UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Dissertação de Mestrado

X-ARM: Um Modelo de Representação de Artefatos de Software

Michael Schuenck dos Santos

Natal – RN
Agosto de 2006

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

X-ARM: Um Modelo de Representação de Artefatos de Software

Michael Schuenck dos Santos

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos necessários para a obtenção do grau de Mestre em Sistemas e Computação (MSc.).

Orientador: Prof. Dr. Glêdson Elias da Silveira

Natal – RN
Agosto de 2006

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial Especializada Especializada do Centro de Ciências Exatas e da Terra – CCET.

Santos, Michael Schuenck dos.
X-ARM : um modelo de representação de artefatos de software / Michael Schuenck dos Santos. – Natal, 2006.
150 f. : il.

Orientador : Glêdson Elias da Silveira.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Software – Dissertação. 2. Modelo X-ARM – Dissertação. 3. Representação de metadados – Dissertação. I. Silveira, Glêdson Elias da. II. Título.

RN/UF/BSE-CCET CDU 004.4

X-ARM: Um Modelo de Representação de Artefatos de Software

Michael Schuenck dos Santos

Esta Dissertação de Mestrado foi avaliada em 04 de agosto de 2006 e considerada aprovada pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

Banca Examinadora:

_____
Prof. Dr. Glêdson Elias da Silveira
Orientador

_____
Prof. Dr. Bruno Motta de Carvalho

_____
Profª. Drª. Itana Maria de Souza Gimenes

A meus queridos pais, Genício e Eliana.

Agradecimentos

Agradeço a Deus, por me dar a vida, saúde, motivação e força não apenas para que este trabalho pudesse ser realizado, mas também para que o mesmo fosse realizado com a maior satisfação possível, já que neste tempo, a vida e o mestrado funcionaram em paralelo, e não sequencialmente.

Agradeço ao meu orientador, Prof. Glêdson Elias da Silveira, por sua sábia orientação, por sua exigência pela perfeição e dedicação, que me fizeram crescer enormemente. Tenho por ele uma grande admiração, seja por sua imensa capacidade técnica, quanto por sua integridade e honestidade como profissional e cidadão. Agradeço também à Profª. Valéria Gonçalves Soares, por seu importante apoio desde o dia em que me tornei um mestrando e por direcionar meus primeiros passos nesta jornada.

Agradeço aos membros do comitê avaliador deste trabalho, a Profª Itana Maria de Souza Gimenes e o Prof. Bruno Motta de Carvalho, pelo tempo e empenho, pois não só avaliaram, mas também auxiliaram no enriquecimento deste trabalho com comentários e discussões de importante valor.

Agradeço aos companheiros do laboratório Compose, em especial, a João Paulo Freitas de Oliveira e Yuri Feitosa Negócio. Ao primeiro, por seu auxílio sempre que precisei durante esta jornada, incluindo contribuições não técnicas, como no dia em que tive que ir para o hospital. Ao Yuri, pelas discussões e por suas relevantes contribuições a este trabalho, desde os tempos em que o X-ARM ainda nem havia sido batizado.

Agradeço aos amigos de mestrado Adriana Takahashi, Álvaro Nunes Prestes, Camila de Araújo, Danilo de Abreu Noleto, Macilon Araújo Costa Neto e Raul Benites

Paradeda, além de Matheus Jancy Bezerra Datas. Agradeço-lhes pelos diversos tipos de apoio oferecido, de contribuições técnicas à própria convivência diária, passando por rodízios de pizza, para espairecer.

Expresso minha gratidão aos professores Fabiano Fagundes e Parcilene Fernandes de Brito, pelo fundamental incentivo para que eu me enveredasse por esta jornada, tanto por seus conselhos, quanto por seus exemplos. Sem eles, eu não teria nem sequer cogitado a possibilidade de querer me tornar um mestre. Também agradeço a Antonio da Luz Júnior, Elaine Maria de Matos, Elaine da Silva Monteiro, Francisco de Assis dos Santos Jr. (Chicão), Heres Edison Valdivieso Tobar Neto, Marcus Barbosa Pinto, Nalva Neila Alves da Silva, Pollyane de Almeida Lustosa e Renatto Pereira Mota pelo incentivo e torcida antes e durante a realização do meu mestrado.

Não poderia deixar de agradecer a meus queridos pais, Genício Gomes dos Santos e Eliana Maria das Graças Schuenck dos Santos, por sempre quererem o melhor para mim e por me apoiarem ao mudar pra tão longe em busca de um ideal, mesmo que eles tenham sofrido por isto. Agradeço aos meus irmãos Tarcísio Schuenck dos Santos e Monize Schuenck dos Santos, por me apoiarem nesta jornada, desde o momento em que tomei a decisão de entrar no programa de mestrado, e por cuidarem de nossos pais.

Meus sinceros agradecimentos também ao meu primo Gladstone Miquillitto dos Santos, cujo incentivo foi decisivo para que eu entrasse no caminho que me levaria ao título de mestre.

Por fim, mas não menos importante, agradeço à querida Lisa Marie Medeiros de Souza, por seu apoio nos meses finais de criação desta dissertação e por sua compreensão pelo dias que deixei de ficar com ela para me dedicar a este trabalho.

Resumo

O Desenvolvimento Baseado em Componentes tem como grande desafio a criação de mecanismos que facilitem a identificação de *assets* reusáveis que satisfaçam os requisitos de sistemas particulares sob desenvolvimento. Neste sentido, repositórios de componentes têm sido propostos como meios para se atender esta necessidade. Porém, repositórios precisam representar as características de *assets* que possam ser levadas em consideração pelos consumidores quando da escolha dos *assets* mais adequados às suas necessidades. Neste contexto, a literatura apresenta alguns modelos que foram propostos com a finalidade de descrever as características de *assets*, tais como identificação, classificação, requisitos não funcionais, informações de implantação e uso, arquivos que compõem os *assets*, e interfaces de componentes. No entanto, o conjunto de características representadas por estes modelos é insuficiente para descrever diversas informações usadas antes, durante e depois da aquisição dos *assets*, dentre as quais pode-se destacar informações de negociação e certificação, histórico de mudanças, processo de desenvolvimento usado, eventos, exceções. De forma a solucionar tais deficiências, este trabalho apresenta um modelo baseado em XML para a representação de diversas características, de diversos tipos de *assets*, que possam ser empregadas no desenvolvimento baseado em componentes. O modelo proposto, denominado X-ARM, fornece informações usadas não apenas pelos consumidores na identificação, aquisição e uso de *assets*, mas também para apoiar as atividades dos desenvolvedores de *assets*. Dado que o modelo proposto representa uma expressiva quantidade de informações, este trabalho também apresenta uma ferramenta chamada X-Packager, desenvolvida com o objetivo de auxiliar a descrição de *assets* com X-ARM.

Abstract

A great challenge of the Component Based Development is the creation of mechanisms to facilitate the finding of reusable *assets* that fulfill the requirements of a particular system under development. In this sense, some component repositories have been proposed in order to answer such a need. However, repositories need to represent the asset characteristics that can be taken into account by the consumers when choosing the more adequate *assets* for their needs. In such a context, the literature presents some models proposed to describe the asset characteristics, such as identification, classification, non-functional requirements, usage and deployment information and component interfaces. Nevertheless, the set of characteristics represented by those models is insufficient to describe information used before, during and after the asset acquisition. This information refers to negotiation, certification, change history, adopted development process, events, exceptions and so on. In order to overcome this gap, this work proposes an XML-based model to represent several characteristics, of different asset types, that may be employed in the component-based development. Besides representing metadata used by consumers, useful for asset discovering, acquisition and usage, this model, called X-ARM, also focus on helping asset developers' activities. Since the proposed model represents an expressive amount of information, this work also presents a tool called X-Packager, developed with the goal of helping asset description with X-ARM.

Glossário

CCM – *CORBA Component Model*

CDML - *Component Description Markup Language*

CIDL - *CORBA Interface Description Language*

COM – *Component Object Model*

DBC – Desenvolvimento Baseado em Componentes

DOM - *Document Object Model*

EJB – *Enterprise JavaBeans*

GPL – *GNU General Public License*

MPL – *Mozilla Public License*

OCL - *Object Constraint Language*

OMG – *Object Management Group*

QML - *Quality-of-Service Modeling Language*

RAS – *Reusable Asset Specification*

SWT - *Standard Widget Toolkit*

URL - *Uniform Resource Locator*

X-ARM – *XML-based Asset Representation Model*

X-Packager – *X-ARM Packager*

XML – *Extensible Markup Language*

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Assim como qualquer outra atividade, o desenvolvimento de software possui necessidades relativas ao lançamento de produtos com maior qualidade, com menor custo e com máxima rapidez. Uma das principais formas que podem ser encontradas na literatura e que foram utilizadas no sentido de atender estes objetivos é o reuso de software [McIlroy69]. Diversas abordagens para viabilizar reuso têm sido propostas e utilizadas pela indústria ao longo dos anos. Essas abordagens variam desde a utilização de funções, em programas estruturados, até o emprego de componentes de software, passando por reuso de projeto e a própria orientação a objetos [Frakes05].

Na construção de aplicações a partir de componentes de software, tornou-se popular o termo Desenvolvimento Baseado em Componentes (DBC). Neste sentido, diversas definições de componentes foram propostas. Heineman e Council [Heineman01] apresentam a seguinte definição: “Um componente de software é um elemento de software que adere a um modelo de componente e pode ser independentemente implantado e composto sem modificação de acordo com um padrão de composição”. Já Sametinger [Sametinger97] define um componente da seguinte forma: “Componentes de software reusáveis são auto-contidos, artefatos claramente identificáveis que descrevem e/ou realizam funções específicas e têm interfaces claras, documentação apropriada e um status de reuso definido”. Nesta definição, o fato de ser auto-contido implica que para um componente ser utilizado, não deve ser necessária a implantação de outros componentes. Já o status de reuso de um componente inclui informações sobre seu proprietário, quem o mantém e quem pode ser contactado em caso de problemas, além de informações sobre a qualidade do componente. Uma definição que tem tido boa aceitação é a proposta por Szyperski [Szyperski02], que diz: “Um componente de software é uma unidade de

composição com interfaces contratualmente especificadas e dependências de contexto explícitas. Um componente de software pode ser implantado independentemente e é sujeito a composição por terceiros".

Em uma companhia ou projeto, o DBC possibilita uma divisão clara das atividades dos desevolvedores, uma vez que componentes podem ser desenvolvidos de forma independente por diferentes pessoas, possivelmente trabalhando em paralelo [Sametinger97]. Outra vantagem desta abordagem é não ter que empregar tempo em uma solução que realize as tarefas de componentes já desenvolvidos em outros contextos, quando do desenvolvimento de uma nova aplicação. Por outro lado, torna-se necessário que os componentes utilizados em uma aplicação sejam confiáveis, o que pode ser conseguido com a certificação de componentes, processo que é definido como o conjunto de procedimentos regulamentados e padronizados que resultam na expedição de certificado ou declaração de conformidade específica para produtos ou serviços de uma determinada área de atuação [ISO05].

Componentes de software são construídos seguindo-se um modelo de componentes, conforme indicado em [Heineman01]. Um modelo de componentes define padrões para interoperabilidade, interfaces, nomeação, metadados, customização, composição, evolução e instalação [Heineman01]. A indústria de software tem desenvolvido diversos modelos de componentes, dentre os quais, vários são vastamente utilizados na construção de aplicações de diversos tamanhos e níveis de complexidade. Alguns dos principais modelos de componentes são CCM [CCM06], COM [Brooke01], EJB [Pharoah01a] e .NET [Brooke01].

Os componentes provêem interfaces para serem acessadas pelas unidades de software que os utilizam. Uma interface representa um subconjunto das interações que um componente pode manter com outras entidades de software [Heineman01]. Normalmente ela é expressa como um conjunto de operações que são providas por um componente ou que são requeridas, no caso de um componente necessitar de operações implementadas por outro componente. Geralmente, interfaces são representadas em arquivos de descrição de interfaces, que são dependentes do modelo de componentes, tais como CIDL e arquivos de interfaces de Java. Uma mesma interface pode ser implementada por diversos componentes.

Na utilização de componentes para o desenvolvimento de aplicações, tem-se duas facetas: o desenvolvimento para reuso e o desenvolvimento com reuso [Ribot94]. A primeira é realizada pelo produtor de componentes e refere-se à atividade de desenvolver componentes com a finalidade de distribuí-los, de forma gratuita ou paga. Já o desenvolvimento com reuso é realizado pelo consumidor e consiste em construir aplicações ou novos componentes utilizando componentes previamente desenvolvidos. Neste caso, pode-se notar que a criação de componentes também pode ser considerada desenvolvimento para reuso.

O desenvolvimento com reuso possui como fator limitante, a identificação dos componentes que satisfazem os requisitos de aplicações particulares sob desenvolvimento. Desta forma, tem sido cada vez mais evidente a necessidade de desenvolvimento e popularização de repositórios de componentes. Algumas iniciativas abertas e proprietárias foram propostas pela indústria e pela academia, tais como Component Source [CompSource05], SPARS-J [Inoue03], OSCAR [Nutter03] e Agora [Seacord98]. Repositórios podem ser construídos levando-se em consideração características variadas, como por exemplo, tipos de componentes suportados, controle de versões, tipo de interface de busca e recuperação, suporte a certificação de componentes, modelos de negócios, suporte a processos de desenvolvimento, mecanismos de classificação [Schuenck05].

Da mesma forma, a existência de repositórios de componentes pode beneficiar o desenvolvimento para reuso. Isto pode ser dito pelo fato de que, considerando componentes como produtos comercializáveis, um repositório constitui um mercado onde consumidores podem adquirir os componentes que desejarem, possivelmente com pagamento. Desta forma, este mecanismo propicia a viabilização da atividade de produção de componentes como atividade comercial.

1.1 Motivação

De acordo com [Orso00] e [RAS05], um repositório deve armazenar não somente componentes de software, mas também metadados que os descrevam. Tais metadados precisam ser representados em uma linguagem não ambígua e têm o objetivo de descrever o que o componente é, quais interfaces ele provê, as interfaces que ele requer, os eventos que ele dispara e recebe, quais outros componentes o compõem, a quais certificações ele foi

submetido e por quais entidades certificadoras, e quem pode recuperá-lo, além do preço e das formas de pagamento possíveis. Uma vez disponíveis em um repositório, metadados podem ser utilizados na indexação e classificação de componentes, tornando possível a descoberta dos componentes adequados às necessidades dos consumidores.

Além de componentes de software, repositórios também podem armazenar e manter metadados de interfaces e dos demais artefatos que possam ser gerados no processo de desenvolvimento de componentes para reuso ou utilizados no desenvolvimento de software com reuso. Exemplos desses artefatos são especificações de interfaces, documentação de uso, casos de uso, diagramas de classes, planos de testes, código fonte, dentre outros. No contexto deste trabalho, seguindo a nomenclatura adotada em [RAS05], os diversos tipos de artefatos mantidos em um repositório são denominados genericamente *assets*.

Por outro lado, diversos processos de desenvolvimento baseados em componentes foram propostos e têm sido utilizados na construção de componentes. Exemplos são UML Components [Cheesman01], KobrA [Atkinson02] e Catalysis [D'Souza99]. A utilização de um processo de DBC facilita o uso de componentes por meio da padronização de procedimentos que guiam o produtor desde a concepção das funcionalidades de um sistema até ele estar pronto para ser utilizado, passando pela criação e reuso de componentes. Por outro lado, o desenvolvimento com um processo gera um conjunto de artefatos previstos, que auxiliam os consumidores na utilização dos componentes, tornando possível o correto entendimento de suas funcionalidades.

Apesar da necessidade de repositórios representarem metadados de *assets*, poucos deles tornaram públicos seus modelos de representação, que se referem à notação e às informações que são utilizadas na representação dos metadados dos artefatos que eles mantêm. Além disto, informações como modelos de negócios, certificação, informações sobre o processo de desenvolvimento, controle de acesso e composição por outros componentes são aspectos pouco cobertos pelos modelos de metadados existentes na literatura, principalmente se considerada a cobertura de todos eles simultaneamente.

1.2 Objetivos

Dada a escassez de abordagens para representação de metadados de *assets*, este trabalho propõe um modelo denominado X-ARM, acrônimo para *XML-based Asset Representation*

Model. Conforme explícito no próprio nome, este modelo utiliza XML como meio de representação e organização dos dados. Este modelo foi primeiramente apresentado em [Elias06a], como um modelo independente. Entretanto, uma nova versão do X-ARM foi construída [Schuenck06] de forma a ser uma extensão do RAS [RAS05], visto que esta especificação define as características básicas aos objetivos do X-ARM. RAS é um modelo extensível criado pela OMG para descrever *assets*, mas, apesar do próprio RAS definir uma extensão para representação de componentes, ele apresenta deficiências relativas a reuso de interfaces e eventos por mais de um componente, controle de acesso, suporte a certificação, dentre outros, que serão vistos com mais detalhes na Seção 2.1.

O X-ARM tem como objetivo representar os tipos de *assets* cuja descrição se faz necessária em um repositório de componentes. Neste caso, incluem-se descrições de componentes, de interfaces, de eventos, de artefatos gerados ao longo da construção de um componente e dos demais arquivos de suporte ao funcionamento do repositório, tais como os próprios esquemas do X-ARM.

Além disto, o X-ARM também tem como objetivo apoiar o desenvolvimento de componentes criados com quaisquer processos. Para que isto seja possível, é utilizada uma abordagem em que cada processo é descrito por um documento XML, no qual é possível a representação das etapas com seus respectivos artefatos. A partir disto, torna-se possível o estabelecimento de correlações dos artefatos gerados com os tipos previstos na descrição do processo empregado.

Além de especificar o modelo X-ARM, este trabalho também tem o objetivo de apresentar a ferramenta X-Packager, criada como um *plug-in* para o ambiente Eclipse [Eclipse06] com o objetivo de facilitar a descrição e o empacotamento de *assets* de acordo com o modelo X-ARM.

1.3 Organização do trabalho

O restante deste trabalho está organizado da seguinte forma: no Capítulo 2 são discutidas algumas abordagens correlacionadas; o Capítulo 3 apresenta o modelo X-ARM; o Capítulo 4 apresenta a ferramenta X-Packager e um exemplo de uso envolvendo o empacotamento de *assets*; por fim, o Capítulo 5 trata das considerações finais.

2 Trabalhos Relacionados

Idealmente, os metadados dos *assets* mantidos em um repositório devem seguir sintaxes bem definidas, possibilitando que ferramentas construídas para lidar com os *assets* consigam interpretar as informações que os descrevem de forma previsível e adequada. No entanto, poucos modelos de representação foram definidos com o propósito de descrever diferentes tipos de unidades de software, incluindo diversos artefatos utilizados no DBC. Além disso, as propostas existentes apresentam algumas deficiências. Assim, as próximas seções apresentam alguns destes modelos, que constituem trabalhos relacionados ao modelo proposto neste trabalho.

Neste capítulo são apresentados modelos baseados em XML. Para facilitar a compreensão, exemplos destes modelos são apresentados com apoio de notação UML. As classes UML representam elementos XML e os atributos das classes representam atributos XML. A opcionalidade de atributos e as quantidades de ocorrências de elementos são expressas com estereótipos e cardinalidades, respectivamente. A representação de elementos filhos de outros elementos é feita utilizando-se uma notação onde classes são compostas por outras.

2.1 RAS

O RAS [RAS05] é uma proposta da OMG cujo objetivo é definir um padrão de empacotamento e descrição de *assets*. Neste contexto, o termo *asset* define um agrupamento de artefatos gerados em um ciclo de desenvolvimento de software com objetivo de atender um determinado problema ou grupo de problemas. Exemplos desses artefatos são modelos, requisitos, arquivos de código fonte, descritores de instalação, casos de teste, scripts, dentre outros.

RAS define formas de organização dos arquivos que compõem um *asset*, nas quais as metainformações de descrição ficam armazenadas em um arquivo nomeado *manifest.rmd*, conhecido como manifesto, que descreve em XML os artefatos relacionados e as características do *asset*. São identificados três cenários diferentes de organização dos arquivos: um no qual todos os arquivos do *asset* são empacotados em um único arquivo com o algoritmo de compressão ZIP [Moffat97]; um cenário em que os arquivos dos artefatos que compõem um *asset* são dispostos em suas localizações originais e o arquivo *manifest.rmd* reúne as informações do *asset* como um todo e referencia todos os arquivos que compõem o *asset* em questão; e, por fim, o cenário em que para comporem o *asset*, os arquivos precisam de alguma adaptação, sendo copiados para localizações diferentes de onde estão os arquivos originais, com o manifesto referenciando as cópias [RAS05].

Os manifestos podem representar diferentes características com seus elementos XML, dependendo do tipo de *asset* que ele descreve. As características consideradas como básicas a todos os tipos de *assets* são descritas por uma categoria do RAS chamada *Core RAS*. Por outro lado, para a representação de características particulares dos diferentes tipos de *assets*, RAS utiliza uma organização baseada em *profiles*, que consistem de extensões das características do *Core RAS* e que podem ser estendidos recursivamente a fim de permitir a representação de tipos de *assets* mais especializados. Cada *profile* é definido com esquemas XML. RAS define três *profiles* principais: o *Default Profile*, que na prática é uma realização do *Core RAS* , representando o grupo principal de características de *assets*; o *Default Component Profile*, que é uma extensão do *Default Profile* e adiciona elementos XML relacionados à representação de componentes; e o *Default Web Service Profile*, que também é uma extensão do *Default Profile*, adicionando características relacionadas a *Web services*. A hierarquia dos *profiles* definidos pelo RAS é ilustrada na Figura 1.

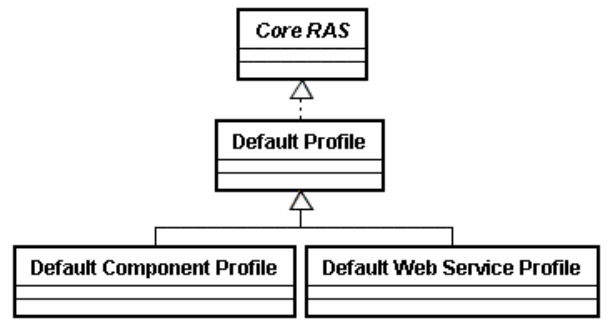

Figura 1: Hierarquia de *profiles* do RAS.

Visto que o X-ARM é uma extensão do RAS, detalhes de cada um destes *profiles* são apresentados nas próximas subseções. Como o *Default Profile* é uma realização do *Core RAS*, não adicionando novos elementos, ambos são apresentados na mesma subseção.

2.1.1 Core RAS e Default Profile

Além de dados de identificação, RAS considera cinco tipos de informações como sendo essenciais para a descrição de quaisquer tipos de *assets*, que são representadas no *Core RAS*. A organização dos principais elementos definidos pelo *Core RAS*, usados para a representação destas informações é ilustrada na Figura 2.

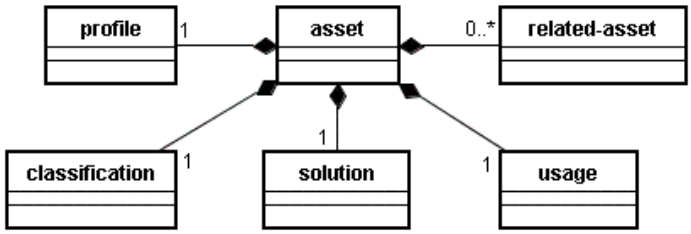

Figura 2: Principais elementos definidos pelo Core RAS.

O elemento <*classification*> representa metadados que possibilitem descrever os *assets* de forma que estes possam ser encontrados dentro de um conjunto de *assets*. A fim de representar os arquivos que compõem o *asset*, é empregado o elemento <*solution*>. O terceiro tipo de informação é utilizado para auxiliar nos procedimentos para instalação, customização e uso dos *assets* e é representado pelo elemento <*usage*>. É definido também o elemento <*related-asset*>, usado para descrever os relacionamentos entre *assets*. Por fim, informações sobre o tipo de *asset* descrito são representadas com o elemento <*profile*>. Além destes, é definido o elemento *asset*, que é o elemento raiz nos documentos XML de descrição dos *assets*. Para representar todas as suas informações, cada um destes elementos possui atributos e elementos filhos. Mais detalhes podem ser encontrados na especificação do RAS [RAS05].

Em um manifesto, pode ser indicado o arquivo que contém o esquema XML empregado. Isto é feito no próprio elemento raiz, <*asset*>, com o atributo *xsi:schemaLocation*, que é comumente utilizado em documentos XML para indicar os esquemas XML que os validam [Benz03]. Considerando a utilização de um repositório

distribuído, esta característica constitui um ponto fraco do RAS, visto que com isto, não se consegue transparência de localização do esquema XML adotado.

2.1.2 Default Component Profile

O *Default Component Profile* é uma extensão do *Default Profile* que adiciona elementos a fim de permitir a representação de componentes. Ele considera vários conceitos e princípios definidos pelo processo UML Components [Cheesman01]. Neste sentido, para permitir a representação dos diferentes tipos de artefatos que compõem um componente, são definidos quatro elementos principais [RAS05]: <*requirements*>, usado para indicar os artefatos gerados na fase destinada ao levantamento de requisitos; <*design*>, que descreve os arquivos gerados durante o projeto do componente; <*implementation*>, que indica os arquivos que contêm a implementação do componente; e <*test*>, empregado para representar artefatos relevantes a testes sobre o componente. A organização destes elementos é apresentada na Figura 3, onde a maioria dos elementos definidos pelo *Core RAS* estão implícitos, com exceção de <*asset*> e <*solution*>, além de <*artifact*>, que é um elemento filho de <*solution*>, definido pelo *Core RAS*.

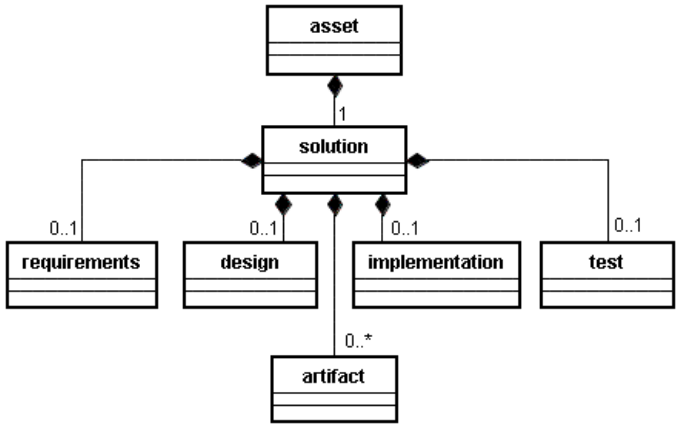

Figura 3: Principais elementos do Default Component Profile.

Da mesma forma como apresentado no caso do *Core RAS*, cada um destes elementos possui atributos e outros elementos como filhos. Nota-se que, apesar de considerar o processo UML Components, não existe um mapeamento direto entre as fases deste processo e os elementos definidos no *Default Component Profile*. Além disso, o elemento <*artifact*> permite representar artefatos que não se encaixem em quaisquer dos

quatro elementos introduzidos neste *profile*, o que proporciona certa flexibilidade na organização dos artefatos. Porém, esta flexibilidade é limitada, já que componentes construídos em processos que envolvam um número maior de fases podem ter artefatos de diferentes fases misturados em diversos elementos <*artifact*>.

Além disto, é possível destacar outras três desvantagens deste *profile*. A primeira reside justamente no fato do *Default Component Profile* considerar aspectos particulares de UML Components, tal como a especificação de um modelo de informação, que pode não se aplicar à descrição de componentes desenvolvidos com outros processos. Outra desvantagem refere-se ao fato deste profile representar em um só documento XML, o componente e suas partes, não oferecendo suporte à reusabilidade de descrições. Por exemplo, na prática, interfaces são bastante reusadas entre diferentes componentes, mas com o *Default Component Profile* não é possível o reuso da descrição de uma interface. Finalmente, este *profile* não possibilita a descrição de outras características de componentes, tais como eventos, propriedades, requisitos não funcionais, composição por outros componentes, certificação e modelos de negócios.

2.1.3 Default Web Service Profile

RAS provê meios para a descrição das funcionalidades de *Web Services* do ponto de vista de seus clientes [RAS05]. Assim, o objetivo do *Default Web Service Profile* é representar apenas os artefatos gerados ao longo do desenvolvimento de um *Web Service* além de referenciar um arquivo WSDL [W3C04a].

Em geral, a organização do *profile* para *Web Services* do RAS segue a mesma estrutura do *profile* para descrição de componentes, diferindo-se basicamente sobre como descrevem as interfaces. No *Default Component Profile*, as interfaces de um componente são especificadas dentro do elemento <*design*>, onde são detalhadas em termos de operações e modelo de informação. Por outro lado, o *Default Web Service Profile*, simplesmente provê um elemento para indicar seções dentro de um arquivo WSDL que descrevem as interfaces providas de um *Web Service* [RAS05]. Porém, esta indicação é feita usando-se atributo textual, não existindo uma padronização sobre como referenciar estas seções.

Outra diferença em relação ao *Default Component Profile* é que a descrição de *Web services* com RAS inclui um elemento, filho do elemento <*implementation*>, para referenciar a URL que contém o documento WSDL.

2.2 OSD

OSD (*Open Software Description*) é um vocabulário XML que tem o objetivo de descrever pacotes de software e seus relacionamentos com outros pacotes. Em OSD, é possível expressar diversas características de softwares tais como versionamento, estrutura interna dos pacotes e dependências existentes [Hoff97].

O propósito de OSD é oferecer suporte para ambientes de distribuição automatizada de software. Neste sentido, uma ferramenta é responsável por ler e interpretar um arquivo OSD e recuperar as informações necessárias para fazer o *download* e instalar ou atualizar o respectivo pacote de software, levando em consideração suas dependências.

É importante destacar que, embora o foco de OSD seja a descrição da distribuição de softwares em geral, ele trata várias características relacionadas às tecnologias de componentes. Por exemplo, OSD cobre a representação de características como nomeação, versionamento, dependências e requisitos não funcionais [Hoff97]. Entretanto, devido ao objetivo de OSD ser basicamente a descrição de software pronto para execução por usuários finais [Hoff97], diversas particularidades de componentes não são tratadas. Por exemplo, OSD não especifica interfaces, informações de modelos de negócios, certificação, visibilidade e eventos.

2.3 CDML

A CDML [Varadarajan01] é o meio de descrição de componentes empregado pelo ComponentXchange [Varadarajan01], que é um sistema projetado para intermediar a interação entre consumidores e produtores de componentes. Descrições de componentes em CDML representam quatro tipos de informações, denominadas pelo autor como *aspectos*: sintáticos, funcionais, não-funcionais, e de licença e comercialização. Os primeiros descrevem as interfaces providas, requeridas e eventos. Os aspectos funcionais são descritos na forma de chave/valor e os aspectos não-funcionais são descritos em uma abordagem semelhante à adotada por QML [Frolund98], em que as informações são descritas em termos de nomes, valores e os tipos de relacionamentos entre estes dois. Por

fim, para os aspectos de licença e comercialização, a CDML dá suporte apenas a licenças do tipo *pay-per-use*.

Apesar de ser adequado aos requisitos do sistema ComponentXchange, a CDML apresenta diversas desvantagens em relação ao X-ARM. Por exemplo, CDML descreve interfaces e eventos, porém, uma mesma descrição de um destes itens não pode ser reusada por mais de um componente, o que é possível em X-ARM. Diversas outras informações representadas por X-ARM não são descritas pela CDML, tais como informações sobre classificação, propriedades, certificação dos componentes, processo de desenvolvimento adotado na construção dos componentes, histórico de mudanças, arquivos que compõem os componentes. Além disto, mesmo no caso de licenças do tipo *pay-per-use*, informações importantes como forma de pagamento e tempo de suporte não são descritas por CDML.

2.4 Modelo proposto por Redolfi et al.

Em [Redolfi05] é proposto um modelo para descrição de componentes sem um nome específico. Apesar de também poder ser empregado para auxiliar as buscas por componentes, seu principal propósito é facilitar a utilização de componentes pelos desenvolvedores que os adquirirem. Outro objetivo deste modelo é ser usado para guiar a definição de um repositório capaz de armazenar, catalogar, buscar e recuperar componentes. O modelo não define uma notação específica a ser usada para representação das informações, tal como XML, adotada pelos demais modelos apresentados neste capítulo.

Assim como o X-ARM, o modelo apresentado em [Redolfi05] descreve não apenas componentes, mas também qualquer artefato gerado durante o processo de desenvolvimento. Estes artefatos compõem a especificação do componente, que, da mesma forma adotada no X-ARM, é separada da implementação do componente. A especificação de um componente é descrita apenas com diagramas e artefatos textuais, não sendo definidos mecanismos padronizados para sua representação. Além disto, os artefatos da especificação de um componente podem referenciar outros artefatos. Por exemplo, um diagrama de classes pode referenciar um modelo de casos de uso a partir do qual ele foi gerado.

Já a implementação de um componente inclui informações sobre a linguagem de programação, a plataforma de execução e sobre sua documentação, além de dados sobre testes realizados e sobre as interfaces. Os testes indicam as condições sob as quais o componente funciona corretamente, porém, ao contrário das certificações representadas pelo X-ARM, eles são realizados pelos próprios produtores de componentes, e não por uma entidade de certificação externa. Associar interfaces às implementações faz com que uma mesma interface tenha que ser especificada para todo componente que a implementar.

2.5 Considerações Finais

Nota-se que a maioria dos modelos apresentados neste capítulo é baseada em XML, permitindo a interpretação das informações dos *assets* de forma programática por aplicações de software. Isto atende um requisito do modelo proposto neste trabalho, já que as descrições serão utilizadas principalmente por repositórios e ferramentas para manipulação de *assets*.

A fim de sumarizar os aspectos descritos pelos modelos apresentados, a Tabela 1 apresenta um comparativo com o modelo X-ARM, que será apresentado em detalhes no Capítulo 3.

	RAS	**OSD**	**CDML**	**Redolfi et al.**	**X-ARM**
Nomeação	Sim	Sim	Sim	Sim	Sim
Versionamento	Sim	Sim	Não	Sim	Sim
Requisitos não funcionais	Não	Sim	Sim	Sim	Sim
Interfaces providas	Sim (sem reuso)	Não	Sim (sem reuso)	Sim (sem reuso)	Sim (com reuso)
Interfaces requeridas	Não	Não	Sim (sem reuso)	Sim (sem reuso)	Sim (com reuso)
Eventos	Não	Não	Sim	Não	Sim
Propriedades	Não	Não	Não	Não	Sim
Modelos de negócios	Não	Não	Sim (rudimentar)	Não	Sim
Certificação do *asset*	Não	Não	Não	Sim (teste)	Sim
Certificação do produtor	Não	Não	Não	Não	Sim

Classificação	Sim (rudimentar)	Não	Não	Não	Sim
Informações sobre o processo de desenvolvimento	Sim (UML Components)	Não	Não	Não	Sim (Qualquer processo)
Composição de componentes	Não	Não	Não	Não	Sim
Controle de visibilidade	Sim (rudimentar)	Não	Não	Não	Sim

Tabela 1: Comparação dos modelos de representação de artefatos de software.

Nomeação e versionamento são características que precisam ser representadas a fim de identificar os *assets* e suas diferentes versões. Nota-se que todos os modelos apresentados representam os nomes de seus artefatos e apenas a CDML não representa dados de versões.

A representação de requisitos não funcionais é importante para que o consumidor possa saber se um *asset* é adequado às restrições de seus recursos de software e hardware [Chung99]. Neste sentido, com exceção de RAS, todos os outros modelos suportam a representação de requisitos não funcionais.

Quando os artefatos descritos são componentes, é natural que suas interfaces sejam representadas. Assim, apenas OSD não descreve interfaces, já que seu objetivo não é descrever componentes. Podem ser identificados dois tipos de interfaces: providas e requeridas. Os modelos que representam componentes descrevem ambos os tipos de interfaces, com exceção de RAS, que descreve apenas interfaces providas. Além disto, estes modelos adotam uma abordagem em que para cada componente que provê ou requer uma determinada interface, esta interface deve ser novamente descrita. Por outro lado, X-ARM adota uma abordagem em que uma mesma interface é descrita apenas uma vez, fazendo com que os componentes apenas indiquem suas interfaces providas ou requeridas.

Eventos são elementos que permitem a comunicação assíncrona entre componentes, sendo úteis para notificação de componentes quando determinadas condições forem satisfeitas em outros componentes [Szyperski02]. Dentre os modelos apresentados, nota-se que apenas CDML suporta eventos, também suportados por X-ARM.

Componentes podem possuir propriedades, que são atributos usados para customização, normalmente acessados por meio de métodos de acesso (*getters* e *setters*) [Szyperski02]. Além de X-ARM, nenhum outro modelo apresentado representa propriedades.

A representação de modelos de negócios é importante no contexto de um repositório de componentes porque é possível que os produtores estejam interessados em receber algum retorno financeiro da sua atividade de produção de componentes. Dentre os modelos pesquisados, apenas CDML representa informações sobre negociação, porém, em comparação com X-ARM, tal modelo não permite a especificação de informações sobre possíveis formas de pagamento. Além disto, CDML não permite que informações adicionais, não descritas pelos elementos XML definidos pelo modelo, sejam especificadas.

Principalmente se considerado que reuso de software normalmente envolve produtores e consumidores diferentes, é importante que a qualidade e confiabilidade dos *assets* sejam atestadas por processos de certificação [Wohlin94] [Wallnau04]. Dentre os modelos apresentados, nenhum deles representa informações sobre certificação, porém, o modelo apresentado em [Redolfi05] chega a permitir a representação das condições sob as quais os componentes foram testados, e sob as quais eles funcionam corretamente. Isto não significa que os componentes têm qualidade e são confiáveis, já que possivelmente quem informa os dados dos testes são os próprios produtores dos componentes, ao contrário do X-ARM, que possibilita que entidades certificadoras testem e certifiquem os *assets*. Além de *assets* individuais, o próprio produtor de *assets* pode ser certificado. No entanto, além do X-ARM, nenhum outro modelo apresentado representa esta característica.

A fim de facilitar a busca por *assets* é útil a adoção de um esquema de classificação [Sametinger97]. Apenas o RAS adota este mecanismo, permitindo a associação de palavras-chave a um *asset*. Além deste mecanismo, X-ARM também adota um esquema de especificação de áreas de aplicação aplicáveis para o *asset*, aderentes a um ou mais conjuntos de áreas.

A representação de informações sobre o processo de desenvolvimento adotado na construção dos *assets* é importante para que consumidores possam conhecer os demais *assets* gerados no mesmo projeto de um *asset* específico. Assim, por exemplo, se um

consumidor está interessado em um componente, é possível conhecer os diagramas UML relacionados àquele componente e que podem auxiliar na compreensão de suas funcionalidades. As informações sobre o processo também podem ser úteis para os próprios produtores, ao possibilitar que a partir do modelo de representação sejam informados os *assets* já gerados e os que ainda restam ser gerados em um processo de desenvolvimento. Dentre os modelos apresentados, apenas RAS mantém este tipo de informações, mas elas são referentes apenas ao processo de desenvolvimento UML Components [Cheesman01]. O modelo apresentado em [Redolfi05] também permite relacionar diferentes artefatos, porém, sem definir significados para estes relacionamentos e sem associar informações sobre algum processo de desenvolvimento.

Componentes são unidades de composição [Szyperski02]. Ou seja, eles podem ser compostos a fim de formar componentes maiores ou mesmo aplicações. Assim, quando um componente for composto por outros, torna-se útil conhecer quais componentes o compõem, a fim de facilitar a compreensão das funcionalidades do componente composto. Dentre os modelos apresentados, nenhum representa informações sobre composição de componentes.

Considerando-se que um repositório pode armazenar *assets* de um projeto ainda em andamento, no qual outros *assets* ainda serão produzidos, torna-se interessante que os *assets* possam se tornar públicos apenas após o término do projeto. Por outro lado, os *assets* também podem ficar visíveis apenas para um conjunto de produtores que possam estar trabalhando no mesmo projeto. Assim, um esquema de visibilidade se torna importante para definir os produtores autorizados a acessar os *assets* mantidos em um repositório. Apenas RAS representa alguma informação sobre permissões de acesso, mas apenas como um único atributo textual, ao contrário do X-ARM, que define uma estrutura de elementos para representação de produtores autorizados e proibidos de acessar os *assets*.

3 X-ARM

A representação de metadados de *assets* pode ser utilizada em diversos aspectos das funcionalidades de um repositório de componentes, incluindo controle de visibilidade, suporte a processos de desenvolvimento, classificação, modelos de negócios, certificação, composição por outros componentes, requisitos não funcionais e representação de componentes de acordo com diversos modelos de componentes.

Visando representar informações sobre estes aspectos, este trabalho define um modelo para representação de *assets* denominado X-ARM. Uma primeira versão deste modelo foi apresentada em [Elias06a], consistindo de um modelo independente. No entanto, uma nova versão foi construída [Schuenck06] com um grupo de *profiles* que estendem as características do *Default Profile* do RAS [RAS05]. Esta alteração foi realizada pelo fato de que esta especificação define os elementos básicos aos propósitos do X-ARM, além de que o RAS é um padrão proposto pela OMG e adotado por importantes empresas da indústria de software.

3.1 Assets em X-ARM

No contexto do X-ARM, o termo *asset* representa qualquer artefato de software gerado por um processo de desenvolvimento, tais como: casos de uso, código fonte, código executável, planos de testes, etc. Além destes, outros tipos de *assets* representam modelos, que são usados para descrever conceitos associados ao DBC. Na prática, estes modelos podem restringir valores associados a algumas características dos *assets*, como, por exemplo, o modelo de componentes adotado, que representa os possíveis tipos de interfaces e eventos que um componente daquele modelo deve possuir.

Conforme ilustrado na Figura 4, no X-ARM, os *assets* estão divididos em 4 categorias, especificadas por seus respectivos *profiles* (*Artifact Profile*, *Interaction Profile*,

Component Profile e *Model Profile*). Por sua vez, estas categorias estão subdivididas em 18 tipos de *assets*, que representam artefatos gerados por processos de desenvolvimento ou modelos que descrevem conceitos associados ao DBC. As categorias e seus tipos de *assets* são detalhados nas próximas subseções.

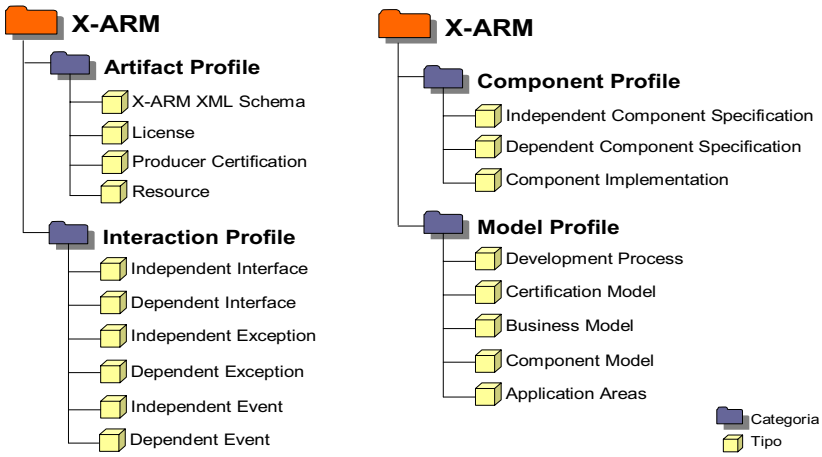

Figura 4: Categorias e tipos de assets do X-ARM.

3.1.1 Especificação e Implementação de Componentes

Considerando que um consumidor normalmente acessa um repositório de componentes à procura de componentes, eles podem ser considerados os principais tipos de *assets*. No X-ARM, componentes são especificados usando o *X-ARM Component Profile* e podem ser representados em três níveis de abstração, que definem diferentes tipos de *assets*:

- *Independent Component Specification*: representa a especificação de um componente de forma independente de qualquer modelo de componente.
- *Dependent Component Specification*: representa a especificação de um componente de forma dependente de seu modelo de componente. Neste caso, a especificação do componente considera as características particulares do modelo de componente adotado, tais como CCM, *JavaBeans*, EJB, COM e *Web services*.
- *Component Implementation*: representa a implementação do componente constituída dos códigos fonte e executável correspondentes a uma especificação dependente do modelo de componente.

Desta forma, no X-ARM, uma determinada especificação de componente independente de modelo pode derivar várias especificações de componentes dependentes de modelo. Por sua vez, cada especificação de componente dependente de modelo pode derivar diversas implementações de componentes.

3.1.2 Interfaces, Eventos e Exceções

Interfaces, eventos e exceções são especificados usando o *X-ARM Interaction Profile*. Interfaces são coleções de operações que são utilizadas para especificar os serviços providos e requeridos pelos componentes. Desta forma, as interfaces descrevem o comportamento dos seus respectivos componentes e são definidas levando em consideração as interações que são importantes para os clientes dos componentes [RAS05]. No X-ARM, as especificações de interfaces são descritas em dois níveis de abstração, que representam diferentes tipos de *assets*:

- *Independent Interface*: representa a especificação de uma interface de forma independente dos modelos de componentes adotados pelas implementações de componentes que possuem aquela interface como provida ou requerida.
- *Dependent Interface*: representa a especificação de uma interface de forma dependente dos modelos de componentes, adotados pelas implementações de componentes que possuem aquela interface como provida ou requerida. Neste caso, a especificação da interface considera as características particulares do modelo de componente adotado.

Geralmente, interfaces dependentes de modelo são também descritas usando uma linguagem de descrição, específica do respectivo modelo de componente. Por exemplo, componentes CCM e *JavaBeans* têm suas interfaces descritas usando a linguagem CIDL e a própria linguagem Java, respectivamente. Vale ressaltar que as especificações de interfaces dependentes de modelo contêm os seus arquivos de descrição expressos nas linguagens padronizadas pelos respectivos modelos de componentes.

Eventos são mecanismos de interação assíncrona [Pyarali00], que basicamente se caracterizam pela existência de fontes de eventos, que disparam mensagens que comunicam receptores de eventos registrados sobre a ocorrência de determinadas situações. Para o registro dos receptores de eventos, as fontes provêem operações específicas para adição e

remoção de receptores de eventos. Além disto, as mensagens trocadas para notificação dos receptores têm como parâmetros estruturas de dados ou objetos que representam os eventos.

As descrições de interfaces e as descrições de eventos são utilizadas nas especificações dos componentes, definindo seus comportamentos e formas de interações. De forma semelhante às interfaces, no X-ARM, os eventos também podem ser especificados em dois níveis de abstração, também representando diferentes tipos de *assets*:

- *Independent Event*: representa a especificação de um evento de forma independente dos modelos de componentes, adotados pelos componentes que podem dispará-lo ou recebê-lo.
- *Dependent Event*: representa a especificação de um evento de forma dependente dos modelos de componentes, adotados pelos componentes que podem dispará-lo ou recebê-lo. Neste caso, a especificação do evento considera as características particulares do modelo de componente adotado.

Exceções são recursos utilizados para descrever erros que podem acontecer durante a execução de uma operação. Os erros podem ser representados por estruturas de dados ou objetos. Considerando que um determinado tipo de exceção pode acontecer em diversas operações, no X-ARM as exceções são descritas de forma independente das operações, proporcionando assim o reuso das mesmas. De forma semelhante às interfaces e eventos, as exceções também podem ser especificadas em dois níveis de abstração, representado diferentes tipos de *assets*:

- *Independent Exception*: representa a especificação de uma exceção de forma independente dos modelos de componentes, adotados pelos componentes que podem dispará-la ou recebê-la.
- *Dependent Exception*: representa a especificação de uma exceção de forma dependente dos modelos de componentes, adotados pelos componentes que podem dispará-la ou recebê-la. Neste caso, a especificação da exceção considera as características particulares do modelo de componente adotado.

Como pode ser observado, interfaces, eventos e exceções podem ser representadas em dois níveis de abstração. O primeiro é o nível independente de modelo de componente, onde se tem uma descrição abstrata que pode ser reutilizada para a geração de descrições dependentes de modelo. O segundo é o nível dependente de modelo de componente, onde se tem uma descrição que depende do modelo de componente adotado. As especificações dependentes de modelo podem referenciar suas respectivas especificações independentes de modelo. Desta forma, as especificações dependentes de modelo reusam as especificações independentes de modelo.

3.1.3 Artefatos

Artefatos são quaisquer outros produtos de software, diferentes de componentes, interfaces, eventos e exceções, gerados durante um processo de desenvolvimento, tais como: diagramas de atividade e de classes, casos de uso e planos de testes. Em alguns casos, artefatos podem ser reutilizados integralmente entre versões de um mesmo componente. No entanto, em outros casos, artefatos também podem ser reutilizados com adaptações e servir de base para a construção de outros componentes de software.

Artefatos são especificados usando o *X-ARM Artifact Profile*. Além de representar produtos de software gerados por um processo de desenvolvimento, este *profile* também representa outros três diferentes tipos de *assets*. Desta forma, esta categoria de *assets* é dividida nos seguintes tipos de *assets*:

- *X-ARM XML Schema*: armazena um esquema XML que descreve um determinado *profile* do X-ARM. O esquema mantido neste tipo de *asset* é usado para validar as especificações X-ARM de outros *assets*, que são instâncias daquele *profile*.
- *License*: armazena uma determinada licença de uso (GPL, MPL, etc.). Este *asset* é reusado pelos demais *assets* que adotam aquela licença específica.
- *Producer Certification*: descreve as informações da certificação do processo de software adotado por um determinado produtor.
- *Resource*: descreve quaisquer outros artefatos de software, diferentes de componentes, interfaces, eventos e exceções, gerados por um processo de desenvolvimento.

Na realidade, estes diferentes tipos de *assets* foram subdivididos apenas para facilitar a identificação e diferenciação dos mesmos em um repositório. No entanto, as estruturas de representação destes tipos são similares.

3.1.4 Modelos

No X-ARM, diversas informações descritas nos *assets* devem adotar ou seguir um determinado padrão, modelo ou processo. Por exemplo, um componente deve: adotar um modelo de componente, ser gerado usando um processo de desenvolvimento, ser classificado usando um padrão de classificação, e suportar um conjunto de modelos de negócio. Para ser possível especificar os padrões, modelos ou processos adotados pelos diversos *assets*, o *X-ARM Model Profile* está subdividido em cinco tipos de *assets*:

- *Development Process*: define um processo de desenvolvimento de software baseado em componentes, identificando suas fases e respectivos tipos de artefatos gerados.
- *Certification Model*: descreve um processo de certificação de componentes, identificando os diferentes parâmetros avaliados e seus possíveis conceitos.
- *Business Model*: define um modelo de negócio, identificando as diferentes características e seus possíveis valores.
- *Component Model*: especifica um modelo de componentes, enumerando os diferentes tipos de características e elementos que o compõem.
- *Application Areas*: define um esquema de classificação baseado em áreas de aplicação.

Os tipos de *assets* gerados a partir do *X-ARM Model Profile* representam os conjuntos de valores possíveis de serem utilizados nos demais *profiles*, assegurando um mecanismo uniforme de descrever e indicar os padrões, modelos ou processos associados. Por exemplo, na definição das áreas de aplicação de um *asset*, somente serão aceitos valores especificados em um determinado *asset* do tipo especificação de áreas de aplicação (*Application Areas*). Os outros tipos desta seção seguem lógica similar ao exemplo, envolvendo o uso destas especificações por outros tipos de *assets*.

3.1.5 Empacotamento de assets

No contexto do X-ARM, todo *asset* deve ser descrito em um arquivo de manifesto para descrição de *assets*, chamado *manifest.xml*. Os arquivos que compõem o *asset* devem estar dentro do arquivo de pacote e devem ser referenciados pelo manifesto. O manifesto e todos os arquivos referenciados são empacotados em um único arquivo usando o algoritmo de compressão ZIP [Moffat97].

Os arquivos de manifesto habilitam: (a) verificação de consistência dos arquivos empacotados antes do armazenamento de um *asset* em um repositório, garantindo que os *assets* referenciados já estão registrados no repositório; (b) o *download* de *assets* requeridos em tempo de execução ou desenvolvimento; e (c) detectar dependências entre *assets*, tornando possível assegurar que um *asset* somente pode ser excluído de um repositório quando nenhum outro *asset* depende dele.

3.1.6 Identificação única

Para garantir a unicidade de identificação, independência e transparência de localização dos *assets* mantidos em um repositório, o X-ARM adota um esquema de nomeação hierárquico, similar a uma URL [Berners-Lee94], mas sem referenciar a localização física.

Em um repositório compatível com o X-ARM, os *assets* armazenados são logicamente agrupados em zonas e domínios. Nesta abordagem, zonas representam os produtores de *assets*, enquanto domínios são usados basicamente para agrupar famílias de produtos. Uma zona pode possuir domínios, que por sua vez, sempre são subordinados a alguma zona. A Figura 5 exemplifica a hierarquia de nomes, onde a zona raiz é representada por um ponto.

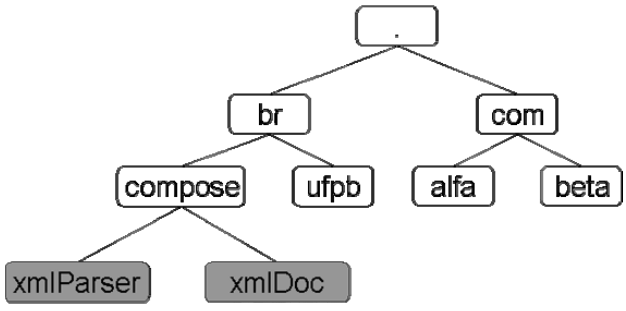

Figura 5: Hierarquia de nomes no X-ARM.

De acordo com a Figura 5, pode-se dizer que *br.compose.xmlParser* identifica o *asset* cujo nome é *xmlParser,* produzido por desenvolvedores pertencentes à zona ou domínio *br.compose*. Uma subzona ou subdomínio é qualquer zona ou domínio abaixo de outro em uma hierarquia de zonas e domínios. Por exemplo, *br.ufpb* é uma subzona ou subdomínio da zona ou domínio *br*.

A fim de possibilitar que atividades relacionadas a gerência de configuração possam ser realizadas, torna-se necessário que as versões dos produtos gerenciados possam ser identificadas [Conradi98]. Neste sentido, X-ARM define um esquema de versionamento simples, onde a versão do *asset* é definida por uma seqüência de números inteiros separados por pontos. Por exemplo, "*2.0.2*". Ressalta-se que a nova versão de um *asset* obrigatoriamente deve ser maior do que a versão anterior.

Para gerar o identificador único de um *asset*, une-se o seu nome à sua versão, separando-os por um hífen ("-"). Como exemplo, considerando o componente *br.compose.xmlParser* em sua versão *2.0.2*, a identificação única deste *asset* é "*br.compose.xmlParser-2.0.2*". Vale ressaltar que o atributo *version* definido pelo RAS é usado para indicar a versão do *asset* em termos comerciais, tal como, por exemplo, a versão 2.0 do aplicativo *OpenOffice*. Por outro lado, a versão para identificar um *asset* em um repositório é sempre concatenada ao nome do *asset*.

3.2 Hierarquia de *Profiles*

O X-ARM é considerado compatível com o RAS porque estende o *Default Profile*. Assim, é proposto o *X-ARM Artifact Profile*, que acrescenta elementos para representação de informações sobre controle de visibilidade, processo de desenvolvimento adotado, áreas de aplicação, histórico da evolução de *assets* e modelos de negócios. Com os elementos definidos neste *profile*, é possível descrever os *assets* do tipo *X-ARM XML Schema*, *License*, *Producer Certification* e *Resource*. Este *profile* é especializado em três outros, para representação das demais categorias e tipos de *assets*:

- *X-ARM Component Profile*: usado para descrever especificações e implementações de componentes.
- *X-ARM Interaction Profile*: tem a função de representar as características de interfaces, eventos e exceções.

- *X-ARM Model Profile*: descreve especificações de processos de desenvolvimento, áreas de aplicação, modelos de certificação, modelos de negócios e modelos de componentes.

A hierarquia de *profiles*, a partir do *Core RAS* e do *Default Profile*, ambos definidos pelo RAS, é apresentada na Figura 6. Nesta figura, a classe *Asset* representa o *Core RAS*, cuja realização é feita pelo *Default Profile*. Nos exemplos adotados no texto, assume-se que uma instância de um *profile* é automaticamente uma instância do *profile* superior também.

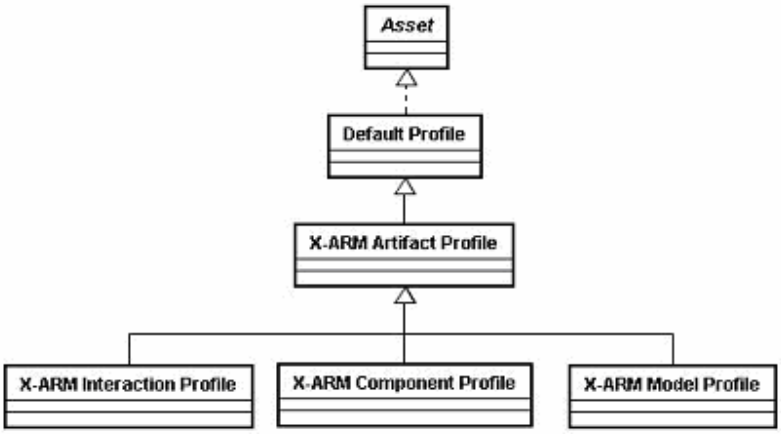

Figura 6: Hierarquia de profiles do X-ARM.

O *CoreRAS* define <*asset*> como elemento raiz, onde, por meio de seus atributos e elementos, é possível representar algumas características básicas de *assets*, tais como a identificação e o produtor do *asset*, os arquivos empacotados, o *profile* referente ao *asset* descrito e palavras-chave utilizadas para classificação. A organização e as funcionalidades dos elementos definidos no elemento <*asset*> podem ser encontradas com detalhes na especificação do RAS [RAS05]. Desta forma, as próximas seções deste documento apenas descrevem e discutem os elementos e atributos introduzidos pelos *profiles* do X-ARM.

3.3 X-ARM Artifact Profile

Em RAS, o objetivo do *Default Profile* é especificar os elementos que representam as características básicas das descrições dos *assets*. Quando da criação de novos *profiles*, estas características já estão automaticamente incorporadas. No contexto deste trabalho, foi identificada a necessidade de representar novas características, não suportadas *pelo Default Profile* do RAS. Esta necessidade motivou a criação do *X-ARM Artifact Profile*.

A representação do *X-ARM Artifact Profile* é apresentada na Figura 7. Para ser possível distinguir as classes definidas em *profiles* mais genéricos das classes definidas em *profiles* mais especializados (neste caso as classes do *Default Profile* e as classes do *X-ARM Artifact Profile*, respectivamente), a seguinte notação é adotada: as classes brancas são aquelas definidas no *profile* mais genérico, mas que não foram alteradas; as classes com o estereótipo <<*modified*>> são provenientes do *profile* mais genérico, mas foram alteradas por meio da adição de atributos ou de alterações semânticas; e as classes na cor cinza são as classes definidas pelo *profile* mais especializado. Esta notação também é empregada na apresentação dos demais *profiles* descritos neste trabalho.

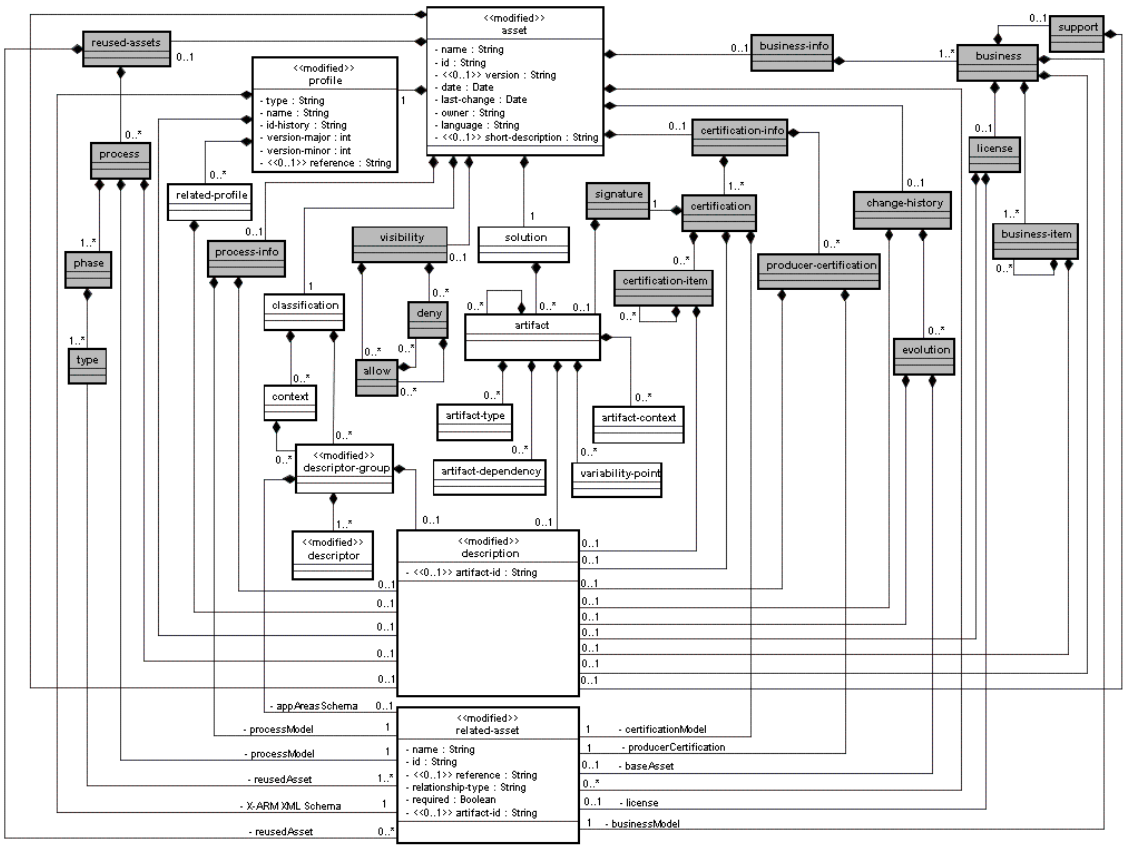

Figura 7: X-ARM Artifact Profile.

Os elementos adicionados foram <*reused-assets*>, <*process-info*>, <*visibility*>, <*certification-info*>, <*change-history*> e <*business-info*>, que representam, respectivamente, informações relativas aos *assets* utilizados para a geração de um determinado *asset*, indicação do tipo de artefato ao qual se refere o *asset* e do processo de

desenvolvimento adotado, controle da visibilidade pelos consumidores, certificação do *asset*, histórico da evolução do *asset* e representação de modelos de negócios. Estes elementos são detalhados nas próximas subseções.

O elemento <*asset*> foi alterado de forma a adicionar os atributos *owner*, *last-change* e *language*. O primeiro indica o nome do produtor. O segundo representa a data da última mudança realizada no *asset*, sem que fosse gerada uma nova versão. O terceiro e último atributo, *language*, é responsável por indicar o idioma no qual o manifesto do *asset* foi descrito. O valor do atributo *language* deve ser um dos valores especificados no *ISO 639 Language Codes* [ISO05b]. Além da adição destes atributos, o elemento <*asset*> foi alterado de forma a ter seus atributos *state* e *access-rights* desconsiderados, já que eles são opcionais e que no contexto de X-ARM estes atributos não são necessários.

No elemento <*profile*> foi adicionado o atributo *type* e um elemento <*related-asset*>. O atributo *type* é usado para indicar o tipo do *asset* descrito e tem como valores possíveis, aqueles indicados na Figura 4. Já o elemento <*related-asset*> indica um *asset* do tipo *X-ARM XML Schema* que representa o arquivo *XML Schema* usado para validar o manifesto do *asset*. Neste caso, o valor do atributo *relationship-type* do elemento <*related-asset*> é "*X-ARM XML Schema*".

O elemento <*related-asset*> permite indicar um *asset* relacionado, identificado pelo atributo *id*. A este elemento foi adicionado o atributo *required*, que indica se o *asset* atual só funciona adequadamente se o *asset* relacionado for recuperado. Também foi adicionado o atributo opcional *artifact-id*. Se este atributo não é usado, o *asset* relacionado está armazenado no repositório. Por outro lado, se o atributo *artifact-id* é usado, o *asset* relacionado está armazenado no próprio pacote do *asset* descrito. Neste caso, o atributo *artifact-id* representa o identificador do arquivo que contém o *asset* relacionado dentro do pacote do *asset* descrito.

O elemento <*description*> é utilizado para incluir uma descrição sobre algum elemento definido pelo X-ARM. Esta descrição pode ser realizada de duas formas. Na primeira, o elemento <*description*> contém um pequeno texto que descreve o respectivo elemento associado. Na segunda, o elemento <*description*> contém o atributo *artifact-id*, cujo propósito é apontar para um arquivo interno do pacote, indicado em um elemento

<*artifact*>, que contém uma descrição mais detalhada. O elemento <*description*> é empregado por diversos elementos nos *profiles* definidos pelo X-ARM.

As próximas subseções tratam das funcionalidades dos elementos introduzidos pelo *X-ARM Artifact Profile*.

3.3.1 Artefatos do Processo de Desenvolvimento

Um processo de desenvolvimento normalmente é organizado como um conjunto de fases, constituídas por atividades, métodos, práticas e transformações, usados para atingir um objetivo específico. Este objetivo é associado a um ou mais resultados concretos finais, que são os produtos da execução do processo [Cheesman01]. Tais produtos são artefatos específicos gerados em cada fase, tais como diagramas, especificações de interfaces, código fonte e o próprio componente. Por exemplo, o processo de desenvolvimento *UML Components* [Cheesman01] tem como artefatos de saída do *workflow Requirements*: o modelo conceitual do negócio, que é um diagrama de classes; e o modelo de casos de uso, que é um diagrama de casos de uso. Estes artefatos, segundo a classificação definida pelo X-ARM, são considerados *assets* do tipo *Resource*, descritos pelo *X-ARM Artifact Profile*.

A identificação da fase do processo de desenvolvimento na qual o artefato foi gerado permite oferecer mecanismos que possam verificar e validar se um determinado tipo de artefato pode ser gerado naquela fase do processo. Por outro lado, considerando a identificação da fase do processo, é possível controlar o conjunto e a seqüência de artefatos que devem ser gerados ao longo do processo.

Para indicar o processo de desenvolvimento adotado, o *X-ARM Artifact Profile* define o elemento <*process-info*>, cuja estrutura é apresentada na Figura 8. O atributo *phase* indica a fase do processo na qual o *asset* foi gerado. O atributo *output-type* indica o tipo de artefato associado àquele *asset*. Considerando que as fases dos processos podem ser organizadas hierarquicamente em diversos níveis, o valor do atributo *phase* é representado usando os nomes individuais das fases, que compõem a hierarquia, separados por pontos. Por exemplo, no caso do processo *UML Components*, cuja fase (*workflow*) *Specification* possui a subfase (estágio) *Component Identification*, o valor do atributo *phase* é "*Specification.Component Identification*".

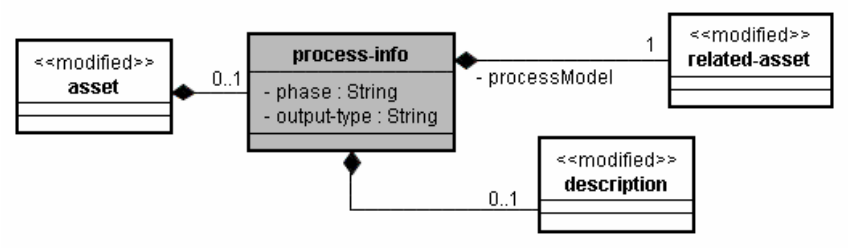

Figura 8: Representação de processo de desenvolvimento adotado.

Opcionalmente, cada elemento <*process-info*> também pode ter um elemento <*description*>, que contém uma descrição do processo, a fase e o tipo de artefato. O *asset* que especifica o processo de desenvolvimento adotado é identificado pelo elemento <*related-asset*>, que deve possuir no seu atributo *relationship-type* (definido no RAS) o valor "*processModel*". A partir da identificação do processo adotado, é possível verificar se o tipo de *asset* indicado no atributo *output-type* pode ser gerado na fase indicada pelo atributo *phase*. Portanto, os atributos *phase* e *output-type* devem ter seus valores em conformidade com um *asset* do tipo *Development Process*, especificado no *X-ARM Model Profile* na Seção 4.4.1.

A Figura 9 ilustra um trecho da utilização do elemento <*process-info*> na descrição de um *asset* com o *X-ARM Artifact Profile*. Neste exemplo, o *asset* descrito segue o processo *UML Components*. No elemento <*process-info*> são indicados a fase (*Requirements*) e o tipo de artefato (*Use Case Model*) contido no *asset*. O processo de desenvolvimento usado é indicado no elemento <*related-asset*>, que indica o *asset* que descreve o processo *UML Components*. Além disto, o *asset* possui os arquivos *ucm1.jude* e *ucm2.jude*, que correspondem ao modelo de caso de uso (*Use Case Model*), gerado pelo processo *UML Components* no *workflow Requirements*.

```
1. <asset name="TextEditUCM" id="br.compose.teucm-1.0" date="2005-10-07" owner="Compose" language="en">
2.     <process-info phase="Requirements" output-type="Use Case Model">
3.         <related-asset name="UML Components" id="br.ufpb.umlcs-1.0"
                        relationship-type="processModel" required="false"/>
4.         <description>
5.             <artifact id="20" reference="umlcomponentsdoc.doc" type="doc"/>
6.         </description>
7.     </process-info>
8.     <solution>
9.         <artifact id="9" reference="ucm1.jude" type="jude"/>
10.        <artifact id="10" reference="ucm2.jude" type="jude"/>
11.    </solution>
…
12. </asset>
```

Figura 9: Exemplo de representação do processo de desenvolvimento adotado.

3.3.2 Rastreabilidade do Processo de Desenvolvimento

Conforme apresentado na Seção 3.3.1, para cada *asset* pode ser representado o tipo de artefato a que ele se refere, dentro de um processo de desenvolvimento. Porém, ao final do processo de desenvolvimento ou de um marco dele, apenas este mecanismo não permite representar informações que identifiquem todos os *assets* que foram gerados. A fim de atender esta necessidade, torna-se necessária a existência de um mecanismo de rastreabilidade que possibilite descobrir os *assets* que foram produzidos em momentos anteriores à criação de um dado *asset* sob descrição. Na prática, os *assets* já existentes são reusados ou tomados como base para a criação de novos *assets*. Desta maneira, ao final da execução de um processo de desenvolvimento é possível identificar, a partir do componente pronto, todos os demais *assets* que foram gerados para se chegar nele. Isto é feito em um procedimento recursivo, em que cada descrição de *asset* pode informar os *assets* nos quais ele se baseou.

Para apoiar este mecanismo, o X-ARM define o elemento <*reused-assets*>, ilustrado na Figura 10, responsável por agrupar todos os *assets* reusados pelo *asset* que está sendo descrito.

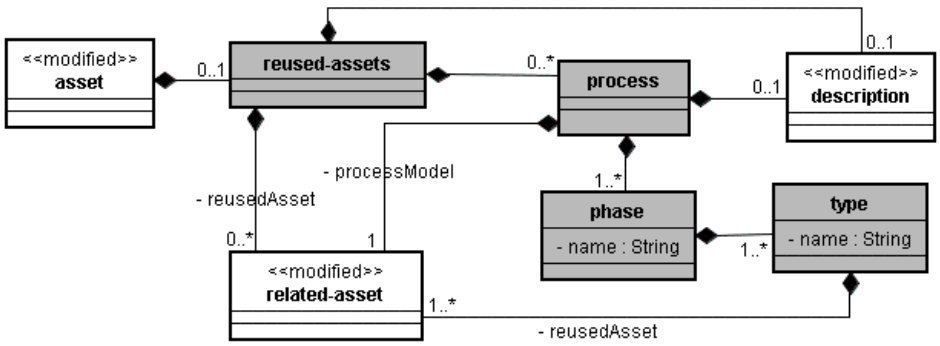

Figura 10: Representação de assets reusados.

O elemento <*reused-assets*> pode ter ocorrências do elemento <*process*>, que é usado para informar o *asset* que descreve o processo de desenvolvimento adotado. Para isto, o elemento <*process*> emprega o elemento <*related-asset*>, cujo atributo *relationship-type* deve ter "*processModel*" como valor. Caso os *assets* reusados tenham sido produzidos em diferentes processos, mais de uma ocorrência do elemento <*process*> pode existir.

Para indicar a fase do processo na qual os *assets* reusados foram gerados é utilizado o elemento <*phase*>. O atributo *name* indica o nome da fase. Como as fases podem ser organizadas hierarquicamente, o valor deste atributo é representado com os nomes das fases e subfases separados por pontos, da mesma forma que o atributo *phase* do elemento <*process-info*>, apresentado na Seção 3.3.1.

Para representar os *assets* reusados e seus tipos é utilizado o elemento <*type*>, cujo atributo *name* informa o tipo de artefato (definido pelo processo de desenvolvimento) ao qual o *asset* se refere. Para indicar os próprios *assets* reusados, é utilizado o elemento <*related-asset*> que, neste caso, tem em seu atributo *relationship-type* o valor "*reusedAsset*".

Além de permitir a representação dos *assets* reusados que foram criados seguindo processos de desenvolvimento, X-ARM também possibilita a representação do reuso de *assets* que foram produzidos sem qualquer processo. Assim, o próprio elemento <*reused-assets*> pode ter elementos <*related-asset*>, usados para informar as identificações dos *assets* reusados. Neste caso, o valor do atributo *relationship-type* de <*related-asset*> é "*reusedAsset*".

Na representação dos *assets* reusados, como ilustrado na Figura 11, o X-ARM permite a adoção de três abordagens de rastreabilidade. Na Figura 11, as setas indicam as referências a *assets* reusados. Na abordagem de rastreabilidade por *asset* (esquerda da Figura 11) cada *asset* referencia os demais *assets* diretamente reusados. Nesta abordagem, para identificar todos os *assets* reusados ao longo do processo, é necessário a adoção de um procedimento recursivo que explora os *assets* reusados por cada *asset*. Por outro lado, esta abordagem permite que, a qualquer momento do processo de desenvolvimento, seja possível rastrear os *assets* reusados.

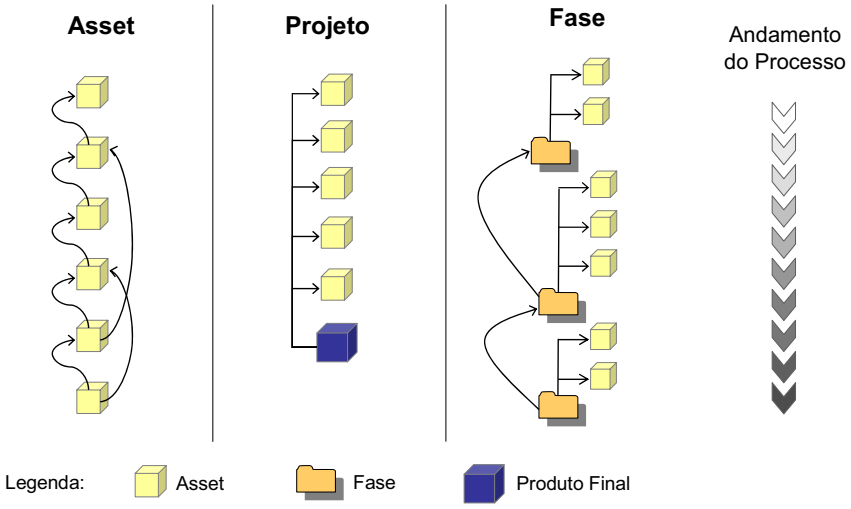

Figura 11: Abordagens de rastreabilidade de assets.

A abordagem por projeto (centro da Figura 11) se baseia no fato de que diversos artefatos são produzidos ao longo do processo de desenvolvimento, antes da geração do produto final. Assim, nesta abordagem o produto final é responsável por indicar todos os artefatos produzidos antes da sua geração. Desta forma, não é necessário que cada *asset* represente os *assets* que foram necessários para sua criação, bastando que as implementações de componentes representem estas informações. Neste caso, a rastreabilidade é feita apenas verificando-se a descrição X-ARM da implementação do componente, sem necessidade de qualquer procedimento recursivo. No entanto, esta abordagem somente permite rastrear os *assets* reusados ao final do processo de desenvolvimento, quando as implementações dos componentes são produzidas.

Na abordagem por fase (direita da Figura 11), não é necessário que todos os *assets* indiquem os *assets* reusados, mas também não é exigido que o processo seja concluído para que a rastreabilidade seja informada. Esta abordagem se baseia na existência de marcos no processo de desenvolvimento, tais como as próprias fases do processo. Neste caso, quando uma fase é concluída, um *asset* do tipo *Resource* deve ser gerado para agregar todos os *assets* que foram gerados naquela fase. Além disso, o *asset* do tipo *Resource* que representa cada fase pode referenciar o respectivo *asset* da fase imediatamente anterior. Assim, ao final da última fase, a rastreabilidade é possível com um procedimento recursivo onde, para cada fase, pode-se encontrar os *assets* que foram gerados nela, além das fases anteriores, a partir das quais também podem ser encontrados seus *assets*, e assim por diante. No entanto,

esta abordagem apresenta um esforço adicional, já que um novo *asset* deve ser gerado para agrupar as referências.

Ressalta-se que estas três abordagens não são as únicas possíveis, pois o X-ARM também permite a combinação das mesmas. Por exemplo, é possível adotar a abordagem por *asset* ao longo do processo de desenvolvimento, e, ao final, nas implementações dos componentes, também adotar a abordagem por projeto. Desta forma, por um lado, ao longo do projeto, as informações de rastreabilidade podem ser recuperadas recursivamente. Por outro lado, ao final do projeto, as informações de rastreabilidade podem ser rapidamente recuperadas diretamente das implementações de componentes.

3.3.3 Classificação Baseada em Áreas de Aplicação

Encontrar o *asset* apropriado para as necessidades de uma aplicação é essencial para o sucesso do reuso. Desta forma, torna-se necessário um mecanismo efetivo de classificação dos *assets* dentro de um repositório. Diversos métodos podem ser utilizados para classificação, incluindo textos livres e palavras-chaves [Sametinger97]. Porém, estas abordagens não são muito vantajosas quando considerado que, por não serem padronizadas e não estarem relacionadas a um determinado contexto, palavras-chaves iguais podem ter significados diferentes, podendo gerar resultados indesejados em uma consulta.

Para evitar tais problemas e facilitar a localização de *assets*, o X-ARM padroniza os termos de classificação utilizando uma abordagem em que é pré-definida uma hierarquia de áreas de aplicação, que podem ser posteriormente utilizadas na classificação de *assets*. O conceito de áreas de aplicação adotado no X-ARM é similar ao conceito de domínio adotado na engenharia de domínios [Sametinger97]. A escolha do termo *áreas de aplicação* tem como base a afirmação encontrada em [Sametinger97], que diz que "no contexto de engenharia de software, domínios são áreas de aplicação". No X-ARM, a definição de uma hierarquia de áreas de aplicação é realiza em um *asset* do tipo *Application Areas*, especificado no *X-ARM Model Profile* na Seção 3.4.2.

Como abordagem de classificação, o RAS já provê um mecanismo baseado em descritores agrupados [RAS05] que podem ser usados para classificar os *assets* usando palavras-chave. No RAS, as palavras-chave podem ser definidas sem seguir qualquer regra ou de acordo com um esquema de classificação.

Considerando que o RAS já propõe uma abordagem de classificação, o *X-ARM Artifact Profile* utiliza a estrutura do elemento <*classification*>, previamente definida pelo RAS. Além disso, para representar os conceitos de áreas de aplicação, o X-ARM especifica a sintaxe e a semântica de alguns atributos e define novos relacionamentos. A estrutura do elemento <*classification*> no *X-ARM Artifact Profile* é apresentada na Figura 12.

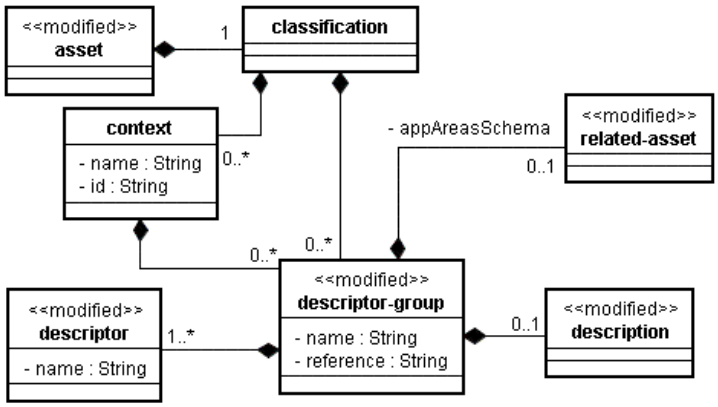

Figura 12: Classificação baseada em áreas de aplicação.

Em termos de estrutura, a única alteração é a adição do relacionamento entre o elemento <*descriptor-group*> e o elemento <*related-asset*>. Neste relacionamento, o elemento <*related-asset*> representa um *asset* do tipo *Application Areas*, que define a hierarquia de áreas de aplicação adotada na classificação do *asset*. No RAS, o atributo *reference* do elemento <*descriptor-group*> tem um propósito similar. No entanto, para manter a uniformidade do X-ARM, onde é necessário referenciar um *asset* do tipo *Application Areas*, optou-se por acrescentar o relacionamento com <*related-asset*>, ao invés de adotar o atributo *reference*, que é opcional no RAS. Ressalta-se que a utilização do esquema de classificação do X-ARM não exclui a possibilidade de utilizar o esquema de classificação do RAS.

No X-ARM, o atributo *name* do elemento <*context*> deve possuir o valor "*X-ARM Application Areas*" para indicar a adoção da abordagem de classificação baseada em áreas de aplicação. No sentido de representar informações sobre áreas de aplicação, o elemento <*descriptor-group*> deve ser usado apenas na forma de filho do elemento <*context*>. No X-ARM, os atributos *name* e *reference* do elemento <*descriptor-group*> não são usados.

O elemento <*descriptor-group*> deve possuir, obrigatoriamente, um ou mais elementos <*descriptor*>, cada um deles indicando em seu atributo *name* uma determinada área de aplicação. Considerando que as áreas de aplicação são organizadas hierarquicamente em diversos níveis, o valor do atributo *name* do elemento <*descriptor*> é representado usando os nomes individuais das áreas de aplicação, que compõem a hierarquia, separados por pontos.

Além disso, como mencionado anteriormente, no caso de áreas de aplicação, o elemento <*descriptor-group*> deve possuir um único elemento <*related-asset*>, que referencia um *asset* do tipo *Application Areas*. Para indicar o tipo do relacionamento, o valor do atributo *relationship-type* do elemento <*related-asset*> deve ser "*appAreasSchema*".

A Figura 13 ilustra um trecho da descrição de um *asset*, em que é apresentada a classificação por áreas de aplicação. Como indicado no elemento <*related-asset*>, o esquema de classificação adotado é especificado no *asset* cujo identificador único é *br.areas-1.0*. No primeiro elemento <*descriptor*>, o valor "*Entertainment.Game.Computer*" representa uma hierarquia de áreas de aplicação, onde "*Entertainment*" é o nível mais alto e "*Computer*" é o nível mais baixo e mais especializado. Vale ressaltar que, neste exemplo, todas as áreas de aplicação indicadas devem ter sido previamente definidas no *asset* relacionado, denominado *br.areas-1.0*, que especifica o esquema de classificação adotado.

```
…
1. <classification>
…
2.   <context name="X-ARM Application Areas" id="19">
…
3.     <descriptor-group>
4.          <related-asset name="Areas" id="br.areas-1.0" relationship-type="appAreasSchema" required="false"/>
5.          <descriptor name="Entertainment.Game.Computer"/>
6.          <descriptor name="Entertainment.Music.MP3.Decoder"/>
7.     </descriptor-group>
8.   </context>
9. </classification>
…
```

Figura 13: Exemplo de classificação baseada em áreas de aplicação.

3.3.4 Controle de Visibilidade

Em determinadas situações, um produtor pode não desejar que seus *assets* possam ser recuperados por qualquer consumidor. Por exemplo, o produtor pode desejar que seus *assets* sejam reusados apenas por desenvolvedores do mesmo grupo de trabalho ou da

mesma empresa. Para limitar as possibilidades de recuperação, o X-ARM define o conceito de visibilidade, que permite a especificação dos domínios (produtores) que são permitidos ou negados. Desta forma, pode-se especificar que um dado *asset* apenas poderá ser recuperado por produtores cujos domínios estão explicitamente autorizados na descrição do *asset*. Neste sentido, o *X-ARM Artifact Profile* define o elemento <*visibility*>, que é exibido na Figura 14.

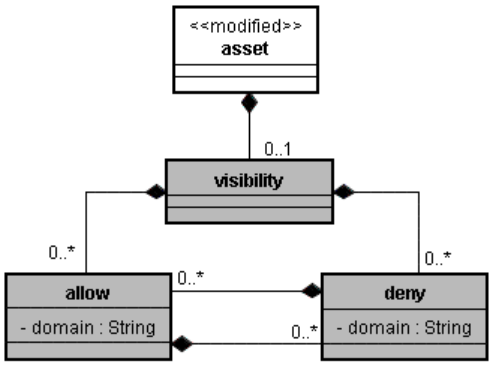

Figura 14: Representação do controle de visibilidade.

O elemento <*visibility*> possui os elementos <*allow*> e <*deny*>, cujos atributos *domain* permitem ao produtor especificar os domínios (produtores) que possuem ou não permissão para recuperar o *asset*. Qualquer domínio especificado nos elementos <*allow*> ou <*deny*> automaticamente aplica a visibilidade aos seus respectivos subdomínios, exceto quando tais subdomínios são explicitamente enumerados em outros elementos <*allow*> ou <*deny*>.

Na descrição de um determinado *asset*, se o termo <*visibility*> não está presente ou, quando presente, os termos <*allow*> ou <*deny*> não estão presentes, o respectivo *asset* pode ser recuperado por qualquer entidade (consumidor ou produtor). Entretanto, se algum termo <*allow*> ou <*deny*> está presente, as permissões seguem o que estiver especificado, sendo que domínios representados com o elemento <*allow*> estão autorizados e domínios representados com o elemento <*deny*> estão impedidos de recuperar o *asset*. Quando a visibilidade é especificada, apenas produtores podem recuperar o *asset*, ou seja, os consumidores estão automaticamente impedidos de recuperar o *asset*. No caso de existir apenas elementos <*deny*>, assume-se que quaisquer domínios não especificados são capazes de recuperar o *asset*. Por outro lado, no caso de existir apenas elementos <*allow*>,

os domínios não especificados são proibidos de recuperar o *asset*. A Figura 15 ilustra a utilização do elemento <*visibility*>.

```
…
1. <visibility>
2.     <allow domain="br.compose">
3.         <deny domain="br.compose.tmp"/>
4.     </allow>
5. </visibility>
…
```

Figura 15: Exemplo de controle de visibilidade.

No exemplo da Figura 15, o *asset* descrito somente pode ser recuperado por desenvolvedores que pertencem ao domínio *br.compose* ou aos seus respectivos subdomínios, exceto os desenvolvedores do subdomínio *br.compose.tmp*, que está explicitamente proibido. Neste caso, um desenvolvedor pertencente ao domínio *br.ufpb* também não será capaz de recuperar o *asset* descrito.

Vale ressaltar que, para que o controle de visibilidade seja efetivo, repositórios que provêem suporte ao X-ARM devem implementar mecanismos que permitam associar desenvolvedores a domínios. Além disso, em operações de recuperação de *assets*, o repositório deve prover mecanismos que permitam autenticar os desenvolvedores.

3.3.5 Histórico da Evolução do Asset

É conveniente que o consumidor possa saber de qual versão anterior um determinado *asset* evoluiu. Com isto, ele pode verificar se um *asset* está sendo melhorado continuamente, o que indica que seu produtor tem trabalhado nele e que possíveis problemas podem ser reparados em versões futuras. Um controle deste tipo é empregado pelo sistema *SourceForge* [SourceForge06], que para cada software disponibilizado, são apresentadas suas versões anteriores. Inclusive, a preocupação com a melhoria contínua dos projetos armazenados no *SourceForge* é evidente porque, para cada categoria de software, ele apresenta os projetos mais atuais. Além disto, outro benefício em utilizar um histórico de mudanças é permitir um controle por parte dos produtores sobre os *assets* que originaram outros, possibilitando a rastreabilidade da evolução.

Neste sentido, o *X-ARM Artifact Profile* adota o elemento <*change-history*>, apresentado na Figura 16, que tem a responsabilidade de representar as mudanças ocorridas

com relação às versões anteriores. Como pode ser observado, ao elemento <*change-history*>, pode-se opcionalmente associar uma descrição textual sobre a evolução ocorrida.

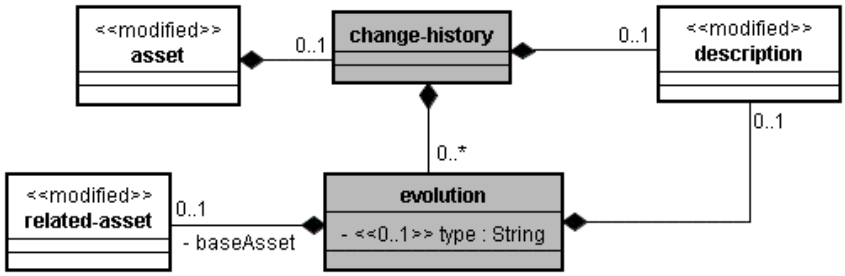

Figura 16: Representação do histórico de evolução.

O elemento <*change-history*> pode possuir ocorrências do elemento <*evolution*>, que indicam, com seus respectivos elementos <*related-asset*>, os *assets* a partir dos quais a versão atual foi gerada. Neste caso, em cada elemento <*related-asset*>, o atributo *relationship-type* deve possuir o valor "*baseAsset"*. Além disto, o elemento <*evolution*> também indica, usando o atributo opcional *type*, o tipo da evolução ocorrida. A Figura 17 ilustra a utilização do elemento <*change-history*>.

```
...
1. <change-history>
2.      <description>Versão do Modelo Conceitual de Negócios que foi refinada.</description>
3.      <evolution type="Adição de novos conceitos>
4.          <related-asset name="Modelo Conceitual" id="br.compose.mcn-1.0" relationship-type="baseAsset"
                    required="false"/>
5.      </evolution>
6. </change-history>
...
```

Figura 17: Exemplo de histórico de evolução.

A Figura 17 informa que o *asset* representado foi gerado a partir da adição de novos conceitos ao *asset* identificado por *br.compose.mcn-1.*0. Além disto, como este *asset* não é essencial para o entendimento do *asset* sob descrição, o valor do atributo *required* é "*false*".

3.3.6 Certificação

A fim de oferecer maior confiabilidade aos usuários de um repositório, é necessário que os *assets* armazenados apresentem um nível mínimo de qualidade, visto que o reuso tem como fundamento a utilização de *assets* desenvolvidos por terceiros. Neste sentido, torna-se importante o papel de entidades certificadoras, responsáveis por avaliar a qualidade dos *assets* [Wallnau04]. Estas entidades podem adotar diferentes metodologias de certificação,

cujas descrições podem ser expressas usando termos específicos definidos por cada entidade certificadora. Outra abordagem também adotada é a certificação do produtor, em que o processo de software adotado pelo produtor do *asset* deve atender uma série de requisitos impostos por uma entidade certificadora.

No sentido de representar a certificação dos *assets* e dos seus produtores, o X-ARM define o elemento <*certification-info*>, cuja estrutura é ilustrada na Figura 18. Este elemento é opcional, não devendo ocorrer em *assets* que não possuem qualquer tipo de certificação.

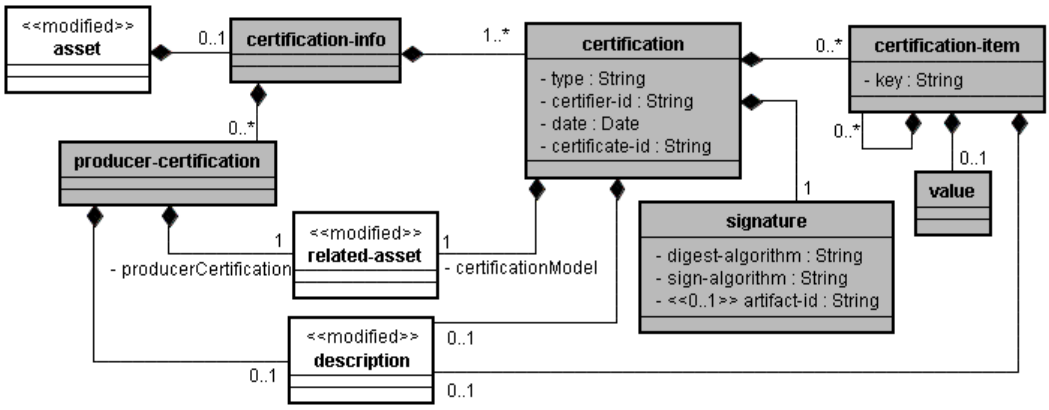

Figura 18: Representação de certificação.

Como pode ser observado na Figura 18, o elemento <*certification-info*> possui dois elementos: <*producer-certification*> e <*certification*>. O elemento <*producer-certification*> é usado para representar a certificação do processo de software adotado pelo produtor do *asset*. Por outro lado, o elemento <*certification*> é usado para representar a certificação do próprio *asset*.

Na representação da certificação do produtor, usando o elemento <*related-asset*>, o elemento <*producer-certification*> apenas referencia um *asset* do tipo *Producer Certification* do X-ARM *Artifact Profile*. No elemento <*related-asset*>, o atributo *relationship-type* deve possuir o valor "*producerCertification*". Além disso, opcionalmente, o elemento <*producer-certification*> pode possuir uma descrição textual explicativa, representada com o elemento <*description*>.

Na representação da certificação do próprio *asset*, usando o elemento <*related-asset*>, o elemento <*certification*> indica o modelo de certificação adotado, referenciando

um *asset* do tipo *Certification Model* do *X-ARM Model Profile*, descrito na Seção 3.4.3. Portanto, a certificação dos *assets* deve adotar um modelo de certificação. No elemento <*related-asset*>, o atributo *relationship-type* deve possuir o valor "*certificationModel"*.

No elemento <*certification*>, os atributos *type* e *date* identificam o tipo de modelo de certificação adotado e a data que foi realizada a certificação, respectivamente. O valor do atributo *type* deve estar definido no *asset* de especificação de modelo de certificação, indicado no elemento <*related-asset*>. Para permitir que um repositório possa autenticar uma certificação, o elemento <*certification*> define o atributo *certifier-id*, que é usado para representar o identificador único da respectiva entidade certificadora. Neste caso, entidades certificadoras podem estar registradas no repositório ou não, desde que possam ser identificadas de alguma forma, como por exemplo, por URLs. Desta forma, um repositório que suporte o X-ARM pode acessar os dados da entidade certificadora, inclusive a sua chave pública. De forma semelhante a outros elementos, o elemento <*certification*> pode possuir uma descrição textual para auxiliar no entendimento da certificação, que pode ser representada usando o elemento <*description*>.

Para assegurar que o certificado foi de fato emitido pela entidade certificadora, o X-ARM define o elemento <*signature*>, que contém uma assinatura digital do certificado, emitida pela própria entidade certificadora. No elemento <*signature*>, o atributo *digest-algorithm* indica o algoritmo de *hash* usado para gerar o *Digest* e o atributo *sign-algorithm* indica o algoritmo usado para gerar a assinatura digital.

A assinatura digital pode ser informada de duas formas. Na primeira, o atributo *artifact-id* referencia um arquivo existente dentro do pacote do *asset*, que possui a assinatura digital gerada pela entidade certificadora. Na segunda, a assinatura é representada como conteúdo do próprio elemento <*signature*>, no formato Base64.

A assinatura digital é gerada considerando todas as partes não modificáveis do manifesto, que são os elementos XML do manifesto que, se alterados, produzem uma nova versão do *asset*. Estas partes não modificáveis devem ser definidas pelo próprio repositório. No entanto, para exemplificar, é provável que um repositório que adote o X-ARM defina que informações sobre modelos de negócios podem ser modificadas sem gerar novas versões dos *assets*, afinal, o preço de um *asset* pode variar de acordo com fatores

econômicos. Além disto, para geração da assinatura digital também são considerados todos os arquivos indicados nos elementos <*artifact*> do manifesto, com exceção do arquivo que contém a própria assinatura, quando o mesmo existir. Neste caso, o nome do arquivo que contém a assinatura já deve ser previsto antes da assinatura ser criada, já que o manifesto indica o nome do arquivo de assinatura. Desta forma, os consumidores podem confiar que o certificado foi de fato emitido pela entidade certificadora, considerando todos os arquivos que compõem o *asset*.

Os detalhes da certificação são representados por uma hierarquia de elementos <*certification-item*>, que permitem representar diversos tipos de parâmetros de certificação. Para cada elemento <*certification-item*>, o atributo *key* permite a especificação de um parâmetro do modelo de certificação adotado, enquanto o elemento <*value*> representa o valor relacionado a este parâmetro. O atributo *key* identifica o nome do parâmetro e o atributo *value* indica o resultado da avaliação daquele parâmetro. Como mencionado anteriormente, os detalhes da certificação devem seguir um modelo de certificação, especificado como um *asset* do tipo *Certification Model* do *X-ARM Model Profile*.

De forma semelhante a outros elementos previamente discutidos, o elemento <*certification-item*> pode possuir uma descrição associada que auxilie no entendimento, especificada usando o elemento <*description*>.

A Figura 19 apresenta um exemplo de representação de certificação de *asset*.

```
…
1. <certification-info>
2.    <certification type="ISO 9126-based" certifier-id="br.compose.compsign" certificate-id="A" date="2005-10-05">
3.       <related-asset id="br.compsign.certification-1.0" name="ISO 9126-based" relationship-type="certificationModel"
                        required="true"/>
4.       <certification-item key="Accuracy">
5.          <certification-item key="Correctness">
6.             <value>87</value>
7.          </certification-item>
8.       </certification-item>
9.       <certification-item key="Security">
10.         <certification-item key="Data Encryption">
11.            <value>yes</value>
12.         </certification-item>
13.         <certification-item key="Controllability">
14.            <value>98</value>
15.         </certification-item>
16.         <certification-item key="Auditability">
17.            <value>yes</value>
18.         </certification-item>
19.      </certification-item>
20.      <signature digest-algorithm="SHA1" sign-algorithm="RSA">
                  e+1x6YuUMCEAP/Wvz2CwC0kKtOLgWvJaNVVfMLNuhYpp5DjXX/ZzPJwXec
                    nR/w7SjoMP55pkWh5jPzWBudsKMaA7f0bf3zO5+5NFQwb+p9OBuTXeChQ+cx/A5x
                    gxA9+q8mI7LoRbXHbEgyGgxHV/pSCiyiwH1N7y9b5ZixcrvUAIdp/XJBIB8QGiEE
                    AKEJFIyC9oBvoL+m0cnTdt7y+F1rQKDNYIGpDgNf+9uVlbGAKOJToU14U5TmwQmJ
                  Ogqx5vhr3HasII69ZdU8ijh5Hlucjzd/6KQgn1LbQT+oQzKTAB4T61FIlOf7SQZm
21.      </signature>
22.   </certification>
23. </certification-info>
```

Figura 19: Exemplo de certificação.

No exemplo apresentado na Figura 19, observa-se que o tipo de certificação adotado foi chamado *ISO 9126-based* e que a certificação foi realiza pela entidade certificadora que tem a identificação *br.compose.compsign*. O modelo de certificação adotado está especificado no *asset* identificado por *br.compsign.certification-1.0*. São incluídas algumas ocorrências do elemento <*certification-item*>, que descrevem a certificação do *asset*. Por fim, o elemento <*signature*> representa a assinatura digital emitida pela entidade certificadora, também neste elemento foram definidos os algoritmos utilizados para gerar a assinatura, neste caso o SHA1 e o RSA. Neste exemplo, apenas o *asset* foi certificado, e não o seu produtor também.

3.3.7 Informações de Negociação

Assim como na produção de qualquer tipo de software, quando um produtor desenvolve um *asset* para reuso por terceiros, normalmente ele está interessado em um retorno financeiro. Portanto, existe a necessidade de representação das formas de negociação que um produtor deseja para seus *assets*. A fim de atender este requisito, X-ARM define o elemento <*business-info*>, cujo elemento filho <*business*> representa os possíveis modelos de negócios. A estrutura do elemento <*business-info*> segue uma organização semelhante à

estrutura adotada para descrição de certificações, sendo ilustrada na Figura 20. É importante ressaltar que, se nenhuma informação de negócio estiver especificada, considera-se que o *asset* pode ser livremente recuperado.

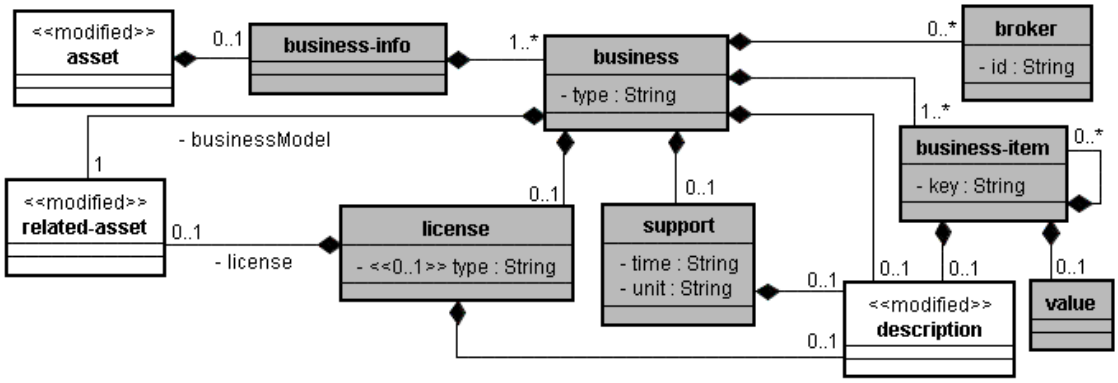

Figura 20: Representação de informações de negociação.

Para cada modelo de negócio (ocorrência do elemento <*business*>), podem ser especificados a licença, o tempo de suporte oferecido para o *asset*, as entidades de negociação autorizadas a negociar o *asset* e detalhes de negociação do *asset*. O elemento <*business*> contém o atributo obrigatório *type*, que tem a função de definir qual o modelo de negócios seguido, como, por exemplo, "*pay-per-copy*". Além disso, o elemento <*business*> possui uma descrição textual, especificada pelo elemento <*description*>.

Uma licença pode ser de um tipo específico, tal como GPL [GPL91], *Creative Commons* [CC06] ou qualquer outro. Para representar a licença de uso foi definido o elemento <*license*>, cujo atributo *type* é usado para indicar o tipo específico de licença. O elemento <*license*> pode referenciar um *asset* relacionado que detalha as características da licença. Vale ressaltar que o elemento <*related-asset*> permite referenciar um *asset* armazenado no repositório ou dentro do próprio pacote. Em ambos os casos, o atributo *relationship-type* do elemento <*related-asset*> deve possuir o valor "*license*". Como pode ser observado, o elemento <*license*> pode conter uma descrição textual usando o elemento <*description*>.

O tempo de suporte representa o intervalo de tempo no qual o produtor tem o compromisso de auxiliar o consumidor no uso do *asset*. Esta informação é representada por meio do elemento <*support*>, que possui os atributos *time* e *unit*. O primeiro indica a

quantidade de tempo após a aquisição do *asset* para o qual o suporte é oferecido, e o segundo indica a grandeza na qual este tempo é medido. Os valores permitidos para *unit* são "*days*", "*months*" ou "*years*". De forma complementar, o produtor pode descrever as características do suporte usando o elemento <*description*>.

As entidades de negociação autorizadas a negociar o *asset* são referenciadas pelo atributo *id* do elemento <*broker*>. Estas entidades podem ser cadastradas ou não, dependendo das políticas do repositório. Caso as entidades de negociação não sejam cadastradas, elas podem ser identificadas por uma URL, por exemplo. Estas entidades provêem serviços para negociar o *asset* com o consumidor, receber o pagamento pelo *asset*, recuperá-lo do repositório e entregá-lo corretamente ao consumidor que o adquiriu. Os serviços para negociação podem ser oferecidos por entidades especializadas em negociar *assets*, contratadas pelos produtores de *assets*, ou também podem ser implementados pelos próprios produtores de *assets*. Neste último caso, o elemento <*broker*> indica os identificadores dos próprios produtores.

Os detalhes de negociação são representados por uma hierarquia de elementos <*business-item*>, que permitem representar diversos tipos de informações, tais como: preço, formas de pagamento, quantidades de possíveis instalações, entre outros. Os detalhes de negociação devem seguir um modelo de negócio, especificado como um *asset* do tipo *Business Model* do *X-ARM Model Profile*, descrito na Seção 3.4.4. O modelo de negócio adotado é indicado usando o elemento <*related-asset*>, relacionado com o elemento <*business*>. Vale ressaltar que o atributo *relationship-type* do elemento <*related-asset*> deve possuir o valor "*businessModel*".

Para cada elemento <*business-item*>, o atributo *key* e o elemento *value* permitem a especificação de um item, termo ou detalhe do modelo de negócio adotado. O elemento *value* pode ter um único valor discreto, múltiplos valores discretos ou intervalos de valores numéricos. Múltiplos valores discretos são representados pelos valores discretos separados por vírgula. Como exemplo de múltiplos valores discretos podemos citar: "visa, mastecard, american express". Para representar um intervalo de valores numéricos, deve-se utilizar a seguinte notação: "*valor_mínimo..valor_máximo*". Por exemplo, o valor "1..10" representa um intervalo de 1 a 10. Para especificar um intervalo de valor ilimitado utiliza-se o termo "*". Por exemplo, o valor "1..*" representa um intervalo de 1 até infinito. No caso do tipo

string, o termo "*" indica que o item pode possuir qualquer valor. Também é possível combinar valores discretos, múltiplos valores discretos e intervalos de valores, como por exemplo: "1..10, 25, 30..40, 50, 60".

De forma semelhante a outros elementos previamente discutidos, o elemento <*business-item*> pode possuir uma descrição associada que auxilie no entendimento, especificada usando o elemento <*description*>.

A Figura 21 ilustra um exemplo de descrição das informações de negociação de um *asset*. O *asset* representado segue um modelo de negócios do tipo "*pay-per-copy*", que não é distribuído livremente.

```
…
1. <business-info>
2.    <business type="pay-per-copy">
3.       <related-asset relationship-type="businessModel" id="br.business.paypercopy-1.0" required="false"/>
4.       <business-item key="copies">
5.          <value>1..10</value>
6.          <business-item key="unitary price">
7.             <value>150,00</value>
8.          </business-item>
9.          <business-item key="currency">
10.            <value>R$</value>
11.         </business-item>
12.      </business-item>
13.      <business-item key="copies">
14.         <value>11..100</value>
15.         <business-item key="unitary price">
16.            <value>140,00</value>
17.         </business-item>
18.         <business-item key="currency">
19.            <value>R$</value>
20.         </business-item>
21.      </business-item>
22.      <business-item key="credit-card">
23.         <value>Visa</value>
24.      </business-item>
25.      <license type="GPL">
26.         <related-asset relationship-type="license" id="org.gnu.gpl -1.0" required="true"/>
27.      </license>
28.      <support time="2" unit="years"/>
29.      <broker id=”com.compsource”/>
30.   </business>
31. </business-info>
…
```

Figura 21: Exemplo de informações de negociação.

Na Linha 4, o elemento <*business-item*> e seus elementos filhos indicam que, para aquisições de 1 a 10 cópias, o preço unitário é de R$ 150,00. Na Linha 13, o elemento <*business-item*> e seus elementos filhos representam que, em aquisições de 11 a 100 cópias, o preço unitário é de R$ 140,00. Na Linha 22, o elemento <*business-item*> sinaliza que o cartão de crédito para pagamento é o *Visa*. Além disso, como indicado na Linha 25

pelo elemento <*license*>, o tipo da licença é GPL, cuja descrição se encontra em um arquivo dentro do pacote do *asset*. Neste exemplo, como especificado nas Linhas 28 e 29, o tempo de suporte é de dois anos e a entidade de negociação autorizada a negociar o *asset* é identificada por *com.compsource*.

Depois de apresentados os elementos introduzidos pelo *X-ARM Artifact Profile*, as próximas seções são dedicadas aos *profiles* criados a partir dele.

3.4 X-ARM Model Profile

O X-ARM tem o propósito de ser o mais flexível possível, de forma a suportar diversos processos de desenvolvimento, diversos conjuntos de áreas de aplicação, diversos modelos de certificação, de negócios e de componentes. Assim, não são definidos elementos e atributos XML específicos para representar as informações sobre um determinado tipo de modelo de negócios, como *pay-per-use* ou *pay-per-copy*, por exemplo. Neste caso, para cada tipo de modelo de negócios, deveria ser definida uma estrutura diferente, com novos elementos e atributos. Esta seria uma solução trivial para representação de modelos de negócios e outros tipos de informações. Neste exemplo, as informações envolvidas poderiam ser divididas em duas categorias: a estrutura, representada pelos nomes dos elementos e atributos, possivelmente definida em *XML Schema*; e os dados sobre o modelo de negócios, representados pelos valores dos elementos e atributos.

Por outro lado, X-ARM adota uma abordagem em que é definida mais uma categoria de informação para representação de aspectos como processos de desenvolvimento, áreas de aplicação e modelos de certificação, de negócios e de componentes. Esta categoria compreende os modelos, representados como *assets* do *X-ARM Model Profile*. Estes modelos especificam as informações que devem ser providas para descrição dos aspectos mencionados. Com esta abordagem, por exemplo, a estrutura de elementos para a especificação de informações sobre modelos de negócios de um *asset*, não inclui informações sobre um tipo de modelo de negócios específico, como *pay-per-copy* ou *pay-per-use*. A Figura 22 ilustra a utilização dos modelos descritos com o *X-ARM Model Profile*.

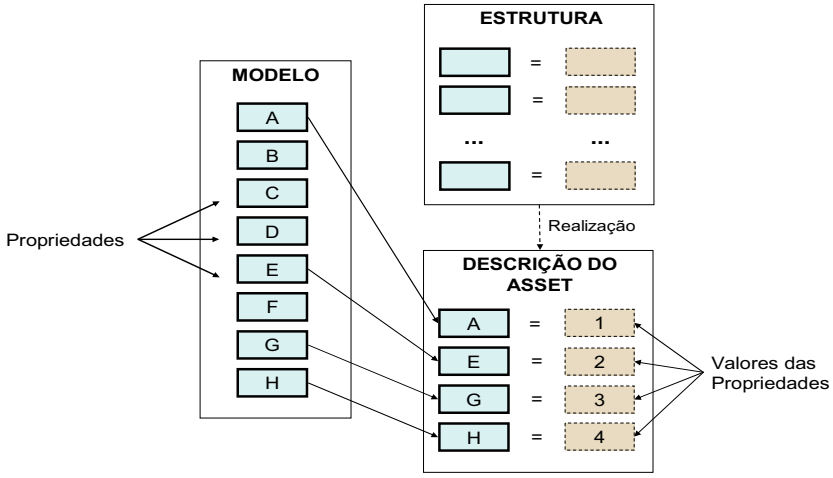

Figura 22: Ilustração da utilização de modelos.

A caixa identificada como "estrutura" refere-se aos elementos e atributos XML definidos com *XML Schema*, para representar a descrição dos *assets*. A caixa identificada como "modelo" representa um *asset* representado com *X-ARM Model Profile*, que define as propriedades que podem ou devem ser representadas nas descrições dos *assets* criados de acordo com aquele modelo. Por fim, a descrição de um *asset* deve seguir a estrutura definida pelo X-ARM e deve adotar um determinado modelo, representando as propriedades definidas por este modelo e informando os valores para estas propriedades. Nem todas as propriedades deverão ter valores associados, dependendo da especificação do modelo, ou do tipo de modelo. Por exemplo, modelos para especificação de áreas de aplicação não devem ter valores associados.

Em resumo, estes modelos são definidos com o intuito de padronizar os valores usados para representar características particulares, o que habilita buscas e comparações padronizadas entre diferentes *assets*, além de permitir tratamento apropriado de *assets* cujos conjuntos de valores para determinados elementos seguem as mesmas regras.

Produtores de *assets* podem definir os modelos com o *X-ARM Model Profile* que melhor se adeqüem às suas necessidades. Um exemplo do benefício desta abordagem é o caso dos modelos de negócios, que podem seguir sintaxes específicas, a fim de serem interpretadas adequadamente pelas ferramentas de negociação dos produtores. Por outro lado, um produtor também pode utilizar modelos já definidos por outros produtores de *assets*.

Em X-ARM, estes modelos são representados usando tipos de *assets* específicos, definidos no *X-ARM Model Profile*. Os *assets* de especificação de modelos podem ser definidos por qualquer produtor de *assets* ou algum outro ator relacionado ao repositório. Com exceção de modelos de componentes, os demais tipos de modelos já foram anteriormente citados na descrição do *X-ARM Artifact Profile*. A Figura 23 ilustra a organização completa dos elementos do *X-ARM Model Profile*.

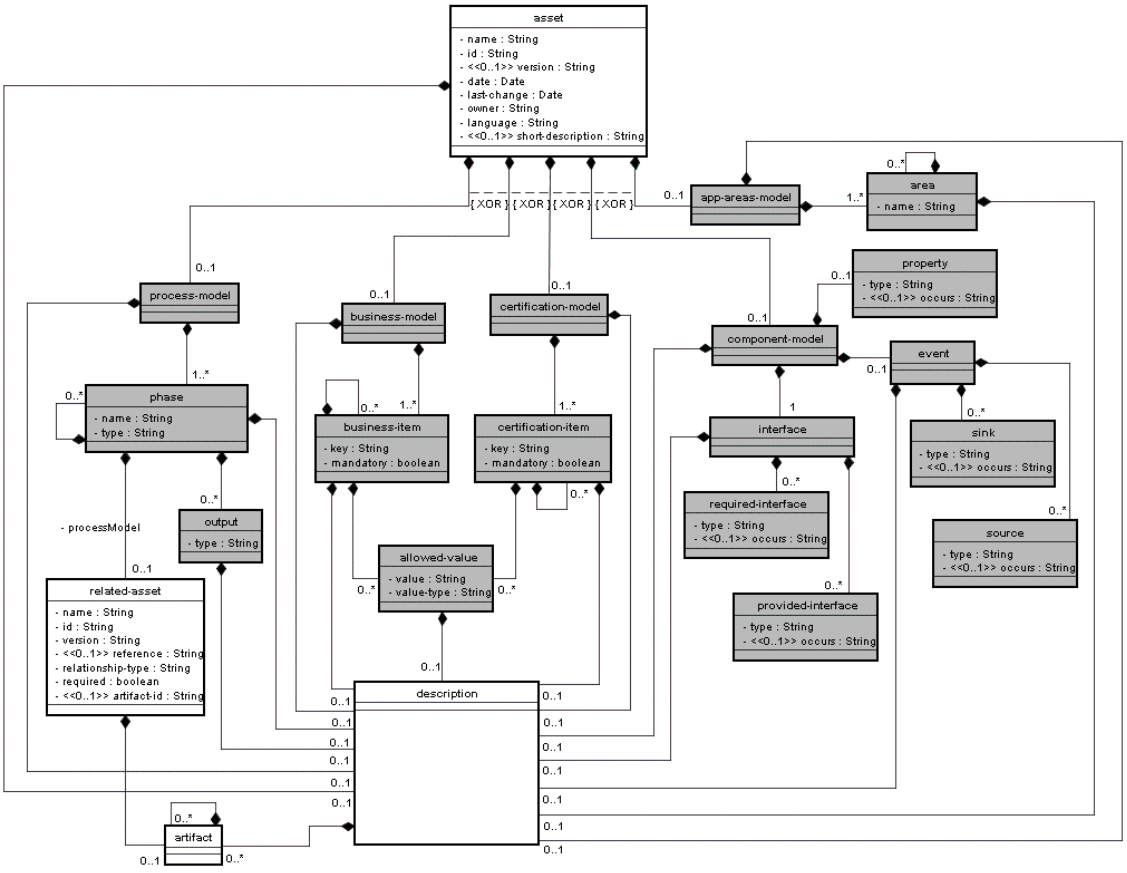

Figura 23: *X-ARM Model Profile*.

Em relação ao *X-ARM Artifact Profile*, o *X-ARM Model Profile* adiciona os elementos principais <*process-model*>, <*business-model*>, <*certification-model*>, <*component-model*> e <*app-areas-model*>.

3.4.1 Processos de Desenvolvimento

Diversos processos de desenvolvimento guiam o desenvolvimento de software baseado em componentes, tais como UML Components [Cheesman01], Catalysis [D'Souza99] e Kobra [Atkinson02]. A partir da avaliação destes processos, nota-se que em geral, eles são

organizados como conjuntos de fases, nas quais podem ser gerados um ou mais artefatos, de tipos determinados pelos processos. Assim, para especificação de processos de desenvolvimento com o X-ARM, são utilizados *assets* do tipo *Development Process*, descritos com *X-ARM Model Profile*. Estes *assets* permitem a validação das fases e tipos de artefatos, no *X-ARM Artifact Profile*, conforme mencionado nas Seções 3.3.1 e 3.3.2. A organização dos elementos para a especificação de processos de desenvolvimento no *X-ARM Model Profile* é apresentada na Figura 24.

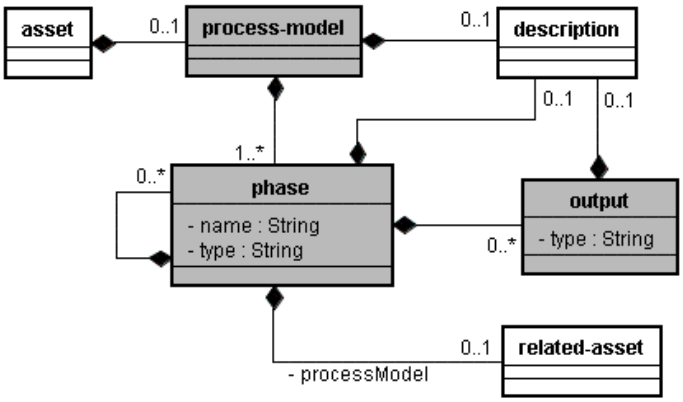

Figura 24: Especificação de processos de desenvolvimento.

Cada processo, representado pelo elemento <*process-model*>, tem um nome (atributo *name* do elemento <*asset*>) e é composto por uma ou mais fases hierárquicas, representadas por elementos <*phase*>. Cada fase possui um nome (*name*) e um tipo (*type*). O tipo é usado para indicar o nome específico adotado pelo processo para nomear aquele nível na hierarquia de fases. Por exemplo, no *UML Components*, as fases são denominadas *workflows* e as subfases são denominadas *stages*. Portanto, para o *workflow Specification*, o valor dos atributos *name* e *type* são *Specification* e *Workflow*, respecticamente. Por outro lado, para o estágio *Component Identification*, o valor destes atributos são *Component Identification* e *Stage*, respectivamente. Os elementos <*process-model*> e <*phase*> possuem o elemento <*description*> que pode conter um texto ou fazer referência a um artefato que descreva melhor estas informações.

Cada fase pode ter diversos tipos de artefatos de saída, representados pelo elemento <*output*>. O atributo *type* do elemento <*output*> indica o tipo do artefato. Cada elemento <*output*> também pode possuir uma descrição associada. Da mesma forma que o elemento

<*phase*>, a descrição do elemento <*output*> pode ser realizada usando o elemento <*description*>. Ressalta-se que é possível associar uma fase a outro processo, para isto utiliza-se o elemento <*related-asset*> com o valor "*processModel*" no atributo *relationship-type*.

A Figura 25 apresenta um trecho da especificação do processo *UML Components*. Este trecho representa o *workflow Requirements* com seus possíveis artefatos de saída (elementos <*output*>): "*Business Concept Model*" e "*Use Case Model*". Observe que, no elemento <*phase*>, o nome da fase (*name)* é "*Requirements*" e o tipo (*type*) é "*Workflow*". Além disso, o elemento <*artifact*>, interno ao elemento <*description*>, identifica um arquivo interno do pacote que contém uma descrição do *workflow Requirements*.

```
1. <asset id="br.process.umlc-2.0" name="UML Components"…>
2.  <process-model>
3.      <phase name="Requirements" type="workflow">
4.          <description>
5.              <artifact id="13" reference="workflows-Requirements.html" type="html"/>
6.          </description>
7.          <output type="Business Concept Model"/>
8.          <output type="Use Case Model"/>
9.      </phase>
…
10. </process-model>
…
```

Figura 25: Exemplo de especificação do processo *UML Components*.

De acordo com a Seção 3.3.1, para ser possível validar o tipo do *asset* em relação ao processo de desenvolvimento adotado, cada *asset* deve referenciar um *asset* do tipo *Development Process*, que descreve o processo de desenvolvimento adotado. Além disso, conforme apresentado na Seção 3.3.2, para ser possível a rastreabilidade, os *assets* gerados precisam referenciar os *assets* produzidos em momentos anteriores, além do *asset* que representa o processo. Portanto, a fim de auxiliar a compreensão dos relacionamentos entre os diferentes tipos de *assets*, a Figura 26 ilustra a representação destas associações usando trechos de descrições de *assets*. São utilizadas setas inteiras para apontar para o processo de desenvolvimento utilizado e, no caso de artefatos, também indicar os tipos de artefatos do processo a que eles se referem. Setas tracejadas são usadas para indicar as informações de rastreabilidade.

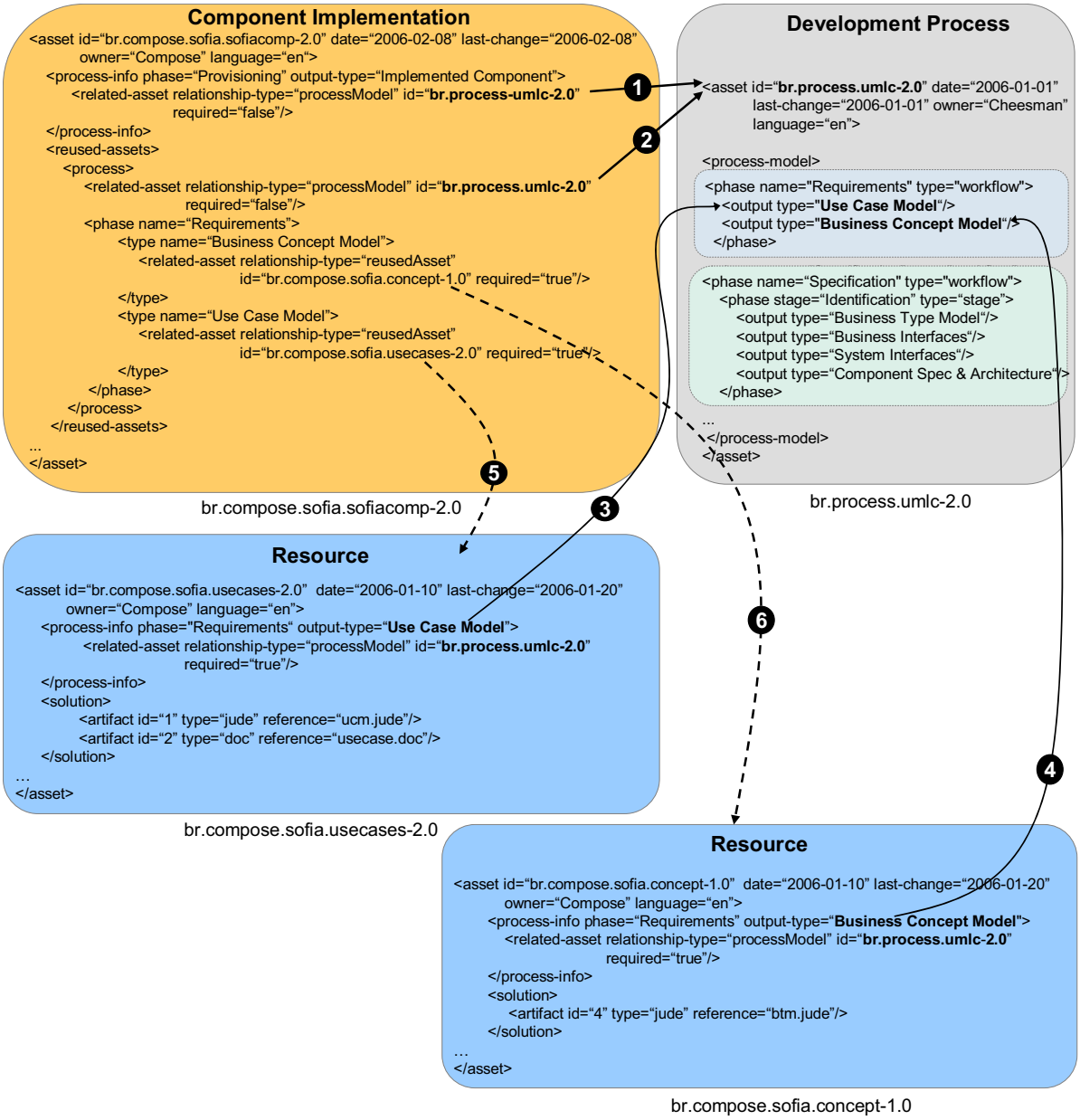

Figura 26: Exemplo de relacionamentos entre assets gerados em um processo.

Neste exemplo observa-se que, dentre os *assets* ilustrados, o *asset* do tipo especificação de processo de desenvolvimento deve ser o primeiro a ser especificado, pois os artefatos de software que surgirem posteriormente devem estar de acordo com o que ele especifica. Assim, os demais *assets* fazem referência à especificação do processo de desenvolvimento (setas 1, 2, 3 e 4). No caso dos artefatos, também é indicado (setas 3 e 4)

que ambos foram gerados na fase *Requirements* e que são dos tipos *Use Case Model* e *Business Concept Model*, respectivamente.

Para representar a rastreabilidade com a abordagem por projeto, é utilizada uma implementação de componente (identificada por *br.compose.sofia.sofiacomp-2.0*) que faz a referência aos outros três *assets*. Ela referencia (setas 1 e 2) a especificação do processo de desenvolvimento no sentido de informar qual processo foi seguido para a construção do componente. Esta implementação também referencia dois artefatos (setas 5 e 6) gerados durante o seu desenvolvimento de acordo com a fase do processo em que eles foram gerados.

3.4.2 Classificação Baseada em Áreas de Aplicação

A partir de um determinado modelo de áreas de classificação, especificado anteriormente em um *asset* do tipo *Application Areas*, para realizar a classificação de um *asset*, o produtor pode selecionar as respectivas áreas de aplicação, que mais se adequarem às características de um determinado *asset* sob descrição. Qualquer produtor de componentes ou administrador do repositório pode criar um esquema de classificação, isto é, um *asset* do tipo *Application Area*, que contém uma hierarquia de áreas de aplicação definidas pelo produtor.

No *X-ARM Model Profile*, o esquema de classificação é especificado usando o elemento <*app-areas-model*> e uma hierarquia de elementos <*area*>. A Figura 27 apresenta a estrutura de elementos para representação de áreas de aplicação. O elemento <*app-areas-model*> pode, opcionalmente, conter um elemento <*description*>, que mantém uma descrição do esquema de classificação.

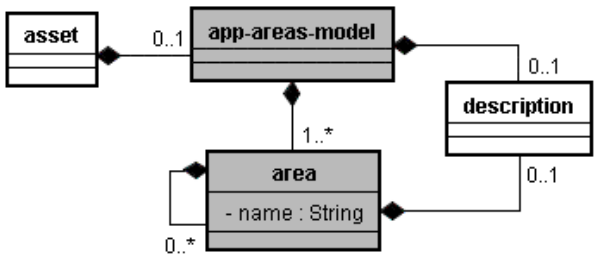

Figura 27: Especificação de áreas de aplicação.

A hierarquia de elementos <*area*> define a hierarquia de áreas de aplicação. Em cada elemento <*area*>, o atributo *name* contém o nome da área de aplicação naquele nível da hierarquia. Um exemplo de esquema de classificação é apresentado na Figura 28.

```
1. <asset id="br.compose.areas-1.0" name="Compose Areas"…>
2.   <app-areas-model>
3.     <area name="Entertainment">
4.       <area name="Music">
5.         <area name="MP3">
6.           <area name="Encoder"/>
7.           <area name="Decoder"/>
8.         </area>
9.       </area>
10.      <area name="Game">
11.        <area name="Computer"/>
12.        <area name="PDA"/>
13.      </area>
14.    </area>
…
15.  </app-areas-model>
…
16. </asset>
```

Figura 28: Exemplo de especificação de áreas de aplicação.

Opcionalmente, cada elemento <*area*> pode ter uma descrição associada, que, de forma semelhante ao elemento <*app-areas-model*>, pode ser realizada usando o elemento <*description*>.

3.4.3 Modelo de Certificação

Na Seção 3.3.6 foi apresentada a descrição de informações de certificação. Estas informações devem estar em conformidade com um modelo de certificação, definido como um *asset* do tipo *Certification Model* no *X-ARM Model Profile*. A estrutura de uma especificação da certificação de um *asset* é apresentada na Figura 29.

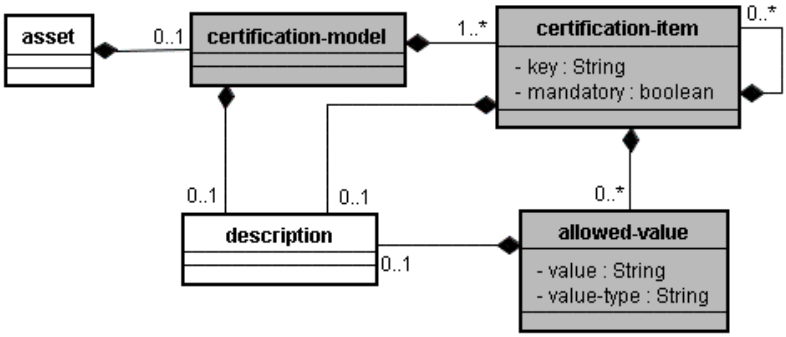

Figura 29: Especificação de modelos de certificação.

Para identificar o nome do modelo de certificação, atribuído pela entidade certificadora, é usado o atributo *name* do elemento <*asset*>. Da mesma forma que outros elementos apresentados anteriormente, o elemento <*certification-model*> pode conter uma descrição do modelo de certificação usando o elemento <*description*>.

O elemento <*certification-model*> também possui um ou vários itens de certificação, identificados pelo elemento <*certification-item*>, que podem estar estruturados de forma hierárquica. Em cada item, o atributo *key* identifica o nome de um parâmetro de certificação. Os itens de certificação que compõem um modelo de certificação podem ser obrigatórios ou opcionais. Para representar tal possibilidade, o atributo *mandatory* indica se o item especificado é obrigatório ou não em uma descrição da certificação de um *asset*. Da mesma forma que o elemento <*certification-model*>, o elemento <*certification-item*> pode ter uma descrição associada, especificada usando o elemento <*description*>.

Para cada item, opcionalmente, podem ser definidos os valores permitidos usando o elemento <*allowed-value*>, que possui os atributos *value* e *value-type*. O primeiro atributo indica um valor possível para o <*certification-item*> e o segundo atributo indica o tipo deste valor, sendo possível apenas os tipos *string*, *integer, unsigned integer, float* e *unsigned float*. Adotando uma sintaxe similar ao elemento <*business-item*> dos *assets* do tipo *Business Model*, o atributo *value* pode ter um único valor discreto, múltiplos valores discretos, intervalos de valores numéricos ou uma combinação destas formas.

Um exemplo da representação de modelo de certificação é apresentado na Figura 30. Este exemplo mostra um trecho da especificação de um modelo de certificação baseado no modelo ISO 9126 [ISO01], denominado *ISO 9126-based*, onde são definidos os itens *Accuracy*, *Security* e *Resource Behavior*. O primeiro é subdividido em *Correctness*, que tem como valores permitidos números inteiros entre 0 e 100. O item *Security* é subdividido em *Data Encryption*, *Controllability* e *Auditability*. *Data Encryption* e *Auditability* define os valores “yes” e “no” como permitidos. Já o item *Controllability* tem números inteiros entre 0 e 100 como permitidos. Por fim, o item *Resource Behavior* é subdividido nos itens *Memory Utilization* e *Disk Utilization*, que podem possuir valores inteiros.

```
1. <asset id=”br.compose.certification-1.0” name="ISO 9126-based" ... >
…
2.          <certification-model>
3.                  <certification-item key="Accuracy" mandatory="false">
4.                          <certification-item key="Correctness" mandatory="false">
5.                                  <allowed-value value="0..100" value-type="integer"/>
6.                          </certification-item>
7.                  </certification-item>
8.                  <certification-item key="Security" mandatory="false">
9.                          <certification-item key="Data Encryption" mandatory="false">
10.                                 <allowed-value value="yes" value-type="string"/>
11.                                 <allowed-value value="no" value-type="string"/>
12.                         </certification-item>
13.                         <certification-item key="Controllability" mandatory="false">
14.                                 <allowed-value value="0..100" value-type="integer"/>
15.                         </certification-item>
16.                         <certification-item key="Auditability" mandatory="false">
17.                                 <allowed-value value="yes" value-type="string"/>
18.                                 <allowed-value value="no" value-type="string"/>
19.                         </certification-item>
20.                 </certification-item>
21.                 <certification-item key="Resource Behavior" mandatory="false">
22.                         <certification-item key="Memory Utilization" mandatory="false">
23.                                 <allowed-value value="*" value-type="integer"/>
24.                         </certification-item>
25.                         <certification-item key="Disk Utilization" mandatory="false">
26.                                 <allowed-value value="*" value-type=" integer "/>
27.                         </certification-item>
28.                 </certification-item>
29.         </certification-model>
30. </asset>
```

Figura 30: Exemplo de especificação de modelo de certificação.

Um exemplo de utilização do modelo de certificação ilustrado na Figura 30 foi apresentado na Figura 19.

3.4.4 Modelos de Negócio

As informações de negociação especificadas nos *assets* devem seguir uma padronização, a fim de que as ferramentas que realizam a negociação dos *assets* possam interpretar os parâmetros e seus valores. Neste sentido, é adotada uma abordagem semelhante ao caso de informações de certificação, em que as informações de negociação especificadas nos *assets*, usando o elemento *<business>* do *X-ARM Artifact Profile*, devem estar aderir às informações especificadas um modelo de negócio (*asset* do tipo *Business Model* especificado pelo *X-ARM Model Profile*). Neste sentido, um modelo de negócio refere-se às formas pelas quais um *asset* pode ser negociado, adquirido e utilizado.

Os *assets* de especificação de modelos de negócio podem ser criados por qualquer produtor para representar os tipos de negociações e os parâmetros permitidos, com seus possíveis valores. Este tipo de *asset* permite especificar informações para representar modelos de negócios permitidos, a serem adotados por qualquer *asset* descrito com *X-ARM*.

Assim, as ferramentas para negociação precisam conhecer os parâmetros definidos nas instâncias do *X-ARM Model Profile* para que possam interpretar corretamente os dados especificados para negociação dos *assets*.

A estrutura de elementos para especificação de modelos de negócio do *X-ARM Model Profile* é apresentada na Figura 31.

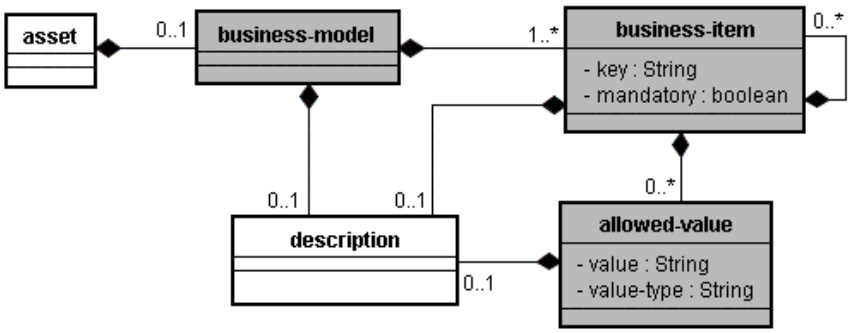

Figura 31: Especificação de modelos de negócios.

Para informar o nome do modelo de negócio adotado é usado o atributo *name* do elemento <*asset*>. O elemento <*business-model*> pode conter uma descrição associada, que, de forma semelhante a outros elementos, pode ser especificada usando o elemento <*description*>.

O elemento <*business-model*> também pode possuir um ou vários itens de negociação, identificados pelo elemento <*business-item*>, que podem estar estruturados de forma hierárquica. Em cada item, o atributo *key* define o nome de um item de negócio que pode existir em uma descrição de *asset*. Por exemplo, possíveis valores para o atributo *key* podem ser: "*price*" e "*creditcard*". Os itens de negócio que compõem um modelo de negócio podem ser obrigatórios ou opcionais. Para representar tal possibilidade, o atributo *mandatory* indica a obrigatoriedade ou não da existência do respectivo item na descrição de um *asset*. Da mesma forma que o elemento <*business-model*>, o elemento <*business-item*> pode ter uma descrição associada, especificada usando o elemento <*description*>.

Para cada item, opcionalmente, podem ser definidos os valores permitidos usando o elemento <*allowed-value*>, que possui os atributos *value* e *value-type*. O primeiro atributo indica um valor possível para o respectivo elemento <*business-item*> e o segundo atributo indica o tipo deste valor, sendo possível apenas os tipos *string*, *integer, unsigned integer*,

float e *unsigned float*. O atributo *value* pode ter um único valor discreto, múltiplos valores discretos ou intervalos de valores numéricos. Múltiplos valores discretos são representados pelos valores discretos separados por vírgula, por exemplo: "*visa, mastecard, american express*". Para representar um intervalo de valores numéricos, deve-se utilizar a seguinte notação: "*valor_mínimo..valor_máximo*". Por exemplo, o valor "1..10" representa um intervalo de 1 a 10. Para especificar um intervalo de valor ilimitado utiliza-se o termo "*". Por exemplo, o valor "1..*" representa um intervalo de 1 até infinito. No caso do tipo *string*, o termo "*" também indica que o item pode possuir qualquer valor. Por fim, é possível combinar valores discretos, múltiplos valores discretos e intervalos de valores, como por exemplo: "1..10, 25, 30..40, 50, 60".

A Figura 32 ilustra a especificação de um modelo de negócio. Neste exemplo é definido o modelo de negócio "*pay-per-copy*". Para este tipo de modelo de negócio, as descrições de *assets* que adotam este modelo devem informar a quantidade de cópias desejadas (que pode ser um valor entre 1 e 100), o preço unitário e a moeda utilizada. Para o modelo "*pay-per-copy*" é definido ainda que os cartões de crédito que podem ser especificados para pagamento dos *assets* são *Visa* e *MasterCard*.

```
1. <asset id="br.compose.ppc-1.0" name="pay-per-copy" …>
2.     <business-model>
3.         <business-item key="copies" mandatory="true">
4.             <allowed-value value="1..100" value-type="integer"/>
5.             <business-item key="unitary price" mandatory="true">
6.                 <allowed-value value="0..*" value-type="unsigned float"/>
7.             </business-item>
8.             <business-item key="currency" mandatory="true">
9.                 <allowed-value value="R$" value-type="string"/>
10.                <allowed-value value="US$" value-type="string"/>
11.            </business-item>
12.        </business-item>
13.        <business-item key="credit-card" mandatory="true">
14.            <allowed-value value="Visa, Mastercard" value-type="string"/>
15.        </business-item>
16.    </business-model>
…
```

Figura 32: Exemplo de especificação de modelo de negócio.

Um exemplo de utilização do modelo de negócio ilustrado na Figura 32 foi apresentado na Figura 21.

3.4.5 Modelos de Componentes

Além dos modelos de componentes existentes atualmente, tais como CCM [CCM 06], *JavaBeans* [Sun97], EJB [DeMichiel01], COM [Brooke01] e .NET [Brooke01], outros

modelos podem ser desenvolvidos no futuro, ou mesmo, novas características podem ser agregadas aos modelos já existentes. Neste sentido, existe a necessidade de que o modelo de descrição do X-ARM seja genérico a ponto de conseguir representar diferentes modelos de maneira única. Além disso, o modelo de descrição deve suportar a evolução dos modelos de componentes e permitir a extração de informações referentes a características particulares dos mesmos. A extração de informações pode ser bastante útil para engenhos de buscas.

Para atender esta necessidade, o X-ARM define uma abordagem onde os modelos de componentes têm suas características descritas separadamente em relação às descrições de componentes particulares, sendo realizado com instâncias do *X-ARM Model Profile*. Desta forma, na descrição de um componente usando o *X-ARM Component Profile*, é necessário indicar o modelo de componentes adotado, definido como um *asset* do tipo *Component Model* no *X-ARM Model Profile*. A utilização da especificação de um modelo de componentes em uma descrição de componente será discutida na apresentação do *X-ARM Component Profile* na Seção 3.6.

Conforme indicado na Figura 33, o *X-ARM Model Profile* define o elemento <*component-model*>, cujo propósito é permitir a especificação de modelos de componentes.

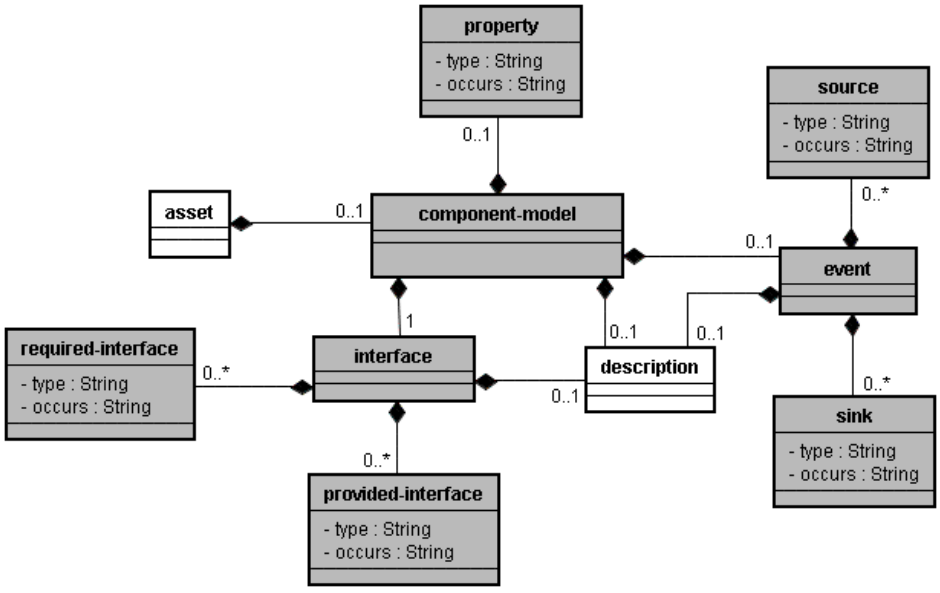

Figura 33: Especificação de modelos de componentes.

O elemento <*component-model*> pode possuir uma descrição associada para auxiliar na compreensão do modelo de componentes. Como em diversos elementos anteriormente apresentados, esta descrição pode ser representada usando o elemento <*description*>.

Para representar as formas de interação dos componentes com outras unidades de software, o elemento <*component-model*> pode possuir os seguintes elementos: <*interface*>, <*event*> e <*property*>. O elemento <*interface*> descreve os possíveis tipos de interfaces que um modelo pode definir. Os elementos <*required-interface*> e <*provided-interface*> são usados para distinguir os tipos de interfaces requeridas e providas, respectivamente.

Em cada elemento <*required-interface*>, o atributo *type* indica o nome adotado pelo modelo de componente para identificar um determinado tipo de interface requerida. Por exemplo, no caso da descrição de CCM, as interfaces requeridas são denominadas receptáculos (*receptacles*). Desta forma, na especificação do modelo CCM, o valor do atributo *type* do elemento <*required-interface*> deve ser "*receptacle*".

Similarmente, em cada elemento <*provided-interface*>, o atributo *type* indica o nome adotado pelo modelo de componente para identificar um determinado tipo de interface provida. Por exemplo, em uma descrição para o modelo CCM teríamos como interface provida o tipo faceta (*facet)*. Sendo assim o valor do atributo *type* no elemento <*provided-interface*> seria "*facet*".

O elemento <*event*> descreve os possíveis tipos de eventos que um modelo de componente pode suportar. Os elementos <*source*> e <*sink*> são usados para distinguir os tipos de eventos gerados e recebidos, respectivamente. Nestes elementos, o atributo *type* indica o nome adotado pelo modelo de componentes para identificar um determinado tipo de evento gerado ou recebido. Apesar dos termos *sink* e *source* serem adotados por vários modelos, como por exemplo, no modelo de componentes CCM, o atributo *type* continua sendo necessário para dar suporte a modelos que não adotem estes termos.

Além de interfaces e eventos, um modelo de componentes também pode prover propriedades. O elemento <*component-model*> representa propriedades usando o elemento <*property*>. Neste elemento, o atributo *type* é usado para representar o nome adotado pelo modelo de componentes para indicar um determinado tipo de propriedade. Por exemplo, no

CCM, as propriedades são denominadas *attributes*. Assim, na especificação do CCM, o valor do atributo *type* do elemento <*property*> deve ser "*attributes*".

Os elementos <*provided-interface*>, <*required-interface*>, <*sink*>, <*source*> e <*property*> também possuem o atributo *occurs*, que é responsável por definir a quantidade de ocorrências possíveis para o tipo definido no elemento. Por exemplo, no modelo de componentes EJB, todo componente deve prover uma interface *home*. Desta forma, na especificação da interface *home* do modelo EJB, o valor do atributo *type* do elemento <*provided-interface*> deve ser "*home*", e o atributo *occurs* deve possuir o valor "1", indicando que existe apenas uma única interface *home* para cada componente. Ressalta-se que o atributo *occurs* possui uma sintaxe similar ao atributo *value* do elemento <*business-item*>, definido na seção 3.4.4. Ou seja, o atributo *occurs* pode ter um único valor discreto, múltiplos valores discretos, intervalos de valores numéricos ou uma combinação destas formas, podendo inclusive adotar o termo "*" para indicar ocorrências ilimitadas. Por exemplo, os valores "*" e "1..*" representam qualquer número inteiro e qualquer número inteiro positivo, respectivamente.

A Figura 34 ilustra um exemplo da especificação do CCM. Observe que existem dois tipos de interfaces providas: *component* e *facet*. No entanto, todas as interfaces requeridas são denominadas *receptacles*. Além disso, em CCM, eventos gerados e recebidos são denominados *source* e *sink*, respectivamente. Por fim, as propriedades são denominadas *attributes*.

```
1. <asset name="CORBA Component Model" id="br.omg.ccm-3.0" owner="OMG" date="2006-01-02">
...
2.    <component-model>
3.       <interface>
4.          <provided-interface type="component" occurs="1"/>
5.          <provided-interface type="facet" occurs="0..*"/>
6.          <required-interface type="receptacle" occurs="0..*"/>
7.       </interface>
8.       <event>
9.          <sink type="sink" occurs="0..*"/>
10.         <source type="source" occurs="0..*"/>
11.      </event>
12.      <property type="attribute" occurs="0..*"/>
13.   </component-model>
14. </asset>
```

Figura 34: Especificação do CCM com *X-ARM Model Profile*.

3.5 X-ARM Interaction Profile

A representação de interfaces, eventos e exceções é realizada usando o *X-ARM Interaction Profile*. Este *profile* adiciona elementos ao *X-ARM Artifact Profile*, conforme ilustrado na Figura 35. Além dos elementos adicionados, apenas os elementos do *X-ARM Artifact Profile* que foram alterados ou estão relacionados com os elementos do *X-ARM Interaction Profile* são explicitados.

No X-ARM, as descrições de interfaces, eventos e exceções podem ser representadas em dois níveis de abstração. No nível independente de modelo de componente, a descrição não considera qualquer característica específica do modelo de componente que será adotado. Por outro lado, no nível dependente de modelo de componente, a descrição representa características específicas do modelo de componente adotado. Por exemplo, na descrição dependente de modelo, uma interface CCM possui um arquivo CIDL [OMG01] associado. Na Figura 35, os elementos com o estereótipo <<*dependent*>> são responsáveis pela distinção entre descrições dependentes e independentes de modelo de componente. Além deste, também é possível notar na figura, a existência dos estereótipos <<*interface*>> e <<*event*>>, que indicam que os elementos associados são utilizados apenas nas descrições de interfaces e eventos, respectivamente. Assim, por exemplo, o elemento <*throws*> não é usado no caso de exceções, e o elemento <*pre-condition*> é usado apenas em descrições de interfaces.

O *X-ARM Interaction Profile* define três elementos principais: <*interface*>, <*exception*> e <*event*>. Estes três elementos definem diferentes tipos de *assets*, e, portanto, não podem coexistir em uma mesma descrição de *asset*. As representações de interfaces, eventos e exceções, dependentes e independentes de modelo de componente, são detalhadas nas próximas subseções.

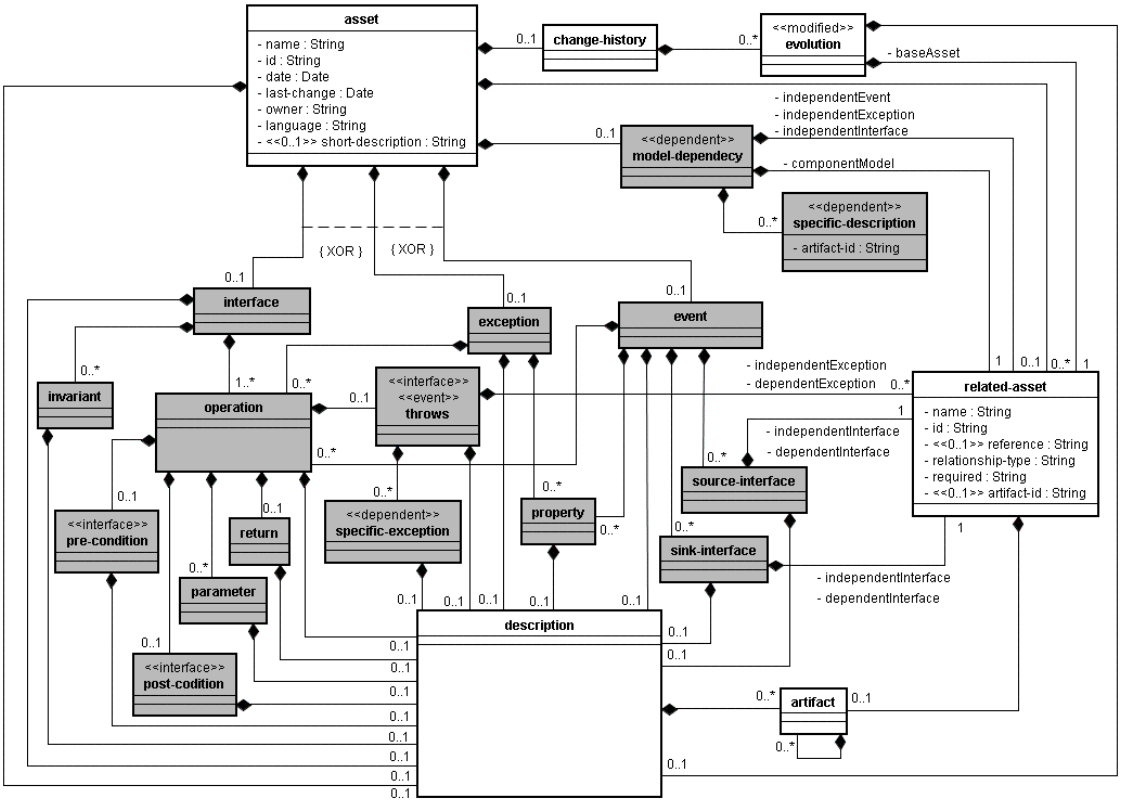

Figura 35: *X-ARM Interaction Profile*.

Além disso, o *X-ARM Interaction Profile* altera o elemento <*evolution*>, originalmente definido no *X-ARM Artifact Profile*. Como mostrado na Figura 36, no elemento <*evolution*>, o atributo *type* passa a ser obrigatório e o atributo opcional *changed-unit* é adicionado.

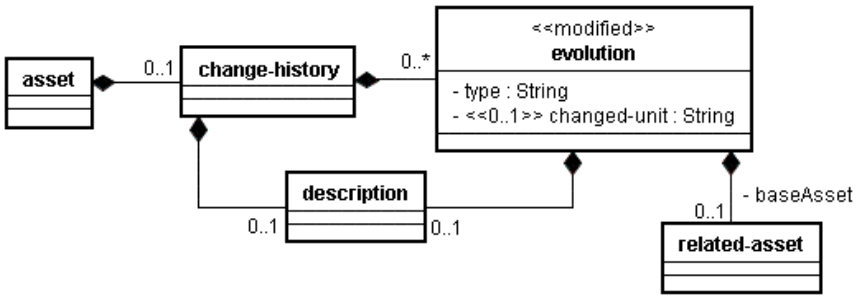

Figura 36: Elementos para representação de evolução.

Neste *profile*, o atributo *type* de <*evolution*> pode ter apenas valores pré-definidos, que são dependentes do tipo de *asset* descrito.

No caso de interfaces, os valores pré-definidos são:

- "*operation addition*": Indica a adição de uma operação de uma interface em relação à versão anterior da mesma interface.
- "*operation removal*": Indica a remoção de uma operação de uma interface.
- "*exception addition*": Indica a adição de uma exceção para uma operação de uma interface.
- "*exception removal*": Indica a remoção de uma exceção para uma operação de uma interface.
- "*others*": Qualquer outro tipo de modificação não especificado.

No caso de eventos, os valores pré-definidos são:

- "*sink interface addition*": Indica a adição de uma interface que especifica as operações relacionadas à recepção do evento.
- "*sink interface removal*": Indica a remoção de uma interface que especifica as operações relacionadas à recepção do evento.
- "*source interface addition*": Indica a adição de uma interface que especifica as operações relacionadas à fonte do evento.
- "*source interface removal*": Indica a remoção de uma interface que especifica as operações relacionadas à recepção do evento.
- "*eventData change*": Indica a alteração da estrutura de dados ou objeto que representa os dados de eventos em relação à versão anterior do mesmo evento.
- "*others*": Qualquer outro tipo de modificação não especificado.

Por fim, no caso de exceções, os valores pré-definidos são:

- "*operation addition*": Indica a adição de uma operação de uma exceção em relação à versão anterior da mesma exceção.
- "*operation removal*": Indica a remoção de uma operação de uma exceção.
- "*others*": Qualquer outro tipo de modificação não especificado.

O atributo *changed-unit* é usado para indicar a parte do *asset* que foi alterada. Por exemplo, no caso de uma interface cujo atributo *type* possui o valor "*operation addition*", o atributo *changed-unit* pode indicar a operação que foi adicionada à versão anterior da interface, representada pelo elemento <*related-asset*> associado ao elemento <*evolution*>. Conforme já informado na Seção 3.3.5, neste caso, o atributo *relationship-type* de <*related-asset*> tem o valor "*baseAsset*".

3.5.1 Interfaces

No *X-ARM Interaction Profile*, o elemento <*interface*> é usado para especificar uma interface, descrevendo as suas operações, invariantes, pré-condições e pós-condições. O elemento <*interface*> permite especificar interfaces independentes e dependentes de modelo de componente com estruturas bastante similares.

3.5.1.1 Interface Independente (*Independent Interface)*

A Figura 37 ilustra a estrutura adotada pelo elemento <*interface*> para especificar uma interface de forma independente de modelo de componente. Como pode ser observado, adotando uma abordagem similar a outros elementos, o elemento <*interface*> pode possuir uma descrição textual adicional, que pode ser representada usando o elemento <*description*>.

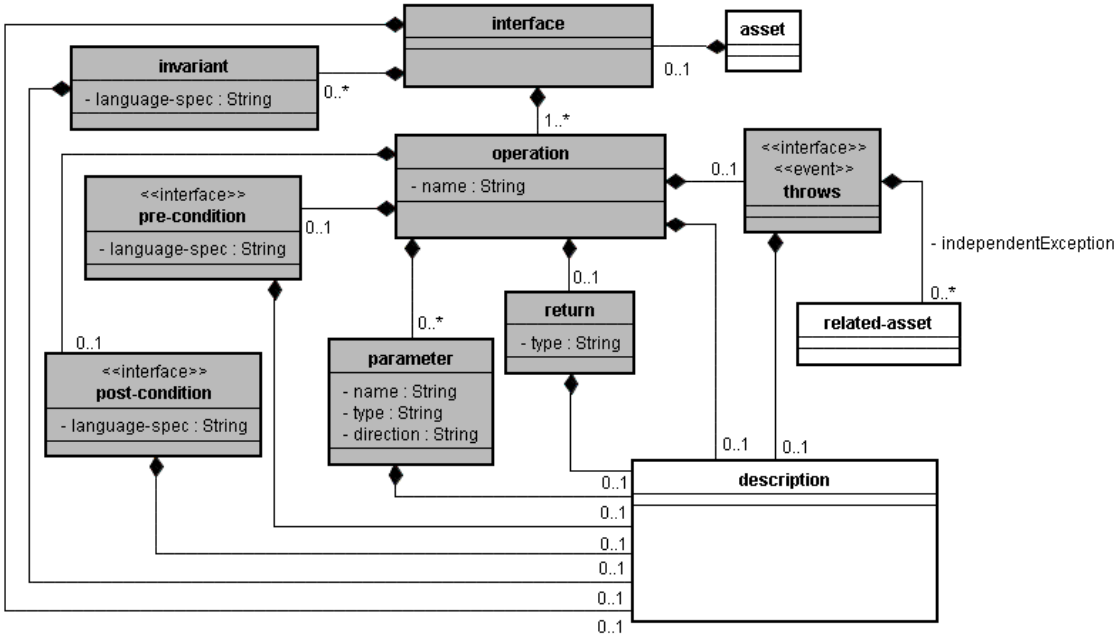

Figura 37: Especificação de interfaces independentes de modelo.

Interfaces podem ter invariantes associadas, representadas pelo elemento <*invariant*>. Cada invariante é descrita usando uma determinada linguagem, representada pelo atributo *language-spec*. A descrição da invariante pode ser representada usando o elemento <*description*>.

Cada interface pode possuir uma ou mais operações, definidas pelo elemento <*operation*>. Cada operação possui um nome, um conjunto de parâmetros e um valor de retorno. O nome de uma operação é especificado pelo atributo *name*. Cada parâmetro da operação é especificado no elemento <*parameter*>, cujos atributos *name*, *type* e *direction* representam o nome, o tipo de dado e a direção (*in*, *out* ou *inout*) do parâmetro, respectivamente. O valor de retorno de uma operação é especificado pelo elemento <*return*>, cujo atributo *type* representa o tipo de dado retornado. Observe que os elementos <*operation*>, <*parameter*> e <*return*> podem possuir descrições textuais associadas, que podem ser representadas usando o elemento <*description*>.

Alem disso, cada operação também pode gerar exceções. O elemento <*throws*> agrupa as possíveis exceções geradas pela operação. Considerando que exceções são *assets*, cada exceção gerada é representada por um elemento <*related-asset*>, que indica o *asset* que descreve a respectiva exceção. Como se trata de uma interface independente de modelo de componente, o elemento <*related-asset*> também deve referenciar uma exceção independente de modelo. Neste caso, o valor do atributo *relationship-type* é "*independentException*". Para facilitar a compreensão, o elemento <*throws*> pode possuir uma descrição associada, representada pelo elemento <*description*>.

A cada operação, também podem ser associadas uma pré-condição e uma pós-condição, representadas pelos elementos <*pre-condition*> e <*post-condition*>, respectivamente. Nestes elementos, o atributo *language-spec* indica a linguagem usada para a especificação da pré-condição ou pós-condição. A descrição de pré-condições e pós-condições também é realizada usando o elemento <*description*>.

A Figura 38 ilustra a especificação de uma interface independente de modelo de componente. Neste exemplo, a interface possui quatro operações (*getCustomerMatching, createCustomer, getCustomerDetails* e *notifyCustomer*). A operação *getCustomerDetails* é especificada com pré e pós- condições.

```
...
1. <interface>
2.    <description> Interface principal de gerenciamento de clientes.<description>
3.    <operation name="getCustomerMatching">
4.          <return type="Integer">
5.               <description>Retorna 0 se o cliente foi achado. Retorna 1 se não existe o cliente. Retorna 2 se o código não
                              foi fornecido e através do nome retornou mais de um cliente.</description>
6.          </return>
7.          <parameter name="custD" type="CustomerDetails" direction="out"/>
8.          <parameter name="cusId" type="int" direction="in"/>
9.    </operation>
10.    <operation name="createCustomer">
11.          <description>Insere um cliente na base de dados.</description>
12.          <return type="Boolean"/>
13.          <parameter name="custD" type="CustomerDetails" direction="in" />
14.          <parameter name="cusId" type="int" direction="out" />
15.    </operation>
16.    <operation name="getCustomerDetails">
17.          <description>Retorna os dados de um cliente.</description>
18.          <return type="CustomerDetails"/>
19.          <parameter name="cus" type="int" direction="in"/>
20.          <pre-condition language-spec="OCL">
21.               <description> customer->exists(c, c.id = cus) </description>
22.          </pre-condition>
23.          <post-condition language-spec="OCL">
24.               <description> Let theCust = customer->select(c,  c.id  = cus) in
                                   result.name = theCust.name and
                                   result.postCode = theCust.postCode and
                                   result.email = theCust.email
25.               </description>
26.          </post-condition>
27.     </operation>
28.     <operation name="notifyCustomer">
29.          <description>Envia uma mensagem de notificação para um cliente.</description>
30.              <return type="void"/>
31.              <parameter name="cus" type="CustId" direction="in"/>
32.              <parameter name="msg" type="String" direction="in"/>
33.     </operation>
34. </interface>
...
```

Figura 38: Exemplo de especificação de interface independente de modelo.

3.5.1.2 Interface Dependente (*Dependent Interface)*

Geralmente, uma especificação de interface dependente de modelo é baseada em uma especificação de interface independente de modelo e acrescenta as características específicas do modelo de componentes adotado. Desta forma, a especificação de uma interface dependente de modelo preserva a estrutura e a semântica do elemento <*interface*>, mas adiciona os elementos <*model-dependency*>, <*specific-description*> e <*specific-exception*> para representar as características do modelo de componentes adotado. A Figura 39 ilustra a estrutura adotada pelo elemento <*interface*> para especificar uma interface de forma dependente de modelo de componente.

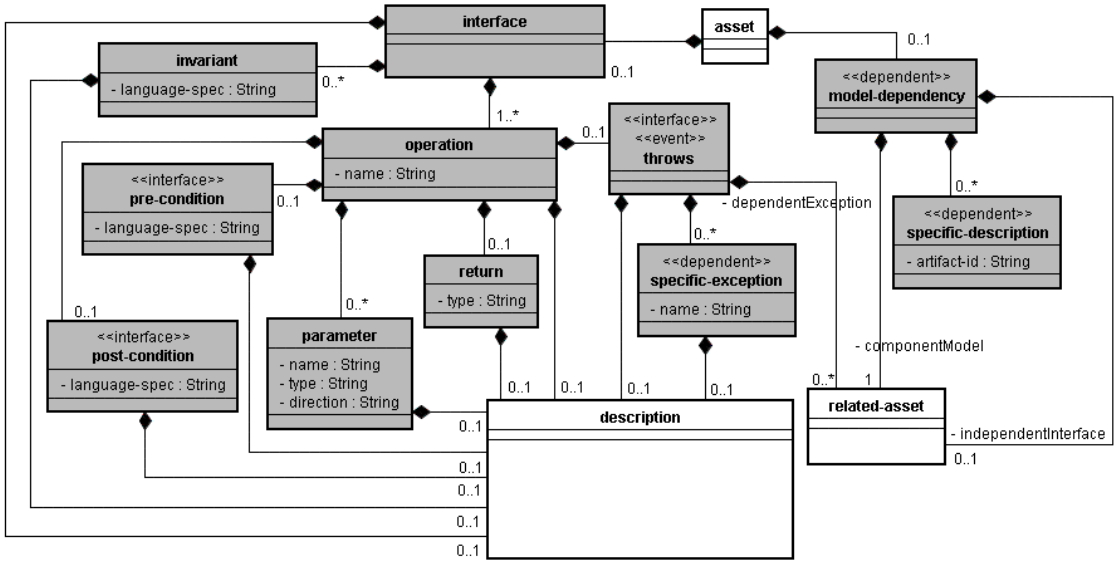

Figura 39: Especificação de interfaces dependentes de modelo.

O elemento <*model-dependency*> indica o modelo de componente adotado pela interface. Além disso, este elemento também referencia a respectiva interface independente de modelo, da qual a interface dependente é uma especialização. Em ambos os casos, o elemento <*model-dependency*> referencia um elemento <*related-asset*>. No primeiro caso, o valor do atributo *relationship-type* deve ser "*componentModel*". No segundo caso, o valor deste atributo deve ser "*independentInterface*".

Geralmente, nos diversos modelos de componentes, as interfaces são descritas em arquivos usando uma linguagem definida pelos seus respectivos modelos. No X-ARM, estes arquivos de descrição de interfaces são representados no elemento <*specific-description*>. Neste elemento, o arquivo de descrição de interface é indicado com atributo *artifact-id*, que contém o identificador de um elemento <*artifact*> do próprio manifesto, que indica o nome do arquivo interno do pacote que contém a descrição da interface.

No X-ARM, exceções dependentes de modelo devem ser especificadas como *assets*. No entanto, no caso de modelos de componentes cujas exceções ainda não tenham sido especificadas na forma de *assets*, o X-ARM permite que tais exceções sejam indicadas textualmente, apenas como forma de documentação. Nestes casos, o elemento <*specific-exception*> deve ser usado para representar as exceções específicas do modelo.

No elemento <*specific-exception*>, o atributo *name* indica o nome da exceção e o elemento <*description*> inclui um texto explicativo. Por exemplo, em EJB, a exceção *Remote Exception* pode ser representada por um elemento <*specific-exception*>, cujo valor do atributo *name* é "*RemoteException*".

Vale ressaltar que o elemento <*specific-exception*> deve apenas ser usado enquanto as exceções específicas do modelo não são especificadas no X-ARM. No entanto, uma vez que as mesmas já tenham sido especificadas como *assets*, os produtores devem referenciá-las no elemento <*throws*> como *assets* relacionados, indicados no elemento <*related-asset*>. Esta abordagem permite a adoção do X-ARM antes que as exceções específicas de modelos de componentes tenham sido especificadas.

3.5.2 Eventos

O mecanismo de eventos permite que dados trafeguem assincronamente entre componentes [Sun97]. Assim, um componente fonte dispara eventos quando determinadas condições são alcançadas. Por exemplo, após uma mudança de estado monitorada, o componente fonte emite um evento ao componente receptor informando a mudança ocorrida. No *X-ARM Interaction Profile*, assume-se que a emissão e a recepção de eventos são realizadas com interfaces, conforme ilustrado na Figura 40.

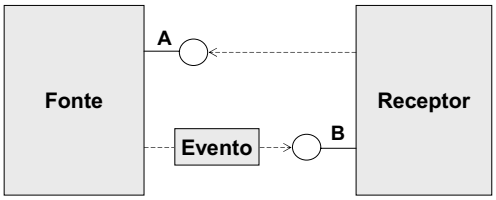

Figura 40: Notificação através de eventos.

A interface de um componente fonte é necessária para padronizar as operações a serem chamadas quando um outro componente deseja se tornar um receptor ou deixar de sê-lo. Na Figura 40, a interface *A* é provida pelo componente fonte e contém as operações para adição e remoção de componentes receptores. Por outro lado, a interface de um componente receptor é usada para representar as operações de *callback*, chamadas pelo componente fonte quando o evento ocorre. Na Figura 40, a interface *B* é provida pelo componente receptor e contém as operações de *callback* chamadas pelo componente fonte para notificá-lo da ocorrência do evento. Nas operações de *callback*, o evento é informado

como uma estrutura de dados ou um objeto, que contém informações que descrevem o evento ou operações que permitem a manipulação do evento.

No *X-ARM Interaction Profile*, o elemento <*event*> é usado para especificar um evento, permitindo descrever as suas informações e operações associadas, como também, indicar as interfaces que devem ser providas por componentes fonte e receptor. De forma similar ao elemento <*interface*>, o elemento <*event*> também permite especificar eventos independente e dependente de modelo de componente.

3.5.2.1 Evento Independente (*Independent Event)*

A Figura 41 ilustra a estrutura adotada pelo elemento <*event*> para especificar um evento de forma independente de modelo de componente. Como pode ser observado, adotando uma abordagem similar a outros elementos, o elemento <*event*> pode possuir uma descrição textual adicional, que pode ser representada usando o elemento <*description*>.

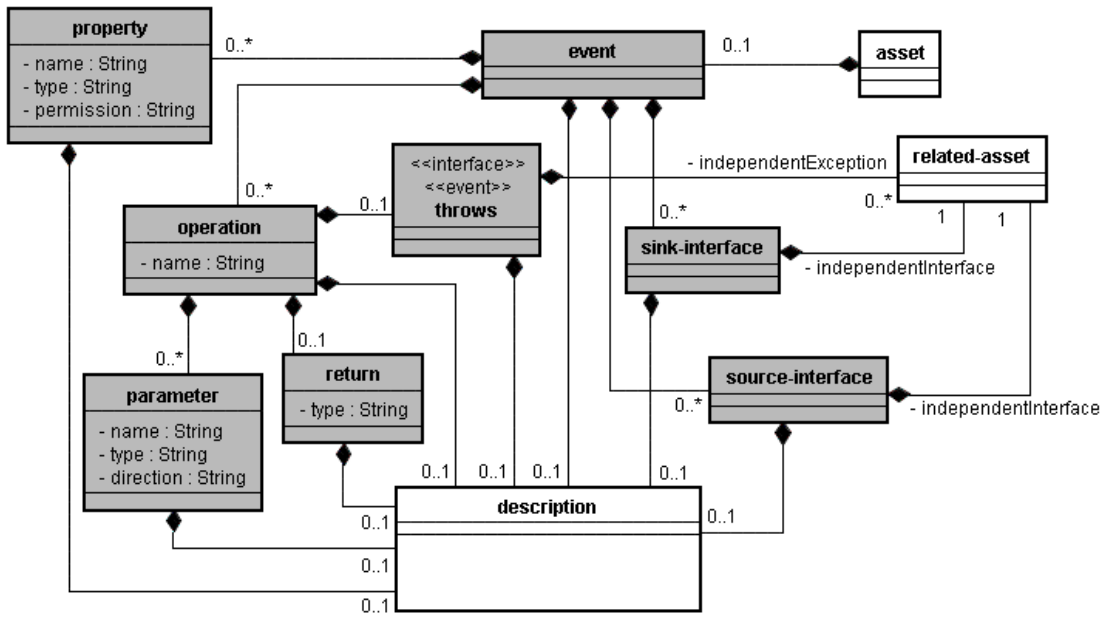

Figura 41: Especificação de eventos independentes de modelo.

Na descrição de um evento, as interfaces providas pelos componentes fonte e receptor são indicadas pelos elementos <*source-interface*> e <*sink-interface*>, respectivamente. Em ambos os elementos, as interfaces devem ser independentes de modelo de componente. Para referenciar as interfaces, os elementos <*source-interface*> e

<*sink-interface*> adotam o elemento <*related-asset*>, cujo valor do atributo *relationship-type* deve ser "*independentInterface*".

Como pode ser observado na Figura 41, os elementos <*source-interface*> e <*sink-interface*> são opcionais, a fim de permitir a representação de eventos que não se baseiem na existência de interfaces tanto para a fonte quanto para o receptor de eventos. De acordo com a figura, podem existir diversas ocorrências dos elementos <*source-interface*> e <*sink-interface*>. Esta abordagem é utilizada para permitir que um mesmo evento possa ser gerado ou recebido por diferentes interfaces, incrementando o reuso da especificação do evento. Além disso, estes elementos podem possuir uma descrição associada, representada pelo elemento <*description*>.

Como apresentado na Figura 41, as informações do evento são representadas como propriedades usando o elemento <*property*>. Por outro lado, as operações do evento são representadas pelo elemento <*operation*>.

Em cada propriedade, os atributos *name* e *type* do elemento <*property*> representam o nome e o tipo de dado associado à propriedade. Além disso, o atributo *permission* indica como a propriedade pode ser acessada: somente leitura (*read*), escrita (*write*) ou leitura e escrita (*readwrite*).

A especificação de operações de um evento adota uma estrutura similar àquela usada na especificação de operações de uma interface. No entanto, a especificação de interfaces inclui os elementos <*pre-condition*> e <*post-condition*>, que não são relevantes no contexto de eventos. Deste modo, os mesmos não devem ser usados na especificação de operações de um evento, e, por isso, não são apresentados na Figura 41.

Como ilustrado na Figura 41, os elementos <*property*>, <*operation*>, <*parameter*> e <*return*> podem possuir uma descrição associada, representada pelo elemento <*description*>.

A Figura 42 apresenta um exemplo de especificação de evento independente de modelo de componente. Este exemplo descreve um evento que provê a operação *getEmailId*. Os componentes fonte e receptor devem implementar as interfaces *br.compose.IReceivedEmailSource-1.0* e *br.compose.IReceivedEmailListener-1.0*.

```
…
1. <event>
2.    <description>Evento de notificação de  recebimento de novo email.</description>
3.    <operation name="getEmailId">
4.       <return type="Integer"/>
5.    </operation>
6.    <source-interface>
7.       <related-asset name="IReceivedEmailSource" id="br.compose.IReceiveEmailSource-1.0"
                          relationship-type="interface" required="true"/>
8.    </source-interface>
9.    <sink-interface>
10.       <related-asset name="IReceivedEmailListener" id="br.compose.IReceiveEmailListener-1.0"
                          relationship-type="interface" required="true"/>
11.   </sink-interface>
12. </event>
…
```

Figura 42: Exemplo de especificação de evento independente de modelo.

3.5.2.2 Evento Dependente (*Dependent Event)*

A especificação de eventos dependentes de modelo de componente adota uma abordagem similar à especificação de interfaces dependentes de modelo de componente. Portanto, uma especificação de evento dependente de modelo é baseada em uma especificação de evento independente de modelo e acrescenta as características específicas do modelo de componentes adotado. Desta forma, a especificação de um evento dependente de modelo preserva a estrutura e a semântica do elemento <*event*>, mas adiciona os elementos <*model-dependency*>, <*specific-description*> e <*specific-exception*> para representar as características do modelo de componentes adotado. A Figura 43 ilustra a estrutura adotada pelo elemento <*event*> para especificar um evento de forma dependente de modelo de componente.

O elemento <*model-dependency*> indica o modelo de componente adotado pelo evento. Além disso, este elemento também referencia o respectivo evento independente de modelo, do qual o evento dependente é uma especialização. Em ambos os casos, o elemento <*model-dependency*> utiliza um elemento <*related-asset*>. No primeiro caso, o valor do atributo *relationship-type* deve ser "*componentModel*". No segundo caso, o valor deste atributo deve ser "*independentEvent*".

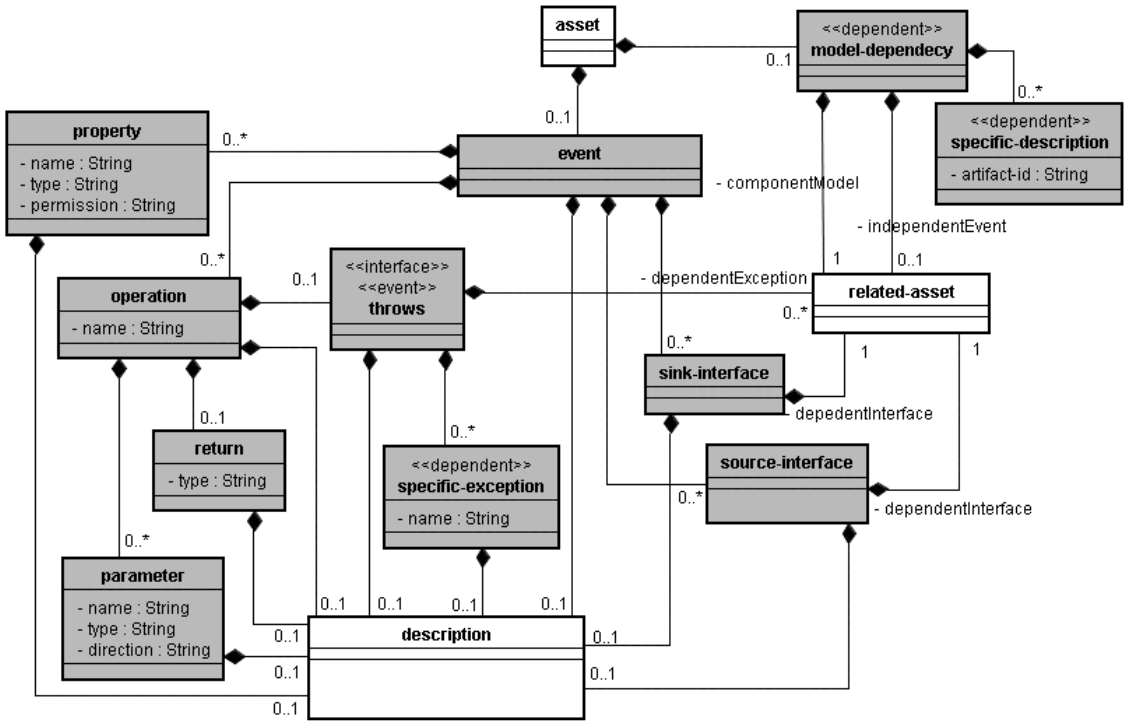

Figura 43: Elementos para especificação de eventos dependentes de modelo.

Em alguns modelos de componentes, os eventos também são descritos em arquivos usando uma linguagem definida pelos seus respectivos modelos. No X-ARM, estes arquivos de descrição de eventos são representados no elemento <*specific-description*>. Neste elemento, o arquivo de descrição do evento deve ser indicado pelo atributo *artifact-id* que contém o identificador de um elemento <*artifact*> do próprio manifesto, indicando o nome do arquivo dentro do pacote que contém a descrição do evento.

Considerando que os eventos podem prover operações, em alguns modelos de componentes, tais operações podem gerar exceções. Novamente, adotando uma abordagem similar a interfaces, o X-ARM permite que exceções específicas de modelos de componentes sejam indicadas textualmente, apenas como forma de documentação. Nestes casos, o elemento <*specific-exception*> deve ser usado para representar as exceções específicas do modelo. No elemento <*specific-exception*>, o atributo *name* indica o nome da exceção e o elemento <*description*> inclui uma pequena descrição da exceção.

Vale ressaltar que o elemento <*specific-exception*> deve apenas ser usado enquanto as exceções específicas do modelo não são especificadas no X-ARM. No entanto, uma vez

que as mesmas já tenham sido especificadas como *assets*, os produtores devem referenciá-las no elemento <*throws*> como *assets* relacionados, indicados no elemento <*related-asset*>. Esta abordagem permite a adoção do X-ARM antes que as exceções específicas de modelos de componentes tenham sido especificadas.

3.5.3 Exceções

Exceções são geradas quando algo não ocorre como esperado na execução de uma operação, permitindo representar informações sobre o erro ocorrido. Em linguagens de programação orientadas a objeto, exceções são normalmente expressas na forma de objetos. Assim, exceções podem possuir operações e propriedades. Como várias interfaces podem reusar as mesmas exceções, a especificação de exceções também é realizada na forma de *assets*.

No *X-ARM Interaction Profile*, o elemento <*exception*> é usado para especificar uma exceção, permitindo descrever as suas propriedades e operações associadas. De forma similar aos elementos <*interface*> e <*event*>, o elemento <*exception*> também permite especificar exceções independente e dependente de modelo de componente.

3.5.3.1 Exceção Independente (Independent Exception)

A Figura 44 ilustra a estrutura adotada pelo elemento <*exception*> para especificar uma exceção de forma independente de modelo de componente. Como pode ser observado, o elemento <*exception*> pode possuir uma descrição textual adicional, representada usando o elemento <*description*>.

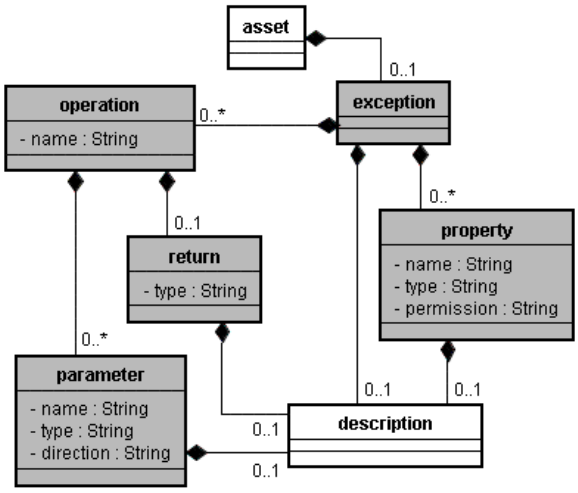

Figura 44: Especificação de exceções independentes de modelo.

As propriedades e operações associadas à exceção são representadas pelos elementos <*property*> e <*operation*>, respectivamente. Em cada propriedade, os atributos *name* e *type* do elemento <*property*> representam o nome e o tipo de dado associado à propriedade. Além disso, o atributo *permission* indica como a propriedade pode ser acessada: somente leitura (*read*), escrita (*write*) ou leitura e escrita (*readwrite*).

A especificação de operações de uma exceção adota uma estrutura similar àquela usada na especificação de operações de um evento. No entanto, a especificação de eventos inclui o elemento <*throws*>, que não é relevante no contexto de exceções. Consequentemente, o elemento <*throws*> não deve ser usado na especificação de operações de uma exceção, e, portanto, não é apresentado na Figura 44.

Como ilustrado na Figura 44, os elementos <*property*>, <*operation*>, <*parameter*> e <*return*> podem possuir uma descrição associada, representada pelo elemento <*description*>.

3.5.3.2 Exceção Dependente (Dependent Exception)

A especificação de exceções dependentes de modelo de componente adota uma abordagem similar à especificação de eventos dependentes de modelo de componente. Portanto, uma especificação de exceção dependente de modelo é baseada em uma especificação de exceção independente de modelo e acrescenta as características específicas do modelo de componentes adotado. Desta forma, a especificação de uma exceção dependente de modelo preserva a estrutura e a semântica do elemento <*exception*>, mas adiciona os elementos <*model-dependency*> e <*specific-description*> para representar as características do modelo de componentes adotado. A Figura 45 ilustra a estrutura adotada pelo elemento <*exception*> para especificar uma exceção de forma dependente de modelo de componente.

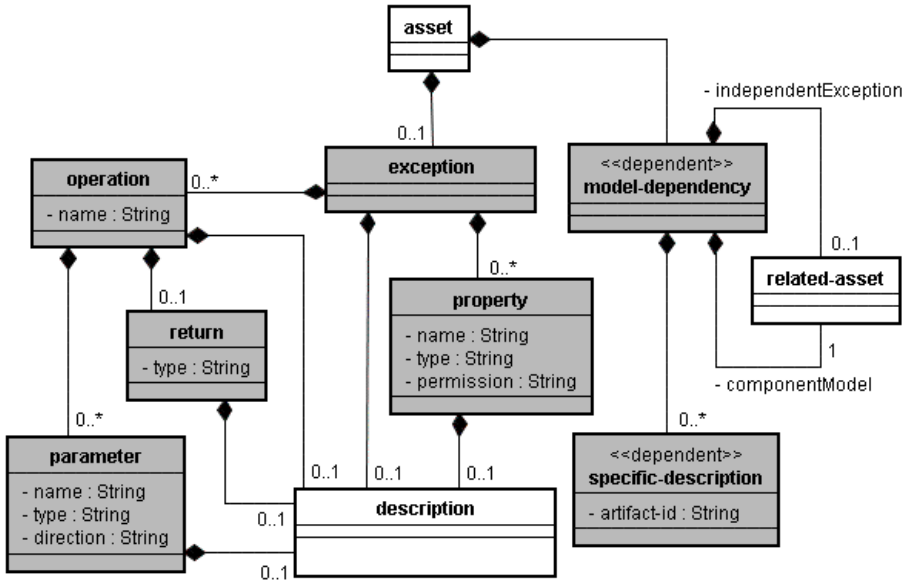

Figura 45: Especificação de exceções dependentes de modelo.

O elemento <*model-dependency*> indica o modelo de componente adotado pela exceção. Além disso, este elemento também referencia a respectiva exceção independente de modelo, da qual a exceção dependente é uma especialização. Em ambos os casos, o elemento <*model-dependency*> utiliza um elemento <*related-asset*>. No primeiro caso, o valor do atributo *relationship-type* deve ser "*componentModel*". No segundo caso, o valor deste atributo deve ser "*independentException*".

Embora ainda não seja amplamente adotado por modelos de componentes, exceções também podem ser descritas em arquivos usando uma linguagem definida pelos seus respectivos modelos. Desta forma, para contemplar modelos que venham a prover esta característica, estes arquivos de descrição de exceções são representados no elemento <*specific-description*>. Neste elemento, o arquivo de descrição da exceção deve ser indicado pelo atributo *artifact-id* que contém o identificador de um elemento <*artifact*> do próprio manifesto, indicando o nome do arquivo interno do pacote que contém a descrição da exceção.

3.6 X-ARM Component Profile

Em DBC, o desenvolvimento de um componente e o seu reuso são tipicamente, mas nem sempre, realizados por diferentes organizações. Assim, o sucesso desta abordagem depende

diretamente do sucesso da escolha dos componentes que devem atender os requisitos necessários para o uso em um determinado contexto e da qualidade dos componentes escolhidos.

Baseado nestas premissas, além do *X-ARM Component Profile* herdar todos os elementos definidos no *X-ARM Artifact Profile*, ele oferece suporte para representar ou referenciar características particulares que possam ser apresentadas por um componente, tais como interfaces, eventos, documentação, código fonte e código executável.

Diferente do *Component Profile* do RAS, em que as interfaces geralmente são especificadas juntamente com a descrição do componente, o *X-ARM Component Profile* provê uma abordagem que busca um menor grau de acoplamento entre os mecanismos de interação (interfaces e eventos). Além disso, o X-ARM também realiza a distinção entre especificações e implementações de componentes. Conjuntamente, o X-ARM proporciona maior reusabilidade dos *assets* envolvidos. Para isto, conforme já apresentado na Seção 3.1.1, o *X-ARM Component Profile* define três tipos de *assets*: *Independent Component Specification* (Especificação de Componente Independente), *Dependent Component Specification* (Especificação de Componente Dependente) e *Component Implementation* (Implementação de Componente). A organização dos elementos do *X-ARM Component Profile* é apresentada na Figura 46, onde os elementos usados apenas por especificações de componentes dependentes de modelos são representados com o estereótipo *<<dependent>>*.

O *X-ARM Component Profile* define dois elementos principais: *<component-specification>* e o *<component-implementation>*. Estes dois elementos definem diferentes tipos de *assets*, e, portanto, não podem coexistir em uma mesma descrição de *asset*. As representações de especificações de componentes (independentes e dependentes de modelo) e implementações de componentes são detalhadas nas próximas subseções.

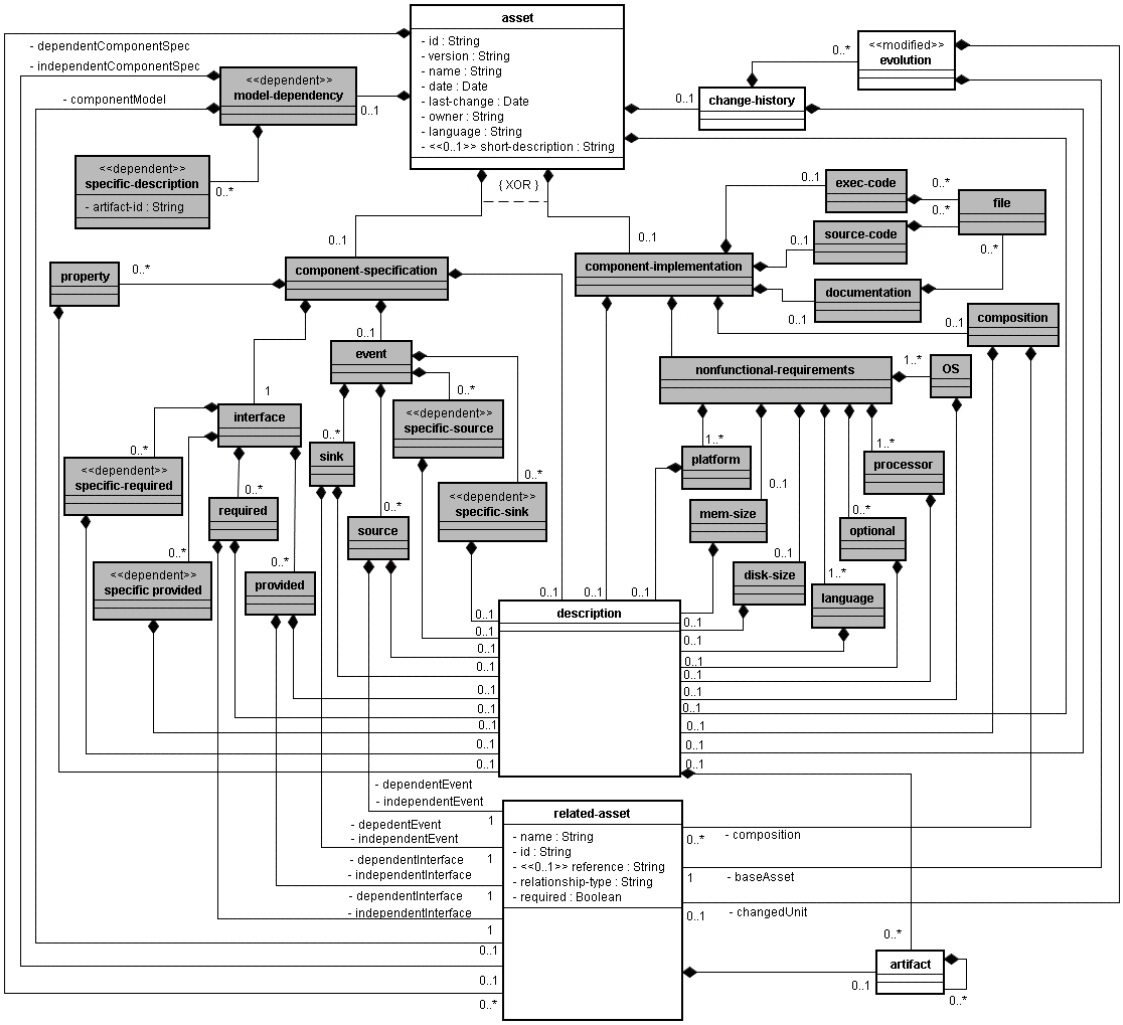

Figura 46: *X-ARM Component Profile*.

No *X-ARM Component Profile*, os elementos para controle de evolução foram alterados em relação ao *X-ARM Artifact Profile* de forma a representar aspectos particulares de especificações e implementações de componentes. Como mostrado na Figura 47, no elemento <*evolution*>, foi adicionado o atributo opcional *changed-unit* e o atributo *type* passou a ser obrigatório. Além disto, o elemento <*evolution*> agora pode ter outro elemento <*related-asset*>, além do identificado pelo relacionamento do tipo "*baseAsset*".

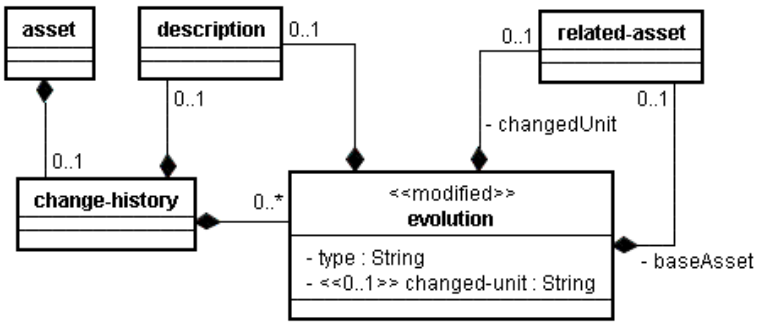

Figura 47: Controle de evolução no *X-ARM Component Profile*.

Neste *profile*, a representação da parte do *asset* descrito que foi alterada pode ser feita de duas formas. A primeira utiliza o atributo *changed-unit*, que permite indicar o nome da parte alterada, da mesma forma como é feito na estrutura para controle de evolução definida pelo *X-ARM Interaction Profile*, apresentada na Seção 3.5. A segunda forma baseia-se no uso do elemento <*related-asset*>, neste caso, com o valor "*changedUnit*" no atributo *relationship-type*. Desta maneira, torna-se possível a representação das partes modificadas quando estas forem outros *assets*, tais como interfaces e eventos. Por exemplo, na adição de uma interface a uma especificação, o elemento <*related-asset*> permite indicar a interface adicionada usando sua identificação.

No *X-ARM Component Profile*, o atributo *type* do elemento <*evolution*> pode ter apenas valores pré-definidos, que são dependentes do tipo de *asset* descrito. Dependendo do tipo de evolução representado pelo atributo *type*, deve ser utilizado o atributo *changed-unit*, do elemento <*evolution*>, ou o elemento <*related-asset*> para indicar a parte alterada.

No caso de especificações de componentes, tanto nas independentes de modelo como nas dependentes de modelo, os possíveis valores para *type* são:

- "*interface addition*": Indica que a especificação do componente adicionou uma nova interface em relação à versão anterior da especificação. Para representar a interface adicionada utiliza-se o elemento <*related-asset*>.
- "*interface removal*": Indica a remoção de uma interface da especificação do componente em relação à versão anterior da especificação. Para representar a interface removida utiliza-se o elemento <*related-asset*>.

- "*property addition*": Indica a adição de uma propriedade na especificação do componente em relação à versão anterior. Na representação da propriedade adicionada utiliza-se o atributo *changed-unit*.
- "*property removal*": Indica a remoção de uma propriedade na especificação do componente em relação à versão anterior. Na representação da propriedade removida utiliza-se o atributo *changed-unit*.
- "*event addition*": Indica a adição de um novo evento na especificação do componente em relação à versão anterior da especificação. Na representação do evento adicionado utiliza-se o elemento <*related-asset*>.
- "*event removal*": Indica a remoção de um evento em relação à versão anterior da especificação. Na representação do evento removido utiliza-se o elemento <*related-asset*>.
- "*others*": Indica outro tipo de evolução não padronizado. Para representar a parte alterada pode-se usar tanto o atributo *changed-unit* quanto o elemento <*related-asset*>.

Já para implementações de componentes, os valores permitidos para o atributo *type* são:

- "*implementation upgrade*": Indica a atualização da implementação de componente. Não é utilizado nem o atributo *changed-unit*, nem o elemento <*related-asset*>, já que a mudança é apenas em termos de implementação, e, portanto, nenhuma parte foi adicionada ou removida.
- "*specification change*": Indica que a implementação trocou sua especificação em relação à versão anterior. Para indicar a especificação removida, é utilizado o elemento <*related-asset*>. Não é necessário representar a nova especificação adotada no elemento <*evolution*>, uma vez que cada implementação de componente já indica a sua atual especificação.
- "*others*": Indica outro tipo de evolução não padronizado. Para representar a parte alterada pode-se usar tanto o atributo *changed-unit* quanto o elemento <*related-asset*>.

Os demais aspectos das representações dos tipos de *assets* descritos pelo *X-ARM Component Profile* são detalhados nas próximas subseções.

3.6.1 Especificações de Componentes

A especificação de um componente engloba a definição de várias características que representam o comportamento do componente. Estas características são as definições das interfaces, eventos e propriedades do componente. O elemento <*component-specification*> permite representar especificações de componentes independentes e dependentes de modelo de componente com estruturas bastante similares.

3.6.1.1 Especificação Independente (*Independent Specification)*

Em uma especificação independente de modelo de componentes, todos os seus elementos (propriedades, interfaces, eventos) obrigatoriamente devem ser independentes de modelo. A estrutura de uma especificação de componente independente de modelo é apresentada na Figura 48. Assim como adotado por outros elementos, o elemento <*component-specification*> pode possuir uma descrição textual representada com o elemento <*description*>.

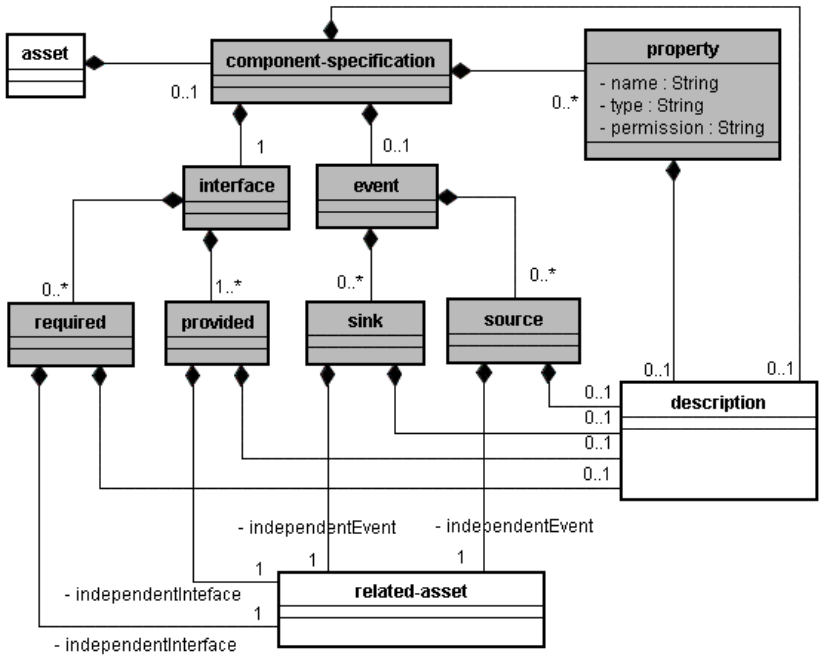

Figura 48: Especificação de componentes independentes de modelo.

Observa-se na Figura 48 que o elemento <*component-specification*> possui três elementos principais, que são apresentados com detalhes nas próximas subseções: <*interface*>, <*event*>, <*property*>.

Interfaces

Uma especificação de componente pode referenciar um conjunto de interfaces. Para tal, o *X-ARM Component Profile* define o elemento <*interface*>. Este elemento possui os elementos <*provided*> e <*required*>, responsáveis por indicar as interfaces providas e requeridas do componente com elementos <*related-asset*>. Em ambos os casos, o valor do atributo *relationship-type* do elemento <*related-asset*> é "*independentInterface*". Um exemplo da representação de interfaces em uma especificação independente de modelo é apresentada na Figura 49.

```
…
1. <component-specification>
2.     <interface>
3.         <required>
4.             <related-asset id="br.ufpb.compose.IEditor-1.0" relationship-type="independentInterface" required="true"/>
5.         </required>
6.         <provided>
7.             <related-asset id="br.ufpb.compose.IMgt-1.0" relationship-type="independentInterface" required="true"/>
8.             <description>Interface de Gerenciamento</description>
9.         </provided>
10.        <provided>
11.            <related-asset id="br.ufpb.compose.ISpell-1.0" relationship-type="independentInterface" required="true"/>
12.        </provided>
13.    </interface>
14. </component-specification>
…
```

Figura 49: Exemplo de interfaces em uma especificação de componente independente.

Neste exemplo são representadas duas interfaces providas, cujos identificadores são "*br.ufpb.compose.IMgt-1.0*" e "*br.ufpb.compose.ISpell-1.0*", e apenas uma requerida, identificada por "*br.ufpb.compose.IEditor-1.0*".

Eventos

Para poder representar os possíveis eventos gerados ou recebidos por um componente, o *X-ARM Componente Profile* define o elemento <*event*>, ilustrado na Figura 44. Este elemento possui outros dois elementos, <*sink*> e <*source*>, usados para indicar os eventos (*assets* do tipo *Independent Event*) nos quais a especificação de componente é receptor ou fonte, respectivamente. Para referenciar os eventos, é utilizado o elemento <*related-asset*>, cujo valor do atributo *relationship-type* é "*independentEvent*". Na Figura 50 é exemplificado um

trecho de representação dos eventos de uma especificação de componente independente de modelo.

```
…
1. <component-specification>
2.     <event>
3.         <sink>
4.             <related-asset id="br.compose.events.editorAlert-1.0" relationship-type="independentEvent" required="true"/>
5.         </sink>
6.         <source>
7.             <related-asset id="br.compose.events.Email-1.2" relationship-type="independentEvent" required="true"/>
8.         </source>
9.     </event>
10. </component-specification>
…
```

Figura 50: Exemplo de eventos em uma especificação de componente independente.

Neste exemplo, é definido que, para o evento descrito no *asset* identificado por "*br.compose.events.editorAlert-1.0*", um componente que adote esta especificação do componente tem o papel de receptor do evento. Por outro lado, para o evento representado no *asset br.compose.events.Email-1.2*, um componente que adote esta especificação tem o papel de fonte do evento.

Propriedades

Propriedades constituem valores associados a um componente que possibilitam a sua customização. Geralmente, a manipulação das propriedades permite a configuração do estado do componente. Para representar propriedades foi criado o elemento *<property>*, ilustrado na Figura 48, que possui três atributos: *name*, usado para identificar o nome da propriedade; *type*, que identifica o tipo de dado da propriedade; e *permission*, que indica a permissão de acesso à propriedade, podendo possuir apenas os valores "*read*", "*write*" ou "*readwrite*". A Figura 51 apresenta um exemplo de descrição de propriedades.

```
…
1. <component-specification>
2.     <property name="username" type="String" permission="readwrite">
3.             <description>Obtém e altera o valor do nome do usuário.</description>
4.     </property>
5. </component-specification>
…
```

Figura 51: Exemplo de propriedade de componente independente.

Neste exemplo, a propriedade chamada *username* é definida tanto para escrita quanto para leitura.

3.6.1.2 Especificação Dependente (*Dependent Specification*)

A especificação de componente dependente de modelo tem como objetivo representar as características de um componente para um modelo de componentes específico, a fim de que, em um passo posterior, sua implementação seja representada.

Assim como acontece com interfaces, eventos e exceções, uma especificação de componente dependente de modelo de componentes é baseada em uma especificação independente de modelo e acrescenta características específicas do modelo de componentes adotado. Assim, a especificação de um componente dependente de modelo preserva a estrutura e a semântica do elemento <*component-specification*>, mas adiciona o elemento <*model-dependency*>. Além disto, as interfaces e eventos que compõem a especificação de componente obrigatoriamente devem ser dependentes de modelo. A estrutura do elemento <*component-specification*> é apresentada na Figura 52.

Uma especificação de componente dependente de modelo pode referenciar uma especificação independente de modelo. Para tal, o elemento <*model-dependency*> utiliza o elemento <*related-asset*> para indicar a especificação independente de modelo da qual a especificação dependente é uma especialização. Neste caso, o valor do atributo *relationship-type* do elemento <*related-asset*> deve ser "*independentComponentSpec*".

Para indicar o modelo de componentes adotado pela especificação dependente de modelo, <*model-dependency*> possui outro elemento <*related-asset*>, que indica um *asset* do tipo *Component Model*. Neste caso, o valor do atributo *relationship-type* de <*related-asset*> deve ser "*componentModel*". A utilização de uma referência para o *asset* que contém a especificação do modelo de componentes é adotada a fim de validar as características de uma especificação de componente dependente de modelo. Por exemplo, este recurso serve para garantir que uma especificação de componente dependente de modelo não possua interfaces, eventos ou propriedades se estes não forem permitidos pelo modelo de componentes adotado.

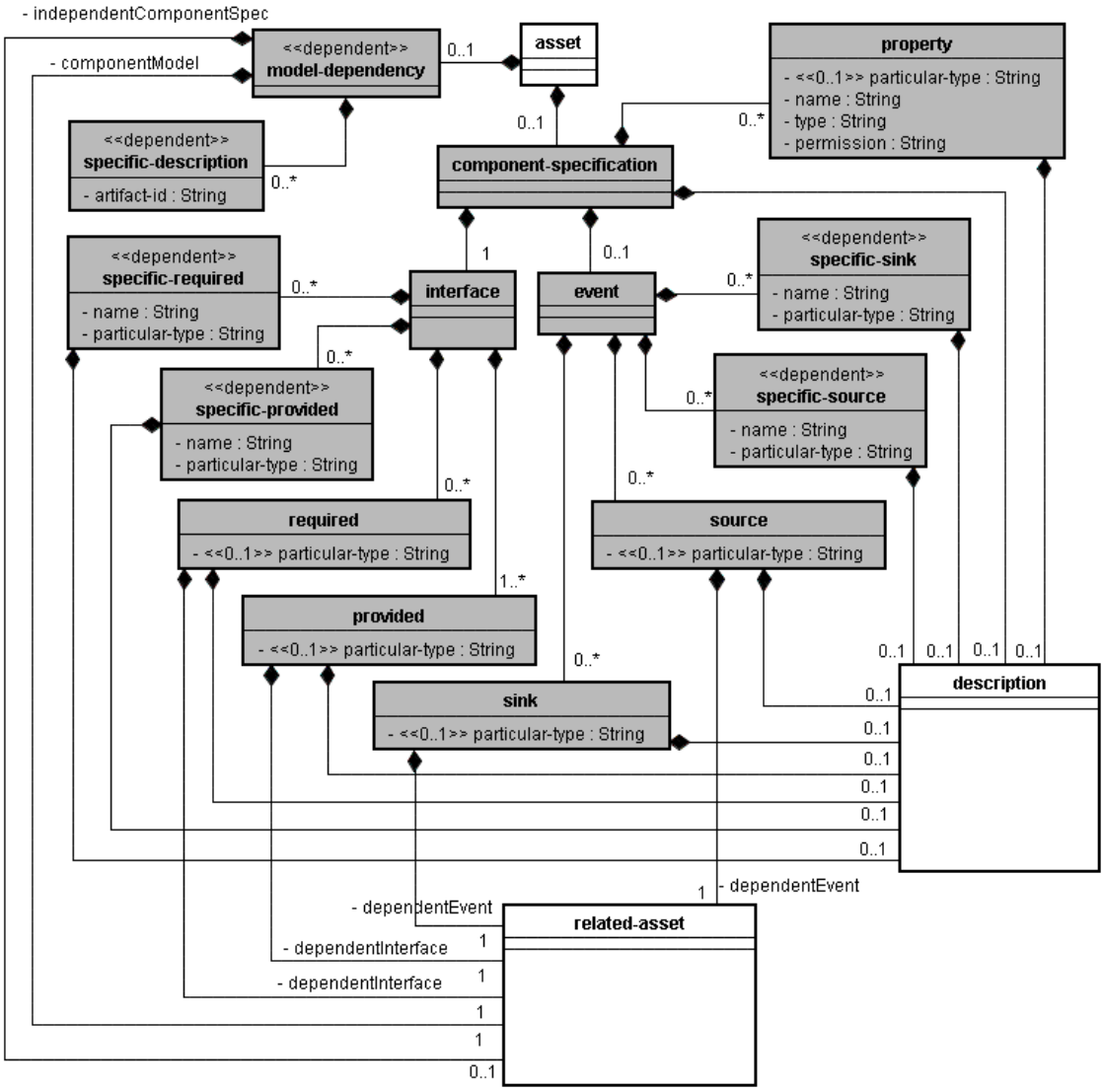

Figura 52: Especificação de componentes dependentes de modelo.

No modelo de componentes de CORBA [CCM06], os próprios componentes também são descritos em arquivos usando uma linguagem específica, CORBA IDL. No X-ARM, estes arquivos de descrição de componentes são representados no elemento <*specific-description*>. Neste elemento, o arquivo de descrição deve ser indicado pelo atributo *artifact-id* que contém o identificador de um elemento <*artifact*> do próprio manifesto, indicando o nome do arquivo dentro do pacote que contém a descrição do componente.

Todas as funcionalidades referentes a uma especificação de componente são agrupadas pelo elemento <*component-specification*>, sendo mais bem descritas a seguir.

As interfaces, eventos e propriedades de uma especificação dependente de modelo são semanticamente similares a uma especificação independente de modelo. Porém, uma diferença é que nas especificações dependentes de modelo, os elementos para representação de interfaces e eventos só devem referenciar (com o elemento <*related-asset*>) interfaces e eventos dependentes do mesmo modelo de componentes.

Além disto, os elementos <*provided*>, <*required*>, <*sink*> e <*source*> passam a ter o atributo *particular-type*, usado para indicar o nome utilizado pelo modelo de componentes para representar a interface ou o evento. Por exemplo, em CCM (especificado com X-ARM na Figura 34) tem-se a possibilidade de usar no atributo *particular-type* o valor "*receptacle*" para uma interface requerida e usar os valores "*facet*" ou "*component*" para interfaces providas. Ressalta-se que, no contexto de uma especificação de componente independente de modelo, o atributo *particular-type* não deve ser utilizado.

Observa-se na Figura 52 que o elemento <*interface*> possui dois novos elementos: <*specific-required*> e <*specific-provided*>. Além disto, o elemento <*event*> também possui dois novos elementos: <*specific-sink*> e <*specific-source*>. Todos estes novos elementos são responsáveis por indicar interfaces e eventos específicos a um modelo de componentes que ainda não tenham sido especificados em X-ARM. Como pode ser observado, estes elementos possuem os atributos *name* e *particular-type*. O primeiro deve indicar um nome de interface ou evento definido pelo próprio modelo de componentes. Por exemplo, o modelo COM define que seus componentes devem possuir a interface *IUnknown* e, portanto, este é um possível valor para o atributo *name* no caso de uma especificação de componente COM. Por outro lado, o segundo atributo indica o nome adotado pelo modelo de componentes para identificar um determinado tipo de interface ou evento. Por exemplo, no caso da representação de uma interface requerida de uma especificação de componente CCM, o valor de *particular-type* seria "*receptacle*".

A Figura 53 apresenta um exemplo de descrição de uma especificação de componente dependente de modelo de componentes.

```
1. <asset id="br.compose.sofia.compCCMSpec-2.0" …>
2.    <model-dependency>
3.       <related-asset id="br.ccm-3.0" relationship-type="componentModel" required="true"/>
4.       <related-asset id="br.compose.sofia.compSpec-1.0" relationship-type="independentComponentSpec"
                        required="true"/>
5.    </model-dependency>
6.    <component-specification>
7.       <interface>
8.          <required particular-type="receptacle">
9.             <related-asset id="br.compose.ccm.IEditor-1.0" relationship-type="dependentInterface" required="true"/>
10.         </required>
11.         <provided particular-type="facet">
12.            <related-asset id="br.compose.ccm.IMgt-1.0" relationship-type="dependentInterface" required="true"/>
13.         </provided>
14.         <provided particular-type="component">
15.            <related-asset id="br.compose.ccm.IComp-1.0" relationship-type="dependentInterface" required="true"/>
16.         </provided>
17.      </interface>
18.      <event>
19.         <sink particular-type="sink">
20.            <related-asset id="br.compose.ccm.editorAlert-1.0" relationship-type="dependentEvent" required="true"/>
21.         </sink>
22.         <source particular-type="source">
23.            <related-asset id="br.compose.ccm.Email-1.2" relationship-type="dependentEvent" required="true"/>
24.         </source>
25.      </event>
26.   </component-specification>
…
27. </asset>
```

Figura 53: Exemplo de especificação de componente dependente.

O exemplo mostra que o modelo de componentes e a especificação de componente independente de modelo seguidos são identificados pelos *assets br.ccm-3.0* e *br.compose.sofia.compSpec-1.0*, respectivamente. A especificação de componente dependente de modelo exemplificada inclui uma interface requerida e duas providas. Além disso, o componente que implementar esta especificação será receptor do evento *br.compose.ccm.editorAlert-1.0* e fonte do evento *br.compose.ccm.Email-1.2*.

3.6.2 Implementações de componentes

A implementação de um componente deve seguir uma especificação de componente dependente de modelo e englobar as características diretamente relacionadas ao componente implementado, como código fonte, código executável e documentação. O *X-ARM Component Profile* define o elemento <*component-implementation*>, que permite a representação destas informações, além de descrever os requisitos não funcionais e a composição interna de um componente. A Figura 54 apresenta a estrutura adotada pelo elemento <*component-implementation*> para representar implementações de componentes. Como pode ser observado, este elemento pode possuir uma descrição textual, que pode ser representada usando o elemento <*description*>.

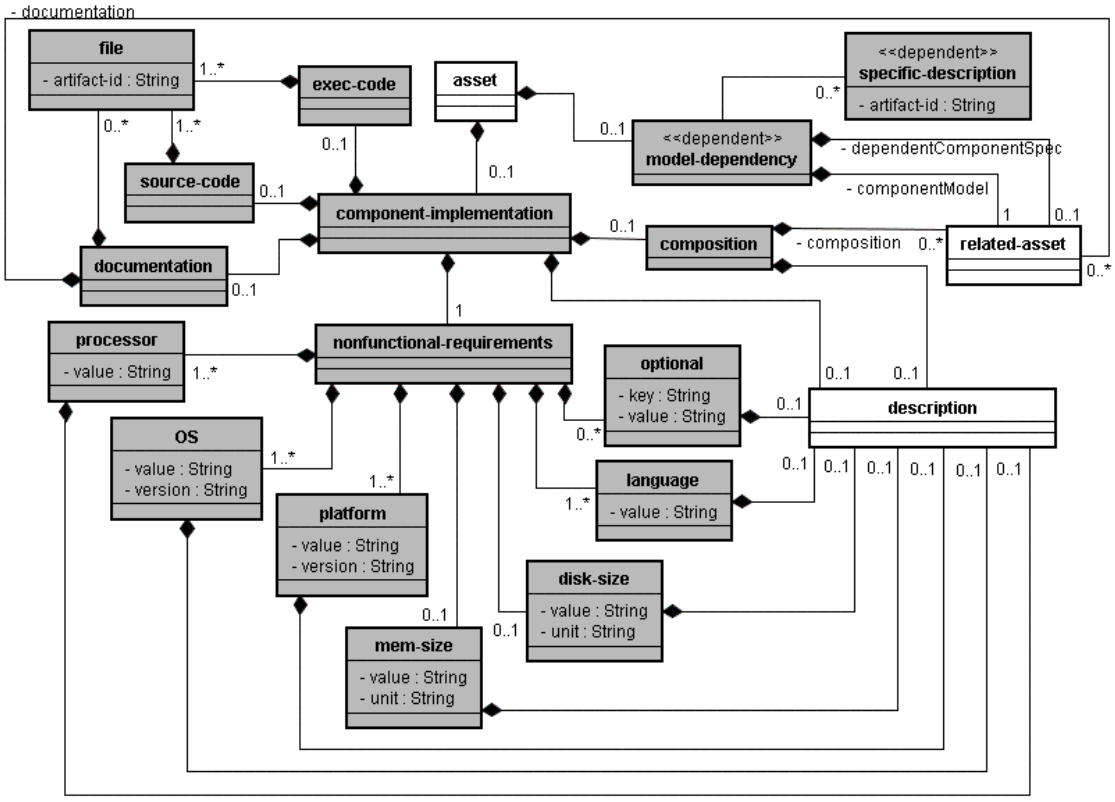

Figura 54: Especificação de implementações de componentes.

Uma implementação de componente deve obrigatoriamente referenciar uma especificação de componente dependente de modelo de componente. Para tal, o elemento <*model-dependency*> contém duas ocorrências do elemento <*related-asset*>. Na primeira, o elemento <*related-asset*> indica a especificação dependente de modelo na qual a implementação se baseia. Neste caso, o atributo *relationship-type* do elemento <*related-asset*> deve ser "*dependentComponentSpec*". Na segunda, o elemento <*related-asset*> indica o modelo de componentes adotado. Neste caso, o valor do atributo *relationship-type* do elemento <*related-asset*> deve ser "*componentModel*".

Do mesmo modo que acontece em especificações de componentes dependentes de modelo, o X-ARM também pode indicar arquivos dentro do pacote do *asset* que contêm descrições do componente, em linguagem definida pelos seus respectivos modelos. Para isto, o elemento <*specific-description*> define o atributo *artifact-id*, que contém o identificador de um elemento <*artifact*> do próprio manifesto, referente ao arquivo dentro do pacote que contém a descrição do componente.

O elemento <*component-implementation*> é utilizado para agrupar todas as características referentes a uma implementação de componente. Essas características são apresentadas nas próximas subseções.

3.6.2.1 Status de Desenvolvimento

No caso de implementações de componentes, a versão também pode representar o status do seu desenvolvimento. Neste sentido, no identificador do *asset* (atributo *id* do elemento <*asset*>) pode-se incluir após o número da versão, algum dos seguintes valores: *pre-alpha*, *alpha*, *beta*, *stable*, *mature* e *inactive*. Ressalta-se que após o número da versão deve ser inserido o caractere separador hífen. Quando este recurso for utilizado, cada um destes valores somente pode ser utilizado por um *asset* se no repositório não existir um outro *asset* com a mesma identificação, mas com um status de desenvolvimento posterior. A ordem dos valores de status é apresentada na Figura 55, onde "*pre-alpha*" é o status mais básico e "*inactive*" é o último. Um exemplo de identificador de um *asset* na fase *beta* pode ser *br.compose.mycomponentasset-2.0-beta*.

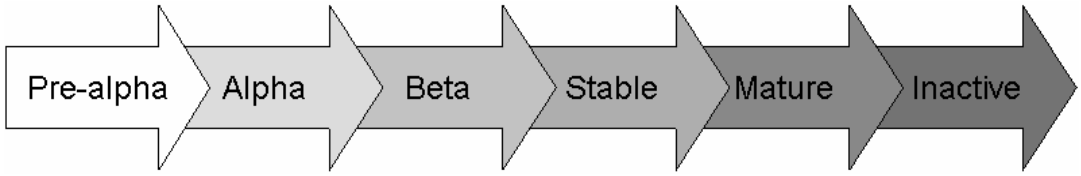

Figura 55: Seqüência de status de uma implementação de componente.

3.6.2.2 Composição

Componentes de software, assim como aplicações finais, podem ser compostos por outros componentes [Orso00]. No caso de componentes armazenados em um repositório, torna-se necessária a representação da composição, visto que, normalmente, para um componente funcionar corretamente, é necessário que os componentes que o compõem estejam instalados também.

O X-ARM permite a representação desta característica usando o elemento <*composition*>, que agrupa as referências a outras implementações de componentes usando ocorrências do elemento <*related-asset*>. Neste caso, o valor do atributo *relationship-type* de <*related-asset*> assume o valor "*composition*". A ligação do elemento <*composition*> com <*related-asset*> é ilustrada na Figura 54.

É importante lembrar que o elemento <*related-asset*> possui o elemento opcional <*artifact*>, que possibilita que o *asset* indicado esteja contido no próprio pacote do *asset* descrito, não sendo necessária uma busca e uma recuperação no repositório. Este recurso pode ser explorado na remoção de *assets* do repositório. Neste caso, antes de remover o *asset* do repositório, deve-se incluí-lo no pacote do *asset* dependente.

3.6.2.3 Código Executável e Código Fonte

Para representar de forma diferenciada os arquivos referentes a estes diferentes tipos de código, foram definidos o elemento <*exec-code*> e o elemento <*source-code*>. O primeiro agrupa os arquivos que compõem o código executável, enquanto que o segundo agrupa os arquivos que compõem o código fonte. Ambos os elementos contêm o elemento <*file*>, responsável por informar seus respectivos arquivos localizados dentro do pacote do componente implementado, usando o atributo *artifact-id*.

3.6.2.4 Documentação

A documentação é um texto, representado por um ou mais arquivos, que acompanha a implementação do componente a fim de explicar como o componente opera e como deve ser utilizado. Neste sentido foi definido o elemento <*documentation*>, que é responsável por agrupar todos os arquivos de documentação relacionados à implementação de um componente. Para isto, o elemento <*documentation*> pode conter o elemento <*file*> e o elemento <*related-asset*>. O primeiro é usado para indicar os arquivos da documentação, localizados no pacote do componente implementado, através do atributo *artifact-id*. Já o elemento <*related-asset*> é usado para indicar documentações que possam estar em outros *assets*, do tipo *Resource*. Neste caso, o atributo *relationship-type* tem o valor "*documentation*".

3.6.2.5 Requisitos Não Funcionais

A representação de requisitos não funcionais contém informações que auxiliam o consumidor no momento de usar o componente. Estas informações abordam qual sistema operacional deve ser utilizado, a família de processadores, a linguagem de programação utilizada, a plataforma de execução, espaços de memória e de disco rígido requeridos, entre outros não previamente definidos, mas que podem ser especificados por meio de pares de chave e valor. Para descrever estes requisitos não funcionais, o *X-ARM Component Profile* provê o elemento <*nonfunctional-requirements*>, apresentado na Figura 54.

O elemento <*nonfunctional-requirements*> agrupa uma série de elementos pré-definidos a fim de padronizar um conjunto de requisitos não funcionais. São definidos seis elementos pré-definidos: <*language*>, <*OS*>, <*platform*>, <*processor*>, <*disk-size*> e <*mem-size*>. O elemento <*language*> é usado para indicar a linguagem de programação utilizada no desenvolvimento do componente. <*OS*> identifica sistemas operacionais compatíveis com o componente. O elemento <*platform*> indica plataformas de execução que podem ser usadas na execução do componente. O elemento <*processor*> é usado para informar o tipo de processador requerido. Por fim, os elementos <*disk-size*> e <*mem-size*> são usados para informar os espaços em disco e em memória requeridos, respectivamente. No caso dos elementos <*disk-size*> e <*mem-size*>, o atributo *value* indica a quantidade de memória requerida e o atributo *unit* indica a unidade na qual esta memória é medida, como por exemplo, *Kb*, *Mb* e *Gb*. Apesar de pré-definir um conjunto de elementos, requisitos adicionais podem ser especificados através do elemento <*optional*>, cujos atributos *key* e *value* representam o nome e o valor do requisito não funcional especificado. Para ilustrar, a Figura 56 apresenta uma possível representação dos requisitos não funcionais de um componente.

```
…
1. <nonfunctional-requirements>
2.      <OS value="Windows" version="XP"/>
3.      <platform value="Windows .NET" version="1.3"/>
4.      <language value="C#"/>
5.      <processor value="0x86"/>
6.      <optional key="video memory" value="32Mb"/>
7. </nonfunctional-requirements>
…
```

Figura 56: Exemplo de requisitos não funcionais.

Observa-se na Figura 56 que o componente funciona no sistema operacional Windows na versão XP. A linguagem de programação em que foi desenvolvido é C#, a plataforma para execução é Windows .NET na versão 1.3 e o processador sobre o qual o componente funciona é o da família 0x86. Além disto, também é especificado que para a execução do componente é necessária uma memória de vídeo de 32Mb.

3.7 Considerações Finais

O X-ARM acrescenta quatro novos profiles ao RAS, possibilitando que novas e importantes características sejam representadas. Estes profiles permitem a representação de

18 diferentes tipos de *assets*, com finalidades específicas, úteis no contexto de repositórios de componentes.

Diversas características representadas pelo X-ARM não estão presentes em outros modelos, porém, muitas destas características são representadas por pelo menos um dos modelos apresentados na Seção 2. Com isto, X-ARM suporta várias características que já foram definidas como importantes ou essenciais por outros modelos de representação. Além disto, outras informações não representadas por nenhum dos modelos apresentados, como certificação dos produtores de *assets* e composição de componentes também são descritas com X-ARM. Tudo isto o torna um modelo bastante abrangente, atendendo boa parte dos requisitos de repositórios de componentes.

Com o *X-ARM Artifact Profile*, o modelo proposto introduz a descrição de importantes informações não representadas pelo RAS. Estas informações se referem a certificação de *assets* e de produtores, formas e termos de negociação de *assets*, histórico de evolução, visibilidade, *assets* reusados na construção de um dado *asset* e dados sobre o processo de desenvolvimento nos quais o *asset* foi produzido.

Alguns aspectos das descrições de *assets* podem seguir diferentes modelos. Por exemplo, um *asset* pode ser certificado com diferentes processos de certificação, pode ser negociado e vendido de diferentes maneiras e, no caso de componentes, podem ser construídos seguindo-se diferentes modelos de componentes. Neste sentido, o X-ARM apresenta uma abordagem inovadora entre os modelos de representação existentes, ao permitir que diferentes modelos sejam utilizados na descrição de determinados tipos de informações. Com o *X-ARM Model Profile*, modelos de certificação, negociação, componentes, processos de desenvolvimento e áreas de aplicação podem ser especificados. A partir destes modelos, os respectivos tipos de informação podem ser representados, com valores definidos nos modelos.

Com o *X-ARM Interaction Profile*, é possível a representação de interfaces, eventos e exceções sem estarem associadas a um componente específico. Assim, especificações de interfaces e eventos podem ser reusadas por diferentes componentes, evitando-se o re-trabalho. Em particular, no caso de exceções, elas podem ser adotadas por diferentes interfaces e eventos. Além disto, o *X-ARM Interaction Profile* permite que estes elementos

sejam especificados de forma independente e dependente de modelos de componentes, permitindo que uma especificação independente não represente características particulares a nenhum modelo e que seja utilizada por especificações dependentes que seguem quaisquer modelos.

O *X-ARM Component Profile* descreve os componentes em três diferentes níveis de abstração: implementações e especificações independentes e dependentes de modelos de componentes. Ao separar especificações de implementações, o X-ARM permite que uma mesma especificação seja empregada por diferentes implementações, habilitando-se o reuso de especificações. Ao dividir as especificações de componentes em independentes e dependentes, o *X-ARM Component Profile* possui os mesmos benefícios apresentados por esta separação no *X-ARM Interaction Profile*, apresentados no parágrafo anterior.

4 Ferramenta de Descrição e Empacotamento de Assets

Uma grande quantidade de informações de *assets* é representada pelo modelo X-ARM. Isto pode tornar bastante trabalhosa e complicada a tarefa de descrever um *asset* utilizando apenas editores de texto ou de documentos XML. A fim de facilitar esta atividade, foi desenvolvida uma ferramenta denominada X-Packager, que tem a finalidade de gerar descrições de *assets* na forma de arquivos de manifesto e agrupar estes e os demais arquivos que compõem os *assets* em pacotes únicos. Assim, este capítulo apresenta a ferramenta X-Packager e um exemplo de sua utilização, que buscam servir como prova de conceito e exemplo para emprego do modelo X-ARM.

A ferramenta X-Packager foi construída na forma de um *plug-in* para a versão 3.1.x deste ambiente, dadas as facilidades do ambiente Eclipse [Eclipse06]. Dentre estas facilidades, destaca-se a própria característica de que *plug-ins* possam ser incorporados ao ambiente, de forma que uma variedade de tipos de tarefas possa ser executada, permitindo a produção de diversos tipos de artefatos.

O X-Packager foi construído em Java, usando JDK 1.5.0. Para o desenvolvimento de interfaces gráficas, foi utilizado o pacote SWT e para a realização de tarefas relacionadas à manipulação de dados XML utilizou-se a implementação de Java da API DOM [W3C04b]. A implementação da ferramenta possui cerca de 41 mil linhas de código, distribuídas em 410 classes e 86 pacotes Java. O espaço em disco requerido pela ferramenta é de 1,98 Mb para o código compilado (*bytecode*) e 2,32 Mb para o código fonte.

4.1 A Ferramenta X-Packager

A ferramenta X-Packager é responsável por receber um conjunto de informações para, a partir disto, gerar a descrição e o pacote do *asset*, com os arquivos que o compõem. Conforme definido pelo modelo RAS, do qual o X-ARM é uma especialização, este pacote consiste de um arquivo ZIP [Moffat97] contendo os arquivos do *asset* e um arquivo XML de manifesto com elementos definidos pelo X-ARM, chamado *manifest.rmd*, referente à descrição do *asset*.

Para esconder do usuário a complexidade da representação das informações do modelo X-ARM, o *plug-in* foi desenvolvido como um *wizard,* ou seja, uma ferramenta que auxilia o usuário ao requerer gradativamente, por meio de formulários, as informações necessárias ao empacotamento do *asset*. Basicamente, cada elemento que é filho direto do elemento <*asset*> do X-ARM, deriva um ou mais formulários. Assim, por exemplo, existem formulários para entrada de informações de negociação, visibilidade, certificação, interfaces.

O *wizard* apresenta um conjunto de formulários variável conforme o tipo de *asset* a ser empacotado. Por exemplo, se o *asset* a ser empacotado é do tipo *Resource* (conforme classificação apresentada na Seção 3.1), a ferramenta não apresenta formulários para informação de dados de interfaces, eventos, modelos ou componentes. A Tabela 2 apresenta um sumário das quantidades de formulários apresentados no empacotamento dos diferentes tipos de *assets*.

Tipos de Assets	Quantidade de formulários
Assets do X-ARM Artifact Profile	12
Assets do X-ARM Model Profile	13 (12 do X-ARM Artifact Profile + 1)
Assets independentes de modelo do X-ARM Interaction Profile	14 (12 do X-ARM Artifact Profile + 2)
Assets dependentes de modelo do X-ARM Interaction Profile	14 (12 do X-ARM Artifact Profile + 2)
Independent Component Specification	15 (12 do X-ARM Artifact Profile + 3)
Dependent Component Specification	16 (12 do X-ARM Artifact Profile + 4)
Component Implementation	16 (12 do X-ARM Artifact Profile + 4)

Tabela 2: Quantidades de formulários por tipo de asset.

4.1.1 Utilização do X-Packager

Para a utilização do X-Packager, o usuário pode estar desenvolvendo normalmente seus artefatos dentro de um projeto do Eclipse e, no momento em que decidir gerar o pacote de um *asset*, ele deve utilizar a opção de exportar, oferecida pelo ambiente. Neste momento, o Eclipse exibe uma tela para escolha do destino da exportação, que inclui a opção "*X-ARM Package File*", conforme ilustrado na Figura 57.

Vale ressaltar que *assets* podem ser empacotados em qualquer momento do desenvolvimento de um projeto, desde que os arquivos que compõem um *asset* a ser empacotado já estejam concluídos. Assim, mesmo que o projeto de um sistema ainda não tenha sido finalizado, caso os artefatos já produzidos relativos a um *asset* tenham sido finalizados, os mesmos já podem ser empacotados.

Figura 57: Tela do Eclipse para escolha da forma de exportação.

Depois que o usuário escolhe exportar arquivos como um pacote X-ARM, o *plug-in* é iniciado, e à medida que o usuário preenche as informações solicitadas em cada formulário, um outro formulário é apresentado, até chegar no passo em que o usuário escolhe o nome e o local onde salvar o manifesto ou o pacote do *asset*. Entretanto, a

descrição de *assets* pode ser uma tarefa relativamente longa, considerada a grande quantidade de características representadas pelo X-ARM e considerado que o usuário pode não possuir algumas informações quando estiver empacotando um *asset*. Por exemplo, no momento do empacotamento, o usuário, que normalmente será um desenvolvedor, pode não ter conhecimento de informações não técnicas, tais como os modelos de negócios permitidos para o *asset*, ou quem estará autorizado a acessar o *asset*. Assim, a fim de evitar que informações já inseridas sejam perdidas, o X-Packager permite que o empacotamento do *asset* seja interrompido, fazendo com que o pacote do *asset*, ou mesmo apenas o manifesto, seja gerado mesmo antes que todas as informações tenham sido fornecidas. Quando as informações restantes forem conseguidas, o usuário pode acessar novamente a ferramenta e pedir para carregar um pacote X-ARM ou um manifesto que foi gerado anteriormente de maneira incompleta. No caso de um pacote, o X-Packager carrega um arquivo .zip, e no caso de um manifesto, a ferramenta carrega arquivos .xml ou .rmd. Desta forma, o *plug-in* apresentará as informações já preenchidas, permitindo a modificação das mesmas e a inserção das informações que restavam.

No entanto, depois que o usuário gera um pacote ou manifesto incompleto, ele pode continuar desenvolvendo seus artefatos e, enquanto isso, arquivos podem ser adicionados ao seu projeto no Eclipse. Portanto, quando o empacotamento de um *asset* é reiniciado, o X-Packager destaca os arquivos que foram adicionados ao projeto do Eclipse, possibilitando que o usuário possa optar por adicionar ou não os novos arquivos ao pacote do *asset*. A Figura 58 ilustra o formulário do *plug-in* que mostra os arquivos existentes no projeto do Eclipse.

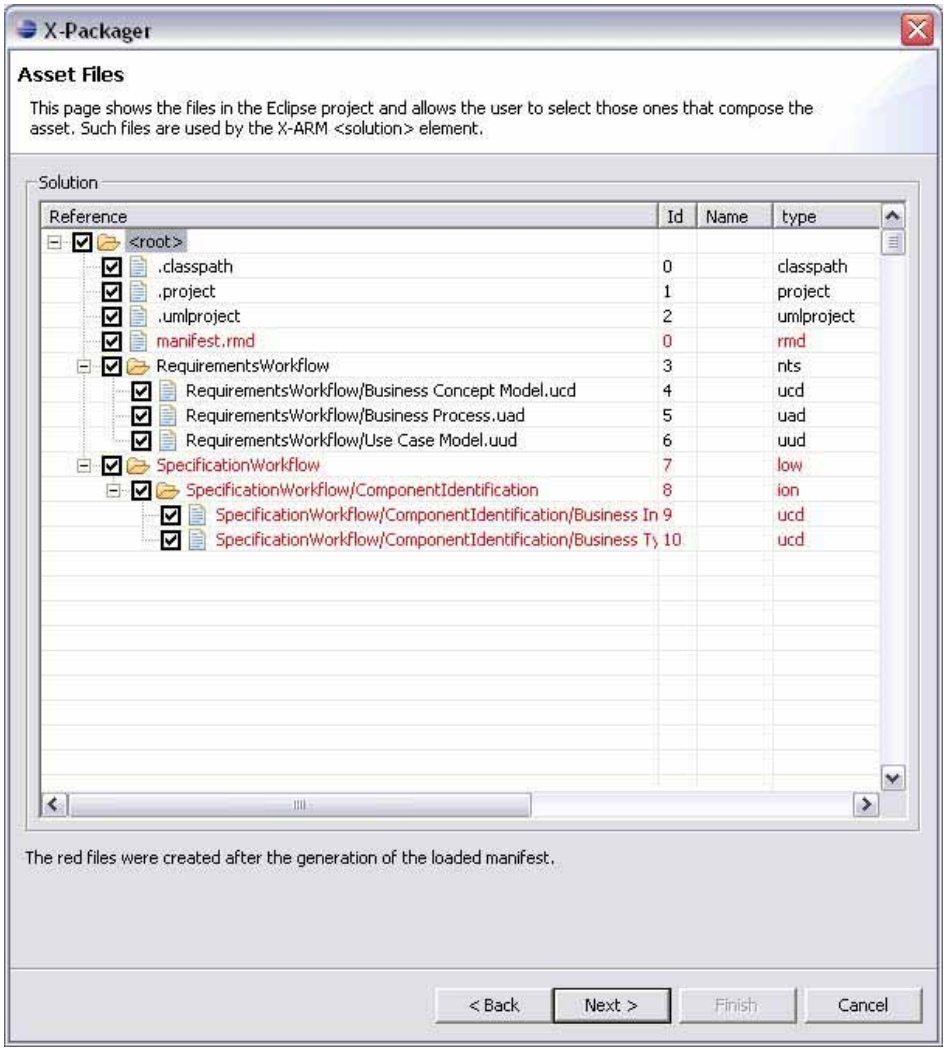

Figura 58: Tela para especificação dos arquivos que compõem o asset.

Além de permitir o carregamento de informações de manifestos incompletos, o X-Packager também possibilita que manifestos de outros *assets* possam ser carregados para auxiliar a descrição de um *asset*. Este recurso é útil para evitar que informações que se repetem em vários *assets* tenham que ser introduzidas novamente para cada *asset*. Desta forma, para a descrição de um *asset*, todas as informações do manifesto de um *asset* já descrito podem ser carregadas, de forma que seja necessário alterar apenas aqueles dados que são diferentes no *asset* que está sendo empacotado.

4.2 Um Exemplo de Uso

A fim de avaliar e apresentar um exemplo da utilização do modelo X-ARM e da ferramenta X-Packager, foi realizada a descrição e empacotamento de um conjunto de *assets* que foram

produzidos utilizando-se o processo de desenvolvimento UML Components [Cheesman01]. Com a intenção de trabalhar com *assets* que realmente foram gerados utilizando-se este processo, optou-se por utilizar os próprios artefatos apresentados em [Cheesman01]. Desta forma, o exemplo de uso se refere aos *assets* gerados na modelagem de um sistema de reservas de hotel. A Figura 59 ilustra o modelo conceitual de negócios que ilustra a relação entre as entidades envolvidas em um sistema de reserva de hotel.

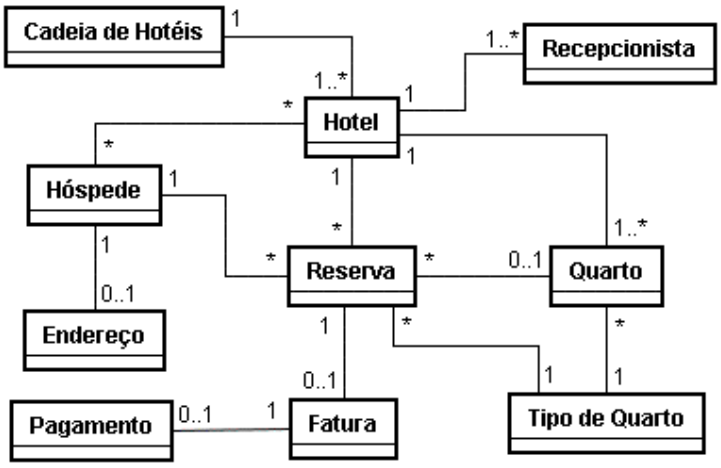

Figura 59: Modelo conceitual de negócios para o sistema de reservas.

Neste sistema, hotéis fazem parte de uma cadeia de hotéis e cada hotel pode possuir diversos hóspedes, para os quais podem ser realizadas diversas reservas ao longo do tempo. Com este sistema, é possível a realização de atividades como realização, cancelameto e alteração de reservas; processar chegadas e não comparecimentos de hóspedes. Além destas, outras possíveis atividades incluem adição, alteração e exclusão de hotéis, quartos e hóspedes.

Em [Cheesman01], cada tipo de artefato gerado no processo é exemplificado, porém, nem todos os artefatos são apresentados. Por exemplo, algumas interfaces de sistema são ilustradas, porém, nem todas as interfaces de sistema relativas à modelagem realizada são apresentadas. Assim, a modelagem apresentada no livro foi completada, de forma que foram criados arquivos referentes aos artefatos que restavam à conclusão do sistema modelado. As próximas seções apresentam os *assets* que foram gerados ao longo do processo de desenvolvimento, agrupados pelas fases do processo UML Components,

com exceção dos modelos, que foram criados e são apresentados primeiro, dado que são utilizados por diversos outros *assets*.

4.2.1 Modelos

No contexto de um sistema de repositório de componentes, normalmente a descrição do *asset* com X-ARM precisa referenciar outros *assets*, que já precisam estar descritos, empacotados e registrados no repositório. Dentre os principais tipos de *assets* que podem ser referenciados por outros estão os modelos, representados com o *X-ARM Model Profile*. Isto se deve ao fato de que informações sobre classificação, modelos de negócios, certificação, modelos de componentes e processos de desenvolvimento precisam referenciar os *assets* que descrevem os respectivos modelos utilizados. Portanto, a realização do exemplo de uso foi iniciada com a criação dos modelos, que são *assets* do *X-ARM Model Profile*, apresentados na Tabela 3.

Nome do Asset	Tipo do Asset
br.compose.umlcomponents-1.0	Development Process
br.compose.certificationmodel-1.0	Certification Model
br.compose.businessmodel-1.0	Business Model
br.compose.javabeanscomponentmodel-1.0 br.compose.ejbcomponentmodel-1.0 br.compose.dotnetcomponentmodel-1.0	Component Model
br.compose.areas-1.0	Application Areas

Tabela 3: Assets do X-ARM Model Profile criados no exemplo de uso.

Com exceção de *assets* do tipo *Component Model*, para cada tipo de *asset* descrito com *X-ARM Model Profile*, foi criado um *asset*. Estes *assets* especificam os modelos que foram utilizados na descrição de *assets* dos demais tipos.

Os componentes especificados no exemplo de uso foram implementados com os modelos de componentes *JavaBeans* [Sun97], *Enterprise JavaBeans* [DeMichiel01] e .NET [Barnett02]. Por isto, foram criados três *assets* do tipo *Component Model*, que descrevem estes modelos.

4.2.2 Workflow Requirements

Depois de definidos os modelos, os *assets* referentes aos artefatos apresentados em [Cheesman01] foram gerados, conforme a organização das fases do processo UML Components. Assim, os *assets* que foram gerados no *workflow Requirements*, a primeira fase do processo, são apresentados na Tabela 4, onde são identificados os tipos de artefatos gerados neste *workflow*, os arquivos que compõem os *assets* gerados e os próprios *assets*.

Tipo de Artefato	Arquivos	Identificador do Asset	Tipo
Business Process	Business Process.uad	br.compose.businessprocess-1.0	Resource
Business Concept Model	Business Concept Model.ucd	br.compose.bcm-1.0	
Use Cases	Use Case Model.uud	br.compose.usecases-1.0	

Tabela 4: Assets gerados no workflow Requirements.

O *workflow Requirements* tem três tipos de artefatos de saída. Para cada um destes tipos foi gerado um *asset* contendo um arquivo. Estes arquivos são referentes a diagramas UML, criados com um *plug-in* para o ambiente Eclipse, chamado EclipseUML [Omondo06]. Estes diagramas são de atividades, de classes e de casos de uso, respectivamente. Por não se tratarem de interfaces, eventos, exceções, componentes ou modelos, os três *assets* gerados neste *workflow* são do tipo *Resource*.

4.2.3 Workflow Specification

Seguindo-se o processo UML Components, após o *workflow Requirements* são gerados os artefatos do *workflow Specification*, que é dividido em três estágios: *Component Identification*, *Component Interaction* e *Component Specification*. A Tabela 5 apresenta os *assets* gerados no primeiro destes estágios, *Component Identification*.

Tipo de Artefato	Arquivos	Identificador do Asset	Tipo
Business Type Model	Business Type Model.ucd	br.compose.btm-1.0	Resource
Business Interfaces	Business Interfaces.ucd	br.compose.businessinterfaces-1.0	
System Interfaces and Opers	System Interfaces.ucd	br.compose.systeminterfaces-1.0	

Initial Comp Specs and Architecture	Initial Component Spec and Architecture.upd	br.compose.initialarchitecture-1.0	

Tabela 5: Assets gerados no estágio Component Identification.

O estágio *Component Identification* possui quatro tipos de artefatos de saída. Cada um destes tipos originou um arquivo. Os três primeiros arquivos são diagramas de classes, enquanto que o último arquivo refere-se a um diagrama de componentes. Todos os *assets* gerados nesta fase são do tipo *Resource*, já que se trata de diagramas e não se referem a interfaces, eventos, exceções, componentes ou modelos.

O segundo e o terceiro *assets* se referem às interfaces a serem implementadas pelos componentes especificados, divididas em dois grupos, conforme definido pelo processo UML Components: interfaces de sistema e interfaces de negócio. As interfaces de sistema são aquelas que definem as funcionalidades do sistema como um todo. Neste exemplo de uso, chegou-se às seguintes interfaces de sistema: *IMakeReservation*, *ITakeUpReservation*, *IUpdateReservation*, *ICancelReservation*, *IIdentifyReservation* e *IProcessNoShow*. Estas interfaces são todas implementadas pelo componente chamado *ReservationSystem*.

Por outro lado, as interfaces de negócio definem funcionalidades mais básicas, geralmente reusadas por diferentes sistemas. As interfaces de negócio geradas foram *IHotelMgt* e *ICustomerMgt*. A primeira é implementada pelo componente *HotelMgr* e a segunda, pelo componente *CustomerMgr*.

O último *asset* apresentado na Tabela 5 se refere a um diagrama que apresenta a arquitetura e especificação inicial dos componentes e suas interfaces. Este diagrama é apresentado na Figura 60, a fim de ilustrar a correlação entre as interfaces e os componentes definidos. Na Figura 60 também é apresentada a interface *IBilling*, da qual o componente *ReservationSystem* também depende. No entanto, as especificações desta interface e do componente que a implementa estão fora do escopo da modelagem realizada em [Cheesman01] e, portanto, não foram especificadas nas fases seguintes do processo de desenvolvimento.

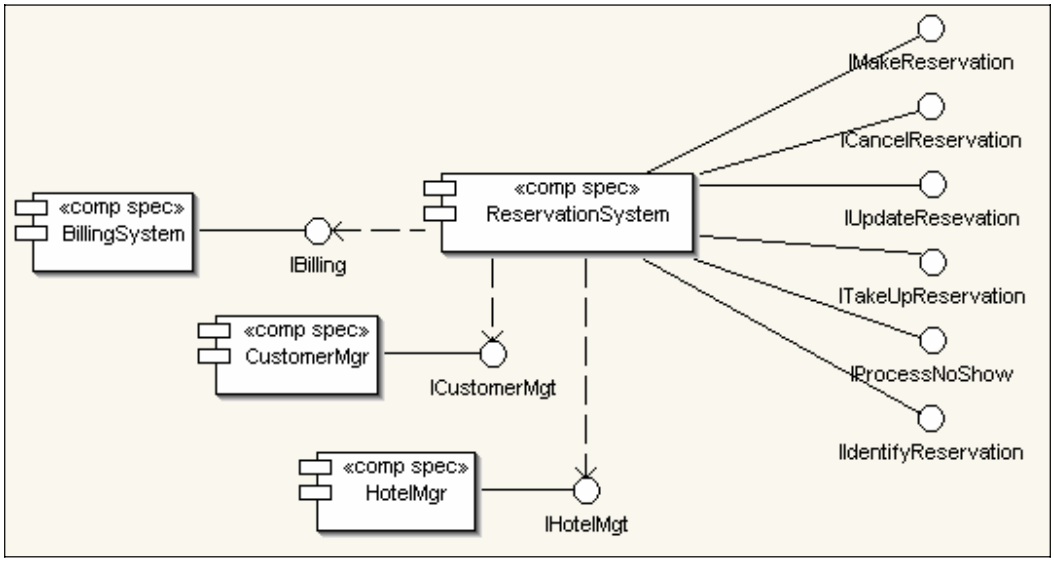

Figura 60: Arquitetura da especificação de componentes.

As seis interfaces de sistema devem ser implementadas pelo componente *ReservationSystem*, que requer as interfaces *IBilling*, *ICustomerMgt* e *IHotelMgt*, que devem ser providas pelos componentes *BillingSystem*, *CustomerMgr* e *HotelMgr*, respectivamente.

Depois de criados os *assets* do estágio *Component Identification*, foram gerados os *assets* no estágio *Component Interaction*. A Tabela 6 apresenta os arquivos e *assets* gerados neste estágio.

Tipo de Artefato	Arquivos	Identificador do Asset	Tipo
Business Operations	BO IMakeReservation.uld	br.compose.businessoperations-1.0	Resource
	BO ICancelReservation.uld		
	BO IUpdateReservation.uld		
	BO ITakeUpReservation.uld		
	BO IProcessNoShow.uld		
	BO IIdentifyReservation.uld		
Interfaces and Opers	iimakereservation.ucd	br.compose.iimakereservation-1.0	Independent Interface
	iicancelreservation.ucd	br.compose.iicancelreservation-1.0	
	iiupdatereservation.ucd	br.compose.iiupdatereservation-1.0	
	iitakeupreservation.ucd	br.compose.iitakeupreservation-1.0	

	iiprocessnoshow.ucd	br.compose.iiprocessnoshow-1.0	
	iidentifyreservation.ucd	br.compose.iidentifyreservation -1.0	
	iihospedemgt.ucd	br.compose.iihospedemgt-1.0	
	iihotelmgt.ucd	br.compose.iihotelmgt-1.0	
Component Specs and Architecture	Component Spec and Architecture.upd	br.compose.icustomermgr-1.0	Ind. Comp. Spec.
		br.compose.ihotelmgr-1.0	
		br.compose.ireservationsystem-1.0	
		br.compose.architecture-1.0	Resource

Tabela 6: Assets gerados no estágio Component Interaction.

No estágio *Component Interaction* foram gerados arquivos referentes aos tipos de artefato *Business Operations*, *Interfaces and Opers* e *Component Specs and Architecture*. Para o tipo de artefato *Business Operations*, foram gerados seis arquivos referentes a diagramas de colaboração, que foram empacotados em um único *asset* do tipo *Resource*. Para o tipo de artefato *Interfaces and Opers*, foram gerados seis diagramas de classes para representação das interfaces de sistema e de negócio. A partir destes diagramas, foram gerados seis *assets* do tipo *Independent Interface*, que especificam as interfaces de sistema e de negócio. A partir do tipo de artefato *Component Specs and Architecture* foi gerado um diagrama de componentes, a partir do qual foram criados três *assets* do tipo *Independent Component Specification*, que descrevem de forma independente de modelos de componentes os três componentes especificados. Além disto, também foi gerado um *asset* do tipo *Resource*, para representar a arquitetura dos componentes.

A partir dos arquivos e *assets* produzidos nos estágios *Component Identification* e *Component Interaction*, foram gerados arquivos e *assets* no estágio *Component Specification*, que são apresentados na Tabela 7.

Tipo de Artefato	Arquivos	Identificador do Asset	Tipo
	IIM IMakeReservation.ucd	br.compose.iim-1.0	
	IIM ICancelReservation.ucd		

Interface Information Models	IIM IUpdateReservation.ucd	br.compose.iim-1.0	Resource
	IIM ITakeUpReservation.ucd		
	IIM IProcessNoShow.ucd		
	IIM IIdentifyReservation.ucd		
	IIM IHospedeMgt.ucd		
	IIM IHotelMgt.ucd		
Operation Pre/postconditions	PrePostConditions.txt	br.compose.prepostconditions-1.0	
Component Interface Constraints	Component-Interface Constraints.txt	br.compose.cic-1.0	
Interface Specification Diagrams	SDImakeReservation.ucd	br.compose.sdimakereservation-1.0	
	SDICancelReservation.ucd	br.compose.sdicancelreservation-1.0	
	SDIupdateReservation.ucd	br.compose.sdiupdatereservation-1.0	
	SDItakeUpReservation.ucd	br.compose.sditakeupreservation-1.0	
	SDIprocessNoShow.ucd	br.compose.sdiprocessnoshow-1.0	
	SDIidentifyReservation.ucd	br.compose.sdiidentifyreservation -1.0	
	SDIhospedeMgt.ucd	br.compose.sdihospedemgt-1.0	
	SDIhotelMgt.ucd	br.compose.sdihotelmgt-1.0	

Tabela 7: Assets gerados no estágio Component Specification.

O estágio *Component Specification* produz quatro tipos de artefatos de saída. O primeiro tipo, chamado *Interface Information Models*, produziu oito arquivos de diagramas de classes, que foram empacotados em um só *asset* do tipo *Resource*.

O segundo tipo de artefato, chamado *Operation Pre/postconditions*, gerou um arquivo de texto, descrevendo pré e pós-condições em OCL, representado por um *asset* do tipo *Resource*, chamado *br.compose.prepostconditions-1.0*.

Para o tipo de artefato *Component Interface Constraints* foi gerado um arquivo de texto representando em OCL as invariantes das interfaces especificadas. Este arquivo foi descrito e empacotado em um *asset* do tipo *Resource*.

Para o último tipo de artefato, chamado *Interface Specification Diagrams*, foram gerados oito diagramas de classes representando diagramas de especificação de interfaces, que deram origem a oito *assets* distintos, do tipo *Resource*.

4.2.4 Workflow Provisioning

Após os *assets* do estágio *Component Specification* serem gerados, o *workflow Specification* foi concluído. Assim, em seguida foram gerados os *assets* do *workflow Provisioning*, onde os componentes especificados são implementados. É importante notar que, seguindo-se o processo UML Components, nas fases anteriores foram gerados *assets* independentes de modelos de componentes. As especificações dependentes de modelos foram geradas apenas no *workflow Provisioning*, em um momento anterior à própria implementação dos componentes.

Tipo de Artefato	Arquivos	Identificador do Asset	Tipo
Components	InvalidReservationDetails.java	br.compose.javainvalidreservationdetails-1.0	Dependent Exception
	InvalidReservationDetails.cs	br.compose.csinvalidreservationdetails-1.0	
	InvalidRoom.java	br.compose.javainvalidroom-1.0	
	InvalidRoom.cs	br.compose.csinvalidroom-1.0	
	InvalidCustomer.java	br.compose.javainvalidcustomer-1.0	
	InvalidCustomer.cs	br.compose.csinvalidcustomer-1.0	
	InvalidHotelDetails.java	br.compose.javainvalidhoteldetails-1.0	
	InvalidHotelDetails.cs	br.compose.csinvalidhoteldetails-1.0	
	IMakeReservation (.java e .cs)	br.compose.dimakereservation-1.0	Dependent Interface
	ICancelReservation (.java e .cs)	br.compose.dicancelreservation-1.0	
	IUpdateReservation (.java e .cs)	br.compose.diupdatereservation-1.0	
Components	ITakeUpReservation (.java e .cs)	br.compose.ditakeupreservation-1.0	
	IProcessNoShow (.java e .cs)	br.compose.diprocessnoshow-1.0	
	IIdentifyReservation (.java e .cs)	br.compose.diidentifyreservation -1.0	

	IHospedeMgt (.java e .cs)	br.compose.dihospedemgt-1.0	
	IHotelMgt (.java e .cs)	br.compose.dihotelmgt-1.0	
	Nenhum	br.compose.djbreservationsystem-1.0 br.compose.dejbreservationsystem-1.0 br.compose.ddotnetreservationsystem-1.0	Dependent Component Specification
		br.compose.djbcustomermgr-1.0 br.compose.dejbcustomermgr-1.0 br.compose.ddotnetcustomermgr-1.0	
		br.compose.djbhotelmgr-1.0 br.compose.dejbhotelmgr-1.0 br.compose.ddotnethotelmgr-1.0	
	ReservationSystem.jar (JavaBeans e EJB) e ReservationSystem.dll	br.compose.cijbreservationsystem-1.0 br.compose.ciejbreservationsystem-1.0 br.compose.cidotnetreservationsystem-1.0	Component Implementation
	CustomerMgr.jar (JavaBeans e EJB) e CustomerMgr.dll	br.compose.cijbcustomermgr-1.0 br.compose.ciejbcustomermgr-1.0 br.compose.cidotnetcustomermgr-1.0	
	HotelMgr.jar (JavaBeans e EJB) e HotelMgr.dll	br.compose.cijbhotelmgr-1.0 br.compose.ciejbhotelmgr-1.0 br.compose.cidotnethotelmgr-1.0	

Tabela 8: Assets gerados no workflow Provisioning.

Conforme mostra a Tabela 8, foram gerados oito *assets* do tipo *Dependent Exception*, a partir de arquivos de exceções nas linguagens Java e C#. Estas exceções foram produzidas a partir das pré-condições identificadas no estágio *Component Specification*. Elas foram expressas nestas linguagens devido ao fato de que os componentes especificados foram implementados com os modelos de componentes *JavaBeans*, EJB e .NET, sendo que os dois primeiros utilizam a linguagem Java e o último pode utilizar a linguagem C#.

Também foram produzidos oito arquivos de interfaces nas linguagens de programação Java e C#, dado que os componentes foram implementados de acordo com os modelos *JavaBeans*, EJB e .NET. Cada um destes arquivos foi descrito em um *asset* do tipo *Dependent Interface*.

Ainda no *workflow Provisioning*, foram criados nove *assets* do tipo *Dependent Component Specification*, já que cada um dos três componentes especificados seria implementado em três modelos de componentes: *JavaBeans*, EJB e .NET. Pelo mesmo motivo, também foram criados nove *assets* do tipo *Component Implementation*, porém, estes contêm os arquivos referentes aos códigos executáveis dos componentes implementados.

Além dos *workflows* apresentados, o processo UML Components também define os *workflows Assembly*, *Test* e *Deployment* [Cheesman01], porém estes estão fora do escopo deste trabalho, já que visam a construção de aplicações inteiras a partir dos componentes especificados, a realização de testes e a instalação das aplicações construídas, respectivamente.

Apresentados os *assets* gerados, a Seção 4.3 apresenta a utilização da ferramenta X-Packager para a descrição e o empacotamento destes *assets*.

4.3 Descrição dos Assets usando X-Packager

A Seção 4.2 apresentou os *assets* que foram produzidos em cada fase do processo UML Components que era completada. Isto foi feito a fim de oferecer uma visão geral dos *assets* que foram gerados durante o processo, de acordo com os artefatos produzidos. A partir disto, esta seção tem a finalidade de apresentar os principais aspectos da utilização da ferramenta X-Packager usando as descrições das principais informações de alguns dos *assets* previamente apresentados. A fim de não tornar esta seção redundante, a descrição de um mesmo tipo de informação é apresentada apenas uma vez, embora vários *assets* possam ter este tipo de informação representada. Por exemplo, as descrições de dos *assets* apresentados na Seção 4.2 incluem informações sobre visibilidade e modelos de negócios, porém estes aspectos não são apresentados para todos os *assets*, mas apenas uma vez.

Devido à expressiva quantidade de formulários para preenchimento de informações sobre os *assets*, apenas os formulários considerados mais importantes são apresentados. As próximas seções apresentam as descrições dos *assets* agrupadas pelos profiles definidos pelo X-ARM.

4.3.1 X-ARM Model Profile

Para o empacotamento de *assets* do *X-ARM Model Profile*, a ferramenta X-Packager apresenta os formulários para inserção das informações definidas no *X-ARM Artifact Profile* e no *RAS Default Profile*, além de formulários específicos para entrada de informações sobre os modelos que estão sendo descritos. No entanto, a criação de *assets* descritos com o *X-ARM Artifact Profile* é apresentada mais à frente, na Seção 4.3.2, onde também são apresentados os aspectos comuns a todos os tipos de *assets* definidos pelo X-ARM. Assim, as próximas subseções apresentam apenas as características e os formulários específicos de *assets* descritos com o *X-ARM Model Profile*.

4.3.1.1 Processo de Desenvolvimento

Considerando que para a realização do exemplo de uso foi utilizado o processo UML Components, a Figura 61 apresenta duas das telas utilizadas para a especificação de *assets* do tipo *Development Process*, preenchidas com informações sobre este processo. Além disto, a Figura 61 mostra ao fundo, a tela do próprio ambiente Eclipse.

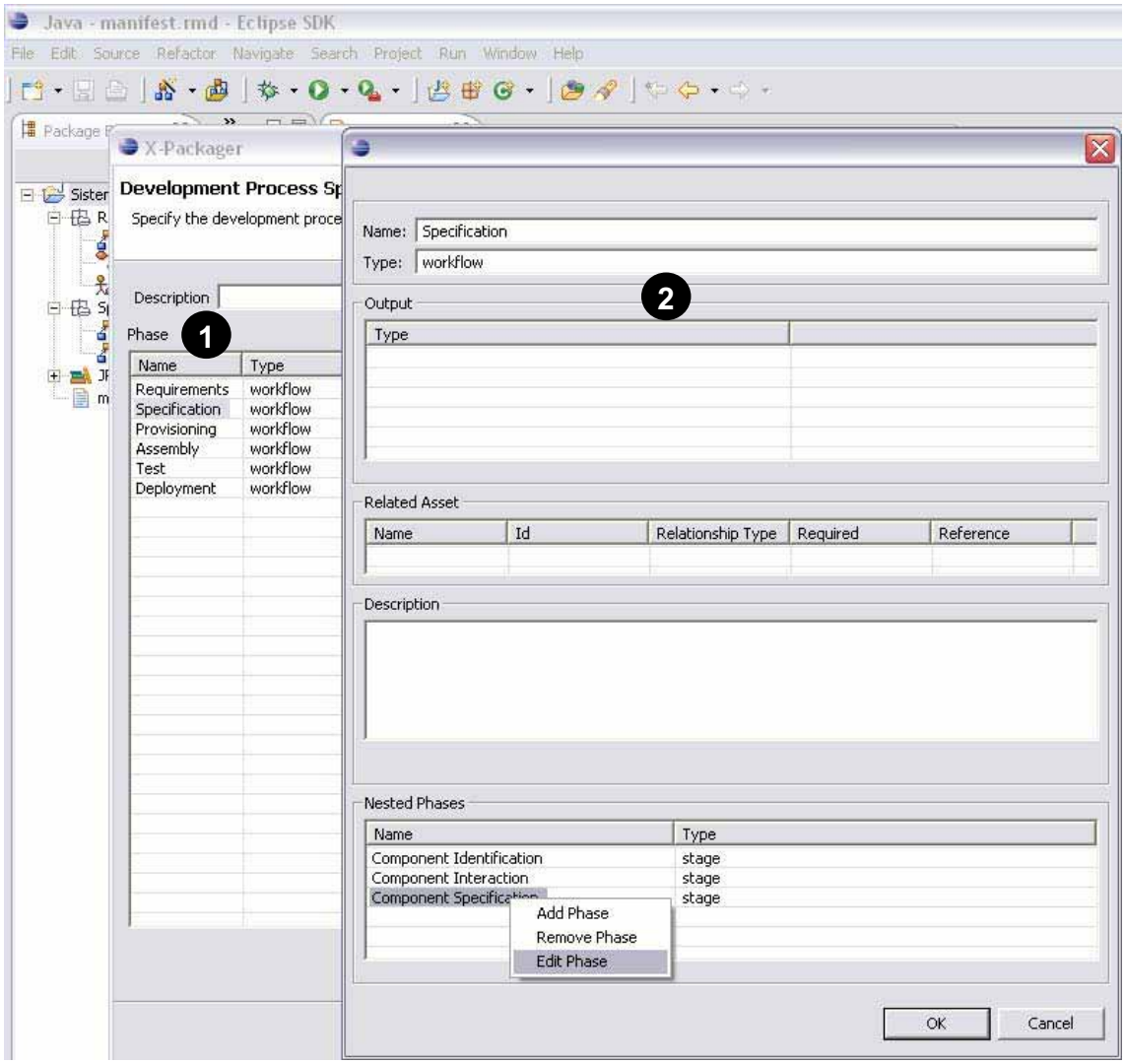

Figura 61: Telas para especificação de processos de desenvolvimento.

A tela identificada pelo número 1 refere-se ao formulário para especificação de processos de desenvolvimento. Neste formulário existe uma tabela onde são inseridas as fases que compõem o processo especificado. Ao clique com o botão direito do mouse sobre o nome "*Specification*", um menu *pop-up* é apresentado, permitindo a adição de uma nova fase, e a remoção e alteração dos dados da fase selecionada. Quando a opção de alteração é escolhida, a Tela 2 é apresentada, exibindo os dados inseridos para aquela fase e permitindo que os mesmos possam ser alterados.

Na Tela 2 é possível notar que foram inseridos dados significando que o *workflow Specification* é composto pelos estágios *Component Identification*, *Component Interaction*

e *Component Specification*, que também podem possuir outras fases aninhadas. O trecho X-ARM gerado na especificação do processo UML Components é apresentado na Figura 62.

```
1. <asset id="br.compose.umlcomponents-1.0" ...>
2.          <process-model>
3.                      <phase name="Requirements" type="workflow">
4.                                  <output type="Business Process"/>
5.                                  <output type="Business Concept Model"/>
6.                                  <output type="Use Cases"/>
7.                                  <description artifact-id="1"/>
8.                      </phase>
9.                      <phase name="Specification" type="workflow">
10                     <phase name="Component Identification" type="stage">
11                                          <output type="Business Type Model"/>
12                                          <output type="Business Interfaces"/>
13.                                         <output type="System Interfaces and Opers"/>
14.                                         <output type="Initial Comp Specs and Architecture"/>
15.                                 </phase>
16.                                 <phase name="Component Interaction" type="stage">
17.                                         <output type="Business Operations"/>
18.                                         <output type="Interfaces and Opers"/>
19.                                         <output type="Component Specs and Architecture"/>
20.                                 </phase>
21.                                 <phase name="Component Specification" type="stage">
22.                                         <output type="Interface Information Models"/>
23.                                         <output type="Operation Pre/Postconditions"/>
24.                                         <output type="Component Interface Constraints"/>
25.                                         <output type="Interfaces"/>
26.                                 </phase>
27.                                 <description>The second workflow defined by UML Components.</description>
28.                     </phase>
29.                     <phase name="Provisioning" type="workflow">
30.                                 <output type="Components"/>
31.                     </phase>
32.                     <phase name="Assembly" type="workflow">
33.                                 <output type="Applications"/>
34.                     </phase>
35.                     <phase name="Test" type="workflow">
36.                                 <output type="Tested applications"/>
37.                     </phase>
38.                     <phase name="Deployment" type="workflow"/>
39.         </process-model>
...
40. </asset>
```

Figura 62: Especificação do processo UML Components gerada pelo X-Packager.

No trecho X-ARM apresentado na Figura 62 foram declarados os seis *workflows* que compõem o processo UML Components, além dos três estágios que compõem o *workflow Specification*. Também é possível notar os tipos de artefatos que são gerados em cada uma das fases do processo.

4.3.1.2 Áreas de Aplicação

As telas do X-Packager utilizadas para a especificação das áreas de aplicação, definidas em um *asset* do tipo *Application Areas*, são apresentadas na Figura 63.

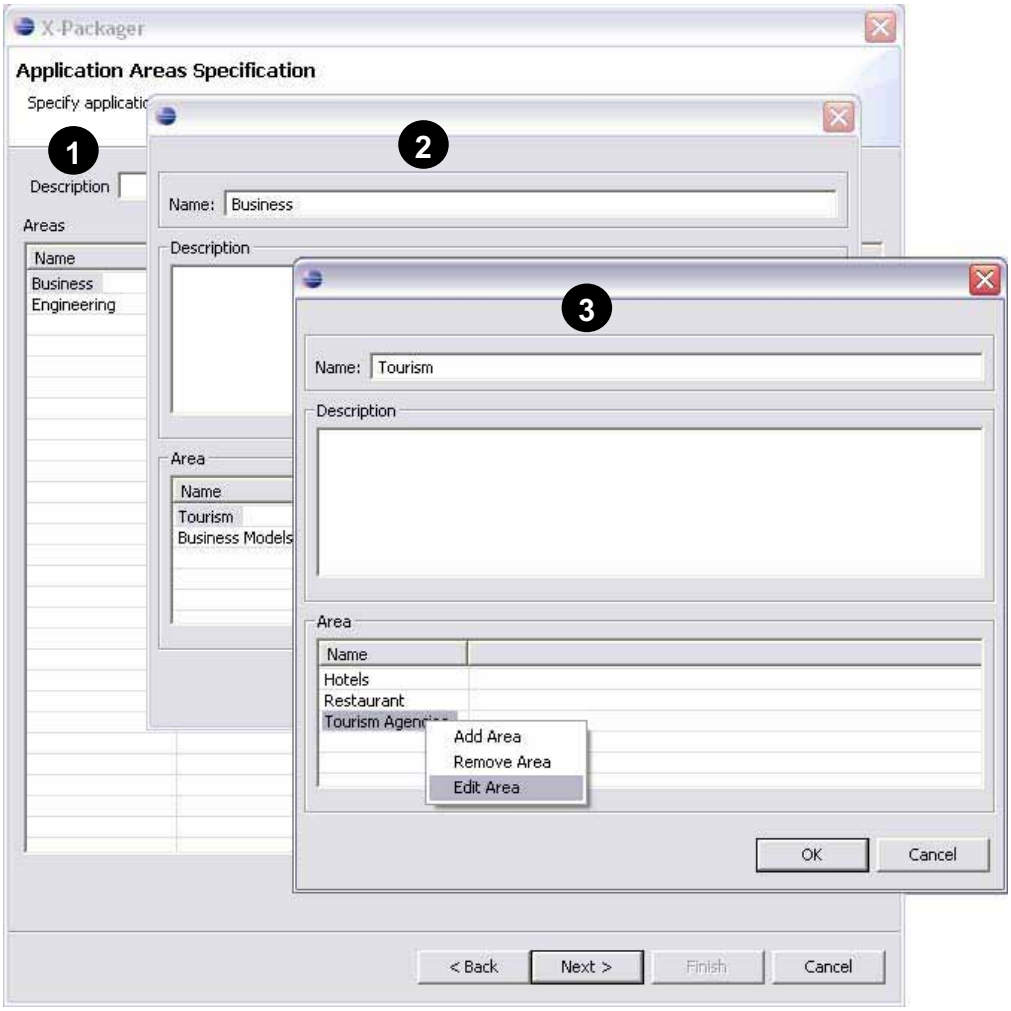

Figura 63: Telas para especificação de áreas de aplicação.

A Tela 1 é bastante parecida com a tela para especificação de processos de desenvolvimento, apresentada na Figura 61. Ela refere-se ao formulário para especificação de áreas de aplicação. Neste formulário existe uma tabela onde são inseridas as áreas que são especificadas. Para cada área nesta tabela podem ser especificadas áreas mais especializadas. Assim, na tela identificada pelo número 2 são especificadas sub-áreas de *Business*. Da mesma forma, na Tela 3 são especificadas as sub-áreas de *Tourism*, que é uma sub-área de *Business*.

O resultado da geração do *asset* de áreas de aplicação é apresentado na Figura 64, onde são mostradas todas as áreas especificadas.

```
1. <asset id="br.compose.areas-1.0" …>
2.          <app-areas-model>
3.                    <area name="Business">
4.                              <area name="Tourism">
5.                                        <area name="Hotels"/>
6.                                        <area name="Restaurant"/>
7.                                        <area name="Tourism Agencies"/>
8.                              </area>
9                               <area name="Business Models"/>
10.                   </area>
11.                   <area name="Engineering">
12.                             <area name="Civil Construction"/>
13.                             <area name="Mechanic"/>
14.                             <area name=”Food”/>
15.                   </area>
16.         </app-areas-model>
...
17. </asset>
```

Figura 64: Especificação de áreas de aplicação gerada pelo X-Packager.

Na figura são apresentadas algumas áreas, com sub-áreas mais especializadas. Assim, para exemplificar, na figura podem ser identificadas as áreas “*Business*”, “*Business.Tourism*” e “*Business.Tourism.Hotels*”, onde a primeira é mais genérica que a segunda, que, por sua vez, é mais genérica que a terceira.

É importante destacar que no contexto de um sistema de repositório, podem ser armazenados *assets* que atendem necessidades de diversas áreas de aplicação. Assim, é possível que determinados *assets* não se adeqüem às áreas de aplicação já especificadas, requerendo que outros *assets* do tipo *Application Areas* sejam especificados. Neste caso, o ideal seria a existência de *assets* deste tipo que definissem o maior número possível de áreas de aplicação, a fim de ser evitada a criação de um número muito grande destes *assets*. Entretanto, é natural que na evolução de um repositório que adote o X-ARM sejam criados alguns *assets* do tipo *Application Areas*, que vão sendo gradativamente estendidos, em novas versões mais abrangentes, as quais tendem a ser mais adotadas em relação às versões iniciais.

4.3.1.3 Modelo de Certificação

A Figura 65 ilustra a utilização da ferramenta na especificação de um modelo chamado *ISO 9126-based*, que representa um subconjunto do modelo ISO 9126 [ISO01]. Para o exemplo de uso foi utilizado este modelo resumido a fim de simplificar o exemplo, já que o modelo ISO 9126 define inúmeros aspectos, que seguem uma organização igual à dos aspectos do modelo resumido.

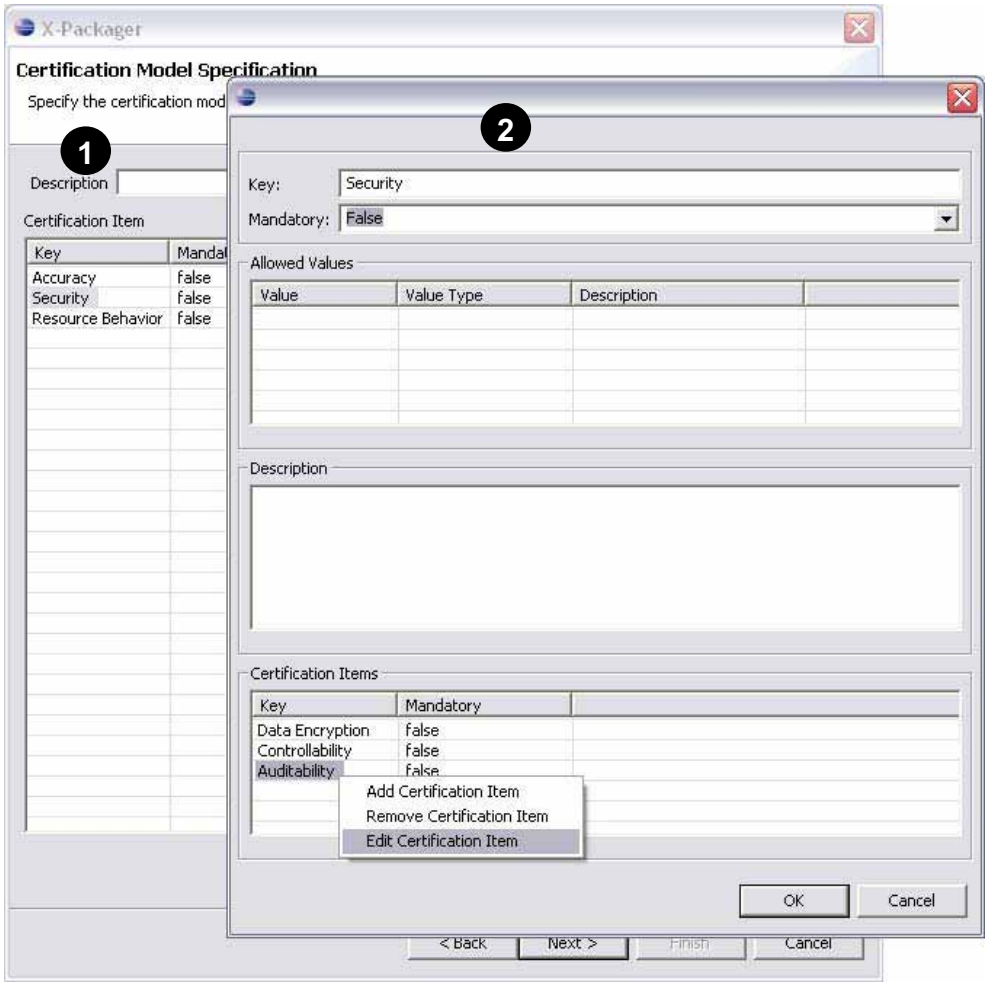

Figura 65: Telas para especificação de modelos de certificação.

Na Figura 65, a Tela 1 refere-se ao formulário para especificação de itens de certificação sendo apresentados. Na figura, pode-se notar os itens cujas chaves são "*Accuracy*", "*Security*" e "*Resource Behavior*". A Tela 2 é apresentada quando se pede para alterar o item de certificação identificado por "*Security*". Esta tela permite a entrada de uma descrição textual e a definição dos valores permitidos para o item "*Security*", além de itens de certificação aninhados, que foram definidos na figura.

A especificação X-ARM gerada do modelo *ISO 9126-based* é apresentada na Figura 66.

```
1. <asset id="br.compose.certificationmodel-1.0" name="ISO 9126-based" …>
2.          <certification-model>
3.                      <certification-item key="Accuracy" mandatory="false">
4.                                  <certification-item key="Correctness" mandatory="false">
5.                                              <allowed-value value="0..100" value-type="integer"/>
6.                                  </certification-item>
7.                      </certification-item>
8.                      <certification-item key="Security" mandatory="false">
9.                                  <certification-item key="Data Encryption" mandatory="false">
10.                                             <allowed-value value="yes" value-type="string"/>
11.                                             <allowed-value value="no" value-type="string"/>
12.                                 </certification-item>
13.                                 <certification-item key="Controllability" mandatory="false">
14.                                             <allowed-value value="0..100" value-type="integer"/>
15.                                 </certification-item>
16.                                 <certification-item key="Auditability" mandatory="false">
17.                                             <allowed-value value="yes" value-type="string"/>
18.                                             <allowed-value value="no" value-type="string"/>
19.                                 </certification-item>
20.                     </certification-item>
21.                     <certification-item key="Resource Behavior" mandatory="false">
22.                                 <certification-item key="Memory Utilization" mandatory="false">
23.                                             <allowed-value value="*" value-type="integer"/>
24.                                 </certification-item>
25.                                 <certification-item key="Disk Utilization" mandatory="false">
26.                                             <allowed-value value="*" value-type="integer"/>
27.                                 </certification-item>
28.                     </certification-item>
29.         </certification-model>
…
30. </asset>
```

Figura 66: Especificação de modelo de certificação gerada pelo X-Packager.

Na figura estão presentes os itens *Accuracy*, *Security* e *Resource Behavior*. Para o primeiro é definido o sub-item *Correctness*, cujos valores permitidos são números inteiros de 0 a 100. O segundo item, *Security*, define os sub-itens *Data Encryption*, *Controllability* e *Auditability*. *Data Encryption* e *Auditability* têm "*yes*" e "*no*" como valores permitidos. Já o item *Controllability* pode possuir valores inteiros entre 0 e 100. Por fim, o item *Resource Behavior* define os itens *Memory Utilization* e *Disk Utilization*, que podem ter quaisquer valores textuais.

4.3.1.4 Modelo de Negócios

As telas do X-Packager para especificação de modelos de negócios são semelhantes às telas para especificação de modelos de certificação, já que os elementos XML para representação destes modelos também são semelhantes. Na especificação de *assets* do tipo *Business Model*, são utilizadas as telas apresentadas na Figura 67, pertencentes ao X-Packager. Estas telas apresentam dados sobre o modelo de negócios *pay-per-copy*, que foram gerados e inseridos na ferramenta durante a realização do exemplo de uso.

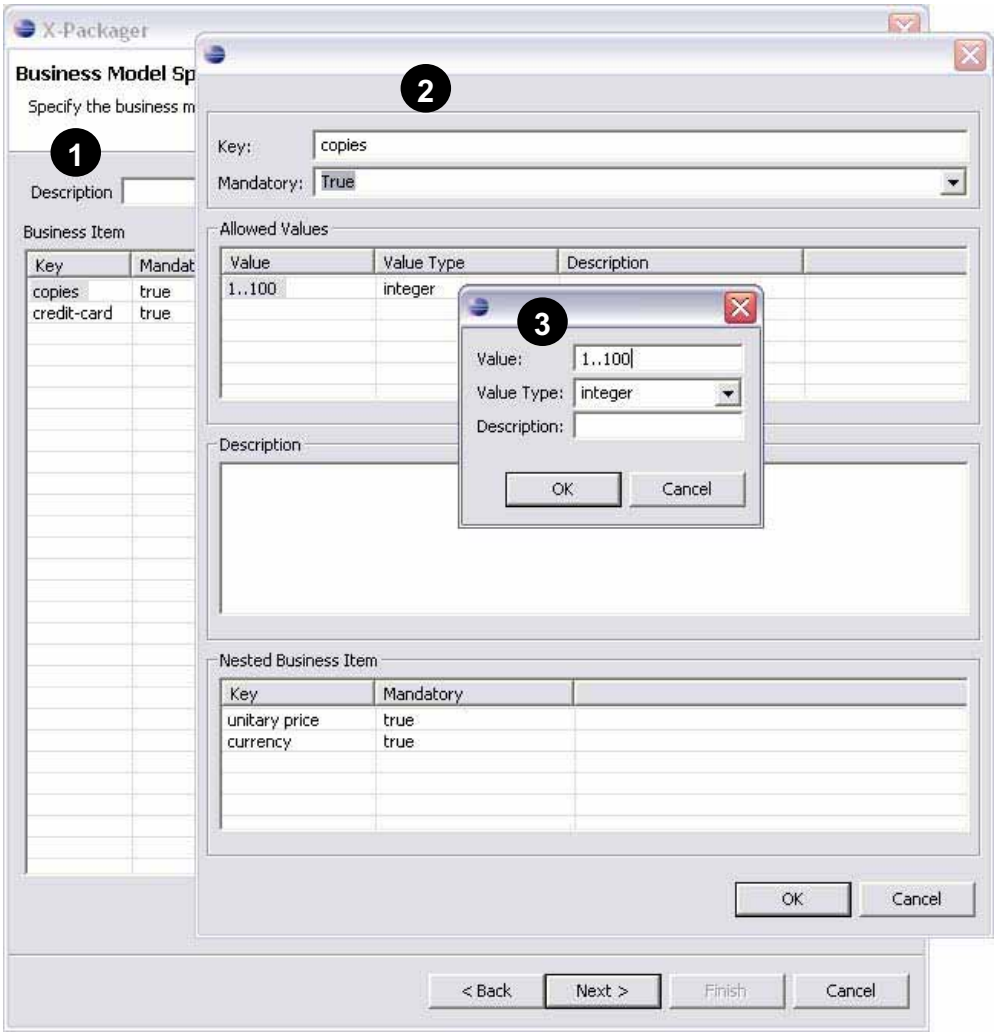

Figura 67: Telas para especificação de modelos de negócios.

No contexto deste trabalho, este modelo define que podem ser adquiridas até 100 cópias dos *assets* que adotarem este modelo. Neste caso, este modelo define que os *assets* que adotarem este modelo devem especificar seu preço e a moeda, que pode ser Dólar ou Real. Por fim, os cartões de crédito possíveis para este modelo de negócios são *Visa*, *Mastercard* e *American Express*.

Na Figura 67, a Tela 1 refere-se ao formulário para adição, remoção e atualização dos itens que, no exemplo de uso, definem o modelo de negócios *pay-per-copy*. É possível perceber na figura os itens obrigatórios cujas chaves são "*copies*" e "*credit-card*". A Tela 2 é utilizada para adição e atualização dos itens de negócios. Neste caso, a tela apresenta os detalhes do item *copies*, cujos valores permitidos e itens de negócios aninhados são

identificados. Por fim, a Tela 3 apresenta a tela apresentada pelo X-Packager para alteração de valores permitidos.

A especificação completa do modelo de negócios *pay-per-copy* gerada pelo X-Packager é apresentada na Figura 68.

```
1. <asset id="br.compose.businessmodel-1.0" name="pay-per-copy" …>
2.          <business-model>
3.                    <business-item key="copies" mandatory="true">
4.                              <allowed-value value="1..100" value-type="integer"/>
5.                              <business-item key="unitary price" mandatory="true">
6.                                        <allowed-value value="0..*" value-type="unsigned float"/>
7.                              </business-item>
8.                              <business-item key="currency" mandatory="true">
9.                                        <allowed-value value="R$" value-type="string"/>
10.                                       <allowed-value value="US$" value-type="string"/>
11.                             </business-item>
12.                   </business-item>
13.                   <business-item key="credit-card" mandatory="true">
14.                             <allowed-value value="Visa, Mastercard" value-type="string"/>
15.                             <allowed-value value="American Express" value-type="string"/>
16.                   </business-item>
17.         </business-model>
...
18. </asset>
```

Figura 68: Especificação de modelo de negócios gerada pelo X-Packager.

A Figura 68 apresenta dois itens de negócios principais: *copies* e *credit-card*. O primeiro define os valores permitidos e apresenta outros dois itens de negócios aninhados: *unitary price* e *currency*. Já para o item *credit-card*, são definidos apenas os valores permitidos.

4.3.1.5 Modelos de Componentes

Na especificação de modelos de componentes, devem ser definidos os tipos de propriedades, interfaces e eventos que um componente pode possuir. Assim, a Figura 69 ilustra telas para especificação de modelos de componentes. Os dados preenchidos nas telas apresentadas se referem à especificação do modelo de componentes *JavaBeans* [Sun97].

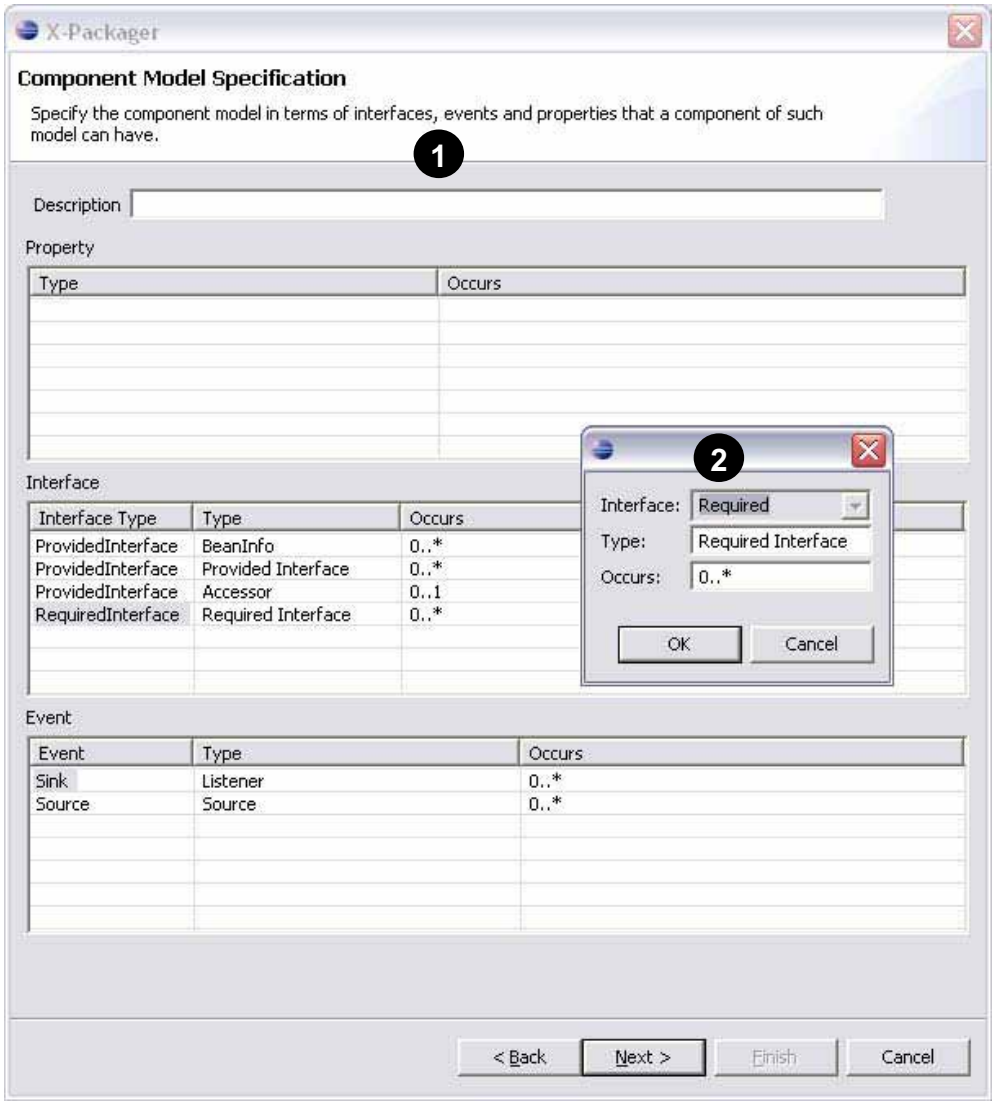

Figura 69: Telas para especificação de modelos de componentes.

Na Figura 69, a tela identificada pelo número 1 refere-se ao formulário para especificação de modelos de componentes, no qual foram definidos os tipos de interfaces e de eventos do modelo *JavaBeans*. A Tela 2 é apresentada ao solicitar a alteração da interface provida do tipo *Accessor*.

A Figura 70 apresenta a especificação do modelo *JavaBeans*, gerada com a ferramenta X-Packager.

```
1. <asset id="br.compose.javabeanscomponentmodel-1.0" …>
2.          <component-model>
3.                    <interface>
4.                              <required-interface type="Required Interface" occurs="0..*"/>
5.                              <provided-interface type="BeanInfo" occurs="0..*"/>
6.                              <provided-interface type="Provided Interface" occurs="0..*"/>
7.                              <provided-interface type="Accessor" occurs="0..1"/>
8.                    </interface>
9.                    <event>
10.                             <sink type="Listener" occurs="0..*"/>
11.                             <source type="Source" occurs="0..*"/>
12.                   </event>
13.         </component-model>
…
14. </asset>
```

Figura 70: Especificação do modelo JavaBeans gerada pelo X-Packager.

Na Figura 70, é possível notar a existência de três tipos de interfaces providas e um tipo de interface requerida, além da existência de ouvintes de eventos em *JavaBeans*.

Visto que os componentes especificados também seriam implementados com os modelos de componentes EJB [DeMichiel01] e .NET [Barnett02], foi necessário descrever estes modelos. A Figura 71 e a Figura 72 ilustram respectivamente as especificações destes modelos, também feitas com X-Packager. No caso de EJB, nota-se a existência de três tipos de interfaces providas, além da interface requerida.

```
1. <asset id="br.compose.ejbcomponentmodel-1.0" …>
2.          <component-model>
3.                    <interface>
4.                              <provided-interface type="Home" occurs="1"/>
5.                              <provided-interface type="Component" occurs="0..*"/>
6.                              <provided-interface type="Accessor" occurs="0..*"/>
7.                              <required-interface type="Required Interface" occurs="0..*"/>
8.                    </interface>
9.          </component-model>
…
10. </asset>
```

Figura 71: Especificação do modelo EJB gerada pelo X-Packager.

A especificação de .NET conta com propriedades e interfaces providas e requeridas, além de fontes e receptores de eventos. A Figura 72 apresenta a especificação deste modelo em X-ARM.

```
1. <asset id="br.compose.dotnetcomponentmodel-1.0" …>
2.          <component-model>
3.                    <interface>
4.                              <provided-interface type="Provided Interface" occurs="0..*"/>
4.                              <required-interface type="Required Interface" occurs="0..*"/>
5.                    </interface>
6.                    <event>
7.                              <source type="Event Source" occurs="0..*"/>
7.                              <sink type="Event Sink" occurs="0..*"/>
8.                    </event>
9.                    <property type="property" occurs="0..*"/>
10.         </component-model>
…
11. </asset>
```

Figura 72: Especificação do modelo .NET gerada pelo X-Packager.

4.3.2 X-ARM Artifact Profile

Depois de definidos os modelos referenciados pelas descrições dos demais tipos de *assets* gerados na realização do exemplo de uso, esta seção apresenta a descrição dos *assets* do tipo *Resource* referentes aos exemplos apresentados em [Cheesman01], gerados principalmente nos workflows *Requirements* e *Specification*. Considerando que os demais tipos de *assets* do *X-ARM Artifact Profile* também são descritos com os mesmos elementos XML, suas descrições não serão apresentadas. Vale lembrar que no caso de *assets* de outros profiles, diversos aspectos são descritos com os elementos definidos pelo *X-ARM Artifact Profile*, porém, a fim de não tornar o texto redundante, estes aspectos são apresentados apenas nesta seção.

Conforme apresentado na Tabela 2, a descrição de *assets* do *X-ARM Artifact Profile* é feita com 12 formulários do X-Packager. Por este motivo, nem todos os formulários serão apresentados, mas apenas aqueles que representarem um número relativamente maior de informações. Outro motivo que também justifica esta decisão é o fato de que diversos formulários, para descrição de diferentes características, são bastante semelhantes. Entretanto, mesmo quando os respectivos formulários não forem apresentados, as descrições em X-ARM das características representadas serão apresentadas.

Conforme apresentado na Seção 4.1.1, na descrição e empacotamento de um *asset*, é possível aproveitar descrições de outros *assets* a fim de evitar o trabalho de descrever os mesmos aspectos com as mesmas informações novamente. Assim, ao longo das próximas subseções será apresentada a descrição de apenas um *asset* referente ao tipo de artefato *Business Concept Model*, gerado no *workflow Requirements* do processo UML

Components [Cheesman01]. A descrição dos *assets* é apresentada desta forma até mesmo porque, na prática, durante a realização do exemplo de uso, o aproveitamento de descrições de *assets* já empacotados foi largamente utilizado.

4.3.2.1 Identificação

A identificação de um *asset* no X-ARM é feita com os atributos do elemento raiz, <*asset*>. Como ela é feita apenas por meio dos valores destes atributos, a ferramenta X-Packager simplesmente apresenta um conjunto de campos que devem ser preenchidos pelo usuário. A Figura 73 ilustra a identificação de um *asset*.

```
1. <asset id="br.compose.bcm-1.0" name="Business Concept Model" date="2006-05-10" last-change="2006-05-
10" owner="Compose" language="en">
```

Figura 73: Especificação da identificação do asset.

O valor "br.compose.bcm-1.0" do atributo *id* é usado para identificar unicamente o *asset* em um repositório. Conforme apresentado na Seção 3.1.6, o valor "br.compose.bcm" refere-se ao nome hierárquico do *asset*, que inclui a zona ou domínio onde o *asset* está registrado. Já o valor "1.0" refere-se à versão do *asset*. Como o *asset* não se trata de uma implementação de componente, o atributo *status* não pode ser utilizado.

Além disto, com as informações de identificação, tem-se que o nome do *asset* é *Business Concept Model*, que sua criação e sua última mudança foram no dia 10/05/2006, que o proprietário é Compose e que o idioma usado na descrição é o inglês.

4.3.2.2 Arquivos que compõem um asset

A tela para especificação dos arquivos que compõem um *asset* foi ilustrada na Figura 58. Porém, um artefato do tipo *Business Concept Model* é composto por apenas um arquivo contendo um diagrama de classes. Assim, a Figura 74 ilustra a indicação deste arquivo, chamado *Business Concept Model.ucd*, que está dentro do diretório *Requirements Workflow* no pacote do *asset*.

```
1. <solution>
2.          <artifact type="ucd" reference="Requirements Workflow/Business Concept Model.ucd" id="1"/>
3. </solution>
```

Figura 74: Especificação do arquivo que compõe o asset.

4.3.2.3 Classificação

Parte da especificação da classificação do *asset* se baseou nas áreas de aplicação especificadas na Seção 4.3.1.2. Outra parte da classificação foi especificada de forma aleatória, sem considerar um conjunto de áreas de aplicação. A Figura 75 ilustra a descrição completa da classificação.

```
1. <classification>
2.     <context name=”X-ARM Application Areas”>
3.         <descriptor-group>
4.             <related-asset name="App Areas" id="br.compose.areas-1.0"
                                relationship-type="appAreas" required="false"/>
5              <descriptor name="Business.Tourism.Hotels"/>
6.         </descriptor-group>
7.     </context>
8.     <descriptor-group name="Reservation Systems">
9.         <descriptor name="Hotels.Reservations"/>
10.    </descriptor-group>
11.</classification>
```

Figura 75: Especificação da classificação.

Nota-se que no primeiro elemento <*descriptor-group*>, filho de <*context*> (linhas de 3 a 6), foram utilizadas como descritores, aquelas áreas de aplicação especificadas no *asset* identificado por *br.compose.areas-1.0* (apresentado na Seção 4.3.1.2) que representam algum sentido em relação ao *asset* cuja classificação é descrita. Já no segundo elemento <*descriptor-group*> (linhas de 8 a 10), o descritor escolhido também tem uma relação semântica com o *asset*, porém, não obedece nenhum conjunto de áreas de aplicação.

4.3.2.4 Informações do Processo de Desenvolvimento

Como o *asset* foi desenvolvido seguindo-se um processo de desenvolvimento, foi informado o processo, a fase e o tipo de artefato. A Figura 76 apresenta a especificação destas informações.

```
1. <process-info phase="Requirements" output-type="Business Concept Model">
2.         <related-asset name="UML Components" id="br.compose.umlcomponents-1.0" relationship-type="processInfo"
                          required="false"/>
3. </process-info>
```

Figura 76: Especificação de informações sobre o processo adotado.

A Figura 76 mostra que o *asset* foi gerado seguindo-se o processo UML Components, que foi especificado no *asset* identificado por *br.compose.umlcomponents-1.0* (linha 2), que foi apresentado na Seção 4.3.1.1. Além disto, o *asset* refere-se ao tipo *Business Concept Model*, tendo sido gerado na fase chamada *Requirements* (linha 1).

4.3.2.5 Assets Reusados

Seguindo-se um processo de desenvolvimento, é normal que a construção de um *asset* reuse outros *assets*, construídos anteriormente no processo. Assim, X-ARM também permite que estes *assets* sejam indicados. A Figura 77 apresenta a informação destes *assets*.

```
1. <reused-assets>
2.         <process>
3.                 <related-asset name="UML Components" id="br.compose.umlcomponents-1.0"
                                relationship-type="processModel" required="false"/>
4.                 <phase name="Requirements">
5.                         <type name="Business Process">
6.                                 <related-asset name="Business Process" id="br.compose.businessprocess-1.0"
                                                relationship-type="reusedAsset" required="false"/>
7.                         </type>
8.                 </phase>
9.         </process>
10. </reused-assets>
```

Figura 77: Especificação dos assets reusados.

Segundo a Figura 77, a construção do *asset* descrito reusou o *asset* identificado por *br.compose.businessprocess-1.0* (linha 6). Este *asset* corresponde ao tipo de artefato *Business Process* (linha 5), gerado na fase *Requirements* (linha 4) do processo UML Components (linha 3).

4.3.2.6 Visibilidade

De acordo com a Seção 3.1.6, produtores de *assets* podem ser identificados por meio de nomes hierárquicos. A partir disto, é possível limitar quais produtores poderão ter acesso ao *asset*. Assim, a Figura 78 apresenta o esquema de visibilidade adotado, identificando produtores fictícios que poderiam ou não acessar o *asset*.

```
1. <visibility>
2.         <allow domain="br.ufpb.compose"/>
3.         <allow domain="br.ufrn">
4.                 <deny domain="br.ufrn.ppgsc"/>
5.         </allow>
6. </visibility>
```

Figura 78: Especificação da visibilidade.

O esquema adotado considera a existência de pelo menos três produtores de *assets*, sendo que os produtores identificados por *br.ufpb.compose* e *br.ufrn* (linhas 2 e 3 respectivamente) podem acessar o *asset*. Entretanto, o produtor identificado por *br.ufrn.ppgsc* não pode acessar o *asset* (linha 4). Caso a linha 4 não fosse especificada, o

produtor identificado por *br.ufrn.ppgsc* estaria automaticamente permitido a acessar o *asset*, já que *br.ufrn* está autorizado.

4.3.2.7 Histórico de Evolução

Para o *asset* descrito, considerou-se que ele estava em sua primeira versão. No entanto, a fim de verificar e demonstrar a descrição de *assets* que não estão na primeira versão, no exemplo de uso o mesmo *asset* foi descrito novamente, porém, considerando-se que uma versão anterior já havia sido criada. Assim, a descrição da evolução do *asset* ficou como mostra a Figura 79.

```
1. <change-history>
2.      <evolution type="term addition">
3.          <related-asset name="Business Concept Model" id=" br.compose.bcm-2.0" relationship-type="baseAsset"
                        required="false"/>
4.      </evolution>
5. </change-history>
```

Figura 79: Especificação da evolução para um asset baseado em outra versão.

Conforme mostra a Figura 79, o *asset* é um aperfeiçoamento do *asset* identificado por *br.compose.bcm-2.0* (linha 3), sobre o qual foi adicionada um *termo* (linha 2), conforme a nomenclatura adotada em UML Components.

4.3.2.8 Informações de Certificação

Um modelo de certificação foi definido na Seção 4.3.1.3. No entanto, como o *asset* que está sendo descrito trata-se de um modelo conceitual, os itens de certificação especificados não podem ser verificados. Assim, especificamente para o caso da certificação de *assets*, optou-se por apresentar a descrição de uma implementação de componente. A Figura 80 ilustra as telas da ferramenta X-Packager com a especificação de informações de certificação.

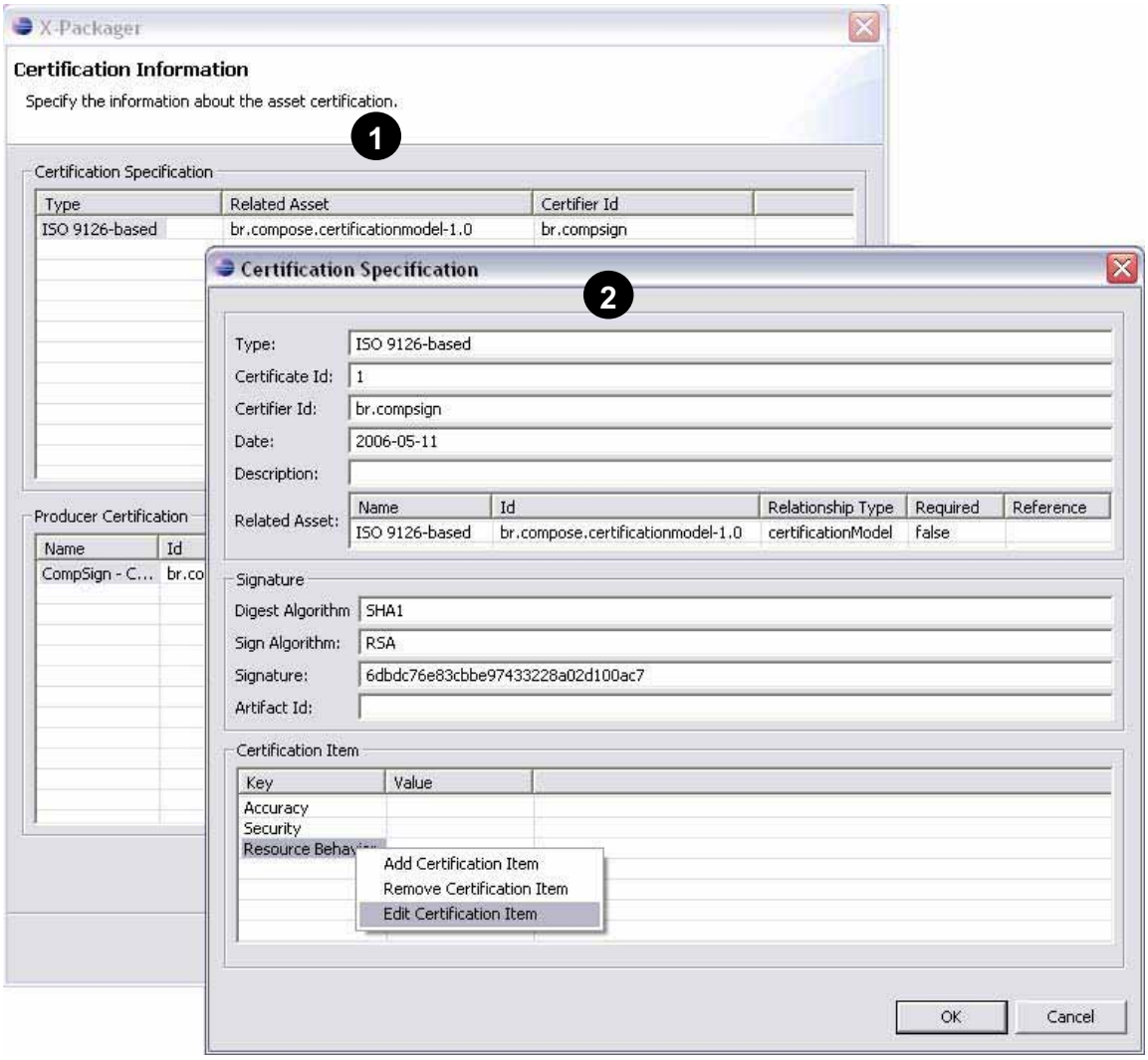

Figura 80: Telas para especificação da certificação do asset.

Na Figura 80, a tela identificada pelo número 1 apresenta o formulário onde são especificadas as certificações de *assets* e de produtores. A Tela 2 refere-se ao formulário utilizado para a especificação dos dados de uma certificação particular do *asset*.

Desta forma, primeiramente, a Figura 81 apresenta a indicação da certificação do produtor do *asset* e, em seguida, na Figura 82, é ilustrada a certificação de uma implementação de componente, também gerada para o exemplo de uso de X-Packager.

A Figura 81 mostra que a certificação do produtor do *asset* está descrita no *asset* identificado por *br.compose.producercertification-1.0*.

```
1. <certification-info>
2.          <producer-certification>
3.                    <related-asset name="CompSign - Compose" id="br.compose.producercertification-1.0"
                                relationship-type="producerCertification" required="false"/>
4.          </producer-certification>
5. </certification-info>
```

Figura 81: Indicação da certificação do produtor do asset.

Na geração dos *assets* do exemplo de uso, foi implementado o componente chamado *ReservationSystem* com os modelos de componentes *JavaBeans*, EJB e .NET. Assim, informações sobre as certificações destes *assets* foram descritas conforme o modelo de certificação apresentado na Seção 4.3.1.3, identificado por *br.compose.certificationmodel-1.0*. Portanto, visto que certificações de quaisquer tipos de *assets* são descritas com o *X-ARM Artifact Profile*, a Figura 82 ilustra a certificação de uma destas implementações do componente.

```
1. <certification-info>
2.    <certification type="ISO 9126-based" certifier-id="br.compsign" certificate-id="1" date="2006-05-11">
3.       <related-asset id="br.compose.certificationmodel-1.0" relationship-type="certificationModel" required="false"…/>
4.       <certification-item key="Accuracy">
5.          <certification-item key="Correctness">
6.             <value>90</value>
7.          </certification-item>
8.       </certification-item>
9.       <certification-item key="Security">
10.         <certification-item key="Data Encryption">
11.            <value>no</value>
12.         </certification-item>
13.         <certification-item key="Controllability">
14.            <value>95</value>
15.         </certification-item>
16.         <certification-item key="Auditability">
17.            <value>yes</value>
18.         </certification-item>
19.      </certification-item>
20.      <certification-item key="Resource Behavior">
21.         <certification-item key="Memory Utilization">
22.            <value>1Mb</value>
23.         </certification-item>
24.         <certification-item key="Disk Utilization">
25.            <value>0.5Mb</value>
26.         </certification-item>
27.      </certification-item>
28.      <signature digest-algorithm="SHA1" sign-algorithm="RSA">
            6dbdc76e83cbbe97433228a02d100ac7
29.      </signature>
30.   </certification>
31. </certification-info>
```

Figura 82: Especificação da certificação do asset.

Para a certificação do *asset*, foi adotado o modelo de certificação identificado por *br.compose.certificationmodel-1.0* (linha 3), que foi apresentado na Seção 4.3.1.3. A corretude do *asset* foi avaliada em 90% (linhas 5 a 7). Informações sobre segurança (linhas

9 a 19) incluíram cifragem dos dados, não oferecida; possibilidade de controle da segurança, estimada em 95%; e auditabilidade, suportada pelo *asset*. Por fim, a certificação também verificou o comportamento dos recursos de máquina (linhas 20 a 27), constatando o consumo de 1 Mb de memória e de 0,5 Mb de disco rígido.

4.3.2.9 Informações de Negociação

As informações de negociação do *asset* se basearam na especificação apresentada na Seção 4.3.1.4, possuindo os itens especificados, com os respectivos valores permitidos. A Figura 83 apresenta as informações de negociação do *asset*. Dado que as telas da ferramenta X-Packager para especificação de informações de negociação são semelhantes às telas para especificação de negociação, elas não serão apresentadas.

```
1. <business-info>
2.          <business type="pay-per-copy" business-id="1">
3.                      <business-item key="copies">
4.                                  <value>1..10</value>
5.                                  <business-item key="unitary price"> <value>10,00</value>  </business-item>
6.                                  <business-item key="currency"> <value>R$</value> </business-item>
7.                      </business-item>
8.                      <business-item key="copies">
9.                                  <value>11..100</value>
10.                                 <business-item key="unitary price"> <value>8,00</value> </business-item>
11.                                 <business-item key="currency"> <value>R$</value> </business-item>
12.                     </business-item>
13.                     <business-item key="credit-card"> <value>Visa</value> </business-item>
14.                     <broker id="br.compose.broker"/>
15.                     <broker id="com.componentsource"/>
16.                     <related-asset name="pay-per-copy" id="br.compose.businessmodel-1.0"
                                    relationship-type="businessModel" required="false"/>
17.                     <license type="GPL"/>
18.                     <support time="2" unit="years"/>
19.         </business>
20. </business-info>
```

Figura 83: Especificação das informações de negociação.

O modelo de negócios utilizado é o *pay-per-copy* (linha 2), sendo que o *asset* é vendido a R$ 10,00 se forem adquiridas de uma a dez cópias (linhas de 3 a 7), ou a R$ 8,00 se forem adquiridas de 11 a 100 cópias (linhas de 8 a 12). O cartão de crédito que pode ser usado para a compra do *asset* é *Visa* (linha 13), e o *asset* pode ser vendido pelos negociadores fictícios identificados por *br.compose.broker* e *com.componentsource* (linhas 14 e 15 respectivamente). O *asset* que especifica os itens de negócios e valores que podem ser usados nas informações de negociação é identificado por *br.compose.businessmodel-1.0* (linha 16). Por fim, a licença é GPL e o tempo de suporte para o *asset* é de 2 anos (linhas 17 e 18).

4.3.3 X-ARM Interaction Profile

Assim como no caso do *X-ARM Model Profile*, para o empacotamento de *assets* do *X-ARM Interaction Profile*, a ferramenta X-Packager apresenta diversos formulários para entrada dos dados definidos pelo *X-ARM Artifact Profile*. Assim, nesta seção são apresentados apenas os formulários e descrições X-ARM referentes aos aspectos presentes somente em *assets* do *X-ARM Interaction Profile*.

Nesta seção são apresentadas descrições apenas de *assets* dependentes de modelos de componentes. Isto se deve ao fato de que os *assets* dependentes descrevem as mesmas informações dos *assets* independentes, porém acrescentando informações referentes aos modelos de componentes adotados.

4.3.3.1 Interfaces

Conforme apresentado na Seção 4.2.3, durante a realização do exemplo de uso foram definidas oito interfaces, sendo seis interfaces de sistema e duas interfaces de negócio, todas descritas com a ferramenta X-Packager. Porém, a fim de evitar redundâncias, aqui será apresentada apenas a interface *IMakeReservation*, já que ela é uma das principais interfaces definidas. Suas operações são apresentadas na Figura 84.

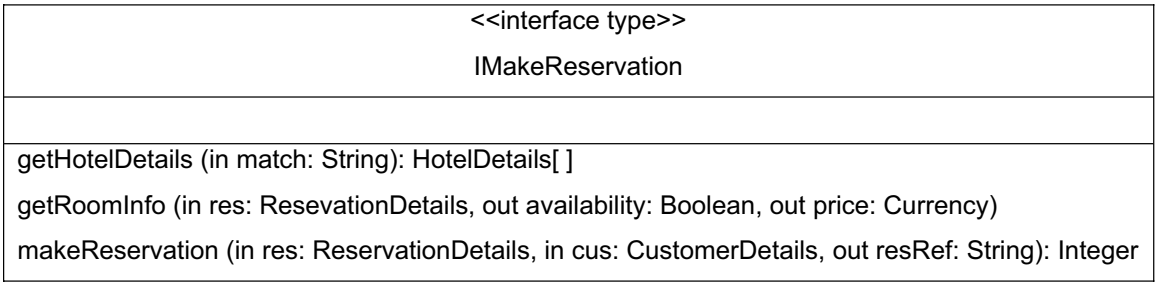

Figura 84: A interface IMakeReservation.

A operação *getHotelDetails* é responsável por recuperar as informações do hotel cujo identificador foi informado. A operação *getRoomInfo* tem a função de descobrir informações de um quarto com as características fornecidas. Por fim, a operação *makeReservation* efetua uma reserva. A partir disto, a Figura 85 apresenta as telas para especificação das operações e invariantes da interface *IMakeReservation*.

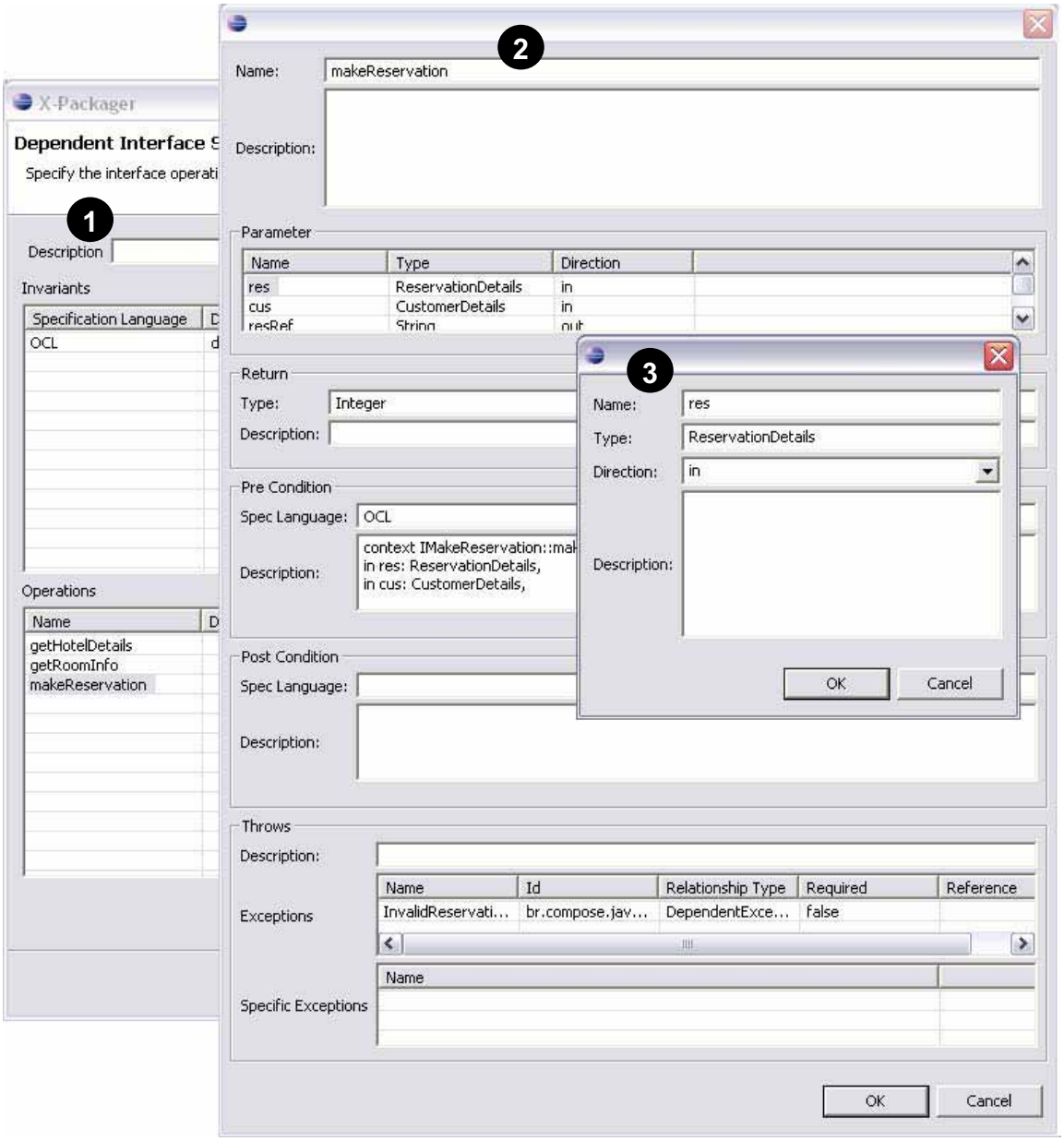

Figura 85: Telas para especificação de interfaces.

A Tela 1 mostra os formulários para especificação das operações e invariantes da interface. A tela identificada pelo número 2 é apresentada na adição ou alteração de uma operação. Na Figura 85, os dados preenchidos nesta tela se referem à operação *makeReservation*. Por fim, a Tela 3 é usada na adição ou alteração de um parâmetro da operação especificada na Tela 2.

Na especificação de interfaces independentes de modelos de componentes, apenas as telas apresentadas na Figura 85 são apresentadas. Já no caso de interfaces dependentes de modelos, também é necessário informar dados referentes à dependência de modelo, tais como o modelo de componentes, os arquivos que contêm a descrição da interface e a interface independente de modelo na qual a interface dependente se baseou. Assim, a Figura 86 apresenta a especificação em X-ARM da interface *IMakeReservation*, cujos dados foram informados à ferramenta X-Packager.

```
1. <asset id="br.compose.dimakereservation-1.0" ...>
2.          <interface>
3.                    <invariant language-spec="OCL">
4.                              <description>context r : Reservation inv :
                                        -- a reservation is claimed if it has a room allocated to it
                                        r.claimed = r.allocation->notEmpty</description>
5.                    </invariant>
6.                    <operation name="getHotelDetails">
7.                              <parameter name="match" type="String" direction="in"/>
8.                              <return type="HotelDetails[]"/>
9.                    </operation>
10.                   <operation name="getRoomInfo">
11.                             <parameter name="res" type="ReservationDetails" direction="in"/>
12.                             <parameter name="availability" type="Boolean" direction="out"/>
13.                             <parameter name="price" type="Currency" direction="out"/>
14.                   </operation>
15.                   <operation name="makeReservation">
16.                             <parameter name="res" type="ResevationDetails" direction="in"/>
17.                             <parameter name="cus" type="CustomerDetails" direction="in"/>
18.                             <parameter name="resRef" type="String" direction="out"/>
19.                             <pre-condition language-spec="OCL">
20.                                       <description>context IMakeReservation::makeReservation (
                                                  in res: ReservationDetails,
                                                  in cus: CustomerDetails,
                                                  out resRef: String): Boolean
                                                  pre:
                                                        -- the hotel id and room type are valid
                                                        hotel->exists(h | h.id = res.hotel and
                                                        h.room.roomType.name->includes(res.roomType))
21.                                       </description>
22.                             </pre-condition>
23.                             <return type="Integer"/>
24.                             <throws>
25.                                   <related-asset name="InvalidReservationDetails"
                                                  id="br.compose.javainvalidreservationdetails-1.0"
                                                  relationship-type="dependentException" required="false"/>
26.                             </throws>
27.                   </operation>
28.         </interface>
…
29. </asset>
```

Figura 86: Especificação em X-ARM da interface IMakeReservation.

Na Figura 86 estão especificadas uma invariante (linhas 3 a 5) e as operações definidas na interface (linhas 6 a 27), que foram apresentadas na Figura 84. Para a operação *makeReservation* é possível perceber que além dos parâmetros e do retorno, também foram definidas uma pré-condição (linhas 19 a 22) e uma exceção lançada (linhas 24 a 26),

identificada por *br.compose.exceptions.dird-1.0*, que é especificada mais à frente, na Seção 4.3.3.3.

4.3.3.2 Eventos

Apesar de não ter sido definido nenhum evento na modelagem realizada em [Cheesman01], um evento foi criado e descrito com a ferramenta X-Packager. Este evento foi implementado seguindo-se as regras definidas pela especificação de *JavaBeans* [Sun97]. O evento tem a função de notificar seus receptores sobre a realização de reservas e é chamado *ReservationCreatedEvent*.

Neste evento, foi definida uma interface chamada *ReservationCreatedListener*, que deve ser implementada pelos *beans* interessados em receber as notificações. Estas notificações são transmitidas aos ouvintes por meio de objetos do tipo *ReservationEvent*, que estendem *java.util.EventObject*. Já os *beans* que são fontes de eventos devem implementar a interface *ReservationCreatedSource*, que possui os métodos *addReservationCreatedListener* e *removeReservationCreatedListener*, utilizados para registrar e remover receptores interessados em ser notificados sobre a criação de uma reserva. No exemplo de uso, o *bean* fonte é o componente *ReservationSystem* e o *bean* receptor foi chamado *SinkBean*. A Figura 87 ilustra os componentes, interfaces e objeto envolvidos na utilização evento.

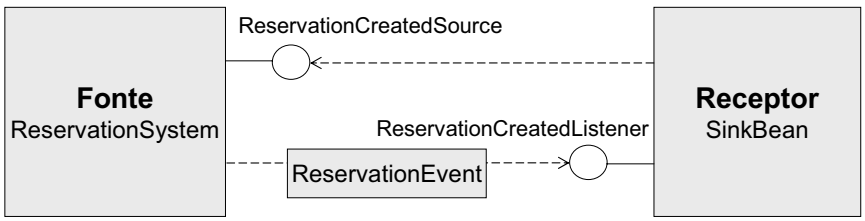

Figura 87: Ilustração do evento criado.

Com base nisto, as telas utilizadas pelo X-Packager para entrada de informações sobre o evento são ilustradas na Figura 88.

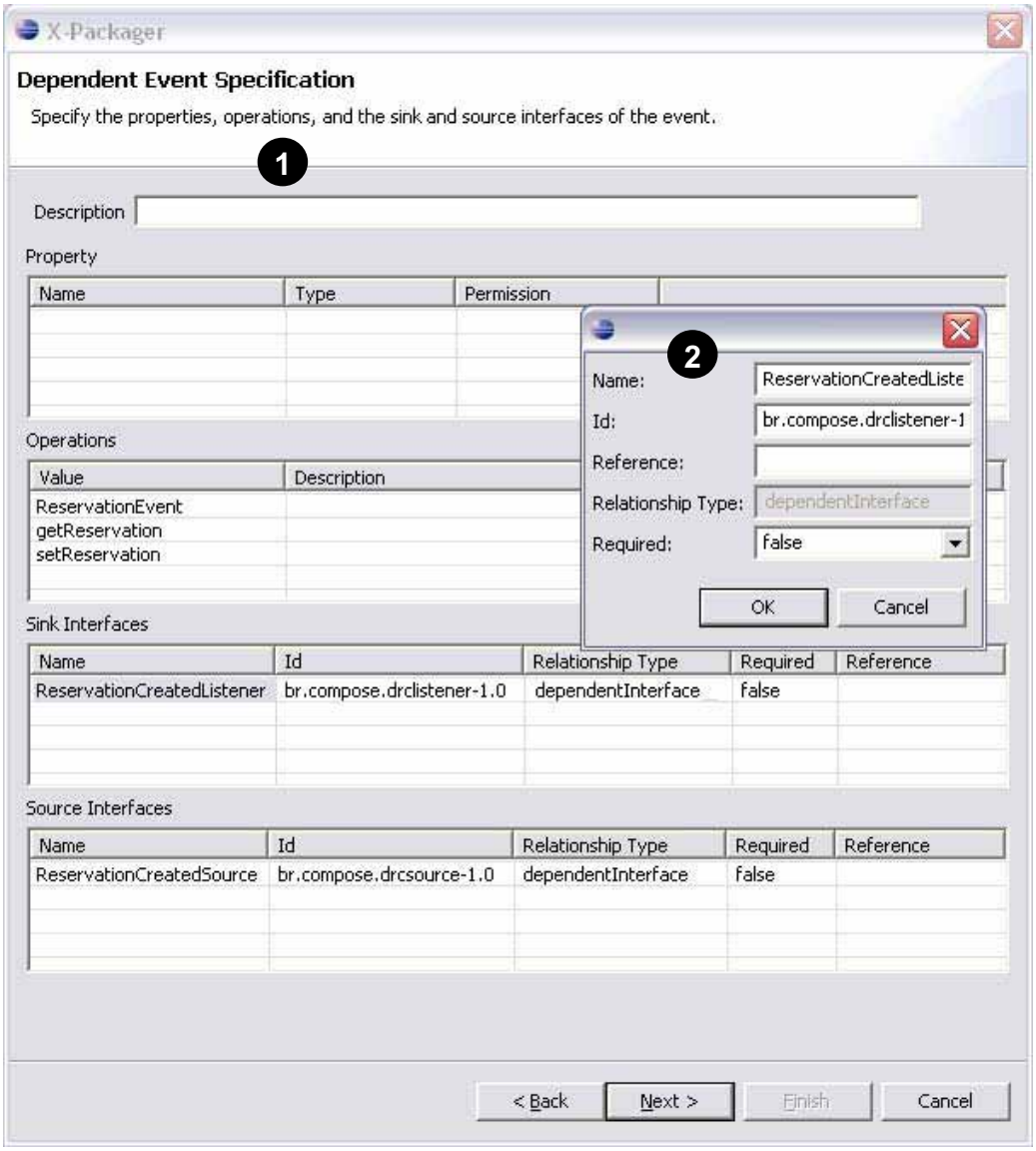

Figura 88: Telas para especificação de eventos.

A tela identificada pelo número 1 refere-se ao formulário para entrada de informações sobre propriedades e operações do evento, além da indicação das interfaces a serem implementadas pelos componentes fontes e receptores de eventos. A Tela 2 é utilizada para adicionar ou alterar interfaces de receptores de eventos.

Por ser um evento dependente de modelos de componentes, informações sobre a dependência de modelo também devem ser informadas, assim como ocorre no caso de interfaces. Estas informações são referentes ao modelo de componentes, à descrição do evento em uma liguagem própria definida pelo modelo e ao evento independente de modelo

no qual o evento dependente se baseou. A especificação do evento é apresentada na Figura 89.

```
1. <asset id="br.compose.dreservationcreatedevent-1.0" ...>
2.          <event>
3.                  <operation name="ReservationEvent">
4.                          <parameter name="source" type=”Object” direction=”in”/>
5.                          <parameter name="res" type=”ReservationDetails” direction=”in”/>
6.                  </operation>
7.                  <operation name="getReservation">
8.                          <return type=”ReservationDetails”/>
9.                  </operation>
10.                 <operation name="setReservation">
11.                         <parameter name="res" type=”ReservationDetails” direction=”in”/>
12.                         <return type=”void”/>
13.                 </operation>
14.                 <sink-interface>
15.                         <related-asset name="ReservationCreatedListener" id="br.compose.drclistener-1.0"
                                    relationship-type="dependentInterface" required="true"/>
16.                 </sink-interface>
17                  <source-interface>
18.                         <related-asset name="ReservationCreatedSource" id="br.compose.drcsource-1.0"
                                    relationship-type="dependentInterface" required="true"/>
19.                 </source-interface>
20.                 <model-dependency>
21.                         <related-asset name="JavaBeans Component Model" relationship-type="componentModel"
                                    id="br.compose.javabeanscomponentmodel-1.0" required="false"/>
22.                 </model-dependency>
23.         </event>
…
24. </asset>
```

Figura 89: Especificação em X-ARM do evento ReservationCreatedEvent.

Como mostra a figura, o evento possui três operações (linhas 3 a 13) e as interfaces a serem implementadas por componentes receptores e fontes do evento são identificadas por *br.compose.drclistener-1.0* e *br.compose.drcsource-1.0* (linhas 14 a 19). Além disto, o modelo de componentes no qual o evento se baseou está especificado no *asset* identificado por *br.compose.javabeanscomponentmodel-1.0*.

4.3.3.3 Exceções

Conforme apresentado na Seção 4.3.3.1, a operação *makeReservation*, definida pela interface *IMakeReservation* lança uma exceção chamada *InvalidReservationDetails*, que também foi especificada utilizando a ferramenta X-Packager. A Figura 90 ilustra as telas usadas para especificação de exceções.

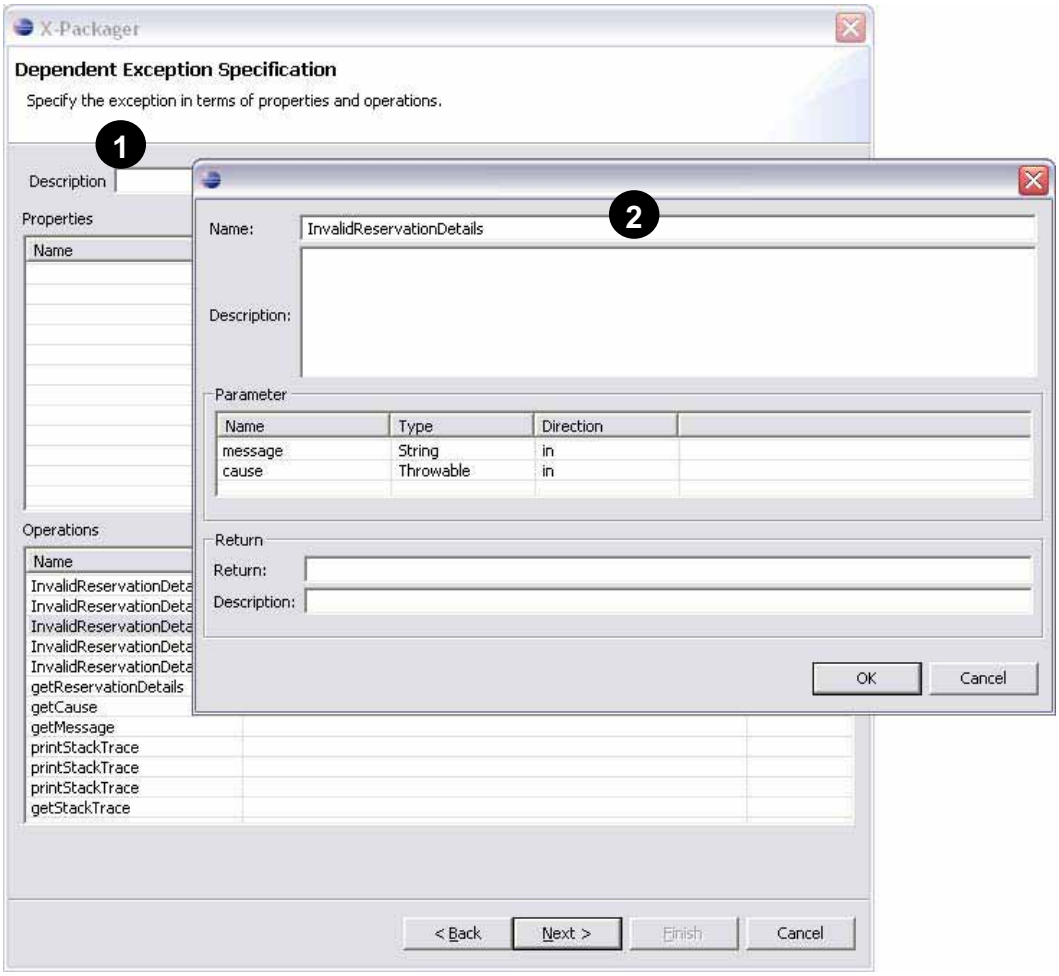

Figura 90: Telas para especificação de exceções.

A tela identificada pelo número 1, apresenta os formulários usados para especificação de propriedades e operações que compõem a exceção. Já a Tela 2 serve para a adição ou alteração de operações da exceção. Além das telas apresentadas na Figura 90, assim como no caso de interfaces e eventos dependentes de modelos de componentes, para exceções dependentes de modelos, a ferramenta X-Packager também apresenta a tela para entrada de informações sobre a dependência de modelos de componentes.

O resultado em X-ARM da especificação da exceção é apresentado na Figura 91. A exceção especificada foi criada na linguagem Java e consiste de uma extensão da classe *java.lang.Exception*, que por sua vez, é uma extensão de *java.lang.Throwable*. Assim, as operações definidas por estas classes também estão presentes na exceção especificada, *InvalidReservationDetails*.

```
1. <asset id=”br.compose.javainvalidreservationdetails-1.0”>
2.          <exception>
3.                      <operation name="InvalidReservationDetails"/>
4.                      <operation name="InvalidReservationDetails">
5.                                  <parameter name=”message” type=”String” direction=”in”/>
6.                      </operation>
7.                      <operation name="InvalidReservationDetails">
8.                                  <parameter name=”message” type=”String” direction=”in”/>
9.                                  <parameter name=”cause” type=”Throwable” direction=”in”/>
10.                     </operation>
11.                     <operation name="InvalidReservationDetails">
12.                                 <parameter name=”cause” type=”Throwable” direction=”in”/>
13.                     </operation>
14.                     <operation name="InvalidReservationDetails">
15.                                 <parameter name=”message” type=”String” direction=”in”/>
16.                                 <parameter name=”res” type=”ReservationDetails” direction=”in”/>
17.                     </operation>
18.                     <operation name="getReservationDetails">
19.                                 <return type=”ReservationDetails”/>
20.                     </operation>
21.                     <operation name="getCause">
22.                                 <return type=”Throwable”/>
23.                     </operation>
24.                     <operation name="getMessage">
25.                                 <return type=”String”/>
26.                     </operation>
27.                     <operation name="printStackTrace">
28.                                 <return type=”void”/>
29.                     </operation>
…
30.         </exception>
31.         <model-dependency>
32.                     <related-asset name="JavaBeans Component Model" relationship-type="componentModel"
                                    id="br.compose.javabeanscomponentmodel-1.0" required="false"/>
33.         </model-dependency>
…
34. </asset>
```

Figura 91: Especificação em X-ARM da exceção InvalidReservationDetails.

A Figura 91 apresenta algumas das principais operações da exceção *InvalidReservationDetails*. Além das operações herdadas de suas superclasses, esta exceção também inclui duas novas operações, com as seguintes assinaturas: *InvalidReservationDetails(String message, ReservationDetails res)* (linhas 14 a 17) e *ReservationDetails getReservationDetails()* (linhas 18 a 20). Nesta especificação também foi informado o modelo de componentes seguido (linhas 31 a 33).

4.3.4 X-ARM Component Profile

O *X-ARM Component Profile* permite descrever especificações independentes e dependentes de modelos de componentes, além de implementações de componentes. Visto que as especificações dependentes representam as mesmas informações de especificações independentes, incluindo informações sobre a dependência de modelos, apenas especificações dependentes serão apresentadas.

No exemplo de uso realizado, foram especificados três componentes, que foram ilustrados na Figura 60. Cada um destes componentes foi implementado em três diferentes modelos de componentes: *JavaBeans*, EJB e .NET. No entanto, a fim de não tornar a apresentação do exemplo de uso demasiado redundante, optou-se por apresentar a descrição da especificação dependente e a implementação apenas do componente *ReservationSystem*, com o modelo *JavaBeans*.

4.3.4.1 Dependent Component Specification

Conforme apresentado na Figura 60, a especificação do componente conta com seis interfaces providas e três interfaces requeridas, que foram especificadas durante a realização do exemplo de uso. Assim, a Figura 92 apresenta as telas da ferramenta X-Packager na especificação destas interfaces.

A tela identificada pelo número 1 corresponde ao formulário para indicação das interfaces providas e requeridas pelo componente. No caso das tabelas indicadas com os nomes "*Provided Interfaces*" e "*Required Interfaces*" são representados os identificadores de interfaces já especificadas em X-ARM. Já as tabelas identificadas por "*Specific Provided Interfaces*" e "*Specific Required Interfaces*" são usadas para informar interfaces não descritas em X-ARM, mas definidas pelos modelos de componentes utilizados. Como se pode perceber, não foram utilizadas interfaces destes tipos.

Apesar da modelagem apresentada em [Cheesman01] não incluir eventos, no para o exemplo de uso foi criado um evento, já apresentado na Seção 4.3.3.2, do qual o componente especificado é fonte.

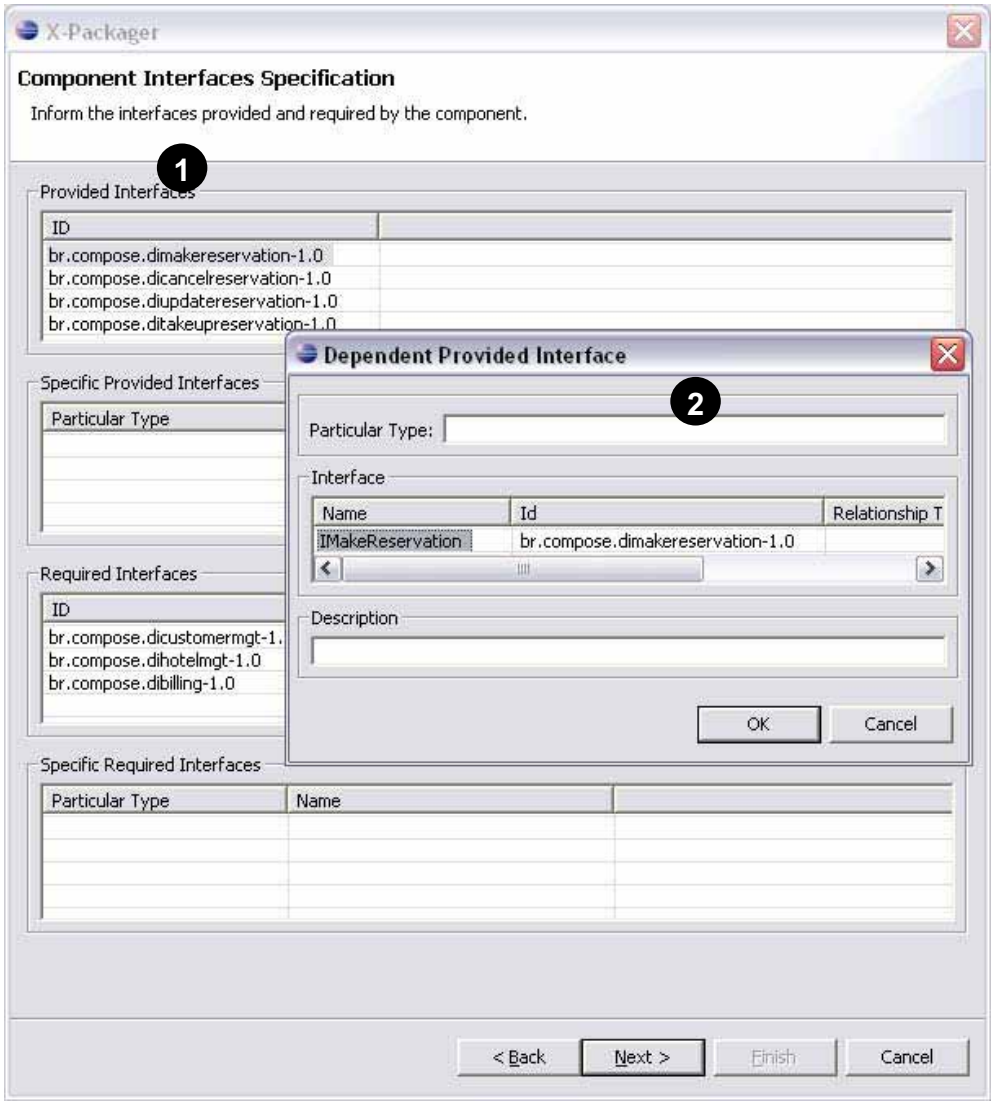

Figura 92: Telas para especificação das interfaces do componente.

Além destas informações, a ferramenta X-Packager também requer dados sobre a dependência de modelos de componentes, assim como no caso de interfaces, eventos e exceções dependentes de modelos de componentes. Estes dados são referentes ao modelo de componentes, à especificação independente de modelo no qual a especificação dependente se baseou, e aos arquivos que descrevem o componente de acordo com a notação definida pelo modelo.

A descrição em X-ARM da especificação do componente *ReservationSystem*, gerada pela ferramenta X-Packager é apresentada na Figura 93.

```
1. <asset id=”br.compose.djbreservationsystem-1.0”>
2.          <component-specification>
3.                    <interface>
4.                              <required>
5                                       <related-asset name="ICustomerMgt" id="br.compose.dicustomermgt-1.0"
                                                relationship-type="dependentInterface" required="true"/>
6.                              </required>
7.                              <required>
8                                       <related-asset name="IHotelMgt" id="br.compose.dihotelmgt-1.0"
                                                relationship-type="dependentInterface" required="true"/>
9.                              </required>
10.                             <required>
11.                                     <related-asset name="IBilling" id="br.compose.dibilling-1.0"
                                                relationship-type="dependentInterface" required="true"/>
12.                             </required>
13.                             <provided>
14.                                     <related-asset name="IMakeReservation" id="br.compose.dimakereservation-1.0"
                                                relationship-type="dependentInterface" required="true"/>
15.                             </provided>
16.                             <provided>
17.                                     <related-asset name="ICancelReservation"
                                                id="br.compose.dicancelreservation-1.0"
                                                relationship-type="dependentInterface" required="true"/>
18.                             </provided>
19.                             <provided>
20.                                     <related-asset name="IUpdateReservation"
                                                id="br.compose.diupdatereservation-1.0"
                                                relationship-type="dependentInterface" required="true"/>
21.                             </provided>
22.                             <provided>
23.                                     <related-asset name="ITakeUpReservation"
                                                id="br.compose.ditakeupreservation-1.0"
                                                relationship-type="dependentInterface" required="true"/>
24.                             </provided>
25.                             <provided>
26.                                     <related-asset name="IProcessNoShow"
                                                id="br.compose.diprocessnoshow-1.0"
                                                relationship-type="dependentInterface" required="true"/>
27.                             </provided>
28.                             <provided>
29.                                     <related-asset name="IIdentifyReservation"
                                                id="br.compose.diidentifyreservation-1.0"
                                                relationship-type="dependentInterface" required="true"/>
30.                             </provided>
31.                   </interface>
32.                   <event>
33.                             <source>
34.                                     <related-asset name="ReservationCreatedEvent"
                                                id="br.compose.dreservationcreatedevent -1.0"
                                                relationship-type="dependentEvent" required="true"/>
35.                             </source>
36.                   </event>
37.         </component-specification>
38.         <model-dependency>
39.                   <related-asset name="JavaBeans Component Model" relationship-type="componentModel"
                                id="br.compose.javabeanscomponentmodel-1.0" required="false"/>
40.         </model-dependency>
…
41. </asset>
```

Figura 93: Especificação em X-ARM do componente *ReservationSystem*.

Na Figura 93 é possível perceber a existência de três interfaces requeridas (linhas 4 a 12), seis interfaces providas (linhas 13 a 30) e de uma fonte de evento (linhas 32 a 36).

Além disto, a especificação X-ARM do modelo de componentes *JavaBeans* também é indicada (linhas 38 a 40).

4.3.4.2 Component Implementation

A partir da especificação do componente *ReservationSystem* apresentada na Seção 4.3.4.1, esta seção apresenta a descrição da implementação deste componente. A ferramenta X-Packager possui três telas para especificação de implementações de componentes. A primeira é usada para a indicação dos artefatos contidos no pacote do *asset*, que contém o código executável, o código fonte e a documentação do componente. Para a entrada de informações sobre requisitos não funcionais, a ferramenta possui outra tela. A última tela usada na descrição de implementações de componentes representa a descrição do componente e sua composição por outros componentes.

Utilizando-se a ferramenta X-Packager, foi gerada a descrição em X-ARM da implementação do componente *ReservationSystem*, que é apresentada na Figura 94.

```
1. <asset id=”br.compose.cijbreservationsystem-1.0”>
2.          <component-implementation>
3.                    <source-code>
4.                              <file artifact-id=”10”/>
5.                              <file artifact-id=”11”/>
6.                              <file artifact-id=”12”/>
7.                              <file artifact-id=”13”/>
8.                              <file artifact-id=”14”/>
9.                              <file artifact-id=”15”/>
10.                             <file artifact-id=”16”/>
11.                   </source-code>
12.                   <exec-code>
13.                             <file artifact-id=”7”/>
14.                   </exec-code>
15.                   <documentation>
16.                             <file artifact-id=”5”/>
17.                   </documentation>
18.                   <nonfunctional-requirements>
19.                             <processor value=”x86”/>
20.                             <OS value=”Linux” version=”Fedora, Ubuntu, Mandriva, Debian, Red Hat”/>
21.                             <OS value=”Windows” version=”2000, XP”/>
22.                             <platform value=”Java” version=”1.5”/>
23.                             <mem-size value=”5” unit=”Mb”/>
24.                             <disk-size value=”10” unit=”Kb”/>
25.                             <language value=”Java” />
26.                   </nonfunctional-requirements>
27.         </component-implementation>
28.         <model-dependency>
29.                   <related-asset name="JavaBeans Component Model" relationship-type="componentModel"
                                  id="br.compose.javabeanscomponentmodel-1.0" required="false"/>
30.                   <related-asset name="Dependent ReservationSystem" relationship-type="dependentComponentSpec"
                                  id="br.compose.djbreservationsystem-1.0" required="false"/>
31.         </model-dependency>
…
32. </asset>
```

Figura 94: Especificação em X-ARM do componente *ReservationSystem*.

Como mostra a Figura 94, o código executável e a documentação do componente são compostos por apenas um arquivo (linhas 12 a 17), cada um, enquanto que o código fonte é composto por sete arquivos (linhas 3 a 11). Os requisitos não funcionais (linhas 18 a 26) especificam que o componente deve ser usado com processadores da família x86, com sistema operacional Linux em diversas distribuições ou Windows nas versões 2000 e XP. Além disto, a plataforma de execução é Java na versão 1.5, até mesmo porque o componente foi construído na linguagem Java. Por fim, o espaço de memória requerido pelo componente é de 5 Mb e o espaço em disco ocupado é 10 Kb.

4.4 Considerações Finais

Com a utilização da ferramenta X-Packager, têm-se basicamente três vantagens principais. A primeira refere-se ao fato de não ser necessário realizar o empacotamento do *asset* de forma *ad-hoc*, ou seja, não é necessário que um documento XML de manifesto seja construído manualmente e que o usuário crie um pacote ZIP com os arquivos contidos pelo *asset* e o próprio arquivo de manifesto. Como uma segunda vantagem, tem-se o fato de não ser necessário que quem descreve o *asset* conheça detalhes da sintaxe do modelo X-ARM. Esta é uma importante vantagem, dada a quantidade de elementos, atributos e restrições definidos pelo X-ARM. Por fim, pelo fato da ferramenta X-Packager ter sido desenvolvida na forma de um *plug-in* para o ambiente Eclipse [Eclipse06], tem-se a vantagem da funcionalidade de empocotamento de *assets* estar integrada ao próprio ambiente de desenvolvimento dos artefatos que compõem os *assets*.

No caso da última vantagem apontada, tem-se como aspecto positivo, o fato do ambiente Eclipse permitir a própria incorporação de *plug-ins*, fazendo com que *plug-ins* com diferentes funcionalidades possam ser utilizados na geração de variados tipos de artefatos. Por outro lado, pode surgir o argumento de que assim, o empacotamento de *assets* desenvolvidos com outros ambientes de desenvolvimento não pode se beneficiar do *plug-in*. No entanto, a ferramenta X-Packager também pode ser utilizada como uma aplicação isolada. Neste caso, os artefatos podem ser desenvolvidos em diferentes ambientes, e no momento do empacotamento de um *asset*, os artefatos que o compõem devem ser importados em um projeto do Eclipse, para que a ferramenta seja utilizada.

Além de apresentar a ferramenta X-Packager, este capítulo também apresentou um exemplo de uso que envolveu a geração dos artefatos definidos em [Cheesman01] e o empacotamento dos respectivos *assets* com a ferramenta X-Packager. A ferramenta foi capaz de gerar as descrições em X-ARM de todos os *assets* definidos, além de ter conseguido empacotá-los e desempacotá-los adequadamente. No entanto, com o objetivo de não tornar a apresentação das descrições dos *assets* redundante, nem todos os *assets* foram apresentados.

Dada a grande quantidade de *assets* gerados na construção dos componentes ao seguir o processo UML Components e considerando-se o pequeno porte do sistema, infere-se que sistemas médios ou grandes devem gerar quantidades ainda maiores de *assets*. Neste sentido, a ferramenta X-Packager é de grande utilidade para a descrição de *assets* com o modelo X-ARM, principalmente se considerada sua capacidade de reaproveitamento de descrições de outros *assets*.

A partir do exemplo de uso apresentado, foi possível constatar que todas as informações relevantes associadas aos artefatos gerados são representadas com o modelo X-ARM. Por outro lado, apesar de realizar a descrição e empacotamento de *assets* adequadamente, dado que a ferramenta X-Packager ainda se encontra em uma versão inicial, aspectos de usabilidade precisam ser aperfeiçoados, já que ainda é necessário que seus usuários tenham noções sobre o modelo X-ARM.

5 Considerações Finais

Neste trabalho, foi especificado um modelo de representação de *assets*, denominado X-ARM, cuja representação é feita em XML, sendo útil na indexação, busca e recuperação padronizadas de *assets* mantidos em um repositório. O modelo proposto é dividido em quatro categorias e 18 tipos, permitindo a representação das características dos diferentes tipos de *assets* de maneira apropriada.

Apesar da existência de outros modelos com objetivos semelhantes, o X-ARM constitui uma importante contribuição para o DBC, uma vez que supera algumas limitações significativas de outros modelos. Exemplos de limitações superadas são: representação de diversos modelos de negócios, controle de visibilidade, representação da certificação de *assets*, reuso de interfaces e eventos, suporte a diferentes processos de desenvolvimento, representação de componentes de diversos modelos de componentes existentes atualmente na indústria (tais como CCM, EJB, .NET e COM) e a representação de componentes de possíveis futuros modelos.

Além das características apresentadas, não suportadas por outros modelos de descrição de *assets*, outras funcionalidades, herdadas do modelo RAS [RAS05], também estão presentes no modelo, tais como identificação, classificação, informações sobre uso dos *assets* e o próprio empacotamento dos artefatos que compõem os *assets*.

Neste trabalho também foi apresentada a ferramenta X-Packager, desenvolvida com o objetivo de facilitar a descrição e empacotamento de *assets* conforme o modelo X-ARM. A necessidade da existência de uma ferramenta como esta surge da grande quantidade de características descritas pelo X-ARM, o que praticamente impossibilita a descrição e empacotamento dos *assets* de maneira *ad-hoc*. A ferramenta foi desenvolvida como um

plug-in para o Eclipse [Eclipse06], possibilitando que *assets* gerados em projetos desenvolvidos com este ambiente possam ser empacotados de forma integrada. Apesar disto, nada impede que a ferramenta seja utilizada para empacotar *assets* cujos artefatos foram desenvolvidos com outros ambientes e ferramentas.

Com a finalidade de validar as funcionalidades do modelo X-ARM e da ferramenta X-Packager, este trabalho também apresentou um exemplo de uso, que envolveu a utilização da ferramenta X-Packager para o empacotamento de um conjunto de *assets*, que foram gerados a partir dos artefatos gerados na modelagem de exemplo apresentada em [Cheesman01]. Este exemplo de uso foi importante no sentido de mostrar que o modelo X-ARM é eficaz na prática, permitindo a descrição de *assets* reais, gerados em um processo de desenvolvimento real. Além disto, este exemplo de uso possibilitou aplicar e testar a ferramenta X-Packager, produzindo como resultado um exemplo prático de sua utilização e a constatação de que aspectos de usabilidade ainda podem ser melhorados.

Afora as contribuições do ponto de vista das características representadas pelo modelo proposto, este trabalho também apresenta contribuições relevantes relativas a determinadas abordagens empregadas pelo X-ARM. A Seção 5.1 apresenta estas contribuições.

5.1 Contribuições

Além da variedade de características descritas pelo modelo X-ARM, apresentadas na seção anterior, a definição deste modelo seguiu um conjunto de abordagens centradas na flexibilidade e no reuso de especificações de *assets*.

A primeira abordagem introduzida refere-se ao fato do X-ARM adotar o uso de modelos, o que permite um número ilimitado de modelos de negócios e de componentes, dentre outros. Assim, o modelo X-ARM não restringe a sintaxe e a semântica adotadas na descrição de determinadas informações, já que *assets* podem ser comercializados sob diferentes modelos de negócios e certificados sob diferentes modelos de certificação, por exemplo. Ao invés disto, o X-ARM adota uma abordagem em que certas características são descritas em pares de chave/valor, bastando apenas que estes pares sigam as especificações dos modelos que os *assets* adotam, descritos com o *X-ARM Model Profile*. Por exemplo, se um modelo de negócios cuja especificação com o *X-ARM Model Profile* diz que o cartão de

crédito deve ser informado, *assets* que seguem este modelo devem informar o cartão de crédito que pode ser usado na aquisição do *asset*, possivelmente especificando o termo "cartão de crédito" como chave e o nome do cartão de crédito como valor.

Ao contrário do próprio RAS, o modelo X-ARM descreve separadamente as especificações e as implementações de componentes. Assim, torna-se possível que uma mesma especificação de componente seja implementada em diferentes linguagens e modelos de componentes. Esta distinção é importante no contexto de um repositório de componentes, quando um usuário realiza uma busca por uma especificação com determinadas características desejadas. A partir da especificação retornada que for mais adequada, ele escolhe aquelas implementações que aderem às suas necessidades em termos de linguagem de programação, de modelo de componentes, de requisitos não funcionais e da existência ou não de código fonte e documentação.

Outra abordagem adotada pelo X-ARM trata-se da separação entre as especificações de componentes e as interfaces e eventos. Desta forma, quando uma especificação de componente provê ou requer uma interface, ela precisa indicar o *asset* referente a esta interface. O mesmo ocorre quando uma especificação de componente é fonte ou receptora de um evento. Com esta abordagem, habilita-se o reuso de interfaces e eventos por diferentes especificações de componentes, evitando que interfaces e eventos tenham que ser descritos em cada especificação de componentes que os utilizar. Por outro lado, no contexto de um repositório, esta abordagem facilita as buscas por especificações de componentes, uma vez que se um usuário descobriu as interfaces e eventos desejados, basta procurar as especificações de componentes que adotem tais interfaces e eventos.

Por fim, tem-se como importante abordagem introduzida, a distinção entre *assets* independentes e dependentes de modelos de componentes. Neste caso, *assets* independentes de modelos podem ser vistos como *assets* gerados para atender requisitos funcionais, enquanto que *assets* dependentes de modelos de componentes acrescentam características particulares, relativas aos modelos seguidos. Assim, esta abordagem também auxilia nas buscas por *assets*, uma vez que se um usuário encontrou um *asset* independente de modelo de componentes que atende seus requisitos, basta encontrar os *assets* dependentes que seguem o modelo desejado e que sejam baseados no *asset* independente que ele encontrou. Outro aspecto positivo desta abordagem refere-se ao fato de que ela incrementa ainda mais

o reuso de descrições de *assets*, já que um *asset* independente pode dar origem a vários *assets* dependentes de modelos.

5.2 Limitações e Perpectivas

Apesar de já estar definido, ainda existem algumas limitações para a adoção ampla do modelo X-ARM, incluindo a falta de um repositório que o utilize como meio de representação de seus *assets*; e a falta de entidades de certificação e negociação dos *assets*, que se comuniquem com um repositório compatível. Outra limitação atual do X-ARM é o fato de não apoiar modelos de componentes com interfaces para geração e apresentação de fluxos contínuos de dados, tal como definido pelos modelos CM-tel [Guimarães04] e Cosmos [Lopes04]. Este tipo de interfaces não se baseia em pares de chamada e resposta, mas no estabelecimento de uma conexão entre componentes e no tráfego contínuo de dados entre os mesmos.

Quanto à ferramenta X-Packager, na realização do exemplo de uso apresentado no Capítulo 4 foi constatada uma deficiência relativa à usabilidade. A ferramenta ainda requer que os usuários tenham noções sobre o X-ARM, enquanto que o ideal é que as características do modelo não fiquem visíveis aos usuários da ferramenta X-Packager, dado o tamanho do modelo X-ARM. No entanto, este não é um problema grave, visto que a descrição e o empacotamento dos *assets* estão sendo realizados de forma efetiva, e que a usabilidade pode ser bastante melhorada acrescentando-se dicas e ajudas aos formulários da ferramenta.

O tratamento das limitações apresentadas pode ser considerado parte dos trabalhos futuros, que estão focados na concepção [Elias06b] e implementação de um repositório de componentes utilizando o X-ARM como meio de descrição de *assets*. A concepção encontra-se finalizada, enquanto que a implementação está sendo iniciada. Além disto, outros trabalhos futuros incluem a adaptação da ferramenta X-Packager de forma a se comunicar diretamente com o repositório especificado e a não requerer que alguns dados dos componentes sejam informados.

Comunicando-se diretamente com o repositório, torna-se possível a verificação da existência dos *assets* referenciados durante a própria descrição dos *assets*. Já para não requerer a informação de alguns dados, no caso de componentes baseados em Java, a

ferramenta X-Packager pode, por exemplo, utilizar reflexão [Sun98] para extração de informações sobre interfaces, operações, eventos, exceções, dentre outros, tirando proveito de convenções de codificação [Sun99a] existentes em implementações dos componentes. De forma semelhante, também existe a possibilidade de que a ferramenta X-Packager seja integrada a ferramentas de geração de diagramas UML, tais como diagramas de atividade, de casos de uso e de classes. Assim, informações sobre *assets* que contenham artefatos referentes a diagramas destes tipos também podem ser automaticamente mapeadas para descrições X-ARM.

6 Referências

[Atkinson02] Atkinson, C. et al. *Component-Based Product Line Engineering with UML*. Addison-Wesley Pub. Co., 1st edition, 2002.

[Benz03] Benz, Brian; Durant, John R. *XML Programming Bible*. Wiley Publishing, Inc., New York, 2003.

[Berners-Lee94] Berners-Lee, T.; Masinter, L.; McCahill, M. *Uniform Resource Locators (URL)*. RFC1738. December 1994. Disponível em: http://www.w3.org/Addressing/rfc1738.txt. Acessado em: 01/09/2005.

[Brooke01] Brooke, Chris; Pharoah, Andrew. *Microsoft® COM and Microsoft® .NET Framework*. Technical White Paper, Component Source, December 2001. Disponível em: http://www.componentsource.com/Services/NETWhitePaper.asp. Acessado em: 27/10/2005.

[Barnett02] Barnett, Mike; Schulte, Wolfram. *Contracts, Components, and their Runtime Verification on the .NET Platform*. Technical Report MSR-TR-2002-38, Microsoft Research, April 2002.

[CC06] Creative Commons. http://creativecommons.org. Acessado em 10/12/2005.

[CCM06] CORBA Component Model Specification. April 2006. Disponível em: http://www.omg.org/docs/formal/06-04-01.pdf. Acessado em 20/05/2006.

[Cheesman01] Cheesman, John; Daniels, John. *UML Components: A Simple Process for Specifying Component Based Software*. Addison-Wesley, 2001.

[Chung99] Chung, Lawrence; Nixon, Brain A.; Yu, Eric; Mylopoulos, John. *Non-Functional Requirements in Software Engineering*. Springer, 1999.

[CompSource05] Component Source. http://www.componentsource.com. Acessado em 06/05/2005.

[Conradi98] Conradi, R. and Westfechtel, B. 1998. *Version models for software configuration management*. ACM Computing Surveys. Volume 30, Issue 2, Pages: 232-282, June 1998.

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

[DeMichiel01] DeMichiel, Linda G.; Yalçinalp, L. Ümit; Krishnan, Sanjeev. *Enterprise JavaBeans (TM) Specification Version 2.0*. Sun Microsystems, August 2001.

[Eclipse06] Eclipse. (2006) Eclipse Consortium, Eclipse – Version 3.1. Disponível em: http://www.eclipse.org.

[Elias06a] Elias, Glêdson; Schuenck, Michael; Negócio, Yuri; Dias Jr., Jorge; Miranda Filho, Sindolfo. *X-ARM: An Asset Representation Model for Component Repository Systems*. Symposium on Applied Computing (SAC'2006), Dijon, France, April 2006.

[Elias06b] Elias, Glêdson; Dias Jr., Jorge; Miranda Filho, Sindolfo; Cavalcanti, Gustavo; Schuenck, Michael; Negócio, Yuri. *Improving Reuse of Off-the-Shelf Components with Shared, Distributed Component Repository Systems*. The 9th International Conference on Software Reuse (ICSR), Lecture Notes in Computer Science, Springer-Verlag, Torino, Italy, 2006.

[Frakes05] Frakes, William. Kang, Kyo. *Software Reuse Research: Status and Future*. IEEE Transactions On Software Engineering, Vol.31, Nº 7, July 2005.

[Frolund98] Frolund, Svend; Koistinen, Jari. *QML: A language for quality of service specification*. Technical Report HPL-98-10, Hewlett-Packard Laboratories, February 1998.

[Guimarães04] Guimarães, Eliane Gomes. *Um Modelo de Componentes para Aplicações Telemáticas e Ubíquas*. Tese de Doutorado, UNICAMP, dezembro de 2004.

[GPL91] GNU General Public License. http://www.gnu.org/copyleft/gpl.html. Acessado em 10/12/2005.

[Heineman01] Heineman, George T.; Councill, William T. *Component-based software engineering: putting the pieces together*. Addison-Wesley, 2001

[Hoff97] Hoff, Arthur van; Partovi, Hadi; Thai, Tom. *The Open Software Description Format (OSD)*. Submission to the World Wide Web Consortium (W3C), 1997. Disponível em: http://www.w3.org/TR/NOTE-OSD. Acessado em 21/06/2005.

[Inoue03] Inoue, Katsuro; Yokomori, Reishi; Fujiwara, Hikaru; Yamamoto, Tetsuo; Matsushita, Makoto; Kusumoto, Shinji. *Component Rank: Relative Significance Rank for Software Component Search*. International Conference on Software Engineering (ICSE'2003), pages 14–24, Portland, OR, 2003.

[ISO01] ISO 9126, *Information Technology – Product Quality – Part1: Quality Model*, International Standard ISO/IEC 9126, International Standard Organization, June, 2001.

[ISO05] *ISO - International Organization for Standardization*. Disponível em: http://www.iso.org. Acessado em 07/06/2005.

[ISO05b] ISO - International Organization for Standardization. *ISO 639 Language Codes*. Disponível em: http://www.oasis-open.org/cover/iso639a.html. Acessado em: 20/12/2005.

[Lopes04] Lopes, Adilson; Borelli, Frederico; Elias, Glêdson; Magalhães, Maurício. *A Component-Based Configuration Framework for Open, Distributed Multimedia Systems*. In 18th IEEE International Conference on Advanced Information Networking and Applications (AINA), Fukuoka, Japan, 2004.

[McIlroy69] McIlroy, M. D. *Mass-Produced Software Components*. In Software Engineering: Reports on a Conference Sponsored by the NATO Science Committee, pp. 151-155, NATO, Brussels, 1969.

[Moffat97] Moffat, A.; Bell, T. C.; Witten, I. H. *Lossless compression for text and images*. International Journal of High Speed Electronics and Systems 8 (1), pp. 179-231, 1997.

[Nutter03] Nutter, David; Boldyreff, Cornelia; Rank, Stephen. *An Artefact Repository to Support Distributed Software Engineering*. In: Workshop on Cooperative Supports For Distributed Software Engineering Processes (CSSE'2003), Benevento, Italy. 2003.

[Omondo06] Omondo. Disponível em: http://www.omondo.com. Acessado em: 05/06/206.

[Orso00] Orso, A., Harrold, M. J., and Rosenblum, D. S. *Component Metadata for Software Engineering Tasks*. In Proc. 2nd International Workshop on Engineering Distributed Objects (EDO 2000), pp. 126-140, Springer, Berlin, 2000.

[OMG01] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. December 2001.

[Pharoah01a] Pharoah, Andrew; Arni, Faiz. *Enterprise JavaBeans™ Technology Based Components*. Technical White Paper, Component Source, January 2001. Disponível em: http://www.componentsource.com/Services/EJBWhitePaper.asp.

[Pyarali00] Pyarali, Irfan; O'Ryan, Carlos; Schmidt, Douglas C. *A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware*. Proceedings, IEEE/IFIP International Symposium on Object-Oriented Real-Time Distributed Computing, Newport Beach, California, United States, 2000.

[RAS05] Reusable *Asset* Specification. 2005. Disponível em: http://www.omg.org/docs/ptc/05-04-02.pdf. Acessado em 02/11/2005.

[Redolfi05] Redolfi, Giliane; Spagnoli, Luciana; Hemesath, Peter; Bastos, Ricardo Melo; Ribeiro, Marcelo Blois; Cristal, Maurício; Espindola, Anete. *A Reference Model for Reusable Components Description*. In: Proceedings of the 38th Hawaii International Conference on System Sciences, Waikoloa, Big Island, Hawaii, United States, 2005.

[Ribot94] Ribot, Denielle; Bongard, Blandine; Villermain, Claude. *Development life-cycle WITH reuse*. In: Proceedings of the 1994 ACM Symposium on Applied Computing, Phoenix, Arizoa, United States, 1994.

[Sametinger97] Sametinger, Johannes. *Software engineering with reusable components*. Berlin: Springer, 1997.

[Schuenck05] Schuenck, Michael; Tanure, Carla; Dias Jr., José Jorge L.; Miranda, Sindolfo; Negócio, Yuri; Elias, Glêdson. *Análise de Repositórios de Componentes para a Identificação das Soluções mais Adotadas*. Workshop de Desenvolvimento Baseado em Componentes (WDBC'2005), pp. 9-26, Juiz de Fora, Brasil, Novembro de 2005.

[Schuenck06] Schuenck, Michael; Negócio, Yuri; Elias, Glêdson; Miranda Filho, Sindolfo; Dias Jr., Jorge; Cavalcanti, Gustavo. *X-ARM: A Step Towards Reuse of Commercial and Open Source Components*. The 9th International Conference on Software Reuse (ICSR), Lecture Notes in Computer Science, Springer-Verlag, Torino, Italy, 2006.

[Seacord98] Seacord, Robert C.; Hissam, Scott A.; Wallnau, Kurt C. *Agora: A Search Engine for Software Components*. CMU/SEI-98-TR-011, 1998.

[SourceForge06] SourceForge. http://www.sourceforge.net. Acessado em: 19/06/2006.

[Sun97] Sun Microsystems, Inc. *JavaBeans API Specification*. Version 1.01. July 1997.

[Sun98] Sun Microsystems, Inc. *Using Java Reflection*. Disponível em: *http://java.sun.com/developer/technicalArticles/ALT/Reflection/*. Acesso em: 19/06/2006.

[Sun99a] Sun Microsystems, Inc. *Code Conventions for the Java™ Programming Language*. Revised April 20, 1999. Disponível em: http://java.sun.com/docs/codeconv/html/ CodeConvTOC.doc.html. Acesso em: 16/06/2006.

[Szyperski02] Szyperski, Clemens. *Component Software: Beyond Object-Oriented Programming*. Second Edition, Addison-Wesley, 2002.

[Varadarajan01] Varadarajan, Sriram. *ComponentXchange: An E-Exchange for Software Components*. Dissertação de Mestrado, Indian Institute of Technology, Kanpur, India, Maio de 2001.

[W3C04a] World Wide Web Consortium (W3C). *Web Services Architecture*. Disponível em: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/. Acessado em: 10/12/2005.

[W3C04b] World Wide Web Consortium (W3C). *Document Object Model (DOM) Level 3 Core Specification*. Disponível em: http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/. Acessado em: 07/06/2006.

[Wallnau04] Wallnau, Kurt C. *Software Component Certification: 10 Useful Distinctions*. Technical Note, CMU/SEI-2004-TN-031, 2004.

[Wohlin94] Wohlin, Claes; Runeson, Per. *Certification of Software Components*. IEEE Transactions on Software Engineering, Vol. 20, No. 06, 1994, pp 494-499.