *Computer Networks*, 2000.

[22] Stefan Van Baelen, David Urting, Werner Van Belle, Viviane Jonckers, Tom Holvoet, Yolande Berbers, and Karel De Vlaminck. Toward a unified terminology for component-based development. In *Proceedings of 14th European Conference on Object-Oriented Programming (ECOOP)*, June 2000.

[23] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[24] Gerd Beneken, Ulrike Hammerschall, Jan Jurjens, Bernhard Humpe, Maurce Schoenmakers, Manfred Broy, and Alexander Pretschner. Componentware, state of the art 2003. In *Understanding Components Workshop of the CUE Initiative*, 2003.

[25] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM Press.

[26] Regina M. M. Braga, Marta Mattoso, and Cláudia M. L. Werner. The use of mediation and ontology technologies for software component information retrieval. In *SSR '01: Proceedings of the 2001 symposium on Software reusability*, pages 19–28, New York, NY, USA, 2001. ACM Press.

[27] John Cheesman and John Daniels. *UML Components*. Addison-Wesley, 2000.

[28] David Cooper, Benjamin Khoo, Brian R. von Konsky, and Michael Robey. Java implementation verification using reverse engineering. In *ACSC '04: Proceedings of the 27th Australasian conference on Computer science*, pages 203–211, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[29] Ivica Crnkovic and Magnus Larsson. Challenges of component-based development. *Journal of Systems and Software*, 61(3):201–212, 2002.

[30] Patrick Henrique da Silva Brito. Um método para modelagem de exceções em desenvolvimento baseado em componentes. Master's thesis, Instituto de Computação - UNICAMP, 2005.

[31] Eric M. Dashofy, André Van der Hoek, and Richard N. Taylor. A highly-extensible, xml-based architecture description language. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 103, Washington, DC, USA, 2001. IEEE Computer Society.

[32] Christiano de O. Braga, Marcus Felipe M. C. da Fontoura, Edward H. Haeusler, and Carlos José de Lucena. Formalizing oo frameworks and framework instantiation. In *I Workshop Brasileiro de Métodos Formais*, 1998.

[33] João P.F. de Oliveira, Michael Schuenck, and Glêdson Elias. Componentforge: Um framework arquitetural para desenvolvimento distribuição baseado em componentes. In *Workshop de Desenvolvimento Baseado em Componentes - Recife*, 2006.

[34] Gledson Elias, Michael Schuenck, Yuri Negocio, Jr Jorge Dias, and Sindolfo Miranda Filho. X-arm: an asset representation model for component repository systems. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1690–1694, New York, NY, USA, 2006. ACM Press.

[35] C. L. Forgy. Rete: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[36] Eclipse Foundation. Eclipse ide. [http://www.eclipse.org] Acessado em Janeiro de 2007.

[37] W.B. Frakes and Kyo Kang. Software reuse research: status and future. In *Software Engineering, IEEE Transactions on*, 2005.

[38] Ernest Friedman-Hill. Jess. [http://www.jessrules.com/jess/] Acessado em Janeiro de 2007.

[39] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47 – 68. Cambridge University Press, 2000.

[40] Leonel A. Gayard, Paulo A.C. Guerra, Ana E.C. Lobo, and Cecilia M.F. Rubira. Automated deployment of component architectures with versioned components. In *International Workshop on Component-Oriented Programming*, 2006.

[41] Rosario Girardi and Alisson Neres Lindoso. An ontology-based knowledge base for the representation and reuse of software patterns. *SIGSOFT Softw. Eng. Notes*, 31(1):1–6, 2006.

[42] Marco Hunsicker and Steve Heyns. Jalopy. [http://jalopy.sourceforge.net/] Acessado em Janeiro de 2007.

[43] Moacir Silva Jnior. Cosmos - um modelo de estruturação de componentes para sistemas orientados a objetos. Master's thesis, Instituto de Computação - UNICAMP, 2003.

[44] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of object-oriented programming*, 1988.

[45] R. Kazman, L. Bass, G. Abowd, and M. Webb. Saam: A method for analysing the properties of the software architectures. *Proc. International Conference on Software Engineering*, 1994.

[46] Gerald Kotonya and John Hutchinson. Analysing the impact of change in cots-based systems. In *COTS-Based Software Systems*, 2005.

[47] M. M. Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag.

[48] Emanuela P. Lins and Ulrik P. Schultz. Supporting transparent evolution of component interfaces. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1629–1630, New York, NY, USA, 2006. ACM Press.

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

[50] C. Lung, S. Bot, K. Kaleichelvan, and R. Kazman. An approach to software architecture analysis for evolution and reusability. *Proc. CASCON'97*, 1997.

[51] Tiago Moronte. Uma infra-estrutura de software para construï¿½o de arquiteturas de software baseadas em componente. Master's thesis, Instituto de Computação - UNICAMP, 2007. a ser defendida.

[52] Eugen C. Nistor, Justin R. Erenkrantz, Scott A. Hendrickson, and André van der Hoek. Archevol: versioning architectural-implementation relationships. In *SCM '05: Proceedings of the 12th international workshop on Software configuration management*, pages 99–111, New York, NY, USA, 2005. ACM Press.

[53] Marcílio Oliveira, Islene Garcia, and Aminadab Nunes. Rccs: uma rede de compartilhamento de componentes de software. In *Simpósio Brasileiro de Redes de Computadores*, 2005.

[54] OMG. Reusable asset specification, November 2005. [http://www.omg.com] Acessado em Janeiro de 2007.

[55] Helder Pinho. Rigel - um repositório com suporte para desenvolvimento baseado em componentes. Master's thesis, Instituto de Computação - UNICAMP, 2005.

[56] Roshanak Roshandel, André Van Der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Trans. Softw. Eng. Methodol.*, 13(2):240–276, 2004.

[57] Dennis Sosnoski. *Java programming dynamics, Part 2: Introducing reflection*, 2003. [http://www-128.ibm.com/developerworks/library/j-dyn0603/] Acessado em Dezembro de 2006.

[58] Vijayan Sugumaran and Veda C. Storey. A semantic-based approach to component retrieval. *SIGMIS Database*, 34(3):8–24, 2003.

[59] Clemens Szyperski. *Component Software*. Addison-Wesley, 2002.

[60] NaiyNaiyana Tansalarak, Kajal T. Claypoolana Tansalarak, and Kajal T. Claypool. Xcm:a component ontology. *Workshop on Ontologies as Software Engineering Artifacts*, 2004.

[61] Richard Taylor and David Redmiles. Archstudio. *SIGSOFT Softw. Eng. Notes*, 25(1):97, 2000.

[62] Leonardo Tizzei, Helder Pinho, Paulo Guerra, and Cecília Rubira. Um repositório de componente com suporte a evolução centrada na arquitetura de software. *Workshop de Desenvolvimento Baseado em Componente - Juiz de Fora - Brasil*, 2005.

[63] Rodrigo Teruo Tomita. Bellatrix: Um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes. Master's thesis, Institute of Computing (UNICAMP), 2006.

[64] Haining Yao and Letha Etzkorn. Towards a semantic-based approach for software reusable component classification and retrieval. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 110–115, New York, NY, USA, 2004. ACM Press.