

Uma Infra-estrutura de Software para Apoiar a Construção de Arquiteturas de Software Baseadas em Componentes

Este exemplar corresponde à redação final da
Dissertação devidamente corrigida e defendida
por Tiago César Moronte e aprovada pela
Banca Examinadora.

Este exemplar corresponde à redação final da Tese/Dissertação devidamente corrigida e defendida por: <u>Tiago César Moronte</u>		
e aprovada pela Banca Examinadora.		
Campinas, <u>06</u> de <u>Janeiro</u>	de	<u>2008</u>
<u>Renato de Azevedo</u> COORDENADOR DE PÓS-GRADUAÇÃO CPG-IC		

Campinas, 23 de fevereiro de 2007.

Cecília Mary Fischer Rubira
Profa. Dra. Cecília Mary Fischer Rubira
(Orientadora)

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

UNIDADE BC
Nº CHAMADA: M829i
T/UNICAMP
V. _____ EX. _____
TOMBO BCCL 16006
PROC 16.P.00.129.08
C 0
PREÇO 11.00
DATA 19/03/08
BIB-ID 927755

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Moronte, Tiago César

M829i Uma infra-estrutura de software para apoiar a construção de
arquiteturas de software baseadas em componentes / Tiago César
Moronte -- Campinas, [S.P. :s.n.], 2007.

Orientador : Cecília Mary Fischer Rubira

Dissertação (mestrado) - Universidade Estadual de Campinas,

Instituto de Computação.

1. Engenharia de software. 2. Software – Desenvolvimento. 3.
Software – Reutilização. 4. Software - Validação. I. Rubira, Cecília
Mary Fischer. II. Universidade Estadual de Campinas. Instituto de
Computação. III. Título.

Título em inglês: A software infrastructure to support component based software architecture construction.

Palavras-chave em inglês (Keywords): 1. Software engineering. 2. Software development.
3. Software reusability. 4. Software validation.

Área de concentração: Sistemas de Informação

Titulação: Mestre em Ciência da Computação

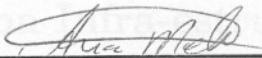
Banca examinadora: Profa. Dra. Ana Cristina Vieira de Melo (IME-USP)
Profa. Dra. Cecília Mary Fischer Rubira (IC-UNICAMP)
Prof. Dr. Hans Kurt Edmund Liesenberg (IC-UNICAMP)
Profa. Dra. Ariadne Maria Brito Rizzoni de Carvalho (IC-UNICAMP)

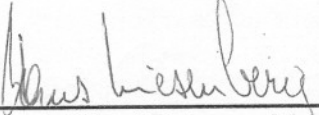
Data da defesa: 23-02-2007

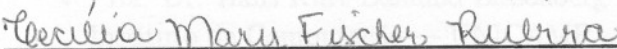
Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 23 de fevereiro de 2007, pela Banca examinadora composta pelos Professores Doutores:


Profa. Dra. Ana Cristina Vieira Melo
IME - USP.


Prof. Dr. Hans Kurt Edmund Liesenberg
IC – UNICAMP.


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

Uma Infra-estrutura de Software para Apoiar a Construção de Arquiteturas de Software Baseadas em Componentes

Tiago César Moronte¹

Fevereiro de 2007

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)
- Profa. Dra. Ana Cristina Vieira de Melo
Instituto de Matemática e Estatística — USP
- Prof. Dr. Hans Kurt Edmund Liesenberg
Instituto de Computação — UNICAMP
- Profa. Dra. Ariadne Maria Brito Rizzoni de Carvalho (Suplente)
Instituto de Computação — UNICAMP

¹Apoio financeiro da Capes

Resumo

Os paradigmas de arquitetura de software e de desenvolvimento baseado em componentes (DBC) são abordagens complementares para o desenvolvimento de sistemas de software. O DBC se baseia na construção de sistemas através da integração de componentes de software reutilizáveis. A arquitetura de software auxilia na forma como estes componentes são integrados levando em consideração atributos de qualidade, tais como confiabilidade e distribuição. Entretanto, observa-se atualmente a falta de consenso entre os conceitos, termos e definições utilizados nas abordagens de arquitetura de software e de DBC, dificultando a integração das respectivas técnicas e ferramentas. As ferramentas e ambientes atuais para descrição de arquiteturas de software não apóiam todas as fases dos processos de DBC, normalmente não geram implementações das arquiteturas e não implementam conceitos importantes de DBC, tais como especificações de interfaces providas e requeridas. Por outro lado, ferramentas e ambientes DBC atuais, em geral, são baseados em modelagem UML e não englobam todos os conceitos presentes em arquitetura de software, tais como estilos arquiteturais e uso explícito de conectores.

Este trabalho apresenta uma infra-estrutura de software para construção de arquiteturas de software baseadas em componentes, composta por um conjunto de ferramentas que estendem o ambiente integrado de desenvolvimento Eclipse. As ferramentas foram construídas sobre um metamodelo conceitual integrado para arquitetura de software e DBC, que define e relaciona os conceitos existentes nas duas abordagens. Esta infra-estrutura faz parte do ambiente BELLATRIX, um ambiente integrado de desenvolvimento que oferece apoio ao DBC com ênfase na arquitetura de software. As ferramentas apóiam a construção de arquiteturas de software baseadas em componentes desde a sua especificação, passando pelo seu projeto até a sua materialização em forma de código. O modelo de componentes utilizado é o COSMOS, um modelo de implementação de componentes que materializa os conceitos de arquiteturas de software em uma linguagem de programação. No caso do ambiente BELLATRIX, a linguagem de programação adotada é Java.

palavras-chave: arquitetura de software, verificação de arquitetura, desenvolvimento baseado em componentes, ambiente integrado de desenvolvimento, Eclipse.

Abstract

Component-based development (CBD) and architecture-centric development are two complementary approaches for developing software systems. CBD is based on the construction of systems using the integration of reusable software components. Software architecture centric development complements the CBD paradigm because it is responsible for the component integration, achieving the final system's desired quality requirements, such as dependability and distribution. However, there is a lack of consensus among the concepts, terms, and definitions used in the software architecture and CBD paradigms, hindering the integration of techniques and tools. Existing software architecture environments and tools do not support all the phases involved in CBD process, normally do not generate architecture implementations and do not implement the main CBD concepts, e.g. specification of provided and required interfaces. CBD tools and environments, in general, use UML modeling and do not cover the main software architecture concepts, e.g. architectural styles and architectural connectors.

In this work, we propose a software infrastructure to construct component-based software architectures. It has been built as a set of tools that extend the Eclipse integrated development environment. These tools were constructed based on an integrated conceptual metamodel for software architectures and CBD. This metamodel defines and relates the main concepts of the two paradigms. The infrastructure is included in the BELLATIX environment, an integrated development environment that supports CBD with emphasis on software architecture. The tools support the construction of component-based software architectures since the specification phase, through the design, until its materialization in code. The component model used is COSMOS, a component implementation model that materializes the elements of a software architecture using the concepts available in object-oriented programming languages. In BELLATIX environment, the adopted programming language is Java.

key words: software architecture, architecture analysis, component-based development, integrated development environment, Eclipse.

Agradecimentos

Gostaria de agradecer à profa. Cecília Rubira pela paciência e pela disponibilidade na orientação deste trabalho, compartilhando comigo um pouco de seu conhecimento.

Agradeço também à CAPES e ao projeto COMPGOV pelo apoio financeiro a este projeto.

Gostaria de agradecer aos professores Ana Cristina e Hans pelas opiniões, críticas e sugestões, que possibilitaram a melhoria deste trabalho.

Em especial agradeço à minha querida esposa Graciela, que tanto me ajudou durante todo o caminho percorrido no mestrado. Agradeço sua paciência, carinho, apoio e, em especial, toda sua compreensão. Agradeço aos meus pais que sempre me ajudaram, me encorajando e apoiando em todos os momentos. Um agradecimento especial também aos meus irmãos, Tassiana e Bruno, que sempre compartilharam comigo as dificuldades e as vitórias, me aconselhando e apoiando. Agradeço também à minha avó Maria, meu exemplo de bondade.

Um agradecimento especial também aos companheiros e amigos do IC. Primeiramente ao Paulo Astério, Fernando e Patrick, pelos ótimos conselhos e pela paciência em contribuir com o meu trabalho. Em especial ao Tomita, meu amigo e companheiro de implementação, que tanto me ajudou e ensinou durante este projeto. Aos meus amigos Leonel e Tizzei pelo amizade e ajuda. Aos amigos Ivan e Ana Elisa por toda a ajuda na etapa final deste projeto. Ainda, aos demais amigos do LSD.

Agradeço também aos professores e funcionários do Instituto de Computação da Unicamp, que me auxiliaram durante as disciplinas cursadas e durante a elaboração deste trabalho.

Conteúdo

Resumo	vii
Abstract	ix
Agradecimentos	xi
1 Introdução	1
1.1 Contexto	1
1.2 Motivação e Problema	2
1.3 Solução Proposta	5
1.4 Trabalhos Relacionados	7
1.5 Organização deste Documento	10
2 Fundamentos de Arquitetura de Software e DBC	13
2.1 Arquitetura de Software	13
2.1.1 Estilos Arquiteturais	13
2.1.2 Linguagens de Descrição de Arquitetura (ADL) e ACME	15
2.2 Desenvolvimento Baseado em Componentes	17
2.2.1 Especificação, Implementação e Instanciação de Componentes	17
2.2.2 Processos de Desenvolvimento Baseado em Componentes	17
2.3 O Modelo de Implementação de Componentes COSMOS	19
2.4 O Ambiente Eclipse	21
2.5 Resumo	22
3 Um Metamodelo Integrado para Arquitetura de Software e DBC	23
3.1 Metamodelo para Arquitetura de Software	24
3.1.1 Metamodelo Conceitual de Arquitetura de Software	24
3.1.2 Metamodelo Conceitual de Estilos Arquiteturais	25
3.1.3 Restrições Aplicadas ao Metamodelo	27
3.2 Metamodelo para DBC	29

3.2.1	Visão Geral	29
3.2.2	Metamodelo Conceitual de Especificação de Componentes	30
3.2.3	Metamodelo Conceitual de Arquiteturas de Software Baseadas em Componentes	31
3.2.4	Restrições Aplicadas ao Metamodelo	32
3.2.5	Exemplo de Uso do Metamodelo	33
3.3	Metamodelo para Configuração Arquitetural de Componentes	34
3.3.1	Visão Geral	34
3.3.2	Restrições Aplicadas ao Metamodelo	35
3.3.3	Exemplo de Uso do Metamodelo	36
3.4	Resumo	37
4	Uma Infra-estrutura de Software para Apoiar a Construção de Arquiteturas de Componentes	39
4.1	Requisitos para a Infra-estrutura de Software	39
4.1.1	Requisitos para modelagem de arquiteturas de componentes	40
4.1.2	Requisitos para ferramentas de apoio a construção de arquiteturas de componentes	42
4.2	Casos de Uso	44
4.3	Comparação com Outras Ferramentas de Arquiteturas de Componentes	46
4.4	Ferramentas para Modelagem de Arquiteturas de Software Baseadas em Componentes	46
4.4.1	Editor de Arquiteturas de Componentes	48
4.4.2	Editor de Estilos Arquiteturais	50
4.4.3	Verificador de Arquiteturas de Componentes	53
4.4.4	Importador/Exportador de Arquiteturas de Componentes	53
4.5	Ferramentas para Construção de Arquiteturas de Software Baseadas em Componentes	56
4.5.1	Editor de Configuração de Componentes	56
4.5.2	Geração de Código em Modelo COSMOS	57
4.6	Implementação da Ferramenta	60
4.6.1	Arquitetura de Componentes da Infra-estrutura de Software	60
4.6.2	Modelo de Componente Utilizado	62
4.6.3	Implementação do Metamodelo	63
4.6.4	Tecnologia Utilizada para a Geração de Código em COSMOS	65
4.7	Resumo	65

5	Estudos de Casos de Uso da Infra-estrutura de Software Proposta	67
5.1	Estudo de Caso: Sistema Financeiro	68
5.1.1	Objetivo	68
5.1.2	Descrição do Estudo de Caso	68
5.1.3	Descrição do Sistema	69
5.1.4	Especificação do Sistema	70
5.1.5	Execução do Estudo de Caso	74
5.1.6	Avaliação e Resultados Obtidos	77
5.2	Estudo de Caso: Um Ambiente para Testes de Componentes de Software .	79
5.2.1	Objetivo	79
5.2.2	Descrição do Estudo de Caso	79
5.2.3	Descrição do Sistema	80
5.2.4	Especificação do Sistema	82
5.2.5	Execução do Estudo de Caso	82
5.2.6	Avaliação dos Resultados Obtidos	83
5.3	Estudo de Caso: Um Sistema para Controle de Matrículas	86
5.3.1	Objetivo	86
5.3.2	Descrição do Estudo de Caso	86
5.3.3	Descrição do Sistema	87
5.3.4	Especificação do Sistema	88
5.3.5	Execução do Estudo de Caso	89
5.3.6	Avaliação dos Resultados Obtidos	90
5.4	Consolidação Geral dos Resultados Obtidos	91
5.5	Resumo	91
6	Conclusões e Trabalhos Futuros	93
6.1	Contribuições	94
6.2	Trabalhos Futuros	95
6.3	Conclusão Final	97
A	Sintaxe Linguagem OCL	99
B	Estilo DBC em Linguagem Acme	101
C	Questionário de Avaliação da Infra-estrutura Proposta	103
C.1	Avaliação Geral	103
C.2	Avaliação Complementar para o Verificador de Arquiteturas	104
C.3	Avaliação Complementar para o Gerador de Código	105

Lista de Tabelas

4.1	Funcionalidades disponibilizadas pelas ferramentas.	46
5.1	Interfaces e casos de uso para os componentes de sistema.	73
5.2	Tipos de componentes do estilo em camadas do sistema de cheques.	74
5.3	Tipos de conectores do estilo em camadas do sistema de cheques.	74
5.4	Tipos de componentes do estilo do componente ideal do sistema de cheques.	76
5.5	Tipos dos conectores do estilo do componente ideal do sistema de cheques.	76
5.6	Notas atribuídas pelos voluntários na avaliação das ferramentas.	77
5.7	Notas atribuídas a ferramenta para especificação do ambiente de testes.	84
5.8	Notas atribuídas para as ferramentas.	90

Lista de Figuras

1.1	Visão geral do ambiente BELLATRIX.	6
1.2	Infra-estrutura de software para construção de arquiteturas de componentes.	8
2.1	Exemplo do estilo arquitetural em camadas.	14
2.2	Elementos que descrevem uma estrutura arquitetural em Acme.	16
2.3	Especificação, implementação e especificação de componentes.	17
2.4	Visão geral das fases do processo UML Components	19
2.5	Visão geral do modelo COSMOS.	21
3.1	Metamodelo conceitual de arquitetura de software.	24
3.2	Metamodelo conceitual de estilos arquiteturais.	26
3.3	Restrições OCL para arquitetura de software.	28
3.4	Metamodelo conceitual para DBC.	29
3.5	Metamodelo conceitual de especificação de componentes.	30
3.6	Metamodelo conceitual de uma arquitetura de componentes.	32
3.7	Restrições OCL para DBC.	33
3.8	Especificação do componente SistemaHotel.	33
3.9	Implementação do componente SistemaHotel.	34
3.10	Metamodelo conceitual de configuração de componentes.	35
3.11	Restrições OCL para configuração de componentes.	36
3.12	Configuração arquitetural do componente SistemaHotel.	36
4.1	Casos de uso da infra-estrutura para construção de arquiteturas de componentes.	44
4.2	Visão geral das ferramenta de modelagem de arquiteturas de componentes.	47
4.3	Exemplo de uma arquitetura de componentes.	49
4.4	Editor de Estilos - modo texto.	51
4.5	Editor de Estilos - modo gráfico.	52
4.6	Exemplo de uma validação de uma arquitetura de componentes.	53
4.7	Exemplo de uma mensagem de erro do validador de arquiteturas.	54
4.8	Exemplo de um componente que segue o estilo DBC.	55

4.9	Visão geral das ferramentas para construção de arquiteturas de componentes.	56
4.10	Exemplo de uma configuração de componentes.	58
4.11	Evolução do modelo COSMOS para representar tipos de dados.	59
4.12	Arquitetura de componentes da infra-estrutura de software.	62
4.13	Estilo arquitetural MVC utilizado pela infra-estrutura de software.	63
5.1	Casos de uso do sistema de cheques.	71
5.2	Estilo arquitetural em camadas do sistema de cheques.	72
5.3	Arquitetura de componentes do sistema de cheques.	75
5.4	Visão geral do ambiente para testes de componentes de software.	80
5.5	Casos de uso do ambiente para testes de componentes de software.	81
5.6	Arquitetura de componentes do ambiente de testes.	83
5.7	Casos de uso do sistema de controle de matrículas.	88
5.8	Arquitetura de componentes do sistema de controle de matrículas.	89

Capítulo 1

Introdução

1.1 Contexto

O aumento na demanda por sistemas computacionais, ocorrido nos últimos anos, tem exigido a construção de sistemas cada vez maiores e mais complexos. Este fenômeno pode ser observado principalmente no contexto das empresas, onde a construção de novos sistemas de software busca aumentar a produtividade e o poder competitivo das organizações. Desta forma, espera-se que estes sistemas de software possam ser construídos mais rapidamente, com menores custos e atingindo um maior nível de qualidade.

A **arquitetura de software** é uma abstração que auxilia na construção desses sistemas de software grandes e complexos. A arquitetura de software representa a estrutura de um sistema e é composta de elementos de software, das propriedades externamente visíveis destes elementos e da relação entre eles [6]. A partir da arquitetura é possível identificar responsabilidades ligadas a cada parte do sistema e determinar suas formas de interação. Cada parte do sistema representa um **componente arquitetural** e os elementos responsáveis por realizar as interações entre estes componentes arquiteturais são os **conectores arquiteturais**. Um conector pode ser responsável também por uma parcela dos aspectos de qualidade (ou não-funcionais) do sistema, tais como distribuição e segurança. Um conjunto de elementos arquiteturais, ligados a partir de conectores arquiteturais, forma uma **configuração arquitetural**. Os principais benefícios do projeto arquitetural de um sistema são: (1) possibilidade de observar o sistema como um todo, abstraindo detalhes de implementação, (2) facilidade na tomada de decisões prévias relativas a projeto e (3) auxílio na comunicação mútua entre os interessados¹ (arquitetos, projetistas, clientes, entre outros) [14]. Arquiteturas de software podem ainda seguir **estilos arquiteturais**, que são conjuntos de restrições que são impostas sobre uma ar-

¹do inglês: *stakeholders*

quitetura, tendo como aspectos positivos apresentar atributos de qualidade conhecidos e representar soluções também conhecidas para uma classe de sistemas. Muitos sistemas de grande porte possuem suas partes projetadas de modo a combinar diferentes estilos arquiteturais.

O **Desenvolvimento Baseado em Componentes (DBC)** é uma abordagem que permite a construção de sistemas de software de forma mais rápida, eficiente e com maior qualidade, através da utilização de componentes de software previamente desenvolvidos [3]. Um **componente de software** é uma unidade executável de composição que se integra ao ambiente onde está inserido a partir de suas interfaces, que possui dependências explícitas de contexto e que pode ser implantado de forma independente [51]. Suas **interfaces providas** especificam as funcionalidades que o componente de software disponibiliza ao ambiente e suas **interfaces requeridas** compreendem as funcionalidades que o componente requer que o ambiente lhe forneça para que possa ser executado corretamente. O DBC pretende diminuir o custo final dos sistemas do software através da reutilização de componentes de software previamente desenvolvidos. Além disso, componentes de software podem permitir que o sistema atinja um nível maior de qualidade quando submetidos a políticas de testes.

1.2 Motivação e Problema

Apesar das abordagens de arquitetura de software e de DBC serem distintas, existe uma complementariedade entre seus conceitos. O DBC atua na produção e reutilização de componentes de software, enquanto que a arquitetura de software auxilia na integração e contextualização destes componentes. Portanto, a utilização destas duas abordagens de forma complementar e conjunta se apresenta como uma interessante solução a ser utilizada para a construção de sistemas computacionais, em especial os sistemas de software grandes e complexos.

Existem atualmente diversos processos de desenvolvimento de software para sistemas baseados em componentes. Alguns exemplos são os processos Catalysis [17], UML Components [12] e Kobra [2]. Processos de desenvolvimento de software definem um conjunto de atividades e de resultados que devem ser obtidos para estas atividades, resultando no desenvolvimento de um produto de software [49]. Nos processos de desenvolvimento de software para sistemas baseados em componentes existe uma atividade onde os componentes de software, sejam eles desenvolvidos “do zero”, reutilizados ou adquiridos de terceiros, são interligados compondo assim um componente maior, ou ainda, o próprio sistema. Esta atividade representa a definição de uma **arquitetura de software baseada em componentes** e suas preocupações são análogas às encontradas na abordagem de arquitetura de software. Em particular, nessa dissertação focamos nas arquiteturas de

software baseadas em componentes.

É interessante que existam ferramentas que auxiliem na realização das atividades de um processo de desenvolvimento de software. O uso de ferramentas facilita a utilização do processo e garante uma maior conformidade com sua especificação. Em especial, ferramentas de modelagem de arquiteturas de componentes são essenciais em processos de desenvolvimento de software baseados em componentes, pois permitem um maior ganho na composição dos componentes dos sistemas. Espera-se que estas ferramentas atuem na modelagem da arquitetura de software baseada em componentes e que possibilitem verificações e análises sobre a arquitetura produzida, como por exemplo, a verificação da conformidade da arquitetura proposta em relação a seus estilos arquiteturais. É desejável que estas ferramentas de modelagem de arquiteturas de componentes englobem conceitos presentes em arquitetura de software, como por exemplo, a especificação explícita dos conectores arquiteturais. Além de auxiliar na modelagem da arquitetura de software baseada em componentes, estas ferramentas devem possuir mecanismos que garantam a materialização desta arquitetura em forma de código. Esta materialização pode ser obtida com a utilização de tecnologias que implementem modelos de componentes, tais como as plataformas Java EE ² [50] e Microsoft .Net [33].

Todos os termos, conceitos e definições presentes nas abordagens de arquitetura de software e DBC devem estar corretamente representados e relacionados para que possam ser utilizados em ferramentas de modelagem de arquiteturas de software baseadas em componentes. Uma forma de se atingir uma maior formalização destas definições é através do uso de **metamodelos conceituais**. Metamodelos conceituais para arquitetura de software e DBC devem definir os principais conceitos envolvidos nas duas abordagens, definir regras sobre estes elementos e relacioná-los através de um mapeamento claro que permita utilizá-los em conjunto.

Porém, o que pode ser observado atualmente é uma falta de consenso entre os conceitos, termos e definições utilizados nas abordagens de arquitetura de software e de DBC. Esta falta de uniformidade nos conceitos adotados por pesquisadores dificulta a integração das respectivas técnicas e ferramentas. Ainda, os metamodelos utilizados não são muitas vezes abrangentes, possuindo foco em apenas uma das áreas ou na implementação de sistemas para determinadas tecnologias. Cox *et al.* [15] propõem em seu trabalho um modelo formal para DBC, expresso utilizando a linguagem Z. Esse modelo possui foco na especificação de componentes, incluindo componentes compostos, entretanto não possui conceitos de arquiteturas de software e nem mapeamento para elementos de implementação. As Linguagens de Descrição de Arquiteturas (ADLs ³) definem uma linguagem formal para

²do inglês: *Java Platform, Enterprise Edition*. Anteriormente conhecido como J2EE (do inglês: *Java 2 Enterprise Edition*)

³do inglês: *Architecture Description Languages*

descrição de arquiteturas de software, tendo como principais preocupações a descrição dos componentes, conectores e configurações arquiteturais. ADLs atuam como metamodelos conceituais para arquiteturas de software, porém não possuem foco nos conceitos de DBC. Balasubramanian *et al.* [4] utilizam um conjunto de ferramentas para compor um ambiente baseado em metamodelos. No centro desse ambiente está a linguagem PICML que é utilizada como metamodelo e tem como foco a especificação do sistema e a inclusão de metainformação relacionada à sua implantação ⁴. Porém, os modelos possíveis de serem construídos a partir da linguagem PICML são relacionados à tecnologias específicas de implementação de componentes e não possuem todos os conceitos de arquitetura de software, tais como conectores arquiteturais.

Esta falta de consenso das definições e a falta de metamodelos conceituais que definam e relacionem os conceitos das duas abordagens dificultam a existência de ferramentas para modelagem de arquitetura de software baseadas em componentes. As ferramentas atuais com este propósito podem ser classificadas em dois grupos: (1) ferramentas e ambientes de arquiteturas de software e (2) ferramentas e ambientes de apoio ao DBC.

Ferramentas e ambientes de modelagem de arquiteturas de software, em sua grande maioria, são projetos que possuem iniciativa no meio acadêmico, como o ambiente AcmeStudio [44], MAE [43] e ViSAC [32]. Tais ferramentas são baseadas sobre ADLs e não dão apoio a conceitos importantes presentes em DBC, tais como interfaces providas e requeridas. Nem todas as ferramentas implementam o conceito de conectores arquiteturais, que podem ser utilizados para implementação de requisitos não-funcionais. Estas ferramentas também não estão inseridas em um ambiente que englobe outras fases de um processo DBC e não permitem, em sua maioria, a materialização de uma modelagem de uma arquitetura de software em forma de código.

Ambientes e ferramentas para DBC, por outro lado, possibilitam a definição de uma arquitetura de componentes, porém não se concentram em implementar todos os benefícios disponibilizados em ferramentas de modelagem de arquitetura de software. Alguns conceitos de arquitetura de software, como especificação explícita de conectores arquiteturais, em geral não são abordados nestes ambientes. A realização de análises sobre arquiteturas também não são normalmente disponibilizadas. Além disso, os ambientes não são desenvolvidos de forma explícita sobre metamodelos conceituais que formalizem e relacionem os principais conceitos de DBC e de arquitetura de software.

⁴do inglês: *deployment*

1.3 Solução Proposta

Este trabalho apresenta uma infra-estrutura de software para apoiar a construção de arquiteturas de software baseadas em componentes. Esta infra-estrutura implementa os principais conceitos e práticas existentes nas abordagens de arquitetura de software e DBC. Um metamodelo conceitual foi criado com o objetivo de formalizar e relacionar os conceitos presentes nas duas abordagens, sendo utilizado como base para o desenvolvimento da infra-estrutura de software deste trabalho. O metamodelo está organizado em quatro partes: (1) um **metamodelo conceitual de arquitetura de software**, (2) um **metamodelo conceitual de especificação de componentes**, (3) um **metamodelo conceitual para arquiteturas de software baseadas em componentes** - que agrega características dos dois primeiros - e (4) um **metamodelo conceitual de configuração arquitetural de componentes** - responsável por descrever e mapear como as arquiteturas de componentes podem ser implementadas em diferentes plataformas e tecnologias de componentes.

Este projeto foi construído no contexto de um ambiente para apoiar o DBC e centrado na arquitetura de software, o ambiente BELLATRIX [53]. O ambiente BELLATRIX disponibiliza ferramentas que auxiliam as fases de projeto e implementação de um processo de desenvolvimento de sistemas baseados em componentes. A Figura 1.1 apresenta a visão geral do ambiente, que é composto por um conjunto de editores e ferramentas. Os editores de interfaces, tipos de dados, exceções e componentes permitem a especificação dos componentes de software de um sistema. A infra-estrutura para apoiar a construção de arquiteturas de software baseadas em componentes permite a criação de arquiteturas de componentes a partir da composição de componentes previamente especificados, permite realizar verificações sobre estas arquiteturas e permite a materialização destas arquiteturas de software baseadas em componentes em forma de uma implementação. Além disso, o ambiente BELLATRIX possui uma ferramenta para teste unitário de componentes e um repositório de componentes, responsável por armazenar e gerenciar diferentes artefatos produzidos durante o processo de desenvolvimento, entre eles, especificações e implementações de componentes. O repositório é acessado através de uma interface construída sobre o ambiente BELLATRIX e possui um gerenciador de serviços, responsável por criar uma abstração entre o repositório e as demais ferramentas inseridas no ambiente. O núcleo do ambiente foi construído sobre o metamodelo conceitual de arquitetura de software e DBC, que disponibiliza a infra-estrutura básica para o funcionamento e integração das ferramentas e editores do ambiente.

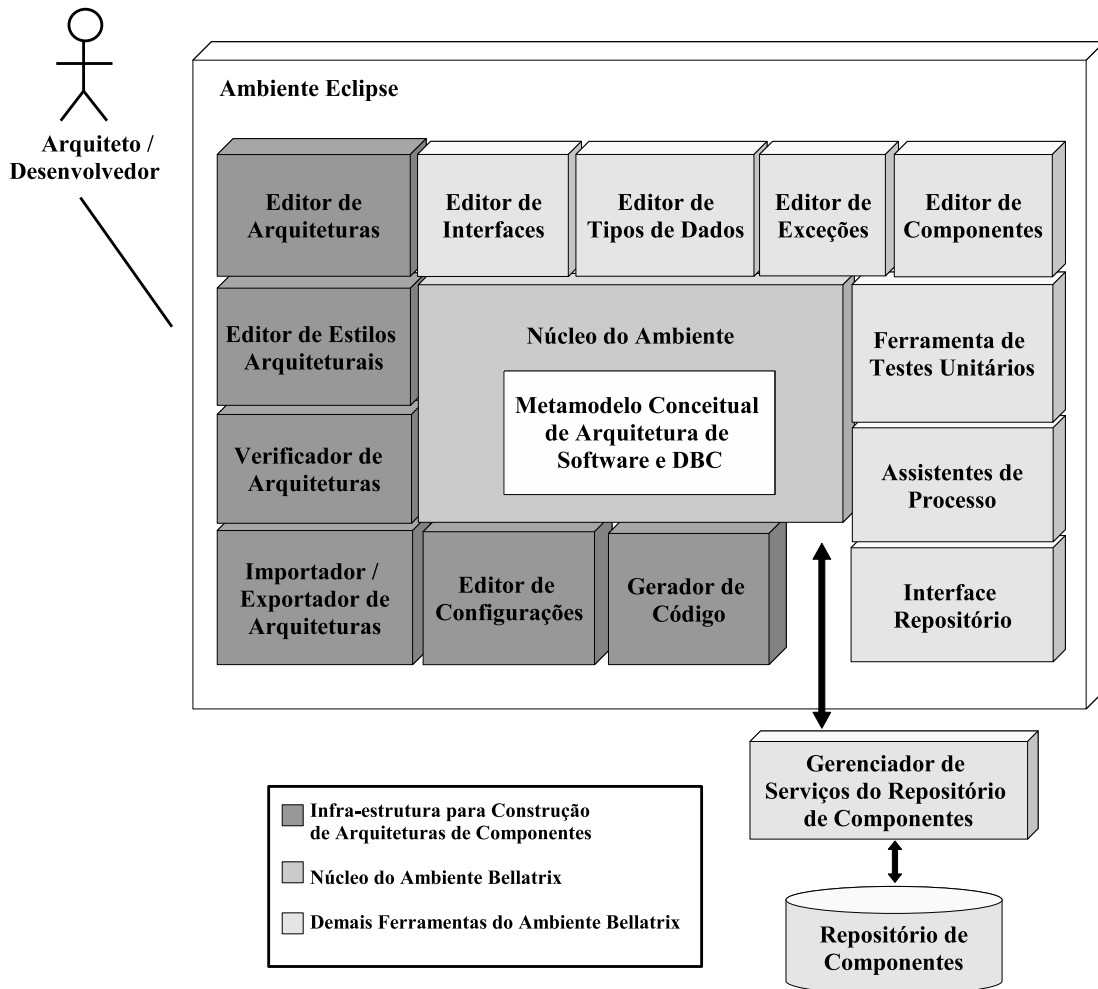


Figura 1.1: Visão geral do ambiente BELLATRIX.

As ferramentas que compõem a infra-estrutura de software para construção de arquiteturas de componentes, assim como os editores e ferramentas do ambiente BELLATRIX, foram desenvolvidas sobre o ambiente integrado de desenvolvimento (IDE ⁵) Eclipse [20]. O Eclipse atua como uma plataforma de integração de ferramentas para desenvolvimento de sistemas e permite que novas ferramentas e funcionalidades sejam anexadas ao ambiente a partir de sua estrutura de *plug-ins*. Desta forma, as ferramentas para construção de arquiteturas de componentes foram desenvolvidas como um conjunto de *plug-ins* para o ambiente Eclipse, permitindo que novas funcionalidades possam ser desenvolvidas e inseridas ao ambiente, caso necessário. Outras características interessantes do ambiente

⁵do inglês: *Integrated Development Environment*

Eclipse são: possuir código aberto e ter versões para diversas plataforma.

A infra-estrutura de software para apoiar a construção de arquiteturas de software baseadas em componentes é composta por um conjunto de ferramentas, conforme apresentado na Figura 1.2. O **Editor de Arquiteturas** permite a modelagem de arquiteturas de software baseadas em componentes a partir da composição de componentes previamente especificados. Neste editor é possível especificar arquiteturas internas para componentes que pertençam a uma arquitetura de componentes, o que torna possível especificar diversos níveis de granularidade da arquitetura de forma recursiva. As arquiteturas de software baseadas em componentes podem seguir determinados estilos arquiteturas, que podem ser criados através do **Editor de Estilos Arquiteturais**. Este editor permite especificar famílias de arquiteturas, que são compostas por tipos de elementos arquiteturais e por restrições que são impostas sobre estes tipos de elementos. As arquiteturas de componentes podem ser verificadas pelo **Verificador de Arquiteturas**. Essa verificação consiste em analisar se todas as restrições existentes nos estilos de uma arquitetura de software baseada em componentes estão sendo respeitadas. O **Importador/Exportador** de arquiteturas permite a integração e interoperabilidade com outras ferramentas de modelagem de arquiteturas de componentes, utilizando a ADL ACME. Isto torna possível importar para o ambiente arquiteturas previamente descritas em outras ferramentas, assim como gerar uma especificação de uma arquitetura de componentes a partir do ambiente que pode ser reconhecida por outras ferramentas, externas ao ambiente. O **Editor de Configurações** permite que sejam especificadas instanciações de arquiteturas para diferentes tecnologias, plataformas e modelos de componentes. O **Gerador de Código** gera um esqueleto de código da arquitetura seguindo o modelo de componentes COSMOS [48], que materializa os elementos arquiteturais em forma de código.

Desta forma, a infra-estrutura de software deste trabalho permite (1) a criação de arquiteturas de componentes, (2) a especificação de estilos arquiteturais, (3) a verificação da conformidade entre uma arquitetura de componentes e seus estilos arquiteturais, (4) a importação e exportação de descrições de arquiteturas, (5) a especificação de configurações arquiteturais baseadas em componentes e (6) a geração de código, que garante que a arquitetura especificada será refletida em uma implementação.

1.4 Trabalhos Relacionados

O ambiente AcmeStudio [44] é um ambiente para modelagem de arquiteturas de software baseado na ADL Acme [24]. No AcmeStudio é possível definir e especificar arquiteturas de software com o auxílio de um editor gráfico. Neste editor os componentes, conectores e configurações arquiteturais podem ser criados para modelar a arquitetura de um sistema. Ainda, estilos arquiteturais podem ser especificados e associados a descrições

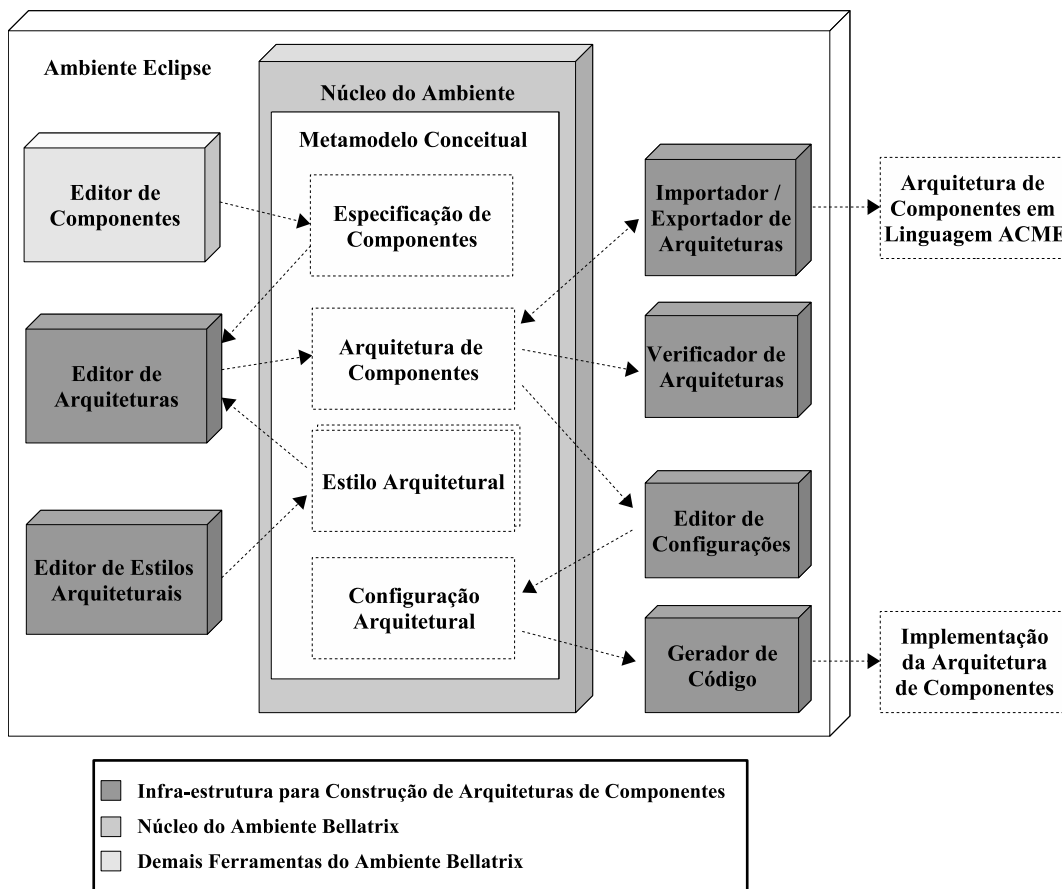


Figura 1.2: Infra-estrutura de software para construção de arquiteturas de componentes.

de arquiteturas de software. O ambiente possui um verificador, que analisa a conformidade das arquiteturas especificadas de acordo com seus respectivos estilos. Apesar do ambiente AcmeStudio cobrir grande parte das preocupações encontradas em arquitetura de software, sua utilização para sistemas DBC é um pouco limitada. A ferramenta não possui a definição de interfaces providas e requeridas, não está inserida em um ambiente DBC e possui limitações na materialização da arquitetura em código. Atualmente somente é possível a geração de código em ArchJava [1], uma linguagem de descrição de arquiteturas baseada na linguagem Java. Apesar da linguagem ArchJava possuir os conceitos de arquitetura de software, ela necessita de um compilador próprio, dificultando sua utilização.

O ambiente GME (*Generic Modeling Environment*) [28, 25] é um ambiente configurável que permite a modelagem de sistemas de acordo com domínios específicos. Estes domínios são definidos no próprio ambiente e são chamados de paradigmas. Os paradigmas

mas são criados a partir de metamodelos que contêm toda sintaxe, semântica e informação de apresentação necessárias para descrever um domínio. Os paradigmas definem os conceitos que serão utilizados em uma modelagem, os relacionamentos permitidos entre estes conceitos, a organização e forma de apresentação dos conceitos no modelo e as regras envolvidas na construção do modelo. Por ser um ambiente genérico, o GME não possui metamodelos já definidos para a modelagem de arquiteturas de software baseadas em componentes. Mesmo que um metamodelo conceitual seja definido com esta finalidade dentro do GME, a análise da arquitetura e a sua materialização através de um gerador automático de código não são disponibilizados pelo ambiente. Ainda, o ambiente GME não está dentro de um ambiente de apoio ao DBC, sendo necessária a integração com outras ferramentas para que todas as fases de um processo de desenvolvimento de sistemas baseados em componentes sejam apoiadas. O ambiente possui também a limitação de ser distribuído apenas para a plataforma Microsoft Windows.

O ambiente Cadena [13] é um ambiente para modelagem e construção de sistemas baseados em componentes que foi construído sobre a ADL CALM (*Cadena Architecture Language with Meta-modeling*). No ambiente Cadena um sistema é modelado em três camadas distintas: camada de estilos, camada de módulos e camada de cenários. Na camada de estilos, que chamaremos de camada de tecnologia, são especificados modelos de componente para determinadas plataformas, como por exemplo, o modelo CCM (*CORBA Component Model*) [38] ou o modelo EJB (*Enterprise JavaBeans*), da plataforma Java EE [50]. Na camada de módulos são criados tipos de componentes e interfaces, que são utilizados na descrição de estilos para as arquiteturas de componentes. Estes tipos de componentes e interfaces devem seguir algum modelo de componente especificado na camada de tecnologia. A arquitetura de software do sistema é criada na camada de cenários, a partir de um modelo de componentes previamente descrito para uma determinada tecnologia e respeitando o estilo ao qual pertence. Apesar do ambiente possuir estilos arquiteturais e tipos, não existe uma separação entre especificação de uma arquitetura e sua implementação. Os estilos, tipos de componentes, tipo de interfaces e a arquitetura de componentes modelados no ambiente são dependentes de algum modelo de componente e tecnologia, que foram previamente descritos na camada de tecnologia. É possível adicionar *plug-ins* ao ambiente Cadena para a geração de código para arquiteturas de componentes.

O projeto Odyssey [9] é um ambiente para desenvolvimento de sistemas, baseado em reutilização de software, que permite a especificação de modelos conceituais, arquiteturas de software e modelos de implementação para domínios de aplicação específicos, previamente selecionados. Estes domínios são especificados seguindo o processo de engenharia de domínio Odyssey-DE [8] e o ambiente conta com o processo de engenharia de aplicação Odyssey-AE [34]. O ambiente é dividido no Odyssey Light, um núcleo que contém as

funcionalidades mandatórias para a execução mínima do ambiente, e num conjunto de ferramentas disponibilizadas na forma de *plug-ins*, que podem ser anexadas ao núcleo do ambiente. Essas ferramentas incluem diversas funcionalidades, tais como ferramentas para a documentação de componentes, especificação e instanciação de arquiteturas específicas de domínios, apoio a engenharia reversa, ferramenta de críticas em modelos UML e ferramentas de modelagem e execução de processos. Existem também outras ferramentas complementares ao ambiente, que envolvem áreas como gerência de configuração de software e desenvolvimento colaborativo. O Odyssey possui apoio a conceitos de DBC como interfaces providas e requeridas e decomposição hierárquica, ainda que nem toda modelagem seja possível através de editores gráficos. O ambiente é disponibilizado para livre uso, apesar de não ter seu código aberto.

A Linguagem PICML [4, 5] define um metamodelo para modelagem de sistemas baseados em componentes. Este metamodelo deve ser utilizado no ambiente GME e possui vários elementos que permitem a modelagem de arquiteturas de componentes segundo determinadas tecnologias. É possível utilizar o ambiente GME para modelar estes sistemas baseados em componentes, realizar análises sobre as interações definidas para os componentes, realizar decomposição hierárquica dos componentes e gerar informações sobre a implantação dos componentes. Outras ferramentas podem ser utilizadas para gerar código em determinadas plataformas previamente definidas. A utilização da linguagem PICML no ambiente GME possui a vantagem de permitir uma descrição detalhada dos componentes, levando em consideração suas propriedades de implementação. Porém, a utilização desta abordagem tem como desvantagens não possuir uma separação clara entre a especificação e a implementação dos componente e não permitir atribuição de estilos arquiteturais. As análises se limitam a apontar inconsistências encontradas na arquitetura de componentes de acordo com a tecnologia utilizada.

Uma comparação entre os trabalhos relacionados e a infra-estrutura de software desta dissertação será apresentada na Seção 4.3, após a apresentação dos requisitos considerados para a construção da infra-estrutura de software.

1.5 Organização deste Documento

Este documento é composto por seis capítulos, organizados da seguinte forma:

- **Capítulo 2 - Fundamentos de Arquitetura de Software e DBC:** O segundo capítulo define os principais termos e conceitos necessários para um melhor entendimento deste trabalho. São apresentados assuntos sobre: arquitetura de software, desenvolvimento baseado em componentes, o modelo de implementação de componentes COSMOS e o ambiente Eclipse.

- **Capítulo 3 - Um Metamodelo Integrado para Arquitetura de Software e DBC:** Este capítulo apresenta um metamodelo conceitual que define e descreve os elementos necessários para a modelagem de arquiteturas de software baseadas em componentes. Os metamodelos são divididos em metamodelos conceituais de arquitetura de software, de DBC e de configuração de componentes. Um exemplo é apresentado para ilustrar o uso do metamodelo.
- **Capítulo 4 - Uma Infra-estrutura de Software para Apoiar a Construção de Arquiteturas de Componente:** Neste capítulo é apresentada a infra-estrutura de software para apoiar a construção de arquiteturas de software baseadas em componentes. A parte inicial do Capítulo apresenta os requisitos identificados, os casos de uso das ferramentas e uma comparação com trabalhos relacionados de acordo com os requisitos identificados. As ferramentas que compõem a infra-estrutura são divididas e apresentadas em duas partes: ferramentas para modelagem de arquiteturas de componentes e ferramentas para construção de arquiteturas de componentes. Na parte final do capítulo são apresentadas as características mais relevantes sobre a implementação da ferramenta: sua arquitetura interna, os modelos de implementação adotados e as principais tecnologias utilizadas.
- **Capítulo 5 - Estudos de Caso:** Este capítulo apresenta três estudos de caso realizados com o objetivo de avaliar as principais funcionalidades da infra-estrutura de software deste trabalho. Cada estudo de caso é apresentado e analisado e ao final do capítulo é apresentada uma consolidação dos resultados obtidos.
- **Capítulo 6 - Conclusões e Trabalhos Futuros:** Este capítulo apresenta a conclusão desse trabalho, listando suas contribuições, suas limitações e sugerindo trabalhos futuros.

Capítulo 2

Fundamentos de Arquitetura de Software e DBC

Neste capítulo são apresentados os principais conceitos e termos para uma melhor compreensão do restante deste trabalho. Na Seção 2.1 são apresentados os fundamentos para arquitetura de software, através dos conceitos e definições relacionados à estilos arquiteturais e a ADLs. Na Seção 2.2 são apresentados os conceitos para DBC, em especial processos DBC e a diferença entre especificação, implementação e instanciação de componentes de software. A Seção 2.3 apresenta o modelo de componentes COSMOS e a Seção 2.4 apresenta o ambiente integrado de desenvolvimento Eclipse, sobre o qual a infra-estrutura de software para arquiteturas de software baseadas em componentes foi desenvolvida.

2.1 Arquitetura de Software

2.1.1 Estilos Arquiteturais

Estilos arquiteturais definem conjuntos de regras para a identificação dos tipos de componentes e conectores arquiteturais que podem ser utilizados para compor um sistema, juntamente com as restrições na maneira como estas arquiteturas de software podem ser criadas [45]. Um estilo arquitetural define uma família de arquiteturas de software, num nível de abstração mais elevado que o de uma configuração específica de componentes e conectores arquiteturais. A análise de um estilo arquitetural, nesse nível de abstração, permite que se deduzam propriedades que irão se verificar em qualquer configuração que possa ser criada em conformidade com esse estilo. São essas propriedades gerais dos vários estilos arquiteturais que, no desenvolvimento de um sistema em particular, motivam a escolha de um ou mais estilos arquiteturais para guiar o projeto de sua arquitetura

de software.

Os estilos arquiteturais mais conhecidos foram descritos inicialmente de maneira informal e não sistematizada, através de definições discursivas e diagramas ilustrativos. A comunidade de padrões se dedica a documentar soluções já comprovadas, incluindo estilos arquiteturais, no contexto de tipos de problemas específicos em que cada solução é aplicável [45]. Surgiram, assim, os chamados **padrões arquiteturais**, que são estilos arquiteturais organizados e documentados de forma sistemática [11].

Um exemplo muito conhecido e utilizado de estilo arquitetural é o estilo arquitetural em camadas. Neste estilo são definidas camadas para a aplicação. O sistema é particionado em subsistemas e cada camada da aplicação passa a conter um ou mais destes subsistemas. Normalmente, em sistemas que utilizam o estilo em camadas, subsistemas localizados em uma determinada camada n usam os serviços disponibilizados por subsistemas da camada $n+1$ (abaixo) e oferecem serviços para subsistemas da camada $n-1$ (acima). Desta maneira, subsistemas inseridos nas camadas $n-1$ não utilizam diretamente serviços de subsistemas inseridos em uma camada $n+1$. Assim, como pode ser verificado na Figura 2.1, uma camada “esconde” de suas camadas superiores os detalhes das implementações das camadas inferiores, tornando possível que a implementação de uma camada seja trocada sem que o sistema inteiro seja afetado. Outros exemplos de estilos arquiteturais comumente conhecidos e utilizados são: Repositório, Quadro Negro (*Blackboard*) e *Pipes and Filters*.

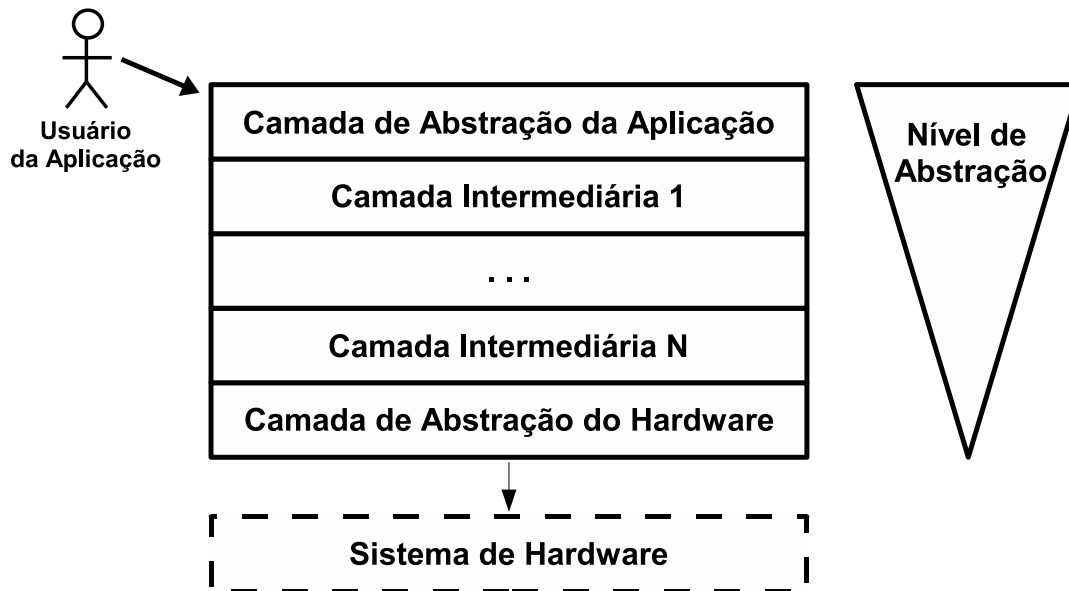


Figura 2.1: Exemplo do estilo arquitetural em camadas.

2.1.2 Linguagens de Descrição de Arquitetura (ADL) e ACME

Para o desenvolvimento centrado na arquitetura de software é interessante que as descrições de arquitetura possam ser feitas de maneira formal. Através de linguagens formais para descrição de arquiteturas é possível que estilos e configurações arquiteturais sejam descritas e analisadas. Ainda, o uso de linguagens formais para a descrição de arquiteturas é importante no contexto de ferramentas, pois disponibilizam a maneira como uma ferramenta irá descrever e manipular as arquiteturas de software de um sistema. ADLs são linguagens que permitem esta formalização para especificações de arquiteturas de sistemas de software. As principais abstrações descritas em uma ADL são componentes arquiteturais, conectores, propriedades arquiteturais e configurações arquiteturais. Um grande número de ADLs possuem também os elementos necessários para a descrição de estilos arquiteturais, através de tipos e famílias de elementos presentes em uma arquitetura. Alguns exemplos conhecidos de ADLs são a linguagem Acme [24], xADL [16] e UniCon [46].

A Linguagem ACME

A linguagem Acme foi desenvolvida a partir do estudo de ferramentas de descrição de arquiteturas de software e de ADLs, se apresentando como uma linguagem de intercâmbio para descrição de arquiteturas de software. A linguagem permite que uma arquitetura seja descrita em quatro aspectos distintos: estrutura, propriedades, restrições e tipos (e estilos). A **estrutura** descreve como o sistema está organizado, sendo composta por sete elementos: componentes, conectores, sistemas (possui um conjunto de componentes e conectores interligados), portas e papéis¹ (definem os pontos de interação de componentes e de conectores, respectivamente), representação (definem arquiteturas internas de componentes ou conectores) e *rep-maps* (realizam um mapeamento entre pontos de interação de um componente ou conector e um ponto de interação pertencente a arquitetura interna deste componente ou conector). A Figura 2.2 apresenta os elementos de estrutura da linguagem Acme.

Através do uso das **propriedades** é possível descrever informações adicionais sobre os sete elementos de estrutura da linguagem Acme. As propriedades podem descrever informações variadas sobre os elementos, tais como protocolo de comunicação utilizado, consumo de recursos e número máximo de portas permitidas em um componente. Isto possibilita à linguagem um maior poder para descrever particularidades existentes em outras ADLs ou ferramentas de descrição de arquiteturas de software. As **restrições** definem regras a serem respeitadas para a descrição de uma arquitetura. A linguagem Acme possui a linguagem Armani, uma linguagem baseada em lógica de primeira ordem

¹do inglês: *role*

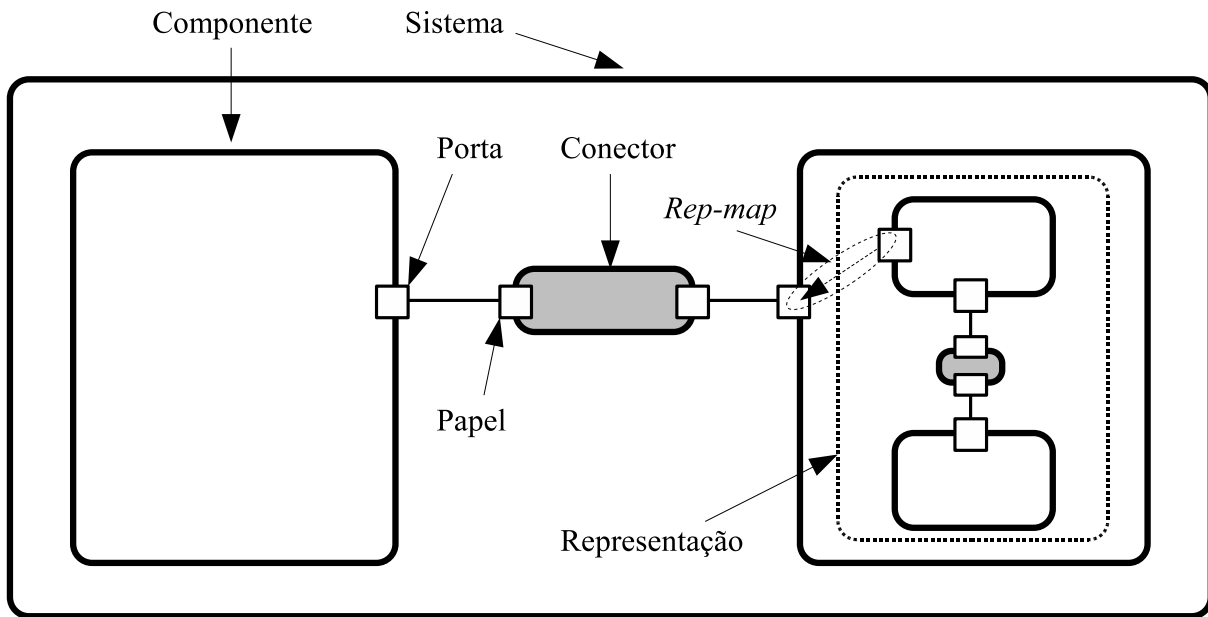


Figura 2.2: Elementos que descrevem uma estrutura arquitetural em Acme.

que pode ser utilizada para descrever tais restrições. Nesta linguagem qualquer elemento de uma descrição arquitetural, seja ele um elemento de estrutura ou uma propriedade, pode ser utilizado para compor uma restrição. Os **tipos e estilos** descrevem famílias de sistemas. Estas famílias são compostas por tipos de componentes, conectores, portas e papéis. Estes tipos de elementos são descritos através de propriedades e restrições e são agrupados para compor uma família de sistemas. Estas famílias de sistemas representam estilos arquiteturais.

A biblioteca Acmelib foi desenvolvida para facilitar a especificação de sistemas em linguagem Acme. A biblioteca disponibiliza todos os elementos presentes na linguagem (elementos estruturais, propriedades, restrições e tipos) e possui um analisador que verifica se todas as restrições estão sendo respeitadas na definição da arquitetura. Esta biblioteca é disponibilizada nas linguagens Java e C++.

2.2 Desenvolvimento Baseado em Componentes

2.2.1 Especificação, Implementação e Instanciação de Componentes

Um componente de software é descrito por uma especificação e pode possuir uma ou mais implementações e instanciações [12]. A **especificação** define o comportamento externamente observável de um componente de software, sendo composto principalmente por suas interfaces providas e requeridas. A especificação deve descrever de forma clara como o componente se integra em diferentes sistemas, porém sem estar ligado a uma implementação específica. Uma especificação de componente é concretizada através de uma **implementação de componente**. É possível que uma mesma especificação gere diferentes implementações de componentes em tecnologias e ambientes diferenciados. Quando uma implementação de componente é integrada em um ambiente, sua execução gera uma **instanciação de componente**. Portanto, conforme mostrado na Figura 2.3, uma especificação de componente pode possuir diferentes implementações de componente, que no momento de execução, por sua vez, podem gerar uma ou mais instâncias de componente.



Figura 2.3: Especificação, implementação e instanciação de componentes.

2.2.2 Processos de Desenvolvimento Baseado em Componentes

O objetivo de um processo de desenvolvimento de software é sistematizar as atividades de construção de um sistema de software. Sua complexidade pode ser dividida em três grupos de atividades gerais [40]: (i) definição do problema; (ii) desenvolvimento do sistema; e (iii) manutenção. Os processos devem possuir um conjunto de métodos estruturados que detalhem e auxiliem o desenvolvimento de sistemas nas suas fases [40]. Esses métodos são procedimentos sistemáticos que usam notações bem definidas para alcançar seus objetivos. De uma maneira geral, os métodos devem incluir informações sobre [49]: (i) os modelos produzidos; (ii) as restrições aplicadas a esses modelos; (iii) diretrizes de projeto; e (iv) a sequência de atividades a ser seguida.

Os processos tradicionais de desenvolvimento de software não se adequam totalmente ao desenvolvimento de sistemas baseados em componentes. Processos de desenvolvimento baseado em componentes devem conter fases e métodos que também ofereçam técnicas que permitam o empacotamento de componentes com o objetivo específico de serem reutilizados. Os métodos também devem auxiliar na definição de como os componentes devem ser conectados uns aos outros para atender aos requisitos especificados, ou seja, devem auxiliar na construção das arquiteturas de software baseadas em componentes.

Um exemplo de um processo de desenvolvimento baseado em componentes é o processo **UML Components** [12], que é descrito a seguir.

O Processo de DBC UML Components

O processo UML Components é um processo iterativo e pode ser utilizado como uma extensão do processo RUP² [27]. O UML Components estende a notação UML [7] adequando-a à representação de elementos necessários ao DBC. A Figura 2.4 mostra uma visão geral das fases definidas pelo processo. Essas fases são: (1) requisitos, que inclui atividades como a modelagem do negócio e a extração de requisitos; (2) especificação, onde é gerado um conjunto de especificações de componentes e uma arquitetura de componentes; (3) provisionamento, que garante que os componentes necessários estão disponíveis, seja através de seu desenvolvimento, aquisição ou reuso; (4) montagem, que compõe os diversos componentes entre si e com artefatos de software pré-existentes, incluindo a interface com o usuário; (5) testes, onde os componentes e a aplicação devem ser testados e (6) implantação, que é a instalação da aplicação em seu ambiente de produção.

Dada a idéia do UML Components ser uma extensão do processo RUP, as fases de análise, projeto e implementação do RUP podem ser substituídas pelas fases de especificação, provisionamento e montagem do processo UML Components. Ainda, o processo não cobre atividades do processo gerencial e dá pouco ou nenhum suporte sobre as atividades de testes e implantação. Ainda, não há apoio para a criação de arquiteturas em estilos arquiteturais diferente de camadas. Entretanto, esse processo tem sido adotado, entre outros processos de DBC, devido à sua simplicidade e ao seu pragmatismo.

²do inglês *Rational Unified Process*.

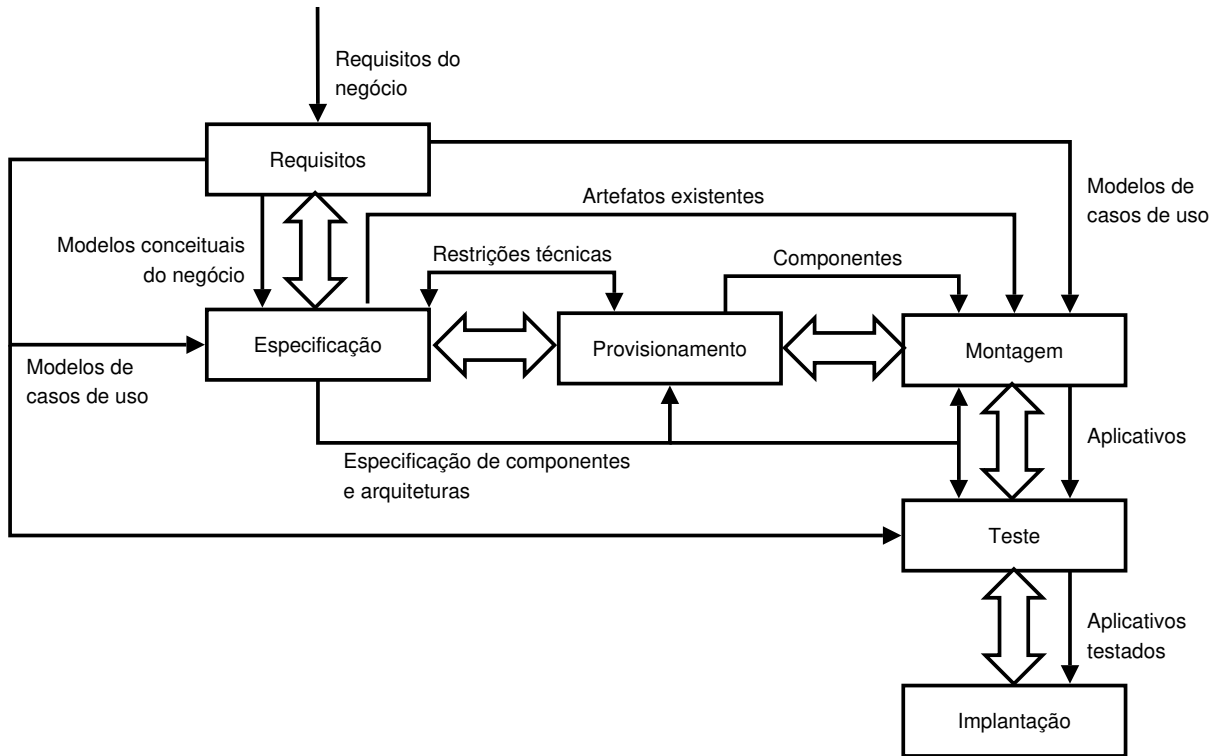


Figura 2.4: Visão geral das fases do processo UML Components

2.3 O Modelo de Implementação de Componentes COSMOS

O COSMOS [48] é um modelo de implementação de componentes de software que permite o mapeamento de uma arquitetura de componentes para linguagens de programação. O modelo define uma estrutura de implementação baseada em programação orientada-a-objetos para a materialização de uma arquitetura de componentes. Este modelo de implementação garante a conformidade da descrição de uma arquitetura de componentes em relação à sua implementação e facilita a evolução desta implementação. O modelo é definido através de uma estrutura de pacotes, classes e interfaces, conceitos presentes nas linguagens orientadas-a-objetos.

O modelo COSMOS possui um conjunto de diretrizes que auxilia na evolução e reutilização dos componentes de software implementados. Algumas diretrizes utilizadas pelo modelo são: (i) materialização de elementos arquiteturais, através de componentes, conectores e configurações arquiteturais; (ii) separação de requisitos não-funcionais, que devem ser inseridos nas implementações dos conectores; (iii) separação explícita entre especificação e implementação dos componentes; (iv) declaração explícita das dependências

existentes entre os componentes, que é viabilizada pelo uso das interfaces requeridas; e (v) baixo acoplamento entre as classes de implementação.

O modelo COSMOS é dividido em quatro sub-modelos: o (i) **modelo de especificação** representa o conjunto de interfaces públicas de um componente, sejam elas providas ou requeridas; o (ii) **modelo de implementação** define a parte privada do componente, responsável por implementar os serviços providos; (iii) o **modelo de conectores** descreve a materialização da conexão entre interfaces requeridas e providas de dois ou mais componentes; e (iv) o **modelo de configuração arquitetural** realiza a composição de vários componentes e conectores em uma arquitetura através da instanciação e inicialização destes componentes e conectores.

A Figura 2.5 apresenta uma visão geral do modelo COSMOS. A parte superior da figura apresenta uma configuração arquitetural, formada pelos componentes A e B que são conectados através do conector AB. O componente A provê a interface I1 e requer a interface I2 e o componente B provê a interface I3. A comunicação entre os componentes é realizada pelo conector AB, que provê a interface I2 e requer a interface I3. O restante da figura descreve os pacotes e algumas classes e interfaces para que esta configuração arquitetural possa ser implementada seguindo o modelo COSMOS. A estrutura básica de um componente ou conector é um pacote. Os componentes possuem dois pacotes internos: o pacote `spec` e o pacote `impl`. O pacote `spec` contém as interfaces públicas do componente, que são agrupadas em interfaces providas (pacote `spec.prov`) e interfaces requeridas (pacote `spec.req`). O pacote `impl` contém as classes que implementam as interfaces providas do componente, que, caso necessário, podem usar as interfaces requeridas. O conector possui classes que realizam a comunicação entre componentes. Estas classes implementam interfaces requeridas de um componente e utilizam interfaces providas de outros componentes.

Outras interfaces e classes são utilizadas para permitir as demais funcionalidades necessárias para a execução de uma implementação, tais como instanciação dos componentes e conectores, e delegação de responsabilidades dentro dos componentes e conectores.

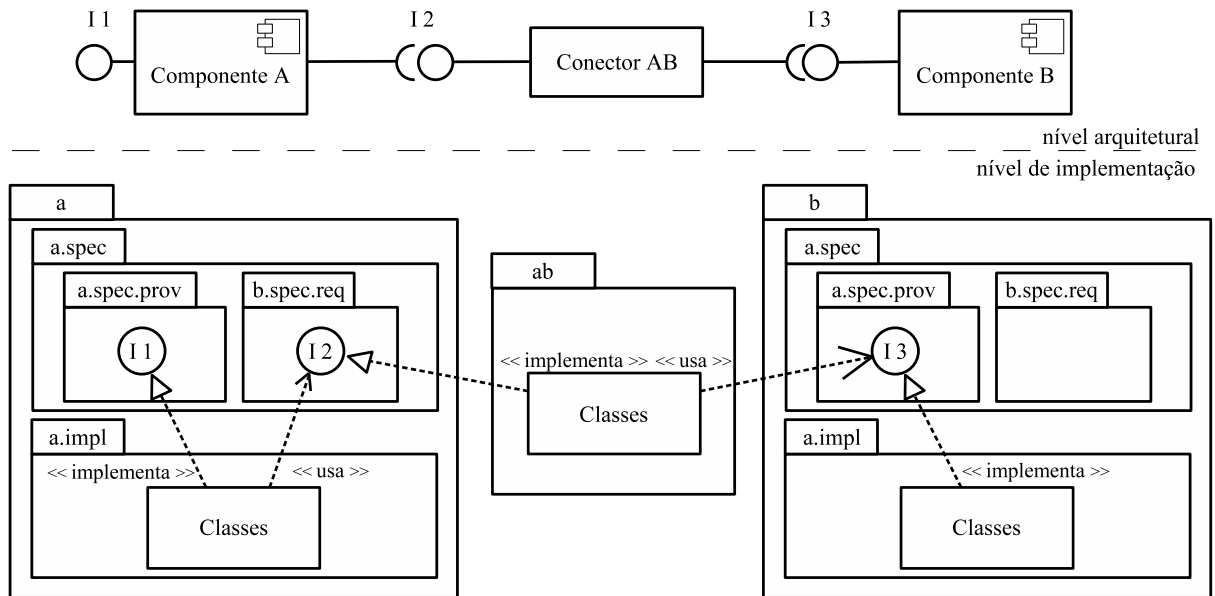


Figura 2.5: Visão geral do modelo COSMOS.

2.4 O Ambiente Eclipse

O Eclipse [20] é um ambiente de código aberto (do inglês: *open source*) para o desenvolvimento de sistemas de software. A distribuição básica do ambiente fornece um conjunto de funcionalidades que facilitam a construção de sistemas, tais como editores de texto, depuradores e interfaces para repositórios de código. O foco principal do ambiente Eclipse é a linguagem Java, porém outras linguagens e modelos podem ser utilizados. Além das funcionalidades básicas, a arquitetura do ambiente permite que ele possa ser estendido através de *plug-ins*, que acrescentam outras funcionalidades ao ambiente. Atualmente existe um número elevado de *plug-ins* desenvolvidos para o ambiente Eclipse, que permitem desde o desenvolvimento de sistemas em diferentes linguagens de programação, como Java, C++ [18] e COBOL [19], até a modelagem de diagramas em linguagem UML [39]. Estes *plug-ins* podem ser desenvolvidos no *Plug-in Development Environment* (PDE), um conjunto de ferramentas disponibilizadas pelo ambiente Eclipse que permitem o desenvolvimento e teste destes *plug-ins* dentro do próprio ambiente.

Originalmente idealizado pela empresa IBM, o Eclipse é distribuído de forma gratuita de acordo com uma licença pública (*Eclipse Public License*). Atualmente o projeto é mantido por um consórcio de empresas que contém mais de 80 membros, entre elas a Borland, IBM, Rational, Red Hat, Oracle, SAP e a OMG - *Object Management Group*, grupo responsável pela especificação do UML, CORBA, entre outros. Estas características deram

ao ambiente Eclipse uma grande força dentro dos ambientes corporativos e universitários, se tornando como uma solução amplamente utilizada para o desenvolvimento de sistemas de software.

2.5 Resumo

Este capítulo apresentou os principais conceitos e termos necessários para o entendimento deste trabalho. Na seção 2.1 foi destacada a importância dos estilos arquiteturais e das ADLs para modelar arquiteturas de software e realizar análises sobre estas arquiteturas. A linguagem Acme foi apresentada como uma ADL que possibilita a descrição dos elementos estruturais de uma arquitetura, de suas propriedades e da sua semântica, se apresentando como linguagem de intercâmbio entre outras ADLs. A Seção 2.2 realizou a diferenciação entre uma especificação, uma implementação e uma instanciação de um componente e apresentou os processos de desenvolvimento baseados em componentes. O Modelo COSMOS, apresentado na Seção 2.3, é um modelo de implementação de componentes de software que descreve um conjunto de regras sobre a estrutura e utilização de linguagens orientadas-a-objetos para a materialização de arquiteturas de software. Na Seção 2.4 foi apresentado o Eclipse, um ambiente integrado de desenvolvimento. O Eclipse oferece apoio ao desenvolvimento em diversas linguagens de programação, em especial Java, e permite a integração de novas ferramentas através de sua estrutura de *plug-ins*. No próximo capítulo é apresentado o metamodelo conceitual integrado para arquitetura de software e DBC, que considerou os fundamentos apresentados neste capítulo para a definição e o mapeamento dos conceitos presentes nestas duas abordagens.

Capítulo 3

Um Metamodelo Integrado para Arquitetura de Software e DBC

Este capítulo apresenta um metamodelo conceitual integrado para arquitetura de software e DBC, definindo e mapeando os principais conceitos envolvidos nas duas abordagens. Conforme apresentado na Seção 1.2, é importante que estes conceitos possam ser corretamente definidos e mapeados para que as abordagens de arquitetura de software e de DBC possam ser utilizadas em conjunto. A partir deste metamodelo conceitual é possível modelar arquiteturas de software baseadas em componentes, que possuirão um maior formalismo e maior conformidade com os conceitos envolvidos nas duas abordagens.

O metamodelo foi dividido em metamodelos menores para facilitar sua compreensão. Na Seção 3.1 é apresentado o metamodelo conceitual de arquitetura de software e na Seção 3.2 é apresentado o metamodelo conceitual para DBC. O metamodelo conceitual para DBC é dividido em um metamodelo conceitual de especificação de componentes e um metamodelo conceitual de arquiteturas de software baseadas em componentes, que realiza um mapeamento para o metamodelo conceitual de arquitetura de software. O metamodelo conceitual que define como uma arquitetura de componentes foi materializada através de uma implementação é descrito no metamodelo conceitual para configuração arquitetural de componentes, apresentado na Seção 3.3.

Os metamodelos estão descritos em diagramas de classe na linguagem UML [36]. Nestes diagramas são apresentados os elementos, atributos e relacionamentos existentes para arquitetura de software e DBC. Algumas restrições em linguagem OCL (*Object Constraint Language*) [37] foram descritas para complementar a descrição dos metamodelos. A linguagem OCL é uma linguagem formal para descrição de restrições para modelos UML. As restrições OCL identificadas foram implementadas nas ferramentas que compõem a infra-estrutura de software para apoiar a construção de arquiteturas de software baseadas em componentes, apresentada no Capítulo 4.

3.1 Metamodelo para Arquitetura de Software

Dois metamodelos conceituais foram criados no contexto de arquitetura de software: um metamodelo conceitual para modelagem de arquiteturas de software e um metamodelo conceitual para modelagem de estilos arquiteturais. Estes metamodelos são apresentados nas seções seguintes. Os principais elementos utilizados para a criação destes metamodelos foram apresentados na Seção 1.1 e na Seção 2.1.1.

3.1.1 Metamodelo Conceitual de Arquitetura de Software

O metamodelo conceitual de arquitetura de software é apresentado na Figura 3.1. Os elementos presentes no metamodelo são definidos como:

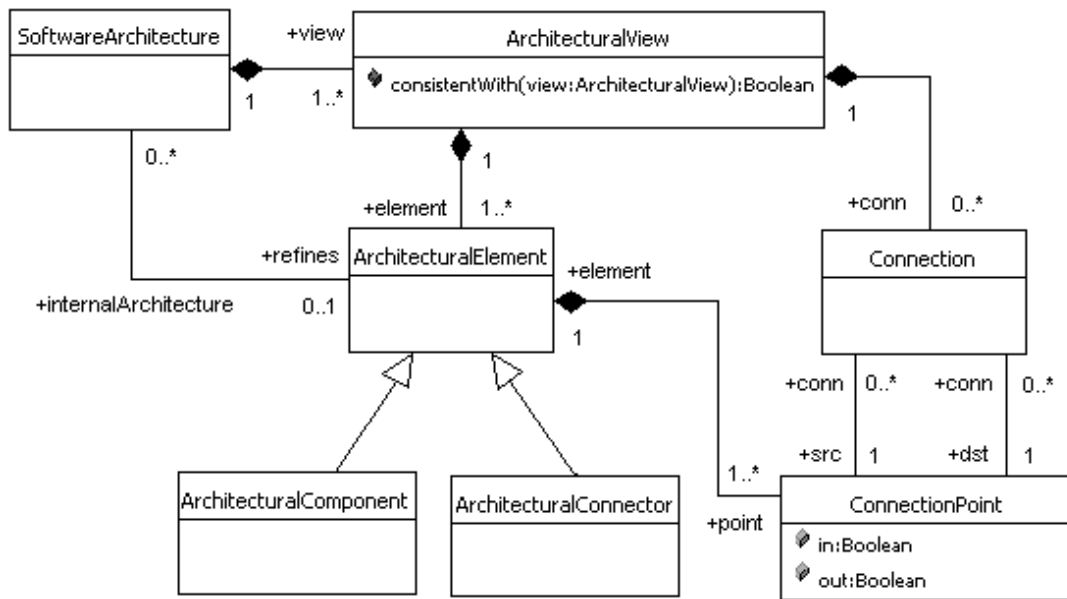


Figura 3.1: Metamodelo conceitual de arquitetura de software.

Arquitetura de Software (*Software Architecture*) define a estrutura de alto nível de um sistema de software, abstraindo detalhes de sua implementação [24]. A descrição de uma arquitetura de software é composta por um conjunto consistente de diferentes visões arquiteturais.

Visão Arquitetural (*Architectural View*) descreve um determinado aspecto de uma arquitetura de software através da sua decomposição num conjunto de elementos arquiteturais interconectados [6]. A organização do código fonte e a distribuição do processamento são exemplos de aspectos usualmente descritos através de uma

visão arquitetural própria. Uma visão arquitetural pode respeitar determinadas características impostas por estilos arquiteturais.

Elemento Arquitetural (*Architectural Element*) é uma abstração que representa uma parte de um sistema de software responsável por determinado comportamento ou propriedade desse sistema. Por exemplo: um elemento arquitetural “Camada-DePersistência” que representa a parte de um sistema de software responsável pela persistência dos objetos do sistema. Um elemento arquitetural pode estar associado a tipos de elementos arquiteturais e pode possuir uma arquitetura interna.

Ponto de conexão (*Connection Point*) é uma propriedade de um elemento arquitetural que possibilita alguma forma de interação entre este elemento e o ambiente onde está inserido [24]. Um ponto de conexão pode estar associado a tipos de pontos de conexão.

Conexão (*Connection*) é uma relação entre dois pontos de conexão, representando uma possível interação entre dois elementos arquiteturais [29].

Componente Arquitetural (*Architectural Component*) é um tipo de elemento arquitetural responsável por uma funcionalidade de um sistema de software. Componentes arquiteturais podem ser definidos com diferentes granularidades, desde um componente que provê apenas uma funcionalidade básica até um subsistema complexo responsável por um amplo conjunto de funcionalidades.

Conector Arquitetural (*Architectural Connector*) é um tipo de elemento arquitetural cuja principal responsabilidade é mediar a interação entre outros elementos arquiteturais [24]. Um conector pode ser responsável também por uma parcela dos aspectos de qualidade (ou não-funcionais) do sistema, tais como distribuição e segurança.

3.1.2 Metamodelo Conceitual de Estilos Arquiteturais

A Figura 3.2 apresenta o diagrama de classes do metamodelo conceitual de estilos arquiteturais. Os elementos presentes no metamodelo são definidos da seguinte forma:

Estilo Arquitetural (*Architectural Style*) define uma classe de sistemas que possuem propriedades arquiteturais e semânticas em comum [47]. Um estilo arquitetural é descrito a partir de um conjunto de tipos arquiteturais e restrições que são impostas a estes tipos.

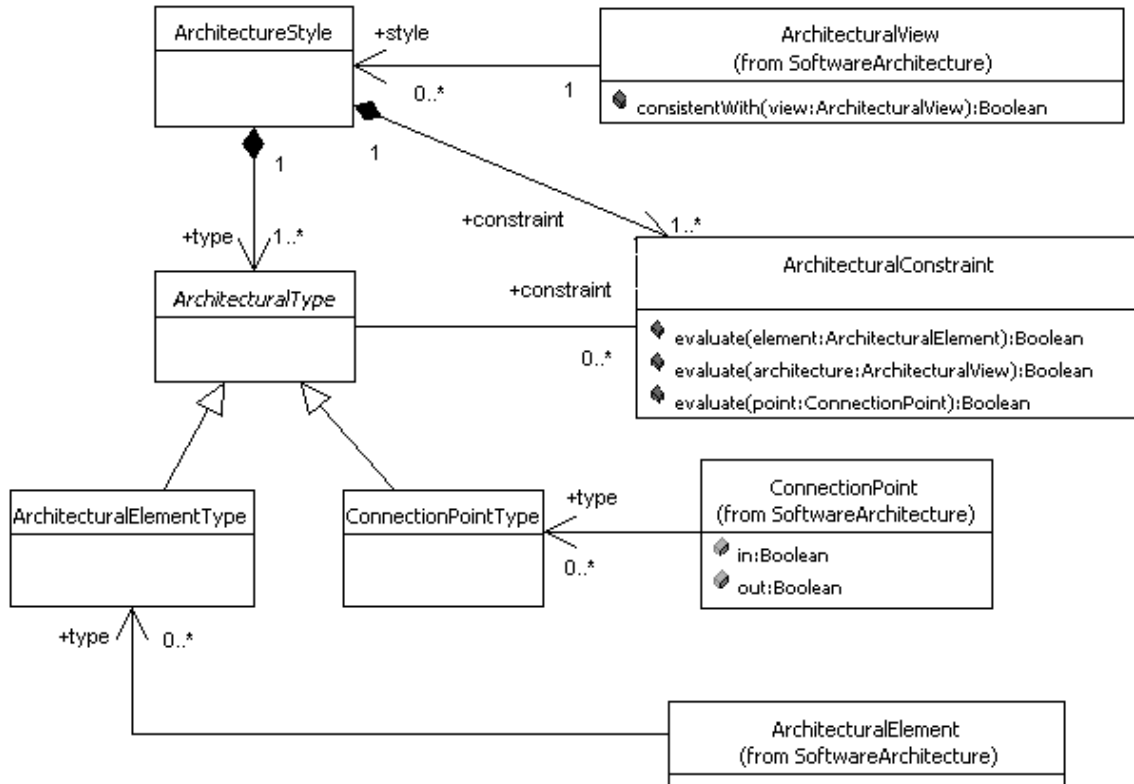


Figura 3.2: Metamodelo conceitual de estilos arquiteturais.

Tipo de Elemento Arquitetural (*Architectural Element Type*) é uma abstração que representa uma classe de elementos arquiteturais (componentes ou conectores) com determinadas características comuns [43].

Tipo de Ponto de Conexão (*Connection Point Type*) é uma abstração que representa uma classe de pontos de conexão com características em comum, como por exemplo, uma forma particular de comunicação.

Restrição Arquitetural (*Architectural Constraint*) impõe limitações sobre a estrutura e a semântica de tipos da arquitetura, definindo as restrições impostas para a classe de sistemas a qual o tipo representa.

3.1.3 Restrições Aplicadas ao Metamodelo

A Figura 3.3 descreve restrições na linguagem OCL que complementam o metamodelo para arquitetura de software. O Apêndice A contém a descrição dos principais elementos da linguagem OCL que foram utilizados nas definições das restrições. O método `consistentWith()`, do elemento *Architectural View*, e o método `evaluate()`, do elemento *ArchitecturalConstraint*, representam a conformidade com uma visão arquitetural e a avaliação de uma restrição arquitetural, respectivamente. As restrições identificadas para o metamodelo de arquitetura de software foram:

consistentViews : garante que todas as visões de uma arquitetura de software são consistentes entre si;

styleConformance : verifica se uma visão arquitetural está em conformidade com seu estilo. Caso uma visão arquitetural possua um estilo, todos os elementos e pontos de conexão desta visão devem possuir tipos pertencentes a este estilo e todas as restrições impostas pelos tipos devem ser satisfeitas;

notRecursive : garante que um elemento arquitetural não pode fazer parte da sua arquitetura interna.

distinctElements : indica que as conexões devem ser realizadas para elementos distintos;

sameView : indica que as conexões devem conectar elementos que pertençam a uma mesma visão arquitetural;

validDirection : indica que os pontos de conexão utilizados em uma conexão devem possuir direções válidas, ou seja, um ponto deve ser do tipo **in** e o outro ponto do tipo **out**;

validTypes : indica que pontos de conexão utilizados em uma conexão devem possuir tipos iguais.


```

context SoftwareArchitecture
inv consistentViews:
    self.view->forAll(v1| self.view->forAll(v2|
        v1<>v2 implies v1.consistentWith(v2)))

context ArchitecturalView
inv styleConformance:
    not self.style.isUndefined() implies
    self.element->forAll(e |self.style.type->includes(e.type)) and
    self.style.constraint->forAll( c |
        if (c.type.isUndefined())
            then c.evaluate(self)
            else self.element->forAll(e | e.type=c.type implies c.evaluate(e))
            and self.element.point->forAll(p|p.type=c.type implies c.evaluate(p))
        endif)

context ArchitecturalElement
def: let ascendants:Set(ArchitecturalElement)=
    parent.parent.refines.ascendants()->union(Set{parent.parent.refines})
inv notRecursive:
    not ascendants->includes(self)

context Connection
inv distinctElements:
    self.src.parent <> self.dst.parent
inv sameView
    self.parent = self.src.parent.parent and
    self.parent = self.dst.parent.parent
inv validDirection:
    self.src.out and self.dst.in
inv validTypes:
    self.dst.type->forAll(t|self.src.type->includes(t))

```

Figura 3.3: Restrições OCL para arquitetura de software.

3.2 Metamodelo para DBC

3.2.1 Visão Geral

Os principais conceitos de DBC também foram modelados em um diagrama de classes UML. A Figura 3.4 apresenta uma visão geral do metamodelo conceitual para DBC. As Seções 3.2.2 e 3.2.3 detalham, respectivamente, os metamodelos conceituais de especificação de componentes e de arquiteturas de software baseadas em componentes. O mapeamento entre os conceitos de arquitetura de software e de DBC foi feito através de associações de herança. Os principais elementos utilizados para a criação destes metamodelos foram apresentados na Seção 1.1 e na Seção 2.2.

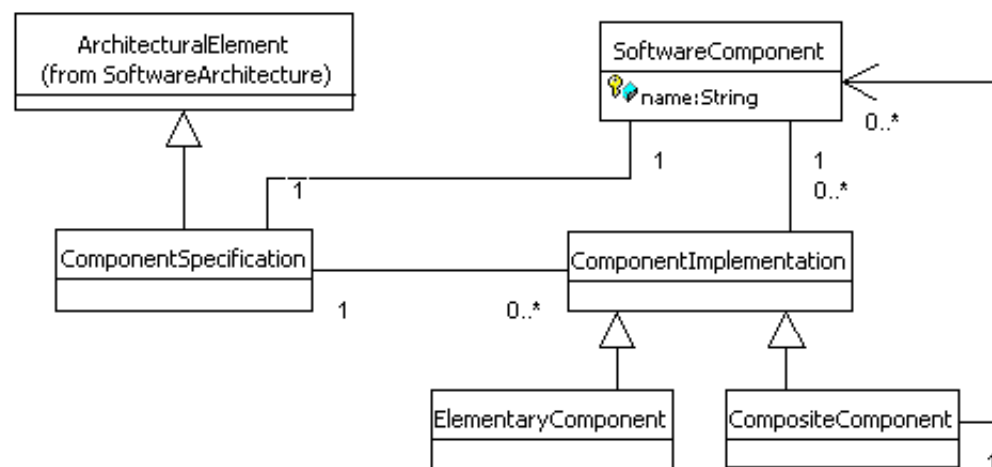


Figura 3.4: Metamodelo conceitual para DBC.

Os elementos presentes no metamodelo para DBC representam:

Componente de Software (*Software Component*) é uma unidade de composição com interfaces e dependências de contexto explicitamente especificadas, que pode ser fornecido isoladamente para integrar sistemas de software desenvolvidos por terceiros. Um componente de software possui uma especificação abstrata que pode ser materializada em diferentes implementações.

Especificação de Componente (*Component Specification*) é uma abstração que define o comportamento observável externamente de um componente de software, de forma independente de qualquer implementação. Uma especificação de componente pode ser utilizada como elemento arquitetural para composição de diferentes arquiteturas de software.

Implementação de Componente (*Component Implementation*) é um módulo executável resultante de um processo de refinamento e tradução de uma especificação de componente. Uma implementação de componente pode ser utilizada como item de configuração para compor diferentes sistemas de software executáveis.

Componente Elementar (*Elementary Component*) é uma implementação atômica de um componente de software.

Componente Composto (*Composite Component*) é uma implementação de componente estruturada internamente como um conjunto de subcomponentes a serem providos separadamente. Uma implementação composta utiliza as especificações de seus subcomponentes como elementos de sua arquitetura de software interna. Em tempo de execução, implementações dos subcomponentes são integradas à implementação composta de acordo com essa arquitetura de software interna.

3.2.2 Metamodelo Conceitual de Especificação de Componentes

Os conceitos para especificação de componentes de software foram modelados no metamodelo conceitual de especificação de componentes, conforme apresentado na Figura 3.5. Os elementos presentes no metamodelo representam:

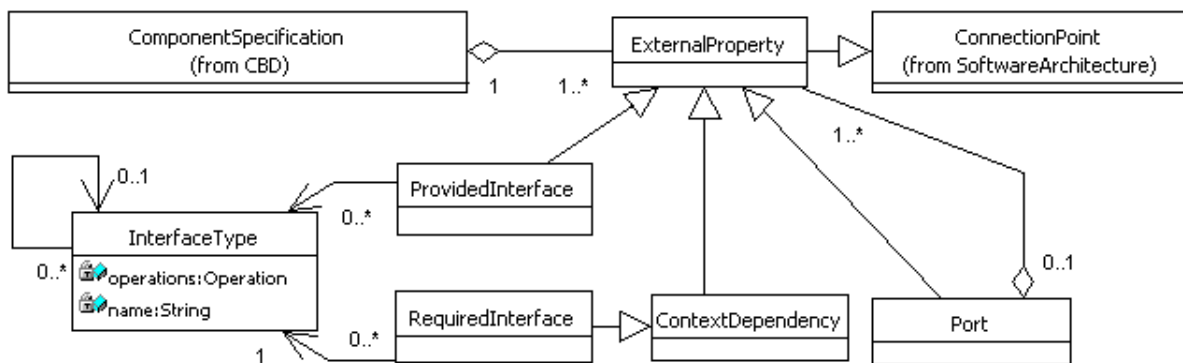


Figura 3.5: Metamodelo conceitual de especificação de componentes.

Propriedade Externa (*External Property*) é uma característica de uma especificação de componente de software que representa um de seus pontos de conexão.

Interface Provida (*Provided Interface*) é uma propriedade externa associada a um conjunto de serviços providos pelo componente de software. Esses serviços são especificados pelo tipo da interface provida. Uma interface provida pode ser utilizada como um ponto de conexão de um elemento arquitetural, fornecendo um meio de interação entre um componente e o ambiente onde está inserido.

Dependência de Contexto (*Context Dependency*) é uma propriedade externa associada a um conjunto de serviços que o componente de software requer do seu ambiente. As dependências de contexto incluem os serviços a serem providos por outros componentes ou pela infra-estrutura do sistema.

Interface Requerida (*Required Interface*) é um tipo de dependência de contexto associada a serviços a serem providos por outros componentes de software. Esses serviços são especificados pelo tipo da interface requerida. Assim como a interface provida, a interface requerida também pode ser utilizada como um ponto de conexão de um elemento arquitetural.

Tipo de Interface (*Interface Type*) define um conjunto de serviços e especifica o comportamento esperado desses serviços [29].

Porta (*Port*) é uma propriedade externa associada a um comportamento do componente de software que é encapsulado por um conjunto de interfaces providas ou requeridas. Uma porta também pode ser utilizada como um ponto de conexão de um elemento arquitetural.

3.2.3 Metamodelo Conceitual de Arquiteturas de Software Baseadas em Componentes

O metamodelo conceitual de componentes compostos realiza a conexão entre os metamodelos de arquitetura de software e de DBC. Neste metamodelo, componentes especificados segundo o metamodelo conceitual de especificação de componentes são interligados, formando uma arquitetura de componentes. A Figura 3.6 apresenta o metamodelo, que possui os seguintes elementos:

Arquitetura de Componentes (*Component Architecture*) descreve um conjunto de componentes de software, seus relacionamentos e suas dependências de comportamento [12].

Sub-Componente (*SubComponent*) é um componente interno a outro componente de software que é descrito por uma especificação.

Conexão de Interface (*Interface Connection*) é uma conexão entre uma interface requerida de um componente e uma interface provida de outro componente, representando uma possível interação uni-direcional, onde o segundo componente provê serviços requeridos pelo primeiro. Uma conexão de interface pode representar uma conexão entre dois elementos arquiteturais.

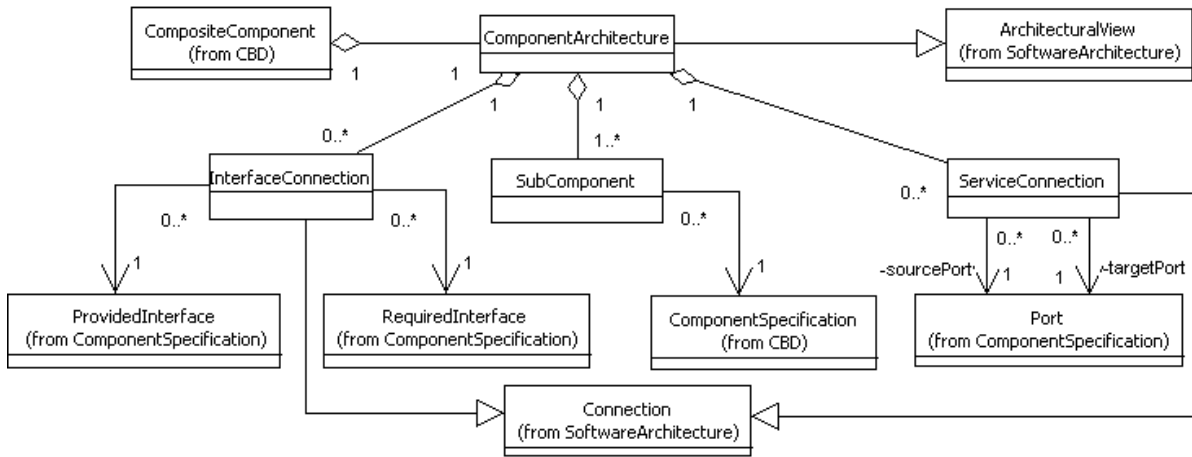


Figura 3.6: Metamodelo conceitual de uma arquitetura de componentes.

Conexão de Serviço (*Service Connection*) é uma conexão entre duas portas simétricas de dois componentes, representando uma possível interação bi-direcional entre esses componentes. Duas portas são ditas simétricas quando para toda interface provida através dessas portas haja uma correspondência com uma interface requerida de mesmo tipo, associada à outra porta. Assim como uma conexão de interface, uma conexão de serviço também pode representar uma conexão entre dois elementos arquiteturais.

3.2.4 Restrições Aplicadas ao Metamodelo

A Figura 3.7 apresenta as restrições OCL para o metamodelo para DBC. As restrições OCL para o metamodelo para arquitetura de software, apresentadas na Seção 3.1.3, também são válidas para o metamodelo conceitual de arquitetura de software baseada em componentes, dado que existe um mapeamento entre estes dois metamodelos. Apêndice A descreve os principais elementos da linguagem OCL utilizados nas restrições para o metamodelo para DBC, que são:

validName : estabelece que componentes de software devem possuir nomes distintos.

correctPortElements : estabelece que uma porta pode conter apenas interfaces providas e requeridas. A restrição define ainda que estas interfaces devem pertencer à mesma especificação de componente à qual a porta pertence.

```

context SoftwareComponent
inv validName :
    SoftwareComponent.allInstances->select(y| y.name=self.name)->size <= 1

context Port
inv correctPortElements :
    self.externalProperty->forAll(e|
        (e.ocllsTypeOf(ProvidedInterface) or e.ocllsTypeOf(RequiredInterface))
        and (e.componentSpecification = self.componentSpecification))

```

Figura 3.7: Restrições OCL para DBC.

3.2.5 Exemplo de Uso do Metamodelo

O exemplo presente no livro de Cheesman *et al.* [12] foi parcialmente modelado para representar um uso do metamodelo proposto. O exemplo consiste em um sistema de gerenciamento de reservas de hotel que tem como objetivo possibilitar a reserva de quartos em qualquer um dos hotéis que compõem uma certa cadeia de hotéis. Assim, esse sistema permite escolher o hotel, o tipo de quarto e o período da estadia para a reserva, bem como registrar a entrada efetiva do hóspede no hotel. O sistema está interligado ao sistema de cobranças da cadeia de hotéis, emitindo pedidos de cobrança das estadias e processando reservas não-concretizadas — usualmente chamadas de *no-show*.

No seu nível mais alto de abstração, o sistema é representado pelo componente de software **SistemaHotel**. A Figura 3.8 apresenta a especificação para o componente **SistemaHotel**. Esta especificação de componente pode ser mapeada para o metamodelo conceitual de especificação de componentes, apresentado na Seção 3.2.2. Um *ComponentSpecification* representará a especificação do componente **SistemaHotel**, que terá dois elementos de tipo *ProvidedInterface* associados e um elemento *RequiredInterface*. Um elemento *ProvidedInterface* do componente estará associado a um elemento *InterfaceType*, que descreve a interface **IFazerReserva**. O outro elemento *ProvidedInterface* estará associado a um *InterfaceType* que descreve a interface **IRegistrarEntrada** e o elemento *RequiredInterface* estará associado a um *InterfaceType* que descreve a interface **ICobranca**.



Figura 3.8: Especificação do componente SistemaHotel.

A visão geral do metamodelo conceitual para DBC, apresentada na Seção 3.2.1, descreve que uma especificação de componente pode possuir implementações. Uma implementação de componente pode ser elementar ou pode ser um componente composto. O componente **SistemaHotel** possui uma implementação formada por um componente composto, conforme apresentado na Figura 3.9. Esta arquitetura é composta pela especificação de componente **SistemaReservas**, responsável pelas regras de negócio relacionadas com reservas no hotel, e pela especificação do componente **GerCliente**, responsável por manipular as informações referentes aos clientes do hotel. A especificação de componente **SisReservas_GerCliente** representa o conector responsável por realizar a conexão entre estes dois componentes. Portanto, utilizando elementos presentes no metamodelo de arquiteturas de software baseadas em componentes, apresentado na Seção 3.2.3, um elemento *CompositeComponent* representará este componente composto para o componente **SistemaHotel**. Ele será composto por três elementos *Sub-Component*, que representarão as especificações de componente para os componentes **SistemaReservas**, **SisReservas_GerCliente** e **GerCliente**. Ainda, dois elementos do tipo *InterfaceConnection* estarão associados ao componente composto do **SistemaHotel**: um para conectar o componente **SistemaReservas** ao conector **SisReservas_GerCliente** e um para conectar o componente **SisReservas_GerCliente** ao componente **GerCliente**.

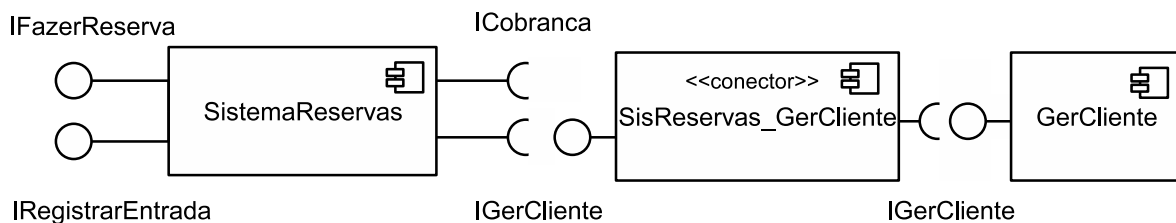


Figura 3.9: Implementação do componente SistemaHotel.

3.3 Metamodelo para Configuração Arquitetural de Componentes

3.3.1 Visão Geral

Nesta seção é apresentada a parte do metamodelo conceitual que trata da configuração arquitetural de componentes de software e seus conceitos. Uma configuração arquitetural de componentes, no contexto deste metamodelo, representa uma implementação para uma arquitetura de software baseada em componentes. A Figura 3.10 mostra o diagrama de classes UML referente a esse metamodelo, que possui os seguintes elementos:

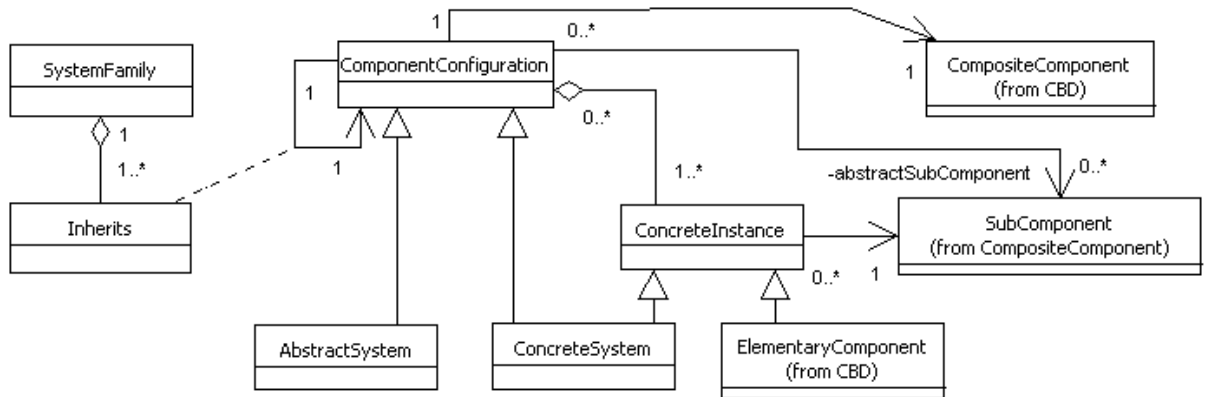


Figura 3.10: Metamodelo conceitual de configuração de componentes.

Configuração de Componente (*Component Configuration*) é um conjunto de instâncias concretas ou abstratas em conformidade com a arquitetura de um componente composto instanciado pela configuração.

Instância Concreta (*Concrete Instance*) é um sistema concreto ou um componente elementar.

Sistema Abstrato (*Abstract System*) é uma configuração de componente que concretiza apenas parte dos subcomponentes do componente composto instanciado pela configuração. Um subcomponente não concretizado por uma configuração é uma instância abstrata da configuração.

Sistema Concreto (*Concrete System*) é uma configuração de componente que não possui instâncias abstratas, ou seja, concretiza todos os subcomponentes do componente composto instanciado pela configuração.

Família de Sistemas (*System Family*) é um conjunto de configurações de componente relacionadas através de associações de herança simples. Sistemas de uma mesma família instanciam um mesmo componente composto. As instâncias concretas definidas pela configuração filha substituem instâncias abstratas ou concretas da configuração mãe. Uma configuração filha herda as instâncias concretas da configuração mãe que não são substituídas pela configuração filha.

3.3.2 Restrições Aplicadas ao Metamodelo

A Figura 3.11 apresenta uma restrição OCL para o metamodelo conceitual de configuração arquitetural de componentes. Apêndice A descreve os elementos da linguagem OCL utilizados nesta restrição:

hasCorrectSize : garante que a quantidade de instâncias concretas de um sistema concreto é igual à quantidade de subcomponentes do componente composto instanciado pela configuração. Isto garante que um sistema concreto possui instâncias concretas para todos os subcomponentes existentes para o componente composto ao qual está relacionado.

```
context ConcreteSystem
inv hasCorrectSize :
    self.concreteInstance->size() =
        self.compositeComponent.componentArchitecture.subComponent->size()
```

Figura 3.11: Restrições OCL para configuração de componentes.

3.3.3 Exemplo de Uso do Metamodelo

Utilizaremos o mesmo exemplo utilizado na Seção 3.2.5 para exemplificar o uso do metamodelo conceitual de configuração arquitetural de componentes. A Figura 3.12 apresenta uma sistema concreto para o componente composto **SistemaHotel**, que pode ser representado por um elemento **ConcreteSystem** do metamodelo. Este sistema concreto possui três elementos **ElementaryComponent**, que representam implementações para os componentes **SistemaReservas**, **SisReservas_GerCliente** e **GerCliente**. Neste exemplo foi assumido que a tecnologia EJB foi utilizada para implementar estes componentes.

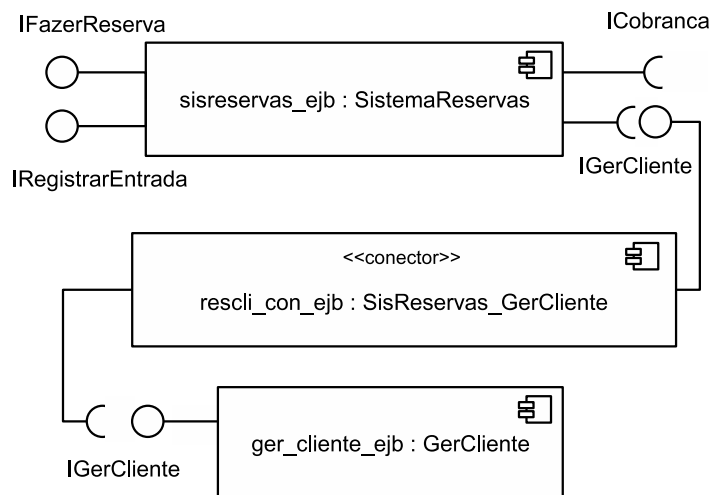


Figura 3.12: Configuração arquitetural do componente **SistemaHotel**.

3.4 Resumo

Neste capítulo foi apresentado um metamodelo conceitual integrado para arquitetura de software e DBC. O metamodelo é definido em linguagem UML e possui restrições em linguagem OCL para complementar a descrição de seus elementos. Ainda, cada elemento existente no metamodelo é definido de acordo com os principais conceitos de arquitetura de software e de DBC presentes na literatura. Durante o capítulo foi apresentado um exemplo que ilustra um uso do metamodelo conceitual. Este metamodelo serviu como base para a infra-estrutura de software para construção de arquiteturas de componentes, que será apresentada no próximo capítulo.

Capítulo 4

Uma Infra-estrutura de Software para Apoiar a Construção de Arquiteturas de Componentes

Neste capítulo é apresentada a infra-estrutura de software para construção de arquiteturas de software baseadas em componentes. Esta infra-estrutura de software usou como base o metamodelo conceitual integrado para arquitetura de software e DBC, apresentado no Capítulo 3. A Seção 4.1 apresenta os requisitos identificados e considerados para a construção da infra-estrutura de software deste trabalho. Estes requisitos influenciaram na criação dos casos de uso da infra-estrutura, que são apresentados na Seção 4.2. Na Seção 4.3 é feita uma comparação dos requisitos identificados para a infra-estrutura de software deste trabalho com os trabalhos relacionados, apresentados na Seção 1.4. Os editores e ferramentas estão divididos em dois grupos: um grupo relacionado a modelagem de arquiteturas de software baseadas em componentes, descrito na Seção 4.4, e um grupo relacionado à materialização destas arquiteturas através de uma implementação, detalhado na Seção 4.5. A Seção 4.6 apresenta as principais soluções e tecnologias utilizadas para a implementação da infra-estrutura de software deste trabalho.

4.1 Requisitos para a Infra-estrutura de Software

Os requisitos para a infra-estrutura de software para construção de arquiteturas de componentes foram identificados a partir da análise e do estudo das características apresentadas no *framework* proposto no trabalho de Medvidovic *et al.* [31]. O trabalho apresenta um *framework* para classificação e comparação de ADLs, elaborado a partir do estudo das principais ADLs existentes. O *framework* é formado por um conjunto de características para construção de arquiteturas de software, dividido em dois grupos: requi-

sitos para **modelagem de arquiteturas de software** e requisitos para **ferramentas de apoio a construção de arquiteturas de software**. Apesar do *framework* apontar características no contexto de arquitetura de software em geral, observamos que estas mesmas características poderiam ser utilizadas para a elaboração dos requisitos para a infra-estrutura de software para construção de arquiteturas de componentes. Apenas os requisitos relacionados a especificação dos componentes arquiteturais foram adaptados. Em DBC, os pontos de interação dos componentes são especificados por interfaces providas e requeridas, e não apenas indicados como pontos de conexão de um componente arquitetural (conforme apresentado na Seção 1.1 e na Seção 3.1.1).

Na Seção 4.1.1 são apresentados os requisitos para modelagem de arquiteturas de software baseadas em componente e na Seção 4.1.2 são apresentados os requisitos para as ferramentas de apoio à construção de arquiteturas de software baseadas em componentes.

4.1.1 Requisitos para modelagem de arquiteturas de componentes

Os requisitos para modelagem de arquiteturas de componentes definem o que deve ser representado em uma modelagem de uma arquitetura. Estes requisitos, seguindo o *framework* proposto por Medvidovic *et al.*, foram divididos em dois grupos: modelagem de componentes e conectores e modelagem de arquiteturas de componentes. Na modelagem de componentes e conectores são levados em consideração os requisitos para modelagem dos elementos que compõem a arquitetura. Para a modelagem de arquiteturas de componentes são compreendidos os requisitos que agem sobre a arquitetura de componentes de um sistema, criada a partir da conexão entre os componentes e conectores arquiteturais.

Os requisitos identificados para a modelagem de componentes e conectores são:

1. **Interfaces:** conforme apresentado na Seção 1.1, as interfaces representam os pontos de conexão entre os componentes ou conectores arquiteturais e o ambiente externo onde estão inseridos. Estas interfaces foram mapeadas no metamodelo conceitual de especificação de componentes através dos elementos *ProvidedInterface*, *RequiredInterface* e *Port*.
2. **Tipos:** um tipo de componente ou conector representa uma classe de elementos arquiteturais que possuem determinadas características em comum, conforme apresentado na Seção 2.1.1. Os tipos de componentes e conectores podem ser agrupados formando um estilo arquitetural. O metamodelo conceitual de estilos arquiteturais, apresentado na Seção 3.1.2, descreve os elementos necessários para a definição de estilos arquiteturais e tipos de componentes e conectores.

3. **Semântica:** semânticas devem ser atribuídas a componentes e conectores para acrescentar informações referentes a seu comportamento. As interfaces acrescentam semântica aos componentes e conectores, porém em um nível mais limitado. Semânticas de componentes ou conectores podem ser utilizadas para análises em arquiteturas, com o auxílio de ferramentas que interpretem estas semânticas e as avaliem. Em relação ao metamodelo de arquitetura de software apresentado na Seção 3.1, informações sobre semântica podem ser adicionadas a elementos da arquitetura (*ArchitecturalElement*) ou a tipos de elemento da arquitetura (*ArchitecturalElementType*).
4. **Restrições:** as restrições são regras impostas sobre os componentes e conectores arquiteturais. Qualquer violação sobre estas regras leva a modelagem da arquitetura a um estado inconsistente. As restrições podem definir limitações sobre a estrutura e a semântica de elementos da arquitetura. Estas restrições foram definidas no metamodelo de estilos arquiteturais (Seção 3.1.2), através do elemento *ArchitecturalConstraint*.
5. **Propriedades Não-Funcionais:** para uma arquitetura deve ser possível modelar propriedades não-funcionais, tais como segurança e desempenho. Estas preocupações em um nível maior de abstração permitem análises prévias sobre a arquitetura do sistema. Assim como informações sobre semântica, informações sobre propriedades não-funcionais podem ser adicionadas a elementos (*ArchitecturalElement*) do metamodelo de arquitetura de software ou para tipos de elemento da arquitetura (*ArchitecturalElementType*), apresentados na Seção 3.1.

Os requisitos identificados para a modelagem de arquiteturas de componentes são:

1. **Facilidade de Entendimento:** a arquitetura deve permitir que um sistema seja visualizado em um nível maior de abstração. Por este motivo, a configuração arquitetural deve ser apresentada de uma maneira em que possa ser facilmente entendida, preferencialmente com o auxílio de representações gráficas (diagramas).
2. **Decomposição Hierárquica:** através da decomposição hierárquica uma arquitetura pode ser descrita em diferentes níveis de detalhamento. A decomposição hierárquica permite a definição de arquiteturas internas a componentes ou conectores arquiteturais. Isto permite maior flexibilidade e poder para a modelagem das arquiteturas de componentes. O metamodelo de arquitetura de software, apresentado na Seção 3.1.1, apresenta este relacionamento de decomposição hierárquica através da arquitetura interna que um elemento arquitetural (*ArchitecturalElement*) pode possuir.

3. **Escalabilidade:** arquiteturas de software facilitam no tratamento da complexidade e do tamanho dos sistemas, pois descrevem estes sistemas em um nível mais alto de abstração. Isto exige que configurações arquiteturais para sistemas grandes e complexos, compostos por um número elevado de componentes e conectores arquiteturais, possam ser descritas. A decomposição hierárquica de elementos da arquitetura também auxilia na escalabilidade da arquitetura de software.
4. **Heterogeneidade:** em uma arquitetura de software os componentes arquiteturais podem possuir diferentes características, tais como: tamanho, protocolo de comunicação ou modelo e linguagem de implementação utilizados. Estes diferentes aspectos devem ser representados nas modelagens das arquiteturas de software baseadas em componentes. Em relação ao metamodelo apresentado na Seção 3.1.2, estas informações podem ser adicionadas a tipos de elemento da arquitetura (*ArchitecturalElementType*).
5. **Restrições:** restrições sobre configurações arquiteturais podem ser utilizadas para complementar as restrições definidas separadamente para componentes e conectores arquiteturais. As restrições sobre configurações arquiteturais definem regras a serem respeitadas quando os componentes e conectores são interligados. Estas restrições foram definidas no metamodelo de estilos arquiteturais (Seção 3.1.2), através do elemento *ArchitecturalConstraint*.
6. **Propriedades Não-Funcionais:** além das propriedades não-funcionais descritas para componentes e conectores, deve ser possível definir propriedades não-funcionais no contexto de uma configuração arquitetural. Estas propriedades poderão ser utilizadas para análises prévias sobre a arquitetura de software, juntamente com as propriedades não-funcionais definidas para os componentes e conectores. Informações sobre propriedades não-funcionais podem ser adicionadas a visões arquiteturais (*ArchitecturalView*) do metamodelo conceitual de arquitetura de software, apresentado na Seção 3.1.1.

4.1.2 Requisitos para ferramentas de apoio a construção de arquiteturas de componentes

Além dos requisitos para modelagem de arquiteturas de componentes, os seguintes requisitos foram definidos para a infra-estrutura de software para construção de arquiteturas de software baseadas em componentes:

1. **Descrição em uma ADL:** as arquiteturas devem ser descritas em uma linguagem formal que permita uma maior precisão na especificação da arquitetura e que

possibilite realizar verificações sobre esta descrição. Ainda que a ferramenta possua um modelo próprio para descrição de arquiteturas, é interessante que estas arquiteturas possam ser mapeadas para outras ADLs. Este mapeamento permite que arquiteturas descritas pela ferramenta possam ser utilizadas em outras ferramentas para descrição de arquiteturas de software. Este requisito não é listado explicitamente como uma característica no *framework* proposto por Medvidovic *et al.*, pois todas as ferramentas analisadas para a criação do *framework* são ferramentas para manipulações em ADLs.

2. **Auxílio à modelagem da especificação:** além de permitir a modelagem de arquiteturas de software, ferramentas para construção de arquiteturas de componentes devem guiar o arquiteto durante esta modelagem. A ferramenta pode auxiliar proativamente o arquiteto, através da emissão de sugestões de ações ou através da desabilitação de funcionalidades que possam gerar resultados inconsistentes. Outra forma de auxílio é o reativo, onde a ferramenta pode informar o arquiteto quando a arquitetura se encontrar em um estado não-permitido.
3. **Análises:** as modelagens de arquiteturas de software podem ser submetidas a análises. Estas análises podem apontar inconsistências descobertas em um nível maior de abstração de um sistema de software, permitindo a descoberta antecipada de problemas de projeto destes sistemas. As análises são realizadas avaliando a estrutura da arquitetura, suas propriedades, suas restrições e os tipos de seus elementos. Encontrar inconsistências a partir de análises é um dos pontos principais nas ferramentas para construção de arquiteturas de componentes.
4. **Refinamento:** uma arquitetura de software pode possuir diferentes níveis de detalhamento, até chegar a uma implementação final do sistema. Estes níveis de detalhamento representam um refinamento da arquitetura de um nível mais abstrato até a sua implementação. As ferramentas para construção de arquiteturas de componentes devem auxiliar este refinamento, disponibilizando maneiras de se rastrear uma modelagem de arquitetura até a sua implementação em forma de código.
5. **Geração de implementação:** a geração automática da implementação, em forma de código, de uma modelagem de arquitetura é outra característica importante das ferramentas que apóiam a construção de arquiteturas de software. Gerar uma implementação de maneira manual, a partir de uma modelagem de uma arquitetura de componentes, pode resultar em um sistema que possui uma arquitetura diferente da esperada. O uso de ferramentas aumenta a possibilidade de conformidade de uma implementação com sua arquitetura proposta e permite ainda realizar um rastreamento desta arquitetura. A implementação gerada pode ser vista como o último

nível de refinamento de uma arquitetura.

4.2 Casos de Uso

Identificamos e elaboramos os casos de uso da infra-estrutura de software desta dissertação a partir dos requisitos apresentados na seção anterior. Os casos de uso foram agrupados em cinco pacotes, conforme apresentado na Figura 4.1. Esta divisão teve como objetivo agrupar os casos de uso que possuísem funcionalidades relacionadas, melhorando a visualização e a descrição dos casos de uso da infra-estrutura de software. Os pacotes de casos de uso da infra-estrutura de software são:

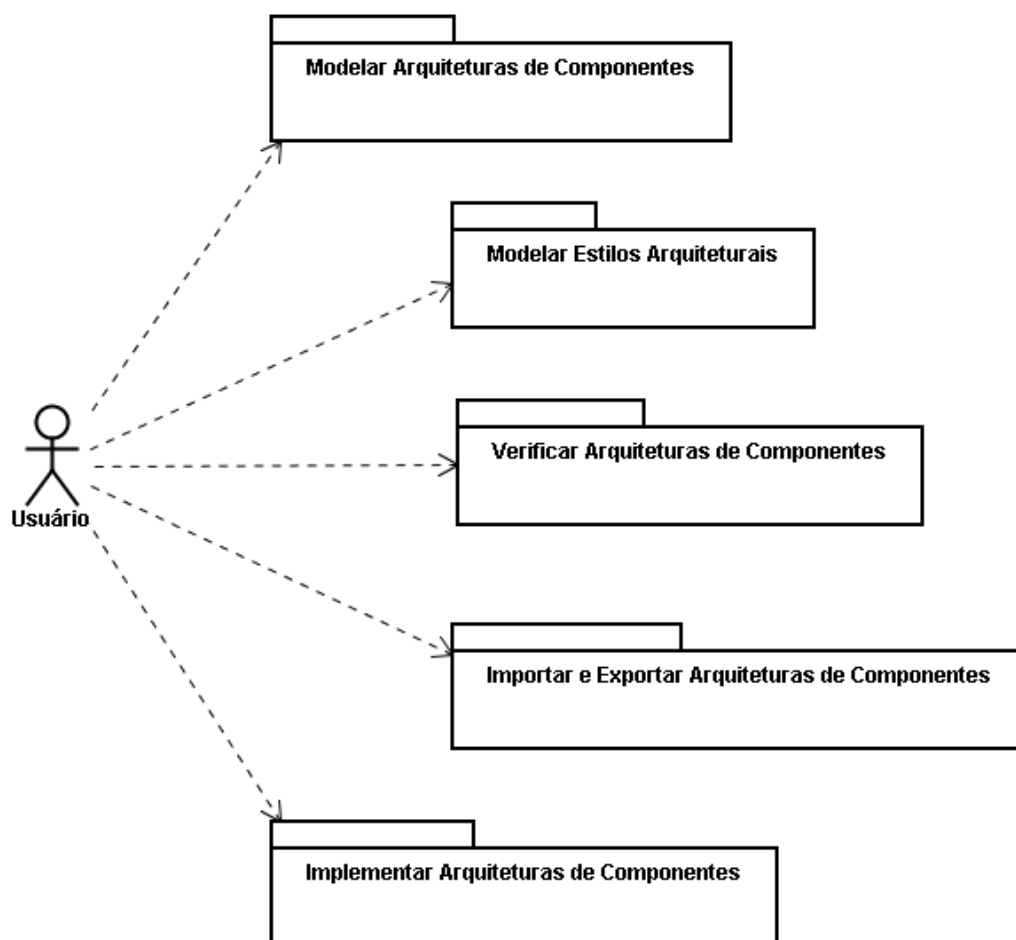


Figura 4.1: Casos de uso da infra-estrutura para construção de arquiteturas de componentes.

Modelar Arquiteturas de Componentes : possui os casos de uso relacionados a criação, edição e remoção de modelagens de arquiteturas de componentes. Estes casos de uso implementam os requisitos referentes a modelagem de arquiteturas de software baseadas em componentes, apresentados na Seção 4.1.1. As funcionalidades disponibilizadas por estes casos de uso são apresentadas com detalhes na Seção 4.4.1.

Modelar Estilos Arquiteturais : possui os casos de uso referentes a modelagem de estilos arquiteturais. Os requisitos utilizados para criação destes casos de uso foram os requisitos relacionados a definição de estilos arquiteturais, apresentados na Seção 4.1.1. As funcionalidades disponibilizadas por estes casos de uso são apresentadas com detalhes na Seção 4.4.2.

Verificar Arquiteturas de Componentes : possui os casos de uso para realização de verificações sobre as modelagens das arquiteturas de software baseadas em componentes. Foram implementados nestes casos de uso os requisitos “**Auxílio à modelagem da especificação**” e “**Análises**”, apresentados na Seção 4.1.2. As funcionalidades disponibilizadas por estes casos de uso são apresentadas com detalhes na Seção 4.4.3.

Importar e Exportar Arquiteturas de Componentes : possui os casos de uso que realizam a interoperabilidade de modelagens de arquiteturas de software baseadas em componentes com outras ferramentas de construção de arquiteturas. Estes casos de uso implementam o requisito “**Análises**”, apresentado na Seção 4.1.2. Através da exportação de arquiteturas é possível utilizar outras ferramentas que realizam análises sobre arquiteturas modeladas em linguagem Acme. O requisito “**Descrição em uma ADL**”, descrito na Seção 4.1.2, também é implementado neste pacote de casos de uso. As funcionalidades disponibilizadas por estes casos de uso são apresentadas com detalhes na Seção 4.4.4.

Implementar Arquiteturas de Componentes : possui os casos de uso para modelar uma implementação de uma arquitetura de componentes (através de uma configuração arquitetural de componentes) e gerar código desta arquitetura no modelo COSMOS (apresentado na Seção 2.5). Estes casos de uso implementam os requisitos “**Refinamento**” e “**Geração de implementação**”, apresentados na Seção 4.1.2. As funcionalidades disponibilizadas por estes casos de uso são apresentadas com detalhes na Seção 4.5.1 e na Seção 4.5.2.

4.3 Comparação com Outras Ferramentas de Arquiteturas de Componentes

Na Seção 1.4 foram apresentadas ferramentas que auxiliam na descrição de arquiteturas de software para sistemas baseados em componentes. As ferramentas apresentadas foram o AcmeStudio [44], o GME [28, 25], o Cadena [13], o Odyssey [9] e o PICML [4, 5]. Esta seção apresenta uma tabela comparativa entre estas ferramentas e a infra-estrutura de software proposta neste trabalho. Para a comparação foram utilizados os requisitos apresentados na Seção 4.1 e foi adicionada também a característica “*Inserção dentro de um ambiente DBC*”. Esta característica foi inserida pois é interessante que as arquiteturas de componentes produzidas possam ser integradas a outras ferramentas, possibilitando um maior ganho na construção de sistemas DBC. Para cada característica contida na tabela foi atribuído um dos seguintes valores: S (Sim), N (Não) ou P (Parcial).

Tabela 4.1: Funcionalidades disponibilizadas pelas ferramentas.

Funcionalidade	Infra-estrutura	Acme Studio	GME	Cadena	Odyssey	PICML
Modelagem de arquiteturas de componentes	S	P	P	S	S	P
Descrição em uma ADL	P	S	N	S	P	N
Auxílio a modelagem	S	S	P	S	S	P
Análises	S	S	P	S	P	P
Refinamento	S	S	S	N	S	S
Geração de Implementação	S	P	N	S	S	S
Inserção dentro de um ambiente DBC	S	N	N	S	S	S

4.4 Ferramentas para Modelagem de Arquiteturas de Software Baseadas em Componentes

No contexto de modelagem de arquiteturas de software baseadas em componentes, a infra-estrutura de software é composta pelas ferramentas: (i) Editor de Arquiteturas, (ii) Editor de Estilos Arquiteturais, (iv) Verificador de Arquiteturas e (iv) Importador/Exportador de Arquiteturas, conforme apresentado na Figura 4.2. Através do Editor de Componentes, existente no ambiente BELLATRIX, arquitetos e desenvolvedores de software podem criar especificações de componentes de software. No Editor de Arquiteturas estas especificações

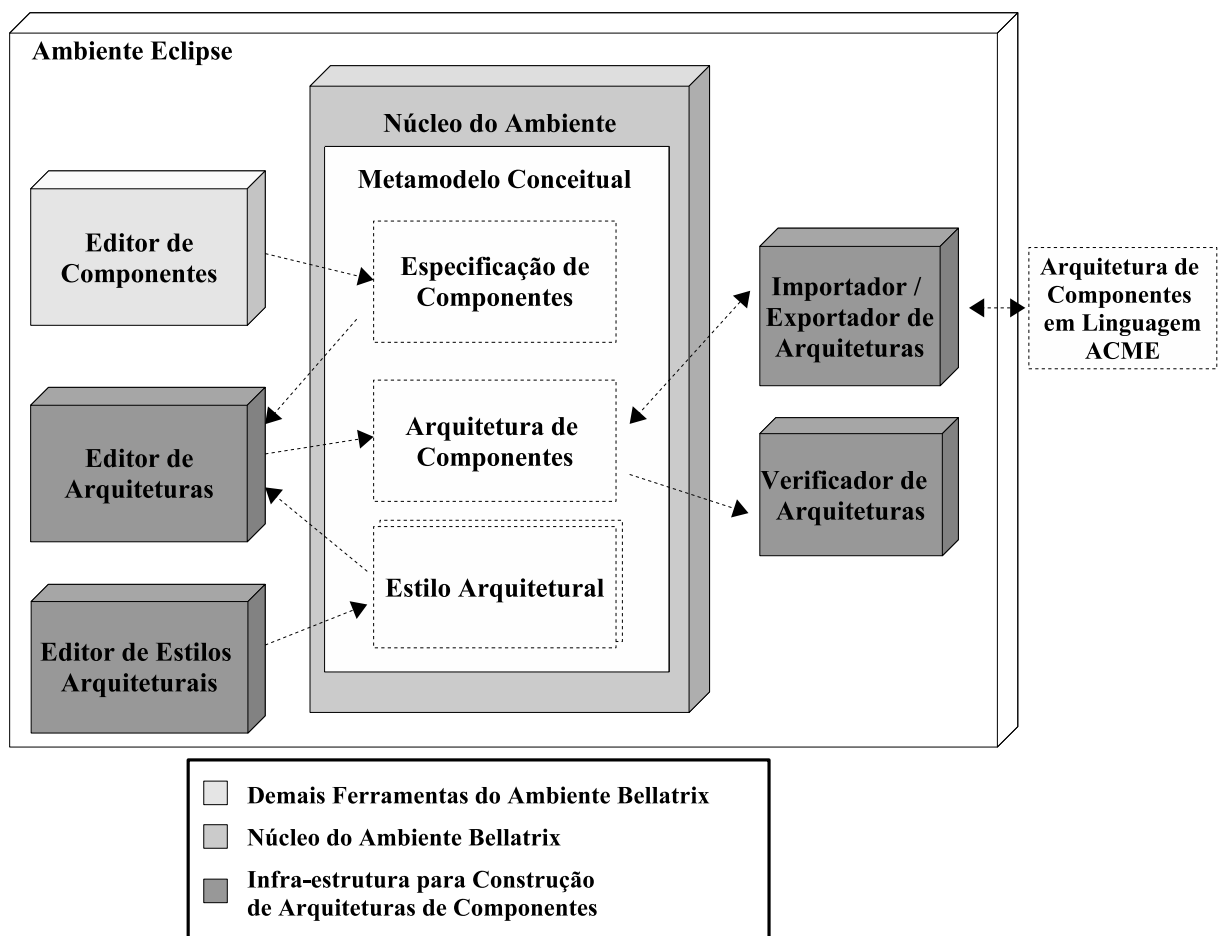


Figura 4.2: Visão geral das ferramenta de modelagem de arquiteturas de componentes.

de componentes podem ser utilizadas para criar modelagens de arquiteturas de componentes. Estas modelagens de arquiteturas podem ser associadas a estilos arquiteturais previamente descritos pelo Editor de Estilos. As arquiteturas de componentes podem ser exportadas ou importadas através do Importador/Exportador de Arquiteturas, utilizando a linguagem Acme. O Verificador de Arquiteturas permite que análises sejam realizadas sobre as arquiteturas de componentes modeladas.

As próximas seções detalham as ferramentas para modelagem de arquiteturas de software baseadas em componentes.

4.4.1 Editor de Arquiteturas de Componentes

O editor de arquiteturas é responsável por criar modelagens de arquiteturas de componentes. Conforme descrito no metamodelo de arquiteturas de software baseadas em componentes, apresentado na Seção 3.2.3, uma arquitetura de componentes sempre representa a implementação de uma especificação de um componente de software. Esta especificação pode ser um componente que represente o próprio sistema de software, ou ainda algum outro componente de software que possua uma arquitetura interna. Isso permite a decomposição hierárquica em níveis ilimitados de detalhamento.

As principais funcionalidades disponibilizadas pelo editor são:

- **Inserção de componentes:** as especificações de componentes são previamente descritas com o auxílio do editor de componentes, presente no ambiente BELLATRIX. O editor permite que uma mesma especificação seja utilizada mais de uma vez em uma arquitetura, conforme representado no metamodelo conceitual de componentes compostos (Seção 3.2.3).
- **Inserção de conectores:** assim como os componentes, os conectores também podem ser previamente especificados no editor de componentes e então inseridos em uma modelagem de arquitetura de componentes. Porém, o editor de arquiteturas exige que toda comunicação entre componentes arquiteturais deva passar por um conector. Por este motivo, o editor permite a criação automática de conectores quando dois componentes são interligados.
- **Conexão de interfaces:** o editor permite a conexão entre interfaces providas e requeridas de componentes e conectores da arquitetura.
- **Mapeamento entre interfaces internas e interfaces externas:** como as arquiteturas sempre representam a implementação de uma especificação de um componente de software, é possível mapear as interfaces desta especificação externa com as interfaces dos componentes de software de sua arquitetura interna. Este mapeamento é válido tanto para as interfaces providas quanto para as interfaces requeridas.

As arquiteturas podem ainda seguir um ou mais estilos arquiteturais. Os estilos arquiteturais devem estar descritos em linguagem Acme para serem associados a arquiteturas. Quando um estilo é adicionado a uma arquitetura de componentes, todos os tipos de componentes e conectores deste estilo são disponibilizados pelo editor para serem utilizados na arquitetura. O editor disponibiliza ainda um painel onde os valores das propriedades arquiteturais, herdadas dos tipos de componentes e conectores, podem ser manipuladas.

A Figura 4.3 apresenta um exemplo de uma arquitetura de componentes criada no editor de arquiteturas. À esquerda da figura são apresentados os artefatos produzidos durante a especificação das interfaces, dos componentes e da arquitetura de componente. Na parte central se encontra o editor gráfico onde as arquiteturas de componentes são criadas. Neste exemplo, a arquitetura é formada pelos componentes **Comp1**, **Comp2** e **Comp3** e pelos conectores **ConectorComp1Comp2** e **ConectorComp2Comp3**. O componente maior, que engloba a arquitetura de componentes, é o componente de software que possui essa arquitetura interna. No exemplo, este componente representa o próprio sistema. Na arquitetura é possível observar ainda as conexões entre as interfaces providas e requeridas dos componentes e o mapeamento entre a interface **I1** do componente **Comp1** e a interface **I1** do componente **Sistema**.

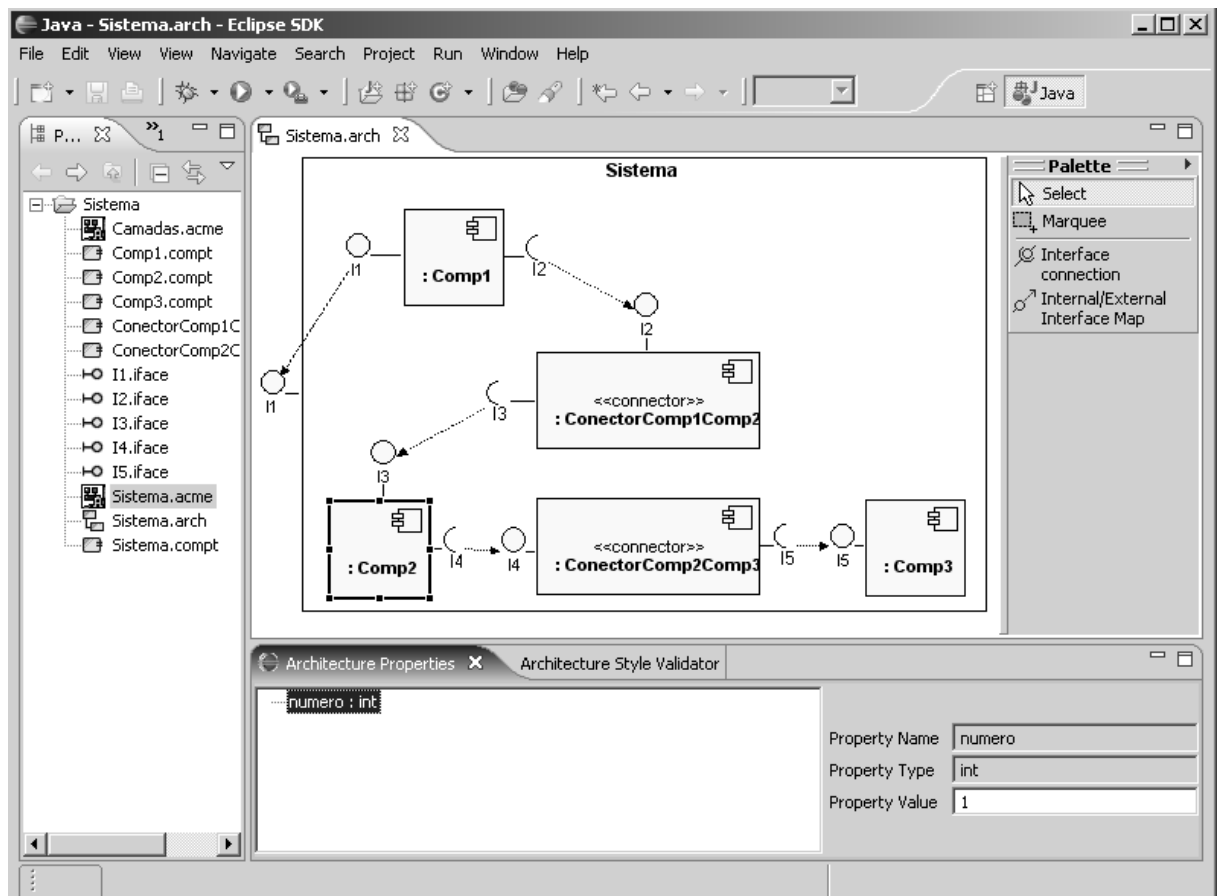


Figura 4.3: Exemplo de uma arquitetura de componentes.

No exemplo apresentado na Figura 4.3, a arquitetura segue o estilo arquitetural *Ca-*

camadas (o estilo em camadas foi definido na Seção 2.1.1 e um exemplo de implementação deste estilo através do Editor de Estilos é apresentado na Seção 4.4.2). Os componentes **Comp1**, **Comp2** e **Comp3** são do tipo *camada*, sendo **Comp1** pertencente a camada 2 e os componentes **Comp2** e **Comp3** pertencentes a camada 1. Na figura, a propriedade *numero* do componente **Comp2** está sendo exibida no painel *Architecture Properties*.

4.4.2 Editor de Estilos Arquiteturais

Através do Editor de Estilos Arquiteturais é possível definir estilos arquiteturais que representem determinadas famílias ou classes de sistemas. Os estilos são compostos por tipos de componentes e conectores (conforme apresentado nas Seções 2.1.1 e 3.1.2). Estes tipos são especificados através de um conjunto de propriedades e de um conjunto de restrições. As propriedades descrevem a semântica e as propriedades não-funcionais para os tipos de elementos arquiteturais. As restrições podem ser impostas sobre a estrutura da arquitetura ou sobre as propriedades dos tipos dos elementos arquiteturais.

Os estilos arquiteturais devem ser descritos em uma ADL para permitir que arquiteturas construídas segundo determinados estilos possam ser analisadas, garantindo sua conformidade com as restrições impostas por sua classe de sistemas. A ADL escolhida para especificar os estilos arquiteturais foi a linguagem Acme, pois permite a descrição de restrições arquiteturais genéricas, que podem englobar diferentes aspectos da família (a linguagem Acme foi apresentada na Seção 2.1.2). A ADL Acme permite ainda maior integração com outras ferramentas, dado que a linguagem se propõe a ser uma linguagem de intercâmbio entre ADLs.

O Editor de Estilos Arquiteturais é o único editor da infra-estrutura de software deste trabalho que foi totalmente reaproveitado de outro ambiente, o AcmeStudio. O AcmeStudio foi construído através de um conjunto de *plug-ins* para o ambiente Eclipse e já possui todas as funcionalidades necessárias para especificar estilos arquiteturais em linguagem Acme. Desta forma, os *plug-ins* do AcmeStudio referentes a edição de estilos arquiteturais foram utilizados dentro da infra-estrutura de software para construção de arquiteturas de componentes.

O editor permite a modelagem de estilos arquiteturais diretamente em forma de texto em linguagem Acme ou com o auxílio de um editor gráfico. O editor em formato texto guia a especificação do estilo através de um verificador de sintaxe para a linguagem Acme. Com a utilização do editor gráfico é possível criar os tipos de componentes e conectores com o auxílio de janelas e atribuir propriedades e restrições sobre os tipos dos componentes e conectores arquiteturais. O editor possui ainda um verificador para a linguagem Armani (apresentada na Seção 2.1.2), auxiliando o arquiteto durante a definição das restrições.

A Figura 4.4 apresenta um exemplo de um estilo arquitetural definido utilizando o

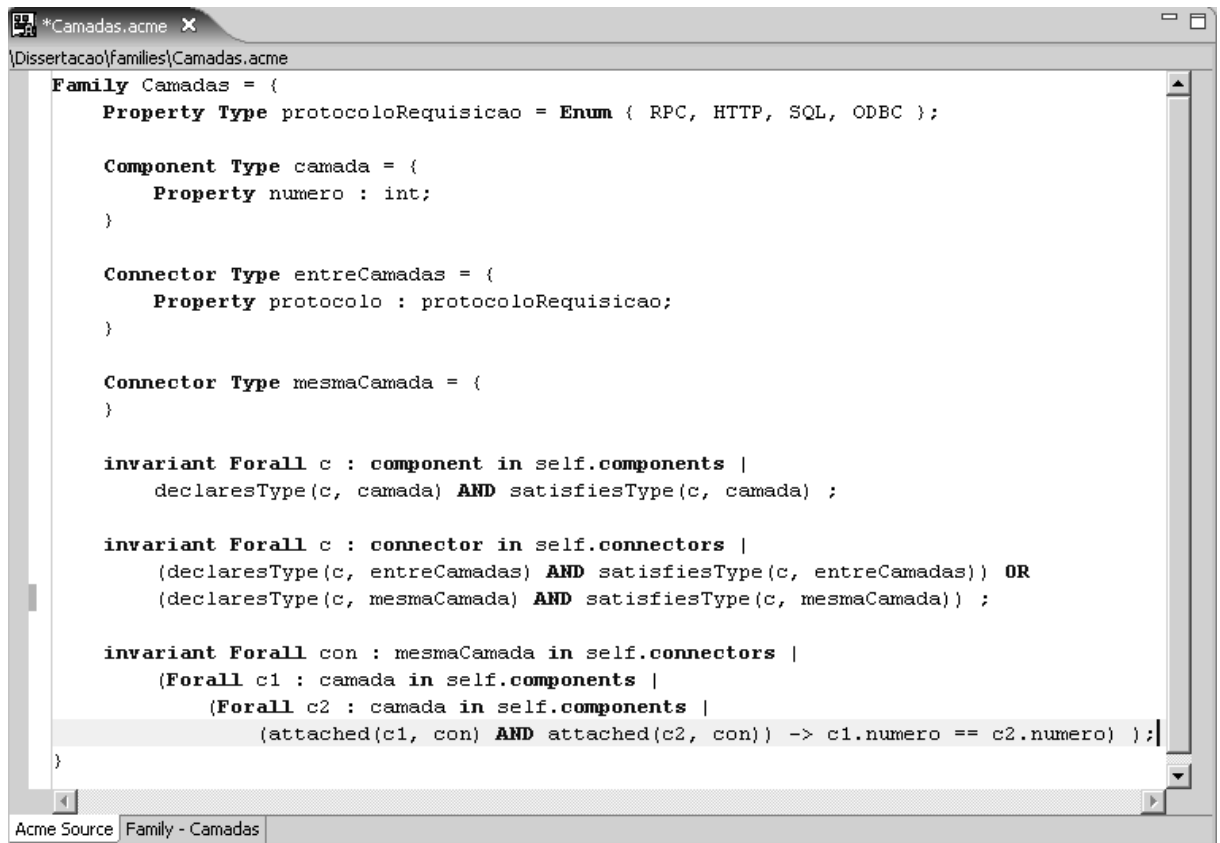


Figura 4.4: Editor de Estilos - modo texto.

modo texto do editor de estilos. O estilo criado se chama **Camadas** e possui o tipo de componente **camada** e dois tipos de conectores, **entreCamadas** e **mesmaCamada**. O tipo de componente **camada** possui a propriedade arquitetural **numero**, que representa qual o número da camada a qual o componente pertence. O tipo de conector **entreCamadas** realiza a conexão de componentes pertencentes a camadas diferentes e possui a propriedade **protocolo**, que indica qual o tipo de protocolo utilizado pelo conector para realizar a comunicação. O tipo de conector **mesmaCamada** conecta componentes de uma mesma camada.

Este mesmo estilo pode ser visualizado e manipulado com o auxílio do modo gráfico do editor de estilos, conforme apresentado na Figura 4.5. Os tipos de componentes e conectores podem ser criados, manipulados ou excluídos na janela apresentada na parte superior da figura. As propriedades arquiteturais e as restrições existentes no estilo e nos tipos de componentes e conectores podem ser manipuladas na aba *Element View*, que é apresentada na parte inferior da figura. As restrições são apresentadas na sub-aba *Rules*,

que realiza a verificação da sintaxe das restrições escritas em linguagem Armani.

Outros artifícios são utilizados para aumentar o poder de representação dos tipos, das propriedades arquiteturais e das restrições. O editor permite a definição de tipos de propriedades arquiteturais, que podem representar estruturas mais complexas para os valores das propriedades arquiteturais. Sem a utilização de tipos de propriedades, apenas tipos primitivos poderiam ser utilizados para as propriedades arquiteturais, tais como *inteiros* e *strings*. É possível também utilizar tipos de portas para os componentes e conectores e descrever restrições sobre estes tipos. Ainda, a linguagem Acme permite que os tipos de elementos arquiteturais possam referenciar outros tipos, utilizando um relacionamento de herança. Isto permite maior poder na especificação dos estilos.

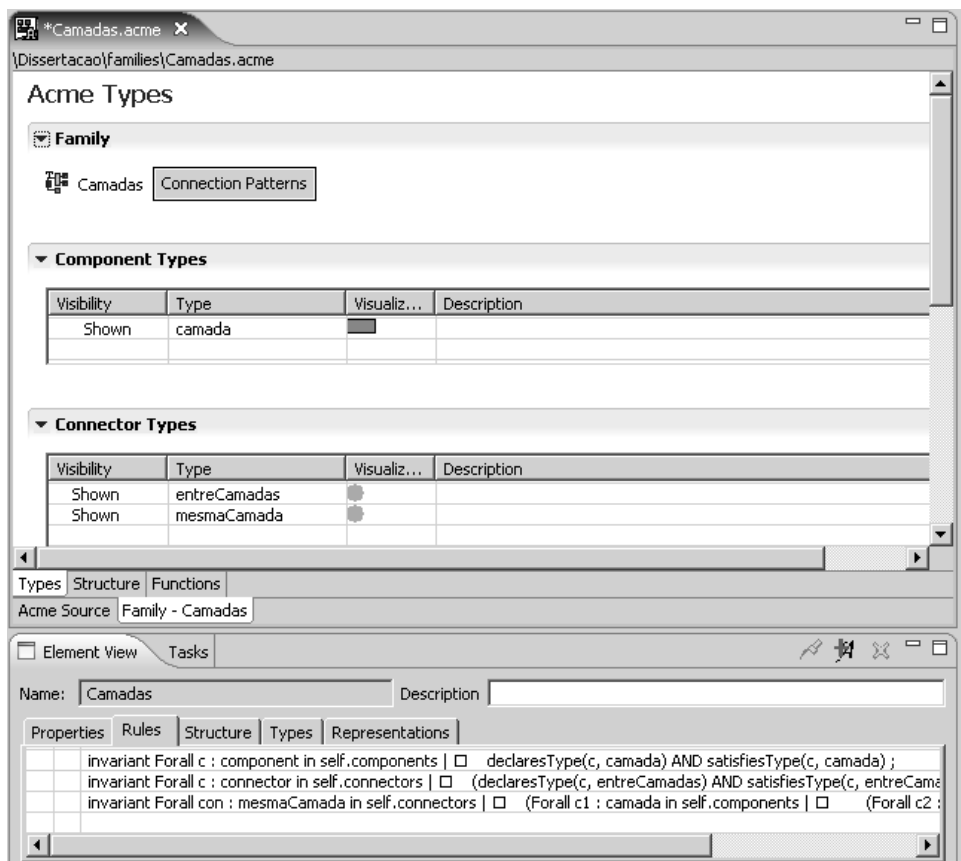


Figura 4.5: Editor de Estilos - modo gráfico.

4.4.3 Verificador de Arquiteturas de Componentes

As arquiteturas de componentes modeladas no editor de arquiteturas podem ser verificadas pelo Verificador de Arquiteturas. O verificador pode ser executado através do painel *Architecture Style Validator* presente no editor de arquiteturas, conforme apresentado na Figura 4.6. A função do verificador é analisar a conformidade da arquitetura de componentes com as restrições impostas por seus estilos arquiteturais. Caso a arquitetura de componentes não siga nenhum estilo arquitetural, nenhuma análise é realizada.

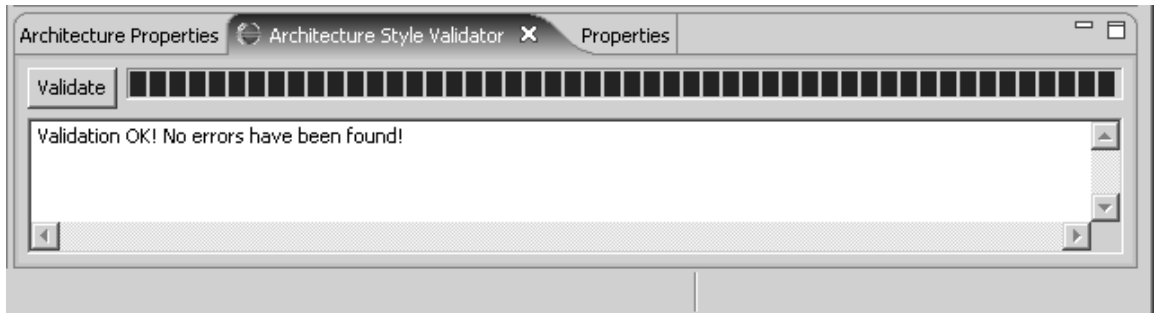


Figura 4.6: Exemplo de uma validação de uma arquitetura de componentes.

Por utilizar a linguagem Acme para a modelagem dos estilos arquiteturais, o validador de arquiteturas analisa restrições arquiteturais amplas e genéricas, de acordo com as cláusulas escritas em linguagem Armani. O validador realiza a análise de acordo com os tipos atribuídos aos elementos da arquitetura e através dos valores definidos para as propriedades arquiteturais definidas para cada elemento. Caso sejam encontradas inconsistências na arquitetura, os erros são apresentados em um campo texto, conforme apresentado na Figura 4.6. O validador permite ainda a personalização das mensagens de erro que serão exibidas quando inconsistências forem encontradas. Na linguagem Acme é possível inserir comentários acima de cada uma das cláusulas de restrição definidas para um estilo. Estes comentários são recuperados e exibidos quando alguma inconsistência é encontrada durante a verificação.

A Figura 4.7 apresenta um exemplo de uma cláusula que possui uma mensagem de erro personalizada. Esta mensagem foi emitida porque a cláusula não foi satisfeita para a arquitetura em questão.

4.4.4 Importador/Exportador de Arquiteturas de Componentes

A infra-estrutura de software deste trabalho possibilita sua integração com outras ferramentas de construção de arquiteturas de componentes, através do Importador/Exportador

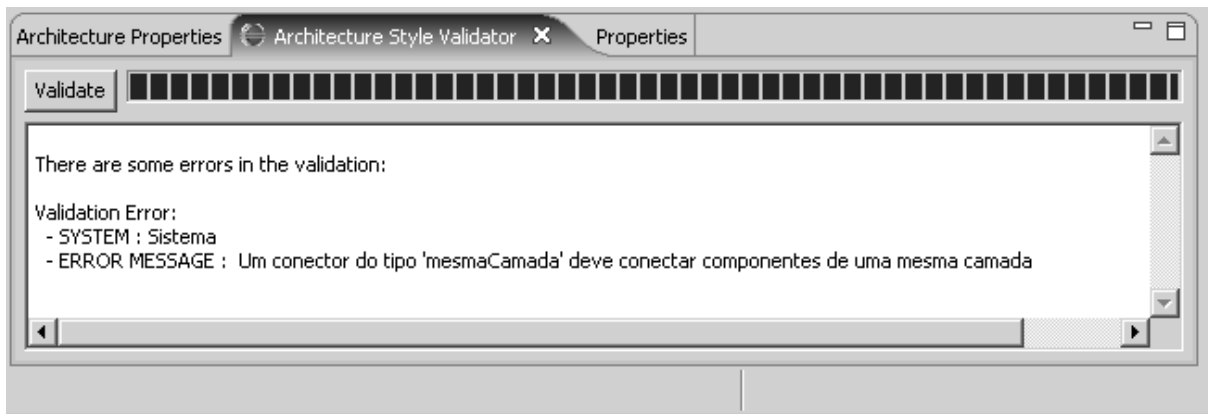


Figura 4.7: Exemplo de uma mensagem de erro do validador de arquiteturas.

de Arquiteturas. A importação de arquiteturas modeladas por outras ferramentas é importante pois permite o reaproveitamento e migração de especificações de arquiteturas para o ambiente BELLATRIX. Por outro lado, a exportação de arquiteturas é interessante pois possibilita que outras ferramentas para análise de arquiteturas possam ser utilizadas. Apesar do verificador realizar análises genéricas sobre as arquiteturas de componentes, algumas ferramentas já foram desenvolvidas para determinadas classes de interesse. Um exemplo é o Aereal [23], um *framework* que analisa os fluxos excepcionais dos componentes de uma arquitetura de software baseada em componentes.

A linguagem utilizada para a importação e exportação de arquiteturas é a ADL Acme, devido a sua capacidade de atuar como uma linguagem de intercâmbio entre ADLs (conforme Seção 2.1.2). Porém, como a linguagem Acme é destinada para a representação de arquiteturas de software, não se restringindo para arquiteturas de sistemas baseados em componentes, alguns conceitos existentes em DBC não estão compreendidos na linguagem. A principal limitação está na especificação dos componentes, em especial as interfaces providas e requeridas. Na linguagem Acme, os pontos de interação entre os componentes e conectores arquiteturais são representados apenas como portas.

Um estilo arquitetural para DBC foi definido para solucionar o problema de estruturar as portas dos componentes arquiteturais como interfaces providas ou requeridas. Este estilo define um conjunto de tipos de propriedades arquiteturais estruturado de forma a representar as interfaces providas e requeridas, assim como suas operações. A Figura 4.8 apresenta como um componente **Comp1** seria exportado pela ferramenta, utilizando o estilo arquitetural DBC. O componente possui uma interface provida **I1**, composta pela operação **operacao1**. A operação possui os parâmetros **param1** (de tipo int) e **param2** (de tipo String). Este estilo DBC possibilita que descrições de arquiteturas sejam exportadas

pela ferramenta sem perder as informações pertencentes às interfaces dos componentes. Ainda, o estilo DBC deve ser utilizado para a importação de arquiteturas. O estilo arquitetural DBC deve ser associado a uma arquitetura descrita em linguagem Acme e as portas de seus componentes e conectores devem receber tipos, definindo assim quais portas representam suas interfaces providas e quais portas representam suas interfaces requeridas. A definição completa do estilo DBC em linguagem Acme se encontra no Apêndice B.

```

System CBDSystem : CBDFamily = new CBDFamily extended with {
  Component Comp1 : CBDComponent = new CBDComponent extended with {
    Port I1 : ProvidedInterface = new ProvidedInterface extended with {
      Property operations : Sequence<Operation> = < [
        identifier : string = "operation1";
        parameters : Parameter = < [
          order : int = 0;
          identifier : string = "param1";
          attribute : ParamAttribute = _IN;
          paramType : string = "int"
        ],
        [order : int = 1;
          identifier : string = "param2";
          attribute : ParamAttribute = _IN;
          paramType : string = "String"
        ]
      >;
      returnType : string = "void";
      postCondition : string = "";
      preCondition : string = ""
    ]>;
  };
};

```

Figura 4.8: Exemplo de um componente que segue o estilo DBC.

4.5 Ferramentas para Construção de Arquiteturas de Software Baseadas em Componentes

Duas ferramentas são responsáveis pela materialização de uma arquitetura de software baseada em componentes em código, conforme apresentado na Figura 4.9. O Editor de Configurações é responsável por mapear uma arquitetura de componentes em uma implementação, modelando como esta arquitetura está sendo refletida em código. Para isso ele utiliza como entrada uma modelagem de uma arquitetura de componentes. O Gerador de Código recupera uma modelagem de uma configuração arquitetural de componentes e a materializa em código seguindo o modelo COSMOS. As próximas seções apresenta com detalhes estas duas ferramentas.

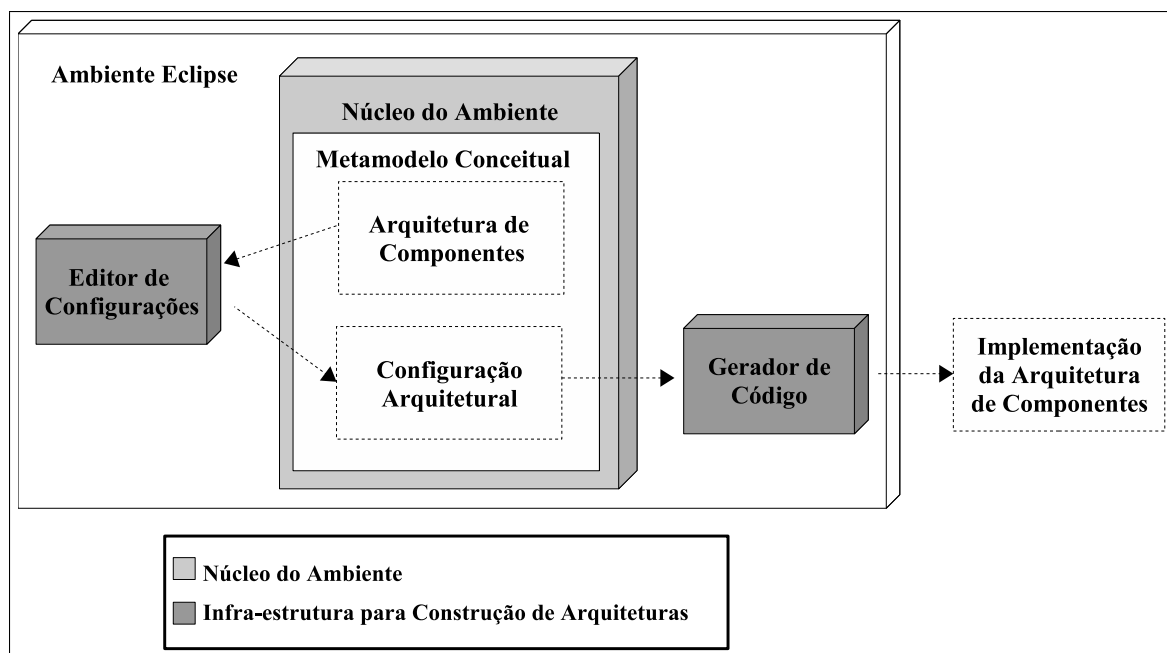


Figura 4.9: Visão geral das ferramentas para construção de arquiteturas de componentes.

4.5.1 Editor de Configuração de Componentes

O objetivo do editor de configurações é descrever como uma arquitetura de componentes foi implementada, ou seja, descrever como uma arquitetura foi materializada em forma de um sistema executável. Para isto é necessário definir, para cada componente e conector da arquitetura, qual a sua implementação.

Conforme descrito no metamodelo de configuração de componentes apresentado na Seção 3.3 do Capítulo 3, os componentes de software e os conectores de uma arquitetura possuem instâncias concretas, que podem ser de dois tipos: componentes elementares ou sistemas concretos. No editor de configurações um componente elementar é definido através das informações de: **localização** - determina o local onde a implementação deste componente se encontra - e **tecnologia** ou **modelo de componente** utilizado.

Os sistemas concretos são representados por outras configurações de componentes. Este caso ocorrerá quando um componente ou conector de uma arquitetura possuir uma arquitetura interna. Esta arquitetura interna possuirá uma configuração de componentes que a materializa. Esta configuração de componentes poderá ser utilizada como um sistema concreto.

A Figura 4.10 apresenta uma configuração arquitetural para a arquitetura apresentada na Figura 4.3. O painel *Properties*, que é apresentado na parte inferior da figura, permite que as informações sobre as implementações dos componentes e conectores sejam exibidas e manipuladas. Neste exemplo, a implementação para o componente **Comp1** foi definida como sendo uma implementação em linguagem Java.

4.5.2 Geração de Código em Modelo COSMOS

Como já discutido anteriormente (Seção 4.1.2), é muito interessante que a arquitetura de software de um sistema possa ser materializada em código de forma automática, garantindo que a implementação da arquitetura segue sua especificação. O modelo de componentes COSMOS é uma forma de materializar os componentes arquiteturais em forma de código através da utilização de uma linguagem orientada-a-objetos, conforme descrito na Seção 2.5. Porém, a implementação manual de uma arquitetura seguindo o modelo COSMOS possui duas dificuldades: (1) trabalho elevado para a criação das estruturas impostas pelo modelo (pacotes, componentes e interfaces) e (2) possibilidade de erros durante a implementação manual de uma especificação de uma arquitetura, o que pode gerar inconsistências com a especificação da arquitetura proposta. O uso de uma ferramenta que automatize esta geração de código em modelo COSMOS permite que o código de um sistema seja gerado mais rapidamente e com maior fidelidade em relação à sua especificação.

O gerador de código realiza a conversão automática de uma especificação de uma arquitetura de componentes em uma implementação em código Java, seguindo o Modelo COSMOS. O gerador de código cria os seguintes elementos para uma implementação de uma arquitetura:

- **Estrutura:** O gerador de código cria todos os pacotes, classes e interfaces descritas no modelo COSMOS para todos os componentes e conectores existente na arquitetura.

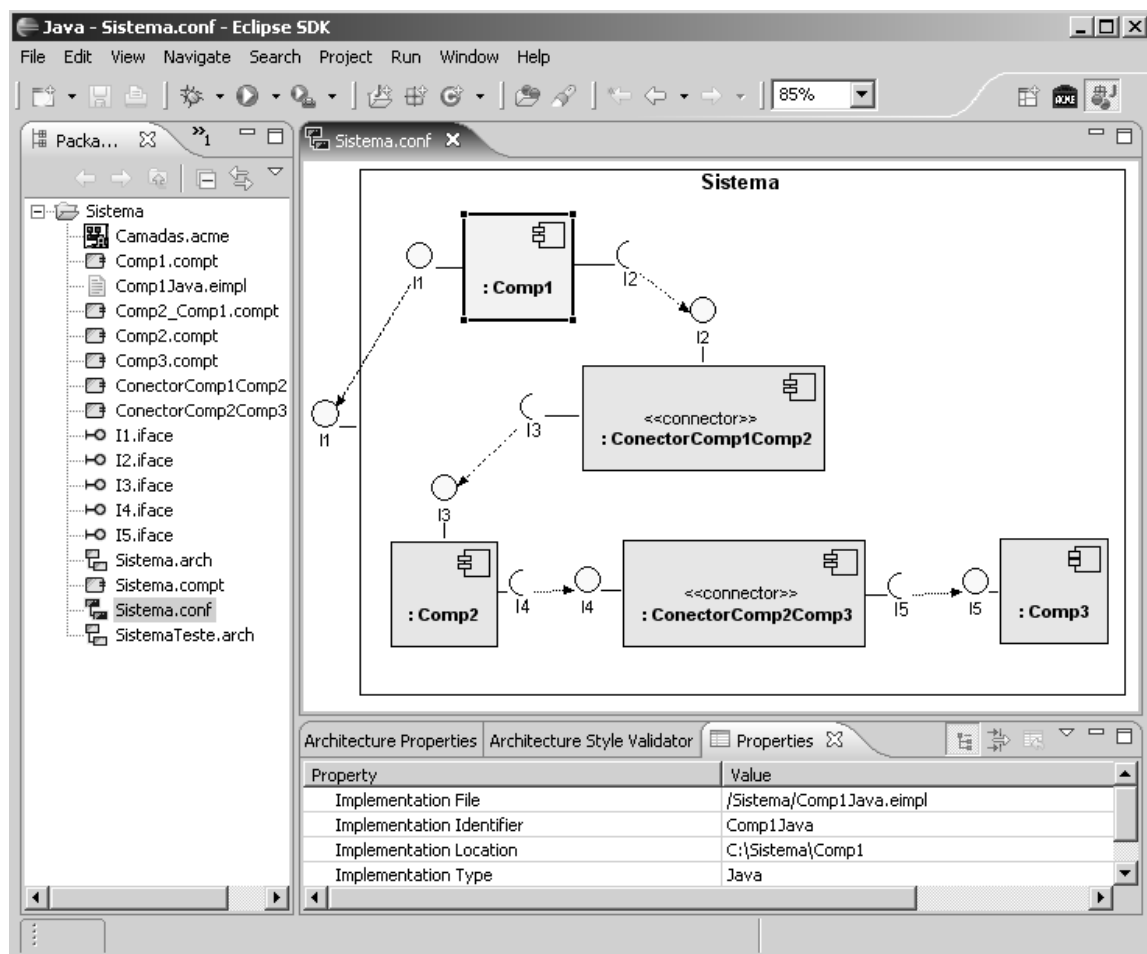


Figura 4.10: Exemplo de uma configuração de componentes.

tura, respeitando os relacionamentos existentes. O modelo COSMOS foi estendido para dar apoio aos sistemas que utilizam tipos de dados em suas interfaces. Um pacote `dt` foi incluído dentro do pacote de especificação (`spec`) para armazenar os tipos de dados. Este pacote deve conter classes públicas que representem os tipos de dados utilizados pelas interfaces providas e requeridas de um componente. Estas classes devem conter apenas um conjunto de atributos que podem ser alterados ou recuperados. A Figura 4.11 apresenta um exemplo de um componente a que possui o pacote `dt` para seus tipos de dados.

Caso os componentes e conectores da arquitetura possuam arquiteturas internas, as estruturas internas também serão geradas, em todos os níveis existentes de decomposição. Todos estes níveis de estruturas, criadas nos diferentes níveis de detalha-

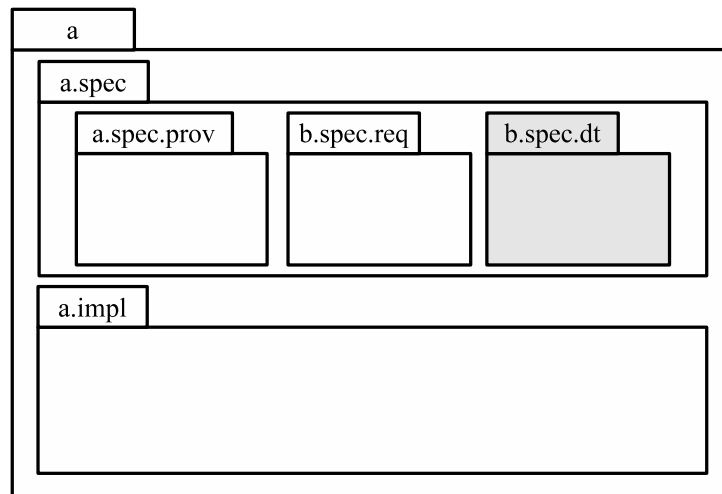


Figura 4.11: Evolução do modelo COSMOS para representar tipos de dados.

mento da arquitetura, garantem que toda a estrutura especificada para a arquitetura está mapeada em código.

- **Código de instanciação:** o gerador de código não se limita a gerar apenas a estrutura dos componentes e conectores. O gerador de código cria também todo o código de instanciação para os componentes e conectores. Como todas as arquiteturas possuem um componente externo materializado em código, a implementação deste componentes (ou conector) externo é responsável por instanciar cada componente e conector de sua arquitetura. Para conectar os componentes a seus conectores, o código de instanciação recupera as classes dos conectores que implementam as interfaces requeridas de cada componente, de acordo com os relacionamentos presentes na arquitetura, e as passam aos componentes que requerem estas interfaces. Isto garante uma correta instanciação dos componentes e conectores e uma correta conexão entre as interfaces providas e requeridas.
- **Ligação entre interfaces internas e externas:** as especificações de arquiteturas de componentes definem mapeamentos entre interfaces pertencentes ao componente externo da arquitetura e as interfaces dos componentes internos. O gerador de código cria estes mapeamentos através de classes que implementam estas interfaces providas externas e usam as interfaces pertencentes a arquitetura interna do componente ou conector. De forma análoga, as interfaces requeridas também são mapeadas através de classes.

Apesar do gerador de código ser restrito a gerar implementações em código Java se-

guindo o modelo COSMOS, a estrutura extensível do ambiente Eclipse através da inclusão de *plug-ins* permite que outros geradores de código para diferentes tecnologias e modelos possam ser incluídos na ferramenta. Isto é facilitado porque as arquiteturas estão especificadas sobre metamodelos conceituais que englobam os conceitos presentes em arquitetura de software e DBC.

4.6 Implementação da Ferramenta

Esta seção descreve a arquitetura interna da infra-estrutura de software para construção de arquiteturas de componentes e as soluções de projeto e técnicas utilizadas para a sua construção. A linguagem de programação Java foi utilizada para a implementação da infra-estrutura de software, dado que as ferramentas foram desenvolvidas como extensões da ambiente Eclipse (o ambiente Eclipse foi apresentado na Seção 2.4). As próximas seções apresentam as principais características e decisões utilizadas durante a construção desta infra-estrutura de software.

4.6.1 Arquitetura de Componentes da Infra-estrutura de Software

A infra-estrutura de software deste trabalho foi internamente desenvolvida baseada em componentes de software. O processo de desenvolvimento utilizado para a especificação dos componentes e da arquitetura da infra-estrutura de software foi o processo UML Components, apresentado na Seção 2.3.

De acordo com o processo UML Components, a primeira fase é a de **requisitos**, onde os casos de uso devem ser descritos e um modelo conceitual de negócio (do inglês: *business conceptual model*) deve ser elaborado. Os casos de uso foram especificados e detalhados em dois relatórios técnicos [54, 35] e o modelo conceitual de negócio utilizado foi o metamodelo conceitual apresentado no Capítulo 3.

Na fase de **especificação de componentes** do processo UML Components os casos de uso e o modelo conceitual de negócio são utilizados para identificar as interfaces de sistema e as interfaces de negócio. Estas interfaces irão guiar a identificação dos componentes de sistema e dos componentes de negócio. As interfaces de sistema das ferramentas, incluindo suas operações, foram descritas a partir das especificações dos casos de uso. As interfaces de sistema foram então agrupadas segundo suas funcionalidades, formando os componentes de software de sistema. Os componentes de sistema resultantes deste agrupamento foram: o componente **ArchitectureSystem**, responsável pelas funcionalidades ligadas à especificação de uma arquitetura; e o componente **ImplementationSystem**, responsável pelas funcionalidades referentes à concretização da arquitetura de componentes

em forma de implementação.

As interfaces de negócio foram identificadas a partir dos elementos centrais existentes no metamodelo conceitual. Foram selecionados o elemento *EspecificaçãoDeComponente* (*ComponentSpecification*) e o elemento *ImplementacaoDeComponente* (*ComponentImplementation*), que resultaram nas interfaces de negócio *IComponentSpecificationMgt* e *IComponentImplementationMgt*. A interface *IComponentSpecificationMgt* contém as operações que permitem recuperar informações sobre especificações de componentes (ou conectores). A interface *IComponentImplementationMgt* contém todas as operações referentes a criação, manipulação e recuperação sobre todos os demais elementos que representam uma arquitetura. Estas operações incluem conexões entre interfaces, mapeamento de interfaces, entre outros.

Os componentes de negócio das ferramentas foram identificados a partir das interfaces de negócio: *ComponentSpecificationMgr* e *ComponentImplementationMgr*. O componente *ComponentSpecificationMgr* faz parte da arquitetura de componentes do ambiente BELLATRIX. O componente de sistema *ArchitectureSystem* utilizou os serviços providos pelo componente *ComponentSpecificationMgr*, realizando a ligação entre as ferramentas para construção de arquiteturas de componentes e os editores e ferramentas relacionados a especificação de componentes existentes no ambiente BELLATRIX. A Figura 4.12 apresenta a arquitetura interna de componentes da infra-estrutura de software para construção de arquiteturas de componentes. As interfaces *ArchitectureSystemProvidedInterfaces* e *ImplementationSystemProvidedInterfaces* representam o conjunto de interfaces providas disponibilizadas pelos componentes de sistema *ArchitectureSystem* e *ImplementationSystem*, respectivamente. Esta forma de notação foi utilizada para facilitar a exibição da arquitetura de componentes da infra-estrutura de software deste trabalho. Os conectores também não foram exibidos na figura para facilitar a sua compreensão. Para cada conexão entre interfaces providas e requeridas apresentada na figura existe um conector que a materializa.

Além da arquitetura de componentes apresentada na Figura 4.12, a camada de apresentação da infra-estrutura de software segue um modelo em estilo MVC (do inglês: *Model View Control*) [11]. A principal imposição para a utilização deste modelo foi a adoção da tecnologia GEF (*Graphical Editing Framework*) [22], um projeto que facilita a construção de editores gráficos no ambiente Eclipse. A Figura 4.13 apresenta uma visão geral do estilo MVC adotado para a infra-estrutura de software. O **Modelo** é responsável por manipular os dados e as operações da aplicação, sem se preocupar com a maneira como estes dados e operações serão apresentados para o usuário. A parte de exibição dos dados do modelo é responsabilidade da **Visão**. Diferentes visões podem apresentar os dados de um modelo de diferentes maneiras. Estas visões devem se registrar no modelo para que possam ser notificadas quando o modelo sofrer alguma alteração. O **Controle**, ou

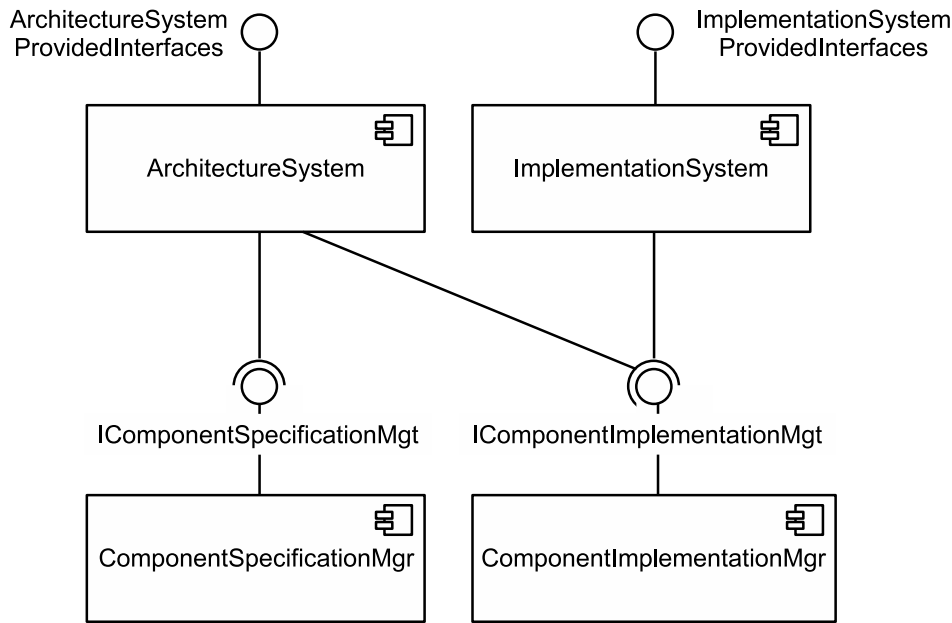


Figura 4.12: Arquitetura de componentes da infra-estrutura de software.

controlador, interpreta as interações do usuário com as visões, alterando o modelo quando necessário.

A arquitetura de componentes, apresentada na Figura 4.12, atua como o **modelo** neste estilo MVC. Os componentes de sistema da ferramenta disponibilizam as operações necessárias para a manipulação das descrições de arquiteturas, porém sem se preocupar com a forma como são exibidas. Os elementos de **visão** e de **controle** realizam a exibição das arquiteturas e as interações com os usuários.

4.6.2 Modelo de Componente Utilizado

Apesar das ferramentas para construção de arquiteturas de componentes terem sido desenvolvidas como um conjunto de *plug-ins* para o ambiente Eclipse, apenas a utilização destes módulos (*plug-ins*) não é suficiente para representar as características desejáveis para um componente de software [30]. Entre as limitações da estrutura de *plug-ins* podemos ressaltar: (1) ausência de formas para declarar componentes requeridos como obrigatórios; e (2) ausência de conectores explícitos. Ainda, não é possível definir interfaces requeridas, somente pacotes ou *plug-ins* requeridos.

Uma adaptação do modelo COSMOS foi desenvolvida para ser utilizada como um conjunto de *plug-ins* do Eclipse para compensar estas deficiências apresentadas pelo modelo

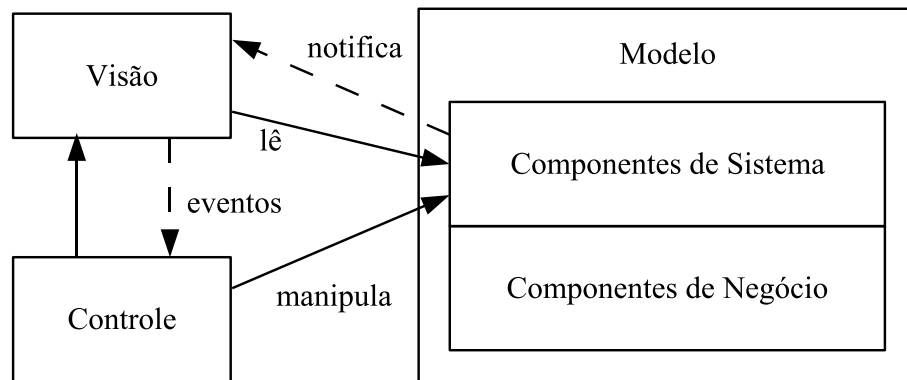


Figura 4.13: Estilo arquitetural MVC utilizado pela infra-estrutura de software.

de *plug-ins*. A estrutura de pacotes, classes e interfaces do modelo COSMOS foi mantida. Um conjunto de artifícios foi utilizado para possibilitar a instanciação e conexão dos componentes e conectores. Uma explicação detalhada do modelo de componentes que combina o modelo COSMOS e os *plug-ins* do Eclipse pode ser encontrada em [52].

4.6.3 Implementação do Metamodelo

O modelo sobre o qual a infra-estrutura de software foi desenvolvida seguiu o metamodelo conceitual de arquitetura de software e DBC apresentado no Capítulo 3. A tecnologia EMF (*Eclipse Modeling Framework*) [21], um subprojeto do projeto Eclipse, foi utilizada para facilitar a implementação do modelo e garantir sua conformidade com o metamodelo conceitual. A tecnologia EMF permite que sejam criadas implementações em linguagem Java que representem determinados modelos. Estas implementações possibilitam a criação, manipulação e persistência de instâncias destes modelos. Por padrão, a tecnologia EMF realiza a persistência destas instâncias de modelos em arquivos em formato XML [55].

Para a geração automática deste modelo em linguagem Java o EMF disponibiliza algumas opções para a especificação do modelo. Entre as possíveis opções está a ferramenta EclipseUML [39], que é uma ferramenta para modelagem de sistemas orientados-a-objetos utilizando a linguagem UML [7]. O EclipseUML permite que sejam criados diagramas de classes que representam modelos para a linguagem EMF.

A utilização da ferramenta EclipseUML e da tecnologia EMF aumentam a conformidade do modelo utilizado pela infra-estrutura de software deste trabalho e o metamodelo conceitual. Desta maneira, todos os elementos descritos no metamodelo conceitual são tratados e manipulados pelas ferramentas para construção de arquiteturas de software

baseadas em componentes, com exceção do modelo que descreve os elementos referentes a estilos arquiteturais, que será discutido na próxima seção.

Além dos elementos contidos no metamodelo conceitual, outras informações são necessárias para criar e persistir os diagramas que representam uma especificação de arquitetura. Informações como posicionamento de um componente na tela ou o tamanho de exibição de um componente precisam ser manipuladas e persistidas durante uma descrição de uma arquitetura de componentes. Para representar estes elementos de forma a não interferir nos elementos centrais de uma especificação de arquitetura, eles foram separados e colocados em pacotes diferentes do modelo central da infra-estrutura de software. As vantagens obtidas nesta abordagem são: (1) maior clareza no modelo gerado, permitindo que outras ferramentas possam ler informações referentes à descrição da arquitetura e (2) maior flexibilidade na manipulação do modelo, pois diferentes diagramas podem exibir o modelo de diferentes maneiras.

Implementação do Metamodelo de Estilos Arquiteturais

A parte do metamodelo conceitual de arquitetura de software relacionada a estilos e propriedades arquiteturais não foi implementada utilizando a tecnologia EMF, mas sim a biblioteca AcmeLib. Através da biblioteca é possível manipular estilos arquiteturais e propriedades arquiteturais, que podem ser persistidos em linguagem Acme. A biblioteca AcmeLib foi utilizada para a manipulação destes elementos pois:

1. Os elementos que representam estilos e propriedades arquiteturais no metamodelo conceitual de arquitetura de software são semelhantes aos elementos existentes na linguagem Acme, que são disponibilizados na biblioteca AcmeLib;
2. O Editor de estilos arquiteturais da infra-estrutura de software desta dissertação é o AcmeStudio, que cria estilos arquiteturais em linguagem Acme;
3. A biblioteca AcmeLib possui um verificador que permite analisar se as restrições impostas pelo estilo estão sendo respeitadas pela arquitetura. Desta maneira basta traduzir a arquitetura de componentes descrita pelo editor de arquitetura para a linguagem Acme e utilizar o verificador de restrições existente na biblioteca AcmeLib.

Um conjunto de elementos adicionais foram inseridos no modelo EMF para possibilitar a integração com a biblioteca AcmeLib. Estes elementos foram utilizados como uma ponte entre os elementos do modelo EMF e os elementos da biblioteca AcmeLib, permitindo descrições mais completas para as arquiteturas de componentes.

4.6.4 Tecnologia Utilizada para a Geração de Código em COSMOS

Para a geração da implementação das arquiteturas em modelo COSMOS foi utilizada a tecnologia JET (*Java Emitter Templates*), uma ferramenta que auxilia na geração de código. O JET é disponibilizado juntamente com o EMF, permitindo uma maior interação entre o modelo manipulado em EMF e a geração de código.

Baseado na linguagem Java, o JET utiliza a idéia de *templates* para a geração de código. Para cada classe, interface e tipo de dado existente no modelo COSMOS foi criado um *template* contendo toda a estrutura de código necessária. Para cada *template* é passado um conjunto de informações que possibilita a geração do código para o elemento representado pelo *template*. Como o JET é baseado em Java, os objetos que descrevem uma arquitetura, modelados em EMF, são passados para os *templates* em JET para que as informações sejam preenchidas de acordo com a arquitetura em questão.

Por utilizar uma estrutura de *templates*, geradores de código para outras linguagens e modelos de implementação de componentes podem ser facilmente adicionados à ferramenta, bastando que outros *templates* sejam criados e utilizados. Estes novos *templates* podem ser adicionados à ferramenta como um conjunto de *plug-ins* do Eclipse. Estes geradores de código podem ser construídos seguindo a mesma estrutura utilizada em JET para a geração de código em modelo COSMOS.

4.7 Resumo

Este capítulo apresentou todas as ferramentas que compõem a infra-estrutura de software para construção de arquiteturas de software baseadas em componentes. As ferramentas foram desenvolvidas a partir de requisitos para modelagem e construção de arquiteturas de componentes e dos casos de uso identificados. As ferramentas são divididas em um conjunto de ferramentas para modelagem e verificação de arquiteturas de componentes e um conjunto para construção destas arquiteturas. A arquitetura de componentes da infra-estrutura de software também foi apresentada, juntamente o modelo de componentes utilizado e as principais tecnologias envolvidas. No próximo capítulo são apresentados os estudos de caso que avaliaram as funcionalidades e a usabilidade das ferramentas.

Capítulo 5

Estudos de Casos de Uso da Infra-estrutura de Software Proposta

Este capítulo apresenta três estudos de caso realizados para avaliar a infra-estrutura de software para construção de arquiteturas de software baseadas em componentes. Os estudos de caso abrangeram as principais funcionalidades disponibilizadas pela infra-estrutura de software deste trabalho, compreendendo diferentes necessidades de uso das ferramentas.

A Seção 5.1 apresenta um estudo de caso realizado para avaliar as ferramentas para modelagem e verificação de arquiteturas de componentes. Este estudo de caso utilizou uma especificação de um sistema financeiro para a sua realização. A especificação do sistema foi passada para quatro voluntários do Instituto de Computação da Unicamp que utilizaram a infra-estrutura de software deste trabalho para modelar e verificar a arquitetura de componentes do sistema financeiro.

A Seção 5.2 apresenta um estudo de caso realizado no contexto de um ambiente de apoio a automação de testes para sistemas DBC. Este estudo foi realizado a partir de uma necessidade real de um grupo de pesquisas em testes do Instituto de Computação da Unicamp em gerar, de forma automática, o código em modelo COSMOS para um ambiente de testes, a partir de sua especificação arquitetural.

A Seção 5.3 apresenta um estudo de caso realizado no contexto de um sistema de controle de matrículas. Este sistema foi implementado para o Centro de Computação da Unicamp (CCUEC) com o objetivo de ser um protótipo para avaliação do modelo de componentes COSMOS. Foram avaliadas as funcionalidades para modelagens de arquiteturas de componentes e para geração de código.

5.1 Estudo de Caso: Sistema Financeiro

5.1.1 Objetivo

Este estudo de caso foi realizado com o objetivo de avaliar se as funcionalidades disponibilizadas pela infra-estrutura de software deste trabalho são suficientes para modelar arquiteturas de componentes e realizar verificações sobre estas arquiteturas. As ferramentas avaliadas foram: editor de estilos arquiteturais (Seção 4.4.2), editor de arquiteturas (Seção 4.4.1) e verificador de arquiteturas (Seção 4.4.3). Neste estudo de caso não foi abordada a materialização da arquitetura através de uma implementação, uma vez que esta funcionalidade será avaliada nos estudos de caso apresentados na Seção 5.2 e na Seção 5.3. O estudo de caso buscou ainda encontrar possíveis defeitos e deficiências da ferramenta, assim como identificar novas funcionalidades a serem acrescentadas e realizar avaliações sobre sua usabilidade.

5.1.2 Descrição do Estudo de Caso

Este estudo de caso foi realizado em conjunto com o estudo de caso que avaliou o ambiente BELLATRIX [52]. O planejamento e execução dos estudos de caso foram realizados em conjunto e seus resultados foram avaliados de forma separada. O estudo de caso foi realizado durante o mês de maio de 2006, levando aproximadamente 2 semanas para ser concluído.

Para que todas ferramentas relacionados a modelagem e verificação de arquiteturas de componentes pudessem ser utilizadas e avaliadas no contexto deste estudo de caso era necessário uma especificação de arquitetura que seguisse pelo menos um estilo arquitetural. A especificação a ser utilizada no estudo deveria ainda representar uma arquitetura de componentes de um sistema real, de preferência no contexto de uma empresa.

A especificação escolhida para o estudo de caso foi a de um sistema financeiro, mais especificamente, o controle de uso de talões de cheque. Esta é uma especificação de um sistema real de uma empresa de médio porte situada na cidade de São Paulo, que já foi apresentada e utilizada no contexto de outros trabalhos [10, 41]. Esta especificação utiliza também dois estilos arquiteturais em sua arquitetura de componentes. A descrição e especificação do sistema, incluindo seus estilos arquiteturais, são apresentadas nas Seções 5.1.3 e 5.1.4, respectivamente.

A execução do estudo de caso foi realizada pelo autor deste trabalho e por mais quatro voluntários, todos alunos de pós-graduação do Instituto de Computação da Unicamp. Os voluntários foram responsáveis pela avaliação das funcionalidades e usabilidade da ferramenta para a descrição e verificação da arquitetura de componentes. Os voluntários nunca haviam utilizado anteriormente a ferramenta e todos possuíam conhecimento em

arquitetura de software para sistemas DBC. A especificação do sistema financeiro foi passada para os voluntários para que eles pudessem utilizá-la durante o estudo de caso. Os voluntários foram identificados pelas letras de A a D e identificados da seguinte forma:

- Voluntário A: é um aluno de mestrado e possui experiência profissional em desenvolvimento de software;
- Voluntário B: é um aluno de mestrado, sem experiência profissional;
- Voluntário C: é um aluno de doutorado que atua profissionalmente em uma empresa de desenvolvimento de software;
- Voluntário D: é um aluno de doutorado que possui alguma experiência profissional (como estagiário).

Um questionário foi elaborado para que os voluntários pudessem avaliar a ferramenta no contexto deste estudo de caso. Ele foi construído com base em testes de usabilidade [42] e procurou colher informações sobre o perfil do usuário, o uso do ambiente em termos das funcionalidades presentes, a facilidade de aprendizado, sua produtividade e seu sistema de ajuda. O questionário possibilitou ainda a identificação de problemas e sugestões de melhoria. O Apêndice C apresenta o questionário elaborado. Como este estudo de caso não avaliou a materialização da arquitetura através de uma implementação, a Seção C.3 do questionário não foi respondida pelos voluntários.

O estudo de caso foi realizado no Laboratório de Sistemas Distribuídos (LSD) do Instituto de Computação da Unicamp. A instalação das ferramentas, juntamente com as orientações necessárias para o estudo de caso, foram disponibilizadas em uma área pública para que os voluntários pudessem instalar e realizar a especificação da arquitetura de componentes com o mínimo auxílio externo.

5.1.3 Descrição do Sistema

O sistema utilizado neste estudo de caso é um sistema de controle de cheques que pode ser utilizado por instituições financeiras. Os requisitos básicos identificados para o sistema são:

1. **Requisição de talões de cheques:** O cliente pode solicitar talões de cheques, dirigindo-se a uma agência bancária ou utilizando canais de relacionamento, tais como telefone ou internet.
2. **Entrega de talões de cheques:** O operador pode registrar a retirada física pelo cliente de talões de cheque solicitados anteriormente.

3. **Sustação de cheques:** O cliente pode cancelar um determinado número de cheques, mediante a informação do motivo para a sustação (por exemplo: roubo do malote, perda, roubo, entre outros) e da situação dos cheques (por exemplo: em branco, preenchido e datado, preenchido e não datado).
4. **Captura de cheque para compensação:** A captura de cheque é feita durante um depósito de cheque. Esses cheques são capturados e processados para a compensação. A operação de captura é identificada por um número identificador e registra a data do movimento.
5. **Cancelamento do contrato da conta:** A qualquer momento, o cliente pode perder o crédito de sua conta. Dessa forma, o seu contrato deve poder ser cancelado.
6. **Cadastramento de limite adicional:** Dependendo da necessidade do cliente e das suas condições de crédito, o cliente pode receber um limite adicional que poderá ser utilizado por ele.

As funcionalidades “cancelamento do contrato da conta” e “sustação de cheques” devem estar disponíveis o máximo de tempo possível. Esse atributo de qualidade imposto pelas regras de negócio tem o objetivo de evitar a utilização de um crédito que já não esteja disponível. Essa utilização pode ocorrer por parte do cliente, isto é, utilização indevida de seu limite de crédito, ou por parte de terceiros, isto é, o resgate de cheques cancelados. Por essa razão, assume-se que essas funcionalidades possuem requisitos críticos de disponibilidade. A Figura 5.1 apresenta o diagrama contendo os casos de uso identificados para o sistema a partir das funcionalidades apresentadas acima.

5.1.4 Especificação do Sistema

Esta seção descreve a especificação do sistema de cheques que foi produzida seguindo o processo MDCE+ [10] - um processo para desenvolvimento de sistemas DBC confiáveis baseado no processo UML Components [12] (processos para DBC foram apresentados na Seção 2.2.2). Esta especificação já havia sido elaborada em outro trabalho [10] e foi apenas utilizada neste estudo de caso. Além das restrições impostas pelo UML Components, a arquitetura adotada para o estudo de caso segue restrições impostas pelo ambiente da empresa, tais como componentes utilitários e a maneira de acesso aos dados. A Seção 5.1.4 apresenta os estilos arquiteturais utilizados, a Seção 5.1.4 apresenta as especificações dos componentes identificados para o sistema e a Seção 5.1.4 apresenta a arquitetura de componentes final do sistema de cheques.

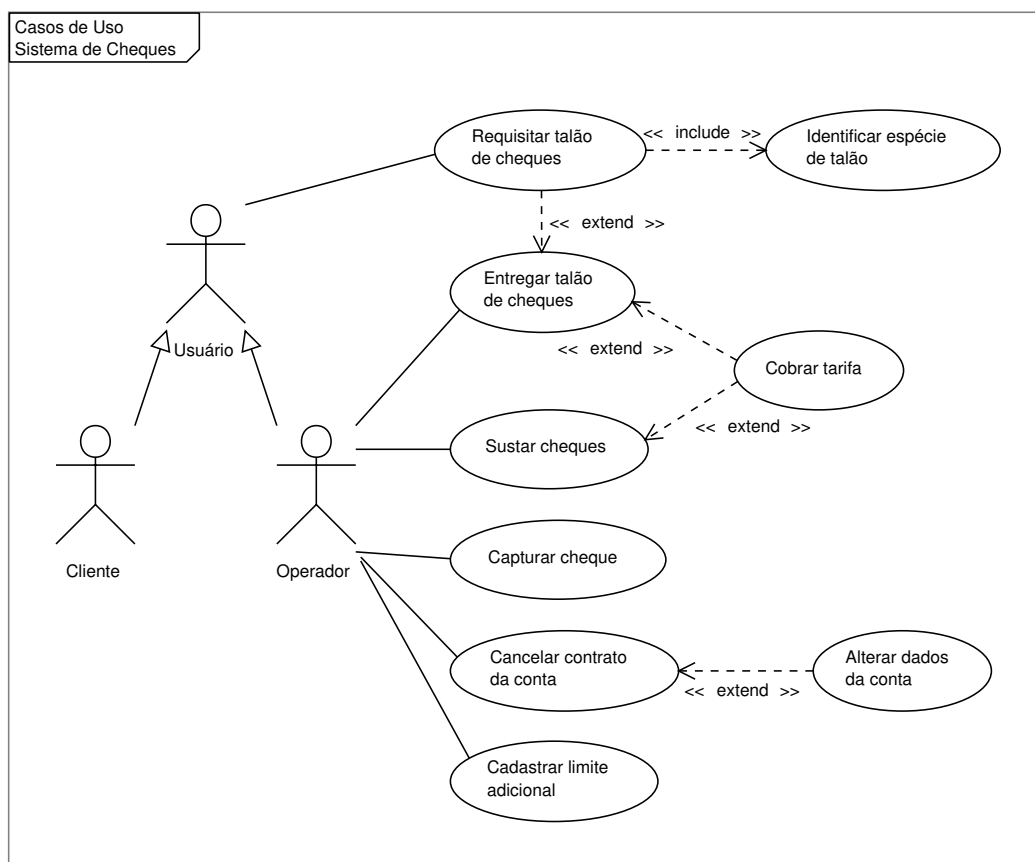


Figura 5.1: Casos de uso do sistema de cheques.

Estilos Arquiteturais do Sistema

O estilo arquitetural imposto para o sistema de cheques segue um estilo em camadas “relaxado”, possuindo três camadas horizontais e uma camada vertical, conforme apresentado na Figura 5.2. A camada de **Dados** é responsável por realizar a intermediação com as bases de dados da aplicação e as camadas de **Sistema** e **Negócio** possuem responsabilidades análogas às camadas de mesmo nome no processo UML Components. A camada **Utilitário** oferece serviços gerais e comuns às camadas de sistema, negócio e dados. Exemplos de serviços desta camada incluem validadores e conversores. As setas na Figura 5.2 representam o sentido com que uma camada pode chamar o serviço de outra camada. Por exemplo, a camada de sistema pode utilizar serviços disponibilizados pela camada de negócio, porém o contrário não é permitido.

Além das camadas impostas para a arquitetura de componentes do sistema, as seguin-

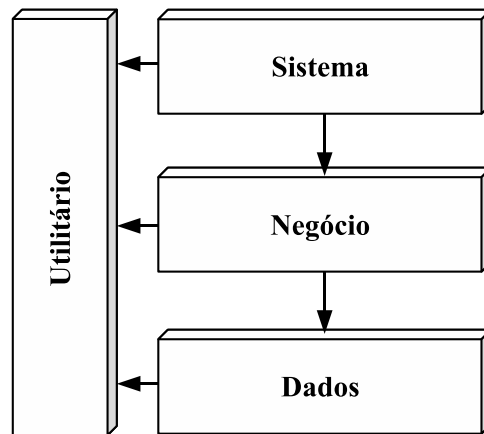


Figura 5.2: Estilo arquitetural em camadas do sistema de cheques.

tes restrições foram identificadas:

- Todos os componentes do sistema devem estar em uma das camadas do estilo arquitetural;
- Caso um componente de uma camada dependa de serviços de outras camadas, apenas um conector pode ser criado para cada camada a qual este componente se comunica. Por exemplo, caso um componente da camada de sistema dependa de dois componentes da camada de negócio, apenas um conector poderá ser criado para conectar o componente da camada de sistema aos dois componentes da camada de negócio;
- As dependências entre os componentes devem respeitar os sentidos de comunicação impostos para as camadas. Por exemplo, um componente da camada de negócios não pode depender de um componente da camada de sistema. As dependências são explicitadas através das interfaces providas e requeridas;
- Componentes na arquitetura podem ser identificados como críticos e conectores podem ser identificados como tolerantes a falhas. Para cada componente crítico na arquitetura deve existir um conector tolerante a falhas conectado a este componente.

Além do estilo em camadas imposto pela arquitetura, os componentes podem possuir uma arquitetura interna seguindo o modelo de componente ideal tolerante a falhas. Neste modelo, a arquitetura interna do componente é constituída de um componente normal, que implementa o comportamento normal, e um ou mais componentes tratadores, que implementam os tratamentos para as exceções lançadas pelo componente normal.

Especificação dos Componentes

Os componentes de sistema foram identificados a partir dos casos de uso do sistema de cheques. Foram identificados os componentes `operacoesConta`, `operacoesTalao`, `operacoesChequeCapturado` e `operacoesChequeSustado`. A Tabela 5.1 apresenta as interfaces de cada componente de sistema e os casos de uso associados a cada uma das interfaces.

Tabela 5.1: Interfaces e casos de uso para os componentes de sistema.

Componente	Interface	Caso de Uso
<code>operacoesTalao</code>	<code>ISBObtencaoTalaoMgr</code>	- Requisitar talão de cheque - Entregar talão de cheque
<code>operacoesChequeSustado</code>	<code>ISBSustarChequeMgr</code>	- Sustar cheques
<code>operacoesChequeCapturado</code>	<code>ISBCapturaChequeMgr</code>	- Capturar cheques
<code>operacoesConta</code>	<code>ISBCancelaContratoContaMgr</code>	- Cancelar contrato da conta
	<code>ISBCadLimiteAdcMgr</code>	- Cadastrar limite adicional

A identificação dos componentes de negócio levou em consideração o reuso de componentes existentes na empresa. Os componentes identificados foram: `gerenciamentoConta`, `gerenciamentoColigada`, `gerenciamentoMovimento` e `gerenciamentoControleAgencia`. Estes componentes são responsáveis pelo gerenciamento das contas dos clientes, das coligadas - que são agências do mesmo banco ou de outros bancos, do movimento bancário e das agências bancárias. Foram ainda identificados os componentes `componentePersistencia` e `componenteMySQL` na camada de dados e o componente `componenteUtilitario` na camada de utilitário.

Arquitetura de Componentes do Sistema

Os componentes do sistema de cheques, identificados e apresentados na Seção 5.1.4, foram organizados em uma arquitetura de componentes, apresentada na Figura 5.3. Na figura é possível identificar em qual camada cada componente está inserido. Por motivo de clareza, a figura abstrai os conectores, que foram criados conforme a seguinte estratégia: para cada componente foi criado um conector para cada camada diferente com que ele se comunica.

Considerando que as funcionalidades “cancelamento do contrato da conta” e “sustação de cheques” devem ter alta disponibilidade, foram identificados como críticos os compo-

mentes `operacoesConta`, `operacoesChequeSustado` e `gerenciamentoConta`. O componente `gerenciamentoConta` foi assim replicado na arquitetura mostrada na Figura 5.3. Caso alguma falha ocorra no componente `gerenciamentoConta`, os conectores são responsáveis por identificar esta falha e começar a utilizar a réplica do componente. Os componentes `operacoesConta` e `operacoesChequeSustado` seguem o modelo de componente ideal tolerante a falhas. Por este motivo, estes componentes possuem uma arquitetura interna que segue o estilo de componente ideal tolerante a falhas, apresentado na Seção 5.1.4.

5.1.5 Execução do Estudo de Caso

O estudo de caso foi dividido em três partes: especificação dos estilos arquiteturais, descrição da arquitetura de componentes do sistema de cheques e verificação desta arquitetura.

A criação dos estilos arquiteturais através do editor de estilos foi realizada pelo autor deste trabalho. Foram criados dois estilos: **CamadasSistemaCheques** e **ComponenteIdeal**. O estilo **CamadasSistemaCheques** contém a descrição do estilo em camadas utilizado pela arquitetura de componentes. Foram definidos quatro tipos de componentes para este estilo, que são apresentados na Tabela 5.2. Cada um destes tipos possui ainda a propriedade arquitetural **critical**, que pode assumir o valor verdadeiro (true) ou falso (false), indicando se o componente é ou não um componente crítico. O estilo define ainda dois tipos de conectores, apresentados na Tabela 5.3.

Tabela 5.2: Tipos de componentes do estilo em camadas do sistema de cheques.

Componente	Descrição
SystemLayer	Componentes pertencentes a camada de sistema.
BusinessLayer	Componentes pertencentes a camada de negócio.
DataLayer	Componentes pertencentes a camada de dados.
UtilityLayer	Componentes pertencentes a camada de utilitários.

Tabela 5.3: Tipos de conectores do estilo em camadas do sistema de cheques.

Conector	Descrição
InterLayerConn	Conecta componentes de diferentes camadas.
IntraLayerConn	Conecta componentes que pertençam a uma mesma camada.

O estilo **ComponenteIdeal** contém a representação da arquitetura interna de um componente ideal tolerante a falhas. A Tabela 5.4 e a Tabela 5.5 apresentam, respectivamente, os componentes e conectores definidos para o estilo.

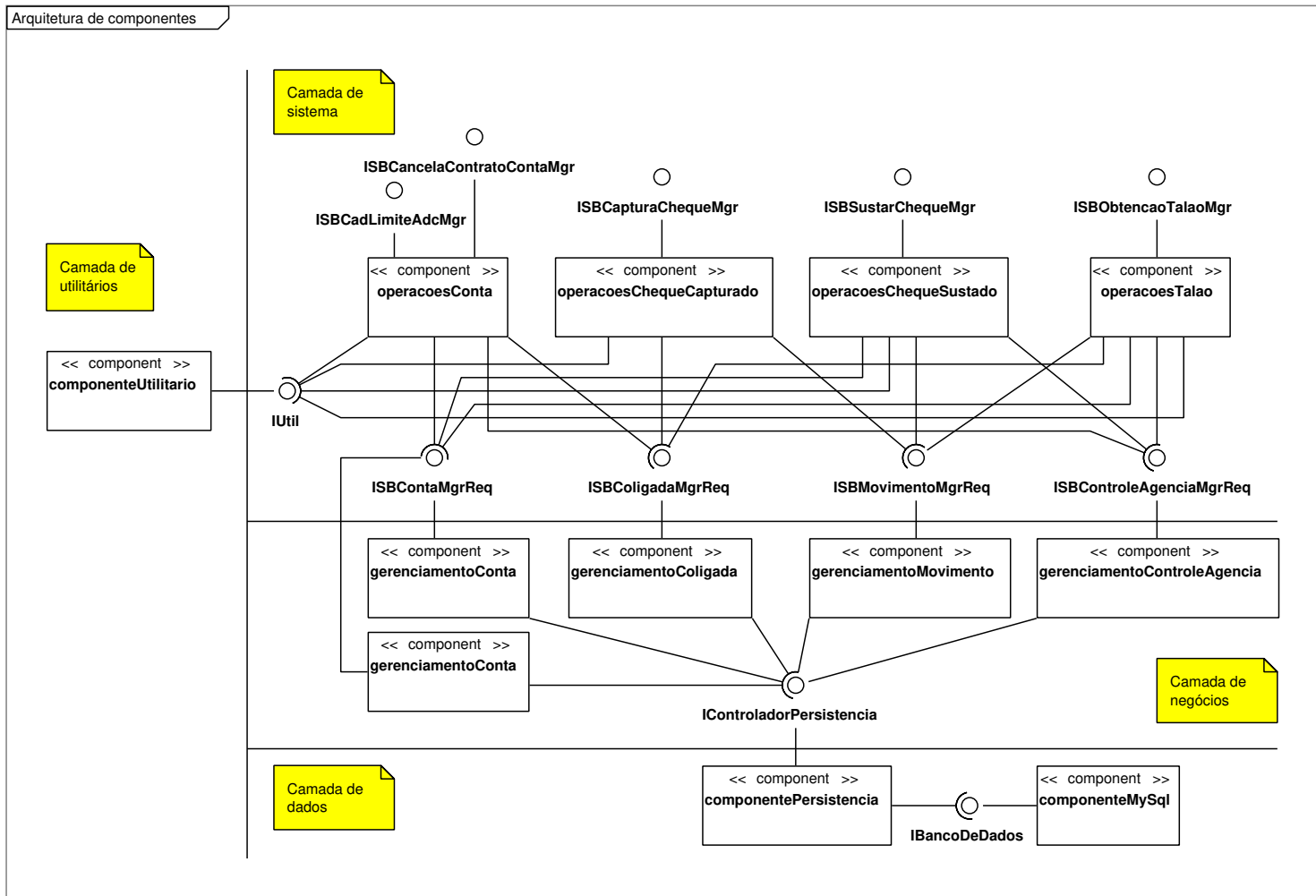


Figura 5.3: Arquitetura de componentes do sistema de cheques.

Tabela 5.4: Tipos de componentes do estilo do componente ideal do sistema de cheques.

Componente	Descrição
NormalBehavior	Indica o componente que possui o comportamento normal.
AbnormalBehavior	Indica um componente tratador, responsável pela implementação dos tratamentos para as exceções lançadas pelo componente normal (NormalBehavior).

Tabela 5.5: Tipos dos conectores do estilo do componente ideal do sistema de cheques.

Conector	Descrição
AbnormalBehaviorConn	Conecta o componente normal a um componente tratador.

As estilos arquiteturais foram disponibilizados juntamente com os demais documentos e artefatos necessários para a execução do estudo de caso para que os voluntários pudessem utilizá-los na especificação da arquitetura de componentes do sistema de cheques. Cada voluntário realizou a descrição da arquitetura de componentes utilizando a ferramenta e preencheu o questionário de avaliação. Utilizando a mesma atribuição de letras para cada um dos voluntários, as informações coletadas sobre a maneira de utilização da ferramenta são:

1. **Voluntário A:** utilizou a ferramenta para descrever a arquitetura de componentes do sistema, utilizou os estilo arquiteturais disponibilizados e utilizou o verificador de arquitetura. Ele se auto-avaliou como conhecedor dos conceitos em DBC e arquitetura de software e levou aproximadamente quatro horas para criar a arquitetura de componentes e verificá-la. O voluntário teve alguns problemas na instalação do ambiente e precisou ainda recorrer ao sistema de ajuda duas vezes durante a avaliação, nem sempre encontrando a informação que desejava.
2. **Voluntário B:** utilizou a ferramenta para descrever a arquitetura de componentes do sistema, utilizou os estilo arquiteturais disponibilizados e utilizou o verificador de arquitetura. Ele se auto-avaliou como conhecedor dos conceitos em DBC e arquitetura de software e levou aproximadamente três horas e meia para criar a arquitetura de componentes e verificá-la. O voluntário não utilizou o sistema de ajuda.
3. **Voluntário C:** utilizou a ferramenta para descrever a arquitetura de componentes, atribuir os estilos, verificar a arquitetura e gerar uma configuração de componentes. Ele se auto-avaliou como possuindo o conhecimento mínimo para a utilização da ferramenta e levou aproximadamente três horas para executar o estudo de caso. O voluntário recorreu ao sistema de ajuda várias vezes, nem sempre encontrando a informação desejada.

4. **Voluntário D:** utilizou a ferramenta para descrever a arquitetura de componentes do sistema e criar sua configuração. Ele se auto-avaliou como tendo pleno domínio sobre os conceitos envolvidos em arquitetura de software e DBC, levando aproximadamente uma hora e quarenta minutos para descrever a arquitetura de componentes e criar sua configuração. O voluntário utilizou o sistema de ajuda duas vezes, conseguindo encontrar as informações que necessitava.

5.1.6 Avaliação e Resultados Obtidos

Avaliação Geral

No geral, os voluntários envolvidos na avaliação das ferramentas julgaram satisfatórias as funcionalidades disponibilizadas e a sua usabilidade. A Tabela 5.6 apresenta as notas dadas por cada um dos voluntários ao final da avaliação, sendo que cada quesito poderia ser avaliado com uma nota de 1 (péssimo) a 5 (ótimo). As principais críticas negativas sobre as ferramentas se concentraram no modo de exibição da arquitetura de componentes modelada. Foram apontadas dificuldades para organizar os elementos no editor de arquiteturas e visualizar a arquitetura de componentes descrita. Apesar disto, foi relatado que o controle de “zoom” auxiliou a lidar com a complexidade de exibir uma arquitetura completa e manipulá-la.

Tabela 5.6: Notas atribuídas pelos voluntários na avaliação das ferramentas.

Quesito	usuário A	usuário B	usuário C	usuário D
Funcionalidades presentes	5	5	4	5
Facilidade de aprendizado por exploração	3	4	3	4
Produtividade e facilidade de uso	4	3	3	4
Nota geral	4	4	4	4

De acordo com as considerações e sugestões apontadas nas respostas dos questionários, os seguintes pontos de melhoria foram identificados:

Robustez : durante a utilização das ferramentas foram encontrados seis erros de código (*bugs*) que ocasionaram em comportamentos inesperados da ferramenta. Destes erros, apenas um ocasionou em comportamentos inaceitáveis, tornando a ferramenta imprópria para utilização. Este erro foi encontrado apenas por um dos voluntários e uma nova versão foi disponibilizada para que o estudo de caso pudesse ser completado.

Funcionalidades : foram identificadas nove novas funcionalidades para a ferramenta de descrição de arquiteturas de componentes. A funcionalidade mais requisitada pelos voluntários foi a melhoria na exibição da arquitetura, principalmente na forma de apresentação dos conectores. Foi identificado que os conectores ocupam uma parte grande na arquitetura de componentes, podendo ser exibidos com ou outro tipo de notação ou até mesmo outra cor.

Usabilidade : foram identificadas três melhorias de usabilidade para o ambiente: (i) inserção de teclas de atalho, (ii) possibilidade de arrastar os artefatos que representam componentes ou conectores diretamente para dentro de uma arquitetura e (ii) mudança de um texto em um dos editores.

Consolidação dos Resultados

As principais vantagens apontadas pelos voluntários para a infra-estrutura de software deste trabalho foram: (i) o conjunto de funcionalidades disponibilizado pelas ferramentas - possibilitando a modelagem de arquiteturas de componentes respeitando os principais conceitos existentes nas abordagens de arquitetura de software e DBC - e (ii) facilidade para realizar análises sobre a arquitetura de componentes. A principal desvantagem encontrada durante o estudo de caso foi a de organizar e visualizar uma modelagem de uma arquitetura de componentes através do editor de arquiteturas.

5.2 Estudo de Caso: Um Ambiente para Testes de Componentes de Software

5.2.1 Objetivo

O objetivo deste estudo de caso é realizar uma avaliação complementar sobre a usabilidade e as funcionalidades básicas da infra-estrutura de software para construção de arquiteturas de componentes e avaliar suas funcionalidades ligadas à construção destas arquiteturas. Foram avaliados o editor de arquiteturas (Seção 4.4.1), o editor de configurações (Seção 4.5.1) e o gerador de código (Seção 4.5.2). Este estudo de caso visou também encontrar possíveis defeitos na ferramenta e colher sugestões de novas funcionalidades.

5.2.2 Descrição do Estudo de Caso

Este estudo de caso foi realizado a partir de uma necessidade real de um grupo de pesquisas em testes de software do Instituto de Computação da Unicamp. O ambiente de testes foi internamente projetado baseado em componentes de software, seguindo o processo UML Components [12]. Os responsáveis por realizar a especificação e a implementação do ambiente de testes foram dois alunos de mestrado do grupo de pesquisa em testes. Dado que a especificação da arquitetura de componentes da aplicação alvo estava definida e existia a necessidade de implementá-la, os responsáveis pelo ambiente decidiram utilizar a ferramenta de descrição de arquiteturas, juntamente com o ambiente BELLATRIX, para especificar o sistema e obter uma implementação seguindo o modelo COSMOS. Como o ambiente de testes possui diversas ferramentas e nem todas ferramentas serão implementadas em uma primeira versão, apenas duas ferramentas foram selecionadas para serem utilizadas neste estudo de caso. Ainda, ficou combinado que toda a arquitetura modelada teria seu código gerado, porém apenas a implementação de alguns componentes seria utilizada para avaliar a qualidade do código gerado. O estudo de caso aconteceu durante o mês de dezembro de 2006 e durou aproximadamente suas semanas.

Como o sistema não possuía estilos arquiteturais que fossem interessantes de serem verificados e dado que já existia uma especificação da arquitetura de componentes do sistema, a preparação do estudo de caso se limitou a disponibilizar uma versão instalável da infra-estrutura de software deste trabalho em um local público e a preparar e disponibilizar o questionário de avaliação. O questionário elaborado se encontra no Apêndice C deste documento, porém a Seção C.2 não foi disponibilizada para a avaliação deste estudo de caso pois não seriam realizadas verificações sobre a arquitetura.

Apesar de um conjunto de alunos do grupo de testes terem participado da elaboração dos requisitos do sistema, do projeto arquitetural e das especificações de componentes,

apenas um aluno ficou responsável por realizar a modelagem da arquitetura de componentes na infra-estrutura de software deste trabalho, gerar o código em modelo COSMOS e implementar alguns dos componentes da arquitetura. Este aluno está em fase final de seu projeto de mestrado no Instituto de Computação da Unicamp e possui experiência profissional de aproximadamente um ano como estagiário.

5.2.3 Descrição do Sistema

O ambiente para testes de componentes de software é composto por um conjunto de ferramentas. No contexto deste estudo de caso, foram abordadas apenas duas ferramentas existentes no ambiente: a **Ferramenta de Geração de Casos de Testes** e a **Ferramenta de Seleção de Testes de Regressão**. A Figura 5.4 apresenta uma visão geral destas duas ferramentas, que tem como principal objetivo automatizar a geração e manipulação de casos de teste.

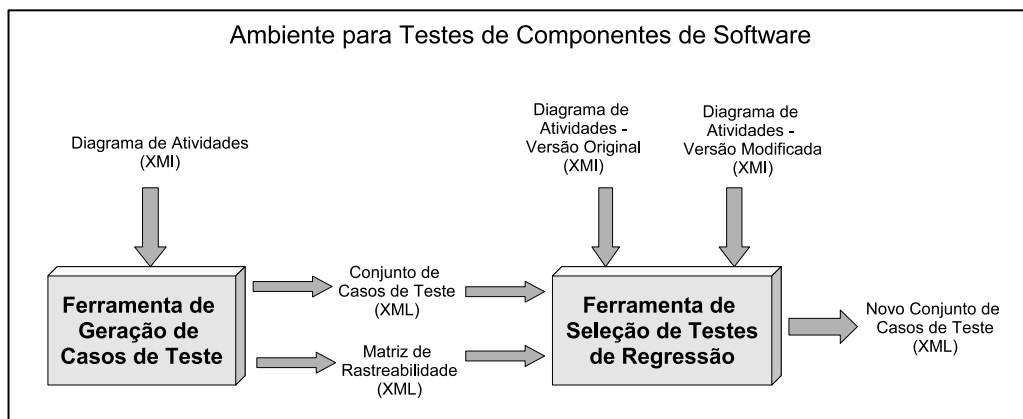


Figura 5.4: Visão geral do ambiente para testes de componentes de software.

A ferramenta de geração de casos de teste recebe como entrada um diagrama de atividades em formato XMI contendo os fluxos de chamadas de operações que podem ser realizados durante a execução de um determinado conjunto de casos de uso. A partir deste diagrama de atividades o usuário informa quais os fluxos ele pretende utilizar para gerar casos de teste, ou seja, quais os critérios de cobertura. A partir deste diagrama de atividades e do critério de cobertura é gerado um conjunto de casos de teste. Juntamente com este conjunto de casos de teste é gerada uma matriz de rastreabilidade, que mapeia os elementos gerados nos casos de teste para os elementos contidos no diagrama de atividades.

A ferramenta de seleção de testes de regressão auxilia na geração de um novo conjunto de casos de testes quando o sistema sofre alguma manutenção. Este novo conjunto

pretende identificar quais casos de teste devem ser utilizados levando em consideração as manutenções realizadas. Esta ferramenta recebe como entrada os diagramas de atividades - contendo a especificação original e a especificação após a manutenção ocorrida, o conjunto de casos de testes original e sua matriz de rastreabilidade. A partir destas informações e das opções do usuário o novo conjunto de casos de teste é criado.

A Figura 5.5 apresenta o diagrama de casos de uso do ambiente. O usuário interage com o sistema através dos casos de uso **Criar casos de teste**, **Definir critério de cobertura**, **Editar casos de teste** e **Analisar modificações para teste de regressão**.

Além dos casos de uso do sistema, um modelo conceitual de negócios foi criado para o ambiente. Este modelo conceitual pode ser encontrado no relatório técnico “Um ambiente para Testes de Componentes”, a ser publicado no instituto de computação da Unicamp. A próxima seção apresenta os componentes identificados para o ambiente e a sua arquitetura interna.

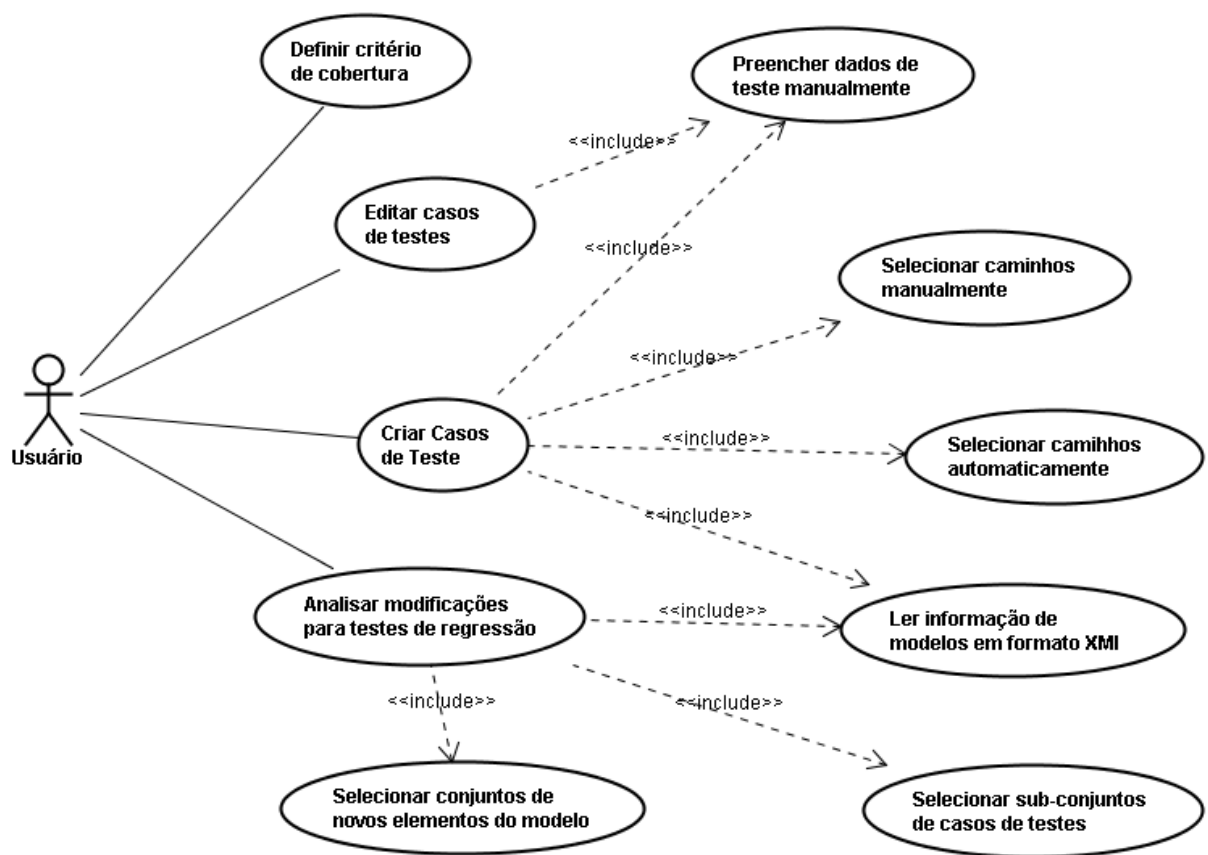


Figura 5.5: Casos de uso do ambiente para testes de componentes de software.

5.2.4 Especificação do Sistema

O ambiente de testes teve seu projeto arquitetural baseado em componentes de software, seguindo o processo de desenvolvimento UML Components. Conforme sugerido pelo processo, o sistema seguiu o estilo em camadas, sendo dividido nas camadas de sistema e de negócio. As interfaces de negócio foram identificadas e especificadas a partir dos elementos contidos no modelo conceitual de negócio. As interfaces identificadas para a camada de negócio foram: `IDadosTesteMgt`, `IConjuntoTesteMgt`, `IGrafoMgt`, `IArquivoXMIMgt` e `IMatrixRastreabilidadeMgt`. Para cada uma destas interfaces foi criada uma especificação de um componente de negócio.

As interfaces de sistema foram identificadas a partir dos casos de uso do ambiente, levando em consideração as descrições de seus cenários. Para cada caso de uso que possui interação direta com o usuário do ambiente foi criada uma interface de sistema. As interfaces de sistema identificadas e especificadas foram: `ICriarCasosTeste`, `IEditarCasosTeste`, `IDefinirCriterioCobertura` e `SistemaTestesRegressao`. A partir destas interfaces foram criadas as especificações dos componentes de sistema: `SistemaConjuntoCasosTeste` - responsável por criar e manipular os casos de teste - e `SistemaTestesRegressao` - responsável pela análise dos testes para regressão.

A partir das especificações das interfaces, das identificações dos componentes e da análise das interações entre estes componentes foi definida a arquitetura de componentes do ambiente, apresentada na Figura 5.6. A arquitetura é composta por dois componentes de sistema e cinco componentes de negócio. Para cada conexão entre interfaces da arquitetura existe um conector responsável por materializar esta conexão. Os conectores não são exibidos na Figura 5.6 para facilitar a visualização da arquitetura, porém nove conectores foram identificados e especificados.

5.2.5 Execução do Estudo de Caso

A execução do estudo de caso foi realizada em três etapas: (1) especificação da arquitetura apresentada na Seção 5.2.4 utilizando as ferramentas para modelagem de arquiteturas de componentes, (2) geração automática de código em modelo COSMOS para essa arquitetura e (3) preenchimento do questionário de avaliação das ferramentas.

O aluno responsável pela execução do estudo de caso se auto-avaliou como possuindo conhecimentos mínimos em DBC e arquitetura de software para a utilização da infraestrutura deste trabalho. Ainda, o aluno informou ter pouco conhecimento sobre o modelo COSMOS e nunca haver utilizado o modelo em uma implementação prática. Primeiramente foram especificadas as interfaces e os componentes de software da arquitetura com o auxílio dos editores do ambiente BELLATRIX, porém esta fase não foi considerada para a avaliação da infra-estrutura de software para construção de arquiteturas de componentes.

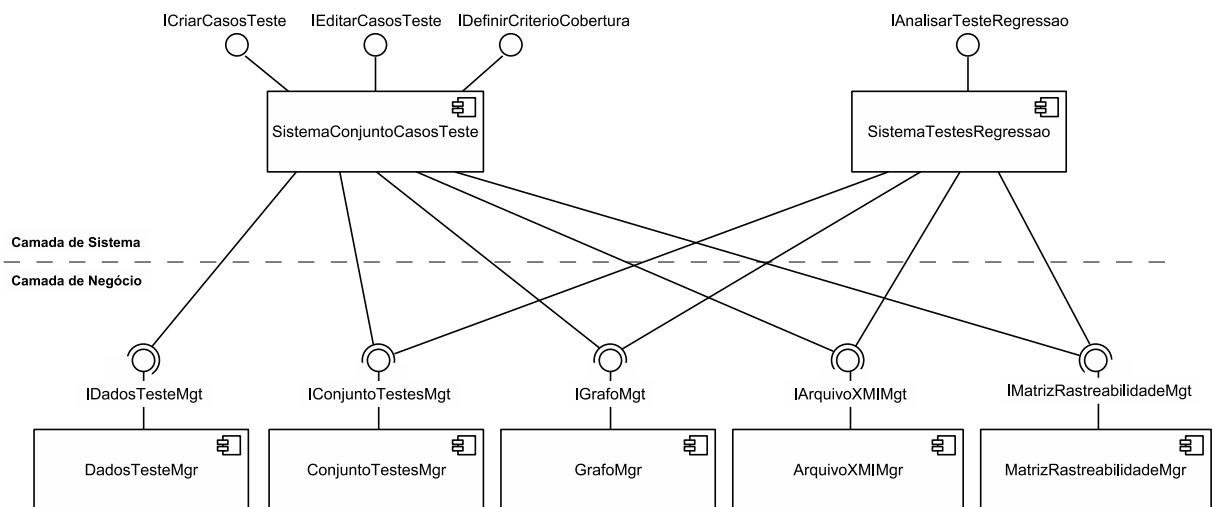


Figura 5.6: Arquitetura de componentes do ambiente de testes.

Durante a especificação da arquitetura de componentes o aluno recorreu ao sistema de ajuda quatro vezes, conseguindo encontrar as informações necessárias em todas as vezes que realizou a pesquisa. Duas das consultas efetuadas ao sistema de ajuda foram referentes a dúvidas conceituais sobre o metamodelo utilizado pelo ambiente (mapeamento entre interfaces internas/externas e definição de uma configuração de componente) e duas dúvidas referentes a maneira como executar determinadas operações. O tempo total para a especificação da arquitetura de componentes e geração de código, considerando o tempo de aprendizado no ambiente, foi de uma hora e cinco minutos. Neste tempo não está contemplado o restante da implementação interna dos componentes.

Portanto, durante a execução do estudo de caso foram geradas: (1) uma arquitetura de componentes, (2) uma configuração de componentes e (3) uma implementação da arquitetura em código seguindo o modelo COSMOS, gerada de forma automática.

5.2.6 Avaliação dos Resultados Obtidos

Avaliação Geral

Dado que o principal objetivo neste estudo de caso era gerar uma implementação da arquitetura de componentes seguindo o modelo COSMOS, a avaliação geral das ferramentas foi positiva, identificando um bom nível de maturidade para a especificação de arquiteturas de componentes e para geração de código. O aluno atribuiu notas de 1 (péssimo) a 5 (ótimo) para avaliar diferentes quesitos da ferramenta. Estas notas e quesitos são

apresentados na Tabela 5.7

Tabela 5.7: Notas atribuídas a ferramenta para especificação do ambiente de testes.

Quesito	Nota
Funcionalidades presentes no ambiente	4
Facilidade de aprendizado por exploração	3
Produtividade e facilidade de uso	5
Nota geral	4

Apesar do aluno nunca ter utilizado outra ferramenta para construção de arquiteturas de componentes, a infra-estrutura de software deste trabalho foi considerada produtiva. O principal ganho apontado foi a geração automática de código em modelo COSMOS. A ferramenta auxiliou que o modelo COSMOS pudesse ser utilizado na implementação do ambiente de testes, mesmo com o baixo conhecimento do modelo relatado pelos envolvidos. As principais dificuldades apontadas foram (i) a compreensão dos conceitos utilizados no ambiente - em especial a definição de uma configuração arquitetural - e (ii) a forma como a arquitetura é exibida nos editores de arquitetura e configuração. Os principais pontos levantados foram:

Robustez da ferramenta : nenhum erro (*bug*) foi identificado durante a utilização das ferramentas. A avaliação apontou que o editor de arquiteturas, o editor de configurações e o gerador de código se mantiveram estáveis durante o estudo de caso.

Funcionalidades : foram apontados cinco pontos de melhoria para o ambiente, sendo quatro pontos relacionados ao modo de exibição e apresentação da arquitetura e um ponto relacionado ao código gerado pela ferramenta de geração de código. As principais observações relacionadas à exibição das arquiteturas de componentes foram direcionadas aos conectores. Foi sugerido que eles ocupassem menos espaço na arquitetura, tornando-a mais clara e de fácil compreensão. Para o gerador de código foi levantada a possibilidade da ferramenta obter informações sobre a interação dos componentes e utilizar esta informação durante a geração de código para os conectores da arquitetura. A interação poderia ser definida em diagramas de atividades que seguissem algum padrão reconhecido pela ferramenta.

Usabilidade : o único ponto identificado para melhoria na usabilidade das ferramentas foi a inclusão de teclas de atalho. Um exemplo de uma operação que poderia possuir uma tecla de atalho é a operação “selecionar todos os elementos em uma arquitetura”.

Consolidação dos Resultados

As principais vantagens apontadas para as ferramentas utilizadas foram: (i) as funcionalidades disponibilizadas para a modelagem de uma arquitetura de componentes, (ii) a facilidade de uso das ferramentas e (iii) a geração automática de código em modelo COSMOS para a arquitetura de componentes. A principal desvantagem identificada foi a forma de organização e exibição da arquitetura de componentes.

5.3 Estudo de Caso: Um Sistema para Controle de Matrículas

5.3.1 Objetivo

O objetivo principal deste estudo de caso é avaliar as ferramentas para construção de arquiteturas de componentes presentes na infra-estrutura de software desta dissertação. O estudo de caso visou verificar se o código gerado pela infra-estrutura de software estava em conformidade com o modelo de componentes COSMOS (apresentado na Seção 2.5). Ainda, o estudo de caso teve como objetivo verificar até que ponto o código gerado auxiliava na implementação de um sistema seguindo este modelo de componentes. Foram avaliados o editor de arquiteturas (Seção 4.4.1), o editor de configurações (Seção 4.5.1) e o gerador de código (Seção 4.5.2). Este estudo de caso visou também encontrar possíveis defeitos na ferramenta e colher sugestões de novas funcionalidades.

5.3.2 Descrição do Estudo de Caso

Um protótipo de um sistema de controle de matrículas foi implementado por um aluno de mestrado do Instituto de Computação da Unicamp. Esta implementação foi feita seguindo o modelo de componentes COSMOS (Seção 2.5) e teve como objetivo servir como um exemplo de utilização deste modelo de componentes. Este protótipo foi disponibilizado para ser avaliado pelos interessados do Centro de Computação da Unicamp (CCUEC). A equipe de desenvolvimento de software do CCUEC tinha interesse em utilizar o modelo de componentes COSMOS, porém julgava necessária uma implementação de um protótipo seguindo o modelo COSMOS a partir de um produto de software previamente desenvolvido pelo Centro de Computação. A partir da implementação deste protótipo os arquitetos e projetistas de software do Centro de Computação poderiam optar pela utilização do modelo de componentes em suas soluções de software.

Este estudo de caso foi dividido em três etapas: (i) criar a modelagem da arquitetura de componentes do sistema de controle de matrículas, (ii) gerar o código, em modelo COSMOS, para esta arquitetura de componentes e (iii) comparar este código gerado com o código anteriormente desenvolvido para o protótipo do sistema. Para realizar a comparação entre o código gerado de forma automática e o código desenvolvido para o protótipo do sistema foi usada a seguinte estratégia: o código gerado de forma automática não poderia ser alterado, apenas as implementações dos métodos poderiam ser adicionadas reaproveitando o código existente no protótipo do sistema. Desta maneira, o código gerado de forma automática poderia ser analisado. Caso o sistema fosse executado corretamente, os componentes teriam sido corretamente gerados, instanciados e conectados.

A primeira parte do estudo de caso, compreendida pela criação da modelagem da arquitetura de componentes, foi elaborada pelo aluno de mestrado que implementou o protótipo do sistema. Apenas a instalação das ferramentas necessárias foi disponibilizada, visto que o aluno já possuía algum conhecimento das ferramentas da infra-estrutura de software para construção de arquiteturas de componentes. Um questionário foi disponibilizado para que o aluno pudesse avaliar as ferramentas utilizadas. O questionário utilizado está disponível no Apêndice C deste documento, porém os itens referentes a geração de código e verificação de arquiteturas de componentes não foram respondidos pelo aluno.

A segunda e terceira etapas do estudo de caso, compreendidas pela geração e análise do código em modelo COSMOS, foi realizada pelo autor deste trabalho. O estudo de caso foi realizado no mês de janeiro de 2007, tendo durado aproximadamente 3 dias. As atividades foram realizadas no Laboratório de Sistemas Distribuídos (LSD) do Instituto de Computação da Unicamp.

5.3.3 Descrição do Sistema

O sistema de controle de matrículas realiza o gerenciamento de alunos, o gerenciamento de cursos e as inscrições de alunos em determinados cursos. A Figura 5.7 apresenta os casos de uso do sistema. As funcionalidades disponibilizadas pelo sistema são:

Cadastrar Aluno : cadastra um novo aluno no sistema;

Cadastrar Curso : cadastra um novo curso no sistema;

Inscriver um Aluno em um Curso : matricula um aluno em um determinado curso.

A única validação necessária é garantir que o aluno não esteja cadastrado para o curso;

Listar Alunos : exibe os alunos cadastrados no sistema;

Listar Cursos : exibe os cursos cadastrados no sistema;

Listar Inscrições : exibe as matrículas cadastradas no sistema.

Um atributo imposto para o sistema é a plataforma de execução. Foi definido que o sistema deve ser disponibilizado para ser acessado via *intranet*, através de navegadores para a internet.

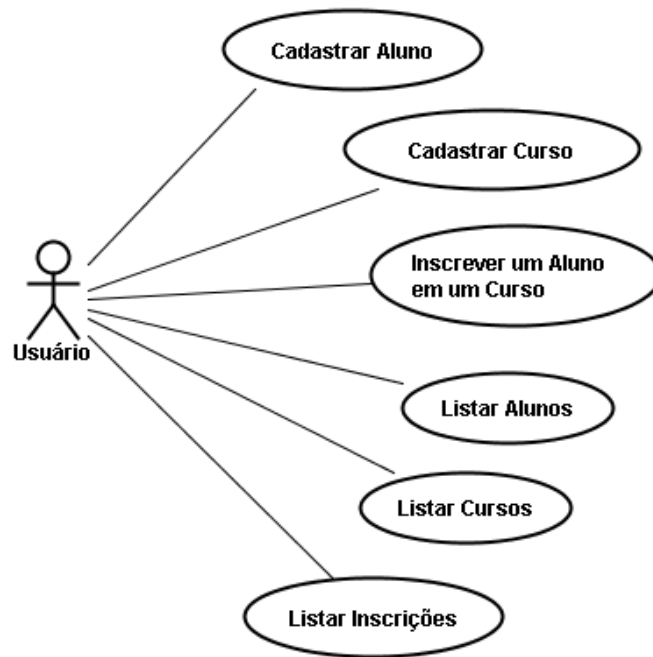


Figura 5.7: Casos de uso do sistema de controle de matrículas.

5.3.4 Especificação do Sistema

A Figura 5.8 apresenta a arquitetura de componentes para o sistema de controle de matrículas. Para facilitar a visualização da arquitetura de componentes, os conectores da arquitetura e as interfaces dos componentes foram omitidas. Para cada dependência apresentada na Figura 5.8 existe um conector que materializa esta conexão entre os componentes. O componente **SigaWeb** é responsável por realizar a interação entre o usuário e o sistema de controle de matrícula. O componente **SigaNegocio** implementa algumas regras gerais de negócio para a execução das operações. Os componentes **CursosCore**, **AlunosCore** e **InscricaoCore** implementam regras de negócio específicas para cada entidade manipulada pelo sistema (cursos, alunos e inscrições, respectivamente). Os componentes **CursosPersistencia**, **AlunosPersistencia** e **InscricaoPersistencia** são responsáveis por persistir as informações manipuladas pelo sistema.

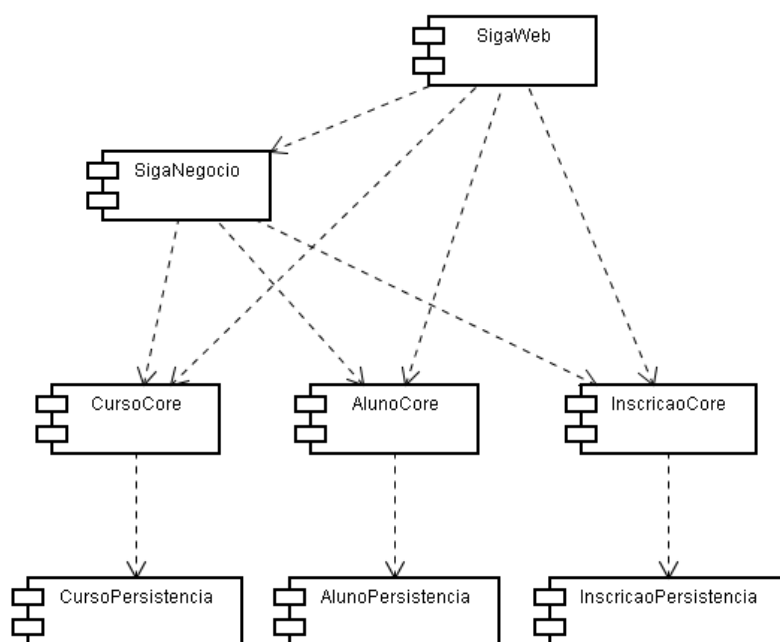


Figura 5.8: Arquitetura de componentes do sistema de controle de matrículas.

5.3.5 Execução do Estudo de Caso

Para a primeira etapa do estudo de caso, compreendida pela modelagem da arquitetura de componentes, o aluno de mestrado utilizou os editores do ambiente BELLATRIX relacionados a especificação de componentes e o editor de arquiteturas. Para efeito de avaliação neste estudo de caso, apenas o editor de arquiteturas foi considerado. O aluno levou aproximadamente uma hora para realizar a modelagem de toda a arquitetura de componentes do sistema. Um questionário de avaliação foi preenchido pelo aluno ao final desta etapa.

A segunda parte do estudo de caso foi realizada pelo autor desse trabalho, compreendendo a geração automática de uma implementação da arquitetura de componentes em modelo COSMOS. Foram gastos aproximadamente vinte minutos para a criação da configuração da arquitetura de componentes no Editor de Configurações e para a geração de código em modelo COSMOS.

A terceira parte do estudo de caso envolveu a análise do código gerado. O caso de uso “Cadastrar Alunos” foi escolhido para ser implementado e utilizado como análise. Foram gastos aproximadamente três horas para gerar a implementação e executá-la. Neste tempo está contabilizado o tempo gasto para configuração do servidor e instalação do sistema. Ao final da implementação, o caso de uso foi executado com sucesso, apresentando corretamente os resultados desejados.

5.3.6 Avaliação dos Resultados Obtidos

Avaliação Geral

A avaliação do estudo de caso pode ser dividida em duas partes: avaliação sobre a modelagem da arquitetura de componentes e avaliação sobre a materialização da arquitetura de componentes em código. A avaliação sobre a modelagem da arquitetura de componentes, realizada pelo aluno de mestrado, apontou que as ferramentas utilizadas possuem boas funcionalidades e boa maturidade. Nenhum erro foi encontrado durante a execução do estudo de caso e o sistema de ajuda não foi utilizado nenhuma vez. As notas atribuídas pelo aluno são apresentadas na Tabela 5.8.

Tabela 5.8: Notas atribuídas para as ferramentas.

Quesito	Nota
Funcionalidades presentes no ambiente	5
Facilidade de aprendizado por exploração	3
Produtividade e facilidade de uso	4
Nota geral	4

Em relação à materialização da arquitetura de componentes em uma implementação, a avaliação também foi positiva. O Gerador de Código gerou uma implementação que estava de acordo com o modelo de implementação COSMOS e facilitou a implementação do sistema. Todo o código de especificação dos componentes e o código de instanciação da arquitetura foram gerados de forma correta. As limitações apresentadas pelo gerador foram:

1. Algumas classes Java utilizadas nas interfaces dos componentes que não são tipos nativos da linguagem (como por exemplo a classe `java.util.Map`), não foram geradas corretamente. Estas classes, normalmente utilizadas em códigos Java, poderiam ser reconhecidas pelo Gerador de Código.
2. As exceções precisaram ser acrescentadas manualmente. Atualmente o Gerador de Código não gera o código para as exceções.
3. O código de implementação dos conectores não foi gerado. O Gerador de código poderia gerar implementações básicas para os conectores, onde a chamada é apenas encaminhada para o próximo componente.

Consolidação dos Resultados

As principais vantagens apontadas para as ferramentas utilizadas foram: (i) as funcionalidades disponibilizadas para a modelagem de uma arquitetura de componentes, (ii) a

facilidade de uso das ferramentas e (iii) a geração automática de código em modelo COSMOS para a arquitetura de componentes. As principais desvantagens apontadas foram: (i) a forma de organização e exibição da arquitetura de componentes e (ii) a limitação do gerador de código para tratar classes conhecidas na linguagem Java e para tratar exceções.

5.4 Consolidação Geral dos Resultados Obtidos

De maneira geral, as principais vantagens apontadas para a infra-estrutura de software baseada em componentes são:

1. Englobar os principais conceitos de arquitetura de software baseadas em componentes;
2. Facilidade em realizar análises sobre arquiteturas de componentes;
3. Possibilidade de geração automática de código em modelo COSMOS para uma arquitetura de componentes modelada na infra-estrutura de software deste trabalho.

As principais desvantagens identificadas para a ferramenta são:

1. Dificuldade em organizar e visualizar a arquitetura de componentes modelada no editor de arquiteturas;
2. Limitação na geração de código para classes conhecidas da linguagem Java;
3. Limitação na geração de código para exceções.

5.5 Resumo

Este capítulo apresentou a avaliação feita para a infra-estrutura de software deste trabalho. Foram apresentados três estudos de caso que, de maneira geral, avaliaram as ferramentas da infra-estrutura de software de maneira positiva, como possuindo boas funcionalidades e boa usabilidade. Também foram levantadas limitações na ferramenta, através de sugestões de novas funcionalidades e através de relatos de *bugs* encontrados. O próximo capítulo apresenta as considerações finais deste trabalho.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho apresentou uma infra-estrutura de software para apoiar a construção de arquiteturas de software baseadas em componentes. As ferramentas da infra-estrutura de software estendem o ambiente integrado de desenvolvimento Eclipse e estão inseridas dentro de um ambiente de suporte ao DBC, o BELLATRIX. Os principais objetivos da infra-estrutura de software são: permitir a criação de projetos arquiteturais para sistemas DBC, permitir a definição e atribuição de estilos arquiteturais, permitir análises sobre as arquiteturas de componentes e facilitar a materialização destas arquiteturas em uma implementação. Inicialmente foi utilizado o modelo de implementação de componentes COSMOS para a geração das implementações das arquiteturas de componentes.

Para a construção da infra-estrutura de software deste trabalho foram realizados o levantamento de requisitos, o projeto e a implementação das ferramentas. O desenvolvimento das ferramentas foi baseado em componentes de software e seguiu o processo UML Components. O levantamento dos requisitos para o ambiente levou em consideração dois aspectos: as informações necessárias para uma correta e completa modelagem de arquiteturas de componentes e as funcionalidades necessárias para as ferramentas de construção de arquiteturas de componentes. A partir deste levantamento e da análise dos conceitos envolvidos na disciplina de DBC foi elaborado um modelo conceitual de negócios, que foi nomeado neste trabalho como um metamodelo integrado para arquitetura de software e DBC.

O projeto arquitetural da infra-estrutura de software seguiu o processo UML Components, resultando inicialmente em uma arquitetura de componentes em camadas. Seguindo o processo, as interfaces de sistema foram criadas a partir dos casos de uso e as interfaces de negócio foram criadas a partir dos tipos principais do modelo conceitual. Foram criados componentes de sistema e de negócio para essas interfaces e uma arquitetura de componentes foi definida. Porém, como as ferramentas estendem o ambiente Eclipse, o estilo MVC imposto pelo ambiente também foi utilizado, resultando em uma arquitetura

de componentes final que segue dois estilos arquiteturais de forma complementar. Os componentes de sistema e de negócio atuaram como elementos de modelo dentro do estilo MVC imposto pelo ambiente Eclipse.

Diferentes tecnologias e modelos de implementação foram utilizados para a implementação das ferramentas. Para os componentes de sistema e de negócio, que compõem o núcleo das ferramentas, foi utilizada uma adaptação do modelo COSMOS para o ambiente Eclipse, permitindo que os *plug-ins* do ambiente pudessem atuar como componentes na arquitetura. Para a manipulação e persistência das arquiteturas de componentes criadas pelas ferramentas foi utilizada a tecnologia EMF. Para a parte de visualização e edição das arquiteturas através de diagramas e janelas foi utilizada a tecnologia GEF. A materialização das arquiteturas de componentes utiliza *templates* criados na tecnologia JET.

Foram realizados três estudos de caso para avaliar a ferramenta. O primeiro, que utilizou um sistema de cheques e contou com um grupo de voluntários, teve como objetivo avaliar as funcionalidades e a usabilidade das ferramentas para modelagem e verificação de arquiteturas de componentes. O segundo estudo de caso, que utilizou um ambiente de testes para sistemas DBC, teve como objetivo avaliar as funcionalidades e usabilidade das ferramentas para a modelagem de arquiteturas de componentes e avaliar a geração automática de uma implementação desta arquitetura de componentes seguindo o modelo de componentes COSMOS. O terceiro estudo de caso foi realizado para um sistema de controle de matrículas, tendo como principal objetivo analisar o código gerado pelas ferramentas da infra-estrutura de software. Nos três estudos de caso as ferramentas foram avaliadas através de um questionário e as análises destes questionários, no geral, relataram que as ferramentas possuem funcionalidades e usabilidade satisfatórias.

Uma versão contendo todos as funcionalidades básicas da infra-estrutura de software foi implementada e será disponibilizada no repositório CVS do grupo de pesquisas. A principal dificuldade encontrada na implementação da ferramenta está relacionada ao ambiente Eclipse e suas tecnologias, dado que existe um grau de dificuldade elevado em sua curva de aprendizado. Outra dificuldade de implementação ocorreu durante a utilização da biblioteca AcmeLib, que possui pouca documentação e apresentou erros durante a verificação dos estilos, sendo necessário corrigi-los.

6.1 Contribuições

As principais contribuições deste trabalho são:

- A participação na criação de um metamodelo integrado de arquitetura de software e DBC, parcialmente descrito em P. A. C. Guerra, T. C. Moronte, R. T. Tomita e C.

M. F. Rubira. “Um modelo conceitual e uma terminologia para o desenvolvimento baseado em componentes e baseado na arquitetura de software”. In: Anais do 5o. Workshop de Desenvolvimento Baseado em Componentes, 2005.

- Levantamento de requisitos para a ferramenta de descrição de arquiteturas de componentes, incluindo seus casos de uso, registrado em T. C. Moronte e C. M. F. Rubira. “Requisitos para uma Ferramenta de Descrição de Arquiteturas de Software para Sistemas Baseados em Componentes”. Relatório Técnico, Instituto de Computação / UNICAMP, 2006 (a ser publicado).
- Projeto arquitetural, especificação de componentes e soluções técnicas e tecnológicas para a ferramenta de descrição de arquiteturas de componentes, registrado em T. C. Moronte e C. M. F. Rubira. “Análise e Projeto de uma Ferramenta de Descrição de Arquiteturas de Software para Sistemas Baseados em Componentes”. Relatório Técnico, Instituto de Computação / UNICAMP, 2006 (a ser publicado).
- A implementação da ferramenta contendo todas as suas funcionalidades foi realizada e disponibilizada do repositório CVS do grupo. Futuramente a implementação será disponibilizada publicamente, para livre acesso.

6.2 Trabalhos Futuros

As seguintes funcionalidades foram identificadas para serem realizadas em trabalhos futuros para a ferramenta:

- Outras tecnologias e modelos de componentes poderiam ser implementados para a geração automática de código a partir de uma especificação de uma arquitetura de componentes. Atualmente somente o modelo COSMOS é disponibilizado.
- Atualmente a geração de código disponibilizada pela infra-estrutura de software pode ser utilizada apenas para gerar uma implementação de uma arquitetura de componentes. Não é possível gerar a implementação para um componente isoladamente. É interessante que estas implementações possam ser geradas para componentes isolados, permitindo que um componente possa ter uma implementação, independente da configuração a qual irá participar.
- Além da geração de código disponibilizada pelas ferramentas, seria muito interessante a possibilidade de realizar engenharia reversa de código em sistemas já implementados, extraindo seu projeto arquitetural. Esta funcionalidade daria maior poder à infra-estrutura de software por dois motivos: (1) ajudaria na evolução do

sistema, facilitando a consistência entre a especificação da arquitetura de componentes e sua especificação e (2) ajudaria na verificação de sistemas implementados, analisando se estão respeitando a arquitetura e os estilos arquiteturais desejados.

- Apesar das ferramentas modelarem a visão lógica dos sistemas através da especificação de sua arquitetura de componentes, outras visões poderiam ser disponibilizadas pela ferramenta [26]. Poderia ser estudada uma forma de integração com alguma ferramenta existente que permita a descrição de outras visões arquiteturais.
- Conforme identificado nos dois estudos de caso, é interessante que a exibição da arquitetura de componentes seja melhorada, permitindo uma visualização mais clara e facilitando sua manipulação.
- A infra-estrutura de software deste trabalho poderia ter uma ligação mais próxima com ferramentas para modelagem em linguagem UML, permitindo que o projeto detalhado de um componente possa ser realizado antes da geração de código. Atualmente, para realizar o projeto detalhado de um componente, primeiramente é necessário gerar o código, realizar a engenharia reversa em uma ferramenta de modelagem UML e depois detalhar o projeto interno do componente.
- As ferramentas poderia dar apoio a linhas de produtos para desenvolvimento de software. O metamodelo conceitual poderia ser estendido com os conceitos necessários em linhas de produtos e novas funcionalidades poderiam ser adicionadas às ferramentas.
- A especificação dos componentes deve possibilitar a definição de portas e de conexões de serviço. O metamodelo conceitual foi definido levando em consideração estes conceitos, porém as funcionalidades não foram disponibilizadas nos editores.

Com relação à avaliação das ferramentas, as seguintes sugestões foram identificadas como trabalhos futuros:

- Nos estudos de caso foram utilizadas especificações de arquiteturas que já haviam sido definidas. Seria interessante um estudo de caso para descrever um sistema de grande porte e que ainda não possua uma especificação de arquitetura de componentes definida. Este cenário seria interessante para avaliar a escalabilidade da ferramenta e a sua capacidade de lidar com uma especificação de uma arquitetura que pode sofrer diversas alterações.
- Os estudos de caso apresentados na Seção 5.2 e na Seção 5.3 geraram uma implementação da arquitetura de componentes em modelo COSMOS. É interessante que

seja criado um outro estudo de caso onde possa ser medido o tempo que pode ser economizado com a utilização do gerador de código.

6.3 Conclusão Final

A partir deste trabalho foi possível identificar a importância de se utilizar uma infraestrutura de software para a construção de arquiteturas de software de sistemas baseados em componentes. As ferramentas podem auxiliar deste a modelagem mais formal e completa dos sistemas de software até a sua materialização em forma de uma implementação. Outro ponto importante enfatizado neste trabalho é o auxílio das ferramentas de análises de arquiteturas de componentes, que possibilitam a descoberta prévia de problemas na modelagem de sistemas de software. Além da importância, este trabalho mostrou também a viabilidade da criação e uso destas ferramentas através de uma infra-estrutura de software.

Apêndice A

Sintaxe Linguagem OCL

Este apêndice descreve brevemente os operadores e funções da linguagem OCL utilizados para a definição das restrições dos metamodelos criados neste trabalho. Os operadores e funções utilizados foram:

allInstances : esta função é utilizada sobre um determinado tipo, tendo como retorno todas as instâncias que pertencem a este tipo e seus subtipos.

forAll : esta função deve ser utilizada sobre uma coleção de elementos, onde todos os elementos da coleção devem seguir uma determinada restrição que é definida como argumento da função. Retorna *verdadeiro* ou *falso* como resultado.

implies : este operador lógico define implicação. Exemplo de utilização deste operador

```
context Person inv:  
    self.wife->notEmpty() implies self.wife.age >= 18
```

includes : esta função é aplicada sobre uma coleção, passando um elemento como argumento. Retorna *verdadeiro* caso a coleção possua o elemento e *falso* caso não possua.

isUndefined : esta função deve ser aplicada sobre um atributo de um elemento, retornando *verdadeiro* caso o atributo não tenha sido definido, ou seja, não possua nenhum valor.

oclIsTypeOf : esta função deve ser aplicada sobre um elemento e recebe como argumento um tipo. A função retorna *verdadeiro* caso o elemento seja do tipo passado como argumento para a função, caso contrário, retorna *falso*.

select : esta função recebe como argumento uma determinada restrição e retorna um subconjunto de elementos de uma determinada coleção.

size : esta função retorna a quantidade de elementos de uma determinada coleção.

union : esta função deve ser aplicada sobre uma coleção, possuindo como argumento uma outra coleção. Retorna uma nova coleção contendo as duas coleção iniciais.

Apêndice B

Estilo DBC em Linguagem Acme

```
Family CBDFamily = {
  Port Type InterfaceType = {
    Property operations : Sequence<Operation>;
  }

  Port Type ProvidedInterface extends InterfaceType with {
  }

  Port Type RequiredInterface extends InterfaceType with {
  }

  Role Type InterfaceRoleType = {
    Property operations : Sequence<Operation>;
  }

  Role Type ProvidedInterfaceRole extends InterfaceRoleType with {
  }

  Role Type RequiredInterfaceRole extends InterfaceRoleType with {
  }

  Property Type ParamAttribute = Enum { _IN, _OUT, _INOUT };

  Property Type Param = Record [
    attribute : ParamAttribute;
    identifier : string;
    order : int;
    paramType : string;
  ];

  Property Type Parameter = Sequence<Param>;
```

```
Property Type Operation = Record [  
  identifier : string;  
  parameters : Parameter;  
  preCondition : string;  
  postCondition : string;  
  returnType : string;  
];  
  
Component Type CBDComponent = {  
  Property identifier : string;  
  invariant Forall p : port in self.ports |  
    declaresType(p, InterfaceType) ;  
}  
  
Connector Type CBDConnector = {  
  Property identifier : string;  
  invariant Forall r : role in self.roles |  
    declaresType(r, InterfaceRoleType) ;  
}
```

Apêndice C

Questionário de Avaliação da Infra-estrutura Proposta

C.1 Avaliação Geral

1. Ferramenta avaliada:
 - ☐ Editor de estilos ☐ Editor de arquiteturas
 - ☐ Verificador de arquiteturas ☐ Importador/Exportador de arquiteturas
 - ☐ Editor de configurações ☐ Gerador de código
2. Como você avalia seus conhecimentos sobre os conceitos de arquitetura de componentes e de DBC envolvidos no uso dos editores?
 - (a) Não conheço todos os conceitos envolvidos;
 - (b) Conheço o mínimo necessário para o uso da ferramenta;
 - (c) Tenho pleno conhecimentos dos conceitos;
 - (d) Tenho domínio sobre o assunto;
3. Sobre o sistema de ajuda (*help*):
 - (a) Você utilizou a ajuda?
 - (b) Quantas vezes?
 - (c) O que procurou na ajuda?
 - (d) A ajuda foi útil, ou seja, encontrou o que precisava?
4. Dê uma nota para:

(a) as funcionalidades presentes na ferramenta:

1	2	3	4	5
péssimo		razoável		ótimo

(b) a facilidade de aprendizado da ferramenta por exploração (ou seja, é fácil aprender a utilizá-la “fuçando”?):

1	2	3	4	5
péssimo		razoável		ótimo

(c) a produtividade e a facilidade de uso na ferramenta:

1	2	3	4	5
péssimo		razoável		ótimo

(d) a ferramenta, no geral:

1	2	3	4	5
péssimo		razoável		ótimo

5. Caso tenha utilizado outra ferramenta anteriormente, você considera a ferramenta de arquiteturas presente no ambiente BELLATRIX melhor, igual ou pior do que a ferramenta anterior para fazer a mesma tarefa? Qual é essa ferramenta?
6. Você tem sugestões de melhoria para problemas de usabilidade? (ou seja, pontos na interface com o usuário que são difíceis de entender ou confusos para se utilizar?)
7. Você tem sugestões de melhoria para as funcionalidades? Faltou alguma funcionalidade importante? Qual?
8. Ocorreram erros (*bugs*) no uso da ferramenta? Por favor, descreva-os para que possamos corrigi-los:

C.2 Avaliação Complementar para o Verificador de Arquiteturas

9. Todos os tipos puderam ser corretamente atribuídos aos elementos da arquitetura?
10. O verificador de arquiteturas apresentou violações na arquitetura? Estas violações puderam ser solucionadas? As mensagens apresentadas pelo verificador auxiliaram na correção destas violações?

C.3 Avaliação Complementar para o Gerador de Código

11. O código gerado pela ferramenta facilitou a implementação dos componentes seguindo o modelo COSMOS? O código gerado estava em conformidade com o modelo COSMOS?
12. Você tem sugestões para a geração de código? Existem outras estruturas de código que poderiam ser geradas automaticamente? Quais?
13. Você possui algum outro comentário sobre o gerador automático de código em modelo COSMOS?

Bibliografia

- [1] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187 – 197. ACM Press, 2002.
- [2] C. Atkinson et al. *Component-Based Product Line Engineering with UML*. Addison-Wesley, 2001.
- [3] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute at Carnegie-Mellon University, April 2000.
- [4] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing applications using model-driven design environments. *Computer*, 39(2):33–40, 2006.
- [5] Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt. A platform-independent component modeling language for distributed real-time and embedded systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 190–199, Washington, DC, USA, 2005. IEEE Computer Society.
- [6] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 2nd edition, 2003.
- [7] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [8] Regina M. M. Braga and Cláudia M. L. Werner. Odyssey-de: Um processo para desenvolvimento de componentes reutilizáveis. In *X Conferencia Internacional de Tecnologia de Software*, pages 177 – 194, 1999.

- [9] Regina M. M. Braga, Cláudia M. L. Werner, and Marta Mattoso. Odyssey: A reuse environment based on domain models. In *2nd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'99)*, pages 50 – 57. IEEE Computer Society, March 1999.
- [10] Patrick H. da Silva Brito. Um método para modelagem de exceções em desenvolvimento baseado em componentes. Master's thesis, Instituto de Computação — Universidade Estadual de Campinas, 2005.
- [11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1. John Wiley & Sons, 1996.
- [12] John Cheesman and John Daniels. *UML Components*. Addison-Wesley, 2000.
- [13] Adam Childs, Jesse Greenwald, Georg Jung, Matthew Hoosier, and John Hatcliff. Calm and cadena: Metamodeling for component-based product-line development. *Computer*, 39(2):42, 2006.
- [14] Paul Clements and Linda Northrop. Software architecture: An executive overview. Technical Report CMU/SEI-96-TR-003, Software Engineering Institute at Carnegie-Mellon University, 1996.
- [15] Philip T Cox and Baoming Song. A formal model for component-based software. In *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, page 304. IEEE Computer Society, 2001.
- [16] Eric M. Dashofy, AndréVan der Hoek, and Richard N. Taylor. A highly-extensible, xml-based architecture description language. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 103, Washington, DC, USA, 2001. IEEE Computer Society.
- [17] D. D'Souza and A. C. Wills. *Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.
- [18] Eclipse Consortium. C/C++ development tools. <http://www.eclipse.org/cdt/>, acessado em 30/01/2007.
- [19] Eclipse Consortium. COBOL IDE project. <http://www.eclipse.org/cobol>, acessado em 30/01/2007.
- [20] Eclipse Consortium. Eclipse.org main page. <http://www.eclipse.org/>, acessado em 30/01/2007.

- [21] Eclipse Consortium. EMF - eclipse modeling framework. <http://www.eclipse.org/emf>, acessado em 30/01/2007.
- [22] Eclipse Consortium. GEF - graphical editing framework. <http://www.eclipse.org/gef>, acessado em 30/01/2007.
- [23] Fernando Castor Filho, Patrick H. S. Brito, and Cecília Mary F. Rubira. A framework for analyzing exception flow in software architectures. In *WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1–7, New York, NY, USA, 2005. ACM Press.
- [24] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47 – 68. Cambridge University Press, 2000.
- [25] Institute for Software Integrated Systems at Vanderbilt University. : The generic modeling environment. <http://www.isis.vanderbilt.edu/Projects/gme/default.html>, acessado em 30/01/2007.
- [26] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.
- [27] Philippe Kruchten. *The Rational Unified Process — An Introduction*. Addison-Wesley, 1999.
- [28] Akos Ledeczki, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Proceedings of IEEE International Workshop on Intelligent Signal Processing (WISP'2001)*. IEEE Computer Society, 2001.
- [29] David C. Luckham, James Vera, and Sigurd Meldal. Key concepts in architecture definition languages. In Gary T. Leavens and Murli Sitaraman, editors, *Foundations of Component-Based Systems*, pages 23–45. Cambridge University Press, Cambridge, UK, 2000.
- [30] Chris Lüer. Evaluating the Eclipse platform as a composition environment. In *ICSE 2003: 3rd International Workshop on Adoption-Centric Software Engineering (ACSE 2003)*, CMU/SEI-2003-SR-004, pages 59 – 61. CMU/SEI, 2003.
- [31] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

- [32] Nikunj R. Mehta, Ramakrishna Soma, and Nenad Medvidovic. Style-based software architectural compositions as domain-specific models. Technical Report USC-CSE-2004-505, Department of Computer Science - University of Southern California, 2004.
- [33] Microsoft Corporation. Microsoft .net information. <http://www.microsoft.com/net/>, acessado em 30/01/2007.
- [34] Nelson Miler. A engenharia de aplicações no contexto da reutilização baseada em modelos de domínio. Master's thesis, COPPE - UFRJ, Julho 2000.
- [35] Tiago C. Moronte and Cecília M. F. Rubira. Requisito e projeto para uma ferramenta de descrição de arquiteturas de software para sistemas baseados em componentes. Technical report, Instituto de Computação / UNICAMP, 2006.
- [36] Object Management Group. Uml superstructure specification, v2.0. 2005.
- [37] Object Management Group. Object constraint language, v2.0. 2006.
- [38] Object Management Group. CORBA Component Model. <http://www.omg.org/technology/documents/formal/components.htm>, acessado em 30/01/2007.
- [39] Omondo. EclipseUML. <http://www.omondo.com>, acessado em 05/12/2006.
- [40] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 6th edition, 2005.
- [41] Camila R. Rocha, Patrick H. S. Brito, Eliane Martins, and Cecília M. F. Rubira. Um método de desenvolvimento e testes para sistemas confiáveis baseados em componentes: um estudo de caso. Technical Report 05-01, Instituto de Computação / UNICAMP, 2005.
- [42] Heloísa Vieira da Rocha and Maria Cecília C. Baranauskas. *Design e Avaliação de Interfaces Humano-Computador*. NIED / UNICAMP, 2003.
- [43] R. Roshandel, A. van der Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae - a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering and Methodology*, 11(2):240 – 276, April 2004.
- [44] Bradley Schmerl and David Garlan. AcmeStudio: Supporting style-centered architecture development. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 704 – 705. IEEE Computer Society, 2004.

- [45] Mary Shaw and Paul C. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, pages 6–13, Washington, DC, USA, 1997. IEEE Computer Society.
- [46] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4):314–335, 1995.
- [47] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [48] Moacir Silva, Jr, Paulo Asterio de C. Guerra, and Cecília Mary F. Rubira. A java component model for evolving software systems. In *Proceedings of IEEE International Conference on Automated Software Engineering (ASE'2003)*. IEEE Computer Society, October 2003.
- [49] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.
- [50] Sun Microsystems. Java 2 platform enterprise edition. <http://java.sun.com/j2ee/>, acessado em 30/01/2007.
- [51] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2nd edition, 2002.
- [52] Rodrigo T. Tomita. Bellatrix: Um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes. Master's thesis, Instituto de Computação — Universidade Estadual de Campinas, 2006.
- [53] Rodrigo T. Tomita, Fernando Castor, Paulo A. C. Guerra, and Cecília M. F. Rubira. Bellatrix: Um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes. In *Anais do 4o Workshop de Desenvolvimento Baseado em Componentes*, pages 43 – 48, 2004.
- [54] Rodrigo T. Tomita and Cecília M. F. Rubira. Requisitos para um ambiente de desenvolvimento baseado em componentes e centrado na arquitetura de software. Technical Report IC-05-23, Instituto de Computação / UNICAMP, 2005.
- [55] W3C Consortium. Extensible markup language (xml) 1.0 (third edition). <http://www.w3.org/TR/REC-xml/>, acessado em 30/01/2007.