COSMOS - Um Modelo de Estruturação de Componentes para Sistemas Orientados a Objetos

Moacir Caetano da Silva Júnior

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

COSMOS - Um Modelo de Estruturação de Componentes para Sistemas Orientados a Objetos

Moacir Caetano da Silva Júnior[1]

17 de Dezembro de 2003

Banca Examinadora:

- Profª. Dr.ª. Cecília Mary Fischer Rubira (Orientadora)
- Profª. Dr.ª. Claudia Maria Lima Werner
  COPPE - UFRJ
- Profª. Dr.ª. Maria Beatriz Felgar de Toledo
  Instituto de Computação - UNICAMP
- Profª. Dr.ª. Ariadne Maria Brito Rizzoni Carvalho (Suplente)
  Instituto de Computação - UNICAMP

[1] Apoio financeiro do CNPq

**FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DO IMECC DA UNICAMP**

Silva Júnior, Moacir Caetano da

Si38c COSMOS – um modelo de estruturação de componentes para sistemas orientados a objetos / Moacir Caetano da Silva Junior -- Campinas, [S.P. :s.n.], 2003.

Orientadora: Cecília Mary Fischer Rubira.

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

1. Engenharia de software. 2. Projeto de sistemas. 3. Software – Desenvolvimento. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Um Modelo de Estruturação de Componentes para Sistemas Orientados a Objetos

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Moacir Caetano da Silva Júnior e aprovada pela banca examinadora.

Campinas 17 de Dezembro de 2003

Prof.a. Dra. Cecília Mary Fischer Rubira (Orientadora)

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

Resumo

Este trabalho propõe um modelo de estruturação de componentes que mapeia arquiteturas de componentes para linguagens de programação. Esse modelo promove a conformidade da arquitetura de componentes do sistema com relação à sua implementação. O modelo também visa a manutenção das propriedades arquiteturais durante o desenvolvimento do software, através de um mapeamento sistemático e bem definido de como arquiteturas de software baseadas em componentes podem ser implementadas usando construções disponíveis em linguagens de programação.

O modelo COSMOS provê um conjunto de diretrizes de projeto para construir componentes de software adaptáveis e reusáveis. A principal preocupação do modelo COSMOS está no projeto dos componentes e conectores, e nas interações entre eles. A solução proposta define três modelos inter-relacionados: o primeiro define a visão externa de um componente; o segundo define como deve ser a implementação de um componente; e o terceiro define como se dá às interações entre os componentes através de conectores. Os três modelos compõem a nossa solução e chamam-se modelo de especificação, modelo de implementação e modelo de conectores, respectivamente.

COSMOS é um modelo de componentes, cuja técnica de estruturação de componentes e o mapeamento das descrições arquiteturais para elementos das linguagens de programação, podem ser adaptados a diferentes estilos arquiteturais e plataformas de componentes, como por exemplo, J2EE e . NET. Neste trabalho, também são apresentados dois estudos de caso da aplicação do modelo COSMOS no desenvolvimento de sistemas reais.

Abstract

This work proposes a model for structuring components for mapping component-based architectures to programming languages. This model enables the development of systems maintaining the conformance with the proposed software architecture, through a well-defined mapping of how component-based software architecture can be implemented using constructions available in programming languages.

The COSMOS model also provides a set of design guidelines to build adaptable and reusable software components. The main concern of the proposed model is the project of components and connectors and the definition of interactions among them. Our solution defines three interconnected models: the first defines the external view of components; the second defines how components should be implemented internally; and the third defines the interactions among components through connectors. The three models are called specification, implementation and connector models, respectively.

COSMOS is a component model, its techniques for structuring components and the mapping for the architectural descriptions to programming languages elements, can be adapted to different architectural styles and component platforms, for example J2EE and .NET. This work also presents two use cases to development of real software systems using the COSMOS model.

Agradecimentos

Primeiramente agradeço a Deus por permitir que eu chegasse até aqui. Tantos foram os obstáculos, mas cheguei.

Agradeço à minha mãe e minhas irmãs pelo apoio incondicional, amor, amizade e presença constante, mesmo que distante durante esses anos.

Um agradecimento todo especial à minha noiva Grace, que suportou a distância e a minha ausência, com muito amor e compreensão.

Agradeço à minha orientadora, Cecília Rubira, pela amizade, por acreditar no meu esforço e trabalho e também por me transmitir sua experiência e seus conhecimentos.

Agradeço a Paulo Astério pela dedicação e pelos momentos de discussão, fundamentais para o resultado final desta dissertação.

Agradeço a Vinícius pela dedicação na fase final deste trabalho.

Agradeço aos meus amigos: Tarcísio, Ricardo Mendonça, Aleksei, Ricardo Monteiro, Ricardo Alexandre e Eduardo pelo apoio.

Agradeço aos amigos da Ci&T, pelo apoio, pela troca de experiências e pelo suporte nas minhas ausências.

Finalmente, agradeço aos professores e funcionários do IC e também aos companheiros de mestrado pelo apoio e incentivo mútuo.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O desenvolvimento de software é um processo intrinsecamente difícil e consumidor de recursos pessoais e financeiros. Como os sistemas de software têm se tornado cada vez mais complexos e maiores, buscam-se mecanismos para reduzir a complexidade dos mesmos. Neste sentido, a engenharia de software age para melhorar a qualidade dos produtos e processos de desenvolvimento de software. Um processo de desenvolvimento de software define um conjunto de passos, métodos, técnicas e práticas que empregam pessoas para o desenvolvimento e manutenção de software e seus artefatos (planos, documentos, modelos, códigos, manuais etc.). É composto de boas práticas de engenharia de software que reduzem os riscos, aumentam a produtividade dos recursos empregados e melhoram a confiabilidade do software produzido [Jacobson+99].

Com a finalidade de diminuir o custo de desenvolvimento, minimizar a complexidade e melhorar a qualidade dos sistemas de software, surgiu a idéia de reusar partes de sistemas já existentes [Mcllroy69]. O conceito de modularização - decomposição do sistema em partes independentes (módulos) [Goguen86] - foi o alicerce para o surgimento de uma metodologia de desenvolvimento de software, centrada na composição de componentes de software - unidades de software desenvolvidas e testadas separadamente e que podem ser integradas com outras para construir algo com maior funcionalidade [Szyperski97] - chamada desenvolvimento de software baseado em componentes. A composição de software é o processo de construir aplicações, interconectando componentes de software [Nierstrasz+95].

A arquitetura de software engloba a definição da estruturas gerais, descreve os componentes do sistema, as interações entre eles, e busca uma solução técnica para um problema de negócio [Shaw+96]. Se a arquitetura estiver bem organizada e estruturada, cada componente ou parte dele pode ser construído visando o reuso. Qualquer que seja o nível do reuso, é necessário que o sistema possa sofrer alterações de forma localizada, sem afetar nenhuma outra parte, e que outras funcionalidades possam ser adicionadas com o menor impacto possível nas aplicações já existentes.

À medida que avança o desenvolvimento do sistema, novas partes vão sendo implementadas e, consequentemente, a manutenção vai se tornando cada vez mais complexa. Aproveitar-se de estruturas ou componentes já prontos garante mais robustez para o sistema, além de aumentar o grau de abstração, escondendo detalhes do problema que podem ser substituídos por uma solução pronta. A vida útil do sistema que possui uma boa arquitetura é acrescida em função da facilidade na incorporação de mudanças [Jacobson+97].

Algumas propriedades arquiteturais influenciam, direcionam e restringem todas as fases do ciclo de vida do software [Pressman01]. Esse é um dos motivos pelos quais o projeto arquitetural é tão importante para a construção de sistemas que realmente atendam às necessidades dos clientes. Neste sentido, a conformidade da arquitetura com relação a sua implementação é um requisito extremamente importante para que propriedades arquiteturais, tais como reusabilidade e facilidade de incorporação de mudanças, sejam efetivamente alcançadas nos sistemas produzidos.

1.1 Motivação e Problema

O crescente movimento em direção ao desenvolvimento baseado em componentes [Szyperski00] contribui para alavancar a arquitetura de software como um artefato central no ciclo de vida do software, assumindo importante papel na sua qualidade final e na obtenção dos benefícios propostos pelo desenvolvimento de software baseado em componentes. Arquitetura de software e desenvolvimento baseado em componentes são intimamente relacionadas e compartilham os conceitos de componentes e conectores. Conectores são entidades de software que mediam as conexões entre componentes, através da realização das dependências dos componentes [Luckham+00]. Contudo, os dois tópicos diferem nas suas motivações básicas para visualizar os sistemas como uma composição de componentes. Desenvolvimento baseado em componentes tem foco na integração de componentes pré-existentes, motivado pela diminuição do tempo de vida das aplicações e pela necessidade de menor tempo de desenvolvimento. Arquitetura de software é voltada para a organização de alto-nível e estruturação dos sistemas em geral, ou seja, na definição dos componentes, conectores e interações entre eles.

A maior parte da pesquisa em arquitetura de software concentra-se em discutir as propriedades de uma arquitetura de software ao nível de sistema, ou seja, pesquisa sobre estilos arquiteturais [Shaw+97] e descrição formal de arquiteturas de software [Medvidovic+00]. Contudo,

os benefícios de uma boa arquitetura de software são totalmente dependentes da forma como ela foi implementada. Se a atividade de implementação não seguir rigorosamente a especificação da arquitetura, inconsistências podem ser inseridas, causando confusão, violação de propriedades arquiteturais e inibindo a evolução do software. Os modelos de infra-estrutura de componentes de software, ou simplesmente modelos de componentes [Bernstein96, StiKeleather94] definem um conjunto de abstrações que devem ser seguidas para o desenvolvimento de componentes de software reutilizáveis. Além da reutilização, outras características como flexibilidade a mudanças, maior facilidade de implementação e adaptabilidade são requisitos que devem ser obtidos de um modelo de componentes.

A distância semântica entre as abstrações de uma descrição arquitetural e as construções disponíveis nas linguagens de programação, utilizadas para implementar o sistema de software, é um problema que os desenvolvedores de software enfrentam para garantir a conformidade da implementação com relação à arquitetura do software. Mais especificamente falando sobre arquiteturas de software baseadas em componentes, as linguagens de programação mais utilizadas atualmente não incorporam explicitamente as abstrações chaves de arquiteturas de software baseadas em componentes, por exemplo, os conceitos de componentes e conectores [Flatt99]. A linguagem de programação Java [Java02], por exemplo, não é exceção a essa regra, embora ela incorpore algumas abstrações bastante adequadas ao desenvolvimento de software baseado em componentes.

As tecnologias modernas de componentes [J2ee, DotNet] definem especificações para componentes que podem ser executados em ambientes de execução de diferentes fornecedores, por exemplo WebSphere da IBM [Websphere]. Tais tecnologias dão apoio a serviços muito complexos, como suporte transacional transação e persistência, mas não tratam explicitamente de aspectos, como flexibilidade a mudanças e adaptabilidade. A característica evolutiva dos sistemas de software modernos também contribui para dificultar a manutenção da conformidade da arquitetura do software com relação a sua implementação, uma vez que o dinamismo dos requisitos dos sistemas de software traz impactos nas definições e especificações já estabelecidas dos mesmos.

Assim, a definição de técnicas que auxiliem o processo de desenvolvimento de software baseado em componentes, enfatizando princípios como flexibilidade, manutenibilidade e reutilização, dentre outros, é de grande relevância no contexto da engenharia de software. Portanto, uma técnica bem estruturada que auxilie a implementação de descrições arquiteturais baseadas em

componentes, com regras claras e bem definidas, para o projeto e implementação dos componentes e conectores que interagem numa composição de software, de modo que características como adaptabilidade, reuso e flexibilidade a mudanças sejam preservadas, tem um importante papel na geração de sistemas de melhor qualidade.

Este trabalho apresenta um modelo de estruturação de componentes que mapeia arquiteturas de componentes para linguagens de programação, que promove a implementação de sistemas baseados em componentes, mantendo a conformidade da implementação com relação à arquitetura proposta para o sistema. A solução proposta define um mapeamento de como as abstrações arquiteturais baseadas em componentes podem ser implementadas, usando construções disponíveis nas linguagens de programação, mas especificamente, nas linguagens de programação orientadas a objetos.

1.2 Objetivos

Este trabalho tem como objetivo principal construir um modelo para mapeamento de arquiteturas de componentes em linguagens de programação. Esse modelo foi chamado de COSMOS (*Component Structuring Model for Object-oriented Systems*). O COSMOS define elementos para a implementação de componentes e conectores, participantes da composição de software, que podem ser implementados em diferentes linguagens de programação. São objetivos deste trabalho:

- Analisar técnicas propostas na literatura para lidar com problemas de evolução de software e propor diretrizes de projeto para guiar o desenvolvimento de sistemas mais flexíveis a mudanças;
- Analisar como a estruturação dos sistemas em termos de componentes e conectores pode tornar os sistemas mais flexíveis a mudanças;
- Estabelecer um modelo bem definido de estruturação de componentes, dando ênfase na definição dos componentes, conectores e interações entre eles;
- Garantir a manutenção das propriedades arquiteturais, durante o desenvolvimento do software, através de um mapeamento claro e bem definido de como arquiteturas de

software baseadas em componentes podem ser implementadas, usando construções disponíveis, em linguagens de programação orientadas a objetos;

- Construir componentes de software com maior potencial de reusabilidade, mais flexíveis a modificações e mais facilmente adaptáveis a outras composições de software.

São também objetivos deste trabalho:

- A aplicação do modelo COSMOS em sistemas implementados com tecnologias modernas de componentes como J2EE e .NET [J2ee, DotNet];
- A apresentação de dois estudos de caso de aplicação do modelo COSMOS no desenvolvimento de sistemas reais.

1.3 A Solução Proposta

O COSMOS é um modelo de estruturação de componentes que age, principalmente, na organização do sistema em termos de seus componentes, conectores e interações entre eles (Figura 1). O modelo de estruturação de componentes proposto no COSMOS preocupa-se principalmente em garantir as seguintes propriedades arquiteturais:

- Reusabilidade e adaptabilidade: que dependem muito do nível de acoplamento do sistema;
- Modificabilidade: que depende fortemente de como o sistema foi modularizado, pois isto reflete as estratégias de encapsulamento do sistema.

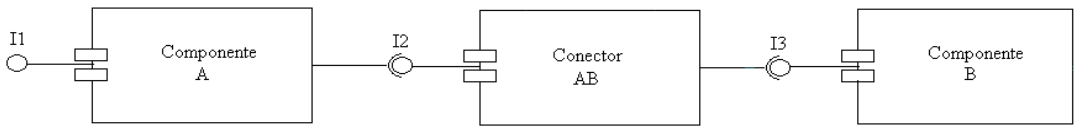

Figura 1. Solução do COSMOS para estruturação dos componentes

Uma propriedade arquitetural representa uma decisão de projeto relacionada com algum requisito não-funcional de um sistema [Sommerville01]. A reusabilidade dos componentes é

promovida pelo baixo nível de acoplamento do sistema, uma vez que os componentes interagem somente através de conectores, e um componente não tem conhecimento dos demais componentes que participam da composição de software. A adaptabilidade diz respeito à possibilidade de modificar ou substituir um componente, que participa de uma composição de software, isolando o maior número possível de modificações na implementação dos conectores, que podem, por exemplo, passar a referenciar um novo componente. O baixo acoplamento entre os componentes do sistema é obtido através da declaração das dependências dos componentes em interfaces, chamadas de interfaces requeridas. Uma interface é uma especificação do comportamento, através da qual é possível acessar um determinado serviço [Porter92]. Uma interface requerida define os serviços que um componente necessita que lhe sejam providos. Já a modificabilidade, está relacionada com o fato de que apenas a interface do componente é conhecida do usuário, possibilitando assim a modificação da implementação do componente sem afetar a sua utilização. As interfaces providas do componente definem os serviços que o componente provê a seus usuários.

O modelo COSMOS também incorpora um conjunto de diretrizes de projeto que visam habilitar a implementação de sistemas baseados em componentes, mantendo a conformidade com a descrição da arquitetura do software, e construir componentes de software mais flexíveis a mudanças, reutilizáveis e adaptáveis. As diretrizes de projeto são: (i) a materialização de elementos arquiteturais, que se preocupa com a materialização, em elementos do modelo de objetos, dos elementos arquiteturais, por exemplo, componentes e conectores; (ii) a inserção de requisitos não-funcionais do sistema nos conectores, para tornar mais simples o projeto dos componentes e promover a reutilização dos mesmos; (iii) a clara separação entre a especificação e a implementação do componente, de modo a garantir que apenas a especificação do componente seja pública; (iv) a declaração explícita dos serviços que o componente precisa para prover os seus próprios serviços; (v) as restrições quanto ao uso de herança na implementação do componente, e/ou entre classes de diferentes componentes, para facilitar a reutilização de código; (vi) e o baixo acoplamento entre as classes da implementação do componente para facilitar a evolução de sua implementação.

O modelo COSMOS baseia-se nestas diretrizes de projeto para compor as suas regras e definir os seus elementos. Além da estruturação do sistema em termos de componentes, conectores e interfaces, o COSMOS também se preocupa com o projeto dos componentes e conectores e com as interações entre eles. Para isso, a nossa solução define três modelos inter-relacionados: o primeiro define a visão externa de um componente, ou seja, suas interfaces requeridas e providas, através das quais um usuário do

componente pode resolver as suas dependências e acessar os seus serviços; o segundo define como deve ser a implementação de um componente; e o terceiro define como se dá às interações entre os componentes, através de conectores. Os três modelos, descritos acima, compõem a nossa solução e chamam-se modelo de especificação, modelo de implementação e modelo de conectores, respectivamente (Figura 2).

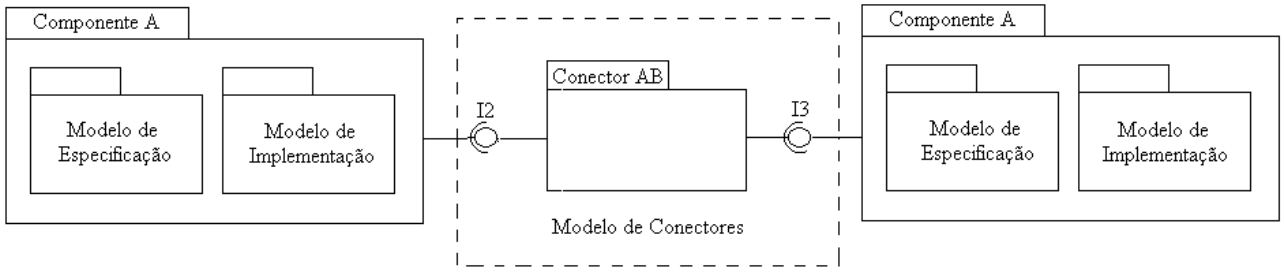

Figura 2. Solução do COSMOS para o projeto dos componentes e conectores

A solução proposta neste trabalho tem como principais contribuições: a integração de conceitos e idéias de pesquisas anteriores em orientação a objetos, arquitetura de software e desenvolvimento baseado em componentes num consistente modelo de estruturação de componentes; a definição de um conjunto de regras para guiar a construção de componentes de software; e a maior proximidade da arquitetura do sistema com relação a sua implementação.

Para os diagramas apresentados neste trabalho, usaremos a notação UML (*Unified Modeling Language*), definida em [Rumbaugh+99] e [Booch+99]. A Figura 1 mostra as interfaces requeridas dos componentes usando a notação que será padronizada para a UML 2.0. Björkander and Kobryn [Björkander+03] apresentam algumas das novas características da versão 2.0 da UML que será lançada em breve. Dentre as novas características, as dependências dos objetos são descritas por meio de portas que definem os serviços que um objeto requer, através de interfaces requeridas. Neste trabalho, usaremos essa notação para descrever as interfaces requeridas nos diagramas.

1.4 Trabalhos Relacionados

Koala [Ommering+00] é um modelo de componentes para sistemas embarcados em produtos eletrônicos. Ele usa uma descrição explícita e visual de arquitetura, baseada na linguagem de descrição arquitetural Darwin [Magee+96]. Como Darwin, ele também provê e requer interfaces

e as tratam como entidades de primeira grandeza. Enquanto Darwin foi originalmente concebida para ser usada em sistemas distribuídos, Koala demonstra a utilidade dessas características num modelo de componentes orientado ao reuso.

Whitehead et al [Whitehead+95] mostra que uma arquitetura bem projetada é um pré-requisito essencial para qualquer componente reusável. Eles identificaram critérios para uma boa arquitetura de software tais como: maior granularidade dos componentes, que pode ser obtida pela possibilidade de encapsular qualquer número de classes dentro de um componente; possibilidade de substituição, que pode ser obtida pelo uso de interfaces para especificar as dependências dos componentes; e facilidade de distribuição, que pode ser obtida pela integração de um ambiente de desenvolvimento e repositório de componentes.

WREN [Luer+01] é um protótipo de implementação de um ambiente integrado de desenvolvimento e execução de componentes. Um componente WREN contém classes e interfaces *Java*, e recursos. Interfaces são centrais na arquitetura de componentes de WREN. Um componente WREN é caracterizado pelas interfaces dos tipos de dados que ele provê ou requer. Ele deve conter uma classe que provê informação sobre o componente. Em particular, ela tem métodos que retornam descrições das portas do componente (isto é, quais tipos de dados são requeridos ou providos). A solução proposta neste trabalho compartilha algumas características da estrutura interna dos componentes, mas difere no sentido de que WREN é um ambiente de desenvolvimento e execução e o modelo COSMOS é um modelo de projeto e implementação de componentes. Além disso, o modelo de componentes de WREN não define detalhadamente a estrutura interna dos componentes como o modelo COSMOS.

McDirmid et al. [McDirmid+01] estende *Java* com componentes compilados separadamente e linkados externamente. Componentes são chamados de unidades, que podem ser de dois tipos: átomos, que são construídos a partir de classes *Java*, e compostos, que são construídos a partir da linguagem de componentes própria. Os átomos compilados como *Java byte codes* e os compostos são linkados para compor os sistemas. O modelo COSMOS difere do proposto em [McDirmid+01] porque o primeiro define um modelo genérico que usa as construções disponíveis nas linguagens de programação orientadas a objetos para implementar as abstrações de arquiteturas de software baseadas em componentes. Enquanto [McDirmid+01], define uma linguagem de alto-nível para descrever configurações arquiteturais e provê ferramentas para

transformá-las automaticamente em *Java byte codes*, linkando com componentes Java, definidos externamente.

ArchJava [Aldrich+02] também define uma extensão para *Java* que permite a transformação de especificações arquiteturais em código de linguagem de programação. A extensões de linguagem, providas por *ArchJava*, permitem que programadores expressem estruturas arquiteturais e transforme-as em código *Java*. Ele provê um pré-processador que transforma código de *ArchJava* em código *Java*. Assim como a extensão definida por McDirmid, *ArchJava* também define sua própria linguagem de alto-nível para expressar estruturas arquiteturais.

Rosenblum e Natarajan [Rosenblum+00] estende o modelo de JavaBeans [JavaBeans] para incorporar a noção de componentes e conectores como definidos no estilo arquitetural C2 [Taylor+95]. Esta extensão é apoiada por um ambiente de desenvolvimento de software chamado de *Arabica*, que é uma extensão do *Sun Microsystems Beans Development Kit*. Ao contrário, o modelo COSMOS não implica em qualquer estilo arquitetural específico.

Assim como *JavaBeans*, nós também usamos padrões para construir os componentes. Contudo, no modelo COSMOS, um componente tem uma granularidade maior que um componente *JavaBean*. No modelo COSMOS, um componente é composto por um conjunto de classes e interfaces, enquanto que um componente *JavaBeans* pode ser uma única classe. Também, a noção de interfaces requeridas e conectores não está presente no modelo de *JavaBeans*, onde a interação entre os componentes é realizada por meio de eventos. Com relação ao modelo de componentes da *Microsoft* COM [Com], os conceitos de interfaces requeridas e conectores também não estão presentes.

Características	**JavaBeans**	**COM**	**ArchJava**	**WREN**	**COSMOS**
Separação entre interface e implementação.	Sim	Sim	Sim	Sim	Sim
Declaração explícita das dependências entre componentes (interfaces requeridas).	Não	Não	Sim	Sim	Sim
Aplicação do conceito de conectores.	Não	Não	Sim	Sim	Sim

Baixo acoplamento entre os componentes de uma composição de software.	Não	Não	Sim	Sim	Sim
Aplicação de diretrizes para a implementação dos componentes.	Não	Não	Não	Não	Sim
Facilidade de evolução	Não	Não	Não	Não	Sim
Suporte ferramental	Sim	Sim	Sim	Sim	Não[2]

Tabela 1. Quadro comparativo entre os modelos de componentes

1.5 Organização deste Documento

Este documento foi dividido em sete capítulos organizados da seguinte forma:

- **Capítulo 2 - Fundamentos teóricos**: Para embasar este trabalho, o segundo capítulo apresenta fundamentos teóricos de componentização, conceitos e técnicas de evolução de sistemas orientados a objetos, e algumas tecnologias de componentes. As técnicas de evolução apresentadas descrevem problemas clássicos de evolução de sistemas orientados a objetos, tal como problema de classe base frágil, e traçam soluções para guiar o desenvolvimento desses sistemas. As tecnologias de componentes apresentadas refletem o grande interesse atual pelo desenvolvimento de software baseado em componentes.
- **Capítulo 3 - Diretrizes de projeto do modelo COSMOS**: Neste capítulo, são apresentadas as diretrizes de projeto nas quais o modelo COSMOS é baseado. Tais diretrizes visam construir de componentes de software flexíveis a mudanças, adaptáveis e reutilizáveis, bem como preservar a conformidade da arquitetura do software com relação a sua implementação.
- **Capítulo 4 - O modelo COSMOS**: Este capítulo descreve o modelo de COSMOS, suas características, seus elementos e suas principais contribuições.

[2] O suporte ferramental para o modelo COSMOS está em desenvolvimento.

- **Capítulo 5 - Avaliação prática do modelo COSMOS**: Este capítulo descreve dois estudos de caso realizados com o COSMOS. O primeiro, consiste na aplicação do modelo por uma empresa privada de desenvolvimento de sistemas de BioInformática, que customizou o modelo de acordo com as suas necessidades Já o segundo, consiste na aplicação completa do modelo no desenvolvimento de um sistema de Informações *Web* sobre condições de conservação de rodovias. Este estudo de caso teve o enfoque de aplicar completamente o modelo, avaliar os resultados da sua aplicação e propor extensões para o mesmo. Neste capítulo, também são discutidas algumas lições aprendidas com os estudos de caso e alguns aspectos da implementação de um sistema baseado em componentes.
- **Capítulo 6 - Conclusões e trabalhos futuros**: Este capítulo sintetiza as contribuições deste trabalho, apresenta as conclusões e sugere trabalhos futuros.

Capítulo 2

Fundamentos Teóricos

Este capítulo estabelece a terminologia e define os conceitos utilizados neste trabalho. Na seção 2.1 apresentamos os princípios de desenvolvimento baseado em componentes, que foram empregados neste trabalho. Na seção 2.2, discutimos o conceito de evolução de software e mostramos algumas soluções da literatura que tratam os problemas de evolução de sistemas orientados a objetos, que foram utilizadas para compor a solução apresentada neste trabalho. Por fim, a seção 2.3 apresenta duas tecnologias de componentes, J2EE e .NET, que serão utilizadas para descrever os experimentos práticos realizados neste trabalho.

2.1 Fundamentos de Desenvolvimento Baseado em Componentes

Esta seção se dedica a apresentar o estado da arte em termos desenvolvimento de software baseado em componentes. Na seção 2.1.1, apresentamos as principais definições de desenvolvimento de sistemas baseado em componentes, que garante um desenvolvimento mais rápido, confiável e barato. Já a seção 2.1.2 apresenta a definição de processo de desenvolvimento de software baseado em componentes, método básico para obter sucesso na construção de qualquer sistema computacional baseado em componentes. Por fim, a seção 2.1.3 apresenta o conceito de Arquitetura de Software e como suas propriedades podem influenciar o sistema resultante.

2.1.1 Desenvolvimento Baseado em Componentes

O desenvolvimento baseado em componentes vem conquistando muita atenção da comunidade de engenharia de software como uma nova perspectiva para o desenvolvimento de software, ao invés daquela baseada em blocos monolíticos, formados por um número de partes interrelacionadas de forma implícita que, até bem pouco tempo, era a estratégia de desenvolvimento da maioria dos produtos de software disponíveis no mercado [Werner+00]. O desenvolvimento baseado em componentes permite que uma aplicação seja construída pela composição de componentes de software que já foram previamente especificados, construídos e testados, o que resulta em ganho de produtividade e qualidade no software produzido. O aumento da produtividade advém da reutilização de componentes existentes em novos sistemas, enquanto que o aumento da qualidade advém do fato do componente já ter sido utilizado, modificado e testado em outros sistemas, eliminando boa parte dos erros que poderiam ocorrer. Com isso, os produtos finais não são mais produtos de software monolíticos e fechados, mas baseados em componentes que podem ser integrados com outros produtos [Larsson99].

Vários autores adotam conceitos correlatos de componentes, mas com diferenças sutis. Algumas definições encontram-se abaixo:

"Um componente é um pacote desenvolvido e testado separadamente, distribuído como uma unidade que pode ser integrada com outros componentes para construir algo com maior funcionalidade" [Szyperski97].

"Um componente é uma parte física, substituível de um sistema que empacota implementação e oferece a realização de um conjunto de interfaces" [Uml99].

"Componentes são módulos ou estruturas de dados, desenvolvidos e compilados separadamente, e unidos para formar aplicações que interagem através de interfaces" [Chambers96].

Apesar da ausência de um consenso geral sobre o conceito de componente, um aspecto muito importante é sempre ressaltado na literatura: um componente deve encapsular dentro de si seu projeto e implementação, e oferecer interfaces bem definidas para o meio externo. A

dissociação entre especificação e implementação, possibilita a interação com outros componentes unicamente através de suas interfaces, que são projetadas visando a composição de componentes [Nierstrasz+95]. O propósito da separação de um componente em duas partes (interfaces e implementação) é garantir a flexibilidade da forma como o componente pode ser conectado a outros componentes e substituído por outros. Com essa separação, é possível, numa aplicação baseada em componentes, modificar a implementação do componente ou, até mesmo, substituí-lo por outro componente com uma especificação semelhante, sem prejudicar as interações e a funcionalidade disponível. Se houver qualquer dependência de implementação, a possibilidade de substituição do componente pode ser perdida.

As interfaces correspondem às portas dos componentes e, a comunicação com outros componentes é feita através de conectores. Neste trabalho, os conectores foram elevados à categoria de componentes, com a particularidade de que eles só existem porque dois ou mais componentes precisam interagir numa composição de software. Conectores possuem um conjunto de características que os tornam especialmente atraentes para tratar os aspectos relacionados com a interação e a comunicação entre componentes [Bishop+96]. Dentre estas características destacam-se:

- Conhecimento das interfaces e referências dos componentes por eles conectados;
- Capacidade de examinar e manipular o conteúdo de requisições e respostas antes de encaminhá-las aos seus destinos finais;
- Possibilidade de controlar o direcionamento das requisições e respostas para os componentes.

Componentes de software possuem representação lógica e binária, ou física. Esta última é representada sob a forma de linguagem de máquina, ou de uma representação intermediária, que pode ser executada num computador, como os *byte-codes* de Java. Um componente lógico é uma representação, ao nível de projeto, de um pacote que deve estar bem separado de seu ambiente e de outros pacotes, encapsulando características e mantendo funcionalidades auto-contidas [Szyperski97].

Neste trabalho, nos concentramos na definição de um modelo de estruturação de componentes para sistemas baseados em arquitetura, através da definição de um mapeamento para

materializar componentes, conectores e interações, descritos arquiteturalmente, em elementos do modelo de objetos. De acordo com Huang et al. [Huang+03], os elementos arquiteturais devem ser materializados em tempo de execução como objetos explícitos. Assim, a nossa preocupação será apenas com componentes lógicos e, a partir deste ponto, quando nos referirmos à componentes de software, estaremos falando especificamente de componentes lógicos.

2.1.2 Processo de Desenvolvimento de Software Baseado em Componentes

O desenvolvimento baseado em componentes visa fornecer um conjunto de procedimentos, ferramentas e notações que possibilitem, ao longo do processo de software, a ocorrência tanto da produção de novos componentes quanto a reutilização de componentes existentes. Alguns modelos de processos de desenvolvimento de software têm surgido na literatura, como é o caso do *Catalysis* [D'Souza+98] e o *UML Componentes* [Cheesman+01]. Um processo de desenvolvimento de software é um conjunto de etapas, métodos, técnicas e práticas que empregam pessoas para o desenvolvimento e manutenção de um software e seus artefatos associados (planos, documentos, modelos, código, casos de testes, manuais, etc.). É composto de boas práticas de engenharia de software que conduzem o desenvolvimento do software, reduzindo os riscos e aumentando a confiabilidade dos sistemas [Jacobson+99].

Embora existam muitos processos de desenvolvimento de software, algumas atividades fundamentais estão presentes em todos eles. São elas: levantamento de requisitos, projeto, implementação e testes. A fase de levantamento de requisitos focaliza na identificação dos serviços que devem ser automatizados, as informações que devem ser processadas, além de questões como desempenho, confiabilidade, disponibilidade e segurança. O modelo de casos de uso é uma ferramenta efetiva e amplamente empregada para o levantamento dos requisitos. Durante a fase de projeto, os desenvolvedores se concentram em definir como os dados serão estruturados, como a funcionalidade será descrita através de uma arquitetura de software e em como as interfaces serão caracterizadas. Durante a implementação, o projeto é transportado para uma linguagem de implementação em forma de código fonte e, durante a etapa de testes, o sistema é verificado para certificar-se de que os requisitos especificados estão todos implementados corretamente.

Os requisitos de um software são as características, propriedades e comportamentos desejáveis para aquele produto. Costuma-se dividir os requisitos em:

- Requisitos *funcionais*, que representam os comportamentos que um sistema deve apresentar diante de certas ações de seus usuários;
- Requisitos *não funcionais*, que quantificam determinados aspectos do comportamento dos sistemas.

Somando-se às características mais comuns dos processos de desenvolvimento de software convencionais, um processo de desenvolvimento de software baseado em componentes apresenta algumas características particulares. Essas particularidades podem ser observadas no acréscimo de estágios técnicos ao processo convencional ou em uma ênfase maior em algumas práticas já realizadas nesses processos. Segundo Jacobson et. al. [Jacobson+99], um processo de desenvolvimento de software baseado em componentes geralmente inclui a definição de estratégias para:

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

2.1.3 Arquitetura de Software

A arquitetura de um sistema de software engloba a definição de suas estruturas gerais, descrevendo os elementos que compõem o sistema e as interações entre estes [Shaw+96]. Além disto, uma arquitetura de software apoia, também, questões importantes de projeto, tais como: a organização do sistema como uma composição de componentes, as estruturas de controle globais, os protocolos de comunicação, a composição dos elementos do projeto e a designação da

funcionalidade dos componentes do projeto. Uma definição da arquitetura nos dá uma clara perspectiva de todo o sistema e do controle necessário para seu desenvolvimento. Isto é conseguido através de várias abstrações e visões do sistema sobre as perspectivas dos diferentes colaboradores, isto é, desenvolvedores, analistas, projetistas, clientes, usuários, entre outros. Uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito não-funcional [Sommerville01]. Exemplos de propriedades arquiteturais são:

- Modificabilidade: característica que define a capacidade do sistema de se adaptar a alterações de requisitos ou mesmo a inclusão de novos requisitos;
- Reusabilidade: característica que define o grau de reuso de um componente, isto é, define quão genérico e independente da aplicação este componente é, para que possa ser reutilizado em diversas aplicações;
- Desempenho: tempo de resposta ou tempo de processamento de uma requisição que deve ser compatível com a realidade e necessidade do cliente;
- Tolerância a Falhas: capacidade do sistema de reagir e recuperar-se diante de situações excepcionais;

As propriedades arquiteturais são derivadas dos requisitos do sistema e influenciam, direcionam e restringem todas as fases do ciclo de vida do software. A Modificabilidade, por exemplo, depende fortemente de como o sistema foi modularizado, pois isto reflete as estratégias de encapsulamento do sistema. A Reusabilidade de componentes depende do nível de acoplamento dos componentes do sistema. O Desempenho depende da complexidade de comunicação entre os componentes e especialmente da distribuição física destes componentes. Tolerância a falhas, por sua vez, só é possível através da aplicação de técnicas de redundância de software e/ou hardware e tratamento de exceções.

A presença de uma determinada propriedade arquitetural pode ser obtida através da utilização de estilos arquiteturais que possam garantir a preservação dessa propriedade durante o desenvolvimento do sistema [Monroe+97, Shaw+96]. Um estilo arquitetural caracteriza uma família de sistemas que são relacionadas pelo compartilhamento de propriedades estruturais e comportamentais (semântica). Um estilo de arquitetura provê um vocabulário de elementos da

arquitetura, também chamados de componentes arquiteturais, e seus conectores. Além disto, apresenta ainda regras e restrições sobre a combinação dos componentes arquiteturais.

2.2 Evolução de Sistemas Orientados a Objetos

Evolução de software representa o ciclo contínuo de atividades envolvidas no desenvolvimento, uso e manutenção de software [Sommerville01]. Sistemas de software evoluem no tempo em resposta a inúmeros requisitos, incluindo a correção de erros, demanda de usuário por maior funcionalidade e especialmente para dar apoio a mudanças no software. A evolução de sistemas orientados a objetos produz mudanças nas classes do sistema que podem ser divididas em três categorias [Dmitriev98]:

- Mudanças no nível de classes ou de hierarquia de classes.
- Mudanças que afetam ou não dados persistentes.
- Mudanças conservativas ou não conservativas.

Mudanças no nível de classes são aquelas que afetam apenas classes individuais, enquanto que mudanças na hierarquia de classes são, por exemplo, a inserção ou exclusão de classes do meio da hierarquia. Mudanças que afetam dados persistentes são essencialmente modificações em campos de dados de instâncias de uma classe. Mudanças conservativas são aquelas que não quebram o contrato entre uma determinada classe e outras, isto significa, por exemplo, que somente o código (não a assinatura) de algum método existente pode ser modificado e novos métodos e campos de dados podem ser acrescentados, enquanto que mudanças não conservativas podem quebrar o contrato estabelecido e, dessa forma, requerem mais cuidado.

Diversas técnicas foram propostas para resolução de problemas relacionados com a evolução de sistemas orientados a objetos, dentre elas pode-se citar técnicas que propõem: restrições sobre o uso da herança, regras para a evolução de hierarquias de classes, separação entre hierarquias de classes e tipos e a separação entre interfaces e implementação.

2.2.1 Restrições sobre o uso de Herança

Mikhajlov e Sekerinsk [Mikhajlov+97] propuseram a utilização disciplinada da herança para evitar o problema de classe base frágil (*fragile base class problem*). Uma classe base é uma classe cuja implementação pode ser herdada por outras classes (subclasses) O nome problema de classe base frágil, por si só, já dá uma idéia intuitiva do problema. Classes de um sistema orientado a objetos podem ser frágeis. Uma pequena modificação dessas classes pode danificar o sistema inteiro. A modificação das classes por seus desenvolvedores não deveria afetar as extensões das classes, realizadas pelos usuários das mesmas, em qualquer aspecto. Primeiramente, a recompilação de classes derivadas deveria ser evitada [Som]. Entretanto, mesmo se a recompilação não for necessária, desenvolvedores ainda podem realizar modificações inconsistentes.

O problema não depende da linguagem de implementação do sistema. Contudo, sistemas que utilizam herança de código como mecanismo de reutilização de implementação com vínculo dinâmico (*dynamic binding*) estão vulneráveis ao problema [Taenzer89]. Através da herança, a funcionalidade provida pelo sistema pode ser reutilizada por seus usuários. Além disso, quando o sistema aberto é um *framework* orientado a objetos, os usuários podem estender a funcionalidade do sistema substituindo suas classes por outras derivadas. Em geral, desenvolvedores de sistemas não estão conscientes da existência de extensões, desenvolvidas por usuários. Para tentar melhorar a funcionalidade do sistema, os desenvolvedores podem produzir uma revisão (nova versão), aparentemente aceitável de classes do sistema, mas que pode causar problemas às subclasses. Num sistema fechado, todas as extensões estão sob controle dos desenvolvedores do sistema que, a princípio, podem analisar os efeitos de certas revisões de classes bases sobre o sistema inteiro.

À primeira vista, o problema parece ser causado por especificações inadequadas do sistema, ou suposições errôneas de usuários sobre aspectos não documentados. No trabalho de Mikhajlov e Sekerinsk, foi realizado um estudo detalhado do problema que demonstrou que, ainda quando o encapsulamento é violado, modificações consistentes de classes bases são possíveis. Nesse trabalho, foram propostas algumas restrições para disciplinar a utilização da herança:

- Uma nova versão de uma classe base e uma extensão não deveriam introduzir juntamente novas dependências de métodos cíclicas.

- Os métodos de uma subclasse deveriam desconsiderar o fato de que auto-chamadas (*self-recursion*) da classe base podem ser redirecionadas para a própria subclasse. Neste caso, a implementação dos métodos correspondentes na classe base deveria ser considerada ao invés, como se não houvesse nenhum vínculo dinâmico.
- Uma subclasse não pode acessar diretamente o estado de sua classe base, mas somente através de chamadas de métodos da classe base.

O nome "problema de classe base frágil" foi introduzido enquanto se discutia padrões de componentes [Com,Som], uma vez que ele tinha uma importância crucial para sistemas de componentes. Este aspecto do problema foi reconhecido por desenvolvedores do modelo de objetos da *Microsoft* (*COM*) [Com]. Eles observaram que a fonte do problema estava na herança que violava o encapsulamento de dados e optaram por abandonar a herança entre elementos de componentes distintos.

Uma regra geral, seguida por Mikhajlov e Sekerinsk, é que todas as superclasses são abstratas. A regra de superclasses abstratas é assumida no modelo de objetos para software adaptável [Hürsh95]. A regra requer que as classes nos extremos da hierarquia de herança sejam classes concretas, enquanto todas as classes em posições intermediárias devem ser classes abstratas. Somente as classes abstratas podem ser herdadas.

2.2.2 Regras para a Evolução de Hierarquias de Classes

Para aplicar técnicas poderosas como herança e generalização eficientemente, conceitos do mundo real têm que ser encapsulados em classes, a fim de que elas possam ser especializadas ou combinadas num grande número de programas. Estruturas de herança inadequadas, abstrações perdidas na hierarquia podem atrapalhar a reusabilidade de uma coleção de classes.

Linguagens orientadas a objetos fornecem construções simples para encadear hierarquias de classes, permitindo a redefinição de propriedades herdadas. O corpo de um método, por exemplo, pode ser completamente modificado numa subclasse, embora seu nome e assinatura permaneçam idênticos. Dessa forma, é possível implementar versões especializadas ou otimizadas do mesmo método. Tais modificações podem deixar a hierarquia de classes corrente não satisfatória, sendo

necessária a verificação da existência de abstrações perdidas, a fim de tornar as classes mais gerais, aumentar a modularidade e reorganizar, pelo menos em parte, a hierarquia de classes.

Uma solução é reestruturar algoritmicamente a hierarquia ao introduzir novas classes, através da criação de nós intermediários, embaralhando classes entre eles, e reorganizando os caminhos de herança, de modo a evitar a redefinição ou rejeição explícita de classes [Casais89]. Esta conduta trabalha incrementalmente e preserva a estrutura de todas as classes originais, exceto os seus caminhos de herança. Algumas propriedades podem ser reforçadas, como permitir que uma classe seja inserida somente num certo ponto da hierarquia [Casais89]. Reorganizações algorítmicas parecem úteis para detectar abstrações perdidas, para propor generalizações de classes muito especializadas, e para refinar a hierarquia de classes. Contudo, como eles executam transformações estritamente estruturais sobre descrições de objetos, seus resultados requerem a intervenção do usuário para compensar a lacuna de conhecimento referente ao domínio da aplicação e aos conceitos envolvidos na coleção de classes.

Uma conduta formal é descrita em [Borgida88], que propõe um mecanismo para excluir casos anormais, que surgem ao modelar um domínio de aplicação, que não combinem com a hierarquia de classes existente. Essas técnicas são úteis para executar ajustes limitados numa coleção de classes, mas não fornecem quaisquer ajuda para detectar falhas de projeto, podendo rapidamente levar a estruturas de especialização incompreensíveis, carregadas de casos especiais. Tal situação é geralmente uma forte indicação de que a hierarquia não contém as abstrações próprias e que ela deveria ser reorganizada. Este problema também surge na área de banco de dados orientados a objetos. Lá, as técnicas disponíveis [Banerjee+87][Penney+87] primeiro determinam um conjunto de regras de integridade que uma coleção de classes deve satisfazer. Num segundo passo, uma taxonomia de todas as possíveis atualizações é estabelecida. Essas mudanças referem-se à estrutura de classes, como “adicionar um método”, “renomear um método”, ou, “restringir o domínio de uma variável”; ou também podem referir-se à hierarquia como um todo, tal como “omitir uma classe,” ou “adicionar uma superclasse a uma classe.”

Decompor todas as modificações de classes em primitivas de atualização e determinar suas conseqüências traz várias vantagens. Durante o projeto de classes, esta política ajuda os desenvolvedores a detectar as implicações de suas ações sobre a coleção de classes e manter a consistência das especificações de classes. Durante o desenvolvimento da aplicação, ela guia a propagação das mudanças para onde a classe é reutilizada. Por exemplo, renomear uma variável de

instância de uma classe, mudar seu tipo ou definir um novo valor *default*, não tem impacto sobre a aplicação que está usando a classe. Já a mudança ou exclusão de métodos, geralmente leva a mudanças nas aplicações. Contudo, esta conduta limita seu escopo a tipos de evolução primitivos e locais; ela forma um sólido *framework* definindo modificações de classes "bem formuladas", mas não fornece nenhuma dica de quando essas modificações devem ser executadas.

2.2.3 Separação entre Hierarquias de Classes e Tipos

A estrutura de herança, usada para compartilhar código, e a hierarquia de subtipagem conceitual, que têm origem na especialização estão em diferentes níveis de abstração no sistema: a herança está ligada à implementação de classes, enquanto que hierarquia de subtipagem está baseada no comportamento das instâncias (como elas são vistas externamente, pelos outros objetos) [Porter92].

Para simples objetos, cuja função principal é armazenar dados, a diferença prática entre esses dois pontos de vista não é muito grande. Mas, para objetos mais complicados, a distinção entre a interface com o mundo externo e a implementação interna é muito mais importante, e assim, a diferença entre herança e subtipagem é clara. Em muitos casos, os relacionamentos hierárquicos induzidos pela herança e pela subtipagem coincidem, mas este certamente não é sempre o caso: é possível definir uma classe que realmente especializa o comportamento de uma outra classe, mas que emprega uma estrutura totalmente diferente de variáveis e tem código diferente ainda para métodos com o mesmo nome (então, tem-se aqui subtipagem sem herança). Também é possível que a simples adição de alguns métodos, faça com que o comportamento ainda que dos antigos métodos seja modificado numa forma essencial (os novos métodos poderiam modificar às variáveis do objeto de tal modo que uma invariante na qual os antigos métodos confiavam, tenha sido violada). Nesse último caso, não pode ser dito que a nova classe dá origem a um subtipo de uma outra classe já existente, então tem-se aqui herança sem subtipagem.

Dessa forma, os conceitos de herança e subtipagem deveriam ser separados e cada um deveria ser considerado independentemente (é claro que as familiaridades entre os dois conceitos não deveriam ser esquecidas).

2.2.4 Separação entre Interface e Implementação

É largamente aceito que a separação entre *interface* e implementação é uma boa prática de projeto [Porter92]. No lado da *interface*, tipos abstratos de dados são especificados por assinaturas e esses tipos são naturalmente relacionados através de relacionamentos de subtipagem. No lado da implementação, uma classe fornece um esquema de representação e corpos de métodos para cada uma das operações especificadas na assinatura do tipo. Classes são construídas herdando código e representação de outras classes, previamente implementadas. Mas, herança de subtipo é bastante diferente de herança de subclasse (ver Seção 2.2.3).

Tipos especificam protocolos. Em outras palavras, uma definição de tipo mostra quais mensagens serão aceitas pelas instâncias daquele tipo. Uma definição de tipo não inclui informação sobre implementação, tal como nomes de campos ou corpos de métodos. Uma implementação, por outro lado, especifica os nomes das variáveis de instância e, para cada mensagem no tipo, a implementação inclui um corpo de método. É possível que um tipo seja implementado por diferentes classes, em diferentes formas. *Interfaces* são realizações de especificações de tipos em código. Tipicamente, uma especificação de tipo inclui a especificação completa de pré e pós-condições para operações, enquanto que uma *interface* especifica somente a assinatura das operações.

Tipos representam também uma forma de diminuir a dependência entre classes. Classes não têm que referenciar diretamente uma outra classe, a referência deve ser feita através de um tipo (ou seja, através de uma Interface), assim a classe original pode manter a referência para o tipo, embora a classe subjacente possa mudar, aumentando assim a reusabilidade da classe. Como *interfaces* definem conjuntos específicos de métodos, referenciar objetos através de uma interface restringe os métodos que podem ser invocados àqueles definidos pela *interface*. Como uma classe que implementa uma interface deve fornecer implementação para todos os métodos especificados na *interface*, cada *interface* deveria ser tão pequena quanto possível. Uma interface não deveria conter mais do que os métodos que são estritamente necessários para permitir que um objeto se comporte da forma como foi especificado pelo tipo. É melhor que um objeto implemente duas interfaces pequenas do que uma extensa, pois interfaces pequenas são normalmente mais reusáveis.

2.3 Tecnologias de Componentes

Com o crescimento e popularização da Internet, cada vez mais estão sendo desenvolvidos sistemas baseados na *Web*, ou simplesmente sistemas *Web*. Ao contrário do que se pensa, construir sistemas para *Web* não é uma tarefa simples, uma vez que esses sistemas possuem características particulares que tornam o seu desenvolvimento mais complexo do que aplicações *desktop*, tais como:

- Concorrência: Os sistemas Web podem ser acessados por muitos usuários simultaneamente. Isto exige um maior controle por parte da aplicação para garantir que o sistema se comporte de forma eficiente e segura. A definição de métodos que guiem o tratamento da concorrência em sistemas *Web* é essencial para a execução de programas em ambiente concorrente;
- Escalabilidade: Na maioria dos casos é difícil fazer uma estimativa do número de usuários de um sistema *Web*. Muitas vezes esse número pode aumentar ou variar bastante durante o tempo de vida do sistema. Por este motivo, é necessário que estes tipos de sistemas sejam escaláveis, para atender aos usuários de forma satisfatória;
- Segurança: Sistemas *Web* podem ser acessados por usuários remotos em todo o mundo. Isto exige um maior comprometimento com o controle da segurança e integridade da aplicação;
- Disponibilidade: Geralmente os sistemas *Web* são acessados por pessoas de diferentes perfis, em várias partes do mundo, em diferentes horários. Por este motivo, é necessário que os mesmos tenham períodos de indisponibilidade muito curtos.

Diversas tecnologias têm sido propostas na literatura para o desenvolvimento de sistemas modernos, com arquiteturas em camadas e baseadas em componentes. Dentre elas, podemos destacar duas tecnologias mais utilizadas pelos desenvolvedores de software: as plataformas de software J2EE e .NET.

2.3.1 A Plataforma J2EE

O uso da linguagem de programação Java para o desenvolvimento de sistemas *Web* tem se mostrado muito eficaz, por isso empresas do mundo estão adotando esta tecnologia no desenvolvimento de seus sistemas. O motivo principal desta adoção em massa é que a plataforma de desenvolvimento de sistemas *Web* em Java é um padrão aberto. Isto significa que vários fabricantes podem produzir ferramentas de desenvolvimento e infra-estruturas de execução, por exemplo *WebSphere* da IBM [WebSphere].

A plataforma de desenvolvimento de sistemas *Web* de Java chama-se J2EE (*Java 2 Enterprise Edition*). É um ambiente para desenvolvimento e distribuição de aplicações que consiste em um conjunto de serviços, interfaces de programação de aplicação (*APIs*) e protocolos, que oferecem a funcionalidade para o desenvolvimento de aplicações multicamadas, baseadas na *Web*. Sistemas *Web* desenvolvidos em *Java*, mais especificamente na plataforma J2EE são naturalmente multicamadas. Esta plataforma por si só é um modelo de aplicação distribuído e multicamadas [Bambars+02]. J2EE especifica quatro camadas para o desenvolvimento de sistemas *Web* (Figura 3).

Figura 3. Camadas da aplicação J2EE

A camada cliente tem o papel de uma interface de entrada e saída para interação do sistema com os usuários. Esta camada de aplicação é implementada com o apoio de um navegador *Web*, que tem o papel de interpretar e apresentar o conteúdo gerado pela camada de apresentação, geralmente através de um protocolo HTTP [Fielding+99]. Já a camada de apresentação, é a primeira camada do servidor de aplicação e tem o papel de disponibilizar os serviços da Camada de

negócios para o ambiente *Web*, oferecendo conteúdo estático e dinâmico gerado pelos componentes *Web*. A camada de negócios representa o núcleo do sistema. É nela onde estão implementadas as regras de negócio da aplicação. Por fim, a camada de dados, ou persistência, é responsável pelo gerenciamento dos dados do sistema. Ela pode ser vista como a infra-estrutura necessária para o gerenciamento dos recursos da aplicação. O Sistema Gerenciador de Banco de Dados é um exemplo de infra-estrutura localizada nesta camada.

Destas camadas, as únicas que envolvem esforço de programação são as camadas de Apresentação e Negócio. A camada Cliente compreende basicamente as telas do sistema e o navegador *Web*, onde as telas do sistema são geradas pela Camada de apresentação. A camada de dados contempla basicamente o gerenciamento dos dados do sistema.

Na camada de apresentação, J2EE oferece suporte às tecnologias *Servlets* [Coward99] e *JavaServer pages* [Jsp]. Os *Servlets* são programas Java que são executados em resposta a requisições Web. Eles residem no servidor Web (mais especificamente no *WebContainer*[3]) e assim como o servidor Web, eles também recebem requisições HTTP. A diferença é que eles são capazes de gerar conteúdo dinâmico como resposta as suas requisições. *JavaServer pages* é a tecnologia da plataforma J2EE para construção de aplicações que geram conteúdo dinâmico. Uma página JSP é basicamente um documento texto que descreve como processar um pedido e gerar uma resposta. Esta descrição é uma mistura de trechos HTML com algumas ações dinâmicas. JSP é uma linguagem de *script* que é executada no servidor, e o resultado desta execução é uma resposta que pode ser HTML ou qualquer outro tipo de dado suportado pelo protocolo HTTP.

Ainda na camada de apresentação, existe um *framework* chamado *Struts*, que vem sendo muito utilizado pelos desenvolvedores de software. o *Struts* é um *framework* de código aberto [Struts], que é um subproduto do projeto *Jakarta* [Apache] e tem como objetivo auxiliar a construção de aplicações *Web* com *Servlets* e *JavaServer pages*. O *Struts* apoia o desenvolvimento de arquiteturas baseadas no padrão arquitetural *Model-View-Controller* (MVC) [Buschmann+96]. O *Struts* apoia a construção das três entidades do padrão MVC: *Model, View* e *Controller*. *Model* é construído como *JavaBeans* [JavaBeans], que é a entidade que representa os dados da aplicação. *View* é implementado como páginas JSP. Para a construção do *Controller*, é oferecido um componente *Servlet* que é responsável pelo recebimento das requisições, execução da ação

[3] Parte do servidor Web ou servidor de aplicação responsável por gerenciar o ciclo de vida dos *Servlets* [Coward99].

associada à requisição, recuperação e extração da *View*. A ação é definida pelo desenvolvedor como uma classe (*Action class*) que encapsula a execução da requisição.

Já na camada de negócios, o J2EE oferece *Enterprise JavaBeans* [Ejb]. EJBs são componentes reutilizáveis de lógica de negócios para uso em arquiteturas de aplicações distribuídas de múltiplas camadas. A lógica de negócio é o código que satisfaz o propósito da aplicação. O uso de EJBs pode ser considerado quando:

- A aplicação deve ser escalável para acomodar o crescente número de usuários. Pode-se precisar distribuir os componentes de uma aplicação através de múltiplas máquinas numa rede de computadores.
- Transações são requeridas para assegurar a integridade dos dados. EJBs dão apoio a transações para administrar acesso concorrente de dados compartilhados.
- A aplicação tiver uma variedade de clientes. Com poucas linhas de código, clientes remotos podem facilmente localizar os EJBs.

EJBs podem ser de três tipos: *Session bean, Entity bean* e *Message-driven bean.* Um *session bean* representa uma requisição de um único cliente dentro do servidor de aplicação. Para acessar uma aplicação remota, o cliente invoca os métodos do *session bean*. Como o nome sugere, um *session bean* pode ter somente um usuário, que não é persistente, e é finalizado após o término da requisição do cliente. Um *entity bean* representa um objeto de negócio num mecanismo de armazenamento persistente. Tipicamente, cada *entity bean* tem uma relação com um tabela de um banco de dados e, cada instância do *bean* corresponde a uma linha daquela tabela. Por fim, um *message-driven bean* permite que aplicações J2EE processem mensagens assíncronas. Ele age como um servidor que fica esperando pelo recebimento de mensagens para processá-las. As mensagens podem vir de qualquer componente J2EE - uma aplicação cliente, outro *enterprise bean*, ou um componente *Web*.

A distribuição de um componente EJB consiste na preparação e envio de um componente (*bean*) para um servidor de aplicação. Nesse momento, quaisquer recursos que o desenvolvedor utilizou em código devem estar associados a recursos reais, atributos transacionais devem ser especificados, atributos de segurança devem ser definidos, e qualquer outra configuração deve ser especificada através de um descritor. O código compilado e os descritores de distribuição são

empacotados, e disponibilizados no servidor de aplicação. Um servidor de aplicações pode ser considerado como um sofisticado sistema de software que fornece um ambiente de execução para componentes desenvolvidos em Java ou em outras linguagens de programação.

2.3.2 A Plataforma .NET

A plataforma .NET é um ambiente independente de linguagem de programação para a criação de programas que possam interoperar facilmente e de forma segura [DotNet]. Em vez de serem desenvolvidos para uma combinação hardware/software particular, os programas são desenvolvidos para o .NET, e funcionarão em qualquer lugar onde o .NET estiver implementado. .NET oferece uma alternativa de ambiente para produzir aplicações *Web*, executando-as nos mais diversos dispositivos (*PCs*, *Palms*, telefones celulares, etc). A *Microsoft* planeja tornar a infra-estrutura de suporte ao .NET amplamente disponível. Tanto isso é verdade que o *framework* de suporte pode ser baixado e instalado livremente e provavelmente será parte integrante das próximas versões do *Windows*. O *framework* .NET é a nova plataforma de programação para desenvolvimento de aplicações *Windows* e *Web*. Ele é composto de duas partes:

- Um engenho de execução chamado *Common Language Runtime* (CLR);
- Uma biblioteca de classes que disponibiliza as principais funções de programação.

CLR é a base comum a todas as linguagens escritas para a plataforma. O CLR é o ambiente que gerencia a execução de código escrito em qualquer linguagem de programação.

Aplicações .NET não são executadas da mesma maneira que uma aplicação *Windows* tradicional. Ao invés de ser compilada em um executável contendo código nativo, o código das aplicações .NET é compilado para MSIL (*Microsoft Intermediate Language*) e é armazenado em um arquivo chamado *assembly*. Em tempo de execução, o *assembly* é compilado para o seu estado final pela CLR, que proporciona checagem de tipos (*type-safety checks*) e outras tarefas de execução.

Aplicações que executam sob o CLR são chamadas *aplicações de código gerenciado* porque o CLR toma conta de muitas tarefas que normalmente seriam de responsabilidade da aplicação. Com o gerenciamento de código, muitos problemas da programação para *Windows* são

resolvidos, tipicamente o registro de componentes e o versionamento dos mesmos. Isto acontece porque o *assembly* possui todas as informações de que o CLR precisa para executar a aplicação. O CLR cuida dinamicamente do registro de componentes, ao contrário do modelo estático anterior (COM), que necessitava de informações do registro do sistema. As principais vantagens do *framework* são:

- A Aplicação *Web* é compilada. Isto significa que ela será executada mais rápido do que aplicações em linguagens interpretadas;
- Uso da linguagem de programação *Visual Basic*, que foi estendida para suporte total à programação orientada à objetos;
- Introdução à nova linguagem de programação C#, que proporciona uma linguagem de programação fortemente tipada, orientada à objetos, baseada na linguagem C.
- Possibilidade de utilização de um sem número de novas linguagens;

2.4 Resumo

Este capítulo apresentou conceitos e terminologias da área de desenvolvimento de software baseado em componentes, conceitos e técnicas relacionadas com a evolução de sistemas orientados a objetos e tecnologia de componentes. Alguns conceitos importantes serão usados no restante desta dissertação com uma semântica mais restrita. O termo processo, quando usado no contexto deste trabalho, designa um processo de desenvolvimento que especifica e implementa um sistema de software. Quando nos referirmos a componentes, estamos concentrados em sua representação em tempo de projeto, isto é, um componente lógico. A definição de um componente lógico é recursiva, isto é, internamente um componente é formado por sub-componentes e assim recursivamente. Este capítulo mostrou também técnicas de evolução de sistemas orientados a objetos que foram utilizadas para compor a solução apresentada neste trabalho e que serão usadas nesta dissertação. O capítulo também deu uma visão geral sobre as plataformas de desenvolvimento J2EE e .NET que serão utilizadas para descrever os estudos de caso realizados neste trabalho.

Até o momento, apresentamos técnicas e termos que serão utilizados na definição de um modelo de estruturação de componentes de software para sistemas baseados em arquitetura. Os capítulos seguintes se destinam a apresentar e exemplificar tal modelo, além de mostrar como os

conceitos aqui definidos serão empregados para que sistemas baseados em componentes sejam construídos em conformidade com a arquitetura proposta para o software e também como os componentes resultantes tornam-se mais adaptáveis e flexíveis a mudanças.

.

Capítulo 3

Diretrizes de Projeto do Modelo COSMOS

Este capítulo apresenta as diretrizes de projeto incorporadas no modelo COSMOS para construir componentes de software flexíveis a mudanças, adaptáveis e reutilizáveis, e também para mapear as descrições arquiteturais em elementos do modelo de objetos. O capítulo também mostra uma visão geral do modelo COSMOS, explicitando como as diretrizes de projeto foram integradas na técnica de estruturação de componentes e no projeto dos componentes e conectores.

3.1 Diretrizes Adotadas

As diretrizes de projeto incorporadas no COSMOS foram extraídas de pesquisas anteriores em orientação a objetos, arquitetura de software e desenvolvimento baseado em componentes. As diretrizes de projeto documentam o que os desenvolvedores de aplicações baseadas em componentes podem fazer (ou precisam fazer), a fim de implementar sistemas baseados em componentes, a partir de uma descrição arquitetural dos componentes, conectores, interfaces e interações entre eles.

3.1.1 Materialização de Elementos Arquiteturais

Os elementos arquiteturais que compõem a abstração de um sistema, tais como uma coleção de componentes, conectores, interfaces e conexões, devem ser representados, em tempo de execução, como objetos explícitos, formada por um conjunto de objetos que encapsulam toda a informação arquitetural do sistema em tempo de execução (ver Seção 2.1.1). No modelo COSMOS, componentes e conectores são materializados por um conjunto de classes de infra-estrutura, permitindo o mapeamento de uma

arquitetura de software baseada em componentes, em tempo de projeto, para uma implementação. Este mapeamento contribui para reforçar a conformidade da implementação com a arquitetura do sistema, e abre uma perspectiva para o uso de arquiteturas de software dinâmicas, de modo que as informações arquiteturais do sistema possam ser acessadas e modificadas em tempo de execução. Arquiteturas de software dinâmicas definem os componentes, conectores e configurações, que conectam instâncias de componentes e conectores de acordo com as propriedades de estilos arquiteturais (ver Seção 2.1.3).

3.1.2 Inserção de Requisitos não-funcionais nos Conectores

A separação entre requisitos funcionais e não-funcionais é um paradigma de projeto de software que vem sendo considerado para a concepção de grandes sistemas (ver Seção 2.1.1). Neste paradigma, o projeto de uma aplicação é dividido em especificações de requisitos funcionais e não-funcionais. Idealmente, aspectos relacionados com as computações básicas de uma aplicação podem ser tratados independentemente de aspectos não-funcionais, como coordenação, comunicação e distribuição. No modelo COSMOS, requisitos não-funcionais, tais como tolerância a falhas, distribuição e segurança, são inseridos na implementação dos conectores, de modo a simplificar a especificação e a implementação de componentes. Os componentes também podem implementar requisitos não-funcionais, mas a tentativa de isolar ao máximo os requisitos não-funcionais nos conectores torna os componentes mais flexíveis e adaptáveis.

3.1.3 Separação explícita entre Especificação e Implementação

Componentes devem ser projetados de modo que sua especificação e implementação sejam explicitamente separadas (ver Seção 2.1.1). Assim, um componente poderia ser composto de uma parte pública e uma outra privada. A parte privada não é acessível externamente ao componente; ela contém a implementação do componente. A parte pública contém a especificação do componente, que deve ser visível aos seus usuários. No modelo COSMOS, a especificação e a implementação de um componente são organizadas em dois pacotes distintos, um para a implementação e outro para a especificação do componente. Esta separação ajuda na organização do processo de desenvolvimento e assegura que a funcionalidade de um componente possa ser acessada, unicamente através de suas interfaces públicas, uma vez que os elementos que compõem a implementação do componente têm visibilidade interna ao

componente, evitando dependências de implementação e mantendo o encapsulamento da implementação do componente.

3.1.4 Declaração explícita das Dependências

Os serviços dos quais um componente necessita para implementar os seus serviços devem ser explicitamente conhecidos pelos usuários do componente, de modo que, em tempo de execução, possa ser estabelecida uma conexão com um outro componente que provê o serviço do qual aquele componente necessita (ver Seção 1.3). No modelo COSMOS, a especificação do componente inclui interfaces providas e interfaces requeridas. Os serviços requeridos por um componente são especificados nas interfaces requeridas e a implementação do componente desconhece os componentes que implementam tais serviços. Essa distinção entre interfaces providas e requeridas permite que os componentes sejam desenvolvidos independentemente, melhorando a reusabilidade dos componentes.

3.1.5 Separação entre Herança de Implementação e Herança de Tipos

Apesar da componentização ser tipicamente uma atividade orientada à composição de componentes, através de relacionamentos de dependência, a herança entre elementos internos à implementação de um componente, ou até mesmo de componentes diferentes, pode ocorrer visando a reutilização de código já existente. Neste sentido, a herança de implementação não deve resultar incondicionalmente em subtipagem (ver Seção 2.2.3), pois os relacionamentos de subtipagem e herança de implementação podem ser conceitualmente diferentes. No COSMOS, somente classes abstratas podem ser herdadas e apenas as classes concretas podem implementar interfaces. Como resultado, a reutilização de código torna-se mais simples e o comportamento externo dos objetos instanciados fica inteiramente sob o controle de suas classes concretas.

3.1.6 Baixo Acoplamento entre Classes de Implementação

As classes que compõem a implementação de um componente devem conhecer somente as interfaces dos objetos com os quais elas interagem. Classes não devem referenciar diretamente outras classes. As referências devem ser feitas para interfaces, podendo ser mantidas, embora as classes

subjacentes possam ser modificadas ou substituídas. No COSMOS, dependências entre classes de implementação devem ser evitadas (ver Seção 2.2.4). As classes de implementação interagem entre si somente por meio de interfaces e usam uma fábrica de objetos para criar novas instâncias de objetos. Interfaces representam uma forma de diminuir as dependências entre classes e permitem que classes de implementação evoluam independentemente, simplificando a manutenção e aumentando a robustez da implementação do componente.

3.2 Visão Geral do Modelo COSMOS

Num nível arquitetural, um sistema de software é definido em termos de seus componentes, conectores, interfaces e as interações entre eles (ver seção 2.1.3). No modelo COSMOS, as interações entre os componentes são realizadas através de conectores, que ligam as interfaces requeridas de um componente a interfaces providas de outros componentes. Uma interação entre componentes e conectores representa a realização de uma ou mais dependências dos componentes (Figura 4).

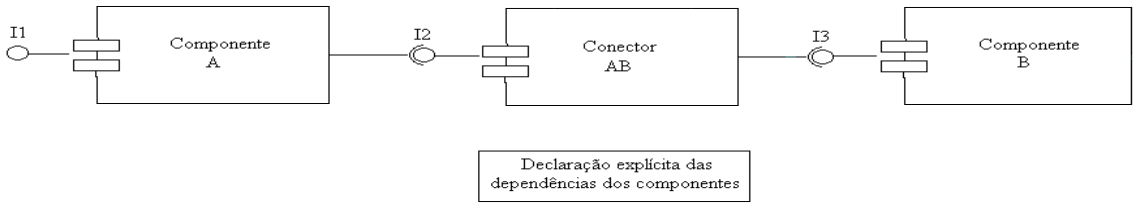

Figura 4. Visão geral do modelo COSMOS - Nível arquitetural

Cada componente e conector são mapeados para o nível de implementação num pacote. Em [UML99], um pacote é definido como um mecanismo de propósito geral para organizar elementos semanticamente relacionados em grupos. Os elementos arquiteturais: componentes, conectores e interfaces são materializadas por um conjunto de classes e interfaces definidas nos pacotes. A definição de dois subpacotes, um para a especificação do componente e um outro para a implementação do componente, separa a especificação do componente de sua implementação e garante o encapsulamento da implementação do componente. Somente os elementos que compõem a especificação do componente são visíveis aos usuários dos componentes.

A especificação do componente é dividida em duas partes: uma para os serviços que o componente requer e uma outra para os serviços que o componente provê. Os serviços que o componente provê são definidos nas interfaces providas do componente e os serviços que o componente requer, para prover os seus próprios serviços, são definidos nas interfaces requeridas do componente. Essa separação aumenta a flexibilidade e adaptabilidade do componente, uma vez que o componente não conhece os demais componentes, que implementam as suas interfaces requeridas. Essa adaptação é realizada pelos conectores que têm conhecimentos das interfaces de todos os componentes e realizam a adaptação de uma interface para outra(s). A implementação de requisitos não-funcionais nos conectores possibilita também que aspectos não-funcionais da interação e coordenação entre os componentes possam ser inseridos ou modificados.

A implementação do componente define algumas classes de infra-estrutura que têm funções específicas no modelo COSMOS, tais como:

(i) Instanciar o componente;
(ii) Materializar o componente em tempo de execução;
(iii) Disponibilizar operações para configurar o componente, associando suas interfaces requeridas a objetos externos que as implementam, e também para acessar os serviços que o componente provê;
(iv) Materializar as interfaces providas do componente;
(v) Diminuir o acoplamento entre as classes de implementação do componente.

Exceto a classe que possibilita a criação de instâncias do componente, ou seja, de objetos que materializam o componente em tempo de execução, todas as demais classes não são visíveis externamente ao componente. A figura a seguir (Figura 5) mostra um esquema geral do modelo COSMOS, com os papéis executados por cada um de seus elementos. O Capítulo a seguir, explica com maiores detalhes o modelo COSMOS e os elementos que compõem a solução proposta.

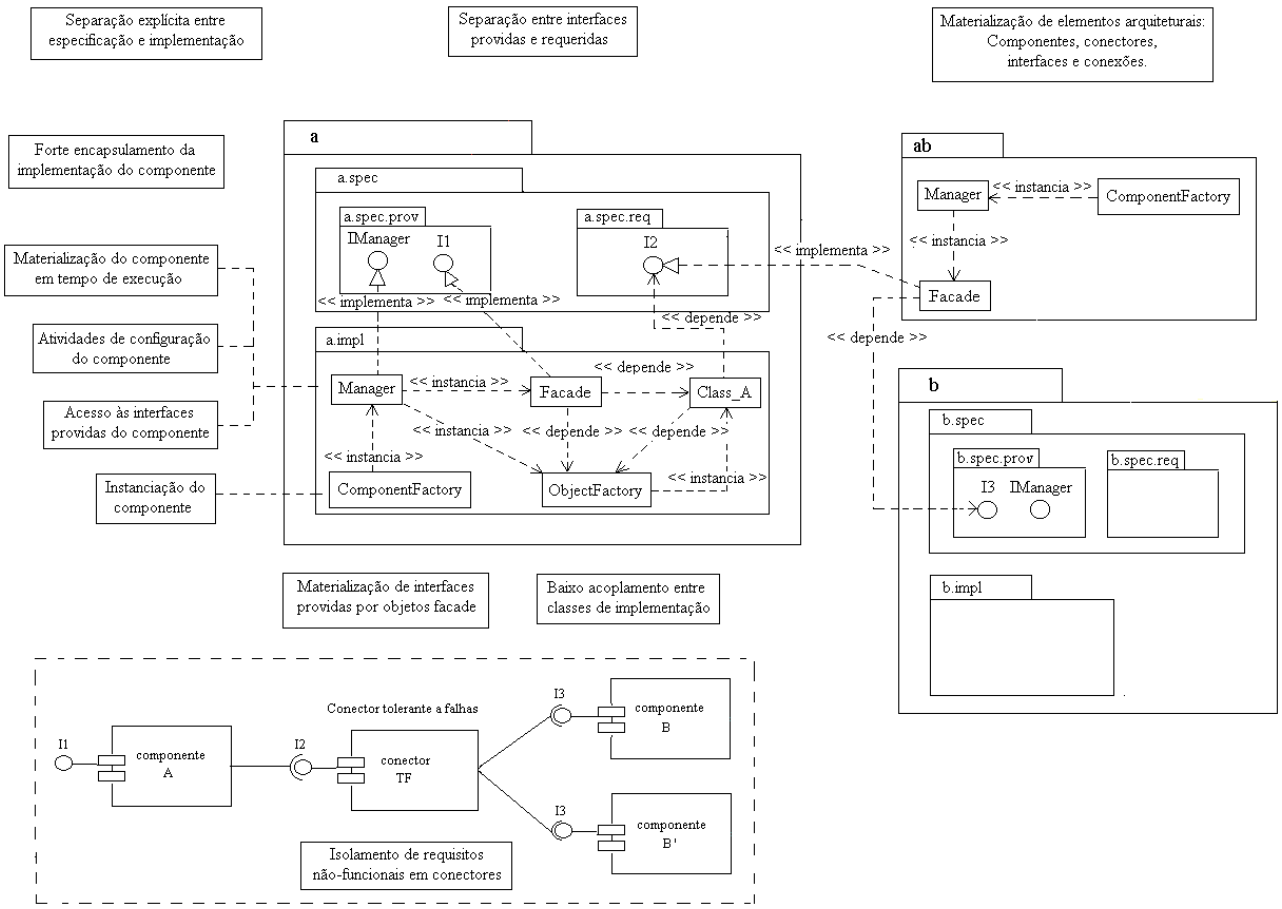

Figura 5. Visão geral do modelo COSMOS - Nível de implementação.

3.3 Resumo

Este capítulo apresentou as diretrizes de projeto incorporadas no modelo COSMOS para a construção de componentes de software flexíveis a mudanças, adaptáveis e também para mapear as descrições arquiteturais baseadas em componentes em elementos do modelo de objetos. A flexibilidade a mudanças é promovida pela separação explícita entre a especificação e a implementação do componente, pela separação entre herança de implementação e herança de interface e também pelo baixo acoplamento entre classes de implementação dos componentes. Já a adaptabilidade dos componentes é aumentada pela incorporação de requisitos não-funcionais nos conectores, pela declaração explícita das dependências dos componentes e pelo uso de conectores para implementar essas dependências. Por fim, para facilitar o mapeamento das descrições

arquiteturais baseadas em componentes em elementos do modelo de objetos, o modelo COSMOS define algumas classes com funções estruturais que serão descritas no capítulo seguinte.

Também foi apresentada uma visão geral do modelo COSMOS, explicitando como essas diretrizes de projeto são aplicadas no modelo. A integração dessas diretrizes de projeto, idéias e conceitos de pesquisas anteriores em arquitetura de software, orientação a objetos e desenvolvimento de software baseado em componentes é uma das principais contribuições deste trabalho.

Capítulo 4

COSMOS - Um Modelo de Estruturação de Componentes para Sistemas Orientados a Objetos

Este capítulo apresenta um modelo para estruturar componentes de software em sistemas baseados em arquitetura chamado de COSMOS. COSMOS é um modelo que se concentra principalmente na estruturação, projeto e implementação dos componentes e conectores de um sistema, visando garantir a conformidade da arquitetura proposta para o sistema com relação a sua implementação. O modelo COSMOS também visa construir componentes de software que sejam mais flexíveis a mudanças e mais facilmente adaptáveis e reutilizáveis.

4.1 Introdução

A unidade fundamental do projeto de um componente é um pacote (ver Seção 2.1.1). No modelo COSMOS, um componente arquitetural é mapeado em um pacote contendo dois subpacotes: o pacote de especificação, que descreve a visão externa do componente, e o pacote de implementação, que descreve como o componente é implementado (Figura 6).

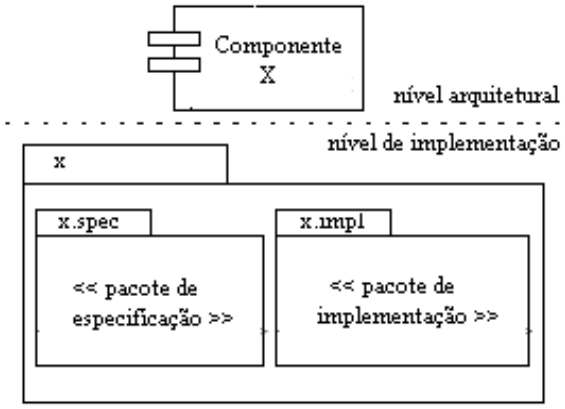

Figura 6. Representação de componente

O pacote de especificação contém um conjunto de interfaces públicas que podem ser providas ou requeridas pelo componente. As interfaces são organizadas em dois subpacotes distintos, um para as interfaces providas (*spec.prov*) e outro para as interfaces requeridas (*spec.req*), e compreendem toda a informação necessária para integrar um componente numa composição de software e acessar os seus serviços. O pacote de implementação (*impl*) contém classes que implementam os serviços providos pelo componente e, opcionalmente, também pode conter interfaces auxiliares, internas ao pacote de implementação (Figura 7). Como regra geral, as classes de implementação e interfaces auxiliares têm visibilidade restrita ao pacote de implementação (*friendly classes* e *friendly interfaces*) e permanecem totalmente escondidas dos clientes do componente. A única exceção a essa regra é uma classe pública que implementa uma operação para instanciação do componente e deve ser usada pelos clientes do componente.

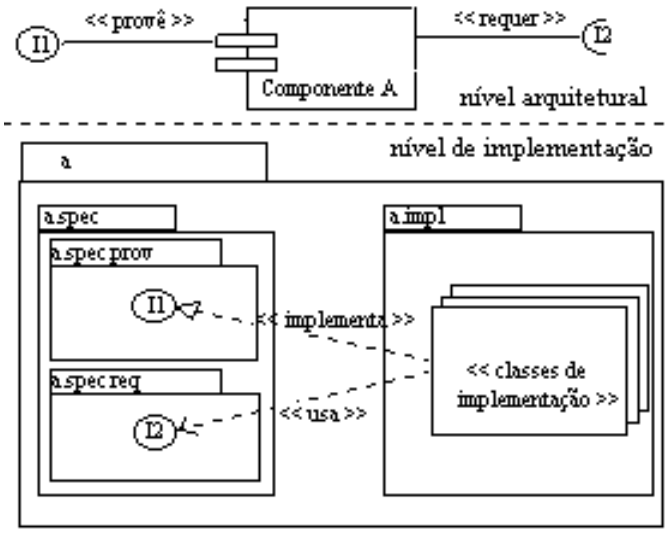

Figura 7. Estrutura interna do componente

Um conector arquitetural é mapeado em um pacote que contém classes que materializam as conexões entre os componentes, também chamadas de conexões de interface (Figura 8). Uma conexão de interface representa uma ligação entre uma interface requerida por um componente e uma ou mais interfaces providas por outros componentes [Luckham+00]. A princípio, o mapeamento de um conector arquitetural, diferentemente do mapeamento feito para um componente arquitetural, não prevê a existência de um pacote de especificação, ou seja, um conector não define novas interfaces públicas. Dessa forma, um conector apenas usa interfaces definidas pelos componentes com os quais ele interage, tendo sua estrutura interna similar ao pacote de implementação de um componente, com seus elementos inacessíveis externamente. No modelo COSMOS, um conector é também considerado como um componente, porém, um componente mais simplificado, cuja existência se dá pela necessidade de conectar outros componentes.

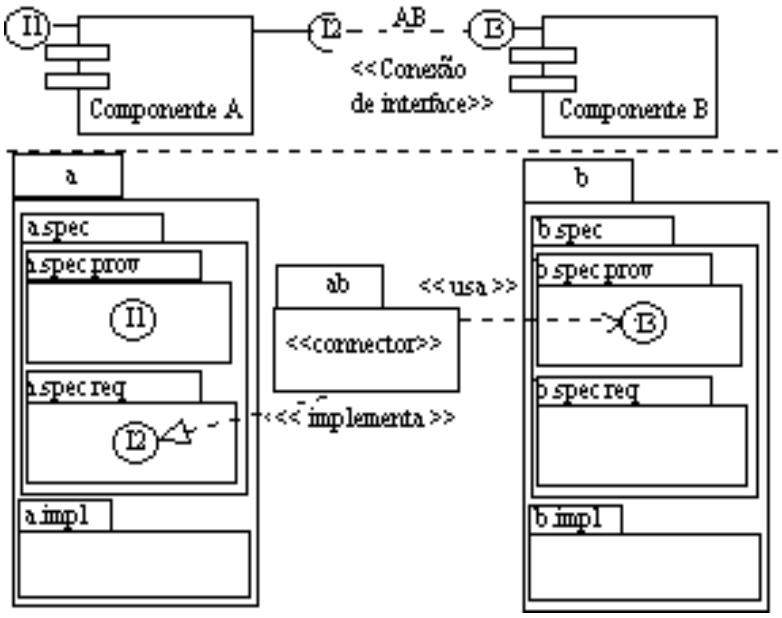

Figura 8. Componentes e conectores

Os pacotes de especificação e implementação, que descrevem um componente arquitetural, e os conectores e conexões de interface, responsáveis pelas inter-conexões entre componentes, compõem o modelo proposto neste trabalho. A nossa solução é formada por três modelos inter-relacionados: (1) o modelo de especificação, que define a visão externa do componente e corresponde ao pacote de especificação; (2) o modelo de implementação, que descreve como o componente é implementado e corresponde ao pacote de implementação; (3) e o modelo de conectores, que descreve as conexões entre

componentes, realizadas através de conectores, e corresponde aos componentes, conectores e conexões de interface.

4.2 O Modelo de Especificação

A separação entre especificação e implementação (ver diretriz de projeto Separação entre especificação e implementação - seção 3), além de organizar o processo de desenvolvimento, possibilita a divisão de um componente em uma parte pública e uma parte privada. A parte pública corresponde à especificação do componente, enquanto que a parte privada corresponde à implementação do componente. A especificação do componente compõe o modelo de especificação que define o comportamento externo de um componente. O modelo de especificação é formado por um conjunto de interfaces públicas que podem ser providas ou requeridas por um componente. As interfaces providas definem serviços que o componente implementa e disponibiliza aos seus clientes, enquanto que as interfaces requeridas definem os serviços que o componente necessita, para implementar os seus próprios serviços, e que devem ser providos ao componente, em tempo de execução, por outros componentes dentro de uma mesma composição de software.

Um componente é caracterizado pelas interfaces que ele provê ou requer. Portanto, o projeto das interfaces de um componente torna-se um importante aspecto de sua especificação. Características como abstração, legibilidade e acoplamento são extremamente importantes. As interfaces devem oferecer um número pequeno de operações, ou seja, somente o comportamento estritamente relevante deve ser disponibilizado aos clientes do componente. Além do número de operações, outro aspecto importante é o nível de abstração das informações manipuladas por operações de interfaces. A utilização apenas de tipos primitivos nos parâmetros e valores de retorno de operações de interfaces melhora a adaptabilidade, mas em compensação, diminui o acoplamento do componente. Enquanto que a utilização de tipos mais abstratos aumenta o poder de legibilidade, consistência e confiabilidade do componente. Como conseqüência, os serviços disponibilizados numa interface provida podem ser compostos por serviços providos por interfaces internas ao componente e, assim, um objeto que realiza uma interface provida pode ser visto como um objeto *facade*, que provê uma interface única e simplificada para um conjunto de interfaces complexas e de nível mais baixo [Gamma+95], facilitando a utilização do componente e minimizando a comunicação externa com o mesmo.

Os objetos *facades* implementam as interfaces providas por componentes, mas apenas redirecionam as chamadas de métodos para classes de implementação internas aos componentes que, de fato, implementam os serviços que os componentes provêm a seus clientes. Neste sentido, os objetos *facades* materializam as interfaces providas pelo componente (ver diretriz de projeto Materialização de elementos arquiteturais - seção 3) e, como um componente pode ter mais de uma interface provida, cada uma delas pode ser implementada por um objeto *facade* diferente. No modelo proposto, os objetos *facades* são usados por instâncias de conectores para estabelecer as conexões entre componentes, ou seja, antes de ser usado, os componentes devem ter suas interfaces requeridas associadas a objetos *facades* de outros componentes, que provêm serviços dos quais eles necessitam.

Além das interfaces providas e requeridas, um componente também deve prover meta-informação a seu respeito, ou seja, toda informação necessária para reutilizar o componente. Primeiramente, um componente deve disponibilizar informações sobre os serviços que ele provê e, posteriormente, informar sobre os serviços que o componente requer [DeRemer+76]. Disponibilizar essa informação através de meta-informação no próprio componente, ao invés de uma documentação anexa, torna essa informação disponível para ferramentas de composição e execução de componentes. As ferramentas podem checar se dois componentes podem ser conectados sem ter acesso ao código-fonte dos componentes. Similarmente, repositórios de componentes podem usar essa informação para buscar e recuperar componentes, validando requisitos de clientes contra a meta-informação dos componentes.

O pacote de especificação contém uma interface que deve ser provida por todos os componentes e que provê serviços que permitem a extração de meta-informação a respeito do componente, mais especificamente os nomes das interfaces providas e requeridas pelo componente, a configuração e o uso do componente. Essa interface é chamada de *IManager*. Um objeto do tipo *IManager* materializa o componente em tempo de execução. A interface *IManager* é um tipo de administrador de interfaces. Por meio dela, o usuário do componente pode ter acesso às interfaces providas e requeridas pelo componente, recuperar os objetos *facades* que implementam as interfaces providas pelos componentes e configurar as instâncias de objetos *facades* que implementam suas interfaces requeridas, através de conectores (Figura 9).

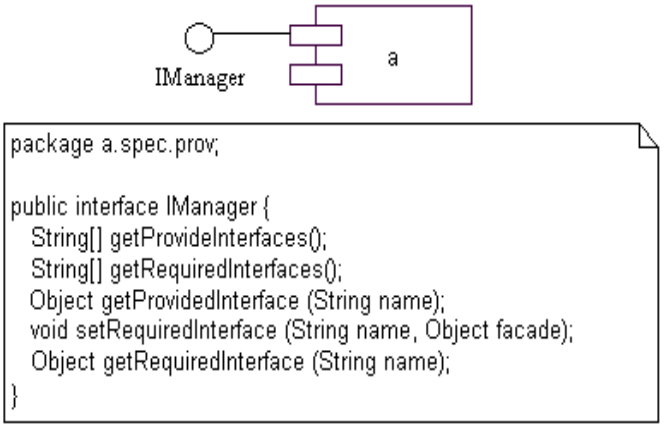

Figura 9. A interface *IManager*

As operações *getProvidedInterfaces()* e *getRequiredInterfaces()* recuperam os nomes das interfaces providas e requeridas pelo componente, respectivamente. A operação *getProvidedInterface(String name)* retorna um objeto *facade*, que implementa uma determinada interface provida pelo componente. A operação *setRequiredInterface(String name, Object ob)* associa uma interface requerida a um dado objeto *facade*, provido por outro componente. Antes de obter uma interface provida por um componente, todas as suas interfaces requeridas devem ter sido, anteriormente, associadas a objetos *facades* providos por outro(s) componente(s). Finalmente, a operação *getRequiredInterface(String st)* retorna o objeto *facade*, anteriormente associado com uma interface requerida.

No modelo COSMOS, as interfaces providas e requeridas pelos componentes são imutáveis. Essa idéia é derivada do modelo de componentes da *Microsoft* COM [Com]. Neste caso, o conjunto de interfaces providas por um componente não pode ser reduzido, apenas acrescido, entre diferentes versões do componente. No caso de *Java*, podem ser emitidos avisos em tempo de compilação, contra o uso de certas interfaces ou classes, declarando as operações como *deprecated*. Isso assegura que as interfaces continuem válidas, de forma que a integridade do código existente seja preservada. Esse requisito torna necessária a existência de um mecanismo que permita ao cliente do componente verificar se uma determinada interface é provida por um componente particular. Esse mecanismo constitui-se na interface *IManager*, descrita anteriormente.

4.3 O Modelo de Implementação

O modelo de implementação define como os serviços providos por um componente são implementados. Cada mapeamento de componente inclui um pacote, chamado *impl*, que contém classes de implementação e, opcionalmente, um conjunto de interfaces auxiliares, utilizadas internamente na implementação do componente. As interfaces internas à implementação do componente têm visibilidade restrita ao pacote de implementação e podem ser subtipos de interfaces providas pelo componente. Esse relacionamento de herança pode ocorrer numa situação na qual se deseja que o conjunto de operações de uma interface pública seja menor do que aquele disponível aos elementos internos à implementação do componente. Essa decisão pode ser tomada para proteger o componente de eventuais inconsistências, expondo operações que podem contribuir para tais inconsistências, ou simplesmente, para não expor operações que não são de interesse externo ao componente.

As classes de implementação têm visibilidade restrita ao pacote de implementação e definem as estruturas de dados e as operações necessárias para implementar as interfaces providas pelo componente. O modelo COSMOS também propõe algumas classes que definem serviços de infra-estrutura dos componentes. Tais classes possibilitam a materialização de componentes arquiteturais e propiciam a flexibilidade e evolutibilidade dos componentes. O pacote de implementação define obrigatoriamente quatro classes de infra-estrutura que desempenham funções específicas no modelo (Figura 10): (i) *ComponentFactory*, responsável pela instanciação do componente; (ii) *Manager*, que realiza a interface *IManager*, utilizada para integrar instâncias de componentes dentro de uma composição de software; (iii) *Facade*, que implementa uma interface provida; e (iv) *ObjectFactory*, responsável pela criação de instâncias de classes de implementação.

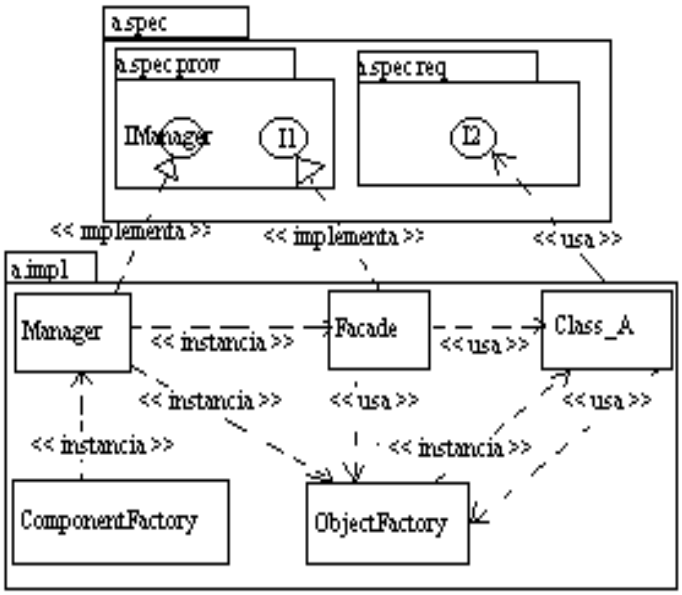

Figura 10. O modelo de implementação

A classe *ComponentFactory* representa a porta de entrada do componente. Ela define um mecanismo para instanciação do componente, implementando um único método *createInstance()* para criar instâncias do componente. A classe *ComponentFactory* é a única classe pública do modelo de implementação e o cliente do componente não precisa instanciá-la, uma vez que ela define um único método estático. O método *createInstance()* tem como retorno um objeto do tipo *IManager*, que materializa uma instância do componente em tempo de execução. A classe *Manager* implementa a interface *IManager* e é instanciada pelo método *createInstance()* da classe *ComponentFactory*. Ela implementa operações para configuração e uso do componente e mantém as referências para os objetos *facades* que implementam as interfaces providas e requeridas pelo componente.

Cada instância da classe *Manager* representa um objeto que materializa o componente em tempo de execução, ou seja, uma outra instância do componente (ver diretriz de projeto Materialização de elementos arquiteturais - seção 3). A meta-informação sobre o componente é representada e armazenada de forma a ser facilmente acessada, através de suas operações, ou através de mecanismos reflexivos. A classe *Manager* também é responsável por manter uma referência para a fábrica de objetos (*ObjectFactory*) e passar essa referência, quando necessário, para as classes de implementação, através da classe *Facade*.

Uma classe *Facade* define um ponto de acesso a uma interface provida pelo componente, delegando funcionalidade para classes de implementação apropriadas [Gamma+95]. Cada interface

provida por um componente deve ser realizada por uma classe *Facade*. Uma interface provida pode ser materializada por duas ou mais classes de implementação usadas pela classe *Facade* para compor suas operações. Para acessar os serviços das classes de implementação, para as quais a classe *Facade* delega as operações, o construtor da classe *Facade* recebe como parâmetro a instância corrente da classe *Manager* e obtém a instância da classe *ObjectFactory*, criada anteriormente.

A fim de aumentar a manutenibilidade e evolutibilidade da implementação do componente, dependências diretas entre classes de implementação devem ser evitadas (ver diretriz de projeto Baixo acoplamento entre classes de implementação - seção 3). A interoperação entre classes de implementação deve ser feita através das interfaces auxiliares e, quando necessário, novos objetos são criados por uma fábrica de objetos. No modelo COSMOS, as regras executadas pelos objetos são representadas explicitamente através de interfaces, e as referências diretas entre classes devem ser evitadas. Para isso, adotou-se o uso de uma fábrica de objetos, a classe *ObjectFactory*, que define operações para criar instâncias de objetos internos à implementação do componente. O uso de uma fábrica de objetos flexibiliza a implementação do componente, uma vez que apenas as interfaces das classes são conhecidas e as classes de implementação respectivas podem ser modificadas mudando somente a implementação da classe *ObjectFactory*. Como resultado, o acoplamento da implementação do componente é reduzido e o esforço de manutenção também é reduzido. Um outro benefício do uso da fábrica de objetos é a possibilidade de versionamento da implementação do componente, através da manutenção das diferentes versões da fábrica de objetos.

As classes de implementação internas ao componente podem precisar de serviços providos por outros componentes para implementar os seus próprios serviços. Tais serviços são definidos nas interfaces requeridas pelo componente. Esse alto nível de abstração resulta em maior flexibilidade do componente, mas também resulta em maior complexidade na implementação e no uso do componente. Para que uma classe de implementação interna possa usar um serviço de uma interface requerida, é necessário que o objeto que a realiza tenha sido associado à interface requerida, através da operação *setRequiredInterface()*, implementada pela classe *Manager*. Neste ponto, esse objeto é armazenado pelo objeto do tipo *IManager* e a instância corrente da classe *Manager* é propagada até o objeto que necessita acessar o serviço da interface requerida, de modo que ele possa chamar a operação getRequiredInterface() e recuperar o objeto desejado.

4.4 O Modelo de Conectores

O modelo de conectores descreve as conexões entre um conjunto de interfaces providas e interfaces requeridas, permitindo assim a interação de dois ou mais componentes numa composição de software. Um conector arquitetural é materializado como um pacote, cuja estrutura é similar ao pacote de implementação de um componente. A estrutura interna de um conector é composta pela classe *ComponentFactory*, a classe *Manager* e uma classe *Adapter* (Figura 11).

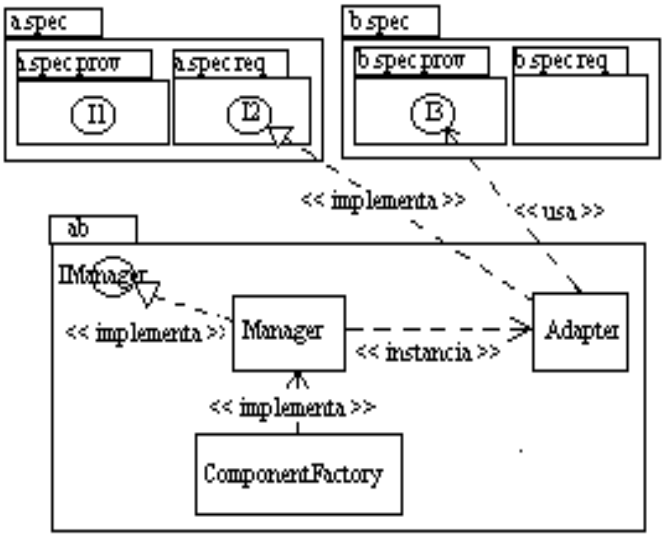

Figura 11. O modelo de conectores

As classes *ComponentFactory* e *Manager* têm as mesmas funções descritas no modelo de implementação (ver seção 4.3). Para estabelecer as conexões de interface que determinam as ligações entre componentes, um conector usa interfaces providas e requeridas dos componentes que irão participar da composição de software. Portanto, um conector não define novas interfaces públicas, apenas usa interfaces públicas definidas pelos componentes que participam da interação, inclusive a interface *IManager*. As conexões de interface podem ser simples (*call-return*), ou seja, entre um componente cuja interface requerida define um conjunto de serviços que é um subconjunto de serviços definidos numa interface provida por outro componente, ou complexas, entre um componente cuja interface requerida define um conjunto de serviços que são implementados por serviços de componentes diferentes (Figura 12). As conexões de interface são implementadas por uma classe, seguindo o padrão de projeto *Adapter*

[Gamma+95], permitindo que componentes com interfaces incompatíveis interajam indiretamente por meio de um conector.

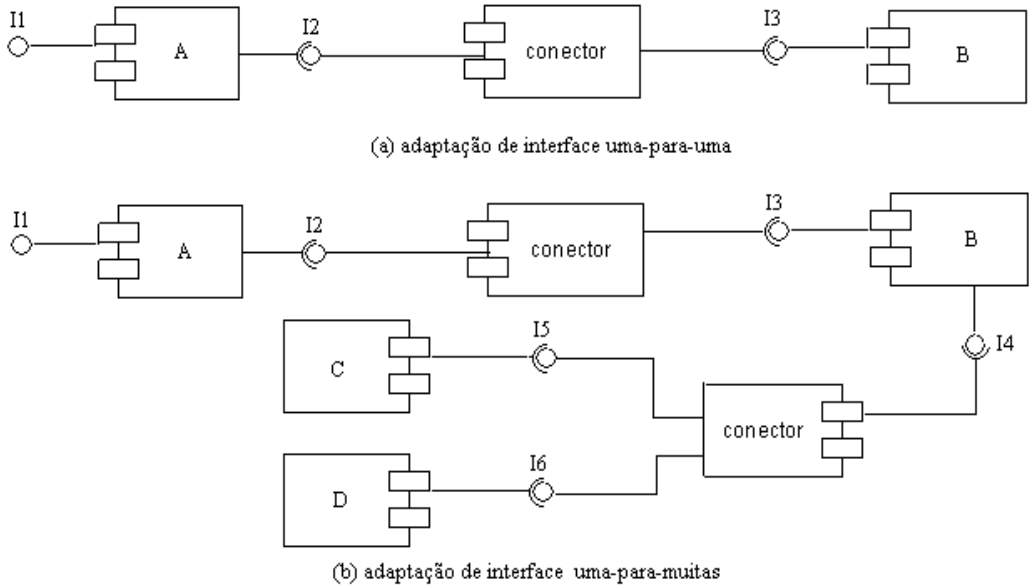

Figura 12. Tipos de conexões de interface

A composição de componentes é uma atividade que pode ser dividida em aspectos estáticos e dinâmicos [Lüer+01]. Composição estática é baseada nas conexões entre interfaces providas e requeridas. Composição dinâmica, contudo, está relacionada com requisitos não funcionais de uma configuração de software, que são extraídos dos componentes e precisam ser tratados, adicionalmente à funcionalidade providas pelos componentes. Neste sentido, a proposta de [Lüer+01] diz que os conectores que mediam as conexões deveriam ser promovidos à categoria de componentes. Porém, conectores devem ser essencialmente abertos e configuráveis, de modo que os aspectos não-funcionais da interação e coordenação entre componentes possam ser inseridos ou modificados. A Figura 13 ilustra o uso de conectores para encapsular requisitos não funcionais.

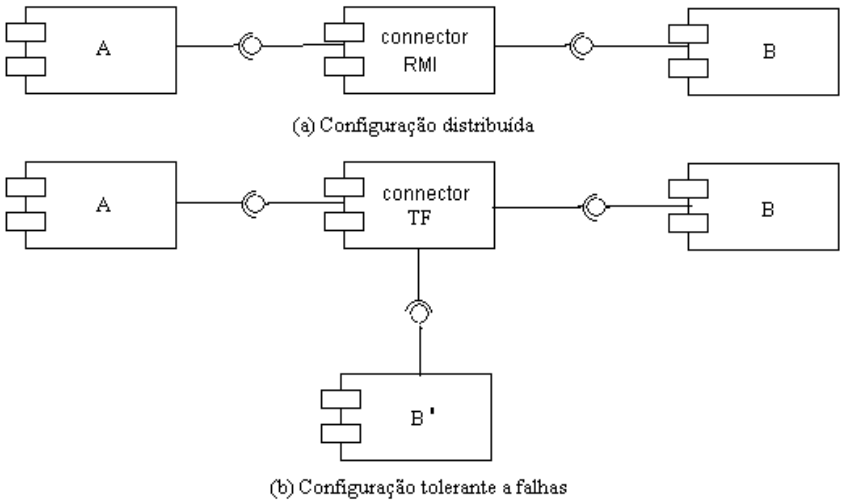

Figura 13. Requisitos não-funcionais em conectores

A figura 13(a) mostra a conexão entre componentes distribuídos, onde os aspectos relacionados com o protocolo de distribuição são encapsulados num conector, por exemplo RMI [Rmi]. A figura 13(b) mostra uma conexão tolerante a falhas entre um componente cliente e um par de componentes replicados, através de um conector. A idéia é que o mínimo de modificações seja necessário aos componentes para atender aos requisitos não-funcionais incorporados. Neste caso, além da classe *Adapter*, o conector pode requerer classes de implementação adicionais para, por exemplo, enviar mensagens a todos os componentes e tomar decisões para aumentar a confiabilidade dos componentes, comparando as respostas obtidas e, gerando um histórico para excluir os componentes não confiáveis. Os componentes também podem ser construídos implementando diferentes políticas e algoritmos, permitindo ao conector escolher os mais apropriados para cada situação.

4.5 O Meta-Modelo

O *meta-modelo*, ou modelo geral (Figura 14), define conceitualmente os elementos que fazem parte dos modelos de especificação, implementação e conectores, as relações entre eles, e também a instanciação e configuração de um componente.

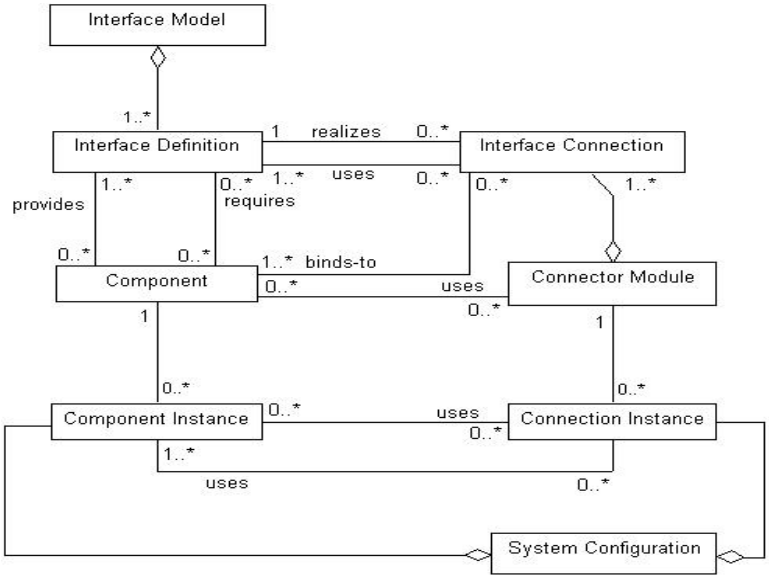

Figura 14. O Meta-Modelo

O modelo de especificação agrega um conjunto de definições de interfaces públicas, que podem ser providas ou requeridas por um componente. Uma definição de interface descreve um conjunto de características de um componente, junto com a especificação do seu comportamento.

Um componente corresponde a um Tipo Abstrato de Dados (TAD), que possui uma estrutura e um comportamento. Essa estrutura é formada a partir de elementos de dados de outros tipos menos abstratos, que podem ser outros TADs ou tipos concretos (nativos) providos pelo ambiente de execução. No modelo de especificação, são definidos TADs que o componente implementa (interfaces providas), TADs que o componente requer (interfaces requeridas) e TADs mais elementares, que por sua vez, são implementados através de outros TADs e/ou tipos nativos, e assim recursivamente.

O modelo de conectores é formado por componentes conectores e conexões de interface. Cada componente usa um conjunto de zero ou muitos conectores e um conector é usado por zero ou muitos componentes. Um conector especifica um conjunto de conexões de interface. Conexão de interface é uma associação entre uma característica, r, de uma interface requerida e uma característica, p, de uma interface provida, tal que qualquer uso de r é associado com um uso correspondente de p. No modelo COSMOS, uma conexão de interface representa uma ligação entre componentes, materializada através de conectores. Neste caso, uma conexão de interface é responsável pela realização de uma interface por meio de vínculos com outros componentes.

Do ponto de vista de execução, o *System Configuration* é responsável pela instanciação e manutenção de instâncias de componentes e conectores. Para isso, ele pode encapsular informações utilizadas na configuração dos componentes e conectores.

4.6 Resumo

Este capítulo apresentou um modelo de estruturação de componentes para sistemas baseados em arquitetura chamado de COSMOS. O objetivo deste modelo é habilitar o desenvolvimento de sistemas, auxiliando no mapeamento de descrições arquiteturais, baseadas em componentes, em elementos do modelo de objetos e promover a reutilização, adaptabilidade e flexibilidade dos componentes construídos. O modelo COSMOS contém atividades estritamente necessárias para projetar os componentes e conectores e estruturar a interação entre os componentes, através de conectores.

Para o projeto dos componentes e conectores, a solução define o modelo de especificação e o modelo de implementação. O modelo de especificação define o comportamento externo dos componentes, isto é, as suas interfaces providas e requeridas, que devem ser acessadas pelos usuários dos componentes. Apesar de terem sido elevados à categoria de componentes, a princípio, os conectores não têm modelo de especificação. Os componentes conectores, ou simplesmente conectores, apenas usam interfaces providas e requeridas dos componentes para realizar as adaptações. Já o modelo de implementação, descreve a implementação dos componentes e conectores, isto é, suas classes de implementação, classes com funções estruturais específicas do modelo e, opcionalmente, interfaces internas à implementação dos componentes.

O modelo de conectores descreve a interação entre os componentes, através dos conectores. A declaração das dependências dos componentes em interfaces requeridas e o uso dos conectores, para realizar a adaptação entre uma interface requerida de um componente e interfaces providas de outros componentes, aumenta a adaptabilidade e reusabilidade dos componentes.

Destacamos também a inserção de requisitos não-funcionais na implementação dos conectores. Isso facilita o projeto dos componentes, aumenta a sua reusabilidade e possibilita a modificação, ou a inserção de novos requisitos não-funcionais no sistema, com um impacto menor na implementação dos componentes, isolando ao máximo as modificações nos conectores do sistema.

O modelo COSMOS é genérico o suficiente para ser aplicado a sistemas que serão implementados em diferentes linguagens de programação e tecnologias de componentes. No capítulo seguinte, mostraremos a aplicação do modelo em dois estudos de caso de sistemas implementados usando a linguagem de programação Java e a tecnologia de componentes J2EE.

Capítulo 5

Avaliação Prática do Modelo COSMOS

Neste capítulo, apresentamos dois estudos de caso da aplicação do modelo COSMOS. Estes estudos de caso foram realizados objetivando a aplicação do modelo em aplicações reais e foram conduzidos por diferentes equipes de projetistas e desenvolvedores.

O primeiro estudo de caso foi realizado por uma empresa privada de desenvolvimento de sistemas de BioInformática. Esse estudo de caso visou à inserção de um processo de desenvolvimento bem-definido e centrado em arquitetura de software nas atividades de desenvolvimento de software da empresa. O modelo COSMOS foi utilizado no mapeamento do projeto arquitetural dos componentes do sistema para a implementação. Uma equipe de projetistas e desenvolvedores da própria empresa realizaram o estudo de caso, customizando o modelo COSMOS de acordo com as suas necessidades, isto é, não o aplicando em sua totalidade, como apresentado aqui nesta dissertação.

O segundo estudo de caso visou à aplicação completa do modelo COSMOS num outro sistema real, de modo que todas as suas características pudessem ser aplicadas e avaliadas. Neste segundo estudo de caso, foi utilizada a especificação de um sistema Web de informações sobre as condições de conservação das rodovias federais, que tem o objetivo de estabelecer um canal de comunicação entre usuários das rodovias e os responsáveis pela operação e conservação dessas vias.

5.1 Estudo de Caso: O Sistema de BioInformática

Neste estudo de caso, o modelo COSMOS foi aplicado no desenvolvimento de um sistema real, por uma empresa de desenvolvimento de software que enfrenta diariamente os problemas do desenvolvimento de software, tais como: cumprimento dos prazos acordados com os clientes, mudanças nos requisitos etc. A liberdade da empresa na aplicação do modelo fez com que o mesmo fosse customizado de acordo com as necessidades da empresa, que não o aplicou em sua totalidade. Porém, essa customização não deixou de nos dar bons resultados. Cláusulas contratuais nos impossibilitam de mostrar os documentos, modelos e o código-fonte resultante deste estudo de caso. Por isso, vamos descrevê-lo informalmente e comentar os seus principais resultados.

5.1.1 O Projeto Arquitetural

A definição dos componentes e interfaces do sistema seguiu o processo definido em *UML Components* (ver seção 2.1.2). Assim, foram definidos os componentes da camada de apresentação, aplicação, sistema e negócio. A arquitetura do sistema seguiu o seguinte modelo em camadas (Figura 15):

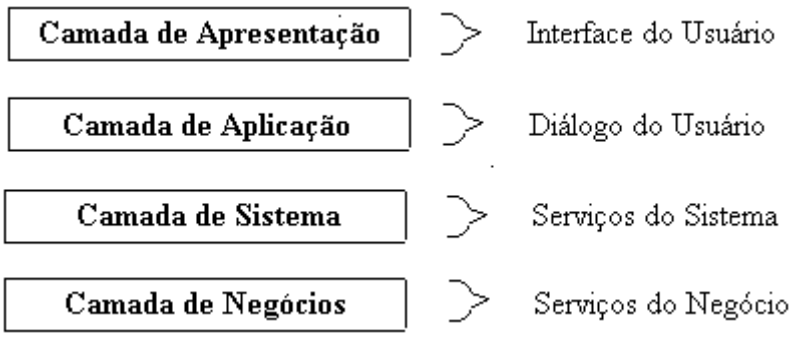

Figura 15. Arquitetura em camadas do sistema de BioInformática

A camada de apresentação define o que o usuário do sistema vê. Ela contém componentes que gerenciam a lógica da interface com o usuário. Os componentes da camada de aplicação gerenciam o estado do diálogo com o usuário e podem ser usados por muitos componentes da camada de apresentação. A camada de sistema define componentes que combinam os serviços do

negócio para construir uma aplicação. Por fim, os componentes da camada de negócios definem as regras, informações e transformações de negócio.

Neste estudo de caso, os componentes das camadas de apresentação, aplicação, sistema e negócio interagem diretamente entre si. A interação entre os componentes é unidirecional, da camada de apresentação até a camada de negócios. Assim, o projeto arquitetural não contém conectores entre os componentes (Figura 16).

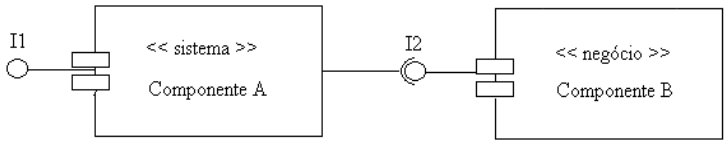

Figura 16. Exemplo de interação entre componentes do sistema de BioInformática

5.1.2 A Implementação

O sistema foi implementado em Java. Quando dizemos que o modelo COSMOS foi customizado de acordo com as necessidades da empresa, significa que algumas características do modelo não foram exploradas e, consequentemente, alguns elementos do modelo não foram aplicados. Como a empresa objetivava a inserção de um processo de desenvolvimento de software bem-definido na sua rotina diária de desenvolvimento de software, os benefícios do modelo que mais interessavam à empresa eram a organização do processo de desenvolvimento componentizado e as regras para o mapeamento da arquitetura de componentes para a implementação. Características como flexibilidade e adaptabilidade e facilidade de manutenção dos componentes não eram fundamentais para este estudo de caso, ou não foram exploradas. A Figura a seguir mostra um exemplo de interação entre dois componentes, seguindo o modelo COSMOS customizado para este estudo de caso (Figura 17).

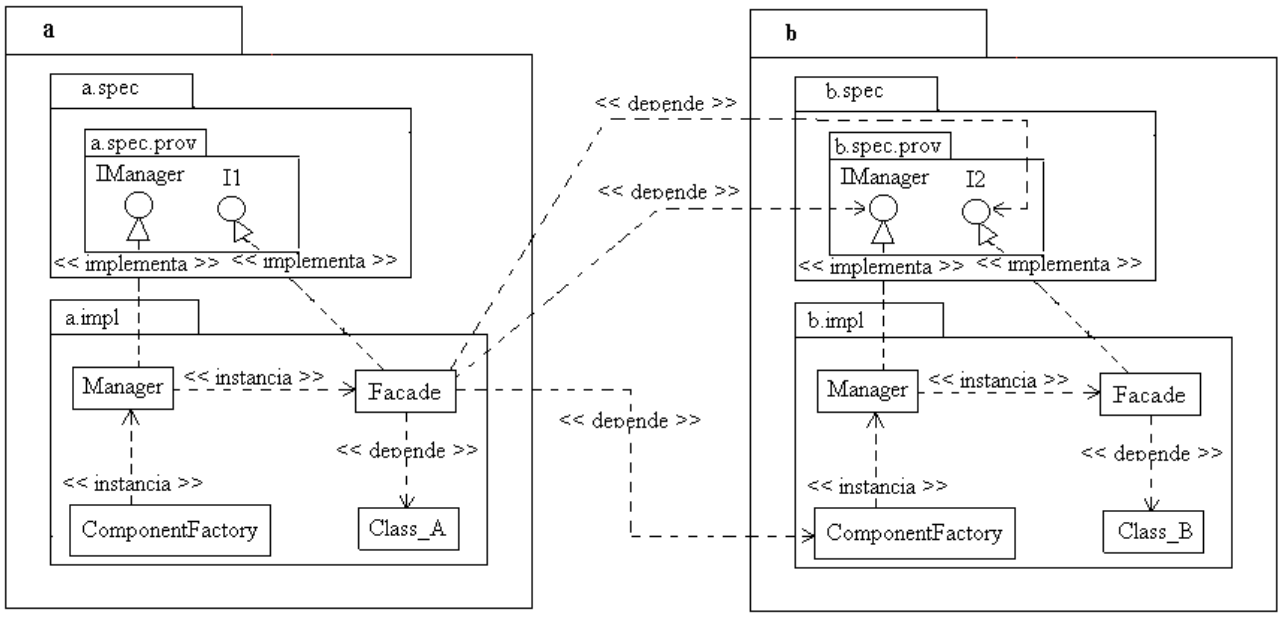

Figura 17. Modelo COSMOS customizado para o sistema de BioInformática

A mudança mais clara é a ausência dos conectores para mediar as interações entre os componentes. Consequentemente, cada componente deve conhecer os demais componentes que implementam os serviços que ele requer, isto é, os componentes dos quais ele depende. A ausência de conectores e das interfaces requeridas aumenta o acoplamento entre os componentes, dificultando a reusabilidade e diminuindo a flexibilidade dos mesmos. Assim, as classes *Facades* que implementam as interfaces providas do componente devem instanciar os componentes dos quais ele depende e acessar as interfaces providas dos componentes, podendo também realizar atividades de adaptação, anteriormente sob o controle dos conectores. A ausência de conectores também pode aumentar a complexidade do projeto dos componentes, uma vez que os requisitos não-funcionais não poderiam ser extraídos da implementação dos componentes (ver diretriz de projeto Isolamento dos requisitos não-funcionais - Seção 3). A flexibilidade do componente, com relação a mudanças na sua implementação, também foi comprometida pela não utilização da fábrica de objetos e de interfaces internas à implementação do componente (ver diretriz de projeto Baixo acoplamento entre classes de implementação - Seção 3).

5.1.3 Resultados

Apesar das customizações realizadas, os resultados obtidos com a utilização do modelo COSMOS foram muito satisfatórios. Tais resultados foram comprovados a partir de entrevistas realizadas com a equipe de desenvolvimento da empresa:

(i) Obteve-se de um processo de desenvolvimento bem definido com passos para identificação de componentes, refinamento, definição da arquitetura do software e posterior implementação em conformidade com a arquitetura proposta.

(ii) A utilização de *UML Components* e o modelo COSMOS para definir a arquitetura do software (componentes, conectores e configuração), foi bem aceita pela equipe de desenvolvimento e deu maior agilidade à atividade de implementação.

(iii) A estruturação do sistema em componentes tornou o sistema muito mais próximo do que foi modelado e, consequentemente, muito mais fácil de implementar.

(iv) Os defeitos encontrados durante a implementação foram facilmente localizados, uma vez que a divisão do sistema em componentes fez com que os problemas ficassem muito mais pontuais.

(v) Conseguiu-se extrair alguns componentes que poderão ser reutilizados em outros sistemas.

Como observação final, a empresa pretende adotar o modelo COSMOS em projetos futuros. A não definição das interfaces requeridas é considerada hoje pelos desenvolvedores como uma pobre decisão de projeto, uma vez que os componentes tornaram-se altamente dependentes uns dos outros, o que pode vir a dificultar a manutenção. Nos projetos seguintes, pretende-se adotar essa diretriz de projeto (ver diretriz de projeto Declaração explícita das dependências dos componentes - Seção 3).

5.2 Estudo de caso: O Telestrada

O Telestrada é um sistema real que visa estabelecer um canal de comunicação entre usuários das rodovias federais e os responsáveis pela operação e conservação dessas vias. Apoiado por um banco de dados, o Telestrada mantém informações sobre as condições das rodovias e outras informações de interesse direto dos usuários, como por exemplo, a localização de postos de combustíveis. São objetivos do Telestrada: criar e manter bases de dados atualizadas sobre as rodovias federais contendo, pelo menos, informações sobre a situação do tráfego e do trânsito, das condições das rodovias e das facilidades existentes ao longo dos trechos; fornecer informações dessas bases aos usuários das rodovias federais, sejam eles viajantes em curso ou que ainda estejam planejando sua viagem; oferecer aos usuários das rodovias um canal de comunicação com o governo federal, para apresentação de sugestões e reclamações sobre os serviços prestados pelas operadoras que administram as vias.

Viajantes que estejam planejando ou realizando uma viagem ocasional de turismo ou a negócios, por exemplo, caminhoneiros, motoristas de táxi, agentes de turismo e outros profissionais que utilizam constantemente as rodovias em decorrência de sua atividade, agentes do antigo DNIT[4] ou DPRF[5], responsáveis por alimentar o sistema com informações a respeito de eventos que possam intervir nas condições das rodovias ou do tráfego e administradores do DNIT ou DPRF, responsáveis pelo planejamento e controle da operação das rodovias são os principais usuários do sistema. Para isso, o Telestrada oferece funcionalidades como registro de eventos que possam intervir nas condições das rodovias ou do trânsito, tais como obras na pista e congestionamentos, emissão de boletins periódicos com informações atualizadas a respeito das condições das rodovias e do trânsito, registro de reclamações e sugestões dos usuários, relativas aos serviços das concessionárias das rodovias, consultas relativas às informações mantidas no seu banco de dados, feitas através de: requisições originadas por outros sistemas de informações, ligados através da *Internet*; requisições enviadas ao *site* do Telestrada na rede *WWW*, originadas por usuários ligados através da Internet.

---

[4] Departamento Nacional de Infra-Estrutura de Transportes

[5] Departamento de Polícia Rodoviária Federal

Devido a sua complexidade, o Telestrada pode ser dividido em cinco módulos básicos: (1) Módulo de cadastro rodoviário: para inserção, consulta e alteração de dados rodoviários; (2) Módulo de eventos: para registro de ocorrências na malha rodoviária, que devam ser observadas pelos usuários; (3) Módulo de consultas de rotas e condições de rodovias: para consulta de informações sobre trechos da malha rodoviária; (4) Módulo de emergências: para solicitação de atendimento de emergência em trechos da malha rodoviária; e (5) Módulo de reclamações: para registro e consulta de reclamações sobre os serviços prestados pelas operadoras que administram malhas rodoviárias. Este último módulo foi utilizado para a aplicação do modelo COSMOS e será melhor descrito a seguir.

5.2.1 O Módulo de Reclamações

Este módulo define operações para registrar, atualizar e consultar processos de reclamação de trechos da malha rodoviária. Através deste módulo, um usuário da rodovia pode registrar reclamação pela *Internet* e consultar o andamento de um processo de reclamação. Além destas operações, o módulo de reclamação também define operações privadas a administradores do sistema (supervisores e responsáveis por trechos) que permitem: registrar e remover tipo de reclamação e definir, encaminhar e finalizar um processo de reclamação. O diagrama a seguir mostra a realização dos casos de uso pelos seus respectivos atores.

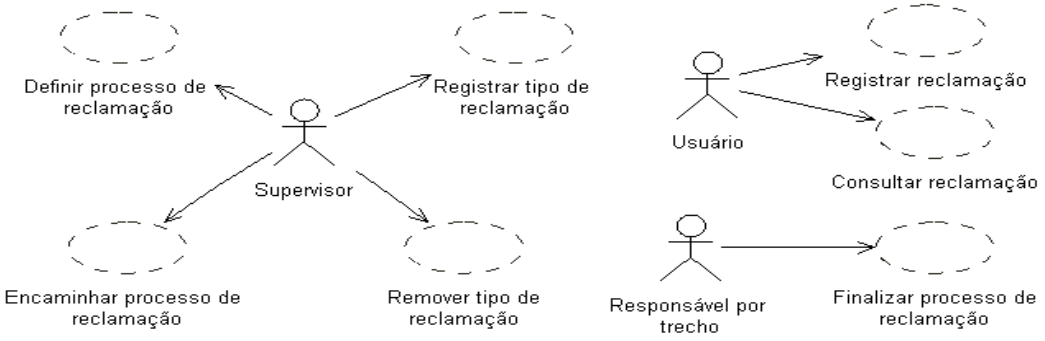

Figura 18. Diagramas de caso de uso do Módulo de Reclamações

5.2.2 Casos de uso do Módulo de Reclamações

- **Caso de uso Cadastrar/Remover Tipo de Reclamação**

Este caso de uso é responsável pela inserção e remoção de um tipo de reclamação na base de dados do sistema. Os tipos de reclamação são usados, no momento do registro de uma reclamação sobre um determinado trecho da malha rodoviária, pelos usuários das rodovias (Figura 19).

Cadastrar/Remover Tipo de Reclamação Iniciador: Supervisor do sistema Objetivo: Definir ou remover um tipo de reclamação do sistema
Cenário de Sucesso: 1. Supervisor acessa página do Telestrada na Internet e seleciona opção 'cadastro de tipo de reclamação'. 2. Supervisor seleciona opção para cadastrar tipo de reclamação. 3. Supervisor define um tipo de reclamação. 4. Sistema valida dados, verificando se o tipo de reclamação já existe. 5. Sistema armazena o tipo de reclamação.
Extensões: 5. Sistema avisa que tipo de reclamação já existe 2.Supervisor seleciona opção para remover tipo de reclamação. 3. Supervisor seleciona um tipo de reclamação para remover. 4. Sistema remove o tipo de reclamação

Figura 19. Descrição do UC Cadastrar/Remover tipo de reclamação

- **Caso de uso Registrar Reclamação**

Este caso de uso começa quando o usuário de uma rodovia acessa o *site* do Telestrada para registrar uma reclamação a respeito de um trecho de uma rodovia (Figura 20).

Registrar Reclamação Iniciador: Usuário final do sistema Objetivo: Permitir a inclusão de reclamação de trechos de rodovias.
Cenário de Sucesso: 1. Usuário acessa página do Telestrada e seleciona a opção 'registrar reclamação'. 2. Sistema exibe lista de rodovias e tipos de reclamação. 3. Usuário seleciona uma rodovia. 4. Sistema exibe trechos da rodovia selecionada. 5. Usuário seleciona um trecho e um tipo de reclamação e seleciona próximo passo. 6. Sistema acessa banco de dados e verifica a existência de uma resposta padrão para o dado trecho e tipo da reclamação do usuário. 7. Sistema apresenta resposta padrão para usuário (se existir), juntamente com um formulário que permita o usuário preencher a descrição da reclamação e seus dados pessoais (Nome e e-mail). 8. Usuário preenche os dados do formulário (podendo deixar seus dados pessoais em branco) e seleciona opção 'incluir reclamação no sistema'. 9. Sistema inclui reclamação, exibe número de registro da reclamação e instruções para usuário acompanhar o status da reclamação no sistema.
Extensões 8. Usuário fica satisfeito com resposta padrão e seleciona opção de cancelar inclusão de reclamação.

Figura 20. Descrição do UC Registrar reclamação

- **Caso de uso Consultar Reclamação**

Este caso de uso é responsável pela consulta do estado de uma reclamação, anteriormente registrada por um determinado usuário de um trecho de uma rodovia.(Figura 21).

Consultar Reclamação Iniciador: Usuário final do sistema Objetivo: Permitir a consulta de reclamação a respeito de trechos de rodovias.
Cenário de Sucesso: 1. Usuário acessa página do Telestrada e seleciona a opção 'consultar reclamação'. 2. Usuário informa número da reclamação. 3. Sistema apresenta reclamação para usuário, contendo: rodovia, UF, trecho e situação em que se encontra a reclamação ('nova', 'sendo analisada' ou 'resolvida') e dados do usuário que incluiu reclamação. 4. Usuário escolhe imprimir a reclamação. 5. Sistema formata relatório e envia para impressora do usuário.
Extensões 4. Usuário escolhe receber por e-mail os dados da reclamação. 5. Sistema abre formulário para usuário definir endereço do destinatário a receber o mail. 6. Sistema formata e envia o mail. 4. Usuário escolhe opção de alterar suas informações e/ou informações da reclamação antes de escolher imprimir ou enviar por e-mail.

Figura 21. Descrição do UC Consultar reclamação

- **Caso de uso Definir Processo de Reclamação**

Este caso de uso começa quando o supervisor do sistema acessa o *site* do Telestrada para definir um processo de reclamação para um dado conjunto de reclamações, a respeito de um mesmo trecho de rodovia (Figura 22).

Definir Processo de Reclamação Iniciador: Supervisor do sistema Objetivo: Criar processos de reclamações
Cenário de Sucesso: 1. Supervisor acessa página do Telestrada e escolhe 'definir processo de reclamação'. 2. Sistema apresenta lista de rodovias e tipos de reclamações. 3. Supervisor escolhe rodovia. 4. Sistema apresenta lista de trechos da rodovia selecionada. 5. Supervisor seleciona trecho e tipo de reclamação. 6. Sistema apresenta todas as reclamações para o dado trecho e tipo. 7. Supervisor seleciona reclamações que devem fazer parte do processo a ser criado. 8. Supervisor seleciona opção 'criar processo de reclamação'. 9. Sistema cria novo processo de reclamações para o trecho e tipo de reclamação.

Figura 22. Descrição do UC Definir Processo de Reclamação

- **Caso de uso Encaminhar Processo de Reclamação**

Este caso de uso é responsável pelo encaminhamento de um processo de reclamação, por parte do supervisor do sistema, para o responsável pelo trecho da rodovia no qual foi criado o processo de reclamação (Figura 23).

Encaminhar Processo de Reclamação Iniciador: Supervisor do sistema Objetivo: Encaminhar processos de reclamações
Cenário de Sucesso: 1. Supervisor acessa página do Telestrada e escolhe a opção 'encaminhar processo de reclamação'. 2. Sistema seleciona processos que ainda não foram encaminhados para responsáveis pelos trechos (trechos estes, associados aos processos). 3. Para cada processo, o supervisor escolhe um responsável para o processo, a partir de uma lista pré-definida pelo sistema. 4. Para cada processo, o supervisor escolhe a forma de encaminhamento para responsável pelo trecho entre: e-mail, fax ou impressão. 5. Supervisor confirma o encaminhamento do processo. 6. Sistema envia cada processo a seu responsável de acordo com forma de encaminhamento. 7. Sistema atualiza as reclamações de cada processo com o status de 'sendo analisada'.
Extensão: 3. Supervisor pode selecionar um processo e ver seus detalhes (reclamacoes, tipo, trecho), em uma nova janela do sistema.

Figura 23. Descrição do UC Encaminhar Processo de Reclamação

- **Finalizar Processo de Reclamação**

Este caso de uso começa quando o responsável por um trecho de rodovia acessa o *site* do Telestrada para finalizar um processo de reclamação, anteriormente encaminhado para este responsável pelo supervisor do sistema (Figura 24).

Finalizar Processo de Reclamação Iniciador: Responsável pelo Trecho Objetivo: Finaliza um processos de reclamação
Cenário de Sucesso: 1. Responsável por trecho acessa página do Telestrada e escolhe 'finalizaar processo de reclamação'. 2. Sistema exibe os processos atribuídos ao responsável. 3. Responsável seleciona um processo para finalização. 4. Sistema exibe todas as reclamações do processo. 5. Responsável atualiza status do processo de reclamação com 'finalizada'. 6. Responsável pode preencher um campo de comentário para cada reclamação, e um campo para o processo de reclamações em questão, se desejar. Pode também atribuir o comentário de um campo para todos os demais. (O campo da reclamação é apresentado para usuário final quando este consulta reclamação. O campo do processo é apresentado para o supervisor quando este visualiza o processo). 7. Sistema atualiza cada reclamação de cada processo com o status de 'finalizada'.

Figura 24. Descrição do UC Finalizar Processo de Reclamação

Com base na descrição dos casos de uso, foi utilizado o processo proposto por *UML Components* (ver Seção 2.1.2) para identificar os componentes do sistema, suas interfaces e definir a sua arquitetura usando também o modelo COSMOS para estruturar o sistema em termos dos componentes, conectores e interfaces.

5.2.3 A Arquitetura do Módulo de Reclamações

Neste estudo de caso, a arquitetura em camadas original de *UML Components* (ver Seção 5.1.1) foi modificada. Os componentes das camadas de apresentação e aplicação foram definidos unicamente na camada de apresentação, tendo em vista que o sistema não prevê outros tipos de interface com o usuário que não seja a *Web*. Foi também definida uma camada intermediária entre as camadas de sistema e negócio, objetivando garantir a flexibilidade e independência dos componentes, através dos conectores. Assim, os componentes do Telestrada foram organizados em 4 camadas: apresentação, sistema, conexão e negócio. A camada de apresentação define as interfaces com os usuários do sistema, que interagem com a camada de sistema, responsável pela definição das macro-operações dos casos de uso. A camada de conexão é responsável por estabelecer a ponte de comunicação entre a camada de sistema e a camada de negócios, que define as operações fundamentais do sistema, baseando-se nas regras de negócio do Telestrada. Para essa implementação, assumimos que a camada de negócios contém também código responsável pela persistência dos objetos.

- **Caso de uso Cadastrar/Remover Tipo de Reclamação**

A Figura 25 descreve a composição de componentes e conectores, responsáveis pela realização dos casos de uso cadastrar e remover tipo de reclamação.

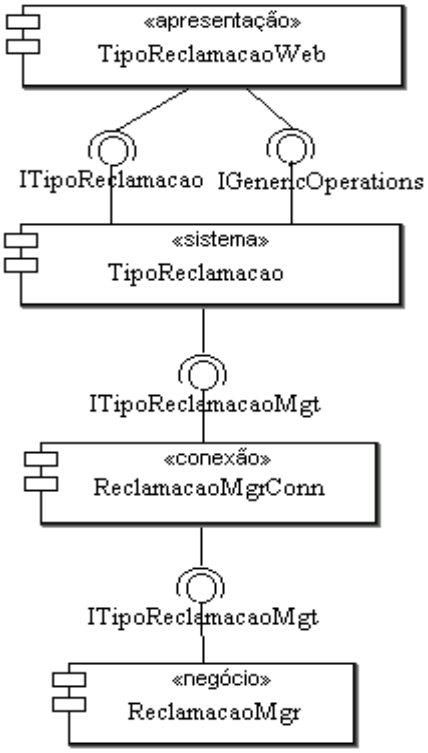

Figura. 25. Arquitetura do UC Cadastrar/Remover Tipo de Reclamação

O componente de apresentação TipoReclamacaoWeb contém os *JSPs* e *Servlets*, (ver Seção 2.3.1) responsáveis pela interação com o usuário do sistema e controle do fluxo de execução do caso de uso. ITipoReclamacao e IGenericOperations são as interfaces providas do componente de sistema TipoReclamacao. Elas definem as macro-operações do caso de uso e são usadas pelo componente de apresentação TipoReclamacaoWeb. Para conectar a interface requerida ITipoReclamacaoMgt do componente TipoReclamacao, o conector ReclamacaoMgrConn usa a interface provida do componente de negócio ReclamacaoMgr, que define as regras e transformações de negócios referentes ao registro de reclamação em geral. A Figura a seguir (Figura 26) mostra o projeto das interfaces providas e requeridas dos componentes que implementam esse caso de uso.

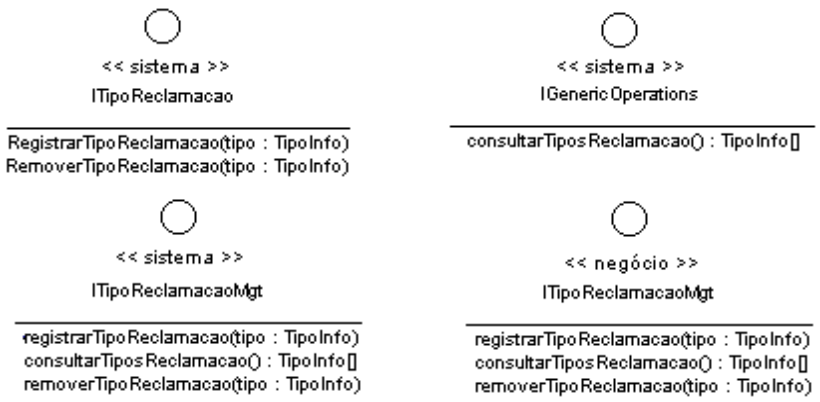

Figura 26. Interfaces dos componentes - UC Cadastrar/Remover Tipo de Reclamação

- **Caso de uso Registrar Reclamação**

A Figura 27 descreve a composição de componentes e conectores, responsáveis pela realização do caso de uso registrar reclamação.

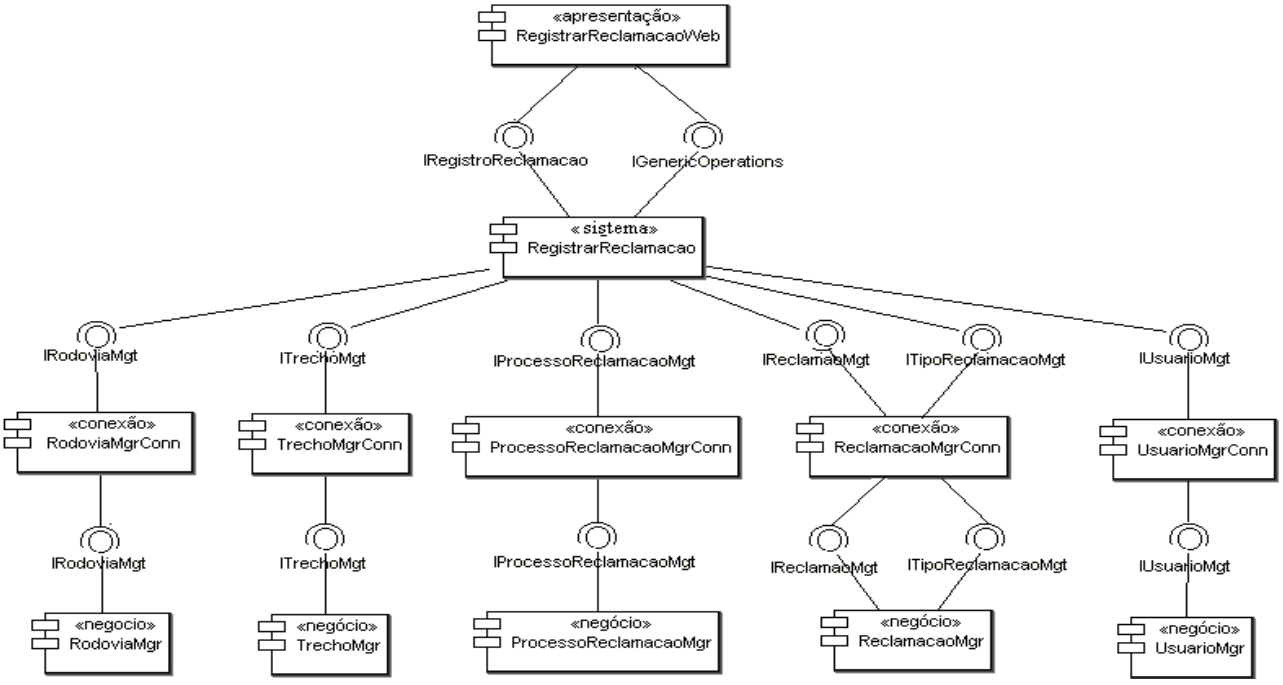

Figura 27. Arquitetura do UC Registrar Reclamação

O componente de apresentação RegistrarReclamacaoWeb contém os *JSPs* e *Servlets* (ver Seção 2.3.1), responsáveis pela interação com os usuários do sistema e controle do fluxo de

execução do caso de uso. IRegistroReclamacao e IGenericOperations são as interfaces providas do componente de sistema RegistrarReclamacao, que definem as macro-operações do caso de uso e são usadas pelo componente de apresentação RegistrarReclamacaoWeb. Para conectar as interfaces requeridas IRodoviaMgt, ITrechoMgt, IProcessoReclamacaoMgt, IReclamacaoMgt, ITipoReclamacaoMgt e IUsuarioMgt do componente de sistema RegistrarReclamacao, os conectores RodoviaMgrConn, TrechoMgrConn, ProcessoReclamacaoMgrConn, ReclamacaoMgrConn e UsuarioMgrConn usam as interfaces providas dos componentes de negócio RodoviaMgr, TrechoMgr, ProcessoReclamacaoMgr, ReclamacaoMgr e UsuarioMgr que definem as regras e transformações de negócios. A Figura a seguir (Figura 28) mostra o projeto das interfaces providas e requeridas dos componentes que implementam esse caso de uso.

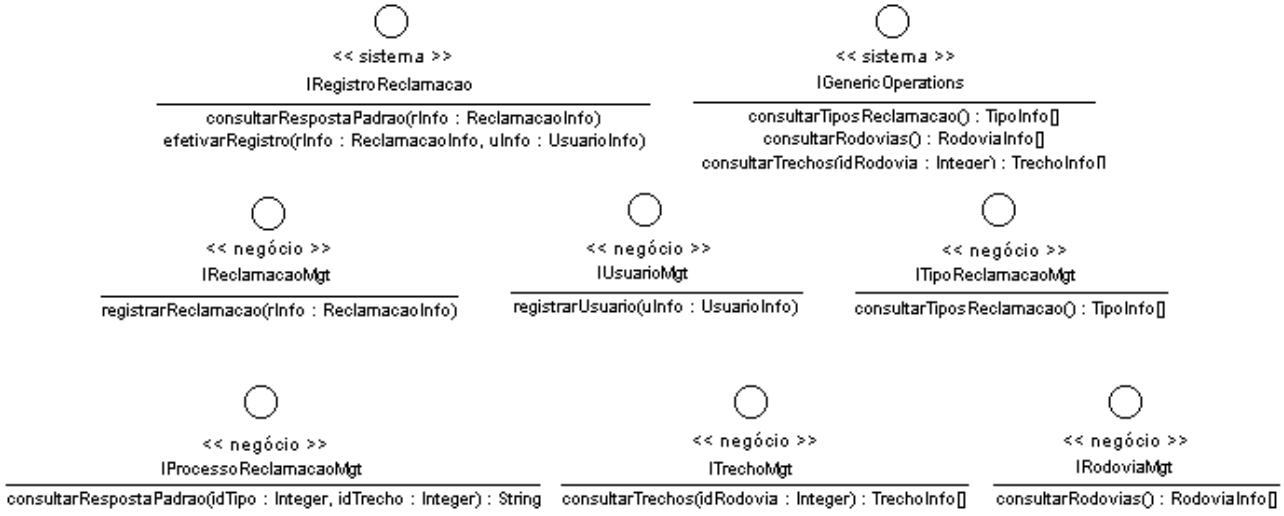

Figura 28. Interfaces dos componentes - UC Registrar Reclamação

- **Caso de uso Consultar Reclamação**

A Figura 29 descreve a composição de componentes e conectores, responsáveis pela realização do caso de uso consultar reclamação.

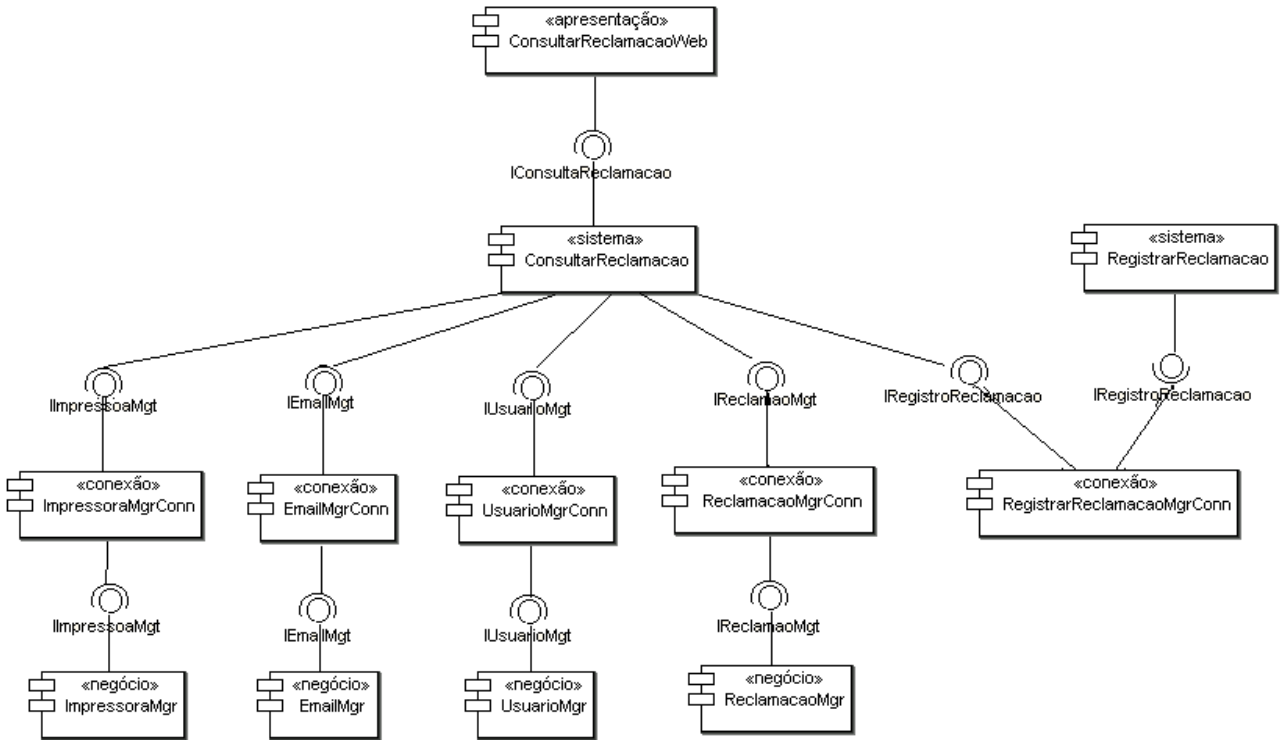

Figura. 29. Arquitetura do UC Consultar Reclamação

O componente de apresentação ConsultarReclamacaoWeb contém os *JSPs* e *Servlets* (ver Seção 2.3.1) responsáveis pela interação com os usuários do sistema e controle do fluxo de execução do caso de uso. IConsultaReclamacao é a interface provida do componente de sistema ConsultarReclamacao. Ela define as macro-operações do caso de uso e é usada pelo componente de apresentação ConsultarReclamacaoWeb. Para conectar as interfaces requeridas IImpressoraMgt, IEmailMgt, IUsuarioMgt e IReclamacaoMgt do componente de sistema ConsultarReclamacao, os conectores ImpressoraMgrConn, EmailMgrConn, UsuarioMgrConn e ReclamacaoMgrConn usam as interfaces providas dos componentes de negócio ImpressoraMgr, EmailMgr, UsuarioMgr e ReclamacaoMgr. O componente de sistema ConsultarReclamacao também tem uma interface requerida IRegistroReclamacao que define operações para atualizar dados da reclamação. Para conectar essa interface requerida, o conector RegistrarReclamacaoConn usa a interface provida IRegistroReclamacao do componente RegistrarReclamacao da própria camada de sistema. A Figura a seguir (Figura 30) mostra o projeto das interfaces providas e requeridas dos componentes que implementam esse caso de uso.

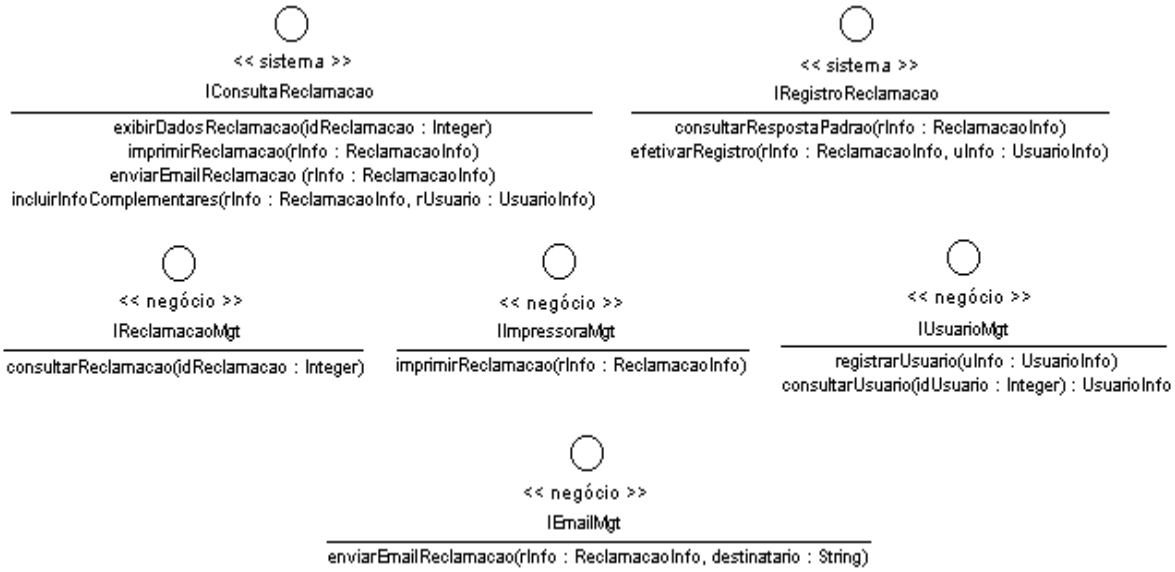

Figura 30. Interfaces dos componentes - UC Consultar Reclamação

- **Caso de uso Definir Processo de Reclamação**

A Figura 31 descreve a composição de componentes e conectores, responsáveis pela realização do caso de uso definir processo de reclamação.

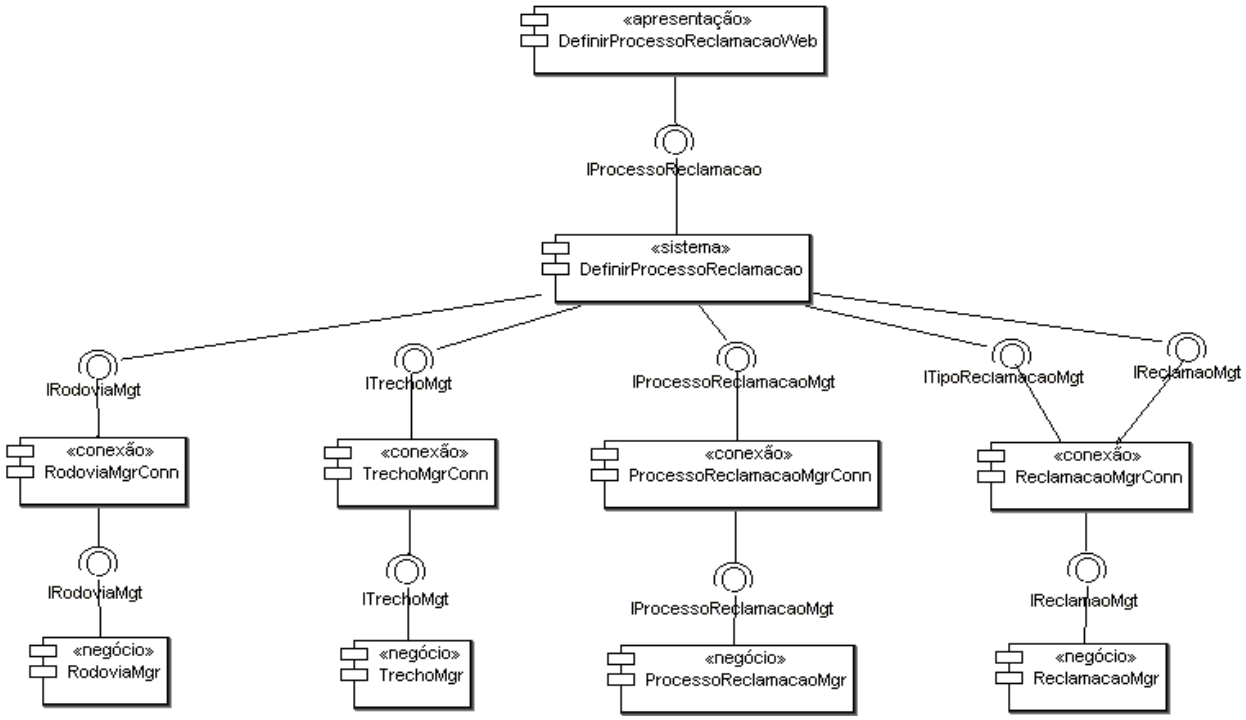

Figura 31. Arquitetura do UC Definir Processo de Reclamação

O componente de apresentação DefinirProcessoReclamacaoWeb contém os *JSPs* e *Servlets* (ver Seção 2.3.1) responsáveis pela interação com os usuários do sistema e controle do fluxo de execução do caso de uso. IProcessoReclamacao e IGenericOperations são as interfaces providas do componente de sistema DefinirProcessoReclamacao. Elas definem as macro-operações do caso de uso e são usadas pelo componente de apresentação DefinirProcessoReclamacaoWeb. Para conectar as interfaces requeridas, IRodoviaMgt, ITrechoMgt, IProcessoReclamacaoMgt ITipoReclamacaoMgt e IReclamacaoMgt do componente de sistema DefinirProcessoReclamacao, os conectores RodoviaMgrConn, TrechoMgrConn, ProcessoReclamacaoMgrConn e ReclamacaoMgrConn usam as interfaces providas dos componentes de negócio RodoviaMgr, TrechoMgr ProcessoReclamacaoMgr e ReclamacaoMgr. A Figura a seguir (Figura 32) mostra o projeto das interfaces providas e requeridas dos componentes que implementam esse caso de uso.

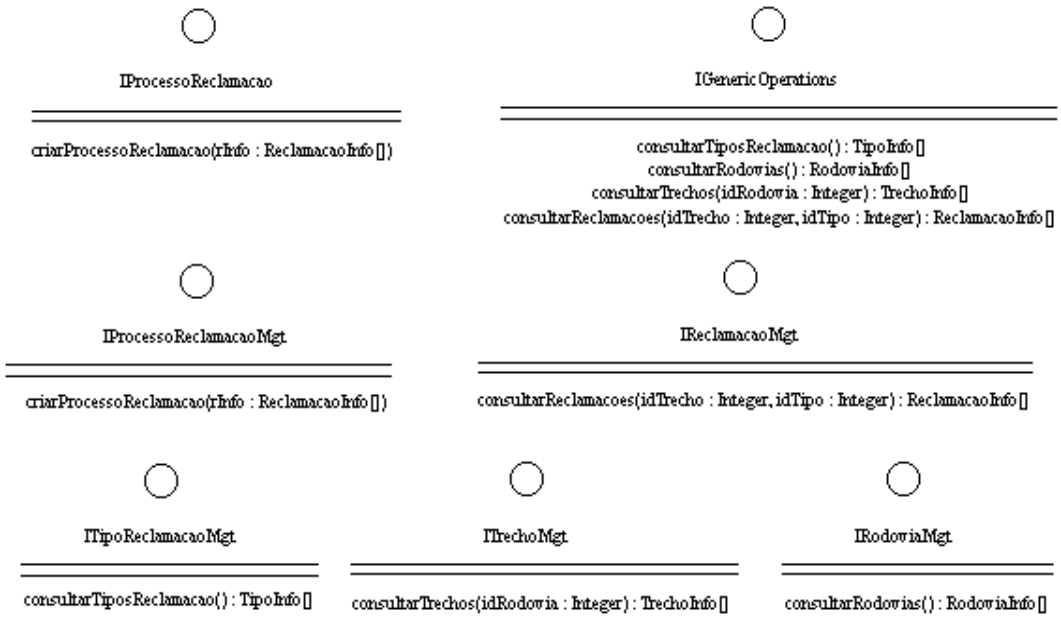

Figura 32. Interfaces dos componentes - UC Definir Processo de Reclamação

- **Caso de uso Encaminhar Processo de Reclamação**

A Figura 33 descreve a composição de componentes e conectores, responsáveis pela realização do caso de uso encaminhar processo de reclamação.

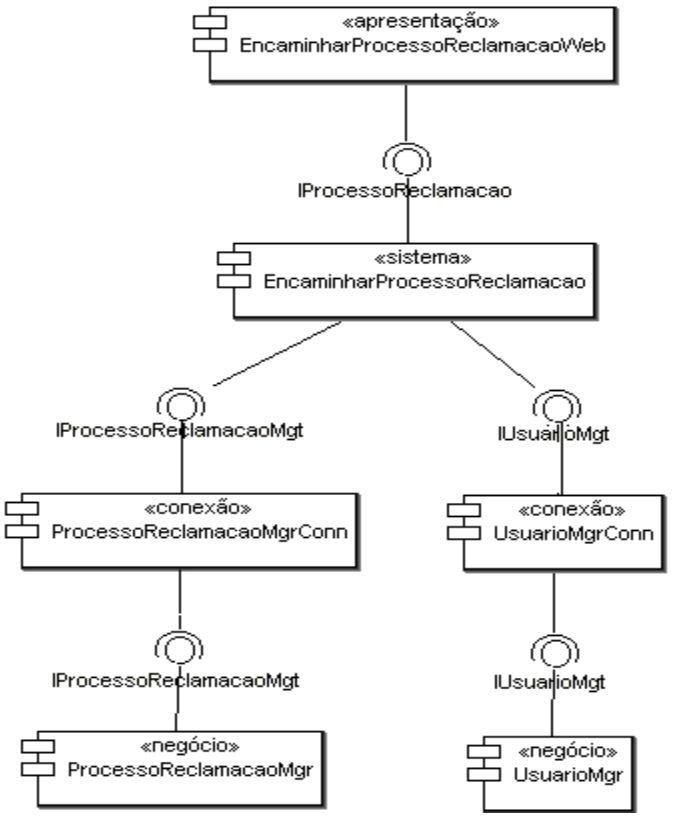

Figura 33. Arquitetura do UC Encaminhar Processo de Reclamação

O componente de apresentação EncaminharProcessoReclamacaoWeb contém os *JSPs* e *Servlets* (ver Seção 2.3.1) responsáveis pela interação com os usuários do sistema e controle do fluxo de execução do caso de uso. IProcessoReclamacao é a interface provida do componente de sistema EncaminharProcessoReclamacao. Ela define as macro-operações do caso de uso e é usada pelo componente de apresentação EncaminharProcessoReclamacaoWeb. Para conectar as interfaces requeridas IProcessoReclamacaoMgt e IUsuarioMgt do componente de sistema EncaminharProcessoReclamacao, os conectores ProcessoReclamacaoMgrConn e UsuarioMgrConn usam as interfaces providas dos componentes de negócio ProcessoReclamacaoMgr e ProcessoReclamacaoMgr. A Figura a seguir (Figura 34) mostra o projeto das interfaces providas e requeridas dos componentes que implementam esse caso de uso.

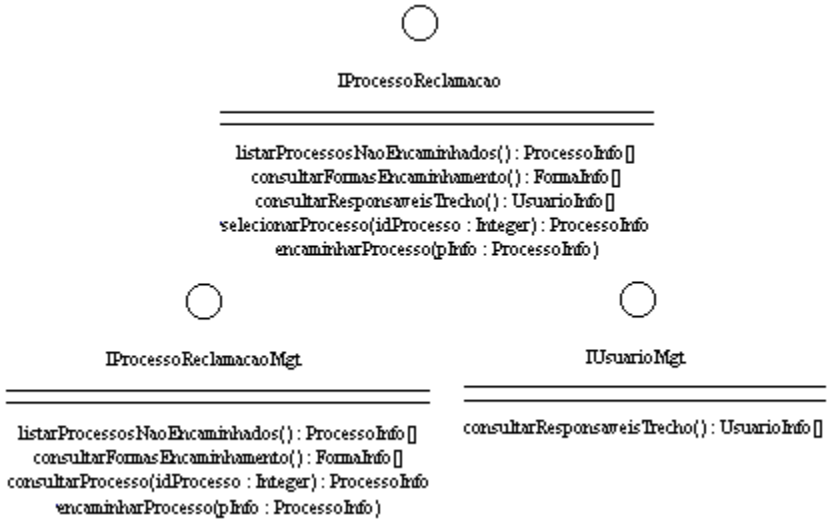

Figura 34. Interfaces dos componentes - UC Encaminhar Processo de Reclamação

- **Finalizar Processo de Reclamação**

A Figura 35 descreve a composição de componentes e conectores, responsáveis pela realização do caso de uso finalizar processo de reclamação.

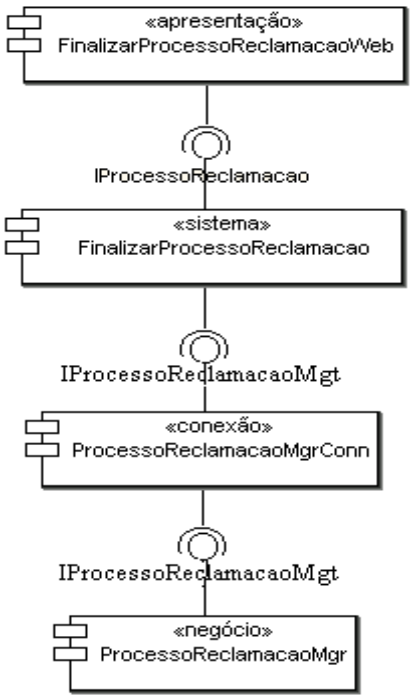

Figura 35. Arquitetura do UC Finalizar Processo de Reclamação

O componente de apresentação FinalizarProcessoReclamacaoWeb contém os *JSPs* e *Servlets* (ver Seção 2.3.1) responsáveis pela interação com os usuários do sistema e controle do fluxo de execução do caso de uso. IProcessoReclamacao é a interface provida do componente de sistema FinalizarProcessoReclamacao, que define as macro-operações do caso de uso e é usada pelo componente de apresentação FinalizarProcessoReclamacaoWeb. Para conectar a interface requerida IProcessoReclamacaoMgt do componente de sistema FinalizarProcessoReclamacao, o conector ProcessoReclamacaoMgrConn usa a interface provida do componente de negócio ProcessoReclamacaoMgr. A Figura a seguir (Figura 36) mostra o projeto das interfaces providas e requeridas dos componentes que implementam esse caso de uso.

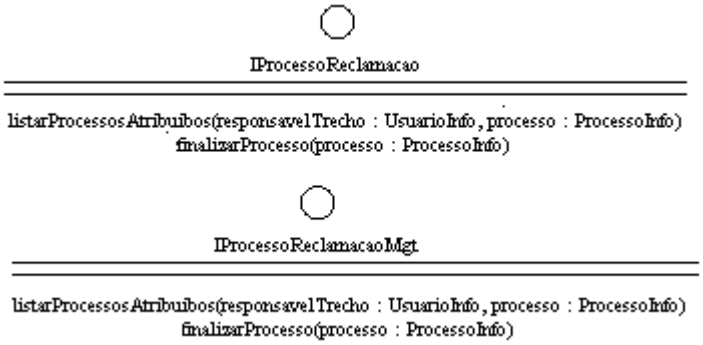

Figura 36. Interfaces dos componentes - UC Finalizar Processo de Reclamação

Tendo como base a arquitetura do módulo de reclamação, usamos o modelo COSMOS para realizar o mapeamento da arquitetura do sistema para à implementação.

5.2.4 A Implementação do Telestrada

O sistema Telestrada foi desenvolvido em Java, mais especificamente na plataforma J2EE (ver Seção 2.3.1). Para implementar a camada de apresentação, nós usamos o *framework Struts* (ver Seção 2.3.1) A integração da camada de apresentação com a camada de sistema do Telestrada é feita a partir das classes *Actions* do *framework Struts*. Cada classe *Action* representa um passo do caso de uso que realiza uma determinada função. Para acessar os componentes da camada de sistema, as classes *Actions* precisam ter acesso ao repositório de componentes, previamente

instalados e configurados, isto é, com suas dependências resolvidas. Supondo a existência deste repositório de componentes, representado por uma classe *ComponentPool*, que será melhor descrita na seção 6.1, e considerando também um fluxo de execução de uma requisição de um usuário do Telestrada para registrar uma reclamação no sistema, o trecho de código a seguir representa a interação da *Action* RegistrarReclamacaoAction do componente de apresentação RegistrarReclamacaoWeb com o componente de sistema RegistrarReclamacao (Figura 37).

```
package apresentacao.registrarReclamacaoWeb;

import sistema.registrarReclamacao.spec.req.IReclamacaoMgt;
import sistema.registrarReclamacao.spec.prov.IRegistroReclamacao;
import sistema.registrarReclamacao.spec.prov.*;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForward;

import framework.ComponentPool;

public class RegistrarReclamacaoAction extends Action {

  public ActionForward perform(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
      throws IOException, ServletException {

    try {
      sistema.registrarReclamacao.spec.prov.IManager mgrApp = (sistema.registrarReclamacao.spec.prov.IManager)
            ComponentPool.getComponentInstance("RegistrarReclamacao","Sistema");
      IRegistroReclamacao registrar =
              (IRegistroReclamacao)mgrApp.getProvidedInterface("sistema.registrarReclamacao.spec.prov.IRegistroReclamacao");
              ...
      Reclamacao reclamacao = new Reclamacao(id, idTipo, idUsuario, idTrecho, idProcessoReclamacao, nomeReclamacao, descricao);
      Usuario usuario = new Usuario(id, idResponsabilidade, nomeUsuario, email, fone);
              reclamacaoId = registrar.efetivarRegistro(reclamacao, usuario);
              ...
    } catch (
      ...
    }
  }
}
```

Figura 37. Exemplo de *Action* da camada de apresentação do Telestrada

A Figura acima mostra o último passo do registro de uma reclamação do usuário no sistema, isto é, a efetivação da reclamação. A *Action* RegistrarReclamacaoAction acessa o *ComponentPool* e recupera a instância do componente de sistema RegistrarReclamacao. Com a instância do componente RegistrarReclamacao, a interface provida IRegistroReclamacao é acessada e a operação para efetivar o registro da reclamação é chamada. A Figura abaixo mostra a estrutura interna do componente de sistema RegistrarReclamacao (Figura 38).

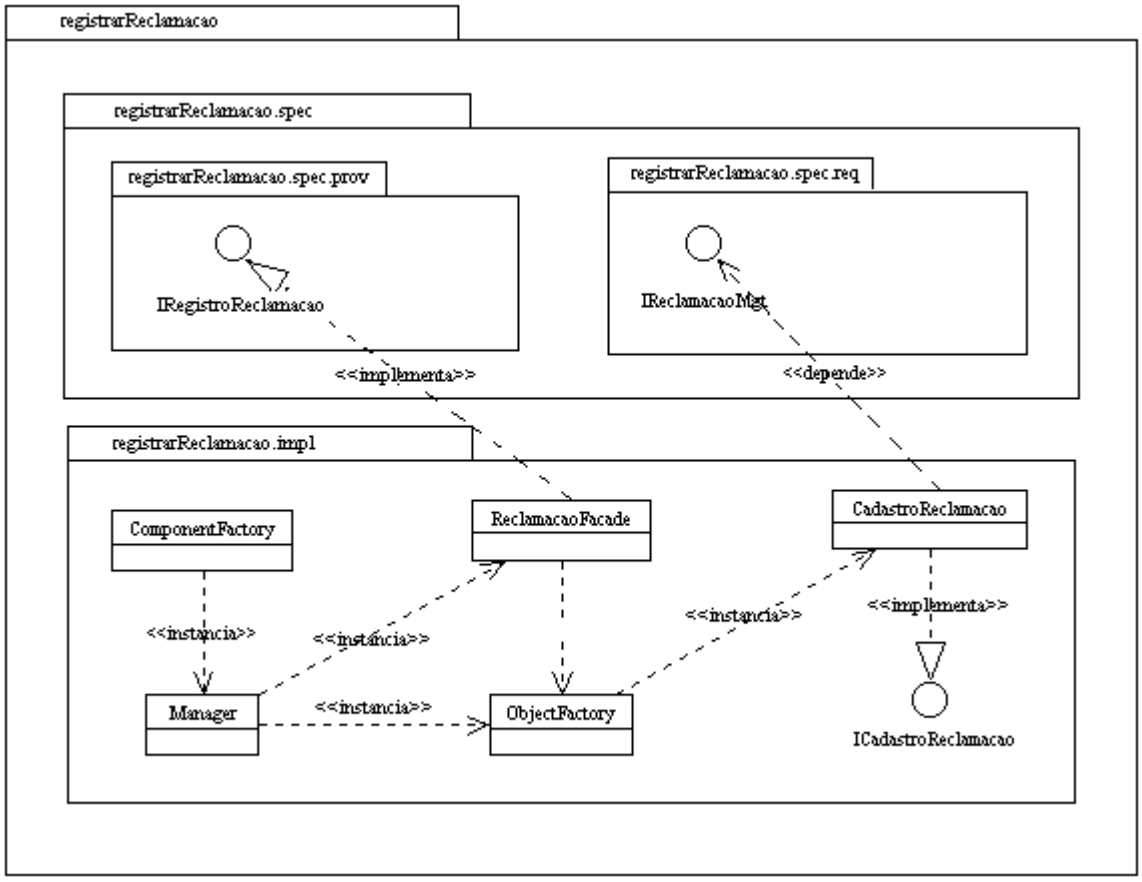

Figura 38. A estrutura interna do componente de sistema RegistrarReclamacao

O diagrama da figura 38 mostra apenas uma parte da estrutura do componente RegistrarReclamacao. Como visto na Figura 27, o componente de sistema RegistrarReclamacao tem uma outra interface provida (IGenericOperations), além de IRegistroReclamacao, e outras interfaces requeridas (IRodoviaMgt, ITrechoMgt, IProcessoReclamacaoMgt, ITipoReclamacaoMgt e IUsuarioMgt), além de IReclamacaoMgt. Porém, apenas a efetivação do registro da reclamação será mostrada aqui. A mesma observação também serve para o conector ReclamacaoMgrConn e para o componente de negócio RegistrarReclamacaoMgr. A Figura 39 continua o fluxo de execução da efetivação do registro de reclamação mostrado na Figura 37.

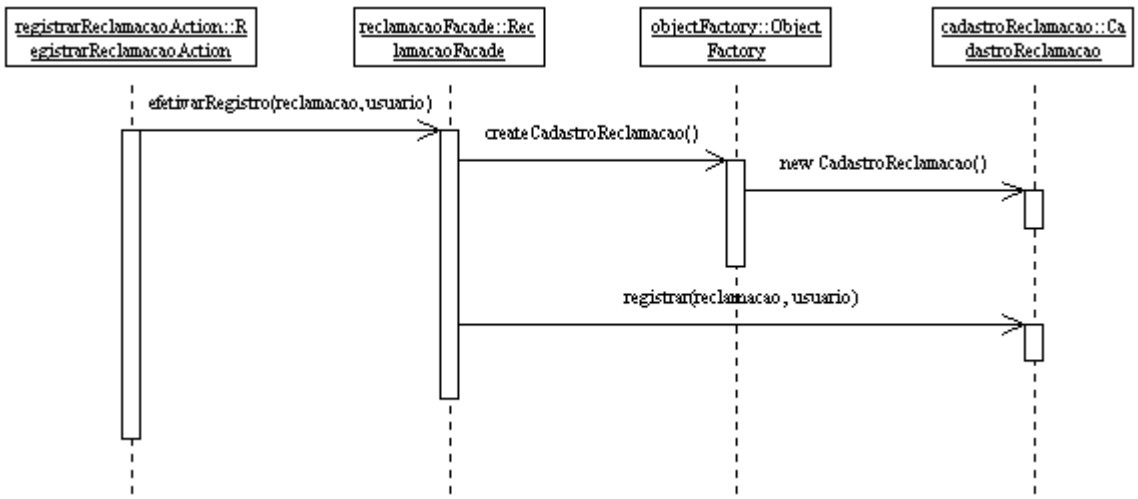

Figura 39. Registro de reclamação no componente de sistema RegistrarReclamacao

A chamada da operação efetivarRegistro, feita pela classe RegistrarReclamacaoAction, a partir da interface provida IRegistrarReclamacao, é propagada internamente ao componente RegistrarReclamacao até a classe CadastroReclamacao. A classe CadastroReclamacao acessa o objeto que implementa a interface requerida IReclamacaoMgt do componente RegistrarReclamacao, previamente configurado, e chama a operação registrarReclamacao. (Figura 40)

```
package sistema.registrarReclamacao.impl;

import sistema.registrarReclamacao.spec.prov.*;
import sistema.registrarReclamacao.spec.req.*;

class ReclamacaoFacade implements IRegistroReclamacao{

 private Manager manager;

 public ReclamacaoFacade(Manager manager){
   this.manager = manager;
 }

 public void efetivarRegistro(Reclamacao reclamacao) {

            ...
            sistema.registrarReclamacao.spec.req.IReclamacaoMgt iReclamacaoMgt =
            (IReclamacaoMgt)manager.getRequiredInterface("sistema.registrarReclamacao.spec.req.IReclamacaoMgt");

            //inserir reclamacao
            reclamacaoId = iReclamacaoMgt.registrarReclamacao( reclamacao );
            ...
 }
}
```

Figura 40. Implementação do registro de reclamação no componente de sistema RegistrarReclamacao

No momento em que a operação registrarReclamacao da interface requerida IReclamacaoMgt é chamada, o fluxo de execução é direcionado para o conector ReclamacaoMgrConn. A Figura 41 mostra a estrutura interna do conector ReclamacaoMgrConn.

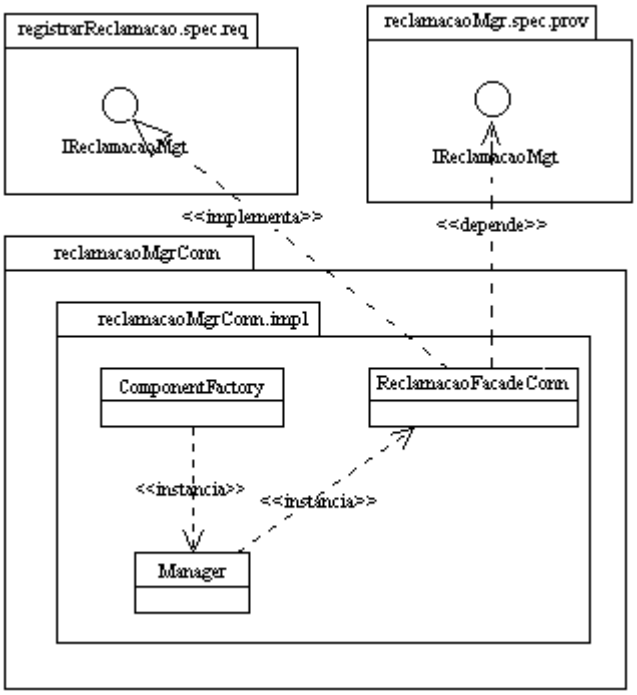

Figura 41. Estrutura interna do conector ReclamacaoMgrConn

A estrutura interna do conector ReclamacaoMgrConn contém uma classe ReclamacaoFacadeConn que implementa a interface requerida IReclamacaoMgt do componente de sistema RegistrarReclamacao, usando a interface provida IReclamacaoMgt do componente de negócio RegistrarReclamacaoMgr. A classe ReclamacaoFacadeConn é responsável pelas adaptações das interfaces requeridas e providas dos componentes que participam da interação, bem como dos tipos de dados presentes nas assinaturas das operações de tais interfaces. (Figura 42).

```
package connNeg.reclamacaoMgrConn.impl;

import sistema.registrarReclamacao.spec.req.*;
import sistema.registrarReclamacaoMgr.spec.prov.*;

import sistema.registrarReclamacao.spec.prov.IManager;

class ReclamacaoFacadeConn
 implements sistema.registrarReclamacao.spec.req.IReclamacaoMgt {

        private IManager manager = null;

        public ReclamacaoFacadeConn(IManager manager) {
         this.manager = manager;
        }

        public void registrarReclamacao(Reclamacao reclamacao) {
                negocios.reclamacaoMgr.spec.prov.IReclamacaoMgt iReclamacaoMgt;

                iReclamacaoMgt = (negocios.reclamacaoMgr.spec.prov.IReclamacaoMgt)
                        manager.getRequiredInterface("negocios.reclamacaoMgr.spec.prov.IReclamacaoMgt");

                // Adaptação dos tipos de dados das interfaces requeridas e providas
                Reclamcao rec = new Reclamcao(reclamacao);
                iReclamacaoMgt.registrarReclamacao(rec);
         }
}
```

Figura 42. Implementação do registro de reclamação no conector ReclamacaoMgrConn

Um problema que surgiu aqui foi onde definir os tipos de dados comuns aos componentes. No caso da interação entre os componentes RegistrarReclamacao e ReclamacaoMgr, o tipo de dados IReclamacao é comum aos dois componentes, visto que ele está presente na assinatura de operações da interface requerida IReclamacaoMgt de RegistrarReclamacao e também da interface provida IReclamacaoMgt de ReclamacaoMgr. Para garantir a independência dos componentes, a interface IReclamacao foi redefinida em cada um dos componentes. Assim, o conector ReclamacaoMgrConn deve adaptar, além das interfaces providas e requeridas dos componentes RegistrarReclamacao e ReclamacaoMgr, a interface IReclamacao do componente RegistrarReclamacao à interface IReclamacao do componente ReclamacaoMgr. A princípio, consideraremos que os tipos de dados públicos dos componentes RegistrarReclamacao e ReclamacaoMgr foram definidos no pacote spec.prov de cada um dos componentes. A seção 6.2 discutirá algumas propostas para definição de tipos de dados das aplicações. A Figura a seguir (Figura 43) mostra a adaptação do tipo de dados IReclamacao, realizada pelo conector ReclamacaoMgrConn.

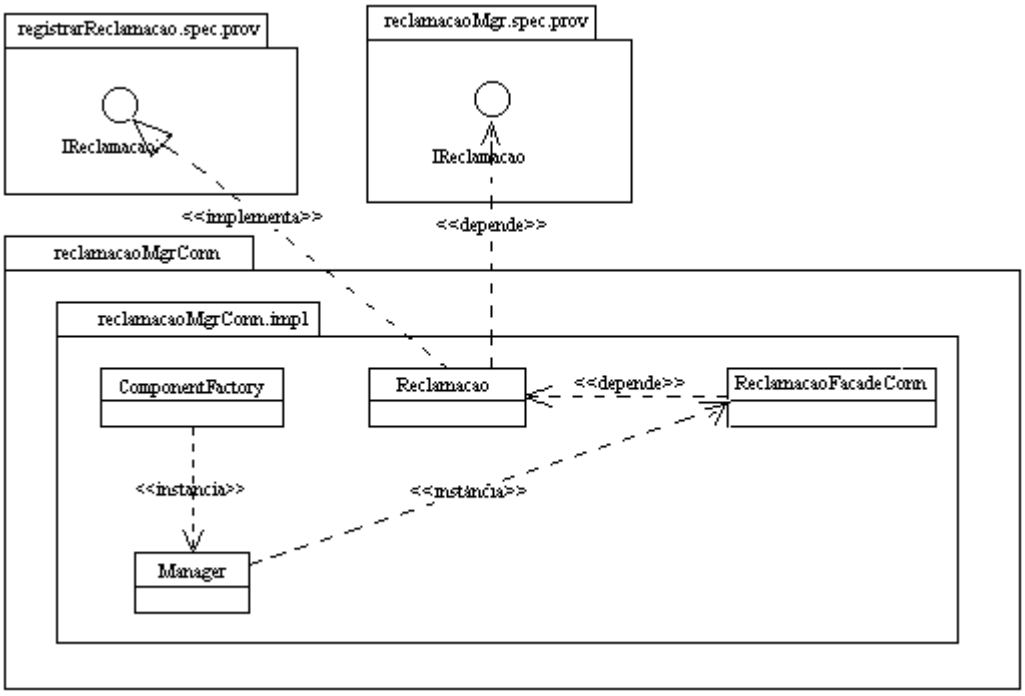

Figura 43. Adaptação do tipo de dados IReclamacao pelo conector ReclamacaoMgrConn

Na Figura acima, o conector ReclamacaoMgrConn define uma classe de implementação, chamada de Reclamacao, que usa a interface IReclamacao de ReclamacaoMgr e implementa a interface IReclamacao de RegistrarReclamacao. A classe Reclamacao é usada pela classe ReclamacaoFacadeConn para adaptar a interface provida de IReclamacaoMgt de ReclamacaoMgr à interface requerida IReclamacaoMgt de RegistrarReclamacao.

Os componentes da camada de negócio representam o núcleo do sistema. É nela que estão implementadas as regras de negócio da aplicação. O componente ReclamacaoMgr da camada de negócios define as regras para o cadastro de reclamação e a sua estrutura interna é semelhante a estrutura interna do componente de sistema RegistrarReclamacao. A única diferença é que o componente ReclamacaoMgr não tem interfaces requeridas.

5.2.5 Resultados

Além dos resultados descritos no estudo de caso do sistema de BioInformática (ver Seção 5.1.3), este estudo de caso também obteve os seguintes resultados:

(i) Conseguiu-se aplicar e analisar o modelo COSMOS completamente;

(ii) A utilização de conectores e a declaração das interfaces requeridas auxiliam na construção de componentes mais flexíveis a mudanças, e mais facilmente reutilizáveis e adaptáveis;

(iii) A integração do modelo COSMOS com uma das plataformas de desenvolvimento de sistemas mais utilizadas atualmente, a plataforma J2EE. Esta integração mostrou que o modelo COSMOS e os componentes EJBs, providos pela especificação J2EE, podem ser usados de forma complementar. Enquanto que o modelo COSMOS poderia ser usado como base para a implementação dos componentes num sistema J2EE, os componentes EJB poderiam ser usados quando for desejável usar os serviços de infra-estrutura providos pela especificação de EJB, tais como suporte transacional e distribuição. Além disso, o uso de conectores entre camadas de um sistema J2EE diminui o acoplamento entre os componentes e facilita a evolução do sistema.

(iv) A grande modularidade, o baixo acoplamento e a flexibilidade dos componentes do sistema contribuem para tornar o processo de manutenção mais facilmente administrável;

(v) Este estudo de caso também mostrou a necessidade de propor soluções para alguns aspectos do desenvolvimento baseado em componentes, tais como a instanciação e configuração dos componentes, a definição de tipos de dados etc.

5.3 Lições Aprendidas

Nesta seção, são apresentadas as lições aprendidas nos estudos de caso realizados com o modelo COSMOS. Propostas para instanciação e configuração de componentes, definição dos tipos

de dados dos sistema, que são compartilhados pelos componentes, e inserção de requisitos não-funcionais nos conectores são discutidas neste capítulo, que discute também a escalabilidade do modelo COSMOS.

5.3.1 Instanciação e Configuração dos Componentes

Pode-se dizer que a configuração de um sistema baseado em componentes consiste em duas fases: a instanciação dos componentes e conectores, e o estabelecimento das conexões entre os componentes e conectores. A instanciação dos componentes consiste na criação de seus objetos internos e na criação de uma porta de entrada para o componente, a partir da qual o usuário do componente pode acessar os seus serviços. A configuração do componente consiste na realização das ligações entre os componentes que participam da configuração, através dos conectores, ou seja, nas conexões entre as interfaces requeridas de componentes com as interfaces providas de outros componentes, usando os conectores para mediar essas conexões.

Para realizar estas tarefas, é necessário um agente configurador do sistema que conheça os componentes e conectores que irão participar da composição de software e de um repositório de componentes, que irá manter os componentes já instanciados e configurados pelo agente configurador do sistema, de modo que os componentes possam ser recuperados e suas interfaces providas acessadas em algum ponto do sistema. Tipicamente, essa instanciação e configuração dos componentes e conectores deve ser feita durante a inicialização do sistema.

No Telestrada, por se tratar de um sistema implementado na plataforma J2EE, a instanciação e configuração dos componentes e conectores foi realizada por um *Servlet* (ver Seção 2.3.1) que é executado durante a inicialização da aplicação, um *StartupServlet*. Assim, o *servlet* instancia os componentes e conectores, resolve as suas dependências e armazena as instâncias dos componentes num repositório de componentes. A Figura a seguir (Figura 44) mostra um trecho de código do *StartupServlet* que instancia os componentes e conectores e resolve as suas dependências.

```
package framework;
import javax.servlet.ServletException;□import javax.servlet.http.HttpServlet;
public class StartupServlet extends HttpServlet {

 public void init()throws javax.servlet.ServletException {
     try {

          // Instanciação dos componentes de negócio ReclamacaoMgr
          negocios.reclamacaoMgr.spec.prov.IManager negocioReclamacaoMgr =
                    negocios.reclamacaoMgr.impl.ComponentFactory.createInstance(null);

          // Instanciação dos conectores ReclamacaoMgrConn
          sistema.registrarReclamacao.spec.prov.IManager connReclamacaoMgr =
                    connNeg.reclamacaoMgrConn.impl.ComponentFactory.createInstance(null);

          // Instanciação do componente de sistema RegistrarReclamacao
          sistema.registrarReclamacao.spec.prov.IManager sisRegistrarReclamacao =
                    sistema.registrarReclamacao.impl.ComponentFactory.createInstance(null);

          // Recuperação das interfaces providas do componente de negócio ReclamacaoMgr
          negocios.reclamacaoMgr.spec.prov.IReclamacaoMgt.reclamacaoMgt = (negocios.reclamacaoMgr.spec.prov.IReclamacaoMgt.)
                    controlMgr.getProvidedInterface("negocios.reclamacaoMgr.spec.prov.IReclamacaoMgt");
          negocios.reclamacaoMgr.spec.prov.ITipoReclamacaoMgt.tipoReclamacaoMgt =
                    (negocios.reclamacaoMgr.spec.prov.ITipoReclamacaoMgt.)
                              controlMgr.getProvidedInterface("negocios.reclamacaoMgr.spec.prov.ITipoReclamacaoMgt");

          // Resolução das interfaces requeridas IReclamacaoMgt e ITipoReclamacaoMgt do componente de sistema RegistrarReclamacao
          connReclamacaoMgr.setRequiredInterface("negocios.reclamacaoMgr.spec.prov.IReclamacaoMgt",reclamacaoMgt.);
          connReclamacaoMgr.setRequiredInterface("negocios.reclamacaoMgr.spec.prov.ITipoReclamacaoMgt",tipoReclamacaoMgt.);

          sisRegistrarReclamacao.setRequiredInterface("sistema.registrarReclamacao.spec.req.IReclamacaoMgt",connReclamacaoMgr.
                    getProvidedInterface("sistema.registrarReclamacao.spec.req.IReclamacaoMgt"));
          sisRegistrarReclamacao.setRequiredInterface("sistema.registrarReclamacao.spec.req.ITipoReclamacaoMgt",connReclamacaoMgr.
                    getProvidedInterface("sistema.registrarReclamacao.spec.req.ITipoReclamacaoMgt"));

          // Armazena as instâncias dos componentes no repositório de componentes
          ComponentPool.setComponentInstance("Negocio","RegistrarReclamacaoMgr",negocioReclamacaoMgr);
          ComponentPool.setComponentInstance("Sistema","RegistrarReclamacao",sisRegistrarReclamacao);

          ...

     }catch(Exception e){
          System.out.println("StartupServlet: problema no servlet durante configuração de componentes:" + e.toString());
     }
}
```

Figura 44. *Servlet* para Inicialização e configuração dos componentes

O Trecho de código do *StartupServlet* inicializa o componente de negócio ReclamacaoMgr, o conector ReclamacaoMgrConn e o componente de sistema ReclamacaoMgr, depois resolve as dependências dos componentes e armazena as instâncias dos componentes no repositório de componentes. Esse mesmo processo é realizado para os demais componentes e conectores do sistema. O repositório de componentes mencionado acima (*ComponentPool*) é uma classe *Java* que contém métodos estáticos para armazenar, recuperar e destruir instâncias dos componentes (Figura 45).

```
package framework;
import java.util.HashMap;
import java.util.Map;

public class ComponentPool {

  static Map inputData = null;

  public static void setComponentInstance(String level, String component, Object componentInstance){
     if(inputData == null)
        inputData = new HashMap();
    String chaveHash = component+level;
    inputData.put(chaveHash,componentInstance);
  }

  public static Object getComponentInstance(String component, String level){
     if(inputData == null)
        return null;
     return inputData.get(component+level);
  }

  public boolean destroyComponenInstance(String component, String level){
    inputData.remove(component+level);
    return true;
  }
}
```

Figura 45. Repositório de componentes do Telestrada

Uma vez que o *ComponentPool* foi inicializado pelo *StartupServlet*, as instâncias dos componentes podem ser acessadas em qualquer parte do código da implementação. Como os *servlets* são restritos à linguagem *Java*, outros mecanismos, seguindo a mesma linha de raciocínio, podem ser utilizados em sistemas desenvolvidos em outras linguagens de programação. O repositório de componentes utilizado na implementação do Telestrada foi extremamente simples. Outras implementações mais sofisticadas podem ser criadas, de acordo com as necessidades de cada um.

5.3.2 Definição dos Tipos de Dados

Um componente também pode definir interfaces presentes na assinatura de alguma operação de uma interface provida pelo componente, seja como tipo de resultado ou parâmetro de entrada. Neste caso, as interfaces também devem estar presentes no modelo de especificação e podem ser supertipos de outras interfaces do mesmo modelo ou internas à implementação do componente. Tipicamente, essas interfaces compreendem os tipos de dados do sistema, que podem ser compartilhados pelos componentes que participam da composição de software. A seguir, são propostas duas abordagens para a definição de tipos de dados do sistema.

- **Pacote global para definição de tipos de dados do sistema**

Nesta abordagem, o sistema baseado em componentes possui um pacote global para definição de tipos de dados compartilhados pelos componentes do sistema. Tipicamente, o pacote global contém as interfaces dos tipos de dados e cada componente pode definir suas próprias classes de implementação para os tipos de dados (Figura 46).

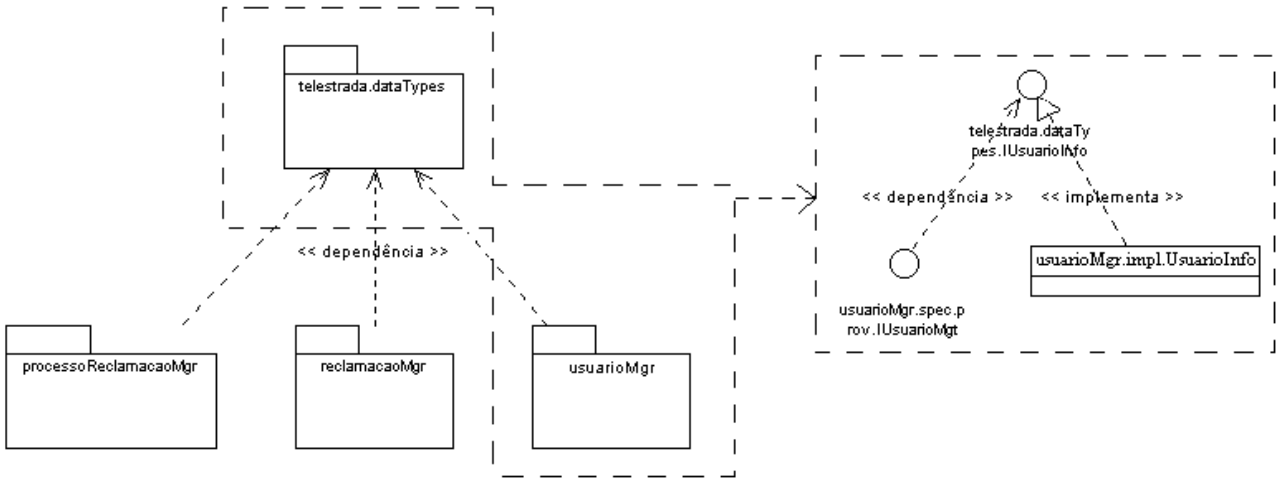

Figura 46. Pacote global para definição de tipos de dados

Na figura acima, os pacotes processoReclamacaoMgr, reclamacaoMgr e usuarioMgr, que materializam os componentes ProcessoReclamacaoMgr, ReclamacaoMgr e UsuarioMgr, têm uma relação de dependência com o pacote global de definição de tipos de dados (telestrada.dataTypes). Por exemplo, a interface provida IUsuarioMgt do componente UsuarioMgr tem uma operação registrarUsuario que recebe como parâmetro de entrada um objeto do tipo IUsuarioInfo, definido no pacote telestrada.dataTypes. Além disso, o componente UsuarioMgr também tem a sua própria implementação do tipo IUsuarioInfo, ou seja, a classe UsuarioInfo.

Essa solução tem como principal vantagem a simplicidade de implementação, uma vez que todos os componentes acessam o pacote global de definição de tipos de dados e não há a necessidade de adaptações entre as instâncias de tipos de dados que participam da comunicação entre componentes de uma composição de software, ao passo que as instâncias de tipos de dados de diferentes componentes têm o mesmo tipo. A desvantagem desta solução está na dependência dos componentes com relação ao pacote de definição de tipos de dados, ou seja, na dependência entre

classes e/ou interfaces internas ao componente e interfaces externas ao componente, que pode dificultar a reutilização do componente.

- **Tipos de dados definidos pelo próprio componente**

Uma segunda abordagem é cada componente definir seus próprios tipos de dados e as suas implementações. Porém, como os tipos de dados fazem parte da assinatura de operações das interfaces providas e requeridas pelos componentes, as interfaces que representam os tipos de dados devem ser públicas. Dessa forma, faz-se necessário estender o modelo de especificação do componente para incorporar mais um elemento de sua especificação, os seus tipos de dados. Portanto, além dos pacotes com as interfaces providas e requeridas, o modelo de especificação também deve ter um outro pacote para definir os tipos de dados presentes nas assinaturas de operações das interfaces providas e requeridas (Figura 47).

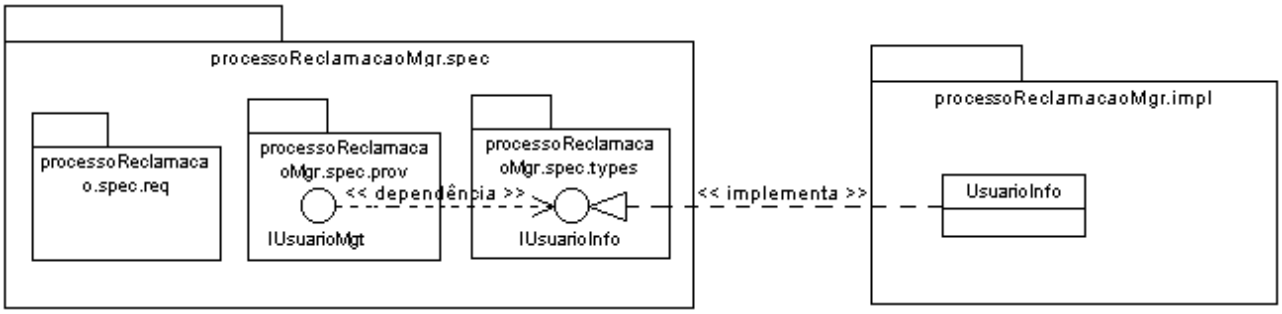

Figura 47. Tipos de dados definidos pelos próprios componentes

Na figura 47, o pacote de especificação do componente processoReclamacaoMgr foi estendido para incorporar um outro pacote *types* para a definição dos tipos de dados presentes na assinatura das operações de suas interfaces providas e requeridas. No componente processoReclamacaoMgr, a interface provida IUsuarioMgt depende do tipo de dados IUsuarioInfo que é implementado por uma classe (UsuarioInfo) do modelo de implementação. Essa abordagem facilita a reutilização do componente, uma vez que não há dependências externas ao componente. Porém, há um maior sobrecarga na implementação de um sistema baseado em componentes que use essa abordagem para definição dos tipos de dados, uma vez que outros componentes também podem usar os mesmos tipos de dados. Para isso, as interfaces que representam os tipos de dados devem ser replicadas nos pacotes de especificação de cada componente e os conectores, que

adaptam as interfaces providas e requeridas dos componentes, têm também a responsabilidade de adaptar os tipos abstratos de dados dos componentes.

As duas abordagens acima podem ser combinadas numa solução geral que contempla a definição de tipos de dados no próprio componente e a referência a tipos de dados externos ao componente. A solução geral expande o modelo de especificação do componente com o pacote *types* (Figura 47) e também expande o modelo de implementação com um pacote para definição de classes públicas cuja implementação pode ser herdada por outras classes de outros componentes (*public*). Assim, um componente pode: definir os seus tipos de dados no modelo de especificação expandido e suas implementações privadas no modelo de implementação; ou usar tipos de dados do modelo de especificação expandido de um outro componente e/ou a implementação pública desse tipo de dados (Figura 48).

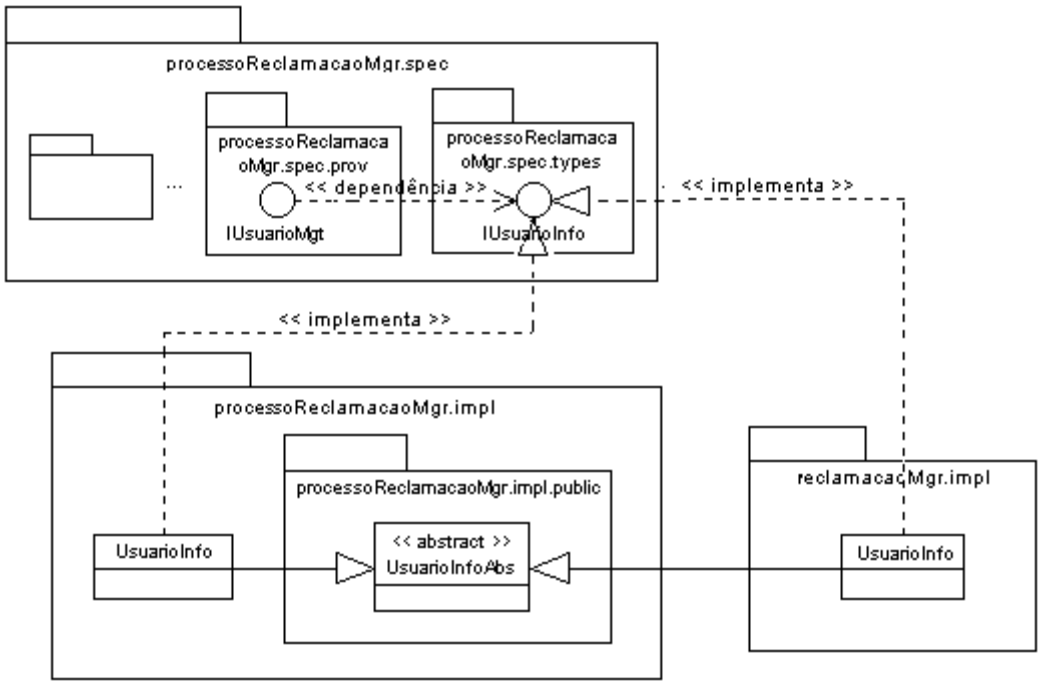

Figura 48. Tipos de dados - Solução geral

Na solução geral, o componente processoReclamacaoMgr define o tipo de dados IUsuarioInfo e a sua respectiva classe de implementação UsuarioInfo, privada ao pacote de implementação. O componente processoReclamacaoMgr também define uma classe abstrata UsuarioInfoAbs, cuja implementação pode ser herdada por classes de outros componentes para criar outras implementações do tipo de dados. Para isso, a classe UsuarioInfoAbs implementa os

métodos definidos na interface pública IUsuarioInfo, mas não implementa (realiza) a interface IUsuarioInfo, de modo que a reutilização do código torne-se mais simples e o comportamento externo dos objetos instanciados fique inteiramente sob o controle das classes concretas (ver diretriz de projeto Separação entre herança de implementação e herança de tipos - Seção 3). Essa solução geral dá maior flexibilidade ao desenvolvedor do componente, que pode optar por uma implementação totalmente independente de outros componentes ou reutilizar implementação de outros componentes. Logicamente, essa abordagem combina as vantagens e desvantagens das duas abordagens anteriores.

5.3.3 Inserção de Requisitos não-funcionais em Conectores

No modelo COSMOS, os requisitos não-funcionais podem ser inseridos na implementação dos conectores. Como resultado, a especificação e a implementação dos componentes tornam-se mais simples e os componentes mais flexíveis e adaptáveis. Em algumas composições de software, um conector pode simplesmente direcionar requisições e respostas entre os componentes (*call-return connections)*. Em outras situações, o conector pode também atuar manipulando os valores dos argumentos das requisições que passam através dele, estabelecer os protocolos de interação e a forma com a qual o encaminhamento das requisições é feito ou, ainda, adicionar novas características à interação entre componentes.

Suponha que os componentes RegistrarReclamacao e ReclamacaoMgr e o conector ReclamacaoMgrConn da Figura 27 estejam localizados fisicamente numa mesma máquina de uma rede de computadores e não há nenhum requisito não-funcional adicional à interação entre os componentes, ou seja, o componente RegistrarReclamacao faz uma requisição a um objeto que implementa a sua interface requerida IReclamacaoMgt, previamente configurado durante a instanciação e configuração dos componentes do sistema, o conector ReclamacaoMgrConn recebe essa requisição e redireciona diretamente a requisição para um objeto que implementa a interface provida IReclamacaoMgt de ReclamacaoMgr e que responde pela requisição do componente RegistrarReclamacao. Neste exemplo, suponha também que tenha sido utilizada a abordagem da Figura 47 para definição de tipos de dados do sistema, sendo necessária a realização de uma adaptação, por parte do conector ReclamacaoMgrConn, das interfaces dos tipos dados em RegistrarReclamacao e ReclamacaoMgr.

Agora suponhamos a necessidade de incorporar um requisito não-funcional para distribuir os componentes em diferentes máquinas numa rede de computadores. Para implementar esse requisito, usamos *Enterprise JavaBeans* [Ejb] que é parte da plataforma J2EE (ver Seção 2.3.1). Aplicando a diretriz para separação entre requisitos funcionais e não-funcionais (ver diretriz de projeto Separação entre requisitos funcionais e não-funcionais - Seção 3), inserimos o controle da distribuição dos componentes do sistema nos conectores. Ainda supondo a Figura 27, apenas o conector ReclamacaoMgrConn foi modificado. Supondo que o componente RegistrarReclamacao esteja instalado numa máquina A e o componente ReclamacaoMgr numa máquina B, criamos um EJB cuja interface Remota estende a interface provida IReclamacaoMgt de ReclamacaoMgr e instalamos o EJB num servidor de aplicações na máquina B. A sequência de interação entre os componentes RegistrarReclamacao e ReclamacaoMgr passa a ser a seguinte (Figura 49):

1. O componente RegistrarReclamacao na máquina A faz uma requisição a um objeto que implementa a sua interface requerida IReclamacaoMgt, previamente configurado durante a instanciação e configuração dos componentes do sistema;
2. Um objeto do conector ReclamacaoMgrConn na máquina A, que implementa a interface requerida de RegistrarReclamacao, recebe essa requisição, instancia o EJB e chama a operação correspondente na interface remota do EJB;
3. O EJB no conector ReclamacaoMgrConn na máquina B recebe essa requisição e redireciona a chamada para o objeto que implementa a interface provida IReclamacaoMgt no componente ReclamacaoMgr da máquina B.

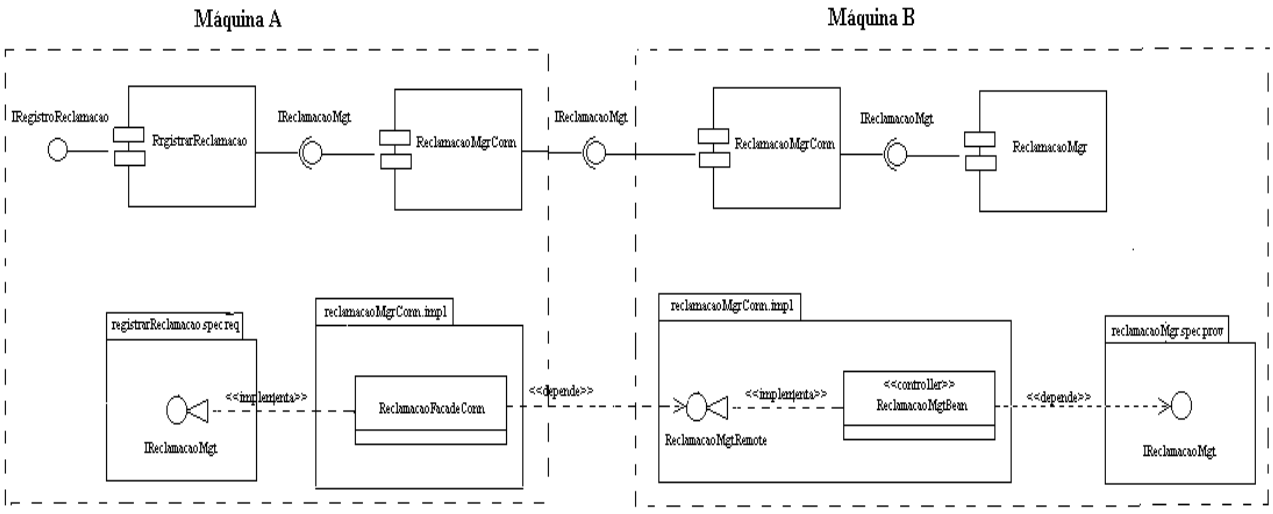

Figura 49. Inserção de requisito não-funcional de distribuição nos componentes do Telestrada

Como o EJB é um *session bean* (ver Seção 2.3.1), criado somente no momento da requisição do componente RegistrarReclamacao, ele não tem acesso ao objeto *Manager* do conector ReclamacaoMgrConn. Consequentemente, o EJB não pode chamar a operação getRequiredInterface(st) para acessar o objeto do componente ReclamacaoMgr que implementa a interface IReclamacaoMgt, previamente configurado no conector durante a inicialização da aplicação na máquina B. Dessa forma, o EJB precisa acessar o repositório de componentes da aplicação na máquina B (ver - seção 5.1), recuperar a instância do componente ReclamacaoMgr, já instanciado e configurado, obter a interface provida de ReclamacaoMgr e executar a operação correspondente no objeto que implementa a interface provida de ReclamacaoMgr. O tempo de vida do EJB (*session bean*) é limitado pelo tempo de vida do processo da máquina virtual Java no qual ele foi criado (ver Seção 2.3.1). Por essa razão, deve-se ter um cuidado maior com o ciclo de vida do objeto que implementa a interface remota do EJB. Assim é aconselhável que o conector possua mecanismos para identificar problemas na comunicação com o objeto remoto e controlar as transações criadas, de modo a preservar a consistência dos objetos que participam da interação.

Esta implementação permite que os componentes de uma aplicação possam ser instanciados em qualquer máquina de um sistema distribuído. Na implementação dos componentes, não é necessária nenhuma referência quanto a sua localização ou qualquer outro suporte para a interação de componentes distribuídos. A localização de uma instância de um componente é transparente em relação à sua funcionalidade. Já a instanciação de um conector pode vir acompanhada de um

parâmetro indicando onde estão as instâncias dos componentes que irão participar da interação. Este é o único lugar no qual pode-se associar componentes as suas localizações. Também, o fato de um conector poder controlar as interações entre os componentes através da interceptação e manipulação de requisições e respostas torna evidente a reflexividade deste elemento. Assim sendo, outros aspectos, além da comunicação, também podem ser configurados a partir dos conectores.

5.3.4 Escalabilidade do Modelo COSMOS

O COSMOS define um modelo de implementação, para descrever os elementos que compõem a implementação do componente. A implementação do componente é materializada num pacote, cujas interfaces do subpacote de especificação são públicas e as classes e eventuais interfaces do subpacote de implementação têm visibilidade restrita a esse pacote. Linguagens de programação como Java e C++ definem três níveis de visibilidade:

(i) *public*: qualquer elemento de implementação definido como *public* pode ser acessado por outros elementos de dentro ou fora do pacote no qual fora definido;

(ii) *private*: qualquer elemento de implementação definido como *private* pode ser acessado somente por outros elementos da mesma classe;

(iii) *protected*: qualquer elemento de implementação definido como *protected* pode ser acessado somente por elementos da própria classe e de suas subclasses.

Há também um outro nível de visibilidade definido como *friendly*. Classes e interfaces definidas sem modificador de visibilidade (*public*, *private* e *protected*) são ditas *friendly* e podem ser acessadas por qualquer outro elemento definido no mesmo pacote. Consequentemente, os elementos do modelo de implementação são definidos sem modificador de visibilidade e são ditos *friendly*. Se existissem subpacotes no modelo de implementação, os elementos desses subpacotes deveriam ser declarados como *public*, de forma que eles pudessem interagir. Porém, isso comprometeria o encapsulamento da implementação do componente (ver diretriz de projeto Separação explícita entre especificação e implementação - Seção 3).

A necessidade de definir subpacotes no pacote de implementação pode ser visualizada como a necessidade de definir subcomponentes, cujo uso é restrito a um determinado componente. Neste

sentido, a diferença básica entre subpacotes do pacote de implementação e subcomponentes seria quanto à localização física dos mesmos. Enquanto um subpacote do pacote de implementação estaria imediatamente um nível abaixo do pacote de implementação na hierarquia de pacotes, os pacotes de um subcomponente e do seu respectivo componente não precisariam ter obrigatoriamente relação hierárquica, ou seja, um subcomponente não precisaria ser definido dentro do modelo de implementação do respectivo componente. A dependência de um componente com relação a um subcomponente não seria externalizada como uma interface requerida. Essa dependência seria resolvida internamente à implementação do componente.

Suponha que os componentes EmailMgr e ImpressoraMgr da Figura 27 tenham sido transformados em subcomponentes de um componente maior UtilMgr que ofereça novas funcionalidades do envio de e-mails e formatação de texto para impressão (Figura 50).

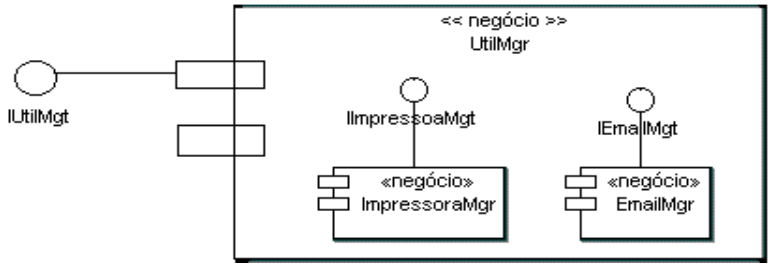

Figura 50. Subcomponentes EmailMgr e ImpressoraMgr

Para incorporar os componentes ImpressoraMgr e EmailMgr, definimos uma classe Facade (UtilFacade) para as interfaces providas IEmailMgt e IImpressoraMgt dos componentes EmailMgr e ImpressoraMgr. A classe UtilFacade também tem visibilidade restrita ao pacote de implementação de UtilMgr. Os componentes EmailMgr e ImpressoraMgr, apesar de serem subcomponentes de UtilMgr não têm nenhuma relação quanto as suas hierarquias de pacotes, ou seja, estão fisicamente localizados em estruturas de pacotes distintas (Figura 51).

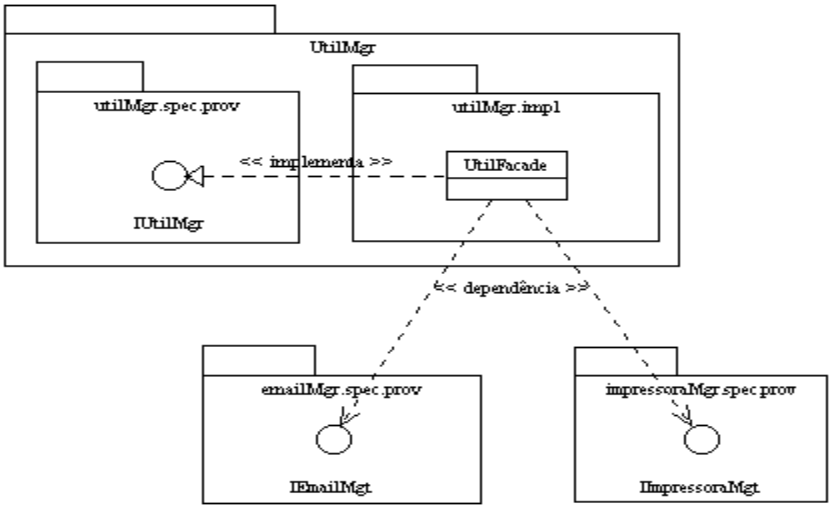

Figura 51. Esquema para a definição de subcomponentes

Caso os subcomponentes EmailMgr e ImpressoraMgr tivessem interfaces requeridas, todas elas seriam resolvidas localmente ao componente UtilMgr, ou seja, a criação e instanciação dos componentes e conectores internos ao componente UtilMgr deveria ocorrer durante a instanciação do componente UtilMgr. Assim, os subcomponentes EmailMgr e ImpressoraMgr não podem ter nenhuma interface requerida que não seja conhecida do componente UtilMgr. O suporte a definição de subcomponentes garante uma característica importante deste modelo: a sua escalabilidade.

5.4 Resumo

Este capítulo apresentou a avaliação prática do modelo COSMOS. Foram apresentados dois estudos de caso de aplicação do modelo COSMOS no desenvolvimento de sistemas reais. O primeiro consistiu no desenvolvimento de um sistema de BioInformática por uma empresa privada de desenvolvimento de software. Já o segundo, consistiu no desenvolvimento de um sistema *Web* de informações sobre as condições de conservação de rodovias. O capítulo também discutiu as lições aprendidas com a aplicação do modelo. Foram propostas abordagens para: instanciar e configurar os componentes; definir os tipos de dados do sistema, comuns aos componentes e inserir requisitos não-funcionais em conectores e também discutiu-se a escalabilidade do modelo.

No primeiro estudo de caso, o modelo COSMOS foi customizado de acordo com as necessidades da empresa e, assim, o modelo não foi aplicado em sua totalidade. Este estudo de caso teve como principais resultados: a obtenção de um processo de desenvolvimento bem definido; e a

maior rapidez na atividade de implementação, uma vez que a estruturação do sistema em componentes tornou o sistema muito mais próximo do que foi modelado e, consequentemente, muito mais fácil de implementar. Além disso, os defeitos encontrados durante a implementação foram mais facilmente localizados, devido à divisão do sistema em componentes.

O segundo estudo de caso permitiu a aplicação completa do modelo COSMOS e teve como principais resultados: a construção de componentes mais flexíveis a mudanças, reutilizáveis e adaptáveis; a integração do modelo COSMOS no desenvolvimento de um sistema para *Web* usando a plataforma J2EE; e a possibilidade de uma atividade de manutenção mais facilitada, tendo em vista a maior modularidade e o baixo acoplamento dos componentes do sistema.

Para a instanciação e configuração dos componentes, propôs-se a utilização de um *Servlet* (ver Seção 2.3.1) que é carregado sempre que a aplicação é inicializada. Como *servlets* são restritos a sistemas desenvolvidos na plataforma J2EE, o papel de instanciação e configuração dos componentes também pode ser executado por uma classe simples que instancia os componentes, resolve as suas dependências e armazena os componentes configurados num repositório de componentes. Esta solução utilizou uma classe que faz o papel de um repositório de componentes. Ela possui métodos estáticos para armazenar, recuperar e destruir instâncias de componentes, que podem ser acessados em qualquer parte da aplicação.

Já as abordagens para a definição dos tipos de dados do sistema, que são comuns aos componentes, são altamente dependentes do grau de reusabilidade que se quer dar aos componentes do sistema. Neste capítulo, foram propostas três soluções para a definição dos tipos de dados, que aumentam ou diminuem o grau de reusabilidade dos componentes.

A inserção de requisitos não funcionais nos conectores foi exemplificado através da inserção de aspectos relacionados com distribuição na comunicação entre os componentes. Utilizou um *Enterprise JavaBean* (ver Seção 2.3.1) para acessar remotamente a interface provida de um componente, que passou a estar fisicamente localizado numa outra máquina de uma rede de computadores. Esse experimento mostrou como requisitos não-funcionais podem ser inseridos numa aplicação baseada em componentes, inserindo as modificações necessárias nos conectores.

Por fim, discutiu-se a escalabilidade do modelo COSMOS para a definição de subcomponentes de forma recursiva. O apoio à definição de subcomponentes também habilita a definição de agentes configuradores (*Managers*) para um conjunto inteiro de componentes, ao invés de um único componente como foi mostrado neste trabalho.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho apresentou um modelo de estruturação de componentes que mapeia arquiteturas de componentes para linguagens de programação. Esse modelo é chamado de COSMOS e age, principalmente, na organização do sistema em termos de seus componentes, conectores e interações entre eles. O modelo COSMOS habilita a implementação de sistemas baseados em componentes, mantendo a conformidade da implementação com relação à arquitetura proposta para o sistema. Abstrações arquiteturais, tais como componentes, conectores, interfaces e conexões, são materializadas usando construções disponíveis nas linguagens de programação. Com isso, pretende-se diminuir a distância entre uma arquitetura de software baseada em componentes e sua implementação.

A solução apresentada neste trabalho é dividida em três modelos inter-relacionados: (1) o modelo de especificação, que descreve a visão externa do componente; (2) o modelo de implementação, que descreve a estrutura interna do componente; e (3) o modelo de conectores, que descreve as conexões entre os componentes e conectores, numa composição de software. Esses modelos foram construídos de modo a implementar algumas diretrizes de projeto, tais como: a materialização de elementos arquiteturais, a inserção de requisitos não-funcionais nos conectores, a separação explícita entre especificação e implementação, a declaração explícita das dependências dos componentes, a separação entre herança de código e herança de implementação e o baixo acoplamento entre classes de implementação. Essas diretrizes trazem diversos benefícios, dentre os quais podemos citar:

(i) Obtenção de um conjunto de diretrizes de projeto bem definidas para o mapeamento das descrições arquiteturais em elementos de linguagens de programação;

(ii) Maior proximidade da arquitetura do software com relação a sua implementação;

(iii) Possibilidade maior de reuso dos componentes, tendo em vista a maior flexibilidade na interação entre os mesmos, que podem ser desenvolvidos independentemente e substituídos ou modificados mais facilmente;

(iv) Maior facilidade na evolução da implementação dos componentes, simplificando a manutenção e aumentando a robustez da implementação dos componentes.

6.1 Contribuições

Como contribuições deste trabalho, ressaltamos:

1. **O modelo COSMOS**: A principal contribuição deste trabalho foi à definição de um modelo de estruturação de componentes que habilita a implementação de sistemas de software baseados em componentes, definindo um mapeamento para implementar abstrações arquiteturais usando as construções disponíveis em linguagens de programação. O Capítulo 4 descreve as características e os elementos que compõe o modelo.
2. **A integração de diretrizes de projeto e idéias de pesquisas anteriores no Modelo COSMOS**: Para definir o modelo COSMOS, foram incorporados conceitos e idéias de pesquisas anteriores em arquitetura de software, desenvolvimento baseado em componentes e orientação a objetos, e também algumas diretrizes de projeto que visam à construção de componentes mais flexíveis a mudanças, reusáveis e adaptáveis. Tais conceitos e diretrizes de projeto estão descritos no Capítulo 3.
3. **A integração do modelo COSMOS com tecnologias de componentes modernas**: exemplificamos a aplicação da teoria apresentada para a construção de componentes de software, através de dois estudos de casos de sistemas desenvolvidos em Java, mais especificamente na plataforma J2EE, que vem sendo muito utilizada pela comunidade de desenvolvimento de software. O Capítulo 5 descreve estudos de caso de integração da plataforma J2EE com o modelo COSMOS.
4. **O sistema Telestrada**: apresentamos neste trabalho o Telestrada, um sistema real cuja especificação foi utilizada para a construção de um sistema baseado em componentes, onde pudemos aplicar a avaliar, em sua totalidade, o que foi proposto neste trabalho.

Realizamos um trabalho, desde a fase de identificação de componentes e definição da arquitetura, até a implementação. O Capítulo 5 mostra um dos módulos do sistema e como foi realizado o trabalho de mapeamento da descrição arquitetural para a implementação.

5. **As lições aprendidas com a aplicação do COSMOS**: foram discutidos alguns aspectos da implementação de um sistema baseado em componentes, tais como: a definição de tipos de dados do sistema, a inserção de requisitos não-funcionais nos conectores, a instanciação e configuração dos componentes. Algumas soluções para contemplar os aspectos citados acima foram propostas. Tais aspectos foram estudados no Capítulo 5.

Publicações

Este trabalho resultou em duas publicações em conferências nacionais e internacionais. São elas:

- Um Modelo de Componentes Java para Evolução de Software
  Autores: Moacir C. da S. Júnior, Paulo A. de C. Guerra e Cecília M. F. Rubira
  Conferência: Terceiro Workshop de Desenvolvimento Baseado em Componentes - WDBC'2003- São Carlos - São Paulo - Brasil, Setembro 2003

- A Java Component Model for Evolving Software Systems
  Autores: Moacir C. da S. Júnior, Paulo A. de C. Guerra e Cecília M. F. Rubira
  Conferência: 18th IEEE Conference on Automated Software Engineering - ASE'03 - Montreal - Quebec - Canada, October 2003

As cópias destes trabalhos encontram-se disponíveis no endereço http://www.ic.unicamp.br/~ra007278/publicacoes.html

6.2 Trabalhos Futuros

Podemos propor algumas linhas de pesquisa como extensões e melhorias do trabalho apresentado nesta dissertação.

Em primeiro lugar, está a necessidade de automatizar o mapeamento de uma descrição de arquitetura de software em elementos do modelo de objetos, usando o modelo COSMOS como base para a transformação dos elementos arquiteturais em implementação. A geração automática de código diminuiria consideravelmente o esforço de implementação dos elementos comuns a todos os componentes do sistema. Seria também possível à geração de conectores padrão para conexões do tipo *call-return*¸ ou até mesmo, conectores mais complexos.

Uma outra necessidade seria identificar métricas que pudessem ser usadas para medir a eficácia da solução proposta neste trabalho. As métricas poderiam ser usadas para medir a conformidade da arquitetura de software com relação a sua implementação, a flexibilidade e a adaptabilidade dos componentes produzidos.

Destacamos também a necessidade da realização de um trabalho para comprovar a utilidade do modelo COSMOS para facilitar a manutenção de sistemas de software baseados em componentes. Acreditamos que o modelo proposto oferece bons mecanismos para facilitar a evolução e a incorporação de novos requisitos de software, mas ainda falta um trabalho experimental mais voltado neste sentido.

Referências Bibliográficas

[Aldrich+02] J. Aldrich, C. Chambers and D. Notkin. Archjava: connecting software architecture to implementation. *In Proceedings of the 24th International conference on Software engineering*, pages 187–197. ACM Press, 2002.

[Apache] *Apache Group*. *The Jakarta Project Site,* Disponível em http://jakarta.apache.org/struts.

[Bambars+02] J. J. Bambars, P. R. Allen, et al. *J2EE Unleashed*. Sams, first Edition, 2002

[Banerjee+87] J. Banerjee, W. Kim, and H. F Korth Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *In Proc. ACM SIGMOD Conference on the Management of Data*, San Francisco, May, 1987.

[Bernstein96] P. A. Bernstein: Middleware: A Model for Distributed System Services. *Communications of the ACM*, Vol. 39, No. 2, February 1996

[Bishop+96] J. Bishop and R. Faria. *Connectors in Configuration Programming Languages: are They Necessary?*, *IEEE Third International Conference on Configurable Distributed Systems*, Annapolis, May, 1996.

[Björkander03] [Björkander] M. Björkander and C. Kobry. Architecting Systems with UML 2.0, *IEEE Software*, July, 2003.

[Booch98] G. Booch. *The Visual Modeling of Software Architecture for the Enterprise* - Rational Software Corporation, 1998.

[Buschmann+96] F. Buschmann, R. Meunier, et al. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[Casais89] E. Casais. *Reorganizing an Object System. In Object Oriented Development,* Centre Universitaire d'Informatique, Université de Genève, pages 161-189, 1989.

[Chambers96] C. Chambers. Towards Reusable, Extensible Components. *ACM Computing Surveys*, 28, December 1996.

[Cheesman+01] J. Cheesman and J. Daniels *UML Components: A Simple Process for Specifying Component-Based Software,* Addison-Wesley, 2001

[Com] Microsoft Developer Network Library, *Common Object Model Specification*, Microsoft Corporation

[Coward99] D. Coward. *Java Servlet Specification* version 2.3, Disponível em http://java.sun.com/products/servlet, 17th September 1999.

[Dmitriev98] M. Dmitriev98. The First Experience of Class Evolution Support in PJama. *In Malcolm Atkinson and Mick Jordan, editors*, The Third Persistence and Java Workshop, Tiburon, California, September, 1998.

[DotNet] Microsoft .NET em http://www.imasters.com.br

[DeRemer+76] F. DeRemer and H. H Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering* 2, 2 (1976), 80-86.

[D'Souza+98] Desmond d'Souza and Alan C. Will. *Objects, Components and Frameworks with UML. The Catalysis Approach* M.A.Addison-Wesley 1998

[Ejb]Sun Microsystems. *Enterprise java beans specification.* Version 2.0, Final Release, 22th August 2001.

[Fielding+99] R. Fielding et. al. *Hipertext Transfer Protocol – HTTP/1.1*. Network Working Group, rfc 2616, proposed standard edition, June 1999.

[Flatt99] M. Flatt, *Programming Languages for Reusable Software Components*, PhD Thesis, Rice University, July, 1999.

[Gamma+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Softwar*e, Addison-Wesley, 1995.

[Goguen86] J. A. Goguen. Reusing and Interconnecting Software Components. *IEEE Computer*, pages 16-27, February 1986.

[Huang+03] G. Huang, H. Mei and Q. Wang. Towards Software Architecture at Runtime. *Software Engineering Notes*, 28(2). March, 2003.

[Hürsh95] W. L. Hürsch Should Superclasses be Abstract?, *ECOOP'94*, pages.12-31, July, 1994.

[Jacobson+97] I. Jacobson, M. Griss, P. Jonsson; *Software Reuse: Architecture, Process and Organization for Business Success*, Addison Wesley, 1997.

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

[JavaBeans] Sun Microsystems. *The javabeans 1.01 specification*, Disponível em http://java.sun.com/products/javabeans/docs/beans.1.01.pdf

[Java02] *Java 2 Platform*, Standard Edition, Version 1.4.1, July 2002. http://java.com/j2se/1.4.1/index.html

[J2ee] Sun Microsystems. *Java 2 platform, enterprise edition*, *v. 1.3 api specification,* Disponível em http://java.sun.com/j2ee/sdk_1.3/techdocs/api.

[Jsp] Sun Microsystems. *Javaserver pages specification version 1.2*, Disponível em http://java.sun.com/products/jsp, 27th August 2001.

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

[Luckham+00] D. C. Luckham, J.Vera, and S. Meldal. *Key Concepts in Architecture Definition Languages, In Foundations of Component-Based Systems*, Cambridge University Press, 2000, pp. 23-25.

[Lüer+01] C. Lüer, D. S. Rosenblum. WREN-An Environment for Component-Based Development*, In the Proc. of the 8th European Software Engineering Conference,* Austria, Sep 2001.

[Magee+96] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. *Software Engineering Notes* 21, 6 (1996), 3-14.

[McDirmid+01] S. McDirmid, M.Flatt, W. C. H. Jiazzi, "Newage components for old-fashioned Java". *In Proc. of ACM SIGPLAN OOPSLA*, ACM SIGPLAN Notices 36(11):211- 222, November, 2001.

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

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

[Mikhajlov+97] L. Mikhajlov and E. Sekerinski. The Fragile Base Class Problem and Its Solution, TUCS Research Report 117, May 1997.

[Monroe+97] R .T Monroe, A. Kompanek, R. Melton and D. Garlan, David. *A*rchitectural styles, design patterns, and objects, *IEEE Software*, pages 43-52, January, 1997.

[Nierstrasz+95] O. Nierstrasz and D. Tsichritzis.*Object-Oriented Software Composition*. The Object Oriented Series Prentice-Hall, 1995.

[Ommering+00] R. van Ommering, et al. The Koala Component Model for Consumer Electronics Software. *Computer* 33, 3 (2000), 33-85.

[Orfali+96] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons., 1996

[Penney+87] D. J . Penney, and J. Stein. Class Modification in the GEMSTONE object-oriented DBMS. *In Proc. OOPSLA'87*, pages 111-117, October, 1987.

[Porter92] H. Porter, Separating the Subtype Hierarchy from the Inheritance of Implementation. *Journal of Object-Oriented Programming*, 4(9):20-29, February 1992.

[Pressman01] R. S. Pressman. *Software Engineering: A practitioner's Approach* 5th edition, McGraw-Hill, 2001

[Rmi] Sun Microsystems. *Java Remote Method Invocation (RMI)*, Disponível em http://java.sun.com/products/jdk/1.2/docs/guide/rmi.

[Rosenblum+00] D. S. Rosenblum, R. Natarajan. Supporting Architectural Concerns in Component-Interoperability Standards. *IEEE Proceedings-Software 147(6)*, 2000.

[Rumbaugh+99] J. Rumbaugh, I. Jacobson e G. Booch. *Unified Modeling Language Reference Manual,* Addison-Wesley, 1999.

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

[Shaw+97] M. Shaw, P.Clements. A Filed Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems, *In Proceedings of the First International Computer Software and Applications Conference*, April, 1997.

[Som] *IBM's System Object Model (Som): Making reuse a reality*. IBM Corporation, Object Technology Products Group, Austin Texas.

[Sommerville01] I. Sommerville *Software Engineering,* Addison-Wesley, 2001

[StiKeleather94] J. StiKeleather. Why Distributed Computing is inevitable. *Object Magazine*, pages 35-39, March 1994.

[Struts] *Apache Jakarta Project. Struts*, Disponível em http://jakarta.apache.org/struts.

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

[Szyperski00] C. Szyperski. Component Software and the Way Ahead, *In Foundations of Component-Based Systems*, N.Y.: Cambridge University Press, 2000

[Taenzer89] D. Taenzer, M. Gandi, S. Podar. Problems in object-oriented software reuse. *Proceedings ECOOP'89*, Cambridge University Press. Nottingham, pages 25-38, July, 1989.

[Taylor+95] R. N. Taylor et al. *A component- and message-based architectural style for GUI software, In* Proceedings of the 17th International Conference on Software Engineering, pages 295-304, Seattle, 1995.

[Uml99] *UML Revision Task Force, OMG Unified Modeling Language Specification*, v. 1.3, document ad/99-06-08. Object Management Group, June 1999.

[Websphere] IBM WebSphere Software Platform http://www.ibm.com/websphere

[Werner+00] Werner C. M. L , Braga R. M. M. *Desenvolvimento baseado em componentes* XIV SBES - Mini-Cursos / Tutoriais Anais, João Pessoa, 2000.

[Whitehead+95] E. J. Whitehead et al. Software Architecture: Foundation of a Software Component Marketplace. In *Proc. First International Workshop on Architectures for Software Systems.* ACM, New York, pages 276-282, 1995.