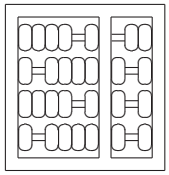

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

EvolutionChecker: Uma Ferramenta para apoiar a evolução de componentes centrada na arquitetura de software

Leonardo P. Tizzei *Cecília M. F. Rubira*

Technical Report - IC-06-667 - Relatório Técnico

November - 2006 - Novembro

EvolutionChecker: Uma Ferramenta para apoiar a evolução de componentes centrada na arquitetura de software

Leonardo P. Tizzei* Cecília M. F. Rubira†

Abstract. *A reutilização de componentes promovida pelo desenvolvimento baseado em componentes cria dependências entre sistemas que podem ser minimizadas através de modelos de evolução. As ferramentas que automatizam o versionamento mais conhecidas consideram apenas o versionamento de arquivos. O EvolutionChecker automatiza um modelo de evolução que é centrado na arquitetura de software minimizando as dependências entre os sistemas. Ele utiliza especificações de componentes um formato padrão de XML para alertar o usuário de incompatibilidades, como a remoção de um interface provida de um componente, causadas pela evolução. O EvolutionChecker também foi desenvolvido almejando ser modificável, propiciando uma fácil alteração no modelo de evolução.*

1 Introdução

O desenvolvimento baseado em componentes (DBC) incentiva a reutilização de bens[1] com o objetivo de reduzir custos e tempo de entrega de sistemas de software. Segundo o RAS[16], bens são a solução para um problema dado um contexto. Interfaces, especificações e implementações de componentes e configurações arquiteturais são exemplos de bens reutilizáveis. Através da reutilização de um bem em diversos sistemas, seu custo de desenvolvimento é amortizado.

Contudo, devido a reutilização de um bem em diversos sistemas, a evolução destes bens pode gerar incompatibilidades entre suas diversas versões. A remoção de uma interface provida de um componente é um exemplo de incompatiblidade. Com o objetivo de resolver essas incompatibilidades, alguns trabalhos definem regras de evolução de um componente e modelos de versionamento, como Lobo[15] e Hoek[19]. O modelo de Lobo apóia a evolução centrada na arquitetura de software através de um modelo de versionamento baseado no impacto que determinadas operações sobre os componentes podem ter. Essas operações são limitadas por regras de evolução. Um exemplo de regra de evolução é a probição de remover uma interface provida de um componente.

Desenvolvedores de sistemas baseados em componentes podem possuir diferentes modelos de versionamento e regras para evoluir os bens. O impacto causado pela adição de uma interface em um componente pode ter peso diferente em dois modelos de versionamento distintos. O que pode e o que não pode ser modificado em um componente também pode diferir em dois modelos de evolução.

*apoiado por CNPq/Brasil, processo 131817/2005-1

†parcialmente apoiada por CNPq/Brasil, processo 351592/97-0 e apoiada por FAPESP/Brasil, processo 2004/10663-8

[1] *asset*, em inglês

Outro problema é que a aplicação das regras de evolução e do modelo de versionamento é feita de forma manual, sujeita a erros de interpretação das regras. Algumas ferramentas de gerência de configuração permitem o versionamento automático [13][18]. Entretanto, elas versionam apenas arquivos, sem distinção se eles são interfaces, componentes ou manuais de uso.

Este trabalho descreve o *EvolutionChecker*, uma ferramenta para automatizar o versionamento de componentes de acordo o modelo proposto por Lobo[15]. Assim, especificações e implementações de componentes podem ser automaticamente versionados de acordo com o modelo de evolução. O *EvolutionChecker* usa um motor de inferência para facilitar a alteração das regras de evolução, tornando fácil a adaptação para outro modelo de evolução ou a simples alteração do modelo corrente.

Este trabalho será apresentado da seguinte forma: na seção 2 apresentamos os fundamentos básicos para entendimento trabalho. Em seguida, mostramos parte da especificação do Evolution-Checker na seção 3 e de sua implementação na seção 4. Na seção 5 detalhamos um estudo de caso e, por fim, apresentamos nossas conclusões na seção 6.

2 Fundamentos teóricos

2.1 Desenvolvimento Baseado em Componentes

O DBC é uma técnica de desenvolvimento de software que se baseia na construção rápida de sistemas a partir de componentes pré-fabricados [5]. A divisão modular promovida pelo DBC propicia um aumento da adaptabilidade, escalabilidade e manutenibilidade do sistema, em relação a uma construção monolítica [17]. 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. 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 [6]. Uma **interface** identifica um ponto de interação entre um componente e o seu ambiente [12]. A Figura 1 mostra um exemplo de um componente e a notação utilizada em UML. Nessa figura, o `ComponenteA` possui duas interfaces: uma interface provida chamada `IProv` e uma interface requerida chamada `IReq`.

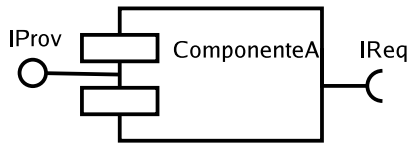

Figura 1: Exemplo de um componente.

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 arquiteturas 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 [6].

2.2 Arquitetura de Software

Sistemas de software complexos são decompostos em subsistemas 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 [7, 12]. 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 [12]. 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**.

2.3 O 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 especificação de componente(fato) não removeu uma interface provida(regra). 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. Isso, segundo Arsanjani[4], 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[7].

O motor de inferência Drools é um projeto cujos códigos-fonte estão disponíveis e ele está associado ao JBoss[14]. Ele é baseado em linguagens declarativas. As regras são descritas em

blocos *if/then* e podem ou não haver dependência entre regras. É possível também estabelecer uma sequência na qual as regras serão executadas.

A Figura 2 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 comparados no `Parelhador de Padrões`, que pode ou não estar ligado a alguma `Agenda` que determinada a ordem de execução das regras.

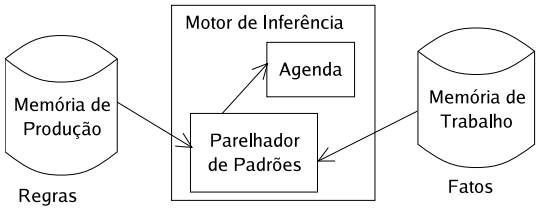

Figura 2: Visão geral do Drools

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. As regras podem ser implementadas em três linguagens de programação diferentes: Java[2], Groovy[1] e Python[3].

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. Abaixo colocamos um exemplo de uma regra escrita usando uma linguagem própria do Drools:

```
1 #arquivo de regras
2 .
3 .
4 .
5 rule "target platform"
6  when
7   oldMetadata: String()
8   newMetadata: String()
9   eval ( XPathHelper.checkProfile( newMetadata, "AbstractComponent" ) )
10  eval ( XPathHelper.checkProfile( oldMetadata, "AbstractComponent" ) )
11  eval ( XPathHelper.isNewMetadata( newMetadata ) )
12 then
13  System.out.println( "fire Target Platform rule..." );
14  String changeImpactFile = "changeImpact.xml";
15  TargetPlatform tp = new TargetPlatform();
16  ImpactLevel impact =
17     tp.measureImpact( oldMetadata, newMetadata, changeImpactFile);
18  RulesManager.addImpact( impact );
19 end
20 .
21 .
22 .
```

Essa amostra do arquivo de regras, verifica se a plataforma alvo (*target platform*) foi alterada ou não. A palavra reservada **rule**(linha 5) precede o nome a regra. As condições para que a regra seja executada são listadas entre as linhas 7 e 11. Nas linhas 7 e 8 restrigimos o tipo de `oldMetadata` e `newMetadata` a String. Nas linhas 9 e 10 o método estático `checkProfile` é chamado para verificar se os *Profiles* dos metadados são do tipo *AbstractComponent*(ver seção 2.5). Por fim, na linha 11, verifica-se se o *newMetadata* representa o metadado do bem alterado. Se todas as condições forem satisfeitas, o comando **then** marca o começo da execução da regra, que é bastante similar a um programa em Java.

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[10] foi criado para dar suporte a edição de regras. Esse ambiente facilita a utilização pelo usuário pois integra a edição e execução das regras em um único lugar.

A Figura 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.

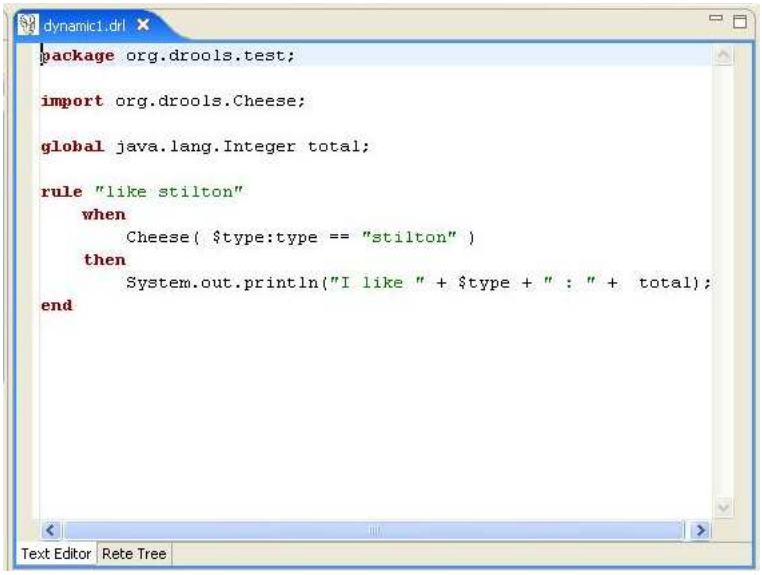

Figura 3: Editor de regras do Drools (fonte:[14])

Ao contrário do *Jess*[11], outro motor de inferência para Java que também usa o algoritmo Rete[9] para processar as regras, 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.

2.4 Modelo de evolução

O modelo de evolução proposto por Lobo[15] 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.

O modelo de versionamento é baseado em trabalhos prévios de gerência de versão de objetos complexos para produtos de engenharia[15]. As regras de evolução são fundamentadas em operações 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 operação existe um grau de impacto resultante que pode ser alto, médio ou baixo. Por exemplo, a operação de 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 2.1). 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.

2.5 Reusable Asset Specification

A especificação RAS(*Reusable Asset Specification*) é um padrão para descrever bens criado pela OMG. De acordo com o RAS, bens são a solução para um problema dado um contexto[16]. A definição é propositalmente abrangente, permitindo que extensões sejam feitas de acordo com a necessidade do usuário. O RAS utiliza a tecnologia XML para armazenar seus dados, o que facilita sua extensão uma vez que um arquivo XML é facilmente extensível.

A Figura 4 mostra a relação entre Bem, Componente e Artefato. Um Componente é um Bem, mas o contrário nem sempre é verdade. Todo Bem possui um ou mais artefatos.

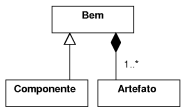

Figura 4: Relação entre Bem, Componente e Artefato

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. Para dar suporte ao modelo de versionamento descrito na seção 2.4, criamos quatro extensões ou *Profiles*. A Figura 2.5 mostra os *Profiles* criados e a relação entre *Profiles* e *Core RAS*. Os *Profiles* criados são: *Interface Definition*, *Abstract Component* , *Concrete Component* e *Configuration*. Mais detalhes sobre as extensões estão no Apêndice A.

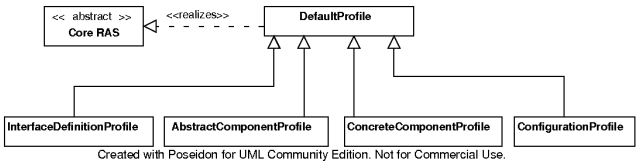

Figura 5: Novos *Profiles* criados

3 Especificação do EvolutionChecker

3.1 Definição dos Requisitos

- versionar bens automaticamente
- apoiar um modelo de evolução
- deve ser fácil alterar o modelo de evolução (modificabilidade)
- alterar as regras deve ser fácil para não-*experts* no assunto (usabilidade)

3.1.1 Modelo Conceitual do Negócio

O *Business Concept Model* (BCM) é um modelo cujo objetivo é identificar os conceitos envolvidos e o relacionamento entre eles. Nós usamos um modelo de classes para criar o BCM, entretanto, ontologias e outras notações também poderiam ser utilizadas.

A Figura 6 mostra que o motor de inferência (`Drools`) executa as (`Regras de Evolução`), que por sua vez seguem um (`Modelo de Versionamento`). As regras de evolução de Lobo[15] são aplicadas em `componentes abstratos` e `componentes concretos`. Esses tipos de componentes são, por sua vez, um `Bem`. Outros tipos de bens são `Definições de Interface` e `Configurações` e todos esses bens são descritos através de *Profiles* que estendem o `RAS`.

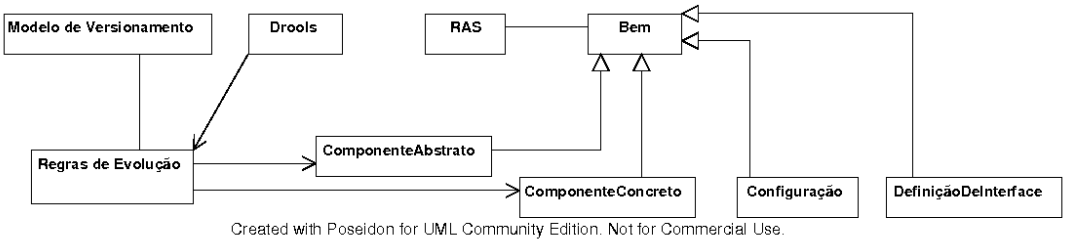

Figura 6: *Business Concept Model* do EvolutionChecker

Através da utilização de um motor de inferência aumentamos a modificabilidade da ferramenta, que é um atributo de qualidade da ferramenta. Não existe um tipo específico de regras que deve ser usado pelo motor de inferência, contudo neste caso utilizamos regras de evolução.

As regras de evolução seguem um modelo de versionamento. Este modelo define o impacto que certas operações causam em em componentes abstratos e concretos. Por exemplo, qual é o impacto causado pela adição de uma interface provida a um componente. Este modelo mantém a compatibilidade entre as versões pois ele limita as operações que podem ser realizadas sobre determinados elementos do componente.

Contudo, para que a ferramenta possa aplicar as regras de evolução é necessário que ela tenha acesso às meta-informações sobre os bens. Essas meta-informações estão especificadas através de extensões do RAS, mais conhecidas como *Profiles*. Foram criados quatro *Profiles* com o objetivo de dar suporte ao modelo de versionamento (ver seção 2.5).

3.1.2 Casos de Uso

Os casos de uso são importantes para a melhor compreensão do sistema. Além disso eles ajudam a definir quais serão as interfaces de sistema.

A seguir listamos os casos de uso bem como suas pré e pós-condições. A Figura 7 mostra todos os casos de uso e os atores de cada um deles.

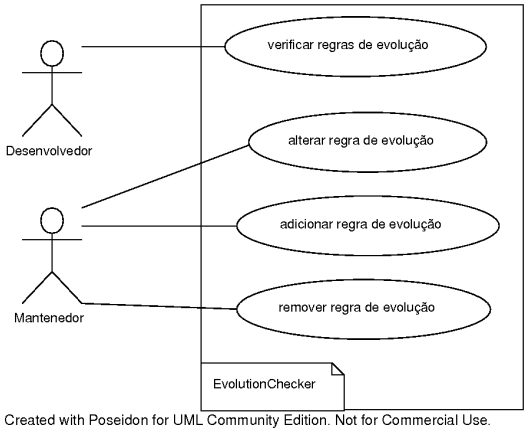

Figura 7: Casos de Uso de Verificador de Regras de Evolução

Verificar regras de evolução
Pré-condição: os metadados dos bens são válidos (satisfazem as restrições do XML e do XSD que eles importam)
Pós-condição: o novo número de versão do bem é definido

1. O desenvolvedor retira um bem do repositório e realiza alterações nele. Depois cria o metadado do bem e chama o verificador de regras de evolução
2. O verificador de regras de evolução define a nova versão de bem, de acordo com um modelo de versionamento, e retorna o valor para o usuário.

Adicionar regra de evolução
Pré-condição:
Pós-condição: regra de evolução criada

1. o mantenedor executa o programa para abrir o editor de regras.
2. o sistema abre o editor de regras.
3. o mantenedor escreve uma nova regra e salva as alterações.

Remover regras de evolução
Pré-condição: pelo menos uma regra já deve ter sido criada
Pós-condição: o arquivo de regras possui uma regra a menos

1. o mantenedor executa o programa para abrir o editor de regras.

2. o sistema abre o editor de regras.

3. o mantenedor apaga uma nova regra e salva as alterações.

Editar regras de evolução
Pré-condição: pelo menos uma regra já deve ter sido criada
Pós-condição: o número de regras antes e depois da alteração deve ser o mesmo

1. o mantenedor executa o programa para abrir o editor de regras.

2. o sistema abre o editor de regras.

3. o mantenedor altera uma regra e salva as alterações.

3.2 Arquitetura de componentes inicial

A Figura 8 mostra uma arquitetura em duas camadas: a camada de sistema e a camada de negócio. Como o Drools possui um *plug-in* para Eclipse que edita regras, reutilizamos esse *plug-in* para satisfazer os requisitos funcionais de adicionar, editar e remover regras. Esse *plug-in* está representado por `EclipseDroolsPlugin`. O componente `EvolutionCheckerMgr` é responsável por gerenciar os meta-dados e chamar o componente `DroolsMgr`. `DroolsMgr` é o motor de inferência que executa as regras de evolução. É importante notar que o componente `EclipseDroolsPlugin` não está diretamente ligado a nenhum outro componente. Isso ocorre porque esse componente é responsável somente por operações no arquivo de regras, que será posteriormente usado como parâmetro pelo componente `EvolutionCheckerMgr` para executar as regras.

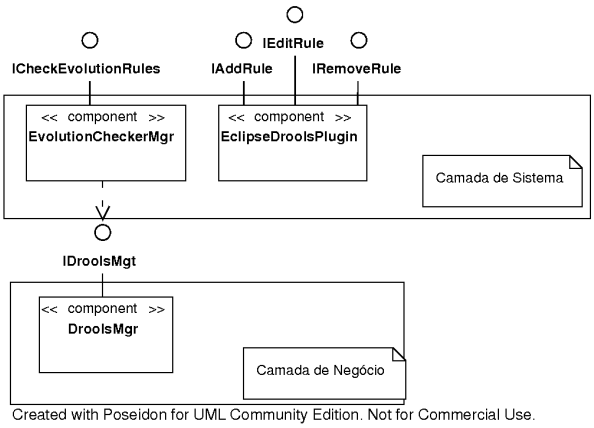

Figura 8: Arquitetura de Componentes Inicial

3.3 Cenário de Uso

A seguir mostraremos um cenário de uso do EvolutionChecker. O EvolutionChecker é uma ferramenta concebida para complementar os serviços oferecidos pelo repositório de componentes. Sendo assim, seu uso está intimamente ligado ao do repositório. Suponha que um componente abstrato obsoleto `CompA` está no repositório e precisa ser alterado. O desenvolvedor recupera o componente do repositório e grava uma cópia em sua área de trabalho. Um número de versão foi atribuído a esse componente assim que ele foi depositado no repositório para permitir a coexistência de diversas versões desse mesmo componente. Na Figura 9, vemos o componente abstrato `CompA` com uma interface e seu número de versão.

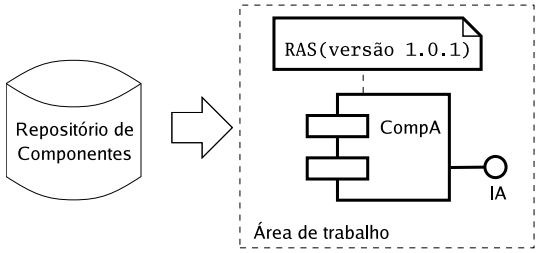

Figura 9: Componente Abstrato retirado do repositório

Após recuperar o componente do repositório, o desenvolvedor resolve adicionar uma interface provida **IB** ao componente. A Figura 10 mostra a adição da nova interface. O número de versão deixa de ser o mesmo, uma vez que o componente também mudou.

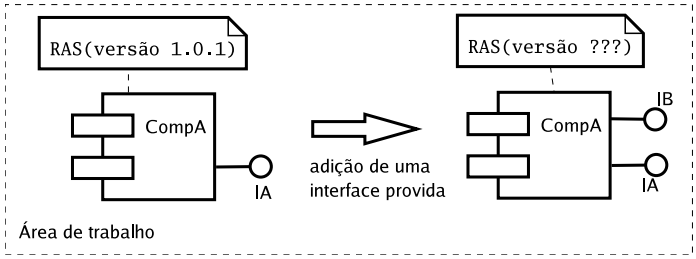

Figura 10: Adição da interface IB ao componente abstrato

Por fim, antes de depositar o novo componente no repositório de componentes, o desenvolvedor decide usar o EvolutionChecker para manter a consistência com modelo de evolução usado pela empresa. Como o desenvolvedor adicionou uma interface provida, o EvolutionChecker retorna que o grau impacto dessa operação é de nível médio e, portanto, o novo número de versão do componente é alterado. A Figura 11 mostra o `EvolutionChecker` e seus três parâmetros de entrada: o metadado do componente antes da alteração, o metadado do componente atual e o arquivo de regras. O resultado é a definição do novo número de versão do componente abstrato, que nesse caso é 1.1.0.

Com o novo número de versão, o desenvolvedor pode preencher o metadado do componente abstrato alterado e reinserir no repositório de componentes. A Figura 12 mostra essa operação que representa o fim deste cenário de uso. Este é o principal cenário de uso do EvolutionChecker. Outros cenários são possíveis como, por exemplo, verificar após a evolução de um bem ele se mantém

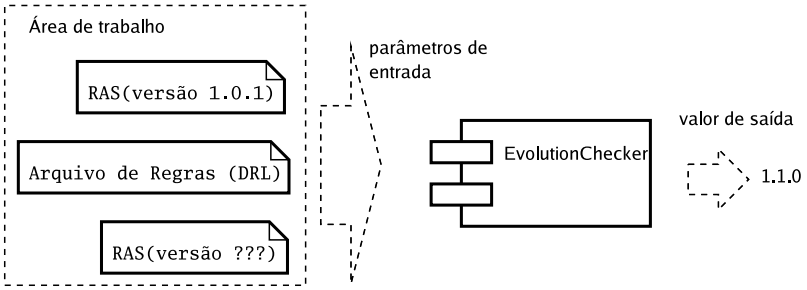

Figura 11: Componente Abstrato recebe um novo número de versão

compatível dentro dos sistemas que ele compõem. A seção 5 mostra esse cenário de uso com mais detalhes.

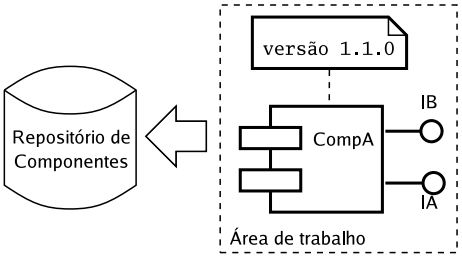

Figura 12: Adição Componente Abstrato novamente no repositório de componentes, com um metadado atualizado

4 Implementação do EvolutionChecker

4.1 Diagrama de classes

Os componentes que compõem o *EvolutionChecker* e *DroolsMgr* seguem o modelo COSMOS[8]. A Figura 13 mostra o diagrama de classes do *EvolutionChecker*.

Nela podemos ver algumas características do COSMOS como a separação entre os pacotes de implementação (`impl`) e especificação (`spec`). As classes de infra-estrutura do COSMOS, `ComponentFactory`, `Manager`, `Facade` e a interface `IManager`, também estão presentes. A classe `Control` apenas prepara o componente para chamar as regras de evolução.

A Figura 14 mostra o diagrama de classes do componente *DroolsMgr*. Este componente que comunica-se diretamente com o motor de inferência Drools, como pode ser visto na dependência que o componente tem em relação a biblioteca `drools.jar`. A classe `DroolsManager` faz a preparação para chamar a biblioteca, ou seja, define qual é o arquivo de regras, quais são os fatos e quando as regras serão disparadas. A classe `ResultManager` é responsável por lidar com as respostas de cada regra. É esta classe que armazena e processa as respostas das regras. Trata-se de uma classe abstrata, que deve ser estendida de acordo com os dados retornados pelas regras e como eles serão processados.

Este componente ainda possui um pacote `droolsmgr.datatype` para guardar os tipos de dados que são utilizados por ele, possivelmente, por outros componentes. `Operation` e `ImpactLevel` são

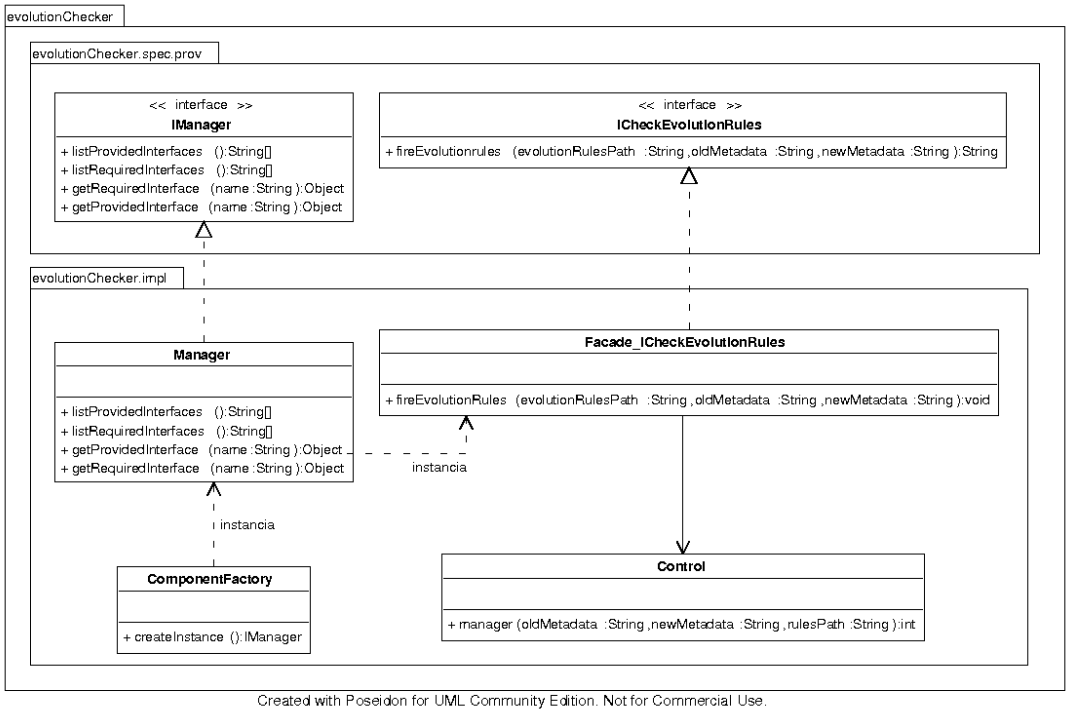

Figura 13: Diagrama de classes do EvolutionChecker

tipos enumerados. `Operation` é o valor retornado pelas regras que depois será processado para descobrir qual é o impacto causado pela operação. Esse impacto é representado através de outro tipo enumerado, o `ImpactLevel`.

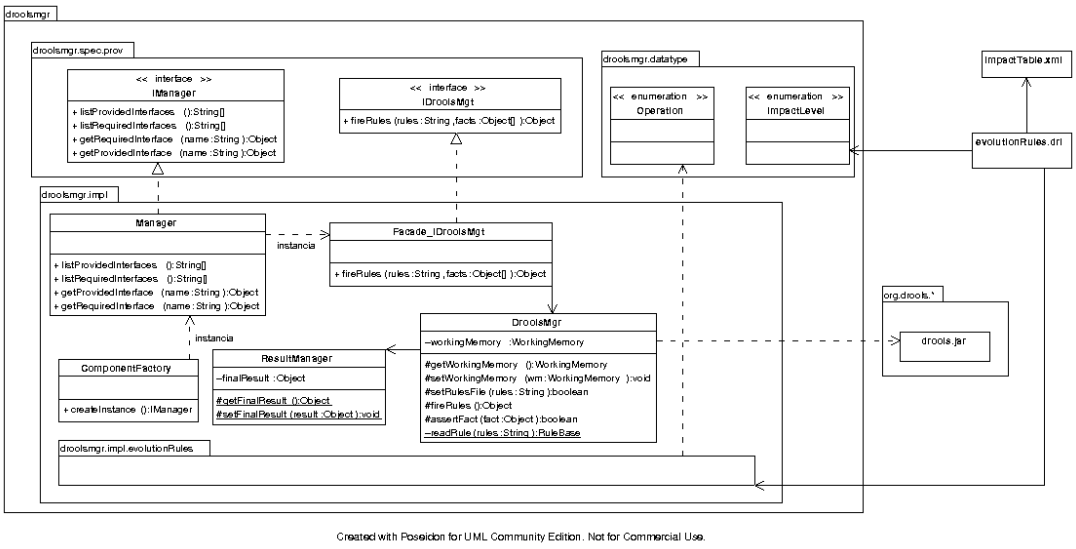

Figura 14: Diagrama de classes do DroolsMgr

O arquivo `evolutionRules.drl`(ver exemplo deste arquivo na seção 2.3) contém as regras de evolução. Como a maioria das regras são complexas, existem classes de auxílio para executá-las. Para facilitar o entendimento do código, as regras foram colocadas todas em um sub-pacote chamado `droolsmgr.impl.evolutionRules`. Por motivos de clareza esse pacote aparece vazio na Figura 14, entretanto, aparece completo na Figura 15. Outro papel importante do arquivo de regras é que ele contém a referência para a tabela com na qual define-se o grau impacto que cada operação de uma determinada regra. A tabela de impacto está representada em um arquivo XML como mostra a listagem abaixo.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <evolutionRules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
3   <abstractComponent>
4     <setOfInterfaces id="1" modification="nothing" addition="medium" subtraction="
        nothing"/>
5     <typeOfProvidedInterface id="2" modification="nothing" addition="medium"
        subtraction="na"/>
6     <typeOfRequiredInterface id="3" modification="nothing" addition="high"
        subtraction="medium"/>
7     <maintenanceStateInterface id="4" modification="high" addition="na" subtraction=
        "na"/>
8     <setOfPorts id="5" modification="nothing" addition="medium" subtraction="nothing
        "/>
9     <setOfInterfacesofAPort id="6" modification="nothing" addition="medium"
        subtraction="nothing"/>
10    <maintenanceStateOfAPort id="7" modification="high" addition="na" subtraction="
        na"/>
11    <targetPlatform id="8" modification="high" addition="na" subtraction="na"/>
12    <synchronizationContract id="9" modification="high" addition="medium"
        subtraction="na"/>
13    <qosContract id="10" modification="medium" addition="medium" subtraction="na"/>
14  </abstractComponent>
15  <concreteComponent>
16    <versionIdOfSpec id="1" modification="rule8" addition="na" subtraction="na"/>
17    <internalDesign id="2" modification="medium" addition="na" subtraction="na"/>
18    <sourceCode id="3" modification="low" addition="na" subtraction="na"/>
19    <setOfSubComponents id="4" modification="high" addition="high" subtraction="low"
        />
20    <setOfExternalTools id="5" modification="high" addition="high" subtraction="low"
        />
21    <setOfTools id="6" modification="low" addition="low" subtraction="low"/>
22  </concreteComponent>
23 </evolutionRules>
```

Na listagem podemos ver que dois bens fazem parte diretamente do modelo de evolução: componente abstrato (`abstract component`, entre as linhas 3 e 14) e componente concreto (`concrete component`, entre as linhas 15 e 22). Interfaces e a configuração arquitetural aparecem indiretamente no modelo. As regras de evolução estão representadas nas folhas da árvore implícita gerada pelo XML. Cada regra possui quatro atributos: `id`, `modification`, `addition` e `subtraction`. Os `ids` são identificadores únicos de cada regra e os demais atributos correspondem às três operações. Os possíveis valores das operações estão representados no tipo enumerado `ImpactLevel`. São eles:

- ERROR(Erro), quando ocorre alguma situação não prevista pelo *EvolutionChecker*, como, por exemplo, não encontrar um elemento no metadado que deveria estar especificado.

- NA[2], a operação realizada no bem não foi prevista pelo modelo e, portanto, um novo bem precisa ser criado. Por exemplo, a remoção de um tipo de interface provida é NA.
- INSIGNIFICANT(insignificante), a operação realizada não é significativa sob o ponto de vista do versionamento. Corresponde ao valor *nothing* de uma operação.
- LOW(baixo), grau de impacto baixo.
- MEDIUM(médio), grau de impacto médio.
- HIGH(alto), grau de impacto alto.

Existe um caso especial no arquivo XML que é a regra `versionIdOfSpec`, na linha 16. O atributo `modification` possui como valor `rule8`. Como se trata de uma regra complexa, é preciso saber que tipo de modificação ocorreu para se determinar o grau de impacto da mudança. De acordo com Lobo[15], esse grau de impacto é dado pela regra 8. Ou seja, na prática essa é a única regra que não consulta a tabela para emitir o resultado do grau de impacto.

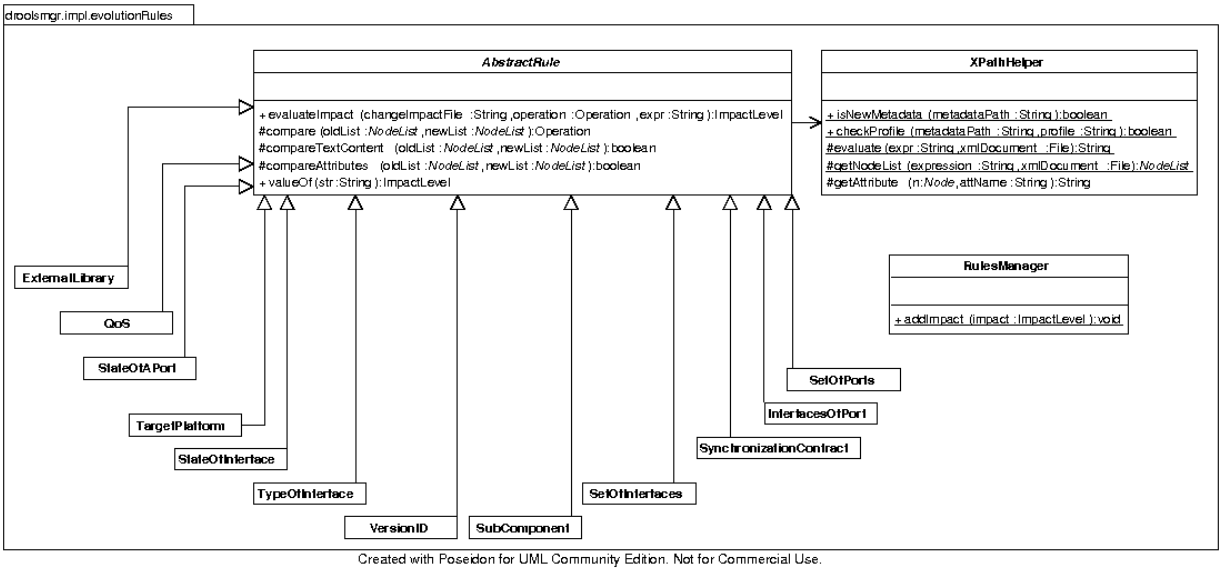

Figura 15: Diagrama de classes do Sub-pacote `droolsmgr.impl.evolutionRules`

A Figura 15 mostra o sub-pacote que contém as regras de evolução. Todas as regras de evolução estendem `AbstractRule`, pois ela possui métodos necessários a todas as regras. A classe `AbstractRule` associa-se com `XPathHelper`, uma classe que utiliza-se de uma linguagem de consultas em XML chamada XPath[20]. Um exemplo de busca que uma regra pode precisar é quantas interfaces providas têm um determinado componente.

As regras processam os metadados sobre os bens para verificar se alguma operação (adição, subtração ou modificação) foi realizada. A classe `RulesManager` estende `droolsmgr.impl.ResultManager` e armazena e processa os resultados obtidos pelas regras.

[2]sigla em inglês que significa *não disponível*

4.2 Cenário de execução

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

O componente EvolutionChecker é bastante simples e, por isso, não construímos um diagrama de sequência para descrevê-lo. Como o componente DroolsMgr é bastante complexo, nos concentramos nele. A Figura 16 mostra o diagrama de sequência do DroolsMgr. O diagrama começa com a classe `Facade_IDroolsMgt` que implementa a interface provida `IDroolsMgt`. A classe DroolsMgr 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). Neste exemplo, o arquivo de regras é o `EvolutionRules.drl` e os fatos são os metadados dos bens, 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 sequência 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).

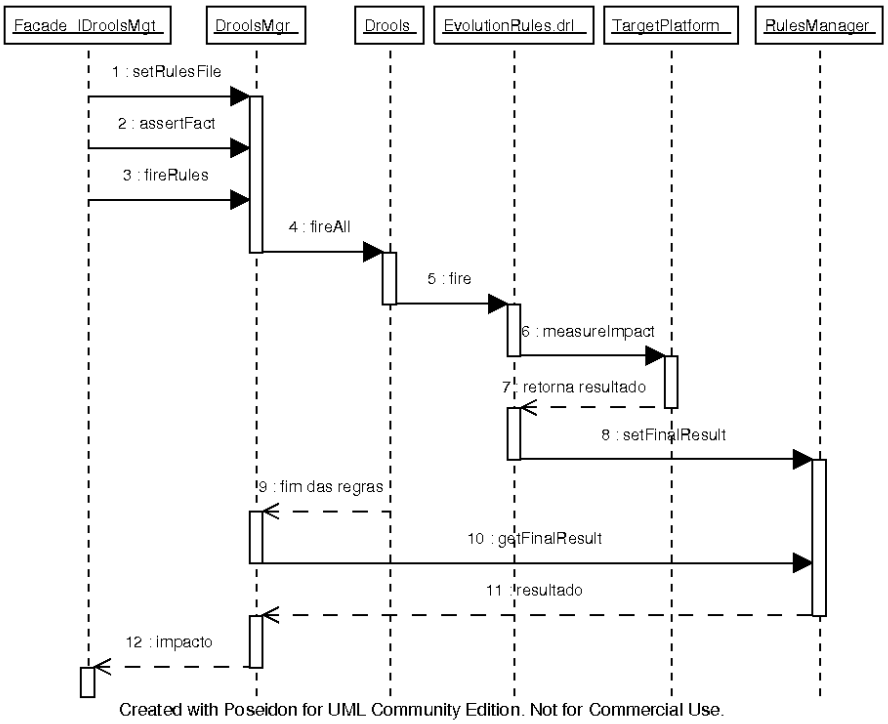

Figura 16: Diagrama de sequência do DroolsMgr

5 Estudo de Caso

Como já foi dito anteriormente, o modelo de evolução de Lobo[15] aumenta a substitubilidade de componentes. Como o EvolutionChecker automatiza parte deste modelo, ele deve permitir ao usuário saber se a alteração de um componente não quebrará alguma regra de evolução o que diminuiria sua substitubilidade. Realizamos um estudo de caso com o objetivo de analisar se o EvolutionChecker indicaria quando a compatibilidade de um componente fosse quebrada por um alteração.

Como componentes, utilizamos pacotes de uma biblioteca Java[2], mais especificamente o pacote `java.rmi.server`. Optamos por usar um pacote de uma biblioteca de Java porque ele é gratuito e possui uma boa documentação. Esse pacote possui nove interfaces, doze classes e seis exceções.

A Figura 17 mostra uma configuração arquitetural hipotética que utilizamos em nosso estudo de caso. Nesta figura o componente abstrato `JavaRMIServer`, que representa o pacote `java.rmi.server`, fornece duas interfaces providas, `LoaderHandler` e `ServerRef`. Essas interfaces são requeridas pelo componente hipotético `CompA`.

Por motivos de clareza, algumas simplificações foram feitas na figura: sete interfaces providas de `JavaRMIServer` foram omitidas e os conectores arquiteturais estão implícitos. A figura foi simplificada, todavia os metadados contêm todas as interfaces (ver apêndice B)

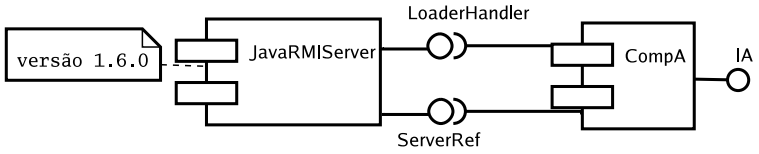

Figura 17: Arquitetura do sistema usando a versão Java S.E. 6

De acordo com a documentação de Java S.E. 6[2], a interface `LoaderHandler` é depreciada e, por isso, supomos que em alguma futura versão ela não será mais usada. Partindo dessa hipótese, um arquiteto de software pode querer saber se a migração para uma nova versão da Java que não possua as interfaces depreciadas trará incompatibilidades para o sistema. De acordo com o modelo de evolução de Lobo, remover uma interface provida de um componente quebra uma regra de evolução e exige a criação de um novo componente.

Dentro deste cenário hipotético, no qual o componente `JavaRMIServer` perderá a interface provida `LoaderHandler`, especificamos o componente `JavaRMIServer` com e sem a interface `LoaderHandler`. Assim, simulamos uma suposta alteração em sua especificação. Essa especificação foi feita utilizando o *profile* de Componente Abstrato pois, apesar de existir um implementação do `JavaRMIServer`, ou seja, um componente concreto, não existe um componente abstrato do `JavaRMIServer`. De acordo com o modelo de Lobo, todo componente concreto implementa um componente abstrato, que por sua vez é o responsável por mostrar o comportamento observável externamente.

Quando submetemos a especificação atual e a hipotética ao EvolutionChecker recebemos como resposta que o novo componente quebra uma regra de evolução ao remover um interface provida de um componente. A listagem abaixo mostra a saída do EvolutionChecker. Entre as linhas 7 e 9 aparece a regra `type of provided interface` que teve como grau de impacto `NA`, o que significa

que esta operação não é permitida. Na linha 44 temos o número de versão antigo do componente, que era `1.6.0` e na linha 45 temos o novo número de versão é `1.0.1`. De acordo com o modelo de versionamento seguido pelo EvolutionChecker, quando uma regra de evolução é quebrada e um novo componente é criado seu número de versão inicial é `1.0.1`. Além do número de versão, o EvolutionChecker também mostra qual operação foi realizada e o impacto daquela operação.

```
1 starting evolution checker...
2 Rules 'evolutionRules.drl' were successfully defined
3 >>fire type of required interface rule...
4 Operation performed: SUBTRACTION
5 Impact level: MEDIUM
6
7 >>fire type of provided interface rule...
8 Operation performed: SUBTRACTION
9 Impact level: NA
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: SUBTRACTION
21 Impact level: INSIGNIFICANT
22
23 >>fire maintenance state of a interface rule...
24 Operation performed: NOTHING
25 Impact level: INSIGNIFICANT
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 Some Evolution Rules has been broken. You must start a new versioning tree
44 The old version was 1.6.0
45 The new version is 1.0.1
```

A Figura 18 mostra esse cenário, no qual a nova versão de `JavaRMIServer` quebra uma regra de evolução e, por isso, o desenvolvedor é obrigado a criar um novo componente `JavaRMIServerB`. Uma possível solução seria a utilização de um terceiro componente `CompB` para implementar `LoaderHandler`

e suprir essa interface requerida de `CompA`.

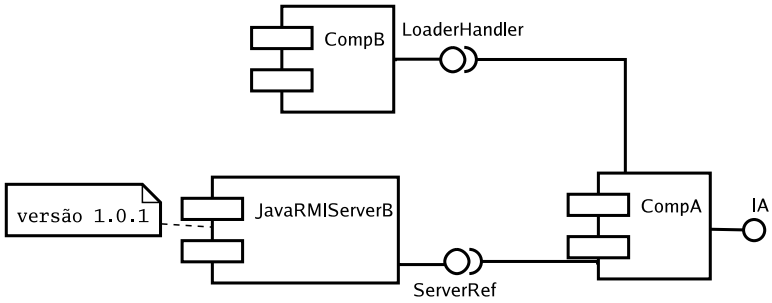

Figura 18: Arquitetura do sistema com uma versão hipotética de Java

6 Conclusões

O DBC incentiva a reutilização de bens para reduzir custos e tempo de entrega de sistemas baseados em componentes. Um bem que é utilizado em diversos sistemas e evolui pode provocar conflitos nesses sistemas. Para resolver esse problema, modelos de evolução definem regras para evoluir um componente procurando aumentar sua substitubilidade. Essas regras podem divergir de acordo com a empresa ou modelo de desenvolvimento.

Este trabalho apresentou uma ferramenta para apoiar a evolução de componentes de forma automatizada chamada EvolutionChecker. Esta ferramenta utiliza o modelo de evolução de Lobo[15] e implementa regras de evolução e o modelo de versionamento. O EvolutionChecker permite que o usuário obtenha componentes compatíveis com um modelo de evolução e, por isso, mais substituíveis.

Outra contribuição do EvolutionChecker é que ele foi desenvolvido de forma a facilitar a alteração das regras de evolução. Por isso, empresas com modelos de evolução distintos podem com pouco esforço adotar o EvolutionChecker.

Atualmente o EvolutionChecker não possui uma interface gráfica para execução, o que facilitaria ainda mais sua utilização. Ele também não foi testado dentro de uma empresa, em um contexto onde poderíamos avaliar se a utilização do EvolutionChecker aumenta a reutilização de componentes e diminui os conflitos gerados pela evolução dos componentes ao longo do tempo. Além disso, poderíamos avaliar a usabilidade e modificabilidade da ferramenta.

A Extensão 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, nos baseamos no *Default Component Profile* (um *Profile* já especificado pelo RAS) e criamos quatro *Profiles*: *AbstractComponent*,*ConcreteComponent*, *InterfaceDefinition* e *Configuration*. 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, apresentamos o *Default Profile*, *Interface Definition Profile*, *Abstract Component Profile*, *Concrete Component Profile* e o *Configuration Profile*.

A.1 *Interface Definition Profile*

A.1.1 Breve descrição

O *Interface Definition Profile* descreve um tipo de interface. Poucas alterações foram realizadas sobre o *Component Profile* descrito pelo RAS. Criamos um novo elemento chamado **InterfaceDefinition** 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.

A.1.2 Novos elementos

InterfaceDefinition
A classe *InterfaceDefinition* 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: *development, published* e *discarded*. 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*, que nesse caso seria a *InterfaceDefinition* que foi estendida.

A.2 *Abstract Component Profile*

A.2.1 Breve descrição

O *Profile* do Componente Abstrato 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.

A.2.2 Novos elementos

AbstractComponent: A classe *AbstractComponent* tem um identificador único, o atributo *name*, dois atributos para representar os contratos especificados por aquela interface. Qualidade de serviço

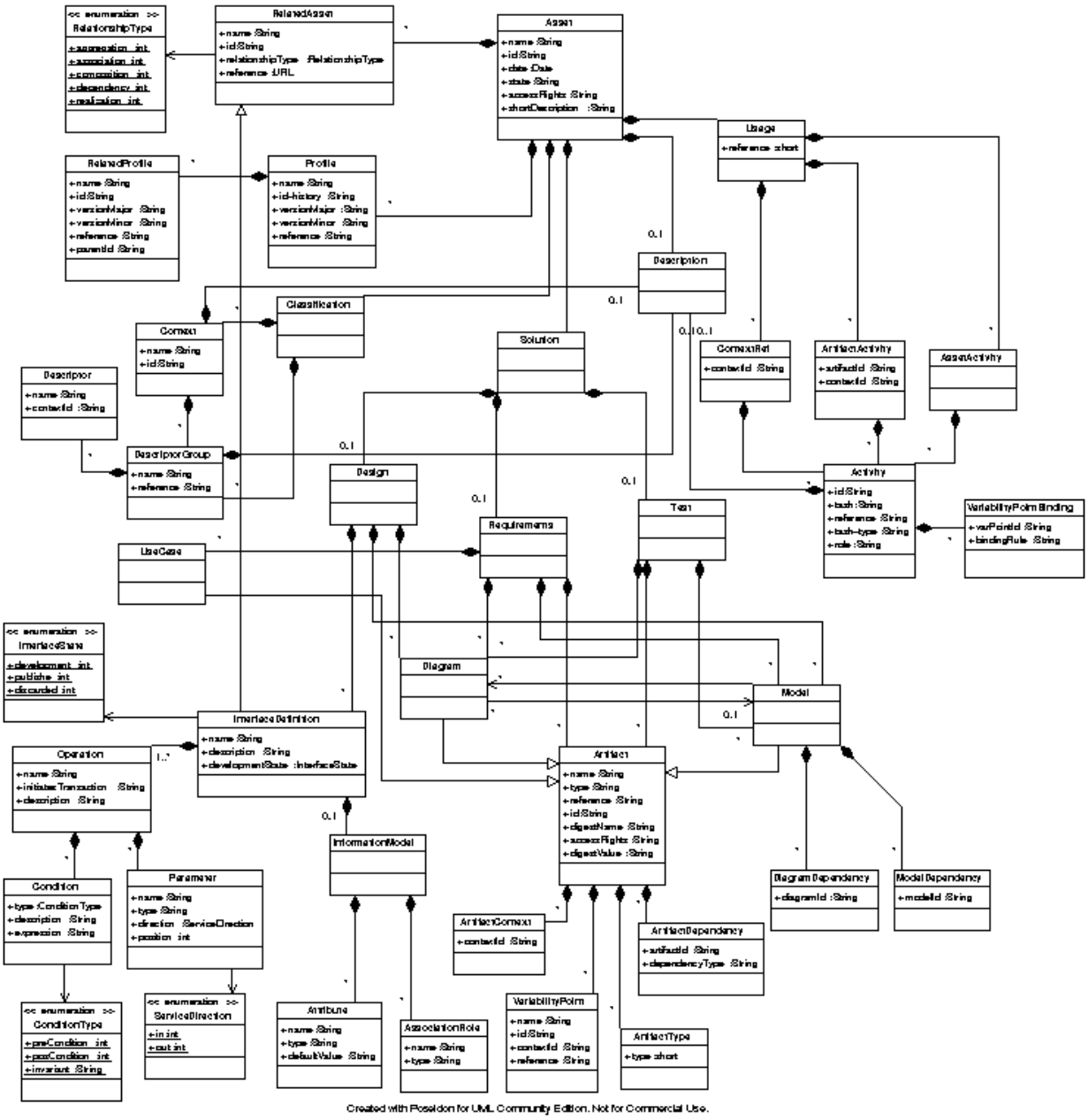

Figura 19: Interface Definition Profile

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.

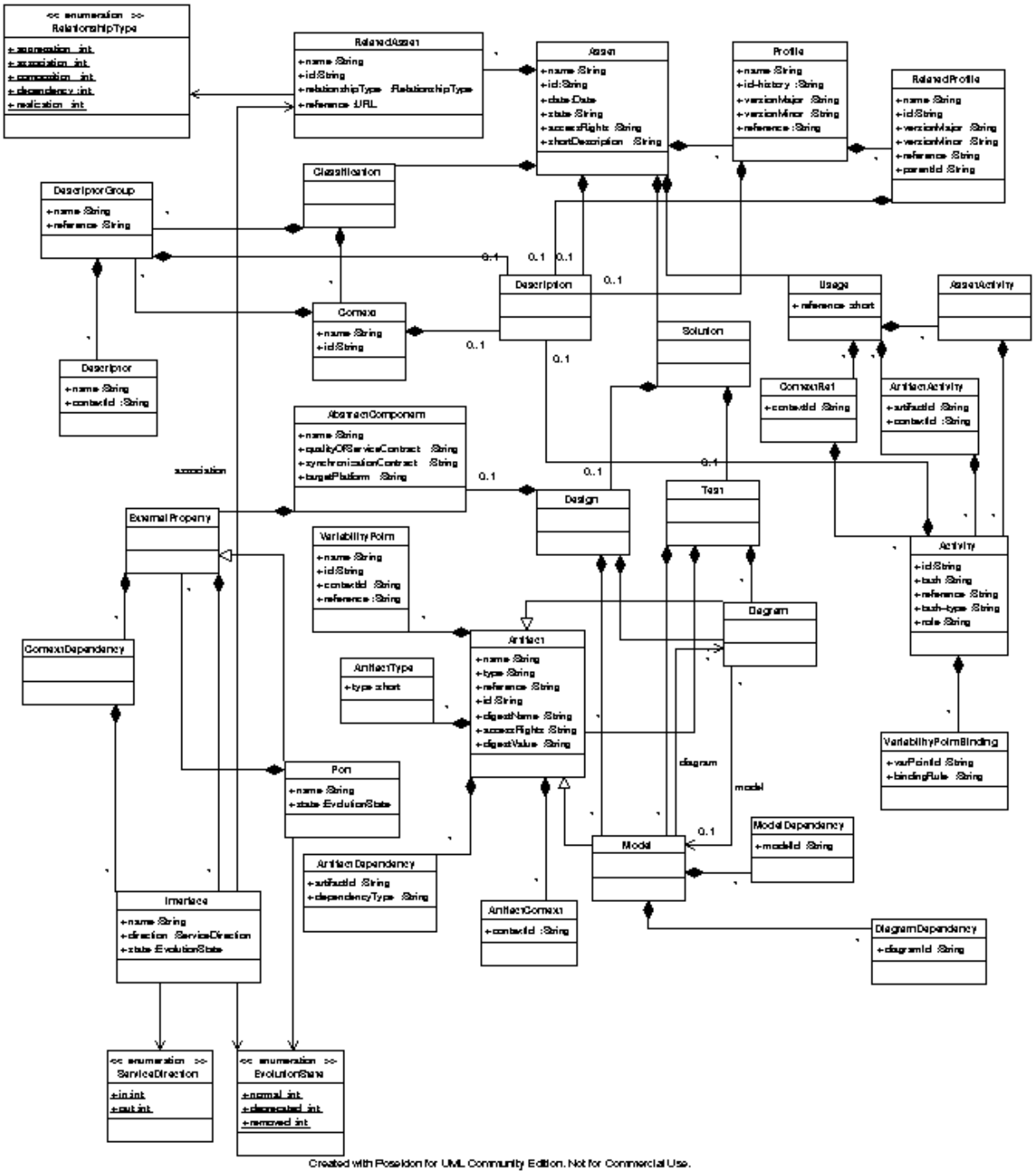

Figura 20: Abstract Component Profile

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

Interface: A classe *Interface* representa as interfaces implementadas pelo componente abstrato. Ela possui um *name*, que é um identificador único de interface, *direction* que informa se ela é provida(*out*) ou requerida(*in*) e *interfaceDefinitionName* 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* representa um conjunto de interfaces e possuem apenas um identificador único: *name*.

A.3 *Concrete Component Profile*

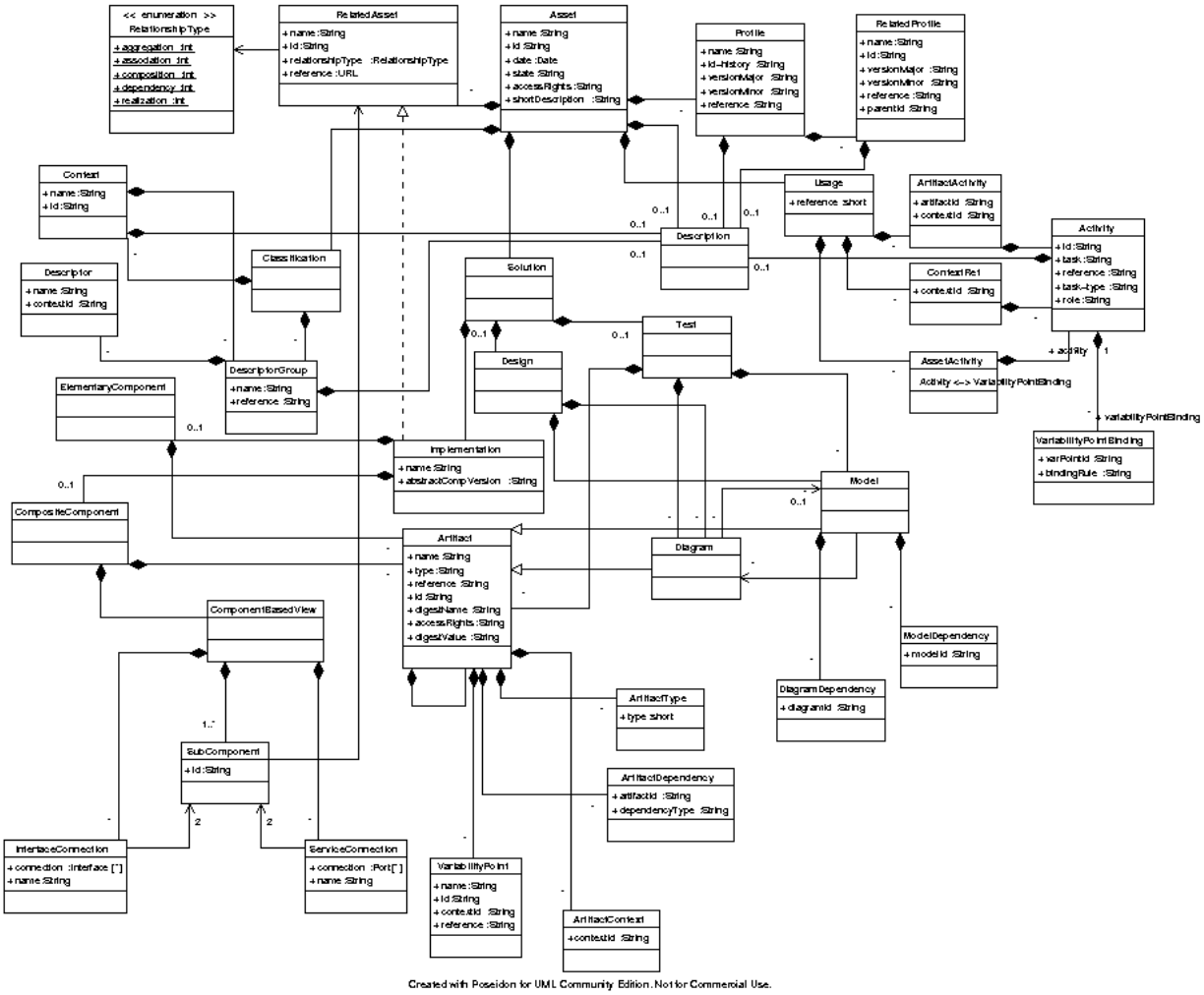

Figura 21: Concrete Component Profile

A.3.1 Breve descrição

No *ConcreteComponent Profile* utilizamos duas formas de representar um componente concreto: o *CompositeComponent* e *ElementaryComponent*. Como possui características de fase de Projeto e da fase de Implementação, *CompositeComponent* compõem tanto *Design* quanto *Implementation*.

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

A.3.2 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**. Esse componente concreto implementa um componente abstrato, representado como **RelatedAsset**. Um componente concreto depende de **External Libraries** e **Tools**, 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 representa a arquitetura interna do componente concreto e composto por **InterfaceConnection**, **ServiceConnection** e **Subcomponents**.

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, `swCompID` 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.4 *Configuration Profile*

A.4.1 Breve Descrição

O *Profile* de Configuração 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.

A.4.2 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 é o elemento que liga um subcomponente de um componente concreto composto a um componente concreto. Este elemento relaciona-se com um componente abstrato.

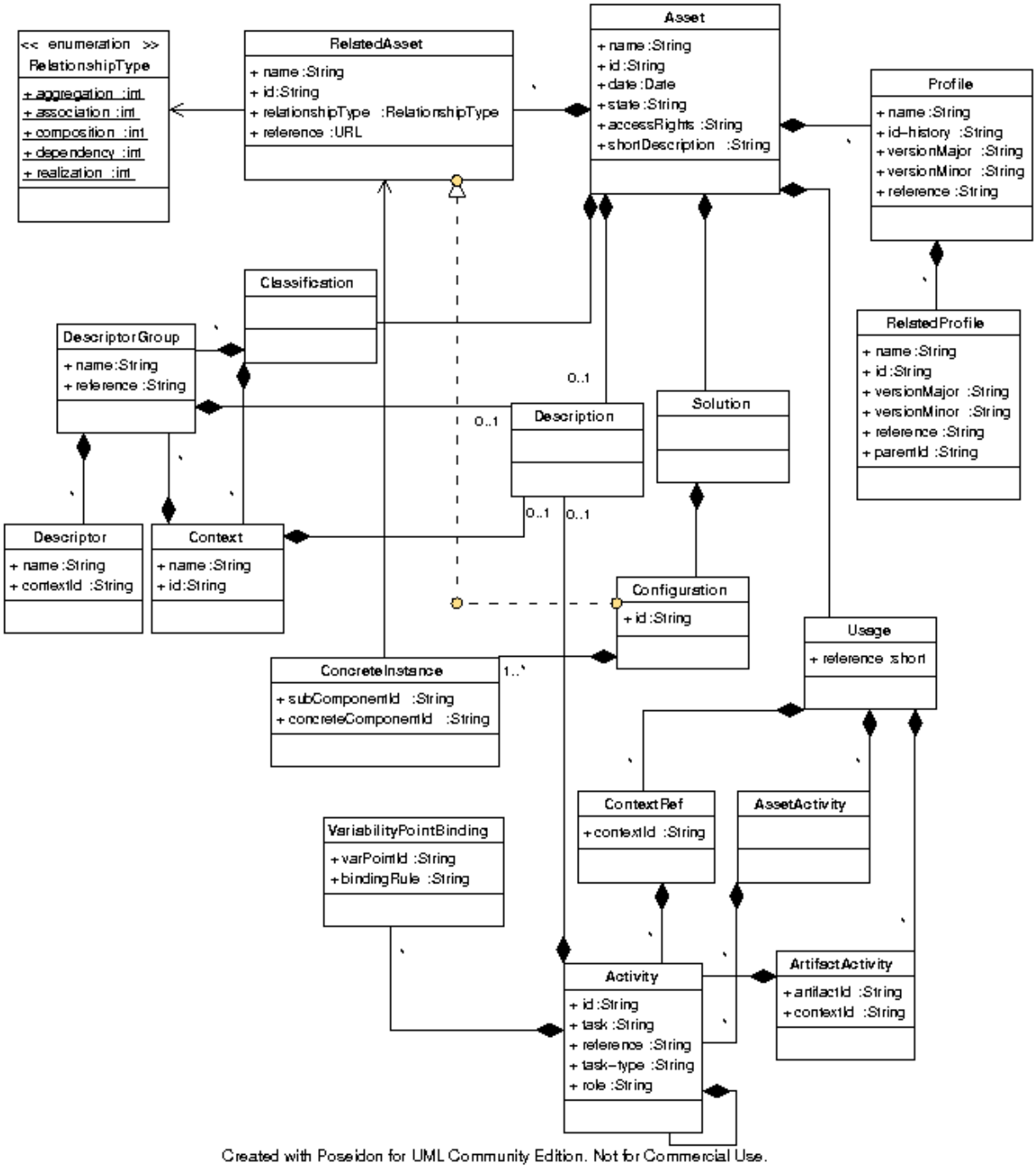

Figura 22: Configuration Profile

B Especificações do Caso de Estudo

B.1 Especificação atual do JavaRMIServer

```
<?xml version="1.0" encoding="UTF-8"?>
<asset id="2316542113" name="JavaRmiServer" version="1.6.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="abstractprofile_v2.xsd">
 <profile name="AbstractComponent" versionmajor="1"
  versionminor="1">
  <relatedprofile />
 </profile>
 <solution>
  <design>
   <abstractcomponent name="JavaRmiServer">
    <externalproperty>
     <contextdependency>
      <interface name="LoaderHandler" state="normal"
       direction="provided">
       <relatedasset>LoaderHandler</relatedasset>
      </interface>
      <interface name="RemoteCall" state="normal"
       direction="provided">
       <relatedasset>RemoteCall</relatedasset>
      </interface>
      <interface name="RemoteRef" state="normal"
       direction="provided">
       <relatedasset>RemoteRef</relatedasset>
      </interface>
      <interface name="RMIClientSocketFactory"
       state="normal" direction="provided">
       <relatedasset>
        RMIClientSocketFactory
       </relatedasset>
      </interface>
      <interface name="RMIFailureHandler"
       state="deprecated" direction="provided">
       <relatedasset>
        RMIFailureHandler
       </relatedasset>
      </interface>
      <interface name="RMIServerSocketFactory"
       state="normal" direction="provided">
       <relatedasset>
        RMIServerSocketFactory
       </relatedasset>
      </interface>
      <interface name="ServerRef" state="normal"
       direction="provided">
       <relatedasset>ServerRef</relatedasset>
      </interface>
      <interface name="Skeleton" state="normal"
       direction="provided">
       <relatedasset>Skeleton</relatedasset>
      </interface>
      <interface name="Unreferenced" state="normal"
       direction="provided">
       <relatedasset>Unreferenced</relatedasset>
      </interface>
```

```
56        </contextdependency>
57      </externalproperty>
58    </abstractcomponent>
59   </design>
60  </solution>
61  <relatedAsset name="LoaderHandler" id="2154131545"></relatedAsset>
62  <relatedAsset name="RemoteCall" id="545344"></relatedAsset>
63  <relatedAsset name="RemoteRef" id="564765724" />
64  <relatedAsset name="RMIClientSocketFactory" id="43535424" />
65  <relatedAsset name="RMIFailureHandler" id="5446543" />
66  <relatedAsset name="RMIServerSocketFactory" id="987653213" />
67  <relatedAsset name="ServerRef" id="7894351" />
68  <relatedAsset name="Skeleton" id="32432454" />
69  <relatedAsset name="Unreferenced" id="798722187" />
70 </asset>
```

B.2 Especificação hipotética do JavaRMIServer

```
<?xml version="1.0" encoding="UTF-8"?>
<asset id="45643128435" name="JavaRmiServer" version="" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="abstractprofile_v2.xsd">
  <profile name="AbstractComponent" versionmajor="1" versionminor="1">
   <relatedprofile />
  </profile>
  <solution>
    <design>
      <abstractcomponent name="JavaRmiServer">
       <externalproperty >
        <contextdependency>
         <interface name="RemoteRef" state="normal"
          direction="provided">
          <relatedasset>RemoteRef</relatedasset>
         </interface>
         <interface name="RMIClientSocketFactory" state="normal"
          direction="provided">
          <relatedasset>RMIClientSocketFactory</relatedasset>
         </interface>
         <interface name="RMIFailureHandler" state="deprecated"
          direction="provided">
          <relatedasset>RMIFailureHandler</relatedasset>
         </interface>
         <interface name="RMIServerSocketFactory" state="normal"
          direction="provided">
          <relatedasset>RMIServerSocketFactory</relatedasset>
         </interface>
         <interface name="ServerRef" state="normal"
          direction="provided">
          <relatedasset>ServerRef</relatedasset>
         </interface>
         <interface name="Unreferenced" state="normal"
          direction="provided">
          <relatedasset>Unreferenced</relatedasset>
         </interface></contextdependency></externalproperty>
      </abstractcomponent>
    </design>
  </solution>
  <relatedAsset name="RemoteRef" id="564765724"/>
  <relatedAsset name="RMIClientSocketFactory" id="43535424" />
  <relatedAsset name="RMIFailureHandler" id="5446543" />
  <relatedAsset name="RMIServerSocketFactory" id="987653213" />
  <relatedAsset name="ServerRef" id="7894351" />
</asset>
```

Referências

[1] *Groovy.* [http://groovy.codehaus.org/].

[2] *Java Platform - Standard Edition 6 - API Specification - DRAFT beta2-b86.* [http://java.sun.com/javase/6/docs/api/index.html].

[3] *Python*. [http://www.python.org/].

[4] 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.

[5] 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.

[6] 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.

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

[8] Moacir Caetano da Silva Júnior. Cosmos - um modelo de estruturação de componentes para sistemas orientados a objetos. Master's thesis.

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

[10] Eclipse Foundation. Eclipse ide. www.eclipse.org.

[11] Ernest Friedman-Hill. Jess. http://www.jessrules.com/jess/.

[12] 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.

[13] GNU. Concurrent versions system. [http://www.nongnu.org/cvs/].

[14] JBoss. Jboss drools. http://www.jboss.com/products/rules.

[15] 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.

[16] OMG. Reusable asset specification, November 2005. www.omg.com.

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

[18] Tigris.org. Subversion. [http://subversion.tigris.org/].

[19] André van der Hoek, Marija Mikic-Rakic, Roshanak Roshandel, and Nenad Medvidovic. Taming architectural evolution. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, 2001. ACM Press.

[20] W3C. Xpath. [http://www.w3.org/TR/xpath].