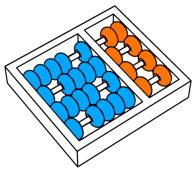
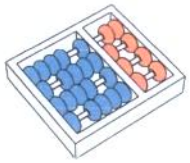


Leonel Aguilar Gayard

"CosmosLoader: uma ferramenta de apoio ao gerenciamento de configuração baseado no modelo Cosmos*"

CAMPINAS
2010



Universidade Estadual de Campinas
Instituto de Computação

Leonel Aguilar Gayard

"CosmosLoader: uma ferramenta de apoio ao gerenciamento de configuração baseado no modelo Cosmos*"

Orientador(a): **Profa. Dra. Cecília Mary Fischer Rubira**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por Leonel Aguilar Gayard, sob orientação de Profa. Dra. Cecília Mary Fischer Rubira.

Cecília Mary Fischer Rubira
Assinatura do Orientador(a)

CAMPINAS
2010

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

G254c

Gayard, Leonel Aguilar, 1983-
CosmosLoader : uma ferramenta de apoio ao gerenciamento de configuração baseado no modelo Cosmos* / Leonel Aguilar Gayard. – Campinas, SP : [s.n.], 2010.

Orientador: Cecília Mary Fischer Rubira.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Software - Arquitetura. 2. Componentes de software. I. Rubira, Cecília Mary Fischer,1964-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: CosmosLoader : a support framework for configuration management based on the Cosmos* model
Palavras-chave em inglês:
Software architecture
Component software
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Cecília Mary Fischer Rubira [Orientador]
Marco Aurélio Gerosa
Eliane Martins
Data de defesa: 20-12-2010
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 20 de dezembro de 2010, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Marco Aurélio Gerosa
IME / USP

Profª. Drª. Eliane Martins
IC / UNICAMP

Profª. Drª. Cecília Mary Fischer Rubira
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

CosmosLoader: uma ferramenta de apoio ao gerenciamento de configuração baseado no modelo Cosmos*

Leonel Aguilar Gayard[1]

20 de dezembro de 2010

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)
- Prof. Dr. Marco Aurélio Gerosa
  IME - USP
- Profa. Dra. Eliane Martins
  IC - UNICAMP
- Prof. Dr. Ivan Ricarte
  FEEC - UNICAMP (suplente)
- Profa. Dra. Ariadne Maria Rizzoni Carvalho
  IC - UNICAMP(suplente)

[1]Suporte financeiro da Capes Processo 01P-05603/2006

Resumo

Nos últimos anos, o desenvolvimento baseado em componentes (DBC) e a arquitetura de software emergiram como disciplinas complementares para promover o reuso no desenvolvimento de software. O desenvolvimento baseado em componentes promove o desenvolvimento de componentes reutilizáveis e a formação de novos sistemas de software a partir da integração de componentes existentes. A arquitetura de um sistema de software descreve o sistema em termos de seus componentes arquiteturais, das propriedades destes e das conexões entre eles. Modelos de componentes possibilitam concretizar elementos de DBC como componentes e conectores a partir de conceitos tradicionais de desenvolvimento de software, como por exemplo, orientação a objetos e arquivos, de modo que um conjunto de classes e arquivos complementares podem formar um componente se seguirem as regras do modelo. A adequação a um modelo de componentes oferece benefícios para o sistema de software: por exemplo, o modelo de componentes EJB possibilita que um componente seja implantado em um contêiner e se beneficie do gerenciamento de segurança e transparência de localização oferecidos pelo contêiner; o modelo Cosmos estabelece regras para a criação de componentes baseados em conceitos de linguagens orientadas a objetos, como classes, interfaces e pacotes.

Assim, a integração de componentes para a composição de um sistema de software de acordo com uma arquitetura e um modelo de componentes se torna uma atividade importante no desenvolvimento de um novo sistema de software. Uma configuração concreta é um conjunto de determinadas versões de componentes de software conectados de acordo com uma arquitetura. No entanto, observa-se que, se modelos facilitam a criação de componentes de software, ainda é necessário um esforço de desenvolvimento para integrar componentes e formar novos sistemas. O uso de ambientes de desenvolvimento orientados a arquitetura e DBC, e também de ferramentas de automatização, reduzem o esforço necessário para a integração de componentes. O ambiente Bellatrix é um ambiente de desenvolvimento integrado que permite a especificação de elementos arquiteturais e a modelagem de arquiteturas de software. No entanto, ainda existe um hiato entre a arquitetura de um sistema modelada no ambiente Bellatrix e a configuração do sistema a partir de componentes concretos. Esta dissertação apresenta a ferramenta CosmosLoader, que auxilia o gerenciamento de configurações de componentes baseados no modelo Cosmos. A solução proposta se baseia na extensão do ambiente Bellatrix e no modelo de componentes Cosmos* ("Cosmos estrela"), que estende o modelo Cosmos com o conceito de composição hierárquica de componentes. Por fim, são descritos estudos de caso realizados com essas ferramentas.

Abstract

In the last years, Component–Based Development (CBD) and Software Architecture emerged as complementary disciplines that promote reuse in software development. Component–Based Development promotes the development of reusable software components and the creation of new software systems by integrating existing software components. The architecture of a software system describes such system in terms of its architectural components, their properties and the connections between them. Component models materialize concepts from CBD such as components and connectors from traditional concepts of software development, such as object orientation and files, so that a set of classes and complementary files form a component if they follow the component model's rules. The adequacy to a component model brings benefits to a software system: for instance, the EJB component model allows a component to be deployed to a container and benefit from the security management and location transparency provided by the container; the Cosmos component model allows the creation of components using only concepts from object-oriented languages, such as classes, interfaces and packages.

Therefore, the integration of components to compose a new software system according to an architecture and a component model becomes an important activity in the development of a new software system. A concrete configuration is the set of specific versions of software components connected according to an architecture. However, it can be observed that while models ease the creation of software components, a development effort is still necessary to integrate components and form new systems. The use of development environments oriented towards architecture and CBD, and also of automations tools, reduce the effort needed to integrate components. The Bellatrix development environment is an integrated development environment that allows the specification of architectural elements and modeling of software architectures. However, there still is a gap between the architecture of a system modeled in Bellatrix and the configuration of a system from concrete software components. This dissertation presents the CosmosLoader tool, which assists in managing the configuration of components based on the Cosmos model. The proposed solution is based on an extension to the Bellatrix development environment and on the Cosmos* component model ("Cosmos star"), which extends the Cosmos component model with hierarchical composition of components. Finally, case studies using these tools are described.

Agradecimentos

Esta dissertação de mestrado é o resultado de uma longa jornada durante os últimos anos. Uma página é pouco espaço para agradecer a todos aqueles que de uma forma ou outra me apoiaram para que eu conseguisse atravessá-la.

Gostaria de agradecer a Deus por todas as oportunidades que apareceram durante a realização deste trabalho. Em muitos dos desafios que superei, imagino que havia um dedo d'Ele para me ajudar; em outros, acho que era uma mão inteira !

Aos amigos do Instituto de Computação: Arthur Castro, Bruno Albertini, Cláudio Carvalho, Diogo Kropiwiec, Edna Hoshino, Eric Ostroski, Ivan Perez, Juliana de Santi, Luciano Digiampetri, Marcelo Couto, Rodrigo Minetto, Vagner Pedrotti, agradeço por compartilhar ideias e conversas durante um almoço, um café e momentos de lazer entre amigos; aos também colegas do grupo de pesquisas, Leonardo Tizzei, Patrick Brito, Rodrigo Tomita, Tiago Moronte, agradeço também pela introdução aos temas de pesquisa e por ideias durante o andamento do trabalho. Todos estes sem exceção estão desfrutando de um grande sucesso em suas carreiras, e são modelos que quero seguir.

Ao pesquisador Paulo Astério de Castro Guerra, que durante um bom tempo acompanhou meu trabalho, agradeço pelas oportunidades e sugestões.

A minha orientadora, Profa. Dra. Cecilia Rubira, agradeço pela orientação e pelos "puxões de orelha"; quando eu quis voltar, depois de ter desistido de tudo, obrigado por ter me recebido de volta. Agradeço principalmente pelo apoio e paciência durante a reta final, quando fui um aluno mais distante e dei mais trabalho.

Aos professores Marco Aurélio Gerosa e Eliane Martins, agradeço por aceitarem o convite para compôr a banca durante a defesa deste trabalho, e pelos comentários e correções que enriqueceram o texto.

Agradeço imensamente a meus pais, Yves e Maria Alzira. Nossa família sempre foi o refúgio mais seguro nos momentos mais complicados; para minha irmã Nicole, que mergulhou na vida acadêmica, agradeço pelo companheirismo e faço votos de sucesso.

Finalmente, agradeço a minha esposa, Liliana Mie, por nunca ter deixado de acreditar que eu conseguiria. Agradeço principalmente pelo encorajamento e por aturar um marido distraído na reta final. Te amo muito, Mie. Nada disso seria possível sem você.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Nos últimos anos, o Desenvolvimento Baseado em Componentes (DBC) e a Arquitetura de Software emergiram como disciplinas complementares para promover o reuso no desenvolvimento de software. O Desenvolvimento Baseado em Componentes é uma disciplina que promove o desenvolvimento de componentes reutilizáveis e a formação de novos sistemas de software a partir da integração de componentes existentes. A arquitetura de um sistema de software descreve o sistema em termos de seus **componentes arquiteturais**, das propriedades destes e das conexões entre eles. Enquanto um componente arquitetural é um conceito abstrato, um componente de software é uma implementação concreta de um ou mais componentes arquiteturais e pode ser executado em um dispositivo físico ou lógico. Um componente de software integra dados e funções com alto encapsulamento, alta coesão e baixo acoplamento. Dessa forma, as **interfaces** de um componente arquitetural são especificações de serviços providos e são implementadas por um componente de software.

Componentes de Software reutilizáveis são geralmente desenvolvidos sem conhecimento dos diferentes contextos em que serão empregados. A arquitetura do software é então responsável por integrar esses componentes com o objetivo de obter os atributos de qualidade desejados para o sistema em desenvolvimento. Assim, uma **configuração arquitetural concreta** é uma organização de componentes concretos em que cada componente concreto implementa um componente arquitetural. A correspondência entre componentes de software e componentes arquiteturais é evidenciada nos principais processos de DBC, como por exemplo, os processos UML Components[8] e Catalysis[20] que também são centrados na arquitetura de software.

Um componente de software está estruturado de acordo com um modelo de componentes. Um modelo de componentes tem dois objetivos: 1) ele provê regras para a especificação de propriedades dos componentes; e 2) provê regras e mecanismos para composição, incluindo regras para composição de propriedades dos componentes [15].

O modelo de componentes Cosmos* (COmponent System MOdel for Software architectures) [27] visa representar explicitamente abstrações da arquitetura de software, como componentes arquiteturais e conectores, em construções de linguagens de programação orientadas a objetos. No modelo Cosmos*, cada elemento arquitetural é organizado como um conjunto de pacotes estruturados de acordo com regras propostas pelo modelo, e é implementado internamente com uso de padrões de projeto amplamente conhecidos, como *Façade*, *Adapter*, *Factory Method* [25] e *Dependency Injection* [23].

O desenvolvimento de sistemas baseados em componentes pode ter apoio de ferramentas específicas, que oferecem suporte a diferentes atividades do desenvolvimento. Tais ferramentas podem oferecer suporte à especificação e desenvolvimento de componentes ou à modelagem da arquitetura.

O ambiente Bellatrix é um ambiente de desenvolvimento integrado que apoia o DBC com ênfase na arquitetura de software [47]. O ambiente Bellatrix estende o ambiente Eclipse com editores gráficos para especificação de elementos arquiteturais, como interfaces, componentes e configurações arquiteturais.

Uma atividade importante do desenvolvimento baseado em componentes é a integração de componentes para compôr uma configuração arquitetural concreta. Uma integração foi realizada corretamente se a configuração dos componentes de software corresponde à arquitetura do sistema, ou seja, se os componentes concretos estão conectados como estabelecido pelos componentes arquiteturais. Modelos de componentes e ferramentas podem oferecer suporte à atividade de integração.

1.1 Motivação e Problema

A integração de componentes para a formação de um novo sistema de software obedece às regras de composição definidas no modelo de componentes empregado. Por exemplo, nos modelo EJB[1] [42] e OSGi[2] [2], as conexões entre componentes são gerenciadas por contêineres; no modelo EJB, utiliza-se o mecanismo de nomes JNDI[3]. No modelo OSGi, as conexões são feitas através de uma API de serviços na qual cada componente se registra.

No modelo Cosmos*, a integração de componentes é feita através de código fonte externo aos componentes chamado de código de instanciação. O modelo Cosmos* estabelece um conjunto de regras para criar este código de instanciação, que dispõe os componentes e as conexões entre eles em uma configuração arquitetural.

No entanto, existe um hiato entre as atividades de modelagem de sistemas baseados em componentes e a integração de componentes de software para a implementação de um novo

[1] Enterprise JavaBeans
[2] Open Service Gateway Initiative
[3] Java Naming and Directory Interface

sistema. Enquanto as ferramentas de suporte são usadas para especificar a arquitetura do sistema em termos de componentes arquiteturais e conexões entre interfaces, a integração de componentes em uma configuração arquitetural concreta é realizada através de código fonte.

Existem problemas com esta abordagem: 1) o código de integração é escrito de forma manual por um programador, que pode introduzir discrepâncias entre a configuração arquitetural e a arquitetura (por exemplo, componentes conectados de forma errônea, ou componentes "esquecidos", ausentes na configuração); e 2) um elemento importante da configuração arquitetural é a versão dos componentes: componentes de diferentes versões apresentam comportamentos diferentes; se o modelo de componentes adotado não apresentar uma abordagem para determinar as versões dos componentes em uma configuração arquitetural, é possível que versões de componentes diferentes das consideradas na arquitetura sejam inseridas na configuração arquitetural.

Estes problemas estão presentes no modelo Cosmos*, porque o código de integração de componentes é extenso; ainda, na plataforma Java, uma das plataformas nas quais o modelo Cosmos* é utilizado, não há um mecanismo direto para a escolha de versões de componentes.

1.2 Solução proposta

Uma solução para o problema da integração de componentes é estender o conjunto de ferramentas envolvidas no desenvolvimento para automatizar essa atividade. Uma vez que as ferramentas de suporte descrevem a arquitetura e a configuração arquitetural concreta, é possível utilizar esta informação para integrar os componentes de forma automatizada. Assim, uma ferramenta pode ler esta descrição e instanciar e conectar automaticamente as versões corretas dos componentes descritos na configuração concreta.

Neste trabalho, apresentamos como solução a ferramenta CosmosLoader, que automatiza a integração de sistemas baseados no modelo Cosmos* a partir da descrição de uma configuração arquitetural gerada pelo ambiente Bellatrix. A ferramenta CosmosLoader instancia e conecta componentes Cosmos*. A ferramenta utiliza o mecanismo de carregadores de classes (*class loaders*) da plataforma Java para instanciar as versões corretas dos componentes.

1.3 Contribuições

As contribuições deste trabalho incluem 1) uma descrição do modelo Cosmos*, que estende o modelo de componentes Cosmos; 2) a descrição e implementação da ferramenta

CosmosLoader; e 3) estudos de caso realizados com a ferramenta CosmosLoader.

1.4 Organização deste Documento

Este documento foi dividido em 6 capítulos e dois apêndices, organizados da seguinte forma:

- **Capítulo 2 - Fundamentos teóricos:** Este capítulo prepara o embasamento teórico para os trabalhos apresentados nesta Dissertação, abordando os conceitos de Arquitetura de Software e Desenvolvimento Baseado em Componentes. Na sequência, são apresentados modelos de componentes existentes. Uma classificação de modelos de componentes é apresentada, e os modelos são listados de acordo com esta classificação. Por fim, o capítulo aborda as ferramentas de apoio à configuração arquitetural dos modelos de componentes apresentados.

- **Capítulo 3 - O modelo Cosmos*:** Este capítulo apresenta o modelo Cosmos*, que estende o modelo Cosmos. São descritos os modelos internos de especificação, implementação e de conectores, que compõem o modelo Cosmos original, e as extensões ao modelo, que constituem o Cosmos*: o modelo de composição, o modelo de componentes de sistema, os tratamentos de bibliotecas e dependências de ambiente de execução e tipos de dados.

- **Capítulo 4 - A ferramenta CosmosLoader:** Este capítulo descreve a ferramenta CosmosLoader. O capítulo inicialmente retoma a motivação para a utilização da ferramenta CosmosLoader. Em seguida, descreve como a ferramenta substitui o código de instanciação de um sistema baseado no modelo Cosmos*. O capítulo também descreve o mecanismos de carregamento de classes na plataforma Java, que é um dos fundamentos da ferramenta, e mecanismos internos da ferramenta. O capítulo termina descrevendo as alterações no ambiente Bellatrix para gerar os arquivos de configuração utilizados pela ferramenta.

- **Capítulo 5 - Estudos de caso:** Este capítulo descreve os estudos de caso realizados com a ferramenta CosmosLoader. Inicialmente, são apresentados os sistemas alvos dos estudos de caso. Em seguida, são descritos os passos para a realização de cada estudo de caso, e os questionários aplicados no fim do estudo de caso. Por fim, os resultados do estudo de caso são interpretados.

- **Capítulo 6 - Conclusões:** Este capítulo conclui esta Dissertação. As contribuições principais desta Dissertação são resumidas e as publicações originadas deste trabalho

de Mestrado são listadas. O capítulo termina com sugestões de trabalhos futuros a partir dos temas apresentados.

- **Apêndice A - Artefatos do estudo de caso com o Sistema Portal2:** Este capítulo contém o questionário aplicado aos voluntários que executaram o estudo de caso com o Sistema Portal2 e que coleta dados sobre a utilização da ferramenta.
- **Apêndice B - Artefatos do estudo de caso com o Sistema eGov-CNH:** Este capítulo contém o questionário aplicado aos voluntários que executaram o estudo de caso sobe o sistema eGov-CNH.

Capítulo 2

Fundamentos teóricos

Neste capítulo são apresentados os conceitos que servem como fundamentos para este trabalho. A Seção 2.1 apresenta a disciplina de Arquitetura de Software. A Seção 2.2 descreve os conceitos fundamentais para o Desenvolvimento Baseado em Componentes. A Seção 2.3 descreve alguns modelos de componentes mais conhecidos e com importâncias na indústria de software e em ambientes de pesquisa. Na Seção 2.4, um conjunto de critérios para classificação de modelos de componentes é apresentado, e os modelos de componentes descritos anteriormente são classificados de acordo com estes critérios. A Seção 2.5 finaliza este capítulo descrevendo como trabalhos relacionados as ferramentas de apoio aos modelos de componentes descritos.

2.1 Arquitetura de Software

A arquitetura de um sistema de software é a identificação das estruturas deste sistema, e consiste nos elementos de software do sistema, suas propriedades externamente visíveis e as relações entre eles [3]. A arquitetura de um sistema de software descreve o sistema em termos de seus componentes arquiteturais, que são elementos de sua arquitetura responsáveis por satisfazer os requisitos do sistema, e de seus conectores arquiteturais, que são componentes mais simples, responsáveis pela comunicação entre os componentes.

Componentes arquiteturais definem interfaces providas e requeridas, que são especificações de serviços que eles provêem para outros componentes e de serviços que requerem de outros componentes para realizar suas funções [5]. Um arranjo de componentes arquiteturais conectados por conectores arquiteturais é chamado de configuração arquitetural [3].

2.2 Desenvolvimento baseado em Componentes

O Desenvolvimento Baseado em Componentes (DBC) é uma prática no desenvolvimento de software que prevê a construção de sistemas a partir de componentes de software existentes.

Szyperski [44] apresenta a seguinte definição para um componente de software:

> Um componente de software é uma unidade de composição com interfaces especificadas em contrato e dependências de contexto explícitas. Um componente de software pode ser implantado de maneira independente e está sujeito a composição por terceiros.

Assim, a prática do desenvolvimento baseado em componentes prevê a organização de um sistema a partir de componentes de software, que podem ser desenvolvidos ou adquiridos de terceiros, possibilitando uma maior facilidade de manutenção do sistema e maior qualidade do sistema de software resultante.

Em um componente de software, existe uma separação entre suas interfaces e sua implementação. O estado e as funcionalidades de um componente são acessíveis somente por meio das interfaces. Assim, para que seja possível descrever completamente um componente e garantir sua integração, um componente deve conter os seguintes elementos [14]:

- Um conjunto de interfaces providas ou requeridas. Estas interfaces são o meio de comunicação do componente com outros componentes ou com o ambiente no qual será implantado.
- Código executável, que realiza as funcionalidades das interfaces e pode ser acoplado com código de outros componentes por meio das interfaces.

Interfaces de um componente são os pontos de acesso de um componente. Uma interface é somente uma descrição de serviços, e não carrega uma implementação. A separação entre interface e implementação possibilita 1) substituir a implementação de um componente sem alterar a interface; e 2) adicionar novas interfaces (e suas implementações) sem alterar a implementação existente, e assim aprimorar a capacidade de adaptação do componente.

A Figura 2.1 ilustra a notação padrão para componentes na linguagem UML 2.0. O componente `A` tem uma interface provida `I1` e uma interface requerida `I2`. O componente `B` implementa a interface `I2`. Esta notação não especifica se a interface `I2` é definida pelo componente `A` ou pelo componente `B`, mas enfatiza a conexão entre as interfaces no sistema. Assim, as interfaces podem tanto ser definidas pelo componente como o componente pode implementar interfaces definidas por terceiros.

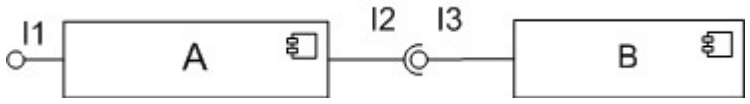


Figura 2.1: Um componente de software, na linguagem UML 2.0

2.3 Modelos de componentes

Modelos de componentes são a base do Desenvolvimentos Baseado em Componentes [44]. Um modelo de componentes é uma definição de o que é um componente, como ele é construído, como é a composição entre componentes e como um sistema de componentes é implantado.

Diversos modelos de componentes existem, com definições diferentes para o que é um componente. Esta seção descreve os modelos JavaBeans (Seção 2.3.1), EJB (Seção 2.3.2), CORBA e CCM (Seção 2.3.3), Fractal (Seção 2.3.4), OpenCom (Seção 2.3.5), OSGi (Seção 2.3.6), e COSMOS (Seção 2.3.7).

2.3.1 JavaBeans

O modelo de componentes JavaBeans [38] foi inicialmente proposto pela Sun Microsystems em 1996 visando um modelo simples para a plataforma Java. A definição de um JavaBean é a seguinte [38]:

> Um Java Bean é um componente de software reutilizável que pode ser manipulado visualmente em uma ferramenta de edição gráfica.

Um componente JavaBean (ou, simplesmente, um Java Bean), é uma classe Java que apresenta as seguintes características:

- possui um construtor público e sem argumentos.
- expõe propriedades; propriedades são métodos cujos nomes começam com os prefixos *get* para leitura ou *set* para alteração.
- podem disparar eventos para notificar outros componentes ou esperar eventos disparados por outros componentes.

Estas características fazem com que um JavaBean ofereça as seguintes funcionalidades:

- Suporte a introspecção, possibilitando a uma ferramenta gráfica analisar o funcionamento do JavaBean.

- Suporte a customização, possibilitando que uma ferramenta altere a aparência e o comportamento do JavaBean.
- Suporte a eventos, como forma de comunicação entre componentes.
- Suporte a propriedades, para customização e para uso programático.
- Suporte a persistência, para que um componente alterado em uma ferramenta possa ter seu estado salvo e restaurado posteriormente.

Inicialmente, este modelo foi imaginado para a tecnologia Java Applets, que possibilitava a inclusão de aplicações Java interativas (*applets*) em páginas web. Com o passar do tempo, a tecnologia de applets perdeu popularidade; no entanto, o modelo de componentes permanece e é utilizado em diversas tecnologias, devido a sua simplicidade.

As características dos componentes JavaBeans possibilitam que a composição de componentes seja feita em ferramentas visuais, como, por exemplo, a ferramenta `Bean Builder` [18]. Esta ferramenta oferece editores para descrever as ligações entre componentes e salvá-los em arquivo. A descrição depois é lida em tempo de execução para instanciar o componente.

No entanto, a visão inicial de que aplicações seriam construídas por meio da composição de Java Beans em ferramentas visuais não aconteceu na extensão prevista por seus autores. Aplicações do modelo inicialmente previstas por seus autores, como componentes JavaBeans para acesso a banco de dados, não foram popularizadas.

Uma aplicação do modelo que se popularizou foi a construção de interfaces gráficas (*Graphical User Interface*, GUI) com uso de ferramentas visuais. Alguns ambientes de programação, como o Netbeans [12], possibilitam a criação de GUIs por meio de um ambiente visual. No entanto, estas ferramentas fazem uso de um subconjunto restrito das características do modelo; por exemplo, a funcionalidade de persistência ditada pelo modelo não tem vantagens para componentes gráficos, e é então deixada de lado.

2.3.2 EJB

Enterprise JavaBeans [42], ou EJB, é um modelo de componentes para aplicações cliente-servidor em Java. Componentes EJB são objetos remotos que têm seu ciclo de vida gerenciado por um contêiner EJB.

A Figura 2.2 ilustra um componente EJB gerenciado por um contêiner.

Para utilizar um componente EJB, uma classe Java utiliza o mecanismo de nomes JNDI[1], que oferece uma interface para objetos remotos. O contexto JNDI possibilita a comunicação entre a classe cliente e a interface remota do EJB. Ainda, um componente

[1] Java Naming and Directory Interface

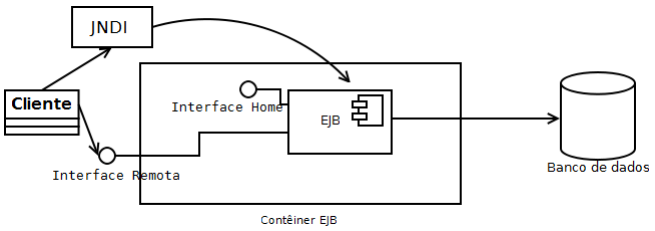


Figura 2.2: Um componente EJB

EJB oferece duas interfaces: uma interface remota, que é invocada por clientes em uma máquina virtual diferente, e uma interface "Home", que pode ser invocada por clientes na mesma máquina virtual. Esta distinção existe para evitar a sobrecarga causada pela comunicação remota desnecessária com um objeto cliente na mesma máquina virtual.

Nas versões mais atuais do modelo, 3.0 e 3.1, um componente EJB pode ser de dois tipos: *session beans* são componentes que implementam regras de negócio da aplicação sem preocupação com propriedades extrafuncionais da aplicação, como performance e segurança, que são gerenciados pelo contêiner; *message-driven beans* são componentes para tratamento assíncrono de mensagens.

Os arquivos que compõem um EJB são ilustrados na Figura 2.3 [11]: um EJB é composto de classes Java e arquivos descritores. Descritores são arquivos no formato texto com metainformação sobre o componente que instrui o contêiner a instanciar o componente.

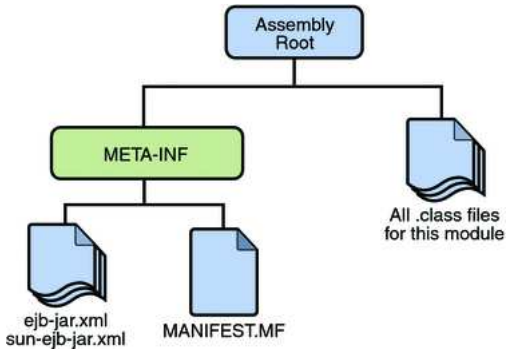


Figura 2.3: Estrutura de arquivos de um componente EJB

A implantação de componentes EJB consiste em criar um arquivo compactado com os arquivos ilustrados na Figura 2.3: os arquivos `MANIFEST.MF` e `ejb-jar.xml` descrevem os componentes EJB da aplicação; o arquivo `sun-ejb-jar.xml` declara extensões proprietárias que funcionam somente no contêiner EJB da Sun Microsystems. Outros arquivos podem ser incluidos para ativar extensões proprietárias quando o arquivo é implantado em demais conteineres.

As versões 2.0 e 2.1 do modelo EJB sofreram críticas pela quantidade de arquivos necessários para a descrição de um componente EJB. Para um único componente, eram necessárias classes para a interfaces Home e remota, as classes de implementação e os arquivos de descrição do componente EJB. Isto levou ao surgimento de ferramentas como o Xdoclet [45]: esta ferramenta processa anotações deixadas no código fonte do projeto e gera automaticamente as demais classes necessárias ao componente EJB.

2.3.3 CORBA e CCM

CORBA[30] (*Common Object Request Broker Architecture*) é um padrão definido pelo grupo OMG (*Object Management Group*) para comunicação entre componentes de software escritos em linguagens de programação diferentes e executados em um ambiente distribuido; CCM (*Corba Component Model*) é uma extensão do padrão CORBA que descreve uma infraestrutura para aplicações baseadas em componentes CORBA.

CORBA provê uma IDL (*Interface Description Language*, linguagem de descrição de interfaces); interfaces de componentes são descritas nesta linguagem e podem ser implementadas por componentes em diferentes linguagens de programação de acordo com um mapeamento.

Um mapeamento é uma especificação que dita como os conceitos de componentes do padrão são implementados na linguagem de programação. Existem mapeamentos CORBA para diversas linguagens de programação: Ada, C, C++, Lisp, Java, COBOL, Python, e outras. Por exemplo, CORBA contém os conceitos de objeto e tratamento de exceções, o que facilita implementar uma IDL em Java; no entanto, o mapeamento para a linguagem C não é tão simples, porque a linguagem C não tem estes conceitos.

O padrão CORBA define que a comunicação entre componentes é intermediada por um gerenciador de requisições (*Object request broker*, ORB). Um ORB é um componente de software que conhece a localização dos componentes e encaminha as requisições para eles.

O modelo CCM[29] (*CORBA Component Model*) é uma extensão do padrão CORBA; no modelo CCM, a comunicação entre componentes de software se faz por meio de *portas*. O modelo define os seguintes tipos de portas:

- Facetas são interfaces utilizadas por clientes para manipular o componente;
- Receptáculos são pontos de conexão pelos quais um componente utiliza outro componente;
- Fontes de eventos são pontos de conexão que emitem eventos de determinado tipo para consumidores ou para um canal de eventos;

- Alvos de eventos são pontos de conexão para os quais eventos de determinado tipo podem ser enviados;
- Atributos são valores expostos por operações de leitura e escrita.

Componentes CCM podem ser dos tipos básico ou extendido: componentes básicos apresentam somente atributos; componentes extendidos podem oferecer qualquer tipo de porta.

2.3.4 Fractal

Fractal [6, 7] é um modelo proposto para a implementação, implantação e gerenciamento de sistemas de software complexos, em particular, sistemas operacionais e *middleware*. O modelo Fractal apresenta as seguintes funcionalidades, voltadas para lidar com a complexidade dos sistemas alvo:

- Composição de componentes;
- Compartilhamento de componentes, que possibilita o compartilhamento de recursos mantendo o encapsulamento dos componentes;
- Funcionalidades de introspecção, para monitorar e controlar a execução de um sistema em funcionamento;
- Funcionalidades de reconfiguração dinâmica, para implantar e reconfigurar dinamicamente um sistema.

O modelo Fractal é extensível: funcionalidades de controle dos componentes não são determinadas no modelo; o modelo se baseia nas funcionalidades de reflexão para possibilitar diferentes níveis de controle dos componentes. Desta forma, enquanto alguns componentes podem não oferecer nenhuma funcionalidade para controle externo (são caixas pretas), outros podem oferecer uma gama funcionalidades de controle que vão desde introspecção até gerenciamento de ciclo de vida.

Um componente Fractal é uma entidade que existe em tempo de execução e é encapsulada, tem uma identidade e apresenta uma ou mais interfaces. Uma interface de um componente Fractal é um ponto de acesso do componente e é tipada. Interfaces podem ser de dois tipos: interfaces servidoras (*server interfaces*) recebem invocações de operações de "entrada"; interfaces clientes recebem invocações de operação de "saída".

O modelo Fractal não define um componente Fractal como um objeto, para possibilitar a implementação do modelo em linguagens sem conceitos de orientação a objetos, como C.

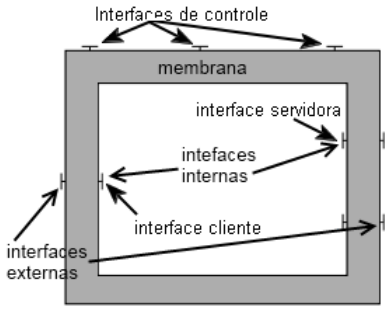


Figura 2.4: Um componente Fractal

A Figura 2.4 ilustra um componente Fractal. Um componente Fractal é composto de uma membrana, que contém interfaces para operações de introspecção e reconfiguração de funcionalidades internas, e um conteúdo, que é um conjunto de subcomponentes. A membrana de um componente pode ter interfaces e externas e internas: interfaces externas são acessíveis de fora do componente; interfaces internas são somente acessíveis por subcomponentes. Ainda, o modelo define interfaces clientes e servidoras: interfaces clientes são conectadas a interfaces providas de componentes internos à membrana; interfaces servidoras são conectadas a interfaces requeridas de componentes internos à membrana. Desse modo, as interfaces providas e requeridas são acessadas através da membrana. No framework Julia[6, 7], que implementa o modelo Fractal em Java, a membrana intercepta chamadas às interfaces dos componentes.

As ligações entre interfaces de componentes no modelo Fractal podem ser de dois tipos: ligações primitivas ocorrem entre interfaces em um mesmo espaço de memória e podem ser implementadas como ponteiros ou referências diretas entre objetos; ligações compostas ocorrem entre um número arbitrário de interfaces e são tipicamente implementadas com *stubs*, *skeletons* e adaptadores.

O modelo Fractal também prevê o compartilhamento de componentes. Uma mesma instância de componente pode ser compartilhada. A Figura 2.5 ilustra duas arquiteturas de um sistema de componentes visuais para um editor de texto. A primeira arquitetura não tem componentes compartilhados; a segunda arquitetura compartilha o componente de Menu. Esta funcionalidade possibilita compartilhar o estado de um componente sem afetar o encapsulamento dos componentes que o contêm. A arquitetura sem compartilhamento de componentes obriga os componentes `Menu` e `Toolbar` a apresentar uma interface requerida externa para o componente `Undo`.

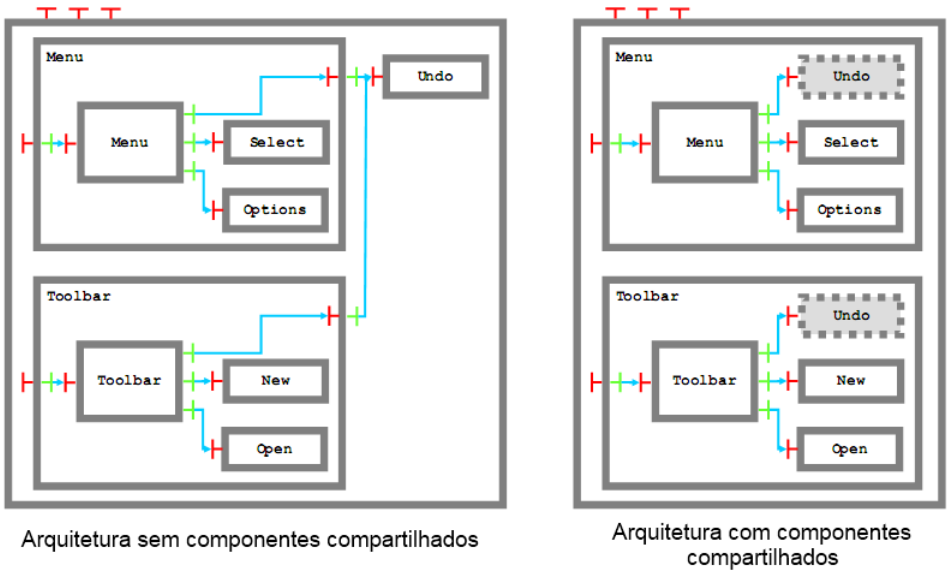


Figura 2.5: Compartilhamento de componentes no modelo Fractal

2.3.5 OpenCom

O modelo de componentes OpenCom [13] é um modelo de componentes de propósito geral voltado para a programação de software de sistemas, como sistemas operacionais, sistemas de comunicações, sistemas embarcados e plataformas de middleware. Assim, o modelo OpenCom procura satisfazer os seguintes requisitos:

- **Independência de domínio:** o modelo OpenCom procura ser genérico para garantir a aplicação em diferentes domínios. Assim, apesar de ser voltado para software de sistemas como sistemas operacionais, cujos requisitos podem incluir execução em tempo real ou alta disponibilidade, estes não são requisitos do modelo; tais características incorrem em alto custo em performance no sistema, mesmo quando não são requeridos. Ao invés disso, estas características podem ser implementadas por um mecanismo de extensão presente no modelo.

- **Independência de ambiente de implantação:** para que sistemas sejam implantados em uma vasta gama de dispositivos que vão desde sistemas embarcados até supercomputadores, com diferentes capacidades de processamento e recursos, o modelo OpenCom tem como requisitos simplicidade de programação, baixo uso de memória e independência de linguagem de programação. Ainda, o modelo oferece suporte a customização e extensibilidade.

- **Baixa sobrecarga**: Além de incorrer em baixo uso de memória, o modelo tem como requisito uma baixa demanda por outros recursos, principalmente processa-

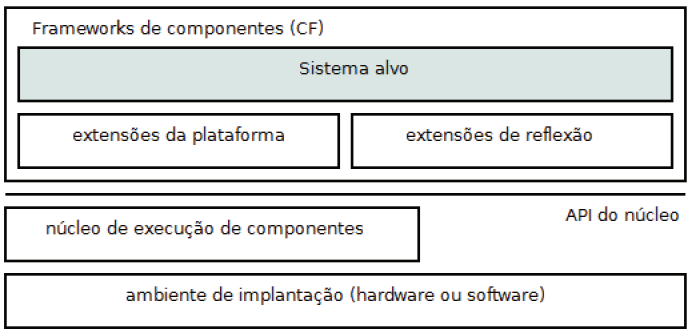


Figura 2.6: Visão geral da arquitetura do modelo OpenCom

mento. Assim, o modelo OpenCom não impõe mecanismos de execução sobre os componentes; por exemplo, a comunicação entre componentes não se dá por um serviço de mensagens mediado por um núcleo.

A Figura 2.6 ilustra a arquitetura do modelo OpenCom. O modelo requer a existência em tempo de execução de um núcleo de componentes, responsável por carregar (*Load*) e ligar (*bind*) componentes. Este núcleo existe imediatamente sobre a plataforma de implantação, que pode ser software ou hardware. Em sistemas estáticos, o núcleo existe somente para carregar e ligar os componentes e é desligado após estas tarefas; em sistemas dinâmicos, o núcleo continua a existir durante a execução.

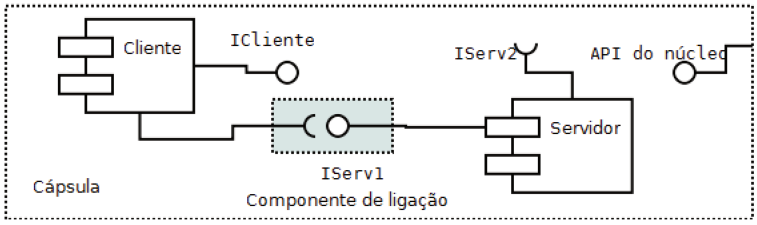


Figura 2.7: Elementos do modelo OpenCom

A Figura 2.7 ilustra os elementos do modelo OpenCom. Componentes são instanciados e conectados dentro de *cápsulas*. Cada cápsula define um espaço de nomes (*namespace*) para as instâncias de componente nela contidas, e lhes provê acesso à API do núcleo de execução. Cápsulas não reconhecem aninhamento ou composição hierárquica de componentes.

Estudos de caso foram realizados com implementações do modelo OpenCom nas linguagens C, C++ e Java [13].

2.3.6 OSGi

O consórcio OSGi Alliance [1] é um consórcio de parceiros na indústria de tecnologia voltado para a criação de especificações abertas de sistemas modulares baseados na plataforma Java. O consório OSGi Alliance define a Plataforma OSGi (*OSGi Service Platform*), uma plataforma comum para desenvolvedores e consumidores de software.

O consórcio OSGi Alliance especifica o padrão OSGi [2], que é um modelo de componentes voltado para plataformas com capacidades que variam desde plataformas embarcadas até sistemas em servidores com grande poder de processamento. Entre as implementações do padrão OSGi, estão o contêiner Apache Felix [21], do grupo Apache; o contêiner Equinox [22], do consórcio Eclipse; e o contêiner Knopflerfish [35], da organização de mesmo nome.

No padrão OSGi, componentes de software são implementados na linguagem Java, e empacotadas em arquivos chamados *bundles*. Um *bundle* é um arquivo .JAR que contém um manifesto (um arquivo texto com nome `MANIFEST.MF`) que contém metainformação sobre o componente no formato chave-valor.

Componentes são implantados em um contêiner OSGi, que gerencia seus ciclos de vida. Uma vez implantado, um bundle tem acesso a serviços providos pelo contêiner sob a forma de APIs. A Figura 2.8 ilustra a camada de serviços providos por um contêiner OSGi. Os serviços estão organizados em camadas dentro de um contêiner, sendo que cada camada tem acesso a serviços da camada inferior adjacente e provê serviços para a camada imediatamente acima.

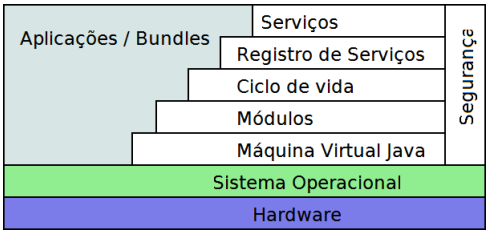


Figura 2.8: Camada de serviços do modelo OSGi

O modelo OSGi prevê o versionamento de componentes. Cada componente declara em seu manifesto a sua versão e as dependências por outros componentes. O contêiner instancia e liga as versões de cada componente de forma que as versões estão de acordo com as declarações nos manifestos de ambos.

2.3.7 Cosmos

O modelo Cosmos (*Component Structuring Model for Object-oriented Systems*) [16, 17] se baseia na implementação de conceitos de Arquitetura de Software e Desenvolvimento Ba-

seado em Componentes utilizando conceitos existentes nas linguagens orientadas a objeto, como pacotes e interfaces, pela aplicação de padrões de projeto amplamente conhecidos, como *Façade*, *Adapter*, *Factory Method* [25] e Dependency Injection [23]. Assim, alguns conceitos de Arquitetura de Software, como componentes, interfaces e conectores, são construídos explicitamente em código fonte seguindo as regras do modelo.

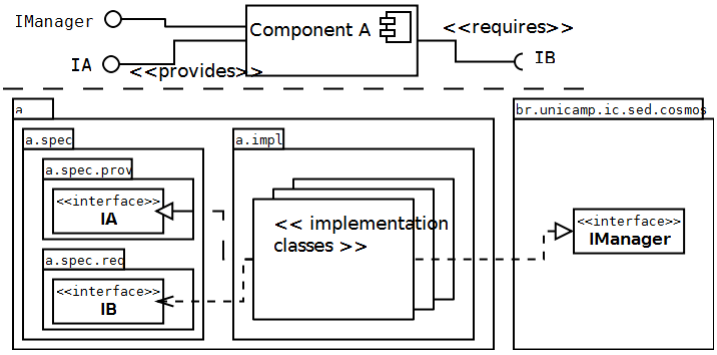


Figura 2.9: Estrutura de pacotes de um componente Cosmos*

A Figura 2.9 ilustra de maneira simplificada a estrutura interna de um componente Cosmos. As interfaces do componente são separadas em um pacote `spec`, enquanto as classes de implementação do componente estão no pacote `impl`. A comunicação entre componentes ocorre somente através das interfaces, sendo que interfaces providas e requeridas de dois componentes estão conectadas por meio de um conector, que é um componente mais simples. Isto favorece um alto encapsulamento de componentes, removendo referências diretas entre classes de dois componentes.

O Capítulo 3 descreve detalhamente o modelo Cosmos, e propõe uma extensão, o modelo Cosmos* ("Cosmos estrela").

2.4 Uma classificação de modelos de componentes

Crnkovic et al. [15] estabelecem uma classificação de modelos de componentes de acordo com determinadas características de cada modelo. Esta classificação contém quatro dimensões:

- **(1) Ciclo de desenvolvimento:** dimensão que identifica o suporte provido de maneira explícita ou implícita pelo modelo de componentes para as diferentes fases de desenvolvimento dos componentes ou sistemas baseados em componentes.

- **(2) Mecanismos de construção:** identifica no modelo a interface entre componentes e o ambiente externo e os meios para ligação (*binding*) e comunicação entre componentes.

- **(3) Propriedades extrafuncionais:** identifica o suporte do modelo de componentes para a especificação e composição de propriedades não funcionais nos componentes.

- **(4) Domínio:** identifica o domínio de aplicações para os quais o modelo de componentes foi projetado. Esta dimensão determina se um modelo de componentes é mais especializado ou mais genérico.

2.4.1 Ciclo de desenvolvimento

Esta dimensão avalia o suporte provido pelo modelo às fases do desenvolvimento dos componentes de um sistema.

Alguns modelos de componentes consideram os componentes somente durante a fase de projeto de um sistema, de modo que durante a execução o sistema é um bloco monolítico, sem divisão. Outros modelos gerenciam o ciclo de vida dos componentes durante a execução do sistema, e os componentes podem verificar o estado dos demais componentes. Ainda outros modelos se concentram na fase de implementação do sistema (por exemplo, EJB [40]) e o benefício provido pelo modelo está no suporte a requisitos não funcionais (por exemplo, distribuição e controle de acesso).

Assim, esta dimensão verifica se o modelo de componentes provê suporte ao desenvolvimento durante as seguintes fases:

1. **Modelagem**: se o modelo de componentes provê suporte à modelagem e ao projeto de sistemas. Em alguns modelos, componentes são mencionados somente para a descrição da arquitetura do sistema, por exemplo, em Linguagens de Descrição de Arquiteturas (*Architecture Description Language*, ADLs), ou para a especificação de propriedades desejadas no sistema.

2. **Implementação**: se o modelo de componentes provê suporte para produção de código; isto pode acontecer se houver suporte de ferramentas para geração de código-fonte ou de código binário executável.

3. **Empacotamento** (*packaging*): a forma como os arquivos do componente estão organizados e como são recuperados para a composição do sistema. O modelo de componentes pode ditar um formato para um pacote de componentes que depois facilita a sua manipulação.

   Por exemplo, em Java, classes são empacotados em um arquivo JAR, que é um arquivo ZIP com a extensão `.jar`; no modelo OSGi [2], componentes são arquivos JAR e devem conter um arquivo texto `MANIFEST.MF` no formato chave-valor, com metainformação sobre o conteúdo, inclusive a versão do componente; no modelo

EJB, o arquivo JAR deve conter arquivos XML, que descrevem as classes e interfaces do pacote.

Koala [48] é um modelo de componentes para aparelhos eletrônicos e tem como alvo a linguagem C. Os arquivos dos componentes estão dispostos em um repositório no sistema de arquivos, com um diretório por componente, e cujos conteúdo é: um arquivo de descrição na linguagem CDL (*Component Description Language*), o cabeçalho e implementação do componente em C, arquivos de teste e documentação.

4. **Implantação** (*deployment*): neste estágio, os componentes são integrados para compor o sistema. Componentes podem ser integrados em tempo de compilação ou de execução.

   No modelo Koala, os componentes são ligados em tempo de compilação. A ligação entre eles é estática (*static binding*) e baseada em convenções de nomes.

   Componentes podem existir em tempo de execução em um contêiner (modelos CCM e EJB), que provê mecanismos para a ligação. Por exemplo, no modelo EJB, componentes são registrados sob nomes JNDI [11]; um componente requisita ao contêiner uma ligação com o componente registrado sob um nome JNDI.

2.4.2 Mecanismos de Composição

Esta dimensão identifica como os componentes são interconectados e se comunicam. No entanto, apesar da comunicação entre componentes ser um aspecto essencial à execução do sistema, ela não é expressa explicitamente em todo modelo de componentes; alguns modelos consideram que ela será tratada por algum mecanismo de suporte externo.

Esta dimensão diferencia as especificações de propriedades funcionais e extrafuncionais do sistema. Assim, a interface de um componente passa a ter duas funções: (1) especificar as propriedades do componente (funcionais e extrafuncionais) e (2) identificar os pontos de conexão pelos quais os componentes se conectam.

Interface

Especificação de interfaces é uma carascterística indispensável a um modelo de componentes. Interfaces podem ser definidas com elementos de linguagens de programação ou em linguagens especiais. Existem várias linguagens para especificar interfaces de componentes e suas interconexões: linguagens de modelagem, como UML ou ADLs, Linguagens de Descrição de Interfaces (*Interface Description Language*, IDL), ou mecanismos de linguagens de programação, como interfaces em Java ou classes abstratas em C++; também é possível utilizar outros mecanismos, como *structs* em C.

Interfaces descritas em linguagens especiais são depois traduzidas para a linguagem de programação. Em alguns modelos de componente (COM), a interface também é definida em um formato binário que é utilizado como uma representação padrão durante a implantação e execução do sistema. Em Java, é possível utilizar reflexão computacional para investigar componentes e descobrir suas interfaces durante a execução do sistema.

Alguns modelos de componentes distinguem interfaces entre providas (especificação das funcionalidades providas pelo componente) e requeridas (funcionalidades requeridas pelo componente).

O conceito de contratos pode ser associado a interfaces. Contratos visam garantir que um componente se comporta de acordo com sua especificação se suas interfaces forem alimentadas com entradas esperadas e se o ambiente estiver de acordo com o esperado pelo componente. De acordo com Beugnard et al [4], contratos podem ser classificados em um hierarquia de quatro níveis independentes, que formam um contrato global. A classificação de Crnkovic et al utiliza os três primeiros níveis de contratos definidos por Beugnard:

1. **Contrato sintático**: descreve o aspecto sintático ou assinatura de uma interface. Este contrato garante a utilização correta do componente. Assim, um componente que utiliza outro deve utilizar os tipos, métodos, sinais e portas corretos, e tratar as exceções de acordo com o contrato. Este é o tipo mais comum e que com mais facilidade de certificar, porque pode ser verificado, de maneira estática ou dinâmica, com técnicas de verificação de tipos.

2. **Contrato semântico**: reforça os contratos sintáticos certificando que os valores de parâmetros e de variáveis de estado dos componentes estão em intervalos corretos. Isto pode ser verificado por pré e póscondições e invariantes.

3. **Contrato de comportamento**: regula o comportamento dinâmico dos serviços. Este contrato expressa as restrições de composição (por exemplo, de ordem temporal) ou o comportamento interno de componentes (por exemplo, dinâmica de estados internos).

Ligação

A conexão entre interfaces de componentes é chamada de ligação (*binding*). A ligação está relacionada ao ciclo de vida dos componentes; ela pode ocorrer em tempo de compilação (quando o compilador conecta componentes por meio de mecanismos da linguagem de programação), ou em tempo de execução, quando mecanismos de conexão são providos pela infraestrutura de execução. Uma infraestrutura de execução pode ser um componente de *middleware* dedicado ou um *framework* de componentes.

A ligação pode ser *exógena* ou *endógena*. Na ligação exógena, a informação sobre a ligação é externa ao componente; componentes não têm conhecimento de com quem estão ligados. Na ligação endógena, um componente se comunica diretamente com a interface de outro; a informação sobre a ligação é interna ao componente.

A ligação pode ser feita por intermédio de conectores. Conectores têm dois propósitos: (1) possibilitar a ligação exógena ou (2) introduzir funcionalidades adicionais, especialmente para mediação entre componentes. Em alguns modelos de componentes, conectores são implementados como tipos especiais de componentes, como adaptadores ou *proxies*, seja para prover propriedades adicionais, funcionais ou extrafuncionais, seja para estender os mecanismos de comunicação.

A especificação de interfaces em um modelo de componentes pode definir implicitamente o tipo de interação entre componentes de acordo com determinados estilos arquiteturais. Em muitos casos, um modelo de componentes provê um único estilo de comunicação básico, por exemplo, "requisição-resposta" ou " *pipe & filters*", mas outros, como Fractal [6, 7], favorecem a construção de diferentes estilos arquiteturais.

Composição

O conceito de composição indica a possibilidade de componentes ligados resultarem em um novo componente, que poderá ser utilizado na composição de um outro sistema ou de outro componente.

As tecnologias de componentes existentes dão suporte a dois tipos de composição: (1) na composição de primeira ordem, a composição não é recursiva; (2) na composição recursiva, ou hierárquica, componentes podem formar um novo componente, composto, e que satisfaz as propriedades de um componente elementar de acordo com o modelo de componentes.

Muitos modelos de componentes oferecem suporte, mesmo que parcial, à composição recursiva. Por exemplo, em alguns modelos de componentes, interfaces podem ser compostas recursivamente na fase de projeto, mas não na fase de implantação.

Classificação de mecanismos de composição

Das observações anteriores sobre interfaces, ligação e composição, é possível identificar as seguintes características nesta dimensão da classificação:

1. **Especificação de interfaces**. As seguintes características das interfaces são identificadas:

   (a) Tipo de interface: a interface pode ser baseada em operações (invocam-se métodos da interface) ou em portas (são passados dados para a interface).

(b) Existência de distinção entre interfaces providas e requeridas.

(c) Existência de funcionalidades específicas que aparecem somente neste modelo de componentes (por exemplo, tipos especiais de portas, ou operações opcionais).

(d) Linguagem utilizada para especificar a interface.

(e) Quais contratos sobre as interfaces o modelo oferece, dentre sintático, semântico e de comportamento.

2. **Interações**. As seguintes características sobre interações são identificadas:

(a) O estilo de interação que descreve o principal estilo arquitetural utilizado.

(b) Se a comunicação entre interfaces é síncrona ou assíncrona.

(c) O tipo de ligação: endógeno/exógeno e existência de composição recursiva.

2.4.3 Propriedades extrafuncionais

Propriedades de componentes podem ser classificadas como funcionais ou extrafuncionais: propriedades funcionais descrevem funções ou serviços de um componente; propriedades extrafuncionais (*extra-functional properties*, EFP) especificam a qualidade ou características de interesse desses serviços. Na área de DBC, existe ainda uma distinção entre propriedades de componentes e propriedades do sistema: uma propriedade do sistema é o resultado da composição das mesmas propriedades dos componentes do sistema.

Modelos de componentes podem oferecer suporte ao gerenciamento de propriedades extrafuncionais. De acordo com a Figura 2.10, o gerenciamento de propriedades extrafuncionais pode ser classificados em duas dimensões:

- **Endógeno ou exógeno:** uma EFP pode ser gerenciada pelo componente (gerenciamento endógeno, abordagens A e B) ou pelo sistema (gerenciamento exógeno, abordagens C e D).

- **Por colaboração ou gerenciado pelo sistema:** uma EFP pode ser gerenciada para todo o sistema (abordagens B e D) ou em cada colaboração entre componentes (abordagens A e C).

- **Gerenciamento endógeno por colaboração (abordagem A):** o modelo de componentes não provê suporte para gerenciamento de EFPs; o desenvolvedor do componente é responsável por implementar este gerenciamento. Esta abordagem possibilita incluir políticas de gerenciamento de EFPs que são otimizadas para um

sistema específico, e possibilita adotar mais de uma política em um mesmo sistema. Esta hetereogeneidade é particularmente útil com componentes preexistentes que devem ser integrados.

Por outro lado, o fato de que tais políticas não estão padronizadas pode trazer dificuldades para integração quando componentes têm arquiteturas muito diferentes.

- **Gerenciamento endógeno pelo sistema (abordagem B):** nesta abordagem, existe um mecanismo na plataforma de execução dos componentes que contém políticas para gerenciar EFPs em componentes individuais e EFPs que envolvem vários componentes. A capacidade de negociar a maneira pela qual EFPs são tratadas requer que os próprios componentes tenham algum conhecimento sobre como as EFPs afetam seu funcionamento. Isto é uma forma de reflexão computacional.

- **Gerenciamentos exógeno por colaboração (abordagem C) e exógeno pelo sistema (abordagem D):** nestas abordagens os componentes são projetados para implementar somente aspectos funcionais e nenhuma EFP. Assim, no ambiente de execução, os componentes são englobados em um contêiner. Este contêiner tem conhecimento de como gerenciar EFPs. Contêineres podem estar conectados a contêineres de outros componentes (abordagem C) ou podem interagir com um mecanismo da plataforma de execução que gerencia EFPs numa escala de sistema (abordagem D). A abordagem com contêineres é uma maneira de separar aspectos da aplicação em camadas de forma que componentes se concentrem em aspectos funcionais e os contêineres, em aspectos extrafuncionais das aplicações.

  Vantagens desta abordagem são que componentes se tornam mais genéricos, dado que não são necessárias modificações para integrá-los em sistemas que utilizam diferentes políticas para EFPs; outra vantagem é que os componentes são mais simples e portanto de menor custo de implementação. Uma desvantagem é que esta abordagem pode reduzir a performance do sistema.

Classificação segundo suporte a propriedades extrafuncionais

Esta dimensão classifica os modelos de componentes segundo sua abordagem para propriedades extrafuncionais:

1. **Gerenciamento:** qual o suporte do modelo de componentes ao gerenciamento de EFPs ?

2. **Especificação:** o modelo de componentes provê meios para especificar e gerenciar EFPs específicas ? Para que propriedades ou que tipos de propriedade ?

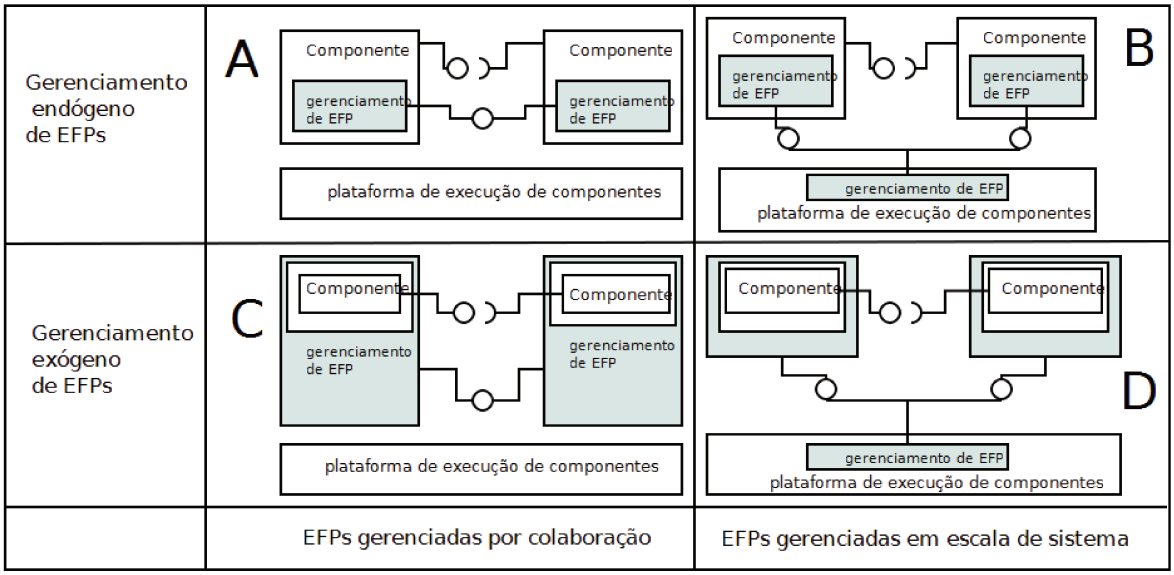


Figura 2.10: Gerenciamento de EFPs

3. **Composição:** o modelo de componentes provê meios, métodos ou técnicas para composição de determinadas EFPs ? Que tipo de composição ?

2.4.4 Domínio

Alguns modelos de componentes são voltados para aplicações de domínios específicos, como sistemas embarcados ou sistemas de informação. Nestes casos, os requisitos do domínio da aplicação permeiam o modelo de componentes. Os benefícios de modelos de componentes específicos são de que a tecnologia de componentes facilita satisfazer alguns destes requisitos. Tais modelos são consequentemente limitados e só serão facilmente utilizados em domínios que estão sujeitos a estes requisitos. Por exemplo, modelos de componentes para sistemas embarcados devem observar as limitações inerentes às plataformas em que o sistema será executado, como limitações de memória e processamento.

Alguns modelos de componentes são de propósito geral. Eles provêem mecanismos básicos para a especificação e composição de componentes, mas não assumem nenhuma arquitetura específica além de premissas básicas (por exemplo, o estilo de interação, suporte a sistemas distribuidos, compilação ou implantação em tempo de execução). Uma solução para que modelos de componentes sejam ao mesmo tempo genéricos e supram domínios específicos é o uso de *frameworks*. Um *framework* é uma extensão de um modelo de componentes que pode ser utilizada, mas não é obrigatória.

Existe ainda um terceiro tipo: os modelos de geração; estes são utilizados para instanciar outros modelos de componentes. Eles provêem princípios comuns, e partes comuns

de tecnologias (por exemplo, modelagem), enquanto outras partes são específicas (por exemplo, diferentes implementações).

Assim, é possível classificar modelos de componentes como:

1. de propósito geral;
2. de domínios específicos;
3. de geração

2.4.5 Resultado da classificação de modelos de componentes

Em Crnkovic et al [15], os modelos de componentes descritos na Seção 2.3 são classificados; os resultados são apresentados nas Tabelas 2.1, 2.2, 2.3 e 2.4.

Modelo	**Modelagem**	**Implementação**	**Empacotamento**	**Implantação**
CCM	N/A	Independente de linguagem	Arquivos (JARs, DLLs)	Tempo de execução
EJB	N/A	Java	Arquivos EJB-Jars	Tempo de execução
Fractal	ADLs (Fractal ADL, Fractal IDL) Anotações (Fractlet)	Java (frameworks Julia, Cecilia) C/C++ (Think) .Net (FracNet)	Repositório em sistema de arquivos	Tempo de execução
JavaBeans	N/A	Java	Arquivos JAR	Compilação
OpenCOM	N/A	Linguagem OO	DLL	Tempo de execução
OSGi	N/A	Java	OSGi JARs (“bundles”)	Compilação e execução
Cosmos*	N/A	Linguagem OO	Arquivos JAR	Compilação e execução

Tabela 2.1: Dimensão de ciclo de desenvolvimento

Modelo	**Tipo de interface**	**Distinção entre providas e requeridas**	**Funcionalidades**	**Linguagem de interface**	**Níveis de interface (sintático, semântico, de comportamento**
CCM	Baseado em operações e portas	Sim	N/A	C	Sintático
EJB	Baseado em operações	Não	N/A	Java + Anotações	Sintático
Fractal	Baseado em operações	Sim	Interfaces do componente e de controle	IDL, Fractal ADL, Java ou C, Protocolo de comportamento	Sintático e de comportamento
JavaBeans	Baseado em operações	Sim	N/A	Java	Sintático
OpenCOM	Baseado em operações	Não	Estende modelo COM com interfaces de ciclo de vida, introspecção, etc;	C, Java	Sintático
OSGi	Baseado em operações	Sim	Interface dinâmicas	Java	Sintático
Cosmos*	Baseado em operações	Sim	Interface dinâmicas	Java	Sintático

Tabela 2.2: Dimensão de mecanismos de composição: especificação de interfaces

Modelo	Estilo de interação	Comunicação	Ligação	
			Exógena	Hierárquica
CCM	Requisição/resposta, gatilhos (triggers)	síncrono e assíncrono	Não	Não
EJB	Requisição/resposta	síncrono e assíncrono	Não	Não
Fractal	Diversos estilos	síncrono e assíncrono	Sim	Delegação e agregação
JavaBeans	Requisição/resposta, gatilhos (triggers)	síncrono	Não	Delegação e agregação
OpenCOM	Requisição/resposta	síncrono	Não	Delegação e agregação
OSGi	Requisição/resposta	síncrono	Não	Não
Cosmos*	Requisição/resposta	síncrono	Não	Não

Tabela 2.3: Dimensão de mecanismos de composição: interação

Modelo	Gerenciamento de EFP	Especificação de propriedades
CCM	Exógeno pelo sistema (D)	N/A
EJB	Exógeno pelo sistema (D)	N/A
Fractal	Exógeno por colaboração (C)	Possibilita adicionar propriedades (por meio de controladores de propriedades)
JavaBeans	Endógeno por colaboração (A)	N/A
OpenCOM	Endógeno por colaboração (A)	N/A
OSGi	Endógeno por colaboração (A)	N/A
Cosmos*	Endógeno por colaboração (A)	N/A

Tabela 2.4: Propriedades extrafuncionais

2.5 Trabalhos Relacionados

As diferentes atividades do Desenvolvimento Baseado em Componentes podem ser realizadas com suporte de ferramentas. As ferramentas podem oferecer suporte específico para alguns modelos de componentes. Esta Seção descreve as ferramentas de suporte aos modelos de componentes descritos na Seção 2.3. São analisadas ferramentas relacionadas a este trabalho que oferecem suporte a dois aspectos do DBC: suporte a ligação de componentes e versões de componente em tempo de execução.

O suporte a ligação de componentes pode ocorrer de diferentes maneiras: 1) uma ferramenta gera o código-fonte dos componentes de acordo com os mecanismos de ligação previstos pelo modelo de componentes; 2) a ferramenta armazena a descrição da arquitetura em um metamodelo próprio da ferramenta; 3) a infraestrutura prevista pelo modelo de componentes (o contêiner) gerencia as ligações entre componentes.

A tabela 2.5 exibe as abordagens adotadas pelos diferentes modelo de componentes apresentados. No modelo JavaBeans, existem ferramentas para as abordagens com geração de código fonte (Netbeans) e em metamodelo próprio (Bean Builder). No modelo EJB 2.0, as abordagens são complementares: a ferramenta XDoclet gera o código fonte do componente, e ligação em tempo de execução é gerenciada pelo mecanismo JNDI no contêiner. Nos modelos Fractal, OSGi e CCM, o gerenciamento de ligações é feito pelo contêiner: o modelo Fractal é concretizado no framework Julia; os modelos OSGi e CCM contam com diversas implementações de conteineres.

Modelo	**Geração de código-fonte**	**Geração de metamodelo**	**Gerenciamento por contêiner**
JavaBeans	Netbeans[12]	Bean Builder[18]	
EJB	XDoclet[45]		JNDI
Fractal			Framework Julia [6]
OpenCOM			Contêiner OpenCom
OSGi			Contêiner OSGi
CCM			Contêiner CCM

Tabela 2.5: Suporte a ligação de componentes

O suporte a versões de componentes é mais escasso. Dentre os modelos apresentados, somente o modelo OSGi oferece suporte a versionamento de componentes. Cada componente declara, em seu arquivo manifesto, sua versão e as versões de suas dependências. O contêiner OSGi conecta as versões dos componentes de acordo com as descrições dos manifestos.

Capítulo 3

O modelo Cosmos*

O modelo Cosmos* [27] ("*Cosmos estrela*") é uma extensão do modelo de componentes COSMOS (*Component Structuring Model for Object-oriented Systems*) apresentado na Seção 2.3.7 [16, 17].

Este capítulo apresenta o modelo de componentes Cosmos*, que é a base da ferramenta CosmosLoader. O modelo Cosmos* é um mapeamento de conceitos de Arquitetura de Software e DBC para linguagens orientadas a objetos; neste capítulo, os exemplos do modelo Cosmos* serão escritos na linguagem Java [28].

As Seções 3.1 a 3.4 descrevem o modelo Cosmos*. A Seção 3.5 lista as extensões do modelo Cosmos* em relação ao modelo Cosmos original. As Seções 3.6 a 3.10 descrevem em detalhes as extensões propostas pelo modelo Cosmos*.

3.1 Visão geral

O modelo Cosmos* é um modelo de componentes que apresenta definições para componentes arquiteturais, conectores e configurações concretas. Estes elementos são mapeados para as abstrações de linguagens orientadas a objetos modernas, como pacotes, módulos e *namespaces*. A Figura 3.1 ilustra um componente `A` com duas interfaces providas `IA` e `IManager` e uma interface requerida `IB`.

- No modelo Cosmos*, componentes e conectores são representados como pacotes com o nome do componente em letras minúsculas. Assim, na Figura 3.1, o componente `A` é representado pelo pacote `a`.

- O pacote `a.spec`, contido no pacote `a`, é o pacote de especificação do componente `A`. Ele é ainda dividido nos dois outros pacotes `a.spec.prov` e `a.spec.req`, que contêm as interfaces providas e requeridas do componente, respectivamente. A interface provida `IA` está definida no pacote `a.spec.prov` e a interface requerida

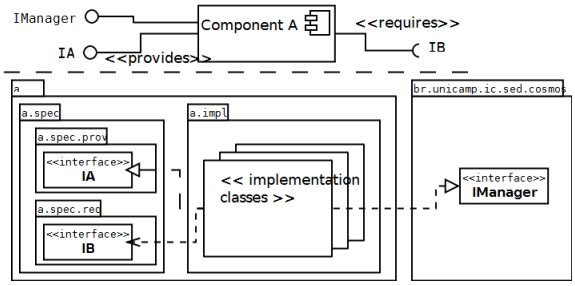


Figura 3.1: Estrutura de pacotes de um componente Cosmos*

`IB` está definida no pacote `a.spec.req`. O modelo Cosmos* contém um modelo de especificação, que define a especificação de um componente (Seção 3.2).

- Componentes e conectores Cosmos* apresentam a interface de gerenciamento `IManager`; esta é uma interface especial para recuperar meta-informações sobre o componente. Ela possibilita que usuários de um componente Cosmos* examinem as interfaces de um componente e a conectem a interfaces de outros componentes. Esta interface não é definida no pacote `spec` de um componente, mas no pacote especial `br.unicamp.ic.sed.cosmos`, de modo que sua definição é compartilhada por todo componente Cosmos*. A interface `IManager` é detalhada na Seção 3.2.2.

- As classes de implementação do componente `A` estão no pacote `a.impl`. Estas classes implementam as interfaces providas pelo componente, inclusive a interface `IManager`, e fazem uso da interface requerida `IB`. O modelo Cosmos* contém um modelo de implementação que detalha a implementação de um componente (Seção 3.3).

Conexões entre dois ou mais componentes são intermediadas por um conector Cosmos*, que é um componente mais simples responsável pela comunicação entre interfaces requeridas de um componente e providas de outros. Ele é implementado de acordo com o padrão de projeto `Adapter` [25]. Um conector Cosmos* não apresenta um pacote de especificação; ao invés disso, ele implementa a conexão entre as interfaces requerida e provida de dois ou mais componentes. A Figura 3.2 ilustra um componente Cosmos* `X`, composto por dois componentes `A` e `B` e um conector `AB`. Na Figura 3.2, o tipo da interface provida `IB` do componente `B` é `b.spec.prov.IB`, que é diferente do tipo da interface requerida `IB` do componente A, que é `a.spec.req.IB`. Assim, o conector `AB` adapta a interface provida `IB` do componente `B` para a interface requerida `IB` do componente `A`. O conector `AB` é implementado em um pacote `ab` e, assim como um componente Cosmos*, também implementa a interface `IManager` do pacote `br.unicamp.ic.sed.cosmos`. O modelo Cosmos* contém um modelo de conectores, que define conectores, e é detalhado na Seção 3.4.

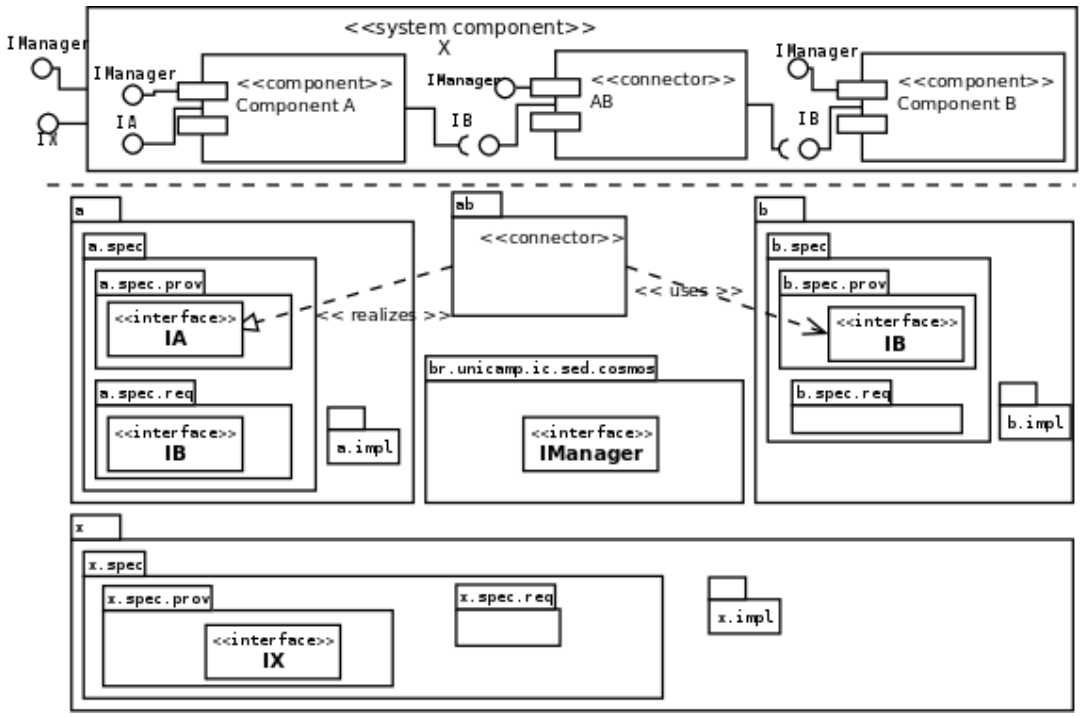


Figura 3.2: Configuração arquitetural

Na Figura 3.2, os componentes `A` e `B`, o conector `AB` e as conexões entre as interfaces providas e requeridas constituem a **configuração arquitetural** do **componente composto** `X`. O modelo Cosmos* também define um modelo de composição para criar novos componentes a partir de componentes existentes. O modelo de composição é detalhado na Seção 3.6.

Ainda na Figura 3.2, o componente `X` é um **componente de sistema**. Um componente de sistema é um componente Cosmos* que é um sistema independente e pode ser implantado e executado. O modelo Cosmos* também define um modelo para componentes de sistema e execução de sistemas baseados em componentes Cosmos*, detalhado na Seção 3.7.

3.2 Modelo de especificação

O modelo de especificação define os pacotes contendo as definições das interfaces de componentes Cosmos* e a interface `IManager`, comum a todos os componentes Cosmos*.

3.2.1 Pacote de especificação de componentes Cosmos*

A especificação de um componente Cosmos* visa enfatizar a separação entre a especificação do componente, que deve ser visível, da sua implementação. A implementação de um componente não deve ser acessível externamente, de modo que toda comunicação com o componente ocorre por meio de suas interfaces.

Como definido na Seção 3.1, a especificação de um componente Cosmos* é descrita por um pacote `spec`, localizado em um pacote maior que representa o componente. Cada interface provida do componente arquitetural é mapeada em uma interface UML no pacote `spec.prov` e cada interface requerida, em uma interface UML no pacote `spec.req`. Na Figura 3.1, o componente `A` apresenta duas interfaces providas `IA` e `IManager` e uma interface requerida `IB`. A Figura 3.3 apresenta as definições destas interfaces:

```
package a.spec.prov;
public interface IA { /* métodos na interface provida IA */ }

package a.spec.req;
public interface IB { /* métodos na interface requerida IB */ }
```

Figura 3.3: Definições das interfaces providas e requeridas do componente A.

3.2.2 A interface IManager

Um componente Cosmos* deve obrigatoriamente apresentar uma interface provida `IManager`, que é implementada por uma classe `Manager` no pacote de implementação do componente. A interface `IManager` é definida no pacote `br.unicamp.ic.sed.cosmos` e tem três propósitos:

1. Uma instância desta interface representa uma instância do componente em tempo de execução. Assim, diferentes instâncias de um componente podem existir em tempo de execução, cada uma com um estado diferente, e representado por diferentes instâncias da interface `IManager`.

2. Uma instância da interface `IManager` possibilita descobrir em tempo de execução quais são as interfaces providas e requeridas de um componente.

3. Através da interface `IManager`, é possível conectar uma interface requerida de um componente à interface provida de outro componente.

A Figura 3.4 ilustra os métodos da interface `IManager`:

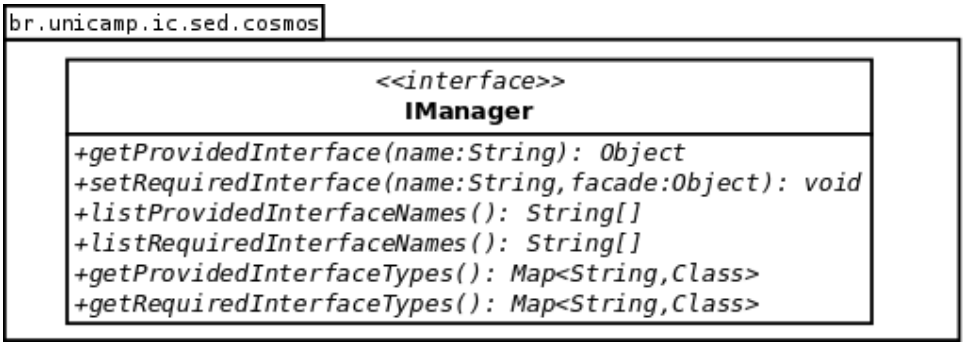


Figura 3.4: A interface `IManager`, em UML

1. **getProvidedInterface(name : String) : Object**. Retorna um objeto que implementa uma interface provida deste componente; o parâmetro é o nome da interface provida solicitada.

2. **setRequiredInterface(name : String, facade : Object) : void**. Conecta uma interface requerida deste componente. O primeiro parâmetro é o nome da interface requerida a ser conectada; o segundo é uma interface provida de outro componente, que implementa os métodos esperados pela interface requerida.

3. **listProvidedInterfaceNames() : String**[]. Retorna um *array* com os nomes das interfaces providas deste componente.

4. **listRequiredInterfaceNames() : String**[]. Retorna um *array* com os nomes das interfaces requeridas deste componente.

5. **getProvidedInterfaceTypes() : Map<String,Class>**. Retorna uma instância da interface `java.util.Map`) onde cada chave é o nome de uma interface provida e o valor é um objeto `Class` que representa o tipo da interface.

6. **getRequiredInterfaceTypes() : Map<String,Class>**. Este método é semelhante ao método `getProvidedInterfaceTypes`, mas retorna os nomes e tipos das interface requeridas do componente.

A Figura 3.5 ilustra a definição da interface `IManager`:

3.3 Modelo de implementação

As classes de implementação de um componente estão localizadas em seu pacote `impl`. A Figura 3.6 ilustra as classes localizadas no pacote de implementação do componente `A`,

```
1  package br.unicamp.ic.sed.cosmos;
2  import java.util.Map;
3  public interface IManager {
4    public Object getProvidedInterface(String name);
5    public void setRequiredInterface(String name, Object facade);
6    public String[] listProvidedInterfaceNames();
7    public String[] listRequiredInterfaceNames();
8    public Map<String,Class<?>> getProvidedInterfaceTypes();
9    public Map<String,Class<?>> getRequiredInterfaceTypes();
10 }
```

Figura 3.5: Definição, em Java, da interface `IManager`

`a.impl`. Este pacote deve obrigatoriamente conter a classe `ComponentFactory`, a classe `Manager`, que implementa a interface `IManager` (Seção 3.2.2) e um conjunto de classes `Façade`, uma para cada interface provida do componente diferente de `IManager`.

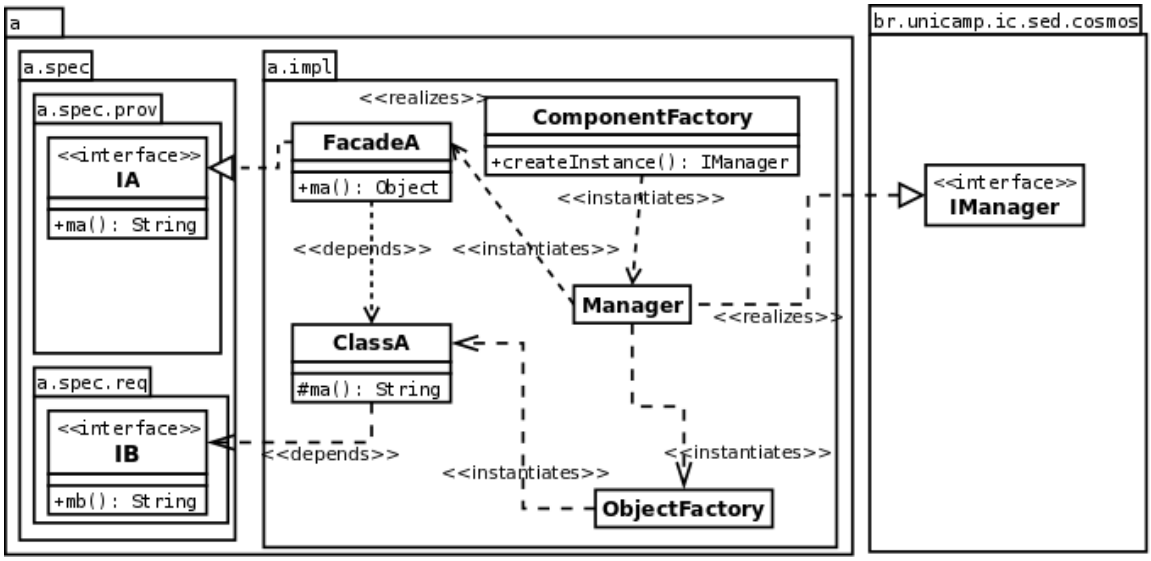


Figura 3.6: Diagrama de classes do pacote de implementação do componente `A`

3.3.1 A classe `ComponentFactory`

A classe `ComponentFactory` de um componente é implementada de acordo com o padrão de projeto *Factory Method* [25]. Ela é responsável por instanciar o componente. Esta classe contém um único método `createInstance()`, estático, responsável por instanciar o componente. É a única classe no pacote `impl` com visibilidade pública.

A implementação da classe `ComponentFactory` do componente `A` é ilustrada na Figura 3.7.

```
1 package a.impl;
2 import br.unicamp.ic.sed.cosmos.IManager;
3 public class ComponentFactory {
4   public static IManager createInstance() {
5     return new Manager();
6   }
7 }
```

Figura 3.7: Implementação da classe ComponentFactory do componente `A`

3.3.2 A classe Manager

Cada componente Cosmos* contém uma classe `Manager` em seu pacote de implementação, que implementa a interface `IManager`. O modelo Cosmos* não estabelece regras para a implementação da classe `Manager`, de modo que o desenvolvedor de um componente pode escolher como implementá-la. Assim, cada componente pode ter sua implementação da classe `Manager`. Diferentes implementações da classe `Manager` gerenciam de maneira diferente os objetos que implementam as interfaces do componente.

A Figura 3.8 sugere uma possível implementação, que consiste em criar um método `setProvidedInterface`. Este método armazena uma instância de cada classe `Façade` em uma tabela indexada pelo nome da interface (linha 9). O método `getProvidedInterface` apenas retorna o objeto `Façade` que foi armazenado na tabela com o nome da interface solicitada (linha 13).

De maneira semelhante, os métodos `setProvidedInterfaceType` e `setRequiredInterfaceType` armazenam os tipos das interfaces providas e requeridas (linhas 19 e 22).

Durante a instanciação do componente, o construtor da classe `Manager` (linhas 27 a 35) invoca os métodos `setProvidedInterface`, `setProvidedInterfaceType` e `setRequiredInterfaceType` para armazenar os objetos que implementam as interfaces providas do componente (Figura 3.8).

Outra possível implementação da classe `Manager` consiste em estender uma classe existente que implementa os métodos da interface `IManager`; por exemplo, o pacote `br.unicamp.ic.sed.cosmos` contém a classe abstrata `AManager`, que implementa a interface `IManager`. Assim, uma possível implementação da classe `Manager` consiste em estender a classe `AManager`. É importante enfatizar que esta abordagem é uma possibilidade e não é obrigatória no modelo Cosmos*. A Seção 3.8 descreve as classes de implementação no pacote `br.unicamp.ic.sed.cosmos`.

```
1  package a.impl;
2  import java.util.Map;
3  import a.spec.prov.IManager;
4  class Manager implements IManager {
5    private Map providedInterfaces;
6    private Map providedInterfaceTypes, requiredInterfaceTypes;
7    /* Armazena um objeto que implementa uma interface provida */
8    void setProvidedInterface(String name, Object facade) {
9      this.providedInterfaces.put(name, facade);
10   }
11   /* Retorna um objeto que implementa a interface solicitada */
12   public Object getProvidedInterface(String name) {
13     return this.providedInterfaces.get(name);
14   }
15   /* Os métodos setProvidedInterfaceType e
16    * setRequiredInterfaceType armazenam os tipos das interface
17    * providas e requeridas */
18   void setProvidedInterfaceType(String name, Class type) {
19     this.providedInterfaceTypes.put(name, type);
20   }
21   void setRequiredInterfaceType(String name, Class type) {
22     this.requiredInterfaceTypes.put(name, type);
23   }
24   /* O construtor invoca os métodos setProvidedInterface,
25    * setProvidedInterfaceType e setRequiredInterfaceType para
26    * armazenar informações sobre as interfaces do componente. */
27    Manager() {
28     this.setProvidedInterface("IA", new FacadeA(this));
29     this.setProvidedInterface("IManager", this);
30     this.setProvidedInterfaceType("IA", a.spec.prov.IA.class);
31     this.setProvidedInterfaceType(
32           "IManager",
33       br.unicamp.ic.sed.cosmos.IManager.class);
34     this.setRequiredInterfaceType("IB", a.spec.req.IB.class);
35    }
36 }
```

Figura 3.8: Sugestão de implementação da classe `Manager` do componente `A`

3.3.3 As classes `Façade`

As classes `Façade` em um componente Cosmos* são o ponto de entrada para implementações das interfaces providas de um componente. Cada interface provida é realizada por uma classe `Façade` (exceto a interface `IManager`, que é realizada pela classe `Manager`). Uma classe `Façade` delega a implementação da interface para outra classe dentro do componente, de acordo com o padrão de projeto *Façade* [25].

Na Figura 3.6, o componente `A` tem uma interface provida `IA`, realizada pela classe `FacadeA`. Esta classe implementa a interface `IA` e delega chamadas a seus métodos para a classe `ClassA`. A implementação da classe `FacadeA` é ilustrada na Figura 3.9:

```
1  package a.impl;
2
3  class FacadeA implements a.spec.prov.IA {
4      private ClassA a;
5      private Manager manager;
6      FacadeA(Manager mgr) {
7          this.manager = mgr;
8          this.a = new ClassA(this.manager);
9      }
10     public Object ma() { return this.a.ma(); }
11 }
```

Figura 3.9: Implementação da classe `FacadeA`.

O construtor da classe `FacadeA` tem um parâmetro usado pela classe para receber uma instância do objeto Manager do componente (linha 6). Esta referência ao objeto `Manager` é armazenada no membro `FacadeA` (linha 7) e deve ser passada para as classes de implementação para as quais o objeto Façade delega chamadas de métodos (linha 8). Assim, as classes de implementação podem acessar o objeto `Manager` do componente e usar as interfaces requeridas deste.

3.3.4 Interfaces requeridas

Interfaces requeridas de um componente representam as dependências do componente por serviços providos por outros componentes. Na Figura 3.6, o componente `A` tem uma interface requerida `IB`, com um método `mb()`.

A Figura 3.10 ilustra a implementação da classe `ClassA`. Quando uma classe do componente `A` precisa de um serviço oferecida pela interface `IB`, ela invoca o método `getRequiredInterface` da classe `Manager` e obtém uma instância da interface `IB`. Na Figura 3.10, a classe `ClassA` obtém uma referência à interface requerida `IB` do componente e invoca seu método `mb` (linha 10).

```
1  package a.impl;
2  class ClassA {
3      private Manager manager;
4      ClassA(Manager mgr) {
5          this.manager = mgr;
6      }
7      String ma() {
8          a.spec.req.IB ib = (a.spec.req.IB)
9              this.manager.getRequiredInterface("IB");
10         return "->" + ib.mb();
11     }
12 }
```

Figura 3.10: Instanciação de um sistema através do CosmosLoader

3.3.5 Opcional: a classe `ObjectFactory`

É possível incluir no pacote de implementação de um componente Cosmos* a classe ObjectFactory. Esta classe é opcional no modelo, e pode ser omitida. Ela age como uma fábrica de objetos, como descrito no padrão de projeto *Abstract Factory* [25], e visa reduzir o acoplamento entre classes de implementação do componente.

Quando um componente contém a classe `ObjectFactory`, suas classes de implementação não se referenciam diretamente. Ao invés disso, existe no componente uma interface para cada classe de implementação; as classes de implementação referenciam somente estas interfaces. Para cada uma destas interfaces, existe um método na classe `ObjectFactory` que retorna uma instância da interface.

Diferentemente das interfaces no pacote de especificação, as classes no pacote de implementação têm visibilidade somente no pacote.

A Figura 3.11 ilustra o pacote de implementação do componente, modificado para utilizar a classe `ObjectFactory`:

- A classe `Manager` instancia a classe `ObjectFactory`
- As classes de implementação `FacadeA`, `Class1` e `Class2` não se referenciam diretamente. Ao invés disso, referenciam as interfaces `Interface1` e `Interface2`. A classe `ObjectFactory` apresenta os métodos `getInterface1()` e `getInterface2()`, que retornam implementações destas interfaces.
- A classe `Manager` apresenta o método `getObjectFactory()`, que retorna uma instância da classe `ObjectFactory`. As demais classes de implementação do componente invocam este método para acessar a fábrica de objetos.

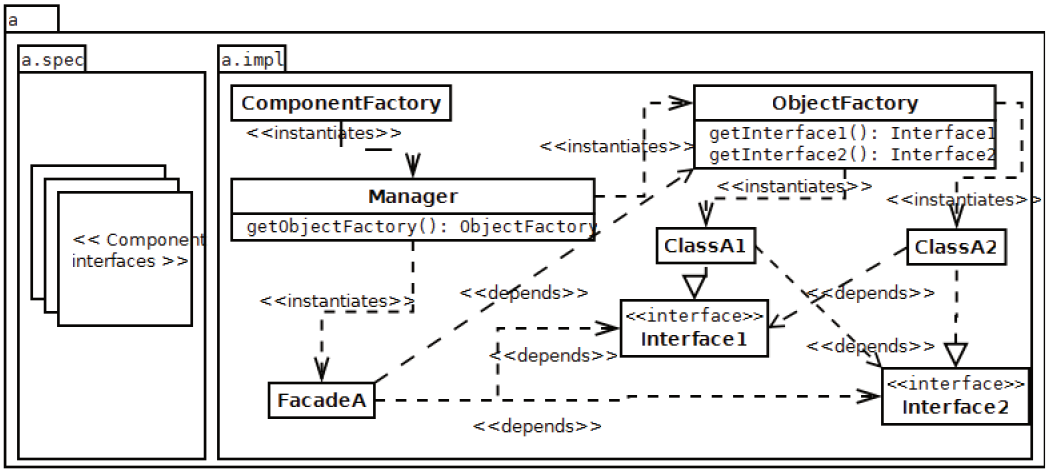


Figura 3.11: Pacote de implementação com a classe `ObjectFactory`

A utilização de uma fábrica de objetos reduz o esforço necessário para manutenção e evolução de um componente mais complexo: como os objetos se comunicam através de interfaces, algumas classes de implementação podem ser modificadas sem que haja um impacto sobre as demais.

Além disso, o uso de uma fábrica de objetos possibilita que diferentes versões de um mesmo componente, através da criação de diferentes fábricas de objetos.

3.4 Modelo de conectores

Um conector Cosmos* implementa a conexão entre interfaces requeridas e providas de dois ou mais componentes, conforme apresentado na Seção 3.1. Conectores são componentes Cosmos* mais simples: são implementados através de pacotes, de maneira similar a componentes Cosmos*, mas não lhes é permitido definir novas interfaces. A Figura 3.12 ilustra uma configuração arquitetural com um conector Cosmos* `AB`, no pacote `ab`, e suas classes.

Como um conector Cosmos* não define novas interfaces, ele não contém pacotes de especificação e implementação; ao invés disso, um conector é materializado em um único pacote, que contém todas as suas classes.

O conector Cosmos* é uma aplicação do padrão *Adapter* [25], e estabelece uma conexão indireta entre as interfaces incompatíveis dos componentes `A` e `B`. A interface requerida do componente `A` e a interface provida do componente `B` são de tipos diferentes e não podem ser conectadas diretamente (a interface requerida do componente `A` é do tipo `a.spec.req.IB` e a interface provida do componente `B` é do tipo `b.spec.prov.IB`). O conector `AB` na Figura 3.12 implementa a interface requerida `IB` do componente `A` e utiliza

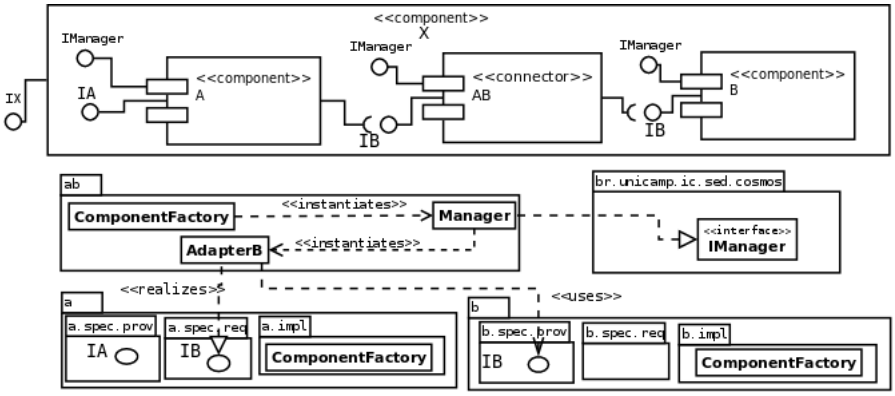


Figura 3.12: Um conector Cosmos* e suas classes.

a interface provida do componente `B`.

Ainda, o conector `AB` possibilita a comunicação entre os componentes sem que o componente `A` referencie interfaces definidas no componente `B`, e, inversamente, sem que `B` referencie interfaces definidas por `A`. Assim, um componente não depende do outro. Este mecanismo é fundamental no modelo Cosmos* para conseguir um baixo acoplamente entre componentes.

Por outro lado, cria-se uma dependência entre o conector e os componentes que ele conecta. Na Figura 3.12, o conector `AB` referencia interfaces definidas nos componentes `A` e `B`, o que qualifica um alto acoplamento entre o conector e os componentes. O modelo Cosmos* estabelece que componentes devem ser desenvolvidos visando sua reutilização, e que conectores devem ser desenvolvidos visando a comunicação entre componentes e a adaptação de suas interfaces, e sem preocupação com seu possível reuso. Um conector pode ser reutilizado se dois ou mais componentes forem reutilizados em uma nova arquitetura e um conector apropriado existir. Caso contrário, um novo conector deve ser desenvolvido.

Na Figura 3.12, o conector `AB` tem as seguintes classes:

- As classes `Adapter` são adaptadores entre as interfaces requeridas e providas dos componentes, de acordo com o padrão *Adapter* [25].
- A interface `IManager`, idêntica à interface `IManager` de um componente Cosmos* (Seção 3.2.2).
- A classe `Manager`, que implementa a interface `IManager`.
- A classe `ComponentFactory`, que instancia o conector.

3.4.1 Classes Adapter em conectores Cosmos*

A Figura 3.13 ilustra a implementação da classe `AdapterB`. O construtor recebe um parâmetro com uma referência ao objeto `Manager` do conector e a armazena em um membro do objeto (linha 6). Para implementar o método `mb()` da interface `IB` do componente `A`, o conector usa o objeto `Manager` para recuperar a interface requerida `IB` (linha 9) e invocar o método mb desta (linha 11). A interface requerida `IB` do conector é do tipo `b.spec.prov.IB`, que é o mesmo tipo da interface provida do componente `B`.

```
1  package ab;
2
3  class AdapterB implements a.spec.req.IB {
4    private Manager manager;
5    AdapterB(Manager mgr) {
6      this.manager = mgr;
7    }
8    public String mb() {
9      b.spec.prov.IB ib = (b.spec.prov.IB)
10         this.manager.getRequiredInterface("IB");
11     return ib.mb();
12   }
13 }
```

Figura 3.13: Classe `AdapterB` do conector `AB`

3.4.2 Classes `Manager` em um conector

A implementação da classe `Manager` de um conector está ilustrada na Figura 3.14. A classe `Manager` do conector é responsável por (1) armazenar os nomes e classes das interfaces providas e requeridas do conector (linhas 5 a 10); e (2) instanciar e armazenar objetos `Adapter`, que funcionam como adaptadores entre as interfaces dos componentes conectados (linhas 11 e 12).

3.5 Extensões ao modelo COSMOS

O modelo Cosmos* [27] estende o modelo COSMOS original [16, 17] com um modelo de composição de componentes (Seção 3.6) e apresentando abordagens para componentes de sistema (Seção 3.7) e bibliotecas (Seção 3.9). Assim, as diferenças entre os modelos Cosmos* e COSMOS são apresentadas na Tabela 3.1:

```
1  package ab;
2
3  class Manager implements br.unicamp.ic.sed.cosmos.IManager {
4    Manager() {
5      this.setProvidedInterfaceTypes(new Class[] {
6        a.spec.req.IB.class });
7      this.setProvidedInterfaceNames(new String[] { "IB" });
8      this.setRequiredInterfaceTypes(new Class[] {
9        b.spec.prov.IB.class });
10     this.setRequiredInterfaceNames(new String[] { "IB" });
11     AdapterB adapter = new AdapterB(this);
12     this.setProvidedInterface("IB", adapter);
13   }
14 }
```

Figura 3.14: Implementação da classe `Manager` do conector `AB`

	COSMOS Original	Cosmos*
Modelo de implementação	•	•
Modelo de conectores	•	•
Interface IManager	Definida pelo componente	Definição comum (pacote br.unicamp.ic.sed.cosmos)
Classe ObjectFactory	Obrigatória	Opcional
Modelo de composição		•
Componente de sistema		•
Tratamento de dependências externas		•

Tabela 3.1: Extensões ao modelo Cosmos

3.6 Modelo de composição

Um componente Cosmos* composto (1) instancia outros componentes Cosmos*; (2) delega para estes a implementação de suas interfaces providas; e (3) expõe as interfaces de seus componentes internos. Assim, componentes internos a um composto não são visíveis por outros componentes do sistema. Esta definição recursiva é fundamental em um modelo de componentes [44] por possibilitar a criação de novos componentes a partir de componentes existentes.

O conjunto de componentes e conectores internos de um composto e as conexões entre as interfaces destes compõem a configuração arquitetural do composto, de acordo com a terminologia apresentada em [19].

A Figura 3.15 ilustra o componente composto `X` da Figura 3.12 revelando suas classes de implementação `FacadeX` e `AdapterXReq`, pelas quais o composto usa seus componentes internos `A` e `B`. Ainda na Figura 3.15, a classe Manager do componente X estende a classe `AManagerComposite` do pacote `br.unicamp.ic.sed.cosmos`; a classe Manager de um componente composto é responsável por gerenciar os componentes internos.

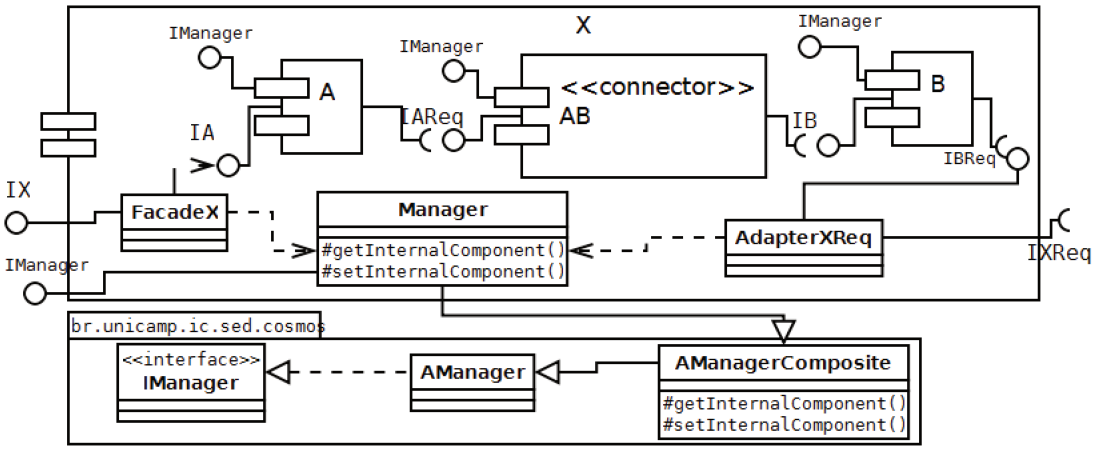


Figura 3.15: Um componente composto.

As classes de implementação de um componente composto diferem das de um componente elementar: na Figura 3.15, a classe `FacadeX` do composto usa a interface requerida `IA` do componente interno `A`. O componente composto `X` contém a classe `AdapterXReq`, semelhante às classes *Adapter* de conectores (Seção 3.4.1), que adapta a interface requerida `IXReq` do composto à interface requerida `IBReq` do componente interno `B`.

3.6.1 Classe Manager em um componente Cosmos* composto

A classe `Manager` de um componente composto implementa a interface `IManager`, conforme descrito na Seção 3.2.2. A classe `Manager` de um composto contém os métodos `getInternalComponent` e `setInternalComponent`, ambos com visibilidade de pacote. As assinaturas destes métodos são:

- **setInternalComponent(String name, IManager component): void**. Registra um componente como interno ao composto. O primeiro parâmetro é o nome sob o qual o componente interno será registrado; o segundo parâmetro, do tipo `IManager`, é a instância do componente a registrar.
- **getInternalComponent(String name): IManager**. Recupera um componente previamente registrado como interno ao composto. O parâmetro é o nome do componente interno a recuperar.

O construtor da classe `Manager` do composto é responsável por instanciar as classes *Façade* do composto; na Figura 3.16, a classe `Manager` do componente `X` instancia a classe `FacadeX` e a expõe como interface provida do componente (linha 6).

```
1  package x.impl;
2  import br.unicamp.ic.sed.cosmos.AManagerComposite;
3  import br.unicamp.ic.sed.cosmos.IManager;
4  class Manager extends AManagerComposite implements IManager {
5    Manager() {
6      this.setProvidedInterface("IX", new FacadeX(this));
7    }
8
9    void setInternalComponent(String name, IManager component) { ... }
10   IManager getInternalComponent(String name) { ... }
11 }
```

Figura 3.16: The class `Manager` of the composite component X in Figure 3.15

3.6.2 Classes Façade em um componente composto

Cada interface provida de um componente composto é implementada por uma classe *Façade*, de maneira semelhante às classes *Façade* de um componente elementar (Seção 3.3.3).

A Figura 3.17 ilustra a classe `FacadeX` do componente composto `X` da Figura 3.15:

- A classe `FacadeX` implementa a interface provida `IX` do composto `X`.
- Por ser uma classe de implementação, a classe `FacadeX` está localizada no pacote `x.impl` e tem visibilidade de pacote.
- O parâmetro do construtor da classe `FacadeX` possibilita a objetos desta classe armazenar referências ao objeto `Manager` do componente (linha 7).
- Para implementar o métodos `mx()` da interface `IX`, a classe `FacadeX` recupera o componente interno `A` através do método `getInternalComponent` da classe `Manager` (linha 9), recupera a interface `IA` (linha 10) e invoca o método `ma()` (linha 11).

3.6.3 Classes Adapter em um componente composto

Classes `Adapter` em um componente composto adaptam interfaces requeridas do composto para interfaces requeridas de componentes internos. Na Figura 3.15, a interface

```
1  package x.impl;
2  import x.spec.prov.IX;
3  import a.spec.prov.IA;
4
5  class FacadeX implements IX {
6    private Manager manager;
7    FacadeX(Manager mgr) { this.manager = mgr; }
8    public void mx() {
9      IManager a = this.manager.getInternalComponent("a");
10     IA ia = (IA) a.getProvidedInterface("IA");
11     ia.ma();
12   }
13 }
```

Figura 3.17: Class FacadeX in composite component X from Figure 3.15

requerida `IBReq` do componente interno `B` é do tipo `b.spec.req.IBReq`, enquanto a interface requerida do composto é do tipo `x.spec.req.IXReq`. A classe `AdapterXReq` está ilustrada na Figura 3.18:

- A classe `AdapterXReq` implementa a interface requerida `IBReq` e usa a interface requerida `IXReq`.
- A classe `AdapterXReq` obtém a referência à classe `Manager` do componente através do construtor (linhas 5 and 6).
- Para implementar o método `mb()` da interface requerida `IBReq`, a classe `AdapterXReq` recupera a interface requerida `IXReq` do composto (linha 8) e invoca nesta o método `mx()` (linha 9).

```
1  package x.impl;
2  import b.spec.req.IBReq;
3  import x.spec.req.IXReq;
4  class AdapterXReq implements IBReq {
5    private Manager manager;
6    AdapterXReq(Manager mgr) { this.manager = mgr; }
7    public String mb() {
8      IXReq ixreq = this.manager.getRequiredInterface("IXReq");
9      return ixreq.mx();
10   }
11 }
```

Figura 3.18: Classe `AdapterXReq` do composto X na Figura 3.15

3.6.4 Classe ComponentFactory em um componente composto

De maneira semelhante à classe `ComponentFactory` de um componente elementar (Seção 3.3.1), a classe `ComponentFactory` de um componente composto tem um método `createInstance()`, responsável por criar uma instância do componente.

A Figura 3.19 ilustra a classe `ComponentFactory` do componente composto `X` da Figura 3.15.

- O método `createInstance()` instancia os componentes internos `A` e `B` e o conector `AB` através das classes `ComponentFactory` destes (linhas 5 a 7). As interfaces dos componentes são conectadas através de chamadas aos métodos `setRequiredInterface` e `getProvidedInterface` (linhas 9 a 10).
- A classe `Manager` do composto é instanciada (linha 12). Cada componente é registrado junto ao `Manager` como interno ao composto através de chamadas ao método `setInternalComponent` (linhas 13 a 15).
- Finalmente, a classe `AdapterXReq` é instanciada com uma referência ao `Manager` do composto, e é conectada à interface requerida do componente interno `B` (linha 17).

```
1  package x.impl;
2  import br.unicamp.ic.sed.cosmos.IManager;
3  public class ComponentFactory {
4    public static IManager createInstance() {
5      IManager a = a.impl.ComponentFactory.createInstance();
6      IManager b = b.impl.ComponentFactory.createInstance();
7      IManager ab = ab.ComponentFactory.createInstance();
8
9      a.setRequiredInterface("IAReq", ab.getProvidedInterface("IAReq"));
10     ab.setRequiredInterface("IB", b.getProvidedInterface("IB"));
11
12     Manager mgr = new Manager();
13     mgr.setInternalComponent("a", a);
14     mgr.setInternalComponent("b", b);
15     mgr.setInternalComponent("ab", ab);
16
17     b.setRequiredInterface("IBReq", new AdapterXReq(mgr));
18
19     return mgr;
20   }
21 }
```

Figura 3.19: Class `ComponentFactory` in composite component X in Figure 3.15

3.7 Componentes de sistema

Um **componente de sistema** é um componente Cosmos* que é um sistema de software completo e pode ser implantado e executado. Um componente Cosmos* que não é um componente de sistema não pode ser executado independentemente; ao invés disso, deve ser instanciado como um componente interno de um componente composto e suas interfaces conectadas às interfaces de outros componentes internos através de conectores. Já um componente de sistema provê um ponto de partida para que ele seja instanciado em um **ambiente de execução**. Ainda, por poder ser implantado e executado, um componente de sistema pode ser considerado uma **configuração arquitetural concreta**, de acordo com a terminologia apresentada em [19].

O ponto de partida de um sistema independente escrito em Java é o método *main*, com a assinatura `public static void main(String[])`. O método *main* também é usado como ponto de partida de sistemas em outras linguagens de programação, como C# e C++. No modelo Cosmos*, a classe `ComponentFactory` de um componente de sistema tem um método *main*, que é responsável por instanciar o componente de sistema e executar uma de suas interfaces. A Figura 3.20 ilustra a classe `ComponentFactory` do componente de sistema `X` da Figura 3.2:

```
1  package x.impl;
2  import br.unicamp.ic.sed.cosmos.IManager;
3  public class ComponentFactory {
4    public static IManager createInstance() {
5          ...
6          /* the method createInstance() is described in Figure 3.19 */
7    }
8
9      public static void main(String args[]) {
10         IManager system = createInstance();
11         IX ix = (IX) system.getProvidedInterface("IX");
12         ix.run(); /* execute the system */
13     }
14 }
```

Figura 3.20: A classe `ComponentFactory` do componente composto X da Figura 3.15

Na Figura 3.20, o método *main* da classe `ComponentFactory` é o ponto de partida do sistema. Este método instancia o componente `X` (linha 10), recupera neste a interface provida `IX` (linha 11) e invoca o método `run()` (linha 12), que executa as funcionalidades do sistema.

O ponto de partida de um sistema depende de seu ambiente de execução. Por exemplo, em um contêiner de servlets na plataforma JavaEE [40], o método *main* é ignorado pelo

contêiner e outros mecanismos são utilizados para instanciar um sistema. O ponto de partida de um componente de sistema Cosmos* deve ser adaptado ao ambiente de execução em que o componente será implantado e executado. A execução de um componente de sistema Cosmos* na plataforma JavaEE é detalhada em [27].

3.8 O pacote `br.unicamp.ic.sed.cosmos`

Componentes Cosmos* oferecem a interface `IManager`, que é definida no pacote `br.unicamp.ic.sed.cosmos`. A Figura 3.21 ilustra este pacote, com a definição da interface `IManager` e das classes `AManager` e `AManagerComposite`. Os pacotes `a.impl`, `b.impl`, `ab` e `x.impl` contêm as classes de implementação dos componentes da Figura 3.15. As classes `Manager` do conector `AB` e dos componentes elementares `A` e `B` estendem a classe `AManager`; a do componente composto `X` estende `AManagerComposite`.

A classe `AManager` provê uma implementação reutilizável dos métodos na interface `IManager` e métodos `setProvidedInterface` e `setProvidedInterfaceType`; a classe `AManagerComposite` estende a classe `Manager` com os métodos `setInternalComponent` e `getInternalComponent`, para compostos.

Como especificado na Seção 3.2.2, somente a interface `IManager` é obrigatória para componentes Cosmos*. As classes `AManager` e `AManagerComposite` facilitam a implementação desta interface; no entanto, este é um detalhe de implementação e não é obrigatório no modelo Cosmos*. O desenvolvedor de um componente Cosmos* está livre para escolher não utilizar estas classes e implementar de outra forma a interface `IManager`.

3.9 Bibliotecas

O modelo de especificação do modelo Cosmos* existe para evitar referências diretas entre componentes e obriga a comunicação entre componentes através de suas interfaces providas e requeridas, de modo a minizar o acoplamento entre componentes (Seção 3.2). No entanto, classes de implementação de um componente Cosmos* podem precisar referenciar diretamente classes de bibliotecas de terceiros, para reutilizar funcionalidades providas por tais bibliotecas. Tais referências diretas qualificam um forte acoplamento entre o componente e a biblioteca referenciada, e é uma dependência que não pode ser explicitada na forma de uma interface requerida. Assim, para ser utilizada por um componente Cosmos*, uma biblioteca deve estar disponível durante a execução do sistema.

Um componente Cosmos* também pode referenciar diretamente outro componente Cosmos*. Isto acontece em componentes compostos (Seção 3.6), quando o composto referencia diretamente as classes `ComponentFactory` de seus componentes internos. Assim,

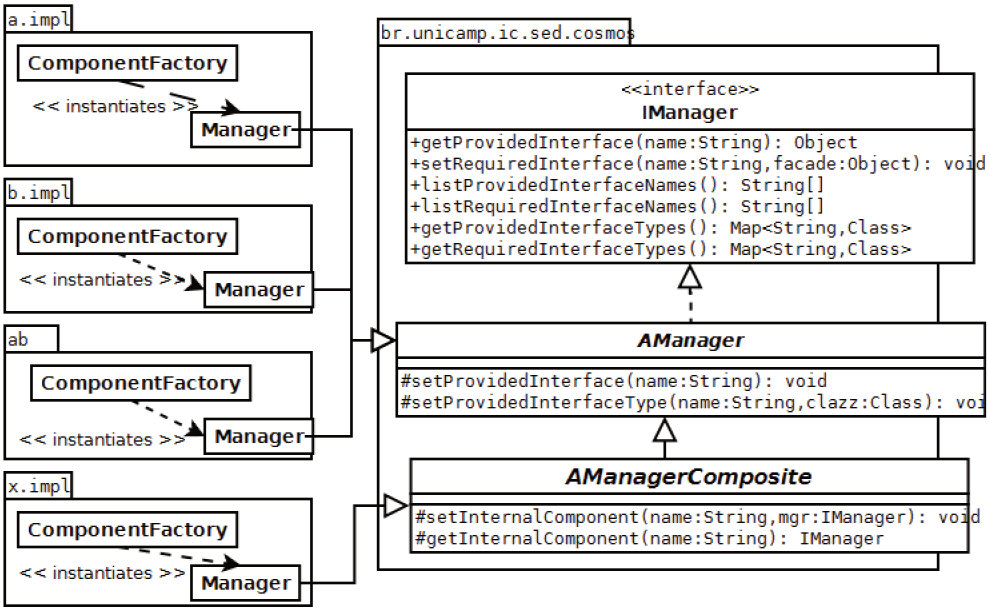


Figura 3.21: A interface `IManager` no pacote `br.unicamp.ic.sed.cosmos`

existe um forte aoplamento entre o composto e seus componentes internos.

Na plataforma Java, disponibilizar uma biblioteca para um programa Java significa adicionar a biblioteca à variável *Class-Path*, reconhecida pela máquina virtual Java (JVM). O *Class-Path* de um programa Java é a lista de arquivos e diretórios nos quais a JVM vai procurar por uma classe quando ela é referenciada, durante a execução do programa [37].

Um componente de software pode ter dependências que não podem ser representadas como interfaces requeridas [44]. Tipicamente, estas dependências são descritas em documentação que acompanha o componente. Estas dependências se dividem em dois tipos: dependências do ambiente de execução e bibliotecas externas.

3.10 Dependências do ambiente de execução

Um componente pode depender de determinadas condições do ambiente em que é executado para a realização de suas funcionalidades. São exemplos deste tipo de dependências:

- **Árvores de diretórios e arquivos.** Um componente pode operar sobre arquivos no sistema operacional e pode presumir a existência de determinados arquivos ou estrutura de diretórios. A estruturação correta de diretórios esperada pelo componente é tipicamente declarada na documentação do componente.

- **Esquema de tabelas em banco de dados.** Um componente pode acessar um

banco de dados relacional e presumir a existência de determinado modelo de entidade e relacionamento no banco de modo a enviar consultas SQL. O esquema correto deve ser descrito na documentação do componente.

O modelo Cosmos* não oferece suporte à declaração de tais dependências em tempo de execução. Elas são tipicamente declaradas na documentação do componente.

3.11 Tipos de dados

As operações nas interfaces dos componentes podem apresentar, em suas assinaturas, tipos de dados que não estão definidos no ambiente de execução. Por exemplo, um componente para matrícula de alunos em um sistema acadêmico pode oferecer uma operação `registrarAluno(Aluno)`. Esta operação recebe o `Aluno`. Tipos de dados podem aparecer na assinatura de uma operação como um parâmetro ou valor de retorno.

Há duas abordagens para a definição de tipos de dados, definidos por Silva Júnior em [16]: na primeira, existe um pacote global com as definições de tipos de dados (Seção 3.11.1); na segunda, cada componente Cosmos* define os tipos de dados usados em suas operações (Seção 3.11.2).

3.11.1 Pacote global com definição de tipos de dados

Nesta abordagem, existe um pacote global que contém as definições de todos os tipos de dados, compartilhada por todos os componentes. Neste pacote global, os tipos de dados aparecem como interfaces, e cada componente contém implementações dos tipos de dados que usa.

A Figura 3.22 ilustra dois componentes, `SistemaUniversidade` e `MatriculaAlunos`, usados em um sistema para matrícula de alunos em uma universidade.

Ambos os componentes utilizam o tipo de dados `IAluno`, definido no pacote `datatypes` e que representa uma matrícula de aluno no banco de dados da universidade. Assim, ambos os componentes tem uma dependência no pacote `datatypes`. Esta abordagem tem a vantagem de que instâncias de `IAluno` podem transitar entre componentes sem a necessidade de adaptação.

A desvantagem desta abordagem é que o pacote de dados global se torna uma dependência externa de cada componente (Seção 3.9).

Além disso, tipos de dados definidos no pacote `datatypes` são interfaces, e cada componente que faz uso destes tipos de dados deve implementar estas interfaces em seus pacotes `impl`. Assim, na Figura 3.22, o pacote `universidade.impl` e `matriculaAlunos.impl` contêm classes `AlunoImpl`, que implementam a interface `IAluno`.

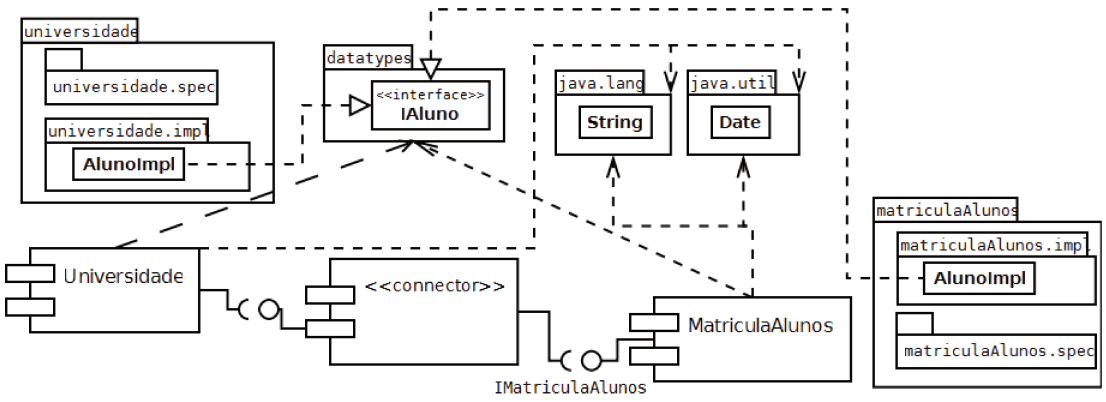


Figura 3.22: Tipos de dados definidos em um pacote global.

Convém enfatizar que a dependência por um pacote global de dados já acontece com a escolha da plataforma de programação. Por exemplo, um componente Cosmos* programado em Java pode referenciar as classes `java.lang.String` e `java.util.Date`, que estão disponíveis na plataforma Java.

A dependência por um pacote global com tipos de dados é um caso de dependência por biblioteca de terceiros: as classes do pacote global devem ser disponibilizadas para o programa. Por exemplo, na plataforma Java, devem ser incluidas na variável *Class-Path*.

3.11.2 Tipos de dados definidos pelo componente

Nesta abordagem, a especificação de um componente é estendida e define os tipos de dados usados pelo componente. Cada componente define um pacote `spec.datatypes` que contém os tipos de dados usados pelo componente. Também nesta abordagem, os tipos de dados são interfaces.

Na Figura 3.23, os componentes `Universidade` e `MatriculaAlunos` definem interfaces `IAluno` em seus pacotes `spec.datatypes` e contêm classes `AlunoImpl` em seus pacotes de implementação e que implementam estas interfaces. Assim, um componente não tem dependências por pacotes de dados globais. O conector contém as classes `AlunoImplUni` (a definição de `IAluno` do componente `Universidade` e `AlunoImplMatr` (definição de `IAluno` do componente `MatriculaAlunos`. Estas classes implementam os tipos de dados de cada componente; o conector as usa para converter dados vindos de um componente no tipo de dados esperado pelo outro componente.

A vantagem desta abordagem é que um componente é autocontido e não apresenta dependências externas para tipos de dados globais. A desvantagem é que o desenvolvimento

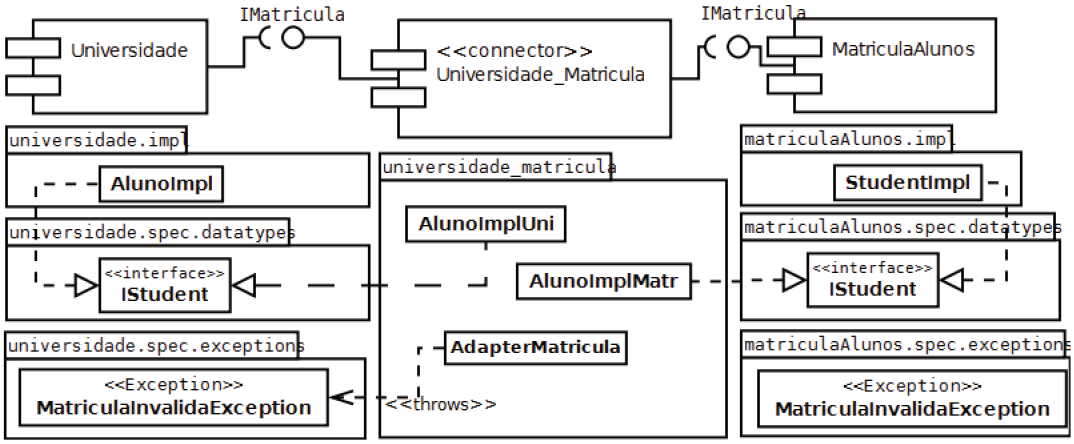


Figura 3.23: Tipos de dados definidos por um componente.

de conectores é mais trabalhoso, pois é preciso adaptar tipos de dados entre diferentes componentes.

3.11.3 Exceções

Uma operação em uma interface de um componente pode declarar que lança uma exceção. Assim, os tipos das classes de exceções transitam entre as interfaces de componentes. Assim como tipos de dados, exceções podem ser declaradas no pacote de dados global ou definidas pelo componente; neste último caso, são declaradas no pacote `spec.exceptions`.

Na Figura 3.23, as operações nas interfaces requeridas do componente `Universidade` apresentam a exceção `MatriculaInvalidaException`, que indica que o componente está preparado para tratar exceções em operações de matrícula. A classe `MatriculaInvalidaException` é definida no pacote `universidade.spec.exceptions`.

Na plataforma Java, classes de exceções não implementam interfaces. Assim, um conector pode referenciar diretamente a classe de exceção e não precisa conter classes que implementam a exceção, como é feito com outros tipos de dados.

O tratamento de exceções com o modelo Cosmos* é detalhado em [27].

Capítulo 4

A Ferramenta CosmosLoader

A ferramenta CosmosLoader foi desenvolvida com o objetivo de auxiliar o desenvolvedor responsável pela criação de um sistema a partir de implementações existentes de componentes de software; a ferramenta CosmosLoader foi desenvolvida com alvo na criação de sistemas baseados no modelo Cosmos* e na plataforma Java.

A Seção 4.1 apresenta as motivações para o desenvolvimento do CosmosLoader. A Seção 4.3 explica o mecanismo de carregamento de classes na plataforma Java, que é a base do CosmosLoader. A Seção 4.4 detalha o metamodelo utilizado para representar uma arquitetura interna de um sistema baseado no modelo COSMOS*, e que é utilizado pelo CosmosLoader para carregar os componentes do sistema.

4.1 Motivações para a criação da ferramenta CosmosLoader

Existem duas motivações principais para o desenvolvimento do CosmosLoader: (1) automatizar a integração de componentes Cosmos* em um sistema de software (Seção 4.1.1) e (2) controlar as versões dos componentes que compõem uma configuração arquitetural específica (Seção 4.1.2).

4.1.1 Automatização da integração de componentes

No modelo COSMOS*, a implementação de uma arquitetura de componentes é realizada através da implementação de um componente COSMOS* para o sistema, e, recursivamente, de um componente COSMOS* para cada componente arquitetural dentro do componente de sistema (Seção 3.6).

De acordo com este modelo de composição, um composto instancia seus subcomponentes através de chamadas aos métodos `createInstance` na classe

`ComponentFactory` de cada subcomponente, e conectando as interfaces através de chamadas aos métodos
`getRequiredInterface` e `setProvidedInterface` na interface `IManager` de cada componente.

Assim, existe um hiato entre a arquitetura de um sistema baseado em componentes COSMOS*, que é realizada em linguagens visuais, como a linguagem UML, e a integração, que é realizada em uma linguagem de programação, como Java. Este processo é propenso a erros, porque não há garantia de que os nomes dos componentes e das interfaces estão de acordo com o especificado na arquitetura.

Por exemplo, ocorre um erro se o desenvolvedor não utilizar na integração do sistema os nomes de componentes e interfaces estabelecidos na especificação do mesmo. Esta situação está ilustrada para o componente de sistema `X` nas Figuras 4.1 e 4.2: na Figura 4.1, as interfaces requeridas `IAReq` e `IAReq2` do componente `A` estão conectadas às interfaces `IB1` e `IB2` do componente `B`; no entanto, na Figura 4.2, por um descuido, a interface `IB2` foi conectada às duas interface requeridas.

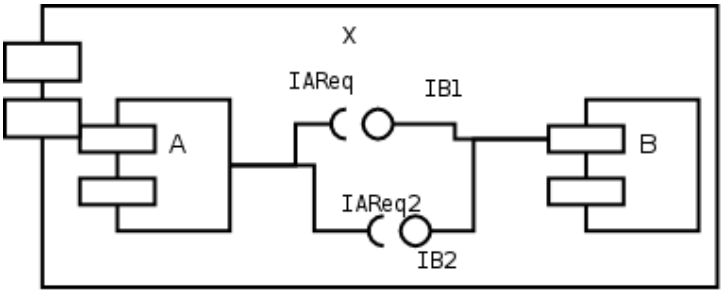


Figura 4.1: Um componente de sistema

Se o código da Figura 4.2 for executado, o sistema consistirá na combinação de componentes e interfaces ilustrada na Figura 4.3, diferente da arquitetura planejada, ilustrada na Figura 4.1.

A motivação para o desenvolvimento do CosmosLoader também é inspirada em outros modelos de componentes, como o modelo JavaBeans (Seção 2.3.1). Assim como o modelo COSMOS*, o modelo JavaBeans é baseado na premissa de que as propriedades externamente visíveis dos componentes (os métodos de um objeto JavaBean) obedecem a convenções de nomes (métodos *get* e *set*). Assim, algumas ferramentas, como a ferramenta Bean Builder [18] possibilitam a integração de componentes JavaBeans de maneira visual, eliminando a necessidade do programador de escrever o código que faz a ligação dos componentes.

Assim, um dos objetivos do CosmosLoader é realizar a integração de componentes Cosmos* de maneira automatizada a partir de uma descrição visual do sistema, e sem a escrita do código de instanciação do sistema.

```
1  package x.impl;
2
3  public class ComponentFactory {
4   public static IManager createInstance() {
5     Manager x = new Manager();
6     IManager a = a.impl.ComponentFactory.createInstance();
7     IManager b = b.impl.ComponentFactory.createInstance();
8
9     a.setRequiredInterface("IAReq", b.getProvidedInterface("IB2"));
10    a.setRequiredInterface("IAReq2",b.getProvidedInterface("IB2"));
11    x.setInternalComponent("a", a);
12    x.setInternalComponent("b", b);
13    return x;
14  }
15 }
```

Figura 4.2: Uma implementação errônea do componente X da Figura 4.1

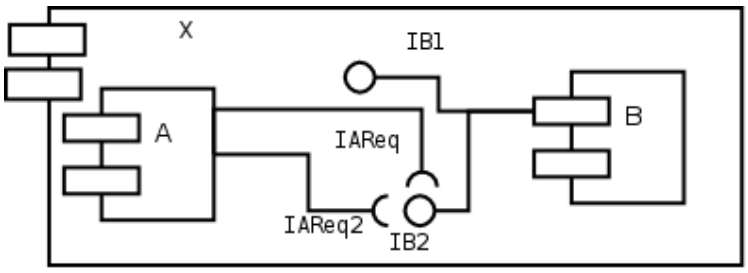


Figura 4.3: Um componente de sistema

4.1.2 Gerenciamento de configurações arquiteturais

A versão dos componentes de um sistema é uma característica importante no desenvolvimento de sistemas baseados em componentes: o comportamento de um sistema varia de acordo com as versões de seus componentes.

O controle de versão é uma prática estabelecida no desenvolvimento de sistemas de software, e se apoia no uso de ferramentas de controle de versão, como o Subversion [46] ou o CVS. Estas ferramentas possibilitam rastrear mudanças entre diferentes versões no desenvolvimento de um componente de software.

No entanto, tais ferramentas permitem controlar o versionamento de componentes de software individuais, e não são voltadas para a integração de sistemas a partir de componentes independentes.

Ainda, é desejável poder incluir em um sistema diferentes versões de um mesmo componente. Isto possibilita, por exemplo, aumentar a robustez do sistema contra erros de programação.

Outro ponto a observar são as dependências dos componentes. As implementações dos componentes podem depender de bibliotecas desenvolvidas por terceiros. Também é possível que diferentes componentes em um sistema dependam de uma mesma biblioteca, mas em versões diferentes.

Cada plataforma de programação deve apresentar uma solução para a presença de diferentes versões de componentes na execução de um sistema. Por exemplo, duas versões de um mesmo componente podem conter classes com o mesmo nome; assim, é necessário desenvolver técnicas que permitam diferenciar as diferentes versões da classe durante a execução do programa. A plataforma Java resolve este problema com o mecanismo de carregamento de classes (*class loaders*, Seção 4.3).

Assim, outra motivação para a ferramenta CosmosLoader é explicitar as versões dos componentes de software que compõem uma configuração arquitetural de um sistema, e permitir a presença de diferentes versões de componentes e bibliotecas durante sua execução.

4.2 Uso da ferramenta CosmosLoader

O CosmosLoader é responsável por instanciar os componentes de um sistema. Para utilizá-lo, o programador inclui um objeto `CosmosLoader` no método `createInstance` da classe `ComponentFactory` do componente de sistema da sua aplicação, de acordo com o modelo Cosmos* (Seção 3.3.1). Ainda, o programador deve fornecer a este objeto `CosmosLoader` um arquivo que descreve o sistema que ele deseja instanciar: é um arquivo XML que utiliza o metamodelo do CosmosLoader para descrever a configuração concreta do sistema (Seção

4.4), ou, simplesmente, um arquivo *cosmosloader*. O arquivo *cosmosloader* é criado em um editor visual específico, como será explicado na Seção 4.5.

A Figura 4.4 ilustra o código de instanciação do componente de sistema `X` da Figura 4.1, modificado para utilizar o CosmosLoader.

```
1  package x.impl;
2
3  public class ComponentFactory {
4      public static IManager createInstance() {
5          String resourceName = "X.cosmosloader";
6          InputStream stream = this.getClass().getClassLoader()
7                                   .getResourceAsStream(resourceName);
8          CosmosLoader loader = new CosmosLoader("/home/user/system");
9          IManager x = loader.instantiateConfiguration( stream );
10         return x;
11     }
12 }
```

Figura 4.4: Instanciação de um sistema através do CosmosLoader

Na Figura 4.4, nas linha 5 a 7 o arquivo de descrição da configuração do componente X é recuperado; na linha 8, é instanciado um objeto CosmosLoader, responsável por instanciar os componentes internos do componente de sistema e conectar as interfaces destes; o parâmetro para o construtor é o diretório, no sistema de arquivos, que contém as implementações dos componentes; na linha 9, o objeto CosmosLoader é instruido a instanciar o componente de sistema `X`.

Observa-se na Figura 4.4 que o código de instanciação do sistema `X` não contém chamadas aos métodos `createInstance` dos componentes internos, e nem realiza as conexões entre as interfaces. Isto é feito pelo objeto CosmosLoader, a partir da descrição do sistema, que é lida do arquivo `X.cosmosloader` (linha 5).

4.2.1 Um repositório simples de componentes

As implementações dos componentes e as bibliotecas utilizadas por eles devem ser disponibilizadas em um repositório de componentes; é um diretório que contém arquivos Jar cujos nomes estão no formato *`nome do componente_versão`*`.jar`. Em tempo de execução, para encontrar os arquivos da implementação de um componente ou de uma biblitoca externa, o CosmosLoader compõe o nome de um arquivo Jar concatenando o nome do componente (ou da biblioteca) e a versão do mesmo e procura o arquivo com este nome no repositório de componentes especificado pelo desenvolvedor. Por exemplo, as classes de implementação da versão 1.0.0 do componente X da Figura 4.4 devem estar no arquivo `x_1.0.0.jar` no diretório `/home/user/system`.

4.3 Carregamento de classes na plataforma Java

Diferentemente de programas desenvolvidos em linguagens compiladas para código de máquina, como a linguagem C, na qual, por padrão, todo o código-fonte da aplicação é compilado para instruções de máquina em um único arquivo (o programa executável) e que é carregado de uma única vez para a memória, na linguagem Java o código-fonte de cada classe é compilado para instruções da máquina virtual Java (JVM), chamadas Java *Bytecodes*, e que são armazenados em um arquivo para cada classe do código-fonte.

Com isto, o código de uma classe não é carregado para a memória assim que o programa é iniciado, e não está imediatamente disponível para execução na JVM. Antes de executar o código de uma classee, a JVM realiza as seguintes operações [36]:

- O *carregamento* de uma classe (*class loading*) é o processo de encontrar os *Bytecodes* de uma classe. Usualmente, este processo consiste simplemente em ler de um arquivo ou receber de uma máquina remota os *Bytecodes* da classe gerados por um compilador.

- A *ligação* de uma classe é o processo pelo qual a JVM verifica os *Bytecodes* encontrados anteriormente e os prepara para serem utilizados. Este processo envolve, por exemplo, verificar que os bytes recebidos correspondem a instruções válidas da máquina virtual.

- A *inicialização* de uma classe consiste em executar qualquer código que deve ser executado assim que a classe é carregada, de acordo com a linguagem Java. Por exemplo, a linguagem especifica que atributos de uma classe são inicializados com o valor zero, se forem tipos primitivos, ou com *null*, se forem referências a objetos. É nesta etapa que os atributos da classe são modificados para seus valores iniciais de acordo com o código-fonte da classe.

As etapas de ligação e inicialização são operações internas da JVM. Por outro lado, o mecanismo de carregamento de classes é exposto para o programador através da classe `java.lang.ClassLoader`.

4.3.1 Carregadores de classes (*Class loaders*)

Na plataforma Java, carregadores de classes (*class loaders*) permitem que um programador modifique o mecanismo de carregamento de classes da JVM. Para isso, o programador deve criar uma sub-classe de `java.lang.ClassLoader`.

Todo objeto de uma aplicação Java em execução tem um metaobjeto da classe `java.lang.Class` associado a ele e que é uma representação da classe de um objeto;

um objeto `ClassLoader` está associado ao objeto `Class` do objeto cujo código está sendo executado. Neste texto, o termo ClassLoader referencia o objeto da classe `ClassLoader` associado à classe do objeto em execução.

À medida que o código de uma classe é executado e quando uma classe é referenciada pela primeira vez, o fluxo de execução do programa é interrompido, e o método **loadClass(String)** do objeto `ClassLoader` da aplicação é invocado, para que o `ClassLoader` encontre a classe referenciada. O parâmetro do método corresponde ao nome da classe referenciada.

A classe `ClassLoader` oferece os seguintes métodos relacionados ao carregamento de classes:

- `loadClass(String): Class`. Este método é invocado pelo ClassLoader quando uma classe é referenciada pela primeira vez pelo código em execução. Este método deve (1) encontrar o *bytecode* da classe referenciada e (2) invocar o método `defineClass`, que realiza as etapas de ligação e inicialização da classe.

  Por fim, este método retorna um objeto `Class`, que é o resultado das etapas de ligação e inicialização; ou ainda, este método lança uma exceção do tipo `ClassNotFoundException`, que indica que a classe requisitada não foi encontrada.

- `defineClass(String, byte[], int, int): Class`. Este método é invocado durante a execução do método `loadClass`, e é responsável pelas etapas de ligação e inicialização da classe a partir da sequência de bytes encontrada na etapa anterior, e que compõe a classe.

  Os parâmetros para este método são o nome da classe e os bytes que a definem. Há ainda dois inteiros como parâmetros, que são índices para o ínicio e o final do vetor de bytes e permitem delimitar uma subsequência dos bytes do array.

  Diferentemente do método `loadClass`, este método é final e não pode ser sobreescrito.

A classe `ClassLoader` padrão da plataforma Java segue um modelo de delegação, como ilustrado na Figura 4.5. Um objeto `ClassLoader` tem um pai (parent), para o qual ele delega a procura por uma classe. Somente se o ClassLoader pai não encontrar a classe (lançando `ClassNotFoundException`), ele procura pela classe, invocando o método `findClass`, que procura e retorna a classe procurada, ou lança `ClassNotFoundException`.

Carregadores de classes do CosmosLoader

O CosmosLoader contém as classes `ComponentLoader`, `InterfaceLoader`, `ExternalLoader` e `DatatypeLoader`, derivadas da classe `java.lang.ClassLoader`, de acordo com a Figura 4.6:

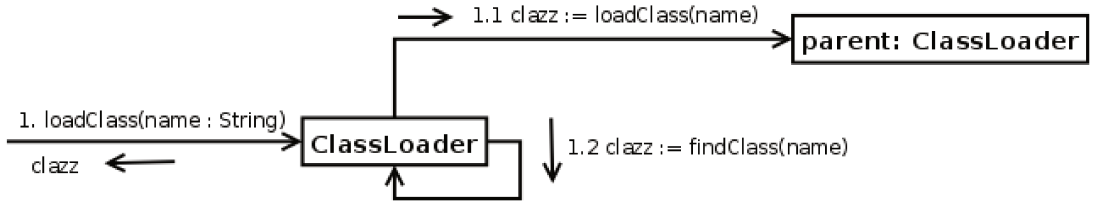


Figura 4.5: O mecanismo de delegação da classe ClassLoader padrão da plataforma Java.

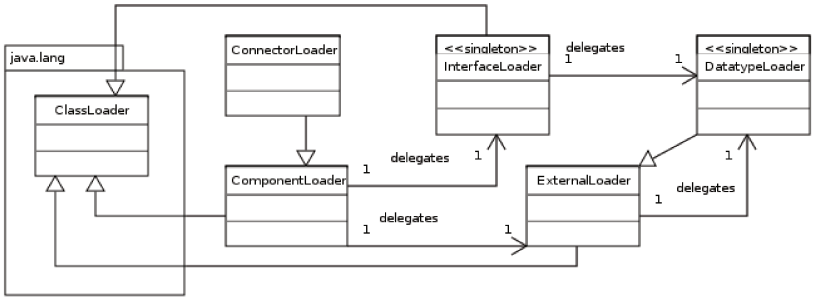


Figura 4.6: Carregadores de classes do CosmosLoader

- A classe `ComponentLoader` é responsável por instanciar componentes COSMOS*, e carregar suas classes de implementação durante a execução do sistema. Para cada componente no sistema, existe uma instância da classe `ComponentLoader`; isto permite isolar em tempo de execução uma instância de componente dos demais componentes do sistema.

- A classe `InterfaceLoader` é responsável por carregar as interfaces dos componentes. Esta classe segue o padrão *singleton* [25], porque a definição de uma interface deve ser compartilhada por diferentes componentes, de modo que todas as classes que representam interfaces de componentes devem ser carregadas pelo mesmo *class loader*. Isto é compatível com a definição de interfaces adotada em processos de DBC (Seção 2.2). Assim, as classes que representam interfaces de componentes devem ser carregadas uma única vez.

  Este compartilhamento de interfaces impede que haja diferentes versões de uma interface no sistema. Assim, o CosmosLoader impõe a restrição de que interfaces devem ser imutáveis ao longo das diferentes versões de um componentes, e que mudanças em uma interface são realizadas através de novas versões da mesma.

- A classe `ExternalLoader` é responsável por carregar as classes das bibliotecas externas utilizadas por um componente COSMOS*; assim como os objetos `ComponentLoader`, existe uma instância de `ExternalLoader` para cada componente

no sistema; isto permite isolar as bibliotecas utiizadas por um componente, e permite que diferentes componentes utilizem diferentes versões de bibliotecas.

- A classe `DatatypeLoader` é responsável por carregar as classes dos tipos de dados que trafegam entre as interfaces de componentes. Ela também segue o padrão *singleton*; assim como a definição das interfaces, a definição de tipos de dados deve ser única entre diferentes componentes, para evitar incompatibilidades entre os tipos de dados presentes nas interfaces de diferentes componentes.

4.4 O metamodelo do CosmosLoader

O metamodelo do CosmosLoader captura elementos de DBC e de Arquitetura de Software, que permitem descrever os componentes de um sistema baseado no modelo COSMOS*. O metamodelo do CosmosLoader é baseado no meta-modelo utilizado pelo Bellatrix [47], que por sua vez é baseado no modelo descrito por Guerra et al. em [19].

O metamodelo do CosmosLoader é apresentado na Figura 4.7; as classes do metamodelo estão descritas a seguir. A Figura 4.8 ilustra um componente composto `X`, que será descrito pelo metamodelo.

- **Configuration**: representa a configuração de um sistema; uma configuração descreve um componente composto, as especificações de seus componentes e conectores internos e as implementações de componentes que tomam os lugares das especificações durante a execução do sistema.

  Os atributos **name** e **version** descrevem o nome e a versão da especificação do componente de sistema; por exemplo, o componente `X` obedece à especificação X, versão 1.0.0, de forma que os atributos deste objeto Configuration são `name: X`, `version: 1.0.0`.

- **Architecture**: descreve a arquitetura interna de um componente composto; é um agrupamento das especificações dos componentes (`internal-components`) e conectores (`internal-connectors)` internos ao composto, de conexões entre as interfaces destes (`interface-connections`) e da especificação do composto (`toplevel`).

- **ComponentSpec**: descreve uma especificação de um componente interno ao componente de sistema. O atributo `id` identifica o componente entre os demais componentes internos. Os atributos `name` e `version` descrevem o nome e a versão da especificação do componente.

  Uma especificação de componente agrupa descrições das interfaces providas (`provided-interfaces`) e requeridas do componente (`required-interfaces`).

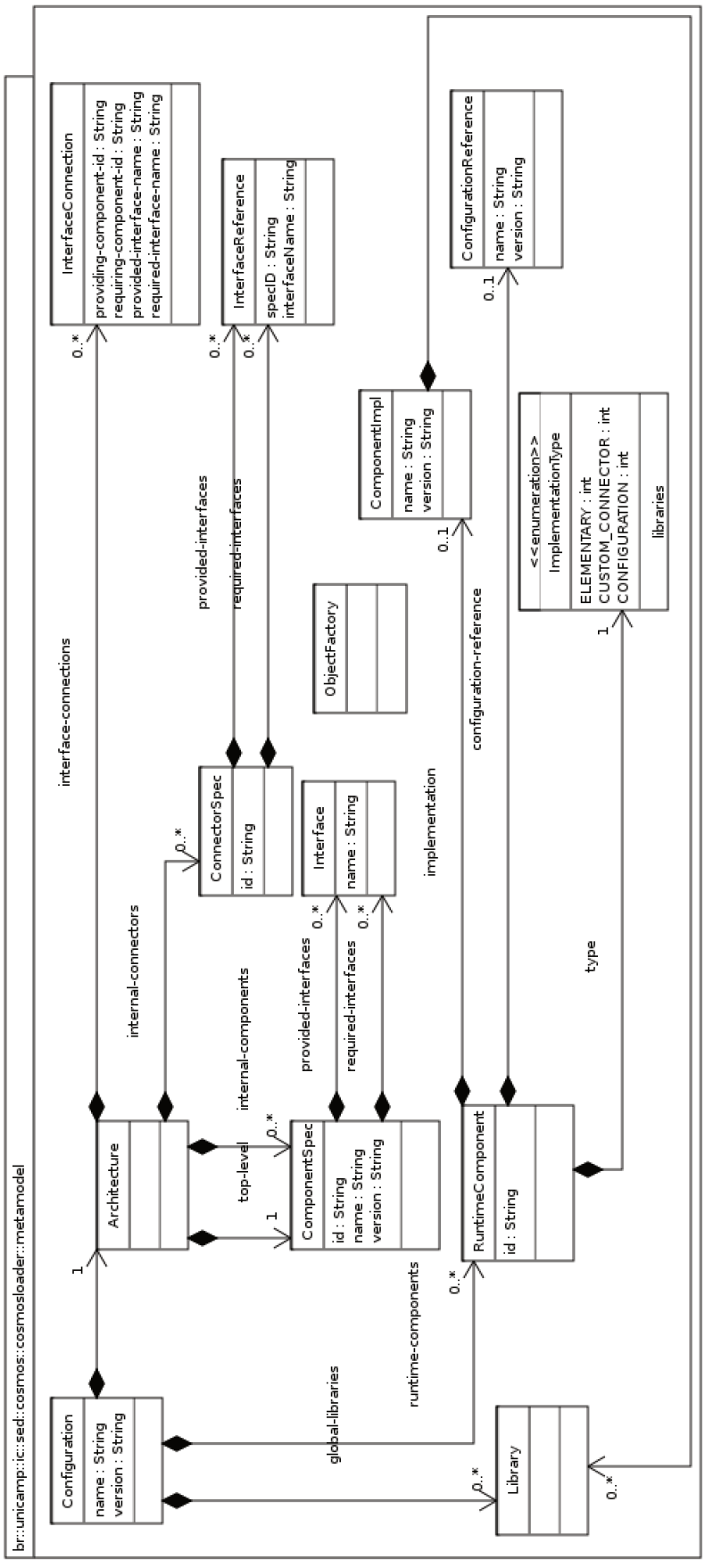


Figura 4.7: O metamodelo do CosmosLoader

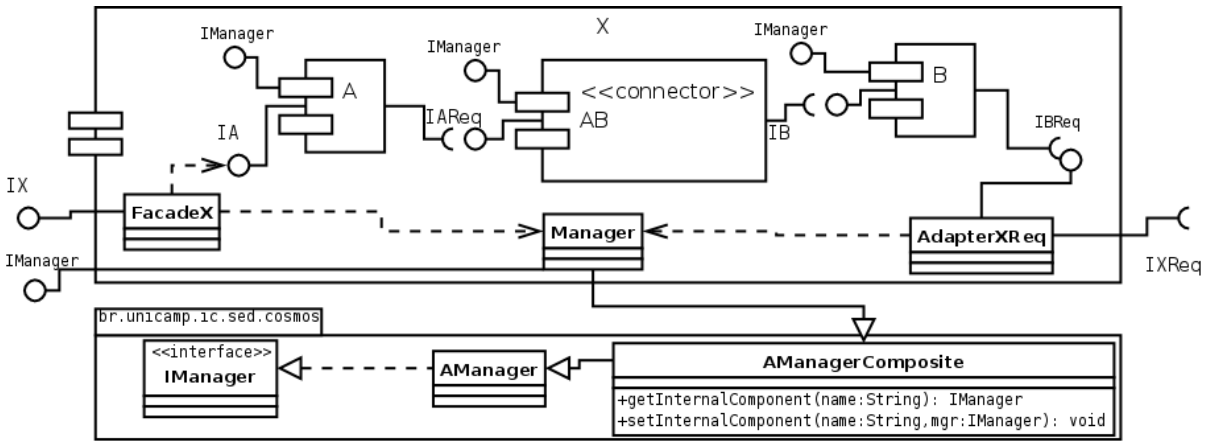


Figura 4.8: Um componente composto X

O componente `X` da Figura 4.8 é descrito por um objeto ComponentSpec com atributos `id: x`, `name: X` e `version: 1.0.0`. Este é o componente "top-level" do objeto Architecture da mesma configuração.

- **Interface**: é uma descrição de uma interface. Somente o nome simples da interface (que não começa com o pacote) é descrito, não há descrição das operações.

- **ConnectorSpec**: descreve uma especificação de um conector interno ao componente de sistema. Esta classe contém somente o atributo `id`, que identifica um conector internamente. Uma especificação de conector não tem nome e versão.

  Assim como um componente, um conector tem interfaces providas (`provided-interfaces`) e requeridas (`required-interfaces`). Um conector não descreve suas interfaces; ao invés disso, contém referências a interfaces descritas pelas especificações dos componentes internos, de acordo com o modelo COSMOS*.

- **InterfaceReference**: objetos desta classe são referências às interfaces descritas pelos componentes. Os atributos `specID` e `interfaceName` são o id do componente interno e o nome da interface.

  O componente `AB` da Figura 4.8 tem uma interface provida com nome `IAReq`; a lista de interfaces providas do conector contém um objeto **InterfaceReference** com atributos `specID: a`, `interfaceName: IAReq`.

- **InterfaceConnection**: descreve uma conexão entre uma interface provida e uma interface requerida de componentes e conectores internos a um componente de sistema.

A arquitetura do componente de sistema contém duas conexões de interface. A conexão entre a interface requerida `IAReq` do componente `A` e a interface provida `IAReq` do conector `AB` é descrita por um objeto `InterfaceConnection` com atributos `requiring-component-id:a`, `required-interface-name:IAReq`, `providing-component-id:ab`, `provided-interface-name:IAReq`.

- **RuntimeComponent**: descreve uma instância de componente Cosmos* que preenche uma especificação de componente. Existe um objeto `RuntimeComponent` para o componente composto que representa o sistema e um para cada componente interno ao composto.

  O atributo `id` de um objeto `RuntimeComponent` identifica a especificação de componente interno (`ComponentSpec`) que a instância de componente Cosmos* implementa, durante a execução do sistema.

  A maneira como o CosmosLoader instancia um componente durante a execução do sistema depende do atributo `type` deste objeto. (Veja **ImplementationType**).

  Se uma instância de componente for do tipo `ELEMENTARY` ou `CUSTOM_CONNECTOR`, então o objeto RuntimeComponent terá um objeto `ComponentImpl` e sua referência `configuration-reference` será nula; se a instância do componente for do tipo ConfigurationReference, é o contrário: a referência à implementação é nula e existe objeto na referência `configuration-reference`.

- **ImplementationType**: é uma enumeração dos diferentes tipos de implementação que podem realizar um componente durante a execução do sistema.

  Uma instância de componente pode ser um componente elementar (`ELEMENTARY`), um componente composto que obedece a uma configuração (`CONFIGURATION`) ou um conector entre dois componentes (`CUSTOM_CONNECTOR`).

- **Library**: descreve uma biblioteca externa a um componente (Seção 3.9). Os atributos `name` e `version` devem seguir o padrão esperado pelo CosmosLoader e que cuja combinação é idêntica ao nome do arquivo que contém a biblioteca e que está disponível em um repositório (Seção 4.2.1).

  Por exemplo, se um componente usa a biblioteca `mysql-jdbc` na versão `4.1`, então os atributos contém os valores `name: mysql-jdbc` e `version: 4.1`.

  Bibliotecas locais são descritas na lista `libraries` de um objeto `ComponentImpl`; bibliotecas globais são descritas na lista `global-libraries` do objeto Configuration que representa a configuração do sistema.

- **ComponentImpl**: indica um componente elementar ou um conector cujo código será carregado de um arquivo JAR no repositório, em contraste com um componente composto.

  De maneira semelhante a descrições de bibliotecas, os atributos para nome e versão nesta classe devem representar arquivos JAR que podem ser encontrados no repositório (Seção 4.2.1). Por convenção, o nome do componente deve conter o `namespace` completo (por exemplo, `br.unicamp.ic.sistema.persistencia` e o nome de bibliotecas contém a forma mais curta (`mysql-jdbc`).

- **ConfigurationReference**: descreve uma referência à configuração de um componente composto. Quando um componente composto contém um componente interno que é um composto, este objeto indica a referência à configuração.

  Os atributos *name* e *version* são os mesmos que para uma implementação de componente; o nome está na forma completa, com o *namespace*, como para uma implementação de componente.

A biblioteca JAXB

A biblioteca JAXB [39] (*Java API for XML Binding*) permite serializar um grafo de objetos em uma representação em XML, e, inversamente, ler uma representação em XML para um grafo de objetos Java em memória. Isto pode ser utilizado para realizar automaticamente a persistência dos dados de uma aplicação no formato XML.

A implementação do metamodelo do CosmosLoader foi criada utilizando a biblioteca JAXB, na versão 2.1.5. Depois que o metamodelo foi estabelecido, foi gerado um esquema XSD que representa as classes e relacionamentos do metamodelo. Em seguida, foi utilizada a biblioteca JAXB para gerar automaticamente as classes do metamodelo.

4.5 Alterações no editor de configurações do Bellatrix

Conforme mencionado na Seção 4.2, o CosmosLoader instancia um sistema baseado em componentes a partir de uma descrição da sua configuração concreta. Na Figura 4.4, esta descrição está no arquivo `X.cosmosloader`. Assim, se faz necessária uma ferramenta que crie a descrição da configuração concreta de um sistema a partir da descrição da sua arquitetura em UML (Seção 4.1.1).

O ambiente Bellatrix apresenta editores gráficos para a especificação dos diferentes artefatos relacionados a DBC durante as etapas do desenvolvimento de um sistema seguindo os princípios de DBC e Arquitetura de Software. Dentre estes editores, o editor

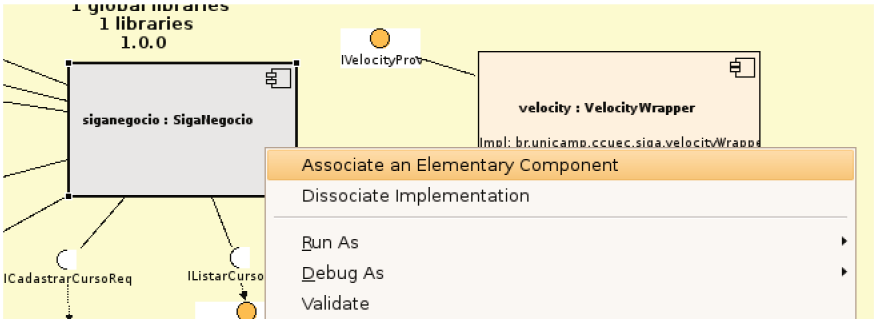


Figura 4.9: Associação de uma implementação de componente a um componente arquitetural.

de configurações permite especificar a configuração concreta de um sistema, relacionando componentes concretos aos componentes arquiteturais de um sistema.

O editor de configurações do Bellatrix foi então escolhido para gerar o arquivo de descrição da configuração concreta de um sistema que será fornecido ao CosmosLoader. Foram realizadas alterações no editor de configurações no sentido de permitir a descrição da configuração concreta do sistema e a geração de um arquivo cosmosloader. As Figuras 4.9 a 4.13 são capturas de telas do editor de configurações do Bellatrix, com as alterações. A alterações são descritas a seguir:

1. **Associação de implementações a componentes arquiteturais:** foi adicionada a opção *"Associate elementary component"* ao editor de configurações (Figura 4.9); esta opção apresenta uma caixa de diálogo que permite ao desenvolvedor inserir o nome e a versão da implementação que assumirá o lugar de um componente arquitetural durante a execução do sistema (Figura 4.10).

2. **Associação de bibliotecas externas locais e globais:** foi adicionada a opção *"Edit libraries"* (Figura 4.11). Esta opção apresenta uma caixa de diálogo com a lista de bibliotecas locais que será disponibilizada para cada implementação de componente durante a execução do sistema (Figura 4.12).

   Também foi adicionada a opção *"Edit Global libraries"*, que apresenta uma caixa de diálogo que permite editar a lista de bibliotecas globais, que serão disponibilizadas para todos os componentes durante a execução do sistema.

3. **Geração do arquivo cosmosloader:** foi adicionada a opção *"Export as CosmosLoader configuration"* (Figura 4.13), que gera um arquivo cosmosloader a partir da descrição da configuração concreta do sistema.

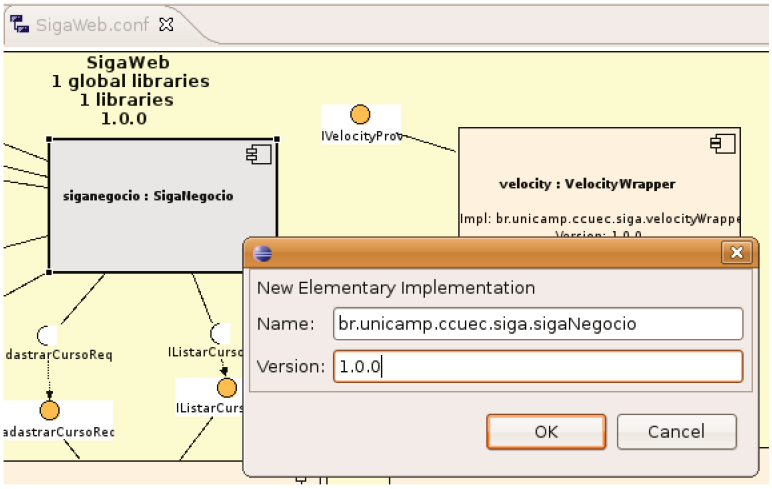


Figura 4.10: Nome e versão de uma implementação.

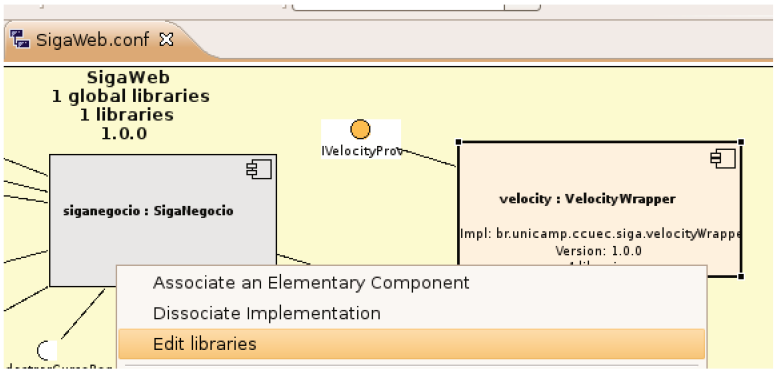


Figura 4.11: Opção para editar a lista de bibliotecas de um componente.

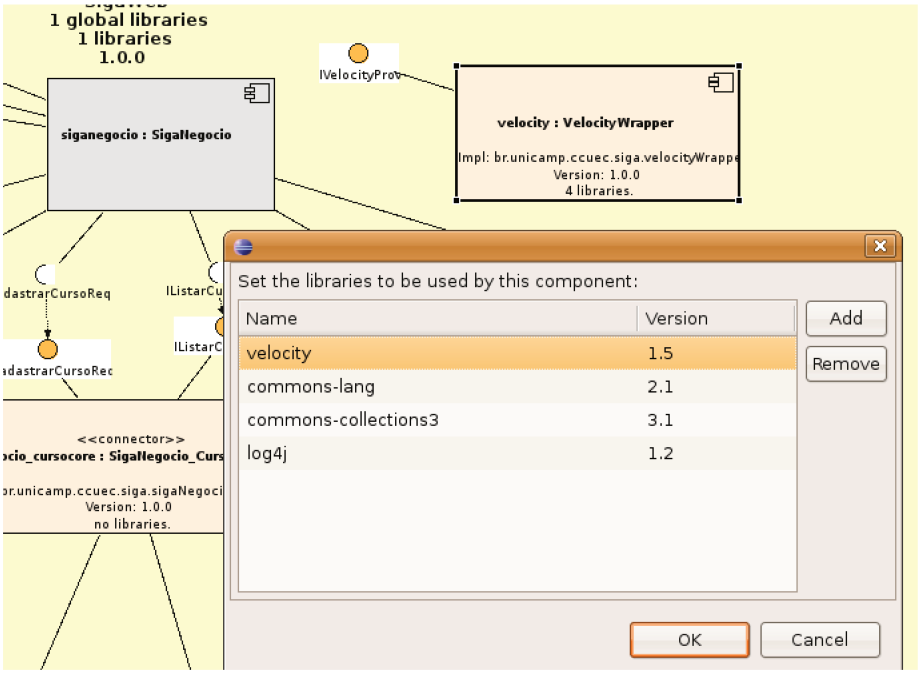


Figura 4.12: Caixa de diálogo com a lista de bibliotecas a serem utilizadas por um componente.

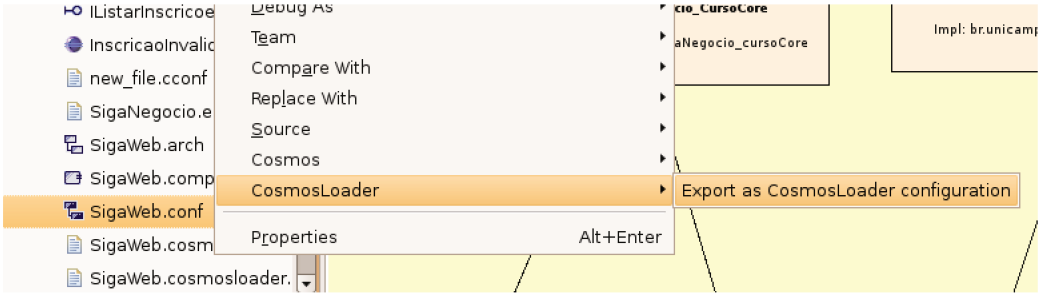


Figura 4.13: Opção para gerar arquivo cosmosloader

Capítulo 5

Estudos de caso

A ferramenta CosmosLoader foi avaliada em dois estudos de caso. O objetivo destes estudos foi verificar o quanto o uso da ferramenta facilita a programação de sistemas baseados no modelo Cosmos.

Dois sistemas implementados de acordo com o modelo Cosmos foram modificados para serem instanciados e executados com a ferramenta CosmosLoader. O primeiro foi realizado sobre uma aplicação real desenvolvida por uma empresa de consultoria e desenvolvimento de sistemas para automação bancária, e está descrito na Seção 5.1. O segundo estudo de caso foi realizado sobre um sistema acadêmico, e executado por alunos de pós-graduação do Instituto de Computação da Unicamp, e está descrito na Seção 5.2.

5.1 Estudo de caso com o Sistema Portal2

Este primeiro estudo de caso teve por objetivo avaliar a viabilidade do uso do ambiente Bellatrix modificado e da ferramenta CosmosLoader para instanciar um sistema baseado no modelo Cosmos*, e visou verificar que as ferramentas poderiam ser facilmente utilizadas no dia a dia por um programador.

O estudo de caso foi realizado no início de 2008 por uma programadora voluntária em uma empresa de sistemas de automação bancária na cidade de São Paulo. O estudo de caso foi realizado usando o sistema Portal2, que é um portal *web* para gerenciamento de contas bancárias desenvolvido pela empresa para bancos de pequeno e médio porte.

O sistema Portal2 foi implementado de acordo com o modelo Cosmos. O estudo de caso consistiu em utilizar o ambiente Bellatrix para modelar o sistema e gerar a descrição da descrição arquitetural e, em seguida, modificar o código de instanciação do sistema para carregá-lo com a ferramenta CosmosLoader. Por fim, a voluntária respondeu a um questionário relatando sua experiência com o uso das ferramentas.

5.1.1 Arquitetura do sistema Portal2

O sistema Portal2 é um sistema web, composto de dois componentes e um conector. Inicialmente, o sistema estava de acordo com o modelo Cosmos, no entanto, sem usar o conceito de componente composto, descrito na Seção 3.6. Os componentes foram agrupados como componentes internos de um novo componente Portal2, de acordo com o modelo Cosmos* para o estudo de caso.

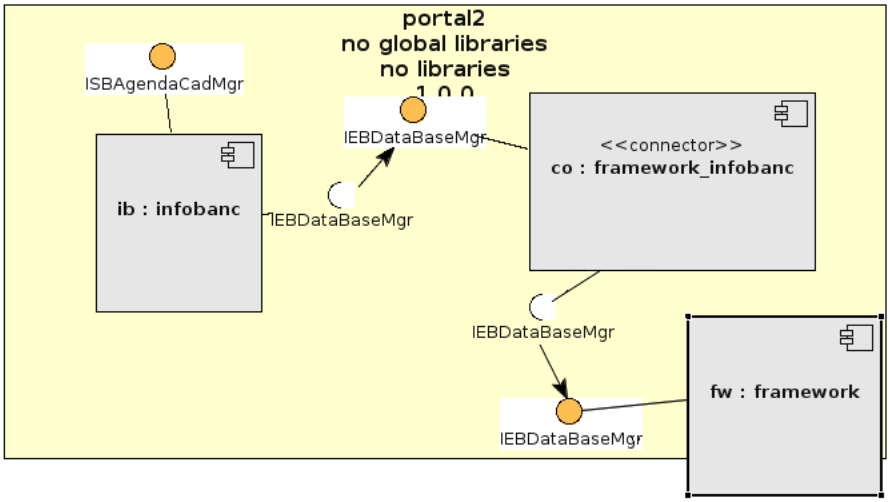


Figura 5.1: Arquitetura do sistema Portal2

A Figura 5.1 ilustra a arquitetura do sistema Portal2. O componente `infobanc` contém regras de negócios para gerenciamento de contas bancárias e o componente `framework` contém classes utilitárias para acesso ao banco de dados.

5.1.2 Execução do estudo de caso

Como dito anteriormente, o estudo de caso com o sistema Portal2 foi executado por uma programador voluntária da empresa de automação bancária proprietária do sistema. Esta voluntária tem extensa experiência com a plataforma Java e teve contato prévio com o modelo Cosmos para o desenvolvimento de projetos na empresa.

O estudo de caso foi executado de acordo com os seguintes passos:

1. A arquitetura do sistema Portal2 foi modelada com o ambiente Bellatrix modificado.

2. O ambiente Bellatrix modificado foi utilizado para gerar o arquivo de configuração arquitetural para a ferramenta CosmosLoader.

3. O sistema Portal2 foi modificado para ser instanciado através da ferramenta CosmosLoader.

4. Os componentes do sistema Portal2 foram adaptados para serem carregados a partir de um repositório, no formato esperado pela ferramenta (Seção 4.2.1).

5. Por fim, a voluntária respondeu a um questionário de avaliação da ferramenta. O questionário com as respostas está disponível no apêndice A.

5.1.3 Avaliação dos resultados

As respostas do questionário de avaliação da ferramenta apresentam os seguintes resultados:

- A voluntária considerou que "Foi necessário pouco esforço para adequar a arquitetura do sistema para que implementá-lo de acordo com o modelo COSMOS."
- A voluntária atribuiu os seguintes conceitos para cada atividade do estudo de caso:
  - Descrever o sistema: **Trabalhoso.**
  - Construir um componente elementar de acordo com o modelo COSMOS: **Pouco trabalhoso.**
  - Construir o componente de sistema de acordo com o modelo COSMOS: **Pouco trabalhoso.**
- A voluntária relata que houve falhas na integração do sistema com a ferramenta CosmosLoader. Estas falhas ocorreram porque os nomes das interfaces dos componentes foram digitados incorretamente durante a modelagem da arquitetura no ambiente Bellatrix
- A modelagem do sistema com o ambiente Bellatrix levou aproximadamente 16 horas. A adaptação dos componentes para um repositório levou também 16 horas.

5.2 Estudo de caso com o sistema eGov-CNH

O segundo estudo de caso foi realizado em meados de 2010 sobre o sistema eGov-CNH. O sistema eGov-CNH foi idealizado por alunos do Instituto de Computação da Unicamp para gerenciar o processo para um cidadão conseguir a Carteira Nacional de Habilitação de trânsito brasileira. Ele gerencia a solicitação do solicitante ao Departamento Estadual de Trânsito (Detran), agenda exames médicos junto a clínicas, as provas teórica e prática no Detran e gera boletos com as tarifas relacionadas.

O sistema eGov-CNH foi implementado com o objetivo de estudar a evolução de sistemas baseados em componentes. Ele foi escolhido para o estudo de caso com a ferramenta CosmosLoader por ser implementado de acordo com o modelo Cosmos*. O estudo de caso consistiu em duas etapas: na primeira etapa, um programador voluntário reescreveu o código de instanciação do sistema uma primeira vez, para medir o esforço necessário para a montagem de um sistema de acordo com o modelo Cosmos*; na segunda etapa, o mesmo voluntário reescreveu este código de instanciação uma segunda vez, utilizando a ferramenta CosmosLoader, para verificar a facilidade de uso da ferramenta, e respondeu a um questionário.

5.2.1 Arquitetura do sistema eGov-CNH

A Figura 5.2 ilustra a arquitetura do eGov-CNH:

O sistema eGov-CNH está arquitetado como um componente Cosmos* composto, de nome CNH, com 20 componentes e conectores internos.

As interfaces providas do composto são mapeadas para interfaces providas de componentes internos. As interfaces providas são `IDataController`, que contém operações para registro de dados do solicitante da Carteira; `IExamController`, para agendamento e registro de exames médicos e provas; e `IExamsPayer`, para controle de pagamento de tarifas.

As interfaces requeridas dos componentes internos são mapeadas como interfaces requeridas do composto: `IDetranReq`, para operar com sistemas existentes do Detran; `IClinicsService`, para operar com sistemas de clínicas cadastradas; `ICCService` e `IBankService`, para pagamento de tarifas através de boleto bancário ou cartão de crédito.

5.2.2 Implementação do sistema eGov-CNH

A Figura 5.3 ilustra a estrutura de pacotes do sistema eGov-CNH.

As classes do sistema eGov-CNH estão organizadas em pacotes para cada componentes e conector, de acordo com o modelo Cosmos*. Cada componente tem os pacotes de especificação (*spec*) e de implementação (*impl*), de acordo com os modelos de especificação

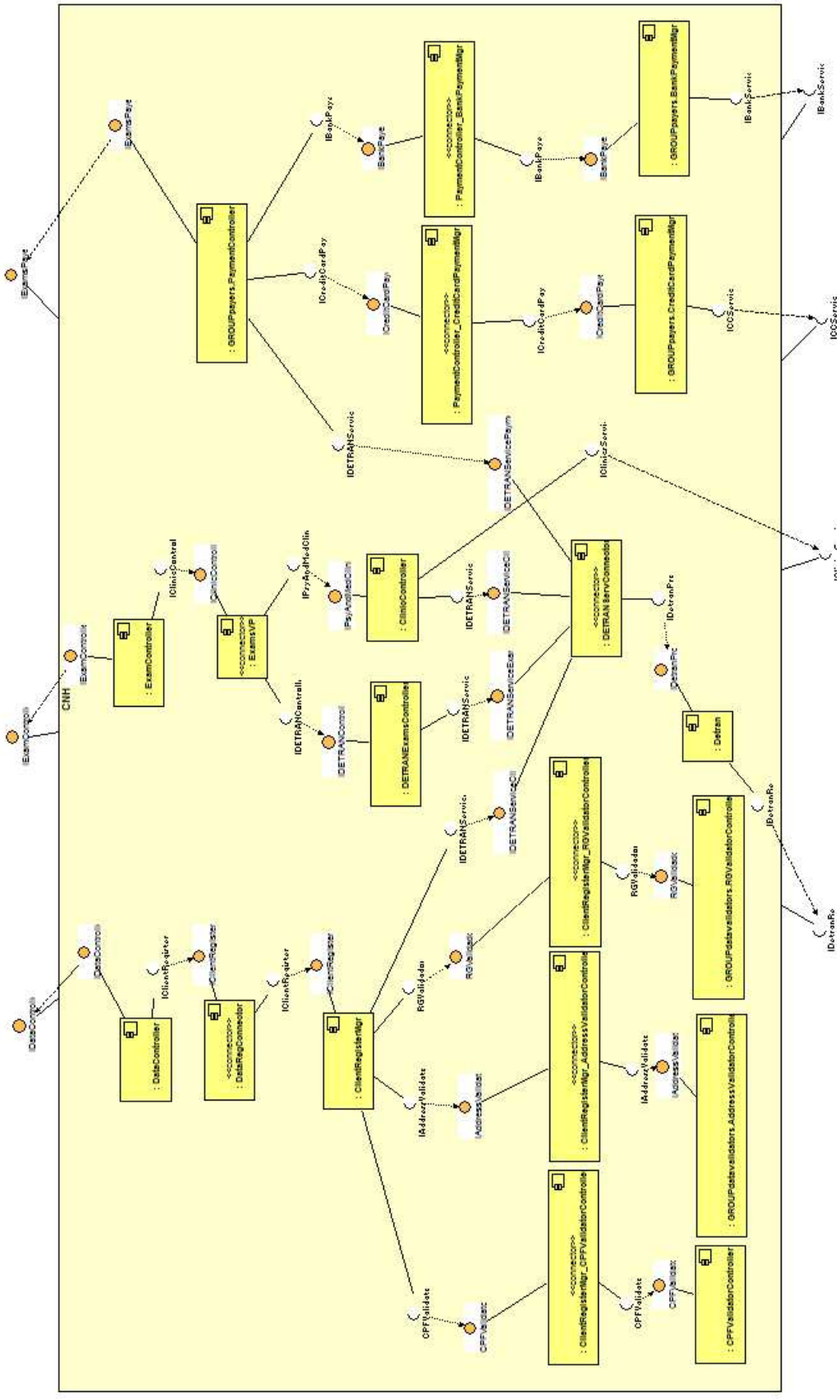


Figura 5.2: Arquitetura do sistema eGov-CNH

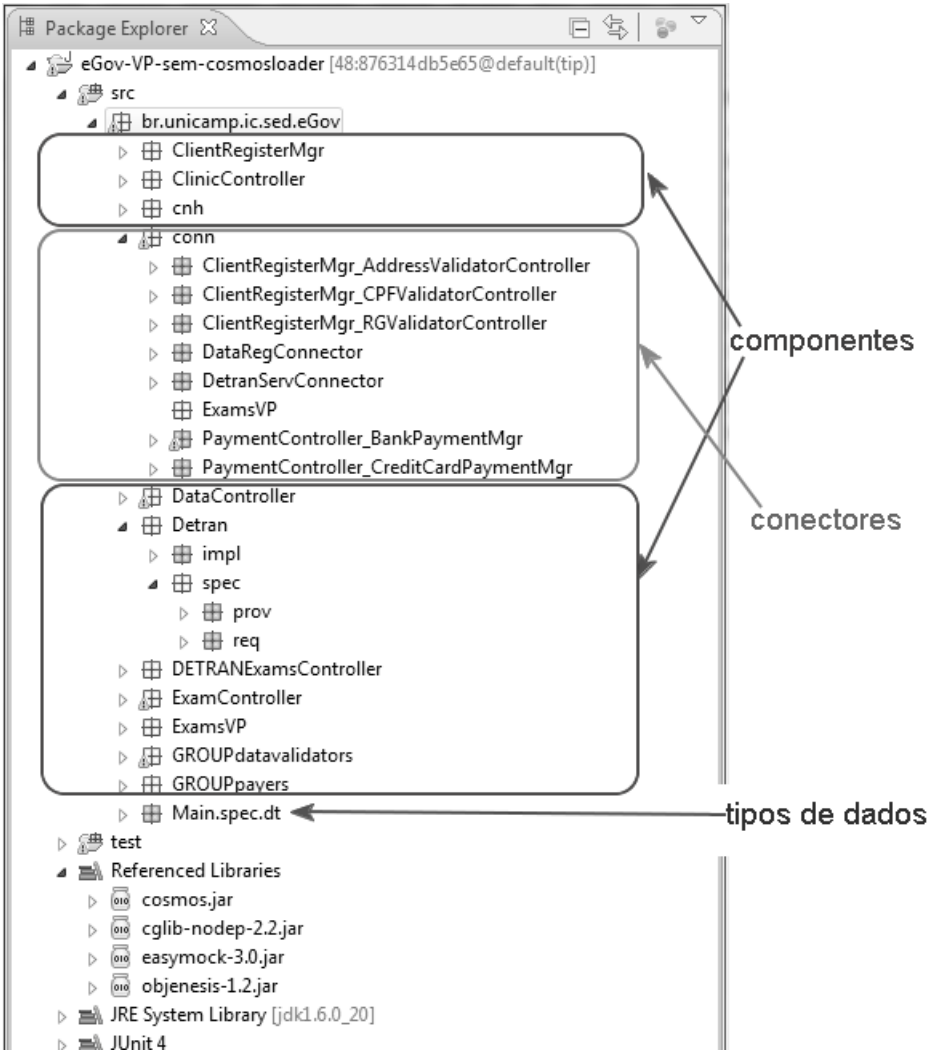


Figura 5.3: Estrutura de pacotes do sistema eGov-CNH

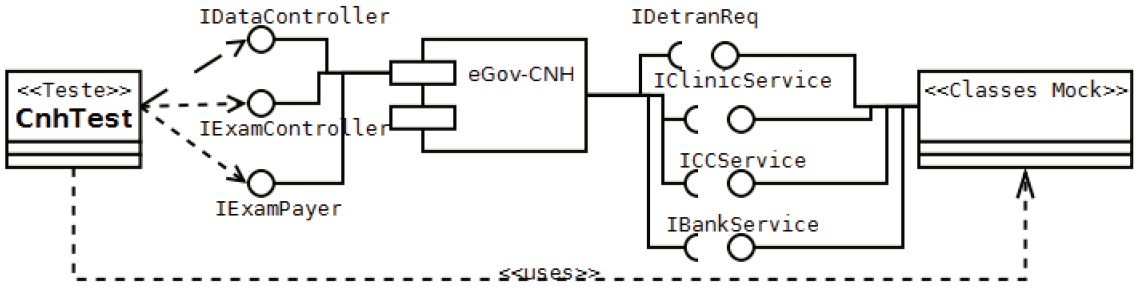


Figura 5.4: Estrutura de pacotes do sistema eGov-CNH

e implementação (Seções 3.2 e 3.3). Na Figura, são exibidos os pacotes de especificação e implementação do componente Detran; os demais estão omitidos. Os conectores do sistema são agrupados dentro do pacote `conn`.

Os componentes do sistema eGov-CNH não definem a interface IManager; eles compartilham a definição da interface contida no arquivo `cosmos.jar`, de acordo com a abordagem descrita na Seção 3.8. O arquivo cosmos.jar é uma dependência do sistema, e é exibido na Figura 5.3 como uma biblioteca (*Referenced library*).

Os tipos de dados que aparecem nas interfaces dos componentes são definidos no pacote global `Main.spec.dt` conforme a abordagem descrita na Seção 3.11.1.

Todos os componentes estão dentro de um pacote `br.unicamp.ic.sed.eGov`, que os identifica como parte dos sistemas de eGov do grupo SED do Instituto de Computação.

5.2.3 Estrutura de testes do sistema eGov-CNH

O sistema eGov-CNH contém uma série de testes de unidade. A Figura 5.4 ilustra o componente e a infraestrutura de testes. Cada classe de testes instancia o componente eGov-CNH e executa os métodos nas interfaces providas do componente.

Os testes são baseados nos *framework* de Testes JUnit [34] e EasyMock [24]. O framework JUnit oferece uma série de classes a serem estendidas pelas classes de testes da aplicação e permitem que os testes sejam executados em um ambiente de desenvolvimento. O framework EasyMock permite a criação, em tempo de execução, de objetos *mock*, que têm duas funcionalidades: eles implementam as interfaces requeridas do componente sob teste; e as chamadas a seus métodos são gravadas durante a execução do teste, de modo que é possível verificar posteriormente se o componente utilizou corretamente suas interfaces requeridas (por exemplo, invocando os métodos com parâmetros válidos).

A Figura 5.5 ilustra a classe de testes DataControllerTest do conjunto de testes.

```
1  package br.unicamp.ic.sed.eGov;
2  import static org.easymock.EasyMock.createMock;
3  import static org.easymock.EasyMock.replay;
4  import static org.easymock.EasyMock.expect;
5  import static org.easymock.EasyMock.verify;
6  import java.util.HashMap;
7  import java.util.Map;
8  import org.junit.Before;
9  import org.junit.Test;
10 import br.unicamp.ic.sed.cosmos.IManager;
11 import br.unicamp.ic.sed.eGov.Main.spec.dt.Client;
12 import br.unicamp.ic.sed.eGov.cnh.spec.prov.IDataController;
13 import br.unicamp.ic.sed.eGov.cnh.spec.req.IDetranReq;
14 import br.unicamp.ic.sed.eGov.dt.AddressImpl;
15 import br.unicamp.ic.sed.eGov.dt.ClientImpl;
16
17 public class DataControllerTest {
18   private IManager cnh;
19   private Client leonel = new Client("Leonel", "111.222.333-45");
20
21   @Test public void testDataController_handleCommand_consultar() {
22     cnh = br.unicamp.ic.sed.eGov.cnh.impl.ComponentFactory
23       .createInstance();
24
25     IDetranReq mockDetran = createMock(IDetranReq.class);
26     expect(mockDetran.doQuery(leonel.getCPF())).andReturn(leonel);
27     replay(mockDetran);
28
29     Map<String,Object> data = new HashMap<String,Object>();
30     data.put("cpf", leonel.getCPF());
31
32     cnh.setRequiredInterface("IDetranReq", mockDetran);
33     IDataController controller =
34       (IDataController) cnh.getProvidedInterface("IDataController");
35     controller.handleCommand("Consultar", data);
36
37     verify(mockDetran);
38   }
39 }
```

Figura 5.5: Teste de unidade do componente CNH

- O método `testDataController_handleCommand_consultar()` instancia o componente CNH (linha 21) e testa um dos métodos da interface `IDataController`.

- Nas linhas 25 a 27, a biblioteca EasyMock é utilizada para criar um objeto que serve de *mock* para a interface requerida `IDetranReq`. Na linha 26, o método expect instrui o objeto mock a esperar uma chamada a seu método doQuery recebendo

como argumento um número de CPF, e o objeto que deve ser retornado após esta chamada. Na linha 27, o método replay indica o fim das instruções e que o objeto deve começar a se comportar como a interface `IDetranReq`. Na linha 32, o objeto mock é conectado como interface requerida do componente cnh.

- As linhas 34 a 35 conduzem o teste: a interface provida `IDataController` do componente cnh é recuperada e o método handleCommand é invocado.

- Na linha 37, o método verify da biblioteca EasyMock é invocado. Este método verifica se as chamadas ao objeto foram feitas com os argumentos esperados. No caso contrário, o método lançará uma exceção para indicar que o componente falhou.

5.2.4 Objetivos do estudo de caso

A ferramenta CosmosLoader foi sugerida a partir da hipótese de que a escrita do código de instanciação no modelo Cosmos* seria trabalhosa e uma possível fonte de erros. Este segundo estudo de caso teve dois objetivos: 1) medir o esforço exigido de um programador para a escrita de código de instanciação em um sistema e verificar a existência de erros introduzidos neste processo; e 2) medir o esforço exigido para a utilização da ferramenta CosmosLoader.

Ao final do estudo de caso, o voluntário respondeu um questionário, que pode ser encontrado no Apêndice B. O questionário visou coletar as seguintes métricas:

- Qual foi o tempo total gasto para implementação da classe `ComponentFactory` do composto ?

- Qual a quantidade de linhas de código na classe `ComponentFactory` ?

- Quantas vezes você executou os testes e encontrou erros na sua implementação ?

O voluntário também foi instruido a coletar os registros de pilha de execução (*stack traces*) de eventuais erros ocorridos durante os testes; estes registros permitem identificar se os erros são devidos a defeitos na classe de instanciação criada pelo usuário ou em outras partes do sistema.

5.2.5 Execução do estudo de caso

O estudo de caso com o sistema eGov-CNH foi executado por um programador voluntário, que é aluno de pós-graduação do Instituto de Computação da Unicamp, tem ampla experiência com a plataforma Java e não teve contato prévio com o modelo Cosmos*.

O voluntário recebeu o código fonte em Java do sistema eGov-CNH, o código com testes de unidade e o diagrama de componentes do sistema no formato do ambiente Bellatrix; o código do sistema foi modificado, sendo removida a classe `ComponentFactory` do componente CNH, para que esta seja desenvolvida novamente pelo voluntário.

Este estudo de caso foi executado seguindo estes passos:

1. O voluntário desenvolveu a classe `ComponentFactory` de acordo com o modelo Cosmos* para substituir a classe removida do sistema, levantando as métricas de acordo com as instruções.

2. O voluntário descartou a classe `ComponentFactory` e modificou os testes de unidade para instanciar o sistema através da ferramenta CosmosLoader.

5.2.6 Avaliação dos resultados

As métricas do estudo de caso e os questionários respondidos pelo voluntário apresentam os seguintes resultados:

- *Qual foi o tempo total gasto para implementação da classe ComponentFactory do composto ?*
  - Sem CosmosLoader: 1 hora e 30 minutos
  - Com CosmosLoader: 5 minutos
- *Qual a quantidade de linhas de código na classe ComponentFactory ?*
  - Sem CosmosLoader: 96 linhas
  - Com CosmosLoader: 3 linhas
- *Quantas vezes você executou os testes e encontrou erros na sua implementação ?*
  - Sem CosmosLoader: 5 erros
  - Com CosmosLoader: nenhum erro

Uma análise das *stack traces* das execuções com erros revelou as causas dos erros durante a criação da classe `ComponentFactory` sem uso da ferramenta CosmosLoader:

- Por duas vezes, o voluntário digitou erroneamente o nome de uma interface.
- Por três vezes, o voluntário se esqueceu de conectar uma interface antes de instanciar o componente.

O voluntário fez também os seguintes comentários sobre o sistema durante o estudo de caso:

> A implementação do componente/conector clientRegisterMgr_RGValidatorController não reflete o conteúdo do arquivo CNH.arch. A interface requerida é "ValidadorRG" que por sua vez é provida por rgValidatorController com o mesmo nome. No CNH.arch está como "RGValidador";

Neste comentário o voluntário aponta um erro no diagrama de componentes provido para o estudo de caso: o nome da interface requerida de um componente no diagrama não condiz com o código do componente.

> Fazer as chamadas/instanciações de objetos usando strings é muito suscetível a erros de digitação;

Este comentário reflete os defeitos na classe de instanciação devidos a erros de digitação.

5.3 Conclusões dos estudos de caso

Os dois estudos de caso validam a viabilidade da ferramenta CosmosLoader para substituir a configuração através de código de instanciação em sistemas baseado em componentes Cosmos*: eles mostram que o esforço para configuração de sistemas baseados no modelo Cosmos* pode ser reduzido se o processo de configuração for automatizado.

Os estudos de caso são complementares: o primeiro estudo de caso avalia o esforço necessário para modelagem da arquitetura e adaptação de componentes para adequação a um repositório; o segundo estudo de caso compara os esforços na configuração através de código de instanciação e com uso da ferramenta CosmosLoader.

Ainda, os sistemas envolvidos nos estudos de caso são diferentes: o primeiro sistema, Portal2, tem uma quantidade menor de componentes, mas são componentes mais complexos e com maior quantidade de linhas de código; o segundo sistema, eGov-CNH, tem mais componentes, mas cada componente tem granularidade menor e menos linhas de código. Os resultados mostram que é possível utilizar a ferramenta para sistemas de diferentes níveis de complexidade.

Os estudos mostram também que grande parte do esforço de configuração se deve a evitar ou corrigir erros em nomes de interfaces: quando a arquitetura do sistema é transformada em componentes de acordo com o modelo Cosmos, existe a possibilidade de o programador digitar incorretamente o nome de uma interface ou conectar interfaces

de componentes errados. Isto é mais evidente no segundo estudo de caso, que tem uma maior quantidade de interfaces e componentes para conectar.

Capítulo 6

Conclusões

Este trabalho apresentou a ferramenta CosmosLoader, uma ferramenta que automatiza a criação de uma configuração arquitetural concreta em um sistema basedo em componentes Cosmos*.

Foram realizados dois estudos de caso com a ferramenta CosmosLoader. O primeiro foi realizado no ano de 2008, sobre um sistema para gerenciamento de finanças pertencente a uma empresa de automação bancária; o segundo foi realizado no ano de 2010, sobre um sistema para gerenciamento do processo para obtenção da carteira nacional de habilitação de trânsito brasileira.

Durante a elaboração deste trabalho, houve uma certa dificuldade com a escolha dos sistemas sobre os quais o estudo de caso seria realizado. Há poucos sistemas baseados no modelo Cosmos* fora do ambiente do grupo de pesquisas SED no Instituto de Computação da Unicamp; dentro do ambiente do grupo de pesquisas, há alguns sistemas construídos como exemplos de uso do modelo de componentes, mas cuja extensão não reflete sistemas reais em uso na indústria.

6.1 Contribuições

São contribuições deste trabalho:

- **Classificação do modelo de componentes Cosmos*** (Seção 2.4.5): neste trabalho, o modelo Cosmos* é inserido na classificação de componentes proposta por Crnkovic et al [15], identificando as dimensões de ciclo de desenvolvimento, mecanismos de composição, propriedades extrafuncionais e domínio de aplicação. O modelo Cosmos* é comparado a outros modelos consolidados na indústria, como EJB, CCM, Fractal, OpenCOM e OSGi.

- **Extensões ao modelo Cosmos** (Capítulo 3): o modelo Cosmos original foi inicialmente descrito na dissertação de mestrado de Silva Júnior [16, 17], contendo os modelos de especificação, implementação e conectores. Este trabalho estende o modelo Cosmos original com os modelos de composição e de componentes de sistema, refina as abordagens para declaração de tipos de dados e define uma especificação para a interface `IManager`, fundamental no modelo Cosmos, no pacote `br.unicamp.ic.sed.cosmos`.
- **Avaliação de esforço de programação do modelo Cosmos*** (Capítulo 5): a composição de componentes Cosmos* requer um esforço de programação para gerar o código que instancia componentes e conecta suas interfaces; este esforço se traduz em tempo de desenvolvimento e pode introduzir falhas no sistema. Os estudos de caso puderam medir este esforço em horas de desenvolvimento para um sistema de tamanho médio; o segundo estudo de caso comprovou a introdução de falhas no código de composição.
- **A ferramenta CosmosLoader** (Capítulo 4): a ferramenta CosmosLoader automatiza a configuração concreta de sistemas baseados no modelo Cosmos*, eliminando o tempo do programador para escrever o código de integração e o risco de introdução de falhas.

6.2 Publicações

Este trabalho originou as seguintes publicações:

- Uma especificação para uma ferramenta para geração automática de configurações concretas e de um repositório de componentes [26]. Esta especificação deu origem à ferramenta CosmosLoader.
- Uma documentação de referência do modelo Cosmos*, contendo especificação do modelo e exemplos de sistemas [27].

6.3 Trabalhos futuros

Existem algumas ideias de trabalhos futuros para estender as contribuições deste trabalho:

6.3.1 Cosmos 3

Para materializar o conceito de conexão de interfaces em construções da Programação Orientada a Objetos, os modelos Cosmos e Cosmos* especificam os métodos

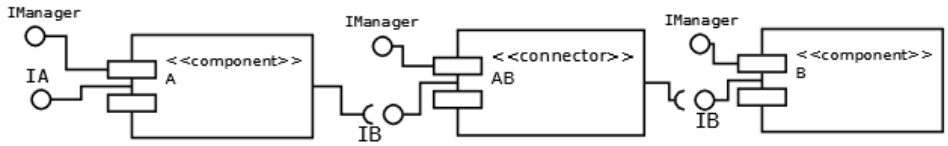


Figura 6.1: Configuração de componentes

`setRequiredInterface` e `getProvidedInterface` na interface `IManager` (Seção 3.4). Esta abordagem permite deduzir a arquitetura de um sistema a partir do código fonte (chamadas a `setRequiredInterface` e `getProvidedInterface` indicam uma conexão entre interfaces), mas somente o mapeamento entre arquitetura e código fonte estiver correto.

No entanto, existem desvantagens com esta abordagem. A desvantagem mais pronunciada é a grande quantidade de código necessária para a conexão de interfaces de componentes, evidenciada nos estudos de caso.

Uma segunda desvantagem é em relação a performance: se um componente utilizar tabelas para armazenar as interfaces, de acordo com a Seção 3.3.2, então cada chamada a um método em uma interface provida estará sujeita a uma consulta a uma tabela, reduzindo o desempenho do sistema.

Por fim, há mais uma desvantagem, que é o enfraquecimento da tipagem de dados da linguagem de programação: o método `getProvidedInterface` tem retorno do tipo `Object`, o que obriga o programador a realizar um *cast* do objeto retornado para a interface desejada. Isto permite que interfaces de tipos incompatíveis sejam conectadas por engano, gerando falhas no sistema e inconsistências entre o sistema e sua arquitetura.

Uma proposta é uma terceira versão do modelo de componentes, que retome a tipagem de dados como objetivo principal e reduza a quantidade de código de composição. Isto pode ser obtido criando para cada componente métodos com prefixos `set` e `get` que substituem `setRequiredInterface` e `getProvidedInterface`. Nos últimos anos, esta abordagem foi popularizada na indústria pelo modelo de componentes JavaBeans[38] e pelo *framework* Spring [43].

A Figura 6.1 retoma uma configuração de componentes apresentada no Capítulo 3. De acordo com este novo modelo de componentes, o código do componente `X` poderia ser reescrito sem sacrificar a tipagem das interfaces, como ilustrado na Figura 6.2.

Estudos para esta nova versão do modelo de componentes devem propôr novos modelos de composição e implementação. As avaliações podem comparar performance ou quantidade de linhas de código. Sugere-se o nome *"Cosmos 3"* para este novo modelo de componentes.

```
1 A a = new A();
2 B b = new B();
3 AB ab = new AB();
4 a.setIB(ab.getIB());
5 ab.setIB(b.getIB());
```

Figura 6.2: Composição de acordo com nova versão do modelo de componentes

6.3.2 Estudo da ferramenta CosmosLoader em diferentes plataformas

O modelo Cosmos* se baseia em conceitos de Programação Orientada a Objetos e pode ser aplicado em outras linguagens orientadas a objeto, além de Java. No entanto, por utilizar mecanismos como *class loaders*, a ferramenta CosmosLoader está fortemente acoplada à plataforma Java.

Uma possível extensão deste trabalho envolveria estudar a viabilidade da ferramenta CosmosLoader em outras plataformas. Por exemplo, a plataforma .Net [10] tem mecanismos semelhantes aos da plataforma Java. Ambas são baseadas em máquinas virtuais e têm arquivos com definições de classes (arquivos *jar* na plataforma Java, arquivos *dll* na plataforma .Net); ambas oferecem bibliotecas para introspecção de código (*reflection*). Isto sugere a possibilidade de reproduzir na plataforma .Net as funcionalidades de automação da ferramenta CosmosLoader.

Outros trabalhos futuros poderiam explorar a viabilidade da ferramenta em mais plataformas.

6.3.3 Aplicação do modelo Cosmos* para diferentes linguagens

Enquanto a Seção anterior sugeriu estudos em novas plataformas mas atendo-se a linguagens orientadas a objetos, esta Seção sugere ampliar o horizonte do Desenvolvimento Baseado em Componentes para novas linguagens de programação.

A plataforma Java já existe há 15 anos e, apesar de ter havido grandes melhorias na Máquina Virtual Java (JVM), nas bibliotecas padrão e nas tecnologias acerca da plataforma, a linguagem Java sofreu poucas mudanças. Isto ocasionou uma tendência recente, que é o surgimento de novas linguagens de programação compatíveis com a JVM (programas nestas linguagens são compilados para *byte-code* Java) e mais expressivas que a linguagem Java. Exemplos destas linguagens que têm se destacado na indústria incluem Scala [41, 49], Clojure [31, 32] e JRuby [33]. Também existe uma linguagem funcional para a plataforma .Net: F# [9] (“F sharp”), que tem suporte oficial da Microsoft.

A linguagem JRuby é uma implementação da linguagem Ruby na plataforma Java;

uma das funcionalidades da linguagem Ruby é que todas as classes do sistema podem ser modificadas em tempo de execução através de um sofisticado protocolo de metaobjetos. As linguagens Scala e Clojure são linguagens funcionais: Scala oferece os conceitos tradicionais de orientação a objetos, mas favorece um estilo de programação funcional. Este estilo de programação é ainda mais acentuado na linguagem Clojure, que deprecia o estilo de programação orientado a objetos, e compele todos os objetos a serem imutáveis. Ainda, um dos fundamentos da linguagem Clojure é utilizar a sintaxe baseada em parênteses da família de linguagens Lisp.

Em linguagens que não priorizam a programação orientada a objetos, quais seriam as características de um componente de software ? Trabalhos futuros nesta direção poderiam estudar a adaptação de um modelo de componentes para estas linguagens.

Apêndice A

Artefatos do estudo de caso com o sistema Portal2

Este capítulo contém o questionário aplicado no estudo de caso com o sistema Portal2. As alternativas selecionadas pelo voluntário e as respostas escritas estão destacadas.

- Qual é o seu perfil de desenvolvimento ?
  - Sou aluno da gradução.
  - Sou aluno da pós-graduação.
  - **Sou formado, e trabalho na indústria com desenvolvimento de sistemas.**
- Qual sua experiência com desenvolvimento de sistemas de software ?
  - Nenhuma. Este é o primeiro sistema que desenvolvo.
  - **Já desenvolvi alguns sistemas.**
  - Tenho ampla experiência com desenvolvimento de sistemas de software.
- Qual sua experiência com a plataforma Java, utilizada no desenvolvimento do sistema no estudo de caso ?
  - Nenhuma. Fui aprendendo à medida que desenvolvi o sistema.
  - Tenho pouca experiência com a plataforma Java.
  - **Tenho ampla experiência com a plataforma Java.**

- Foi fácil adequar a arquitetura do sistema para implementá-lo de acordo com o modelo COSMOS ?
  - A implementação com componentes COSMOS foi imediata.
  - **Foi necessário pouco esforço para adequar a arquitetura do sistema para que implementá-lo de acordo com o modelo COSMOS.**
  - Foi necessário muito esforço para adequar a arquitetura do sistema para que implementá-lo de acordo com o modelo COSMOS.
- Como você gerou o arquivo XML de descrição do sistema lido pelo CosmosLoader ?
  - **Utilizei o ambiente Bellatrix.**
  - Descrevi o modelo através da API do meta-modelo.
  - Escrevi o XML em um editor de texto.
- Atribua um conceito ("Muito trabalhoso (MT)", "Trabalhoso (T)", "Pouco trabalhoso (PT)") para cada etapa do desenvolvimento descrita a seguir:
  - Descrever o sistema.
    Trabalhoso
  - Construir um componente elementar de acordo com o modelo COSMOS.
    Pouco Trabalhoso
  - Construir o componente de sistema de acordo com o modelo COSMOS.
    Pouco Trabalhoso
- Os seguintes eventos aconteceram durante a integração do sistema ? Com que frequência ("Aconteceu com muita frequência (MF)", "Aconteceu com pouca frequência (PF)", "Não aconteceu (NA)")
  - Havia falhas nos componentes internos, desenvolvidos por terceiros.
    Não aconteceu
  - Houve falhas na integração do sistema sem o CosmosLoader.
    Não aconteceu
  - Houve falhas na integração do sistema com o CosmosLoader.
    Aconteceu com muita frequência

- Indique (Sim / Não) se cada uma das seguintes situações ocorreu durante a integração do sistema
  - Durante a integração do sistema sem o CosmosLoader, ocorreram falhas porque interfaces de componentes foram conectadas erroneamente. **Não**
  - Durante a integração do sistema sem o CosmosLoader, ocorreram falhas porque componentes internos foram registrados com nomes errados. **Não**
  - Ocorreram falhas na integração do sistema ocorreram porque os nomes dos componentes ou interfaces descritos no modelo diferiam dos nomes reais. **Sim**
- Avalie o tempo que você empregou em cada uma das atividades:
  - Construção do componente de sistema sem o CosmosLoader.
  - **Não foi construído um componente de sistemas sem o CosmosLoader.**
  - Construção do componente de sistema com o CosmosLoader.
  - **Levou-se aproximadamente 16 hs para modelar o sistema no Bellatrix e, então, gerar o XML de configuração para ser carregado pelo CosmosLoader.**
  - Criação do repositório com as implementações dos componentes internos.
  - **Foi necessário levantar todas as dependências do Portal 2, renomear as bibliotecas e os componentes internos de acordo com o padrão nome_versao.jar. O repositório foi montado em aproximadamente 16 hs.**

Apêndice B

Artefatos do estudo de caso com o sistema eGov-CNH

Durante o estudo de caso com o sistema eGov-CNH, foram aplicados dois questionários. O primeiro visava levantar o perfil do voluntário executando o estudo de caso (Seção B.1); o segundo visava registrar as impressões do voluntário durante o estudo de caso (Seção B.2).

B.1 Perfil do voluntário

- Qual é o seu perfil de desenvolvimento ?
  - Sou aluno da pós-graduação e trabalho na indústria com desenvolvimento de sistemas.
- Qual sua experiência com desenvolvimento de sistemas de software ?
  - Tenho ampla experiência com desenvolvimento de sistemas de software.
- Qual sua experiência com a plataforma Java, utilizada no desenvolvimento do sistema no estudo de caso ?
  - Tenho ampla experiência com a plataforma Java.
- Foi fácil adequar a arquitetura do sistema para implementá-lo de acordo com o modelo COSMOS ?
  - Foi necessário pouco esforço para adequar a arquitetura do sistema para que implementá-lo de acordo com o modelo COSMOS.

- Como você gerou o arquivo XML de descrição do sistema lido pelo CosmosLoader ?
  - Utilizei o ambiente Bellatrix
- Os seguintes eventos aconteceram durante a integração do sistema ? Com que frequência ("Aconteceu com muita frequência (MF)", "Aconteceu c om pouca frequência (PF)", "Não aconteceu (NA)")
  - Havia falhas nos componentes internos, desenvolvidos por terceiros: **NA**
  - Houve falhas na integração do sistema sem o CosmosLoader: **MF**
  - Houve falhas na integração do sistema com o CosmosLoader: **NA**
- Indique (Sim / Não) se cada uma das seguintes situações ocorreu durante a integração do sistema
  - Durante a integração do sistema sem o CosmosLoader, ocorreram falhas porque interfaces de componentes foram conectadas erroneamente: **Não**
  - Durante a integração do sistema sem o CosmosLoader, ocorreram falhas porque componentes internos foram registrados com nomes errados: **Sim**
  - Ocorreram falhas na integração do sistema ocorreram porque os nomes dos componentes ou interfaces descritos no modelo diferiam dos nomes reais: **Não**
- Avalie o tempo que você empregou em cada uma das atividades:
  - Construção do componente de sistema sem o CosmosLoader: **1 hora e 30min**
  - Construção do componente de sistema com o CosmosLoader: **5 min**

B.2 Execução do estudo de caso

O seguinte questionário registra as impressões do voluntário durante a execução do estudo de caso.

- *Qual foi o tempo total gasto para implementação da classe ComponentFactory do composto ?*
  - 1 hora e 30 min, após a primeira implementação, e assim adquirido prática, esse tempo deve diminuir consideravelmente.
- *Qual a quantidade de linhas de código na classe ComponentFactory ?*

  - 96 linhas. Algumas linhas de comentários foram incluídas como guia durante o desenvolvimento, além de uma identação "hierárquica" também usada para facilitar a implementação. A classe será enviada junto a esse questionário.

- *A ideia é escrever o código de uma só vez e executar os testes ao final, mas como este é um processo manual, é esperado que haja erros no código de instanciação. Quantas vezes você executou os testes e encontrou erros na sua implementação ? Se todos os testes correrem com sucesso na primeira vez, sem problemas. No entanto, como há uma grande quantidade de componentes a instanciar e interfaces a conectar, é interessante medir este esforço.*

  - 5 vezes os testes foram executados e retornando erros.
  - a implementação do componente/conector clientRegisterMgr_RGValidatorController não reflete o arquivo CNH.arch. A interface requerida é "ValidadorRG" que por sua vez é provida por rgValidatorController com o mesmo nome. No CNH.arch está como "RGValidador";
  - Fazer as chamadas/instanciações de objetos usando strings é muito suscetível a erros de digitação;

Referências Bibliográficas

[1] OSGi Alliance, Novembro 2010. http://www.osgi.org/.

[2] OSGi Alliance. Osgi service platform release 4, Novembro 2010. http://www.osgi.org/Specifications/HomePage.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[4] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32:38–45, 1999.

[5] Francois Bronsard, Douglas Bryan, W. Kozaczynski, Edy S. Liongosari, Jim Q. Ning, Asgeir Olafsson, and John W. Wetterstrand. Toward software plug-and-play. In *Proceedings of the 1997 symposium on Software reusability*, pages 19–29. ACM Press, 1997.

[6] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. An open component model and its support in java. In *Component-Based Software Engineering*, pages 7–22. 2004.

[7] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36:1257–1284, September 2006.

[8] John Cheesman and John Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[9] Microsoft Corporation. Microsoft f# developer center, Novembro 2010. http://msdn.microsoft.com/en-us/fsharp/default.aspx.

[10] Microsoft Corporation. Microsoft .net framework, Novembro 2010. http://www.microsoft.com/net.

[11] Oracle Corporation. The java ee 5 tutorial, Setembro 2010. http://download.oracle.com/javaee/5/tutorial/doc/.

[12] Oracle Corporation. Netbeans, Novembro 2010. http://www.netbeans.org.

[13] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26:1:1–1:42, March 2008.

[14] Ivica Crnković and Magnus Larsson. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[15] Ivica Crnković, Severine Sentilles, Vulgarakis Aneta, and Michel R.V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2010.

[16] Moacir Caetano da Silva Júnior. COSMOS - um modelo de estruturação de componentes para sistemas orientados a objetos. Master's thesis, Universidade Estadual de Campinas (Unicamp), 2003.

[17] Moacir Caetano da Silva Júnior, Paulo Asterio de Castro Guerra, and Cecília M. F. Rubira. A java component model for evolving software systems. In *ASE*, pages 327–330. IEEE Computer Society, 2003.

[18] Mark Davidson. The bean builder - a beanbox for the new millennium, Setembro 2010.

[19] Paulo Astério de Castro Guerra, Tiago César Moronte, Rodrigo Teruo Tomita, and Cecília Mary Fischer Rubira. Um modelo conceitual e uma terminologia para o desenvolvimento baseado em componentes e centrado na arquitetura de software. In Regina Maria Maciel Braga, editor, *Workshop de Desenvolvimento Baseado em Componentes*, pages 41–48, November 2005.

[20] Desmond F. D'Souza and Alan Cameron Wills. *Objects, components, and frameworks with UML: the catalysis approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[21] Apache Software Foundation. Welcome to apache felix, Novembro 2010. http://felix.apache.org/.

[22] The Eclipse Foundation. equinox osgi, Novembro 2010. http://www.eclipse.org/equinox/.

[23] Martin Fowler. Inversion of control containers and the dependency injection pattern. online, fevereiro 2007.

[24] Tammo Freese and Henri Tremblay. Easymock, 2010. http://easymock.org.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[26] Leonel Aguilar Gayard, Paulo Astério de Castro Guerra, Ana Elisa de Campos Lobo, and Cecilia Mary Fischer Rubira. Automated deployment of component architectures with versioned components. In *11th International Workshop on Component Oriented Programming (WCOP 2006)*, pages 48–53, ECOOP 2006, Nantes, France, 07 2006. Wolfgang Weck, Ralf Reussner and Clemens Szyperski.

[27] Leonel Aguilar Gayard, Cecília Mary Fischer Rubira, and Paulo Astério de Castro Guerra. COSMOS*: a COmponent System MOdel for Software Architectures. Technical Report IC-08-04, IC - Unicamp, 2008.

[28] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[29] Object Management Group. Corba component model, Abril 2006. http://www.omg.org/spec/CCM/.

[30] Object Management Group. Common object request broker architecture (corba), Janeiro 2008. http://www.omg.org/spec/CORBA/.

[31] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM.

[32] Rich Hickey. Clojure, Novembro 2010. http://clojure.org.

[33] JRuby. Jruby. 100language, Novembro 2010. http://jruby.org.

[34] junit.org. Resources for test driven development, 2010. http://www.junit.org.

[35] Knopflerfish. Knopflerfish - open source osgi, Novembro 2010. http://www.knopflerfish.org/.

[36] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[37] Sun Microsystems. Java technology, Dezembro 2007. http://java.sun.com/javase.

[38] Sun Microsystems. Java se desktop technologies, Março 2008. http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp.

[39] Sun Microsystems. jaxb: Jaxb reference implementation, Março 2008. https://jaxb.dev.java.net/.

[40] Sun Microsystems. Javaee at a glance, Maio 2010. http://java.sun.com/javaee.

[41] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.

[42] Oracle. Enterprise javabeans technology, Maio 2010. http://www.oracle.com/technetwork/java/index-jsp-140203.html.

[43] SpringSource. About spring, Novembro 2010. http://www.springsource.org/about.

[44] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[45] XDoclet Team. Xdoclet: Attribute oriented programming, Novembro 2010. http://xdoclet.sourceforge.net/.

[46] Tigris.org. Subversion is an open source version control system., Agosto 2008. http://subversion.tigris.org/.

[47] Rodrigo Teruo Tomita. Bellatrix: Um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes. Master's thesis, Universidade Estadual de Campinas, 2006.

[48] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33:78–85, March 2000.

[49] École Polytechnique Fédérale de Lausanne. The scala programming language, Novembro 2010. http://www.scala-lang.org.