BELLATRIX: **Um Ambiente para Suporte Arquitetural ao Desenvolvimento Baseado em Componentes**

Rodrigo Teruo Tomita

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Bellatrix: Um Ambiente para Suporte Arquitetural ao Desenvolvimento Baseado em Componentes

Rodrigo Teruo Tomita

Junho de 2006

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)
- Profa. Dra. Itana Maria de Souza Gimenes
  Departamento de Informática — Universidade Estadual de Maringá
- Profa. Dra. Eliane Martins
  Instituto de Computação — UNICAMP
- Profa. Dra. Ariadne M. B. R. Carvalho (suplente)
  Instituto de Computação — UNICAMP

IE IBC
MADA
EX
BC/ 70822
16.133.06
D X
11,00
05/12/06
392817
T/UNICAMP
T595b

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecária: Miriam Cristina Alves – CRB8a / 9054

Tomita, Rodrigo Teruo.

T595b Bellatrix: um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes / Rodrigo Teruo Tomita -- Campinas, [S.P. :s.n.], 2006.

Orientadora: Cecília Mary Fischer Rubira

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

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

Título em inglês: Bellatrix: an environment for providing architectural support to component-based development.

Palavras-chave em inglês (Keywords): 1. Software - Reuse . 2. Software - Architecture. 3. Software - Development. 4. Software engineering.

Área de concentração: Sistemas de informação.

Titulação: Mestre em Ciência da Computação

Banca examinadora: Profa. Dra. Cecília Mary Fischer Rubira (IC-UNICAMP)
Profa. Dra. Itana Maria de Souza Gimenes (DIN-UEM)
Profa. Dra. Eliane Martins (IC-UNICAMP)

Data da defesa: 28/07/2006

Programa de Pós-Graduação: Mestrado em Ciência da Computação

Bellatrix: Um Ambiente para Suporte Arquitetural ao Desenvolvimento Baseado em Componentes

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Rodrigo Teruo Tomita e aprovada pela Banca Examinadora.

Campinas, 28 de julho de 2006.

Profa. Dra. Cecília Mary Fischer Rubira
(Orientadora)

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

Termo de Aprovação

Dissertação defendida e aprovado em 28 de julho de 2006, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Cecília Mary Fischer Rubira
IC/UNICAMP

Profa. Dra. Itana Maria de Souza Gimenes
UEM- Maringá

Profa. Dra. Eliane Martins
IC/UNICAMP

© Rodrigo Teruo Tomita, 2006.
Todos os direitos reservados.

Resumo

O desenvolvimento baseado em componentes (DBC), que se baseia na construção de software através da integração planejada de componentes reutilizáveis, tem conquistado ampla aceitação para o desenvolvimento de sistemas de software. O desenvolvimento centrado na arquitetura de software é complementar ao DBC, com a responsabilidade pela integração dos componentes de forma que atributos de qualidade, como confiabilidade e distribuição, desejados para o sistema final sejam obtidos. Assim, processos de DBC também devem ser centrados na arquitetura de software, possibilitando um maior grau de abstração, organização, manutenibilidade e reuso. Para possibilitar a automatização de tarefas e aumentar a produtividade no uso dos conceitos de arquitetura de software e DBC, são necessárias ferramentas que apoiem atividades de arquitetos e desenvolvedores de software. Os ambientes integrados de desenvolvimento existentes atualmente apóiam, em geral, a modelagem UML e a implementação de componentes e de sistemas orientados a objetos. Entretanto, eles não apóiam a prática de modelagem de arquiteturas de componentes e DBC. Ferramentas para modelagem de arquiteturas de software existentes não dão suporte ao mapeamento da arquitetura para código, possuem um foco específico na modelagem e não são apoiadas por um processo de desenvolvimento. Nesse trabalho é proposto o ambiente Bellatrix, um ambiente integrado de desenvolvimento que apoia o DBC com ênfase na arquitetura de software e estende o ambiente integrado de desenvolvimento Eclipse. O ambiente pode ser usado em conjunto com um processo de DBC e utiliza o COSMOS, um modelo de implementação de componentes que faz o mapeamento dos conceitos de arquiteturas de software para linguagens de programação orientadas a objetos. Este trabalho se concentra na especificação e projeto do ambiente Bellatrix, partindo de seus requisitos e protótipos de interface com usuário. Seus principais componentes foram especificados, bem como um modelo de implementação dos mesmos, integrando o modelo COSMOS ao ambiente Eclipse. Uma primeira implementação do ambiente contendo suas principais funcionalidades foi desenvolvida.

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

Abstract

Component-based development (CBD), which is based on the assembly of software systems through planned integration of reusable components, is gaining wide acceptance for developing software systems. 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. Thus, CBD processes should also be software architecture centric, promoting a higher abstraction level, system organization, maintainability, and reuse. In order to allow tasks automation and increase the productivity in using software architecture and CBD concepts, tools that support architects and software developers are needed. Most of the existing integrated development environments support UML modeling, object-oriented and component implementation. However, many of them do not support the practice of component architecture and CBD modeling. Existing software architecture modeling tools lack the ability to translate the architecture to code, have a specific focus on modeling, and are not guided by a development process. In this work, we describe the Bellatrix environment, an integrated development environment that supports CBD with emphasis on software architecture and extends the Eclipse integrated development environment. The proposed environment can be guided by a CBD process and uses COSMOS, a component implementation model that materializes the elements of a software architecture using the concepts available in object-oriented programming languages. This work focuses on the general description of Bellatrix, starting from its requirements and user interface prototypes. Its main components has been specified, as well as an implementation model integrating the COSMOS model with the Eclipse environment. An initial version containing basic functionalities has been developed.

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

Agradecimentos

Agradeço primeiramente a Deus, por tudo.

Gostaria de agradecer à profa. Cecília Rubira, por sua orientação e paciência (e também pelos "puxões de orelha"), sem os quais eu não teria realizado este trabalho.

Agradeço também ao Paulo Asterio e ao Fernando Castor, pela "co-orientação" e por todas as sugestões, críticas e contribuições feitas a esse trabalho.

Gostaria de agradecer à AUTBANK Projetos e Consultoria Ltda. pelo apoio financeiro, e em especial ao Paulo Kato e ao Tomohiro Nakaya, que contribuíram com sua experiência para que este trabalho (e minha própria visão) ficasse mais próximo da realidade da indústria.

Agradecimentos ao projeto "COMPGOV" — MCT/FINEP/Acao Transversal Biblioteca de Componentes - 05/2004 — pelo apoio financeiro.

Agradeço aos membros da banca examinadora, professoras Itana e Eliane pelas sugestões e críticas que possilitaram a melhoria deste trabalho.

Gostaria de agradecer a professora Cecília Baranauskas, a Amanda Melo e a Silvia Soares pelo apoio na indicação de técnicas de projeto e avaliação de interfaces com usuário utilizadas.

Agradecimentos aos professores e funcionários do Instituto de Computação / UNICAMP, que tanto contribuíram para minha formação.

Agradeço também aos colegas do grupo de pesquisas em engenharia de software, em especial a Ana Lobo, Camila Rocha, Leonardo Tizzei, Leonel Gayard, Patrick Brito e Tiago Moronte pelas sugestões, revisões e ajuda no desenvolvimento deste trabalho e em seu estudo de caso. Um agradecimento extra ao Patrick Brito e Tiago Moronte, pelas discussões acaloradas, suas revisões e seu companherismo e ao Leonel Gayard pela ajuda nas experiências de implementação.

Agradecimentos também aos colegas da CC99 e do LSD, em especial ao Ulisses Furquim, Sheila Almeida e Gilberto Pastorello Jr pelo companherismo e apoio.

Por fim, mas não menos importante, gostaria de agradecer minha família, em especial meus pais, sem os quais eu não estaria aqui, minhas queridas irmãs e meu tio Kentian. Um agradecimento especial a minha noiva Arlete, por todo o carinho e apoio que me deu.

Conteúdo

Resumo **vii**

Abstract **viii**

Agradecimentos **ix**

1 Introdução **1**
1.1 Motivação e Problema 4
1.2 Solução Proposta 5
1.3 Trabalhos Relacionados 8
1.4 Organização deste Documento 10

2 Fundamentos de Arquiteturas de Software, DBC e Ambientes de Desenvolvimento **13**
2.1 Arquitetura de Software 13
2.2 Desenvolvimento Baseado em Componentes 15
2.3 O Processo de DBC UML Components 17
2.4 O Modelo de Implementação COSMOS 19
2.5 Ambientes de Desenvolvimento 20
2.6 O Ambiente Eclipse 21
2.7 Resumo 22

3 Requisitos para um Ambiente com Apoio ao DBC **25**
3.1 Requisitos 25
3.1.1 Requisitos Identificados por Guerra 26
3.1.2 Requisitos Identificados por Lüer *et al.* 27
3.1.3 Requisitos Identificados por Lucrédio *et al.* 28
3.2 Consolidação dos Requisitos para o Bellatrix 28
3.2.1 Lista de Requisitos Apoiados pelo Ambiente Bellatrix 30
3.2.2 Lista de Requisitos Não Apoiados pelo Ambiente Bellatrix . . . 30

3.2.3 Atributos de Qualidade . . . 31
3.2.4 Restrições de Implementação . . . 32
3.3 Casos de Uso . . . 33
3.4 Protótipos de Interface com o Usuário . . . 36
3.5 Resumo . . . 38

4 Projeto do Ambiente BELLATRIX 41
4.1 Arquitetura Inicial do Ambiente . . . 41
4.2 Modelo Conceitual do Negócio . . . 43
4.3 Especificação dos Componentes . . . 46
4.4 Arquitetura de Componentes do Ambiente BELLATRIX . . . 48
4.5 Resumo . . . 49

5 Implementação do Ambiente BELLATRIX 53
5.1 Implementação do Núcleo do Ambiente . . . 53
5.1.1 Modelo de *Plug-ins* do Eclipse . . . 54
5.1.2 Modelo de Implementação de Componentes Utilizado . . . 55
5.2 Implementação do Modelo de Tipos . . . 57
5.3 Interface Gráfica com Usuário no Eclipse . . . 57
5.4 Gerenciador do Processo . . . 58
5.5 Estratégia de Testes para o Desenvolvimento . . . 59
5.6 Resumo . . . 60

6 Estudo de Caso: Sistema de Reservas de Hotel 61
6.1 Objetivo . . . 61
6.2 Material e Método . . . 61
6.3 Descrição do Sistema de Reservas de Quartos . . . 63
6.4 Execução . . . 65
6.4.1 Levantamento de Requisitos (Fase 1) . . . 65
6.4.2 Identificação de componentes (Fase 2.1) . . . 66
6.4.3 Interação de componentes (Fase 2.2) . . . 71
6.4.4 Especificação de componentes (Fase 2.3) . . . 75
6.4.5 Provisionamento (Fase 3) e Montagem (Fase 4) . . . 78
6.5 Avaliação . . . 80
6.6 Resumo . . . 81

7 Estudo de Caso: Sistema Financeiro 85
7.1 Objetivo . . . 85
7.2 Material e Método . . . 85

7.3 Descrição do Sistema Financeiro 86
7.4 Especificação do Sistema 89
7.5 Execução 95
7.6 Avaliação 96
7.7 Resumo 97

8 Conclusões e Trabalhos Futuros **99**
8.1 Contribuições 100
8.2 Trabalhos Futuros 101

A Questionário de Avaliação do Ambiente BELLATRIX **105**

Bibliografia **107**

Lista de Tabelas

3.1 Requisitos para o ambiente Bellatrix 29
3.2 Exemplo de descrição de caso de uso. 38

6.1 Descrição de caso de uso Reservar Quarto. 68

7.1 Notas atribuídas pelos voluntários ao Bellatrix. 96

Lista de Figuras

1.1 Visão geral do ambiente Bellatrix. 6

2.1 Estilo arquitetural MVC. 15
2.2 Exemplo de um componente. 16
2.3 Arquitetura proposta pelo processo UML Components. 17
2.4 Visão geral das fases do processo UML Components 18
2.5 Visão geral de uma configuração simples no modelo COSMOS. 20

3.1 Diagrama de atividades do funcionamento do processo UML Components. 34
3.2 Visão geral dos casos de uso do ambiente. 35
3.3 Diagrama de casos de uso do pacote Especificar Componente. 37
3.4 Protótipo de interface com o usuário para associar uma interface à especificação de um componente. 39

4.1 Visão geral do ambiente Bellatrix apresentada na Seção 1.2. 42
4.2 Visão lógica da arquitetura do ambiente Bellatrix 43
4.3 Modelo conceitual de arquiteturas de software. 44
4.4 Modelo conceitual de DBC. 44
4.5 Modelo conceitual de interfaces baseado no padrão CORBA IDL. 45
4.6 Modelo conceitual de testes. 47
4.7 Exemplo de componente de sistema. 48
4.8 Exemplo de componente de negócios. 49
4.9 Arquitetura de componentes das camadas de sistema e negócios do núcleo do ambiente Bellatrix. 50

5.1 Estrutura de um *plug-in* combinado com o modelo COSMOS. 56
5.2 Arquitetura de um editor utilizando o GEF. 58

6.1 Casos de uso do sistema de reservas. 64
6.2 Modelo do processo do negócio. 66
6.3 Modelo conceitual do negócio. 67

6.4 Interface de sistema `IMakeReservation`. 69
6.5 Modelo de tipos do negócio. 70
6.6 Componente de negócio `HotelMgr`. 71
6.7 Editor de componentes do Bellatrix (1). 72
6.8 Editor de componentes do Bellatrix (2). 73
6.9 Editor de arquiteturas do Bellatrix. 74
6.10 Exemplo de diagrama de colaboração para interação de componentes. . . . 75
6.11 Detalhando operações no Bellatrix. 76
6.12 Editor de tipos de dados estruturados do Bellatrix. 77
6.13 Editor de configurações do Bellatrix. 79

7.1 Diagrama de casos de uso do sistema. 88
7.2 Arquitetura do sistema. 89
7.3 Interfaces da camada de negócios. 90
7.4 Tipos de dados (*datatypes*) do sistema. 91
7.5 Exceções lançadas pela interface `ISBContaMgrReq`. 92
7.6 Componentes da camada de negócios. 93
7.7 Arquitetura de componentes do sistema. 94
7.8 Arquitetura interna do componente `operacoesConta`. 95

Capítulo 1

Introdução

A demanda por sistemas de software tem forte crescimento e, na mesma proporção, cresce também a dependência que pessoas e empresas têm sobre esses sistemas. Eles fazem parte de aplicações que vão desde ferramentas de "bate-papo" até sistemas financeiros e de comércio eletrônico. Com essa demanda, aumentam também as exigências sobre o desenvolvimento de software: são necessários sistemas maiores e mais complexos, desenvolvidos com menor custo e maior qualidade, e entregues em menores prazos. Ainda, atributos de qualidade, tais como confiabilidade, segurança e distribuição, são cada vez mais comuns e importantes para essas aplicações.

Uma das abordagens utilizadas para auxiliar desenvolvedores a lidar com essas exigências é o **desenvolvimento baseado em componentes (DBC)**. O DBC é uma abordagem interessada na construção rápida de sistemas a partir de componentes pré-fabricados [5]. O uso de componentes pode trazer vários benefícios [67]. O DBC leva à divisão modular dos sistemas, desde seus requisitos até sua implementação. Essa divisão propicia um aumento da adaptabilidade, escalabilidade e manutenibilidade do sistema, em relação a uma construção monolítica. A construção de sistemas novos a partir de componentes pré-existentes, adquiridos ou desenvolvidos, aliado ao reuso dos mesmos, reduz os custos e o tempo de desenvolvimento, potencialmente aumentando a qualidade dos sistemas, através do uso de componentes previamente testados ou certificados. Um **componente de software** é uma unidade de composição com interfaces especificadas contratualmente e dependências explícitas de contexto. Um componente pode ser implantado individualmente, composto terceiros [67] e deve seguir um modelo de componentes [5]. Um componente também deve ser documentado, possuir alta coesão e baixo acoplamento, sendo uma unidade de desenvolvimento e reuso [6]. Componentes podem ser compostos (conectados entre si) formando **sistemas**, cuja noção é recursiva, ou seja, componentes podem representar subsistemas que possuem arquiteturas internas. Uma **interface** identifica um ponto de interação entre um componente e o seu ambiente [28].

Esse ambiente pode ser constituído por outros componentes de software e partes de software não baseadas em componentes. Ainda, uma interface é a abstração de um serviço, definindo as operações oferecidas pelo mesmo, mas não sua implementação [67]. Um componente possui **interfaces providas**, através das quais ele declara os serviços oferecidos ao ambiente e **interfaces requeridas**, pelas quais ele declara os serviços do ambiente dos quais depende para funcionar [6]. Interfaces e componentes podem ser especificados em momentos distintos do processo de desenvolvimento [15]. Um componente possui uma **especificação** e uma **implementação**. A especificação de um componente é a descrição do mesmo, incluindo suas interfaces providas e requeridas. Uma implementação de componente representa um componente em tempo de execução, ou seja, uma instância de sua especificação. Essa instância deve se comportar como descrito em sua especificação e pode existir mais de uma instância distinta para uma mesma especificação [6]. Um **modelo de implementação de componentes** especifica padrões e convenções que devem ser seguidos pelos desenvolvedores de componentes, com o objetivo de se criar componentes que possam ser compostos de maneira uniforme e intercambiáveis, definindo tipos de componentes e suas formas de interação [5]. Assim, um modelo de componentes possibilita que componentes criados por desenvolvedores distintos possam ser integrados em sistemas, de modo que também torna possível o comércio de tais componentes. Uma **plataforma de componentes** é uma infra-estrutura de software que provê serviços em tempo de execução aos componentes que seguem um dado modelo de componentes. Esses serviços podem ser gerenciamento do ciclo de vida (instanciação, destruição) dos componentes, persistência, transação e segurança no acesso. Exemplos de plataformas incluem J2EE[1] [64], recentemente rebatizado para Java EE[2], e o Microsoft .Net [45].

Outra abordagem utilizada para se lidar com o tamanho e a complexidade de sistemas de software é decompô-los em subsistemas menores, cada qual provendo um conjunto de serviços relacionados. O processo inicial de identificação desses subsistemas e do estabelecimento de um padrão para controle e comunicação entre eles é chamado de **projeto arquitetural**, e seu produto é a descrição da **arquitetura de software** [63]. Arquitetura de software diz respeito à estrutura das partes de um sistema, as propriedades externamente visíveis dessas partes e como elas se relacionam entre si [8, 28]. Tal estrutura fornece um alto grau de abstração, desconsiderando detalhes de implementação das partes, o que permite lidar melhor com a complexidade dos sistemas. Uma arquitetura de software pode ser vista como uma estrutura de **elementos arquiteturais**. Esses elementos podem ser **componentes** e **conectores arquiteturais**, onde os componentes são as principais unidades de computação e os conectores materializam explicitamente as interações entre os componentes [61]. Os pontos de interação entre esses elementos podem

[1] do inglês *Java 2 Enterprise Edition.*
[2] do inglês *Java Platform, Enterprise Edition.*

ser chamados de **interfaces** ou **portas**. Uma dada organização de componentes e conectores interligados entre si em um sistema é denominada **configuração arquitetural**. Assim, uma arquitetura de software é uma abstração do sistema que representa decisões de projeto, podendo ser analisada para construir soluções de nível estrutural e reutilizada entre sistemas semelhantes. Ainda, ela pode ser utilizada como elemento de comunicação entre os interessados pelo sistema (ou *stakeholders*). O projeto arquitetural pode ser baseado em um modelo ou estilo arquitetural. Um **estilo arquitetural** é um conjunto de restrições sobre uma arquitetura, que tem como aspectos positivos apresentar atributos de qualidade conhecidos e representar soluções conhecidas para uma classe de sistemas [8]. Muitos sistemas grandes possuem suas partes projetadas de modo a combinar diferentes estilos arquiteturais. A arquitetura de um sistema é importante para se obter requisitos de qualidade, tais como desempenho, robustez, distribuição e manutenibilidade, de modo que tais requisitos devem ser planejados em nível arquitetural. Assim, a escolha de um estilo ou estrutura em particular depende dos requisitos não-funcionais do sistema. A arquitetura provê infra-estrutura para se alcançar tais requisitos, entretanto ela por si só não garante que os mesmos serão obtidos.

O DBC e o desenvolvimento centrado na arquitetura de software [8] são abordagens complementares, de modo que o DBC possui foco na produção e uso de componentes reutilizáveis e a arquitetura de software, na integração desses componentes. Componentes de software reutilizáveis são, em geral, desenvolvidos sem o conhecimento do contexto específico de todos os sistemas onde serão empregados no futuro. Cabe, assim, à arquitetura de software a responsabilidade pela integração desses componentes de forma que os atributos de qualidade desejados para o sistema em desenvolvimento sejam obtidos. A complementariedade dessas abordagens se evidencia nos principais processos de DBC, tais como Kobra [4], UML Components [15] e Catalysis [18], que além de promoverem o reuso de componentes, são também centrados na arquitetura do software.

O processo de desenvolvimento de software pode ser definido como o conjunto de atividades e seus resultados, que levam ao desenvolvimento de um produto de software [63]. Esse processo deve ser apoiado por ferramentas, de modo a tornar possível e economicamente viável o desenvolvimento de sistemas de software maiores e mais complexos [26]. Um processo de desenvolvimento pode conter atividades dois focos distintos: **processo gerencial**, que trata de planejamento e acompanhamento de prazos e cronogramas, alocação de recursos a tarefas, entregas de produtos, entre outros, e o **processo de desenvolvimento**, que cuida da criação de software que atenda aos requisitos do usuário e que tenha certos atributos de qualidade [15]. No contexto de DBC, o processo de desenvolvimento pode ser dividido em duas fases [31, 72]: a **produção de componentes de software**, onde o foco é a criação de componentes que sejam reutilizáveis, também chamado de desenvolvimento de componentes ou desenvolvimento "para

reuso"; e (2) a **integração de aplicações**, onde o foco é a integração de componentes em aplicações, também conhecido como desenvolvimento com componentes ou desenvolvimento "com reuso". O elo entre essas duas fases são os **repositórios de componentes**, infra-estruturas de software capazes de armazenar componentes, onde os componentes são depositados por seus produtores e recuperados pelos integradores. O repositório de componentes trata de um requisito crítico para o DBC, em particular, e para o reuso de software, em geral: deve ser possível encontrar componentes reutilizáveis, através de uma base devidamente catalogada e documentada [63].

1.1 Motivação e Problema

A eficiência de um processo de desenvolvimento de software depende da existência de ferramentas de software adequadas. Essas ferramentas são chamadas de CASE (do inglês *Computer Aided Software Engineering*) [63] e abrangem desde compiladores e link-editores até sofisticados editores de código, depuradores e geradores. O desenvolvedor de software utiliza tais ferramentas para alcançar maior produtividade e menor número de erros. Um conjunto dessas ferramentas podem ser integradas, dando apoio a grandes partes ou a todo o processo de desenvolvimento, formando um **ambiente** de desenvolvimento [25]. Os ambientes de desenvolvimento podem ser **centrados no processo**, ambientes baseados em uma definição formal do processo de desenvolvimento utilizado [25]. Nesses ambientes, um modelo do processo indica quais atividades devem ser executadas, por quem elas devem ser executadas, com quais ferramentas e o formato dos documentos a serem produzidos. O ambiente interpreta esse modelo e, com base nele, pode guiar desenvolvedores, automatizar tarefas e executar ferramentas [7].

Ferramentas que apóiem as atividades de arquitetos e desenvolvedores são necessárias para se obter produtividade em processos de DBC e centrados em arquiteturas de software. Essas ferramentas devem apoiar explicitamente a construção de especificações de interfaces e componentes, modelagem da arquitetura de componentes, geração de código e integração com um repositório de componentes. As ferramentas de software existentes em geral não oferecem um apoio integral a todas essas atividades, podendo ser classificadas em pelo menos três grandes grupos: (1) ferramentas de modelagem de arquiteturas de software; (2) ferramentas de modelagem UML, voltadas para projeto e implementação de sistemas orientados a objetos; e (3) ferramentas de apoio ao DBC, voltadas para a implementação de componentes e montagem de sistemas.

As ferramentas que dão apoio à modelagem de arquiteturas de software são, em geral, projetos de iniciativa acadêmica. Existem projetos maduros para modelagem de arquiteturas e estilos arquiteturais, como AcmeStudio [59], com suporte a estilos arquiteturais específicos, como o ArchStudio [37] e a famílias de arquiteturas, como o Ménage [17].

Entretanto, essas ferramentas não dão apoio ao DBC devido à falta de conceitos, como interfaces providas e requeridas. Ainda, elas não possuem integração com um processo de desenvolvimento de software e, apesar de algumas dessas ferramentas incluírem recursos para geração de código, não há a materialização explícita dos elementos arquiteturais do sistema no código gerado.

Com relação a ferramentas de modelagem UML, existem diversas ferramentas em diferentes graus de maturidade, entre ferramentas comerciais, gratuitas e *open source*, das quais podemos citar Rational Software Modeler [35] e EclipseUML [55]. A maior parte delas já adota ou está se adequando ao padrão da UML 2.0, em que o suporte à notação envolvendo DBC foi aprimorado. Entretanto, o suporte à geração de código continua relacionado à criação de classes e não é possível descrever o projeto interno de um componente.

No contexto do apoio à implementação de componentes e montagem de sistemas, existem ambientes integrados de desenvolvimento (IDE, do inglês *Integrated Development Environment*) — ferramentas que integram editores de código, compiladores, depuradores, entre outras funcionalidades de apoio ao desenvolvedor, tais como Microsoft Visual Studio .Net [46] e Rational Application Developer for WebSphere Software [34]. Essas ferramentas, em geral, dão suporte a uma linguagem de programação ou a um conjunto delas. As linguagens orientadas a objetos mais modernas — tais como Java, C++ e C# — não possuem suporte nativo para o conceito de componentes, de modo que recorre-se ao uso de plataformas de componentes para o DBC nessas linguagens. As plataformas mais populares, como o J2EE [64] e o Microsoft .Net [45], contam com o suporte das IDEs para a especificação de componentes, sua implementação e implantação. Entretanto, nota-se em tais plataformas — e conseqüentemente nos ambientes que as suportam — a ausência de conceitos importantes de DBC, tais como decomposição hierárquica, interfaces requeridas e representação de interações entre componentes como entidades de primeira ordem (conectores arquiteturais). Além disso, elas não possuem integração com um repositório de componentes e, embora algumas forneçam suporte para a modelagem de sistemas [55], elas não apresentam as características necessárias para a modelagem de arquiteturas de software.

1.2 Solução Proposta

Neste trabalho é proposto o ambiente Bellatrix[3], uma extensão do ambiente Eclipse para DBC, centrado na arquitetura de software e com apoio ao processo DBC. Um dos principais objetivos do ambiente Bellatrix é facilitar a transição de um projeto de

[3]Bellatrix, também chamada de *Gamma Orionis*, é uma estrela de segunda magnitude da constelação de Orion. Seu nome tem origem no latim e significa "guerreira" ou "amazona".

arquitetura de software para a implementação do sistema baseado em componentes, de acordo com o paradigma DBC. O ambiente **Eclipse** é um ambiente integrado de desenvolvimento construído de modo a ser uma plataforma de integração de ferramentas. Cada ferramenta o estende através de módulos anexáveis chamados *plug-ins*. Assim, o ambiente Bellatrix estende o Eclipse através de um conjunto de *plug-ins*. Uma visão geral do ambiente Bellatrix e suas ferramentas é apresentada na Figura 1.1. Nessa figura, pode-se observar o processo de DBC, registrado no Gerenciador de Workflow que indica uma série de ferramentas. Essas ferramentas foram construídas sobre o Núcleo do Ambiente, onde as operações de negócio do ambiente estão armazenadas. O arquiteto e desenvolvedor de componentes utiliza o gerenciador e a plataforma Eclipse, tornando transparente qual ferramenta é utilizada a cada tarefa. Os demais elementos dessa figura são explicados em maiores detalhes a seguir.

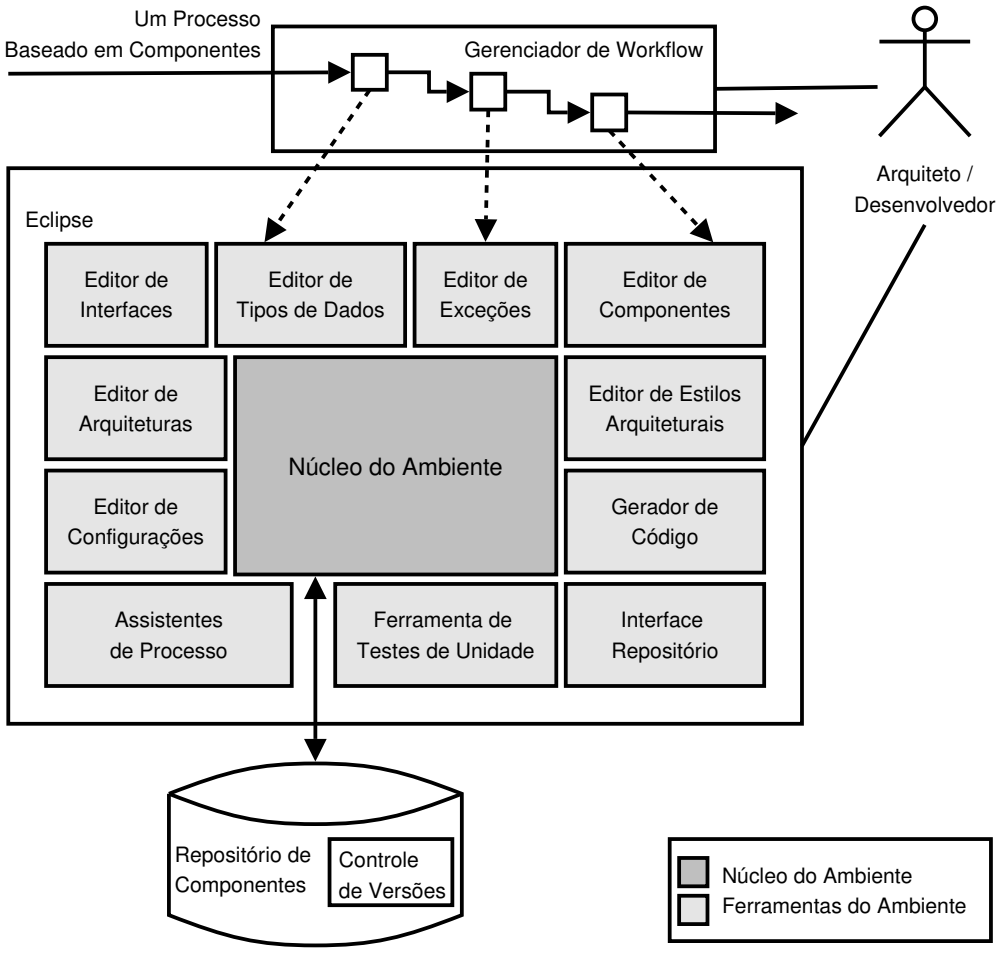

Figura 1.1: Visão geral do ambiente Bellatrix.

O ambiente Bellatrix provê apoio a um processo de DBC. O processo de desenvolvimento pode ser modelado em uma ferramenta de *workflow*, o que torna mais simples adaptar o processo à realidade de cada equipe de desenvolvimento e permite guiar o usuário durante o processo, descrevendo as etapas e associando-as com as ferramentas a serem utilizadas. Essa ferramenta é representada como Gerenciador de Workflow na parte superior da Figura 1.1. Ainda, assistentes do processo (*wizards*) podem ser utilizados para dar apoio à criação de artefatos e tarefas específicas do processo. Esses Assistentes do Processo são ferramentas integradas ao ambiente Bellatrix conforme mostra a mesma figura. Atualmente, o ambiente Bellatrix provê apoio específico ao processo UML Components [15], um processo de DBC com foco na especificação de componentes, possibilitando auxiliar o usuário nas tarefas de especificação de interfaces e componentes. A escolha por esse processo foi feita por sua simplicidade e seu pragmatismo. O foco num processo de DBC em específico nos auxiliou no levantamento das atividades às quais o ambiente deve dar suporte. A construção do ambiente pela composição de ferramentas permite maior adaptabilidade a alterações no processo. Novas ferramentas, específicas a um novo processo, também podem ser integradas no futuro. Assim, o ambiente é baseado no processo UML Components, mas esse não é o único processo que pode ser apoiado.

O ambiente possui, entre outras ferramentas, um conjunto de editores que permitem a modelagem gráfica da arquitetura de componentes. Esses editores, mostrados na Figura 1.1 como as duas primeiras linhas de ferramentas apoiadas sobre o Eclipse, dão suporte às seguintes atividades: (1) criação e edição de interfaces e suas operações (Editor de Interfaces) com assinatura completa, bem como tipos de dados estruturados de parâmetros e retorno (Editor de Tipos de Dados) e exceções (Editor de Exceções); (2) especificação de componentes, associando-os a suas interfaces providas e requeridas (Editor de Componentes); (3) associação entre componentes através de conectores, modelando a arquitetura do sistema em termos das especificações dos componentes e suas interfaces (Editor de Arquiteturas), permitindo ainda a criação e edição de configurações de componentes, especificando um conjunto de implementações de componentes que materializam cada uma das especificações da arquitetura (Editor de Configurações); (4) especificação de estilos arquiteturais, aplicando-os às arquiteturas criadas e validando suas restrições (Editor de Estilos Arquiteturais). Esses editores persistem os modelos em arquivos de especificação em formato XML (do inglês *Extensible Markup Language*) [71] de modo independente da plataforma, possibilitando a separação entre especificação e implementação. Ainda, ferramentas para extrair a especificação (engenharia reversa) de um componente binário em modelos de componentes diferentes podem ser anexadas.

O ambiente Bellatrix permite a geração de código de acordo com o modelo COSMOS, promovendo a conformidade entre a arquitetura de software e o código. Um Gerador de Código cria os esqueletos de código dos componentes, seguindo o modelo COSMOS,

um modelo de implementação de componentes que faz o mapeamento dos conceitos de arquiteturas de software para linguagens de programação orientadas a objetos. Inicialmente, os código será gerado na linguagem Java e para a plataforma J2EE. Com a integração com o Eclipse, o usuário pode se beneficiar de seu apoio para a codificação dos componentes. Ainda, é possível a criação de outros geradores de código para outras plataformas de componentes, como por exemplo .Net. Seguindo as ferramentas na Figura 1.1, uma Ferramenta de Testes de Unidade está presente no ambiente para possibilitar que desenvolvedores possam testar componentes COSMOS individualmente, sem a necessidade de instanciar a configuração inteira. Uma ferramenta de acesso a um repositório de componentes (Interface Repositório) permite a integração do ambiente com diferentes repositórios existentes. Assim, um Repositório de Componentes será integrado ao ambiente. Idealmente, esse repositório deve permitir o armazenamento de diferentes artefatos produzidos durante o processo de desenvolvimento, entre eles, especificações e implementações de componentes. Ainda, ele deve dar suporte à evolução dos componentes depositados, fazendo uso de uma Ferramenta de Controle de Versões.

As principais características do ambiente são: (1) suporte à modelagem de interfaces; (2) suporte à especificação de componentes; (3) suporte à modelagem da arquitetura de software de um sistema, em termos de especificação de componentes, conectores e interfaces; (4) clara separação entre artefatos de especificação e implementação de componentes; (5) apoio gráfico / visual para as atividades de modelagem, facilitando seu uso; (6) aderência a um processo de DBC, inicialmente o processo UML Components; (7) materialização dos componentes modelados utilizando um modelo de componentes, facilitando seu uso através da geração de código; (8) integração das ferramentas, reunindo modelagem e implementação de componentes; (9) desenvolvimento com código aberto (*open source*). Devido à sua importância, o suporte a estilos arquiteturais e o repositório de componentes em si foram considerados durante as fases iniciais da especificação do ambiente. Entretanto, devido ao seu grau de complexidade, eles não foram considerados como parte do escopo desse trabalho.

1.3 Trabalhos Relacionados

O ambiente ADE [49, 50, 13], *Architecture Design Environment*, é um projeto de pesquisa desenvolvido pelo *Center for Strategic Technology Research* da antiga *Andersen Consulting*. Ele é um ambiente gráfico para apoiar o projeto de arquiteturas de componentes utilizando a linguagem de descrição ASL, *Architecture Specification Language*, criada no contexto desse projeto. O ASL é uma extensão de IDL (do inglês *Interface Description Languague*) [52] de CORBA (do inglês *Common Object Request Broker Architecture*) para incluir os conceitos de componentes, interfaces, conexões e configurações. Assim, o ADE

possui conceitos importantes de DBC, tais como interfaces providas e requeridas e decomposição hierárquica. Ainda, ele possui funcionalidades interessantes como a geração automática de conectores (*adapters*), a verificação de compatibilidade entre interfaces conectadas numa configuração e a criação de rotinas para a geração de código de "cola" (*stubs* e *skeletons*) para a plataforma escolhida, por exemplo, modelo de componentes CORBA [51]. Entretanto, esse ambiente não está integrado a um ambiente de desenvolvimento, nem a um repositório de componentes e ele não é baseado num processo. Ainda, ele não está disponível livremente para uso.

O ambiente WREN [42] é um protótipo de um ambiente de DBC integrado ao IDE Web Gain Visual Café e à ferramenta de modelagem UML ArgoUML [3]. Integrado também a um repositório de componentes desenvolvido dentro do contexto do projeto, ele permite localizar componentes e integrá-los em aplicações, permitindo a especificação gráfica de configurações. O ambiente define um modelo de componentes próprio, baseado no modelo Javabeans [66], utilizando a linguagem de programação Java. Esse modelo possui um mecanismo de auto-descrição, através do qual os componentes informam suas interfaces providas e requeridas. Para a execução de um sistema construído nesse modelo, é necessário criar um descritor da configuração especificada no ambiente. Esse descritor é passado a um ambiente de execução, que faz o papel dos conectores. Os trabalhos sobre o WREN descrevem um processo genérico com reuso ao qual o ambiente dá apoio. Entretanto, o ambiente não dá suporte ao desenvolvimento de componentes — ou desenvolvimento para reuso — e não faz separação entre a especificação de um componente e sua implementação, no sentido de não persistir a especificação de modo independente de como ela é materializada em código. Ainda, o WREN possui uma dependência para com o ambiente Visual Café que era distribuído somente de modo comercial e atualmente não é mais comercializado.

O ambiente ORION [40] é um ambiente de DBC com suporte ao desenvolvimento de sistemas distribuídos. Ele é formado pela integração de ferramentas existentes e um processo de DBC, propostos pelo mesmo grupo de pesquisas. Tais ferramentas incluem ferramentas CASE, um repositório de artefatos de software e uma plataforma de comunicação (*middleware*). São elas: (1) MVCASE, uma ferramenta para criação de diagramas UML, que permite múltiplas visões e persistência no formato XMI (do inglês *XML Metadata Interchange*) [53] (um tipo de XML para representação de diagramas UML). Ela dá apoio à modelagem de casos de uso, modelos de tipos e arquitetura de componentes, possibilitando a geração de código em Java, J2EE e CORBA; (2) JADE, ferramenta de desenvolvimento que permite a codificação, compilação, execução e depuração de código na linguagem Java. Ele faz a persistência do código desenvolvido num formato XMI estendido; (3) um Repositório de artefatos de software, baseado no formato XMI, que possibilita, entre outras funções, controle de versão e de acesso; (4) JAMP, uma ferra-

menta desenvolvida para o desenvolvimento de aplicações multimídia, que no ambiente ORION é utilizado como plataforma de comunicação entre as ferramentas e o repositório. Ele é compatível com Java RMI (do inglês *Remote Method Invocation*) [65] e CORBA; (5) MoDPAI, ferramenta para monitoramento de dispositivos móveis, utilizado no ambiente para apoiar a implantação dos componentes distribuídos devido a sua funcionalidade para descoberta da topologia da rede; e (6) IPM, um processo para desenvolvimento de componentes distribuídos. É um processo incremental que permite o desenvolvimento de componentes e a integração de aplicações. Ainda, o ORION foi desenvolvido e é baseado na tecnologia Java, o que lhe confere independência de plataforma. Ele possui integração entre as ferramentas de modelagem e desenvolvimento. A persistência dos artefatos no formato XMI permite a separação de especificação e implementação. Entretanto, ele não possui suporte ao conceito interfaces requeridas e conectores.

O ambiente Odyssey [11] é um ambiente de DBC que consiste de processos e ferramentas para a especificação de modelos e aplicações de um domínio específico, visando promover o reuso no desenvolvimento de software. O ambiente dá apoio ao processo de engenharia de domínio Odyssey-DE [10] e ao processo de engenharia de aplicação Odyssey-AE [47]. Desenvolvido em Java, o ambiente é composto de um núcleo, chamado *Odyssey Light*, que pode ser acrescido de ferramentas *plug-ins*, que são instanciadas dinamicamente. Essas ferramentas incluem diversas funcionalidades, tais como ferramentas para a documentação de componentes, especificação e instanciação de arquiteturas específicas de domínios, apoio à engenharia reversa, ferramenta de críticas em modelos UML, 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. Com o foco em engenharia de domínio, o Odyssey permite a modelagem do domínio, incluindo a criação de modelos de características e casos de uso, a partir dos quais é possível especificar componentes de negócios seguindo o processo. A persistência dos modelos pode ser feita através da serialização de objetos em arquivos ou em um sistema de banco de dados. O Odyssey possui suporte 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 disponível para uso livre, apesar de não ter seu código aberto. Ainda, ele não integra ferramentas de implementação dos componentes individualmente.

1.4 Organização deste Documento

Este documento foi dividido em 7 capítulos ao todo, organizados da seguinte forma:

- **Capítulo 2 - Fundamentos de arquiteturas de software, DBC e ambientes de desenvolvimento:** Neste capítulo são definidos alguns dos conceitos e termos importantes que formam a base teórica para este trabalho. Nele são abordados os seguintes assuntos: arquiteturas de software, DBC, o processo baseado em componentes UML Components, o modelo de implementação de componentes COSMOS, ambientes de desenvolvimento e o ambiente Eclipse.

- **Capítulo 3 - Requisitos para um ambiente para apoio ao DBC:** Este capítulo descreve o levantamento dos requisitos do ambiente Bellatrix. Assim, são descritos os trabalhos de levantamento das funcionalidades as quais o ambiente deve dar apoio, incluindo trabalhos que detalham requisitos para ambientes de DBC, os casos de uso identificados para o ambiente e os protótipos de interface com o usuário desenvolvidos.

- **Capítulo 4 - Projeto do ambiente** Bellatrix**:** Esse capítulo descreve o projeto do ambiente Bellatrix. Primeiramente, é apresentada a arquitetura inicial do ambiente e o modelo conceitual (ou modelo de classes) utilizado no mesmo. Na seqüência, a especificação dos componentes que compõem o ambiente Bellatrix, incluindo como eles foram especificados, é descrita. Por fim, a arquitetura de componentes do ambiente é mostrada.

- **Capítulo 5 - Implementação do ambiente** Bellatrix**:** Neste capítulo é descrita a estratégia utilizada para a implementação do ambiente Bellatrix. Assim, a materialização de componentes das camadas de sistema e negócios no ambiente Eclipse é descrita, propondo-se um modelo de componentes para o mesmo. Ainda, é mostrada a implementação do modelo de tipos do negócio na camada de persistência do ambiente e as tecnologias envolvidas e o desenvolvimento da interface gráfica com o usuário do ambiente, utilizando o estilo arquitetural MVC. Na seqüência, é descrita a implementação do gerenciador do processo do ambiente e a estratégia de testes do ambiente.

- **Capítulo 6 - Estudo de caso: sistema de reservas de hotel:** Este capítulo apresenta o estudo de caso realizado a fim de mostrar a viabilidade do projeto e da implementação inicial do ambiente Bellatrix. Foi modelado um sistema fictício de reservas de quartos em hotel seguindo o processo UML Components. A cada fase do processo, os artefatos gerados e as ferramentas utilizadas foram descritas, bem como limitações e dificuldades encontradas.

- **Capítulo 7 - Estudo de caso: sistema financeiro:** Este capítulo apresenta um estudo de caso em que um sistema financeiro real baseado em componentes foi

modelado no ambiente Bellatrix. A especificação desse sistema foi modelada por diferentes voluntários, que preencheram questionários sobre as funcionalidades e a usabilidade do ambiente.

- **Capítulo 8 - Conclusões e trabalhos futuros:** Este capítulo apresenta as conclusões desse trabalho, sintetizando suas contribuições, identificando limitações e sugerindo trabalhos futuros.

Capítulo 2

Fundamentos de Arquiteturas de Software, DBC e Ambientes de Desenvolvimento

Neste capítulo são definidos os conceitos e termos importantes para a compreensão do restante do trabalho. Na Seção 2.1 são apresentados os conceitos utilizados sobre arquitetura de software. A Seção 2.2 descreve conceitos sobre o desenvolvimento baseado em componentes e, em especial, o que se considera componente de software no contexto deste trabalho. A Seção 2.3 explica o processo de desenvolvimento baseado em componentes UML Components, o qual o ambiente Bellatrix apóia. A Seção 2.4 apresenta o modelo de implementação de componentes COSMOS, no qual a implementação deste trabalho se baseou (Capítulo 5) e cujo uso também é apoiado por este ambiente. A Seção 2.5 discorre sobre ambientes de desenvolvimento orientados a processo e a Seção 2.6 trata especificamente do ambiente integrado de desenvolvimento Eclipse, sobre o qual o Bellatrix foi construído. Por fim, a Seção 2.7 sumariza este capítulo.

2.1 Arquitetura de Software

A decomposição e organização de um sistema de software em subsistemas menores é uma abordagem comumente utilizada para se lidar com a complexidade e o tamanho de sistemas de software. Assim, a **arquitetura de software** de um sistema define uma estrutura de alto nível do mesmo em termos de elementos arquiteturais, abstraindo detalhes de sua implementação [8, 28]. Os **elementos arquiteturais** representam uma parte do sistema que é responsável por um determinado comportamento ou propriedade desse sistema, provendo um conjunto de serviços relacionados. Esses elementos podem ser componentes e conectores arquiteturais. Um **componente arquitetural** é responsável pela funciona-

lidade de um sistema de software. Componentes arquiteturais podem ser definidos com diferentes granularidades, desde um componente que provê apenas uma funcionalidade básica até um subsistema complexo responsável por um amplo conjunto de funcionalidades. Um **conector arquitetural** tem como principal responsabilidade mediar a interação entre outros elementos arquiteturais. Um conector pode ser responsável também por uma parcela dos aspectos de qualidade (ou não-funcionais) do sistema, tais como distribuição e segurança. Os elementos arquiteturais possuem **pontos de conexão** que definem formas de interação entre um elemento e o seu ambiente [28]. Esses pontos de conexão podem ser chamados de **interfaces** ou **portas**. Uma determinada organização de componentes e conectores arquiteturais interligados entre si em um sistema é denominada **configuração arquitetural**. Muitos sistemas de software possuem diferentes estruturas, tais como sua organização lógica, processos, dados e código-fonte. Ainda visando lidar com a complexidade e tamanho dos sistemas, pode-se abstrair alguns tipos de estruturas, concentrando a atenção em outros. Uma **visão arquitetural** descreve uma determinada perspectiva de uma arquitetura de software, representando um conjunto coerente de elementos arquiteturais [8].

Arquiteturas de software são importantes pois elas representam uma abstração comum do sistema que pode ser utilizada na comunicação entre os interessados (*stakeholders*). Ainda, elas documentam decisões de projeto feitas sobre o sistema, que podem ser analisadas para construir soluções reutilizáveis entre sistemas semelhantes [8]. Essas soluções compõem estilos arquiteturais. Um **estilo arquitetural** impõe um conjunto de restrições sobre uma arquitetura, definindo tipos de elementos arquiteturais e padrões de comunicação entre eles. Ele tem como aspectos positivos apresentar atributos de qualidade conhecidos e representar soluções conhecidas para uma classe de sistemas [8]. Sistemas de software podem ser construídos com base em um ou mais estilos arquiteturais diferentes.

Um exemplo de estilo arquitetural é o estilo **MVC** (do inglês *Model View Control*) [14]. A Figura 2.1 mostra uma visão esquemática desse estilo. O MVC tem por objetivo desacoplar a interface gráfica do resto da aplicação. Nesse estilo, o Modelo contém as operações e os dados da aplicação. Ele não contém informações sobre como os dados serão apresentados ao usuário, apenas permite que as Visões se registrem como interessados (ou observadores) do modelo e os notifica quando ocorrem alterações no mesmo. Uma Visão, por sua vez, é responsável pela forma com que os dados serão apresentados ao usuário. Um modelo pode possuir várias visões. Por exemplo, uma série de números (modelo) pode ser apresentada como uma tabela (uma visão) ou um gráfico (outra visão). Os elementos de Controle, ou controladores, são responsáveis pela interação com o usuário, por atualizar o modelo quando o usuário interage com uma visão associada a ele. Assim, o modelo alterado notifica todas as suas visões, fazendo com que elas se atualizem também.

Com o objetivo de se representar uma arquitetura de software, foram propostas na

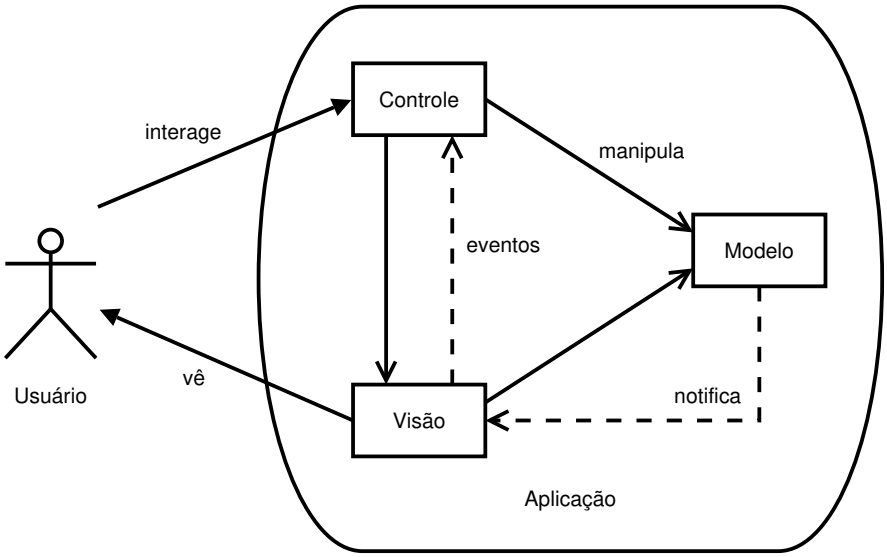

Figura 2.1: Estilo arquitetural MVC.

literatura diferentes linguagens de descrição de arquiteturas (**ADL**, do inglês *Architecture Description Language*), tal como ACME [28]. Em geral, os elementos básicos de uma ADL incluem formas para se representar componente e conectores, assim como regras de conexão e regras sintáticas para a criação de arquiteturas [8]. As principais vantagens do uso de ADLs são a representação formal de uma arquitetura e a possibilidade de análise e manipulação através de ferramentas.

2.2 Desenvolvimento Baseado em Componentes

O DBC é uma técnica de desenvolvimento de software que se baseia na construção rápida de sistemas a partir de componentes pré-fabricados [5]. O DBC leva à divisão modular dos sistemas, desde seus requisitos até sua implementação. Essa divisão propicia um aumento da adaptabilidade, escalabilidade e manutenibilidade do sistema, em relação a uma construção monolítica [67]. A construção sistemas novos a partir de componentes pré-existentes, adquiridos ou desenvolvidos, aliado ao reuso dos mesmos, reduz os custos e o tempo de desenvolvimento, potencialmente aumentando a qualidade dos sistemas, através do uso de componentes previamente testados ou certificados.

Um **componente de software** é uma unidade de composição com interfaces e dependências de contexto explicitamente especificadas, que pode ser fornecido isoladamente para integrar sistemas de software desenvolvidos por terceiros. Um componente possui **interfaces providas**, através das quais ele declara os serviços oferecidos ao ambiente e **interfaces requeridas**, pelas quais ele declara os serviços do ambiente dos quais depende

para funcionar [6]. Uma **interface** identifica um ponto de interação entre um componente e o seu ambiente [28]. A Figura 2.2 mostra um um exemplo de um componente e a notação utilizada em UML. Nessa figura, o `Componente A` possui duas interfaces: uma interface provida chamada `I1` e uma interface requerida chamada `I2`.

Figura 2.2: Exemplo de um componente.

Um componente de software possui uma especificação abstrata que pode ser materializada em diferentes implementações. A **especificação de um componente** 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. Uma especificação de componente possui uma ou mais implementações. Uma **implementação de componente** é um módulo executável resultante de um processo de refinamento e tradução de uma especificação de componente. Uma implementação de componente pode ser utilizada como item de configuração para compor diferentes sistemas de software executáveis. Um exemplo de especificação de componente é um um navegador Internet, que deve seguir a especificação HTTP de comunicação (interface), mas que possui diferentes implementações (Internet Explorer, Netscape, Mozilla, etc).

Um componente pode ser um **componente elementar** — uma implementação atômica de um componente de software — ou um **componente composto** — uma implementação de componente estruturada internamente como um conjunto de subcomponentes a serem providos separadamente. Uma implementação de componente representa um componente em tempo de execução, ou seja, uma instância de sua especificação. Essa instância deve se comportar como descrito em sua especificação e pode existir mais de uma instância distinta para uma mesma especificação [6]. Um **modelo de componentes** especifica padrões e convenções que devem ser seguidos pelos desenvolvedores de componentes, com o objetivo de se criar componentes que possam ser compostos de maneira uniforme e intercambiáveis, definindo tipos de componentes e suas formas de interação [5]. Assim, um modelo de componentes possibilita que componentes criados por desenvolvedores distintos possam ser integrados em sistemas, de modo que também torna possível o comércio de tais componentes. Uma **plataforma de componentes** é uma infra-estrutura de software que provê serviços em tempo de execução aos componentes que seguem um dado modelo de componentes. Esses serviços podem ser gerenciamento do ciclo de vida (instanciação,

destruição) dos componentes, persistência, transação e segurança no acesso. Exemplos de plataformas incluem J2EE[1] [64], recentemente rebatizado para Java EE[2], e o Microsoft .Net [45].

2.3 O Processo de DBC UML Components

O **UML Components** [15] é um processo de DBC, cujo foco é em sistemas de informação e na especificação de seus componentes. O processo é iterativo e pode ser utilizado como uma extensão do processo RUP (do inglês *Rational Unified Process*) [38], com as fases análise, projeto e implementação substituídas pelas fases de especificação, provisionamento e montagem. O UML Components estende a notação UML [9] adequando-a à representação de elementos necessários ao DBC.

O UML Components possui o foco em sistemas de informação, direcionando a criação de arquiteturas seguindo o estilo em camadas [14]. A Figura 2.3 mostra a arquitetura em que a aplicação é dividida em três camadas: (1) a camada de interação com o usuário; (2) a camada de serviços de sistema e; (3) a camada de serviços de negócios. A camada de interação com o usuário é onde fica a parte do sistema responsável pela interação (apresentação e entrada de dados) e diálogo com o usuário. A camada de serviços de sistema contém os componentes de sistema, responsáveis pelos serviços oferecidos pelo sistema. Ela funciona como a visão externa do sistema, uma fachada para a camada inferior. Os componentes dessa camada provêem as interfaces de sistema, que têm origem no modelo de casos de uso da aplicação. A camada de serviços de negócios possui os componentes de negócios, que armazenam as informações e regras de negócios do sistema. Esses componentes provêem as interfaces de negócios, que são originadas do modelo conceitual de negócios, um diagrama de classes contendo os principais conceitos do negócio.

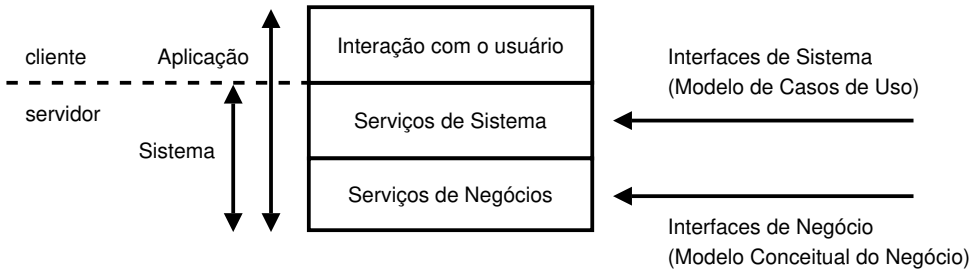

Figura 2.3: Arquitetura proposta pelo processo UML Components.

A Figura 2.4 mostra uma visão geral das fases definidas pelo processo. Essas fases

[1] do inglês *Java 2 Enterprise Edition*.
[2] do inglês *Java Platform, Enterprise Edition*.

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.

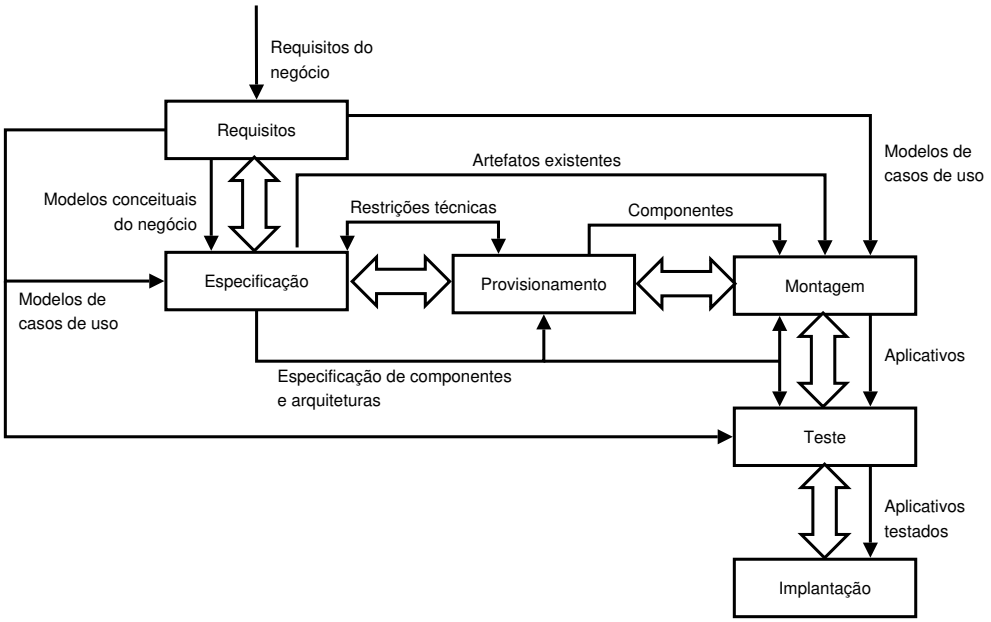

Figura 2.4: Visão geral das fases do processo UML Components

Ainda, a fase de (2) especificação é dividida em três subfases: (2.1) identificação de componentes, fase na qual as interfaces do sistema são identificadas a partir de seus casos de uso e de sua modelagem do negócio; (2.2) interação de componentes, em que são especificadas as interações planejadas entre os componentes. A partir dessas interações, novas operações e a assinatura das mesmas são definidas. (2.3) especificação de componentes, quando se consolida a arquitetura de componentes do sistema, produzindo documentação individual para cada componente.

Dada a idéia do UML Components ser uma extensão do RUP, 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 têm sido adotado, entre outros processos de DBC, devido à sua simplicidade e ao seu pragmatismo.

2.4 O Modelo de Implementação COSMOS

O COSMOS [62] é um modelo genérico e independente de plataforma para a implementação de componentes de software. Os principais objetivos do COSMOS são garantir que a implementação de um sistema está em conformidade com sua arquitetura de componentes e facilitar a evolução dessa implementação. Assim, o modelo define regras de mapeamento para transformar uma visão arquitetural do sistema — composta por componentes arquiteturais, conectores e configurações — em elementos disponíveis em linguagens de programação orientadas a objetos, tais como pacotes e classes, materializando os principais conceitos de arquiteturas de software.

O modelo incorpora um conjunto de diretrizes de projeto propostas na literatura, entre elas: a materialização de elementos arquiteturais, interfaces e conectores; o isolamento dos requisitos não-funcionais; a separação entre especificação e implementação; a declaração explícita das dependências entre componentes, através de interfaces requeridas; o forte encapsulamento da implementação; a separação entre herança de tipos e herança de código; e o baixo acoplamento entre classes de implementação.

O COSMOS define três sub-modelos que focam em diferentes aspectos do DBC: (1) o **modelo de especificação** descreve as interfaces providas e requeridas de cada componente do sistema; (2) o **modelo de implementação** define como os serviços providos pelo componente são implementados; e (3) o **modelo de conectores** especifica as conexões entre componentes, possibilitando que dois ou mais componentes sejam conectados entre si numa configuração.

A Figura 2.5 mostra uma configuração simples composta por dois componentes e uma visão geral dessa configuração no modelo COSMOS. No nível arquitetural da figura, pode-se observar o componente A, que provê a interface I1 e requer a interface I2. Esse componente está conectado ao componente B, que provê a interface I2, através do conector AB. O componente B requer a interface I3. No nível de implementação, os componentes e conectores são materializados como diferentes pacotes de implementação. Cada um dos componentes possui pacotes que materializam os modelos de especificação e implementação. O pacote de especificação ainda é subdividido em dois, um para as interfaces providas e outro para as interfaces requeridas. O pacote de implementação contém as classes que implementam o serviço do componente, além de classes de infra-estrutura como fábricas, fachadas e outros. Essas classes usam as interfaces requeridas do componente. O conector AB materializa o modelo de conectores, possuindo classes que implementam a interface I2 requerida por A e usa uma implementação da interface I2 provida por B.

O modelo COSMOS pode ser utilizado para implementação na linguagem de programação Java e a plataforma J2EE [62], bem como a linguagem C# e a plataforma Microsoft .Net [16].

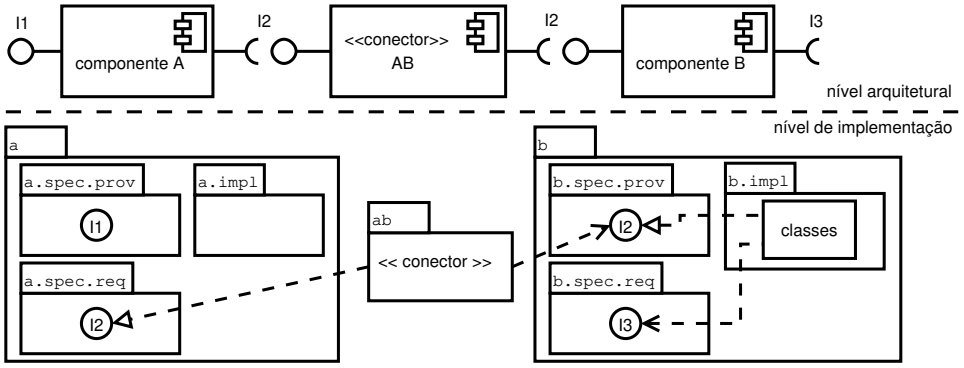

Figura 2.5: Visão geral de uma configuração simples no modelo COSMOS.

2.5 Ambientes de Desenvolvimento

O uso de ferramentas de suporte ao desenvolvimento de software é uma necessidade antiga. Desde compiladores e link-editores até sofisticados editores de código, depuradores e geradores, o desenvolvedor utiliza o apoio de tais ferramentas para alcançar produtividade e menor número de erros. Essas ferramentas são chamadas de CASE (do inglês *Computer Aided Software Engineering*) [63].

Uma **ferramenta CASE** é uma ferramenta de software para auxiliar em uma tarefa específica no processo de produção de software. Essas ferramentas podem ser classificadas de acordo com a atividade a qual elas dão suporte em [25]: (1) ferramentas de edição (por ex. editores de código); (2) de programação (por ex. compiladores); (3) de verificação e validação (por ex. geradores de testes); (4) de gerência de configuração (por ex. controle de versões); (5) de métricas (por ex. analisadores de código); (6) de gestão de projetos (por ex. estimativa de custos); e (7) outras atividades (por ex. visualizadores de HTML). Buscando uma maior integração dessas ferramentas em torno do processo de desenvolvimento, elas podem ser agrupadas em *workbenches* e ambientes.

Workbenches integram em uma única aplicação várias ferramentas, dando apoio a diferentes atividades do desenvolvimento de software. Eles possuem uma interface com o usuário homogênea, navegação entre as ferramentas facilitada e integração dos dados [25]. Essa integração pode ocorrer através de um repositório de dados compartilhado [63]. Assim como as ferramentas, eles podem ser classificadas de acordo com o conjunto de atividades ao qual dão apoio: (1) modelagem do negócio; (2) análise e projeto; (3) desenvolvimento de interface com o usuário; (4) codificação; (5) verificação e validação; (6) manutenção e engenharia reversa; (7) gerência de configuração; e (8) gestão de projetos.

Um **ambiente** é uma coleção de ferramentas integradas, que dá apoio ao processo de desenvolvimento todo ou a grandes partes dele. Os ambientes podem ser classificados em [25]: (1) *toolkits*, que são um conjunto de ferramentas fracamente integradas, mas que

suportam diferentes atividades no processo de desenvolvimento; (2) ambientes centrados em linguagem, que são ambientes de apoio à codificação em uma dada linguagem de programação e que, em geral, permitem customizações e extensões nessa linguagem; (3) ambientes integrados, que possuem mecanismos padronizados para a integração de novas ferramentas, oferecendo ainda uma interface única e consistente ao usuário e integração de dados; (4) ambientes de "quarta geração", que dão suporte específico ao desenvolvimento de sistemas de processamento eletrônico de dados e aplicações orientadas a negócios, utilizando as chamadas linguagens de "quarta geração"; e (5) ambientes centrados em processo.

Os **ambientes centrados no processo** são baseados em uma definição formal do processo de desenvolvimento utilizado [25]. Um modelo do processo indica quais atividades devem ser executadas, por quem elas devem ser executadas, com quais ferramentas e o formato dos documentos a serem produzidos. O ambiente interpreta esse modelo e, com base nele, pode guiar desenvolvedores, automatizar tarefas e executar ferramentas [7]. Desse modo, o ferramental se adequaria ao processo de desenvolvimento utilizado. Entretanto, quanto mais flexíveis são os ambientes centrados no processo, menor é o suporte possível a atividades específicas de desenvolvimento.

Podemos citar como vantagens do uso de um ambiente de desenvolvimento integrado como sendo o fato de que tais ambientes podem: garantir um alto nível de consistência entre os documentos produzidos, guiar os desenvolvedores de software através do processo suportado, ter o conhecimento detalhado do estado do processo de software e automatizar partes do processo [7]. Ainda, as motivações para o uso dessas ferramentas são: o suporte a uma metodologia padrão para o desenvolvimento de sistemas, a existência de um repositório centralizado que assegure o uso da metodologia utilizada e a melhoria em termos de qualidade e segurança do processo de desenvolvimento [1]. Os obstáculos para a adoção de ferramentas CASE são: sua limitada capacidade (podem não possuir ferramentas para cobrir todo o processo de desenvolvimento e podem não se adequar a todos os processos), os poucos benefícios tangíveis que eles provêem, do ponto de vista econômico e o seu alto custo. Ainda podem existir questões culturais entre os desenvolvedores [1].

2.6 O Ambiente Eclipse

O Eclipse [22] é um ambiente de desenvolvimento integrado de software (IDE) *open source*. Ele foi concebido como uma plataforma aberta e expansível através de módulos anexáveis chamados *plug-ins*. Assim, tanto suas funcionalidades básicas quanto as inúmeras ferramentas disponíveis de forma livre ou comercial são estruturadas como *plug-ins*. Desse modo, o Eclipse se torna uma plataforma de integração de ferramentas, através da qual é possível adicionar funcionalidades como, por exemplo, suporte a várias linguagens de

programação [19, 20] ou mesmo modelagem de diagramas UML [55].

Desenvolvido na linguagem de programação Java [30], ele possui um amplo leque de ferramentas para suporte ao desenvolvimento nesta linguagem, que inclui um conjunto de *plug-ins* chamado de *Java Development Tools* (JDT), editores de código, ferramentas de depuração, testes unitários e montagem. Ainda, o ambiente possui suporte ao desenvolvimento de seus próprios *plug-ins*, chamado *Plug-in Development Environment* (PDE), o que favorece o surgimento de novas ferramentas.

Distribuído de forma gratuita, foi originalmente desenvolvido pela empresa IBM, e atualmente é um projeto mantido por um consórcio de empresas com grandes nomes da indústria, entre elas Borland, IBM, Rational, Red Hat, Oracle, SAP e a OMG — Object Management Group, grupo responsável pela especificação do UML, CORBA, entre outros. Graças a essas qualidades, o número de usuários do projeto Eclipse têm crescido.

O **EMF** (do inglês *Eclipse Modeling Framework*) [23] é um sub-projeto do Eclipse que provê uma plataforma de modelagem e geração de código para a construção de aplicações a partir de um modelo de dados estruturado. Assim, a partir da especificação de um modelo, que pode ser descrito em formato XMI criado por ferramentas como Rational Software Modeler [35] e Omondo EclipseUML [55], o EMF possui um conjunto de ferramentas que permite a criação de código na linguagem Java que implementa esse modelo. A implementação do modelo gerado não depende do Eclipse em tempo de execução, e pode ser utilizada em qualquer aplicação Java. O modelo inclui infra-estruturas como *Abstract Factory*, registro de observadores (*listeners*), rotinas de persistência e os tipos são definidos como interfaces.

O **GEF** (do inglês *Graphical Editing Framework*) [24] é um sub-projeto do Eclipse que permite a criação de editores gráficos a partir do modelo de uma aplicação. O GEF utiliza o estilo arquitetural MVC, permitindo maior flexibilidade às aplicações. Ele provê infra-estrutura para a criação de editores, incluindo interfaces e super-classes de editores do Eclipse, classes para materializar os elementos de controle e edição. Ainda, o GEF inclui o sub-projeto **Draw2D**, que oferece infra-estrutura de diagramação e desenho para materializar os elementos de visão, incluindo figuras comuns que podem ser agrupadas para o desenho de diferentes aplicações.

2.7 Resumo

Este capítulo apresentou conceitos e termos importantes no contexto deste trabalho. **Arquitetura de software** é uma abstração da estrutura do software. Essa estrutura é formada por componentes arquiteturais, unidades que realizam processamento, e conectores, que materializam a interação entre os componentes. Uma dada organização de componentes e conectores interligados entre si é denominada sua configuração arquitetu-

ral.

Um **componente de software** é uma unidade independente de desenvolvimento, implementação e implantação, com alto encapsulamento, alta coesão e baixo acoplamento, sujeito à composição (integração em aplicações) por terceiros. Um componente declara os serviços que oferece através de interfaces providas e os serviços dos quais depende, através de interfaces requeridas. Um componente, em geral, é desenvolvido sem o conhecimento do contexto em que ele será integrado. Assim, a arquitetura de software provê esse contexto no momento da composição de aplicações. Um processo DBC pode ser dividido em desenvolvimento de componentes (desenvolvimento para reuso) e integração de aplicações (desenvolvimento com reuso). O **UML Components** é um processo de DBC que foca especialmente na especificação de componentes. Um modelo de componentes define um padrão no formato dos componentes e na sua comunicação. O **COSMOS** é um modelo de implementação de componentes que define uma série de regras que possibilita a materialização da arquitetura de software em código utilizando estruturas presentes nas principais linguagens de programação orientadas a objetos.

Para se obter produtividade no desenvolvimento de software, se recorre ao uso de **ferramentas CASE**. Essas ferramentas apóiam tarefas específicas do processo de desenvolvimento de software. Visando uma maior integração entre as ferramentas, foram criados *workbenches* e **ambientes**, que agregam tais ferramentas dando um suporte a várias atividades do processo. Os ambientes integrados possuem mecanismos padronizados para a integração de novas ferramentas, oferecendo ainda uma interface única e consistente ao usuário e integração de dados. Um ambiente ainda pode ser **centrado no processo**, de modo que, dado uma representação formal do processo, ele pode se adequar ao mesmo, guiando os desenvolvedores e automatizando partes do processo. O **Eclipse** é um ambiente integrado de desenvolvimento construído de modo a ser uma plataforma de integração de ferramentas. Todas as suas funcionalidades são construídas como *plug-ins*. O Eclipse oferece apoio ao desenvolvimento em diversas linguagens de programação, em especial Java, além de seus próprios *plug-ins*. O **EMF**, *Eclipse Modeling Framework*, provê infra-estrutura para a criação de modelos em código Java a partir de uma especificação do mesmo em XMI. Esse modelo é independente do Eclipse e possui características como notificação, programação por interfaces e *Abstract Factory*. O **GEF**, *Graphical Editing Framework*, provê uma plataforma para a criação de editores gráficos no Eclipse. Ele utiliza o estilo MVC, fornecendo infra-estrutura para os elementos de visão e controle. No próximo capítulo, os requisitos necessários para um ambiente integrado de desenvolvimento com apoio ao DBC e centrado na arquitetura de software são apresentados.

Capítulo 3

Requisitos para um Ambiente com Apoio ao DBC

Neste capítulo são descritos os trabalhos realizados para o levantamento dos requisitos do ambiente Bellatrix. Na Seção 3.1 são descritos os requisitos que o ambiente deve dar apoio, incluindo trabalhos relacionados e anteriores que detalham requisitos para ambientes de DBC. A Seção 3.2 consolida os requisitos para o ambiente, identificando seus requisitos funcionais e não funcionais. Na Seção 3.3, os casos de uso identificados para o ambiente são apresentados, bem como o processo utilizado para identificá-los. Na Seção 3.4 são mostrados os trabalhos relacionados com os protótipos de interface com o usuário. Por fim, a Seção 3.5 sumariza este capítulo.

3.1 Requisitos

O ambiente Bellatrix é um ambiente para apoio ao DBC e centrado na arquitetura de software, conforme conceitos vistos nas Seções 2.2 e 2.1, respectivamente. Assim, o escopo desse trabalho inclui a especificação, implementação e testes de componentes, e a especificação de arquiteturas de software e estilos arquiteturais. Com o objetivo de se levantar as funcionalidades às quais o ambiente Bellatrix deveria dar suporte, realizou-se uma pesquisa envolvendo trabalhos relacionados, em especial aqueles que tratam de requisitos para ambientes de DBC. Dentre esses trabalhos, pode-se citar a proposta inicial do ambiente Bellatrix feita por Guerra [31], o ambiente WREN, proposto por Lüer *et al.* [42] e o ambiente Orion, proposto por Lucrédio *et al.* [40]. As próximas seções apresentam os requisitos identificados por esses trabalhos.

3.1.1 Requisitos Identificados por Guerra

A proposta de Guerra [31] lista as seguintes atividades como sendo requisitos básicos num ambiente de DBC:

1. Modelagem de arquiteturas de software em dois níveis de abstração: linhas de produtos e sistemas específicos (Guerra 1);

2. Especificação de interfaces, componentes e conectores (Guerra 2);

3. Implementação de componentes e conectores, com a geração automática de código a partir do projeto arquitetural (Guerra 3);

4. Testes de unidade de componentes e conectores (Guerra 4);

5. Criação, manutenção, versionamento e busca dos artefatos produzidos, utilizando um repositórios de componentes (Guerra 5);

6. Integração de sistemas, a partir de componentes e conectores armazenados nos repositórios de componentes (Guerra 6);

7. Testes de integração de configurações de componentes e conectores (Guerra 7).

Ainda, a proposta inicial do ambiente Bellatrix enumera as seguintes diretrizes para o desenvolvimento do projeto, que também são considerados como requisitos:

1. O ambiente deve ser desenvolvido utilizando ferramentas multi-plataformas, preferencialmente a linguagem Java (Guerra 8);

2. O ambiente deve disponibilizar seus fontes, seguindo um modelo de desenvolvimento *open source* (Guerra 9);

3. Devem ser utilizadas ferramentas e componentes já existentes, quando possível. Ainda, a integração com a plataforma Eclipse deve ser priorizada (Guerra 10);

4. O ambiente deve ser centrado no processo de desenvolvimento, baseado inicialmente no processo UML Components. O ambiente deve permitir a sua adaptação e extensão para adequar-se a outros processos de desenvolvimento DBC (Guerra 11).

3.1.2 Requisitos Identificados por Lüer *et al.*

O trabalho de Lüer *et al.* [42] identifica sete requisitos para ambientes de DBC, alguns destes presentes em soluções comerciais, outros não adotados e talvez até controversos. Alguns requisitos para o ambiente incluem características do modelo de componentes e do repositório definidos também no contexto do trabalho. A lista a seguir mostra esses requisitos:

1. O ambiente deve dar apoio à separação entre as partes pública e privada do componente, seguindo o princípio de encapsulamento. No modelo de componentes proposto por Lüer *et al.*, a parte pública consiste num mecanismo de "auto-descrição", através do qual o componente expõe principalmente suas interfaces providas e requeridas, um mecanismo de instanciação, através do qual clientes do componente podem obter instâncias das interfaces providas e interfaces públicas definidas pelo componente (Lüer 1);

2. O ambiente deve dar apoio e explorar a "auto-descrição" dos componentes, ou seja, meta-informação que o componente carrega em si, de modo a possibilitar e facilitar o seu reuso. Em especial, as interfaces providas e requeridas pelo componente fazem parte dessa informação (Lüer 2);

3. Os componentes devem ser definidos e acessados através de suas interfaces. Ainda, essas interfaces devem estar definidas dentro de um espaço de nomes global, onde duas interfaces com mesmo nome são supostamente equivalentes (Lüer 3);

4. O ambiente deve dar apoio ao desenvolvimento de componentes e à integração de componentes (Lüer 4);

5. O ambiente deve dar apoio à integração de componentes através de conexão (simples ligação entre serviços requeridos e providos) e adaptação de componentes (Lüer 5);

6. O ambiente deve dar apoio a múltiplas visões, incluindo visões para o desenvolvimento e composição de componentes (Lüer 6);

7. O ambiente deve dar suporte ao reuso por referência, que consiste em uma única cópia do componente armazenada em um repositório e referenciada pela Internet (Lüer 7). Cópias podem existir localmente apenas com o propósito de *cache*. Quando o componente é atualizado no repositório, essa atualização é propagada automaticamente aos clientes.

3.1.3 Requisitos Identificados por Lucrédio *et al.*

O trabalho de Lucrédio *et al.* [40] cita como principais requisitos para um ambiente de engenharia de software baseado em componentes:

1. Integração das ferramentas que compõem o ambiente, incluindo o uso de uma plataforma comum, compartilhamento de dados, mesma apresentação ao usuário, controle integrado das ferramentas e existência de processo guiando o uso das mesmas (Lucrédio 1);

2. Apoio a atividades de engenharia de software baseada em componentes, incluindo o desenvolvimento de componentes "para reuso" e desenvolvimento de aplicações "com reuso" (Lucrédio 2);

3. Reusabilidade, ou seja, o ambiente deve promover o reuso de artefatos e evitar duplicação de esforços (Lucrédio 3);

4. Integridade referencial, garantindo que os artefatos produzidos sejam consistentes entre si (Lucrédio 4);

5. Gestão de configuração de software, oferecendo controle sobre as mudanças e versões dos artefatos produzidos (Lucrédio 5);

6. Múltiplas visões da informação, em especial a visão interna e externa do componente, respectivamente sua arquitetura interna e suas interfaces (Lucrédio 6);

7. Segurança no acesso, de modo que somente desenvolvedores autorizados são capazes de acessar e modificar as informações (Lucrédio 7);

8. Independência de tecnologia e linguagem de programação (Lucrédio 8).

3.2 Consolidação dos Requisitos para o Bellatrix

Pode-se notar, a partir dos trabalhos descritos nas seções anteriores, a diversidade do que são considerados requisitos para um ambiente de DBC. Na proposta de Guerra, os requisitos são atividades do processo de DBC que desenvolvedores devem ser capazes de realizar. No trabalho de Lüer *et al.*, os requisitos do ambiente mesclam-se com características do modelo de componentes e do repositório definidos no mesmo trabalho. Já o trabalho de Lucrédio *et al.* é mais abrangente, descrevendo inclusive alguns requisitos não-funcionais para o ambiente. Ainda, existem alguns requisitos que são equivalentes. Na Tabela 3.1, são apresentados os requisitos para o ambiente Bellatrix, indicando a referência para os mesmos nas listas das seções anteriores.

Tabela 3.1: Requisitos para o ambiente BELLATRIX

Requisito	Requisitos relacionados			Apoiado?
Guerra 1 (modelagem de arquiteturas)	Lüer 4	Lucrédio 2		sim
Guerra 2 (especificação de componentes)	Lüer 4	Lucrédio 2	Lüer 5	sim
Guerra 3 (implementação de componentes)	Lüer 4	Lucrédio 2	Lüer 5	sim
Guerra 4 (testes de componentes)	Lüer 4	Lucrédio 2	Lüer 5	sim
Guerra 5 (repositório de componentes)	Lüer 4	Lucrédio 2	Lucrédio 5	sim
Guerra 6 (integração de sistemas)	Lüer 4	Lucrédio 2		sim
Guerra 7 (testes de integração)	Lüer 4	Lucrédio 2		sim
Guerra 8 (multi-plataforma)				sim
Guerra 9 (código aberto)				sim
Guerra 10 (integração com Eclipse)				sim
Guerra 11 (centrado no processo)				sim
Lüer 1 (encapsulamento)				parcial
Lüer 2 (meta-informação)				parcial
Lüer 3 (acesso via interfaces)				parcial
Lüer 4 (desenvolvimento e integração de componentes)	Lucrédio 2			sim
Lüer 5 (conexão e adaptação)	Guerra 2	Guerra 3	Guerra 4	sim
Lüer 6 (múltiplas visões)	Lucrédio 6			sim
Lüer 7 (reuso por referência)				não
Lucrédio 1 (integração de ferramentas)				sim
Lucrédio 2 (desenvolvimento "com" e "para reuso")	Lüer 4			sim
Lucrédio 3 (promover reuso)				sim
Lucrédio 4 (integridade referencial)				sim
Lucrédio 5 (gestão de configuração)	Guerra 5			sim
Lucrédio 6 (múltiplas visões)	Lüer 6			sim
Lucrédio 7 (segurança no acesso)				não
Lucrédio 8 (independência de tecnologia)				sim

3.2.1 Lista de Requisitos Apoiados pelo Ambiente Bellatrix

Apresenta-se a seguir a lista que sumariza os requisitos apoiados pelo ambiente Bellatrix, conforme a Tabela 3.1:

- O ambiente deve dar apoio ao desenvolvimento baseado em componentes "para reuso" (desenvolvimento de componentes) e "com reuso" (desenvolvimento de aplicações ou integração de componentes).
- O ambiente deve permitir a modelagem de arquiteturas de software e o uso de estilos arquiteturais.
- O ambiente deve permitir a especificação e implementação de interfaces, componentes e conectores, possibilitando a geração automática de código a partir do projeto arquitetural.
- O ambiente deve dar apoio aos testes de unidade de componentes e conectores, bem como a testes de integração de configurações.
- O ambiente deve dar apoio à gestão de configuração de software, ou seja, manutenção, versionamento e busca de artefatos em um repositório de componentes.
- O ambiente deve ser centrado no processo de desenvolvimento, permitindo ainda sua adaptação e extensão para adequar-se a diferentes processos de desenvolvimento.

3.2.2 Lista de Requisitos Não Apoiados pelo Ambiente Bellatrix

Na lista a seguir, apresentamos requisitos não apoiados ou parcialmente apoiados pelo ambiente Bellatrix e sobre eles, apresentam-se justificativas.

- o requisito Lüer 1 (encapsulamento) é parcialmente apoiado no sentido de que os componentes devem ser encapsulados, possuindo uma visão externa e uma visão interna. Entretanto, os detalhes do modelo de componentes tratado no trabalho não serão apoiados;
- o requisito Lüer 2 (explorar meta-informação) é parcialmente apoiado pois, caso seja possível a extração de meta-informação a partir da implementação do componente, sua especificação pode ser extraída. No entanto, podem haver modelos de componentes não permitam essa atividade.

- o requisito Lüer 3 (acesso via interfaces) é parcialmente apoiado pois depende do modelo de implementação de componentes. Nos modelos mais utilizados, tais como J2EE e .Net, e mesmo no modelo COSMOS, as interfaces seguem as regras descritas. Entretanto, podem haver modelos de componentes com características diferentes. Em termos de especificação, uma interface é definida de modo independente do componente que o provê ou requer. Assim, podem haver duas interfaces possuindo coincidentemente o mesmo nome. Esses casos são tratados com o uso de identificadores internos e no momento da materialização dos componentes.

- o requisito Lüer 7 (reuso por referência) não é apoiado pois essa estratégia, comparável com a ligação dinâmica de bibliotecas, é mais complexa e exige uma conexão permanente com o repositório [43]. Ainda, ele assume um modelo de evolução simples, com compatibilidade com versões anteriores e atualização automática para novas versões. Essas características podem não ser interessantes dependendo dos interesses comerciais de desenvolvedores e integradores.

- o requisito Lucrédio 7 (segurança no acesso) não é apoiado diretamente; espera-se que tal característica seja atingida através do repositório de componentes;

- O ambiente BELLATRIX não dará apoio à modelagem de diagramas UML, visto a existência e diversidade de ferramentas disponíveis que o fazem.

3.2.3 Atributos de Qualidade

Os seguintes atributos de qualidade, ou requisitos não-funcionais, do ambiente BELLATRIX foram identificados e sumarizados a seguir:

1. Portabilidade: o ambiente deve ser capaz de ser executado em diferentes sistemas operacionais, de modo a não restringir seus potenciais usuários. Em especial, os sistemas operacionais Windows e Linux devem ser contemplados.

2. Usabilidade: o ambiente deve ser fácil de ser compreendido e de ser aprendido por usuários familiarizados com os conceitos de DBC. Ele deve ser acompanhado de documentação e sistemas de ajuda, diminuindo a necessidade de suporte direto.

3. Manutenibilidade: o ambiente deve ser de fácil manutenção, tanto corretiva — para a correção de falhas em seu funcionmento — quanto evolutiva — para a inclusão ou alteração de suas funcionalidades.

4. Desempenho: o ambiente é uma aplicação interativa e deve ter desempenho compatível para ferramentas dessa natureza, em equipamentos que são utilizados por

desenvolvedores de sistemas. Será considerado o tempo de resposta de no máximo 2 segundos para as atividades interativas.

5. Integração: as ferramentas que compõem o ambiente devem estar integradas, em relação a sua funcionalidade, modelo de dados utilizado e interface com o usuário. Ainda, a adição de novas ferramentas não deve ser complexa.

6. Código aberto: o código-fonte do ambiente deve estar disponível publicamente ou a grupos de usuários, de modo a possibilitar a sua extensão e contribuições de terceiros.

3.2.4 Restrições de Implementação

De acordo com os requisitos funcionais e não-funcionais sumarizados nas seções anteriores, identificou-se as seguintes restrições de implementação:

- A linguagem de programação a ser utilizada é Java. Isso atende aos requisitos de portabilidade.

- O ambiente utilizará a plataforma Eclipse para seu desenvolvimento. Isso atende aos requisitos de portabilidade, integração e código-aberto. Ainda, a plataforma é desenvolvida na linguagem Java.

- O desenvolvimento do ambiente deve ser baseado em componentes. Isto melhora a manutenibilidade e a integração do ambiente.

- Devido ao uso da plataforma Eclipse, a arquitetura a ser adotada deve seguir as restrições impostas pela plataforma.

A plataforma Eclipse foi escolhida por apresentar uma série de vantagens para o desenvolvimento e o uso do ambiente final. O Eclipse é uma plataforma de código aberto para integração de ferramentas. Sua tecnologia de *plug-ins* permite a instalação e integração de ferramentas desenvolvidas individualmente, de modo que ele pode ser considerado um ambiente para composição [41].

Nesse sentido, foram identificadas as seguintes características do Eclipse como sendo favoráveis a esse uso [41]: (1) arquiteturas multi-nível, ou seja, os *plug-ins* do Eclipse podem não só estender a própria plataforma mas também outros *plug-ins*; (2) portas explícitas, de modo que um *plug-ins* declara quais são seus pontos de conexão com a plataforma e outros *plug-ins*; (3) auto-descrição, pois cada *plug-ins* possui um arquivo descritor, separado do código, que contém informações sobre como o *plug-in* estende o

ambiente; e (4) encapsulamento dos componentes, em que o Eclipse garante o encapsulamento e o isolamento dos *plug-ins*, de modo que somente são acessíveis os *plug-ins* declarados como dependência no arquivo descritor. Ainda, são vantagens do Eclipse sua interface com o usuário estensível de modo transparente ao usuário, sua ampla adoção por desenvolvedores e o fato de ser distribuído gratuitamente. O Eclipse é um projeto de código aberto, que traz também como vantagens o fato de que ele pode ser modificado, seu código pode ser examinado quanto a sua corretude e seu oferecimento potencialmente não será interrompido.

3.3 Casos de Uso

O próprio desenvolvimento do ambiente Bellatrix foi baseado em componentes e seguiu o processo UML Components (Seção 2.3). Assim, uma das primeiras atividades a ser realizada é a modelagem do processo de negócios, que pode ser feita utilizando-se diagramas de atividades UML. O ambiente deve dar apoio ao processo de DBC, de modo que o processo de desenvolvimento é o processo de negócios do Bellatrix. Inicialmente, considerou-se o apoio ao processo UML Components, de modo que foi feito um mapeamento detalhado de suas atividades com base em sua bibliografia de referência [15], visando oferecer um apoio adequado ao processo. Nesse mapeamento, o processo foi modelado num conjunto de diagramas de atividades UML, como o modelo do processo de negócios do ambiente. A Figura 3.1 mostra um desses diagramas. Esse diagrama contém atividade de identificação de interfaces de sistema e operações, da subfase de identificação de componentes, descrita em maiores detalhes na Seção 2.3. Nessa fase, para cada caso de uso se cria uma interface de sistema e uma operação é criada nessa interface para cada passo desse caso de uso.

A modelagem do processo, em conjunto com o modelo conceitual do negócio — apresentado em maiores detalhes na Seção 4.2 — foi utilizada para dar origem aos casos de uso identificados da seguinte forma: para cada atividade que poderia ser apoiada ou automatizada pelo ambiente, criou-se um caso de uso. Ainda, casos de uso foram criados para cada elemento e relacionamento do modelo conceitual que necessitasse ser gerenciado, por exemplo, criado, excluído, entre outros. Os casos de uso foram organizados em pacotes de acordo com a concepção inicial do ambiente [31], facilitando o gerenciamento, visualização dos diagramas e a descrição dos casos de uso. A Figura 3.2 mostra o diagrama de casos de uso.

A seguir, descrevemos os elementos desse diagrama:

1. **Especificar interface.** Neste pacote estão agrupados os casos de uso referentes às atividades de especificação de interfaces externas dos componentes de uma arquitetura de software.

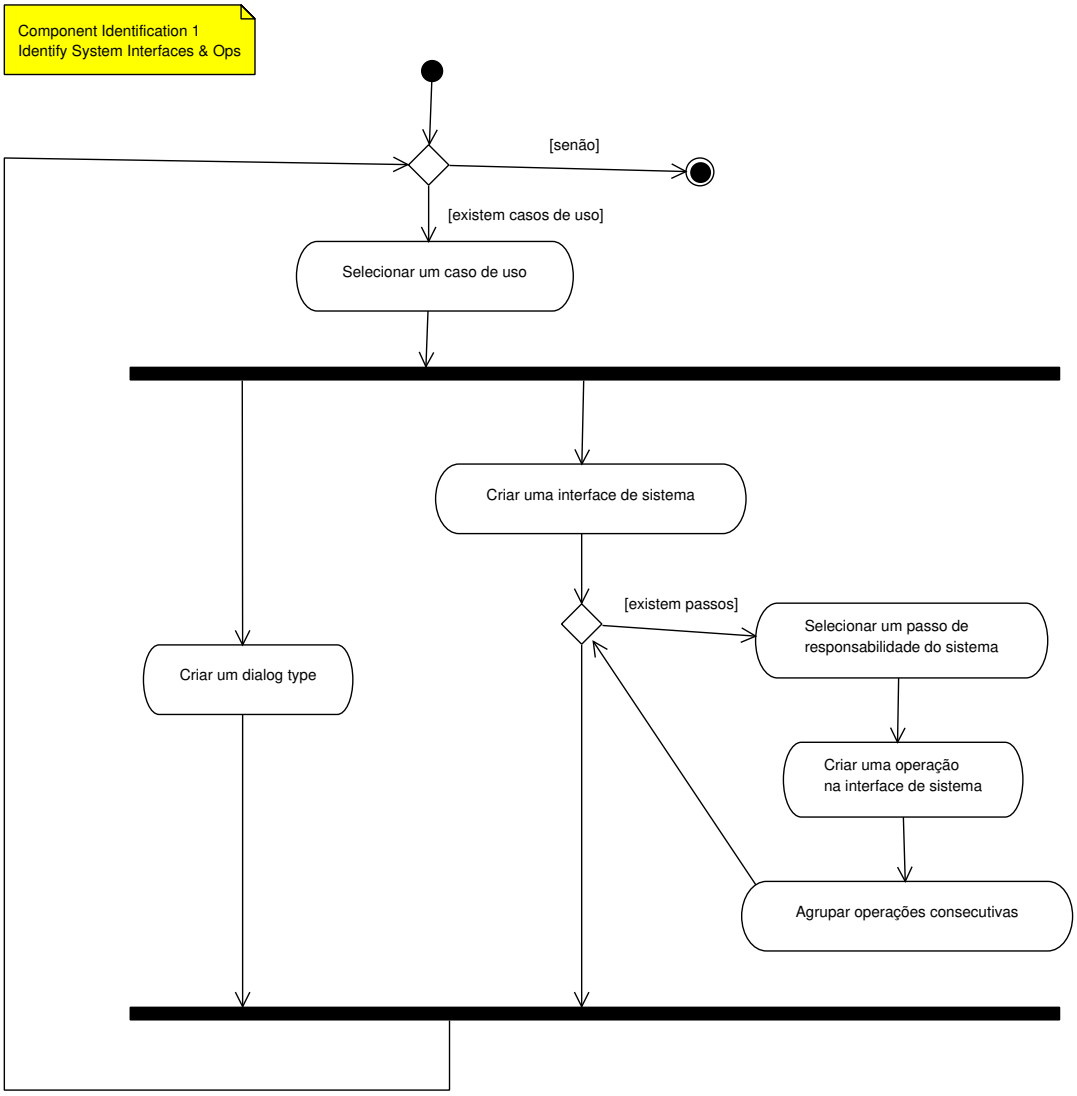

Figura 3.1: Diagrama de atividades do funcionamento do processo UML Components.

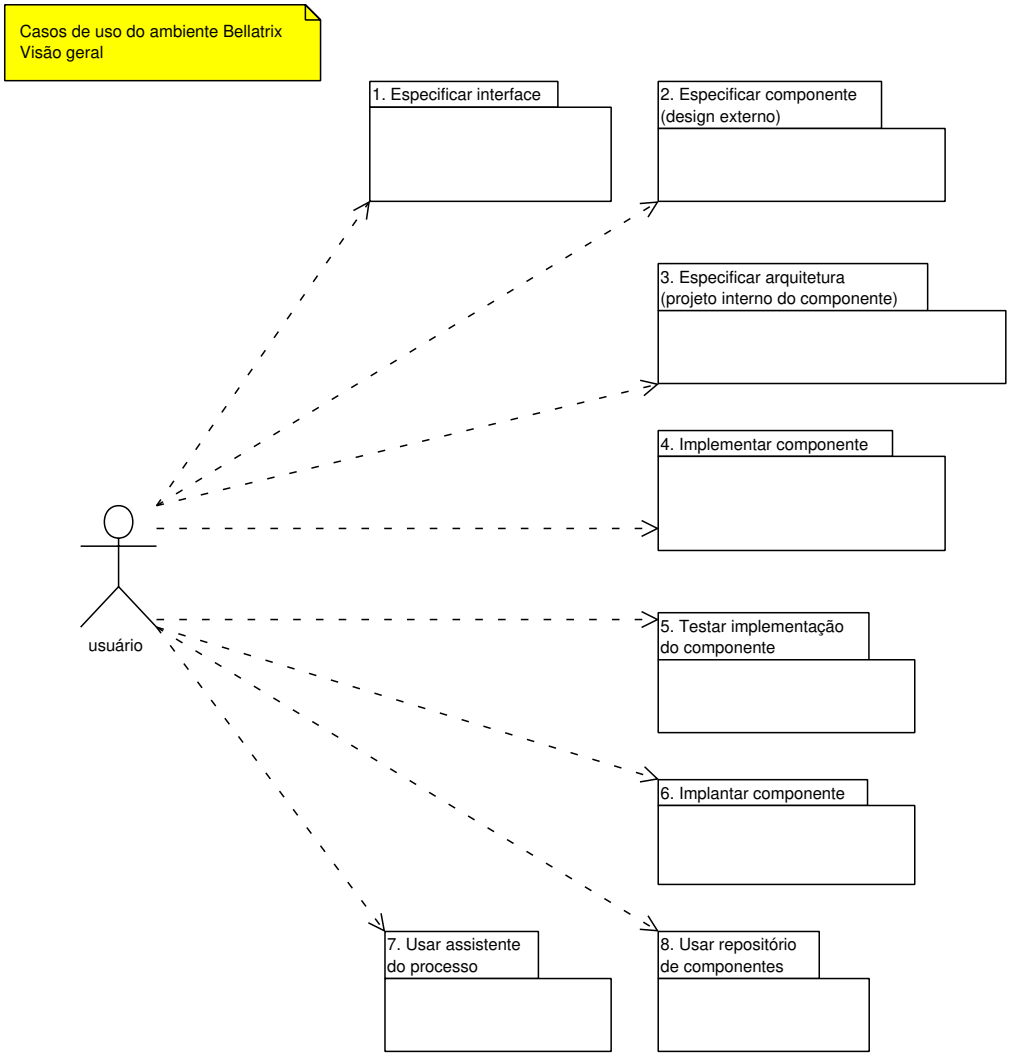

Figura 3.2: Visão geral dos casos de uso do ambiente.

2. **Especificar componente.** Neste pacote estão os casos de uso referentes às atividades de especificação de componentes e conectores, mais especificamente, seu projeto "externo".

3. **Especificar arquitetura.** Neste pacote estão os casos de uso referentes às atividades de modelagem da arquitetura de componentes, mais especificamente, o projeto "interno" de um componente.

4. **Implementar componente.** Neste pacote estão os casos de uso referentes às atividades de implementação dos componentes especificados no ambiente.

5. **Testar implementação do componente.** Neste pacote estão os casos de uso referentes às atividades de especificação de casos teste de um componente.

6. **Implantar componentes.** O usuário pode implantar (*deploy*) a implementação dos componentes.

7. **Usar assistente do processo.** Neste pacote estão os casos de uso relacionados com o apoio ao processo UML Components.

8. **Usar repositório de componentes.** Neste pacote estão os casos de uso básicos relacionados com o uso do repositório a partir do ambiente.

Para cada pacote, um ou mais diagramas de casos de uso foram criados, de modo a detalhá-lo. A Figura 3.3 mostra o diagrama de casos de uso do pacote Especificar Componente (2).

Ainda, cada caso de uso foi descrito textualmente [60]. Um exemplo é mostrado na Tabela 3.2.

A modelagem do processo UML Components, bem como a descrição completa dos casos de uso, foi publicada na forma de um relatório técnico [69].

3.4 Protótipos de Interface com o Usuário

Protótipos de interface gráfica com usuário (interfaces humano-computador ou IHC) foram montados com o objetivo de detalhar os casos de uso identificados e validá-los. Tais protótipos foram inicialmente desenvolvidos em papel, apresentados para discussões com o grupo de pesquisas e com potenciais usuários da empresa Autbank[1] e posteriormente digitalizados. As principais vantagens dessa abordagem são a maior participação do usuário,

[1] A Autbank Projetos e Consultoria Ltda é uma empresa de médio porte que produz sistemas para a área financeira. Ela adota o DBC na construção de seus produtos.

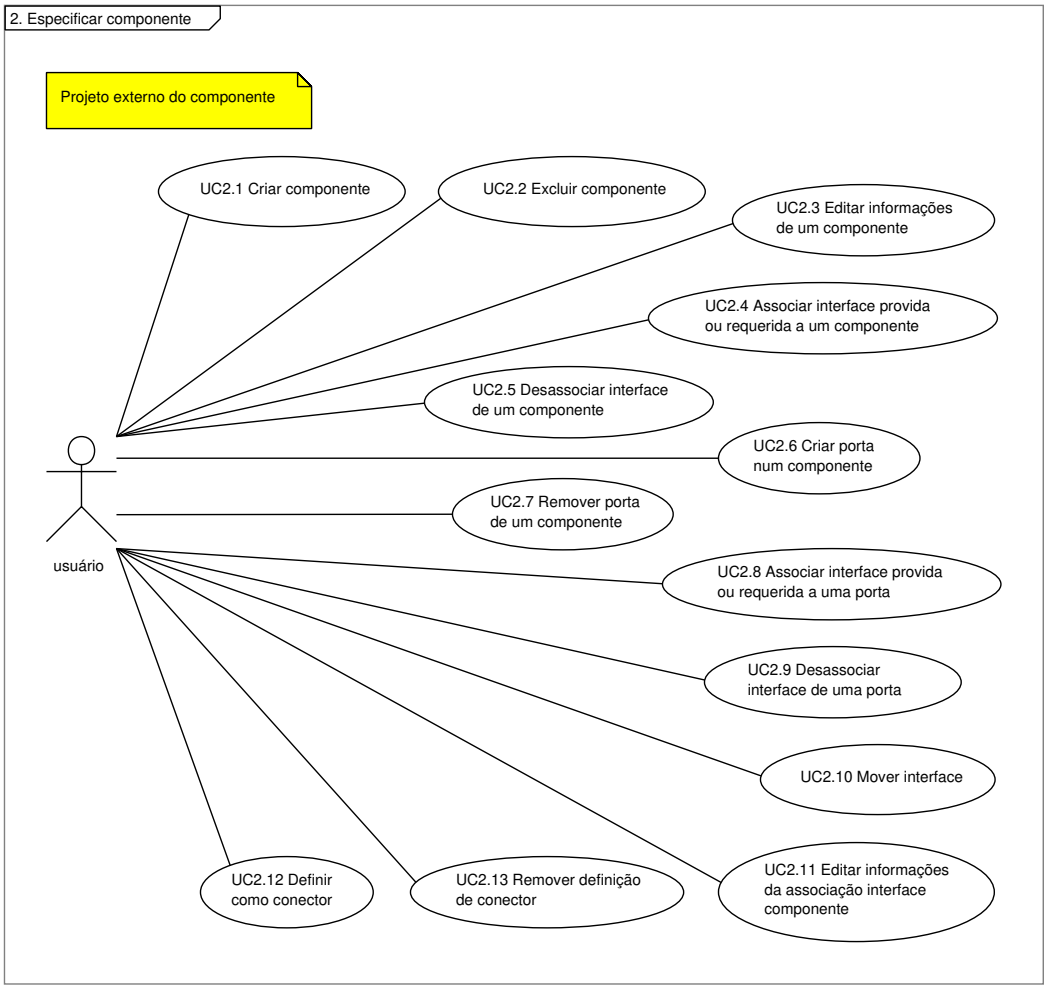

Figura 3.3: Diagrama de casos de uso do pacote Especificar Componente.

auxiliando tanto o projeto de IHC propriamente dito, quanto a extração de requisitos e seu registro. Ainda, os protótipos procuraram exercitar os casos de uso do ambiente, através de cenários de uso.

Organizados de modo similar aos casos de uso, os protótipos montados foram divididos em grupos: protótipos relacionados à especificação de interfaces, à especificação de componentes, protótipos de especificação de arquiteturas, implementação de componentes, assistentes para o processo UML Components e relativos à interface com o repositório de componentes a partir do ambiente. Esses protótipos foram publicados na forma de um relatório técnico [69]. A Figura 3.4 apresenta um exemplo de um protótipo, mostrando o caso de uso para associar uma interface como provida ou requerida de um componente, durante a edição da especificação desse componente.

Tabela 3.2: Exemplo de descrição de caso de uso.

Associar interface provida ou requerida a uma porta (UC2.8)

Descrição: O usuário associará uma especificação de interface já existente à especificação do componente através de uma porta que esse componente possua. Essa interface pode ser associada como uma interface provida ou requerida. Uma porta pode ter várias interfaces associadas a ela. O componente em questão deve estar aberto para edição.
Passos:

1. O caso de uso se inicia quando o usuário seleciona uma porta na especificação do componente e escolhe a opção de associar uma interface à porta.
2. O sistema exibe uma lista das interfaces que estão contidas na área de trabalho do ambiente. Devem haver meios para filtrar essa lista.
3. O usuário seleciona a interface e informa se ela é uma interface provida ou requerida.
4. O sistema atualiza a representação gráfica do componente para exibir a porta com a interface associada. A especificação do componente deve persistir essa associação.

3.5 Resumo

Trabalhos para elucidação dos requisitos do ambiente Bellatrix foram realizados com o intuito de produzir uma especificação completa do ambiente, levantando as atividades às quais ele deveria dar suporte. Inicialmente, buscou-se trabalhos relacionados e anteriores que descrevessem requisitos para ambientes DBC. Consolidada a lista de requisitos, um conjunto de diagramas de atividades UML representando o processo UML Components foi criado. Esses diagramas, em conjunto com o modelo conceitual do negócio — apresentado em maiores detalhes na Seção 4.2 — formam o modelo de negócios do ambiente. A partir desse modelo de negócios, os casos de uso do ambiente foram identificados. Por fim, foram montados protótipos de interfaces com o usuário para detalhar e validar os requisitos e os casos de uso identificados. Uma vez identificados os requisitos para o ambiente Bellatrix, o projeto do ambiente, também baseado em componentes, foi desenvolvido e é apresentado no Capítulo 4.

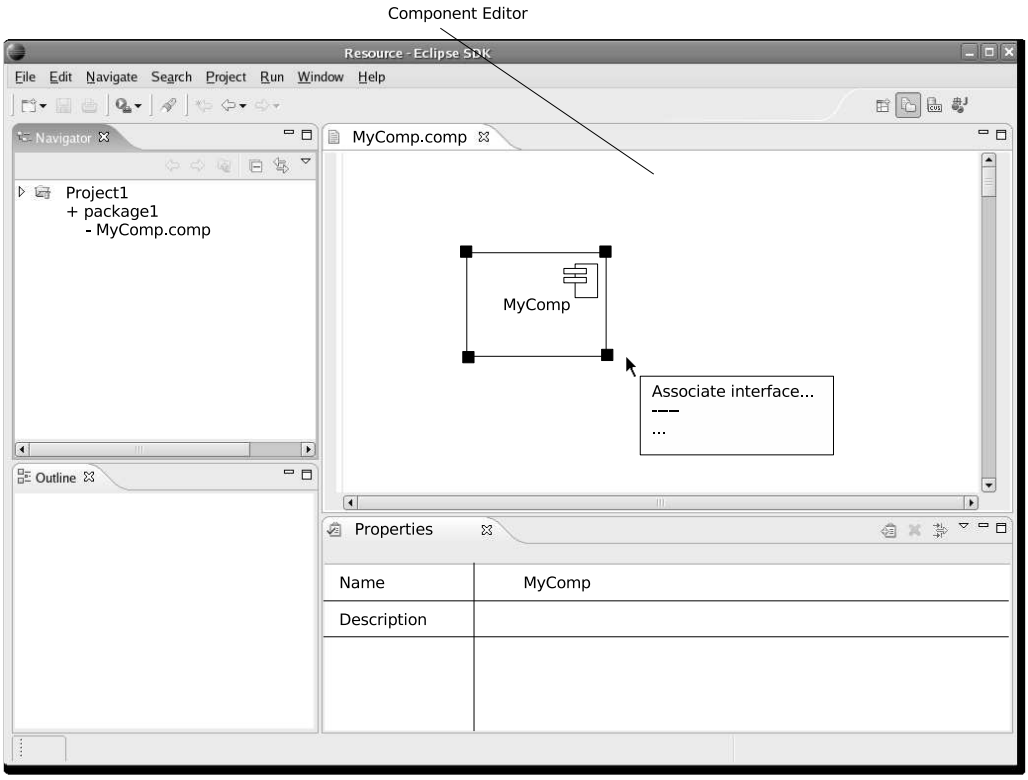

Figura 3.4: Protótipo de interface com o usuário para associar uma interface à especificação de um componente.

Capítulo 4

Projeto do Ambiente Bellatrix

Esse capítulo descreve o projeto do ambiente Bellatrix. A Seção 4.1 mostra a arquitetura inicial do ambiente. A Seção 4.2 apresenta o modelo conceitual (ou modelo de classes) utilizado no ambiente. A Seção 4.3 detalha a especificação dos componentes que compõem o Bellatrix, incluindo como eles foram especificados. Na Seção 4.4, a arquitetura de componentes do ambiente é mostrada e na Seção 4.5 fazemos um sumário deste capítulo.

4.1 Arquitetura Inicial do Ambiente

A Seção 1.2 descreveu em linhas gerais o ambiente Bellatrix. Em particular, a Figura 1.1 apresentou uma visão geral do ambiente, na qual sua arquitetura inicial foi baseada. A Figura 4.1 mostra novamente essa visão geral, para fins de facilidade de referência.

Conforme visto na Seção 3.2, considerou-se importante para o desenvolvimento do ambiente Bellatrix que sua implementação fosse fácil de ser mantida e estendida. Assim, o próprio ambiente é baseado em componentes e foi especificado seguindo o processo UML Components. Um dos requisitos para o ambiente é a integração com o ambiente Eclipse, devido a sua facilidade de extensão e integração com novas ferramentas. Esses fatores guiaram a criação da arquitetura do sistema.

O processo UML Components assume a utilização de uma arquitetura no estilo em camadas [14], conforme visto na Seção 2.3. Essas camadas são: (1) a camada de interação com o usuário; (2) a camada de serviços de sistema e; (3) a camada de serviços de negócios, conforme mostra a Figura 2.3. As tecnologias do Eclipse utilizadas, em especial o GEF [24], descrito em maiores detalhes na Seção 2.6, impõe uma arquitetura seguindo o estilo MVC [14] (Figura 2.1) às aplicações desenvolvidas. Essas tecnologias e suas aplicações na implementação do ambiente Bellatrix são descritas em maiores detalhes

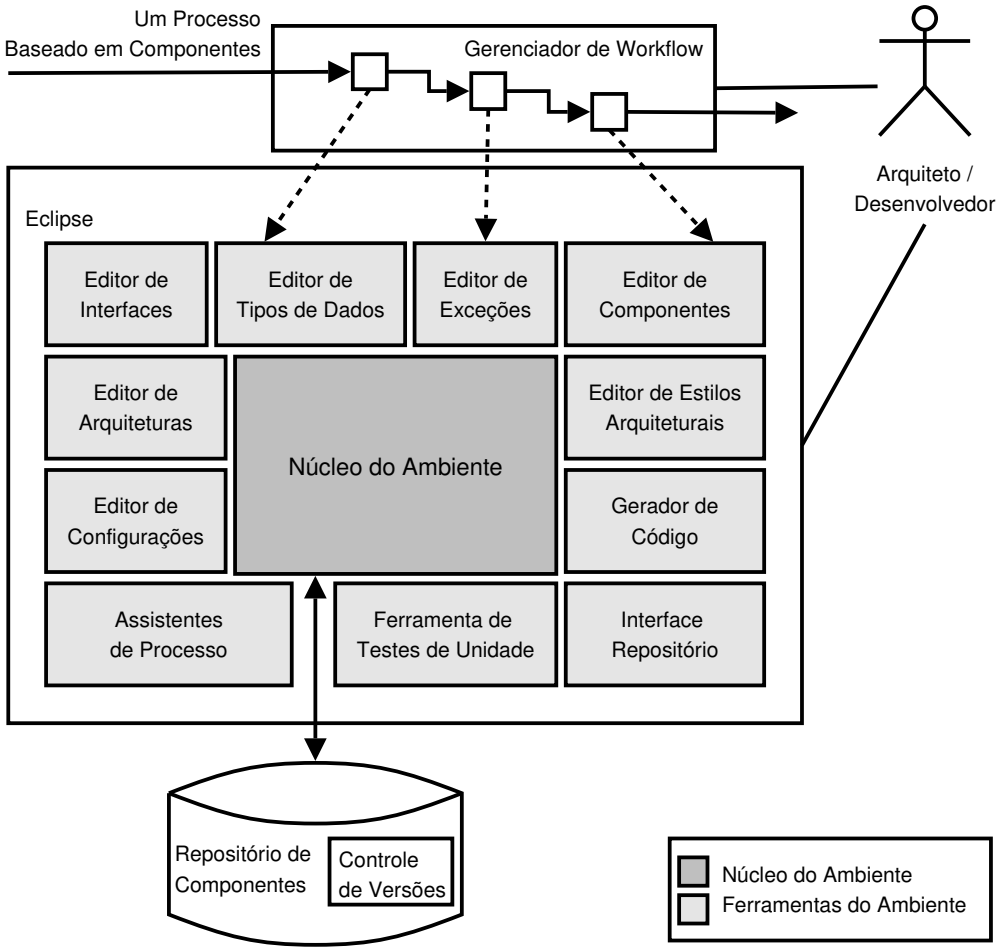

Figura 4.1: Visão geral do ambiente BELLATRIX apresentada na Seção 1.2.

na Seção 5.3.

Dadas as restrições arquiteturais impostas pelo UML Components e pelos *frameworks* do Eclipse, a arquitetura do ambiente BELLATRIX conjuga o estilo MVC e o estilo em camadas. Mais especificamente, o componente arquitetural do Modelo de MVC segue, internamente, o estilo em camadas. Uma visão lógica (estrutural) dessa arquitetura é apresentada na Figura 4.2. Cada Ferramenta do ambiente, por exemplo editores e gerador de código, possui sua própria apresentação para o usuário, contendo os elementos de Visão e Controle. O Núcleo do Ambiente é único para todas as ferramentas e representa o Modelo do estilo MVC. Internamente, ele foi dividido nas camadas recomendadas pelo UML Components. Ainda, uma camada de Persistência foi adicionada a esse componente arquitetural. Tanto as ferramentas quanto o núcleo do ambiente utilizam recursos do Eclipse.

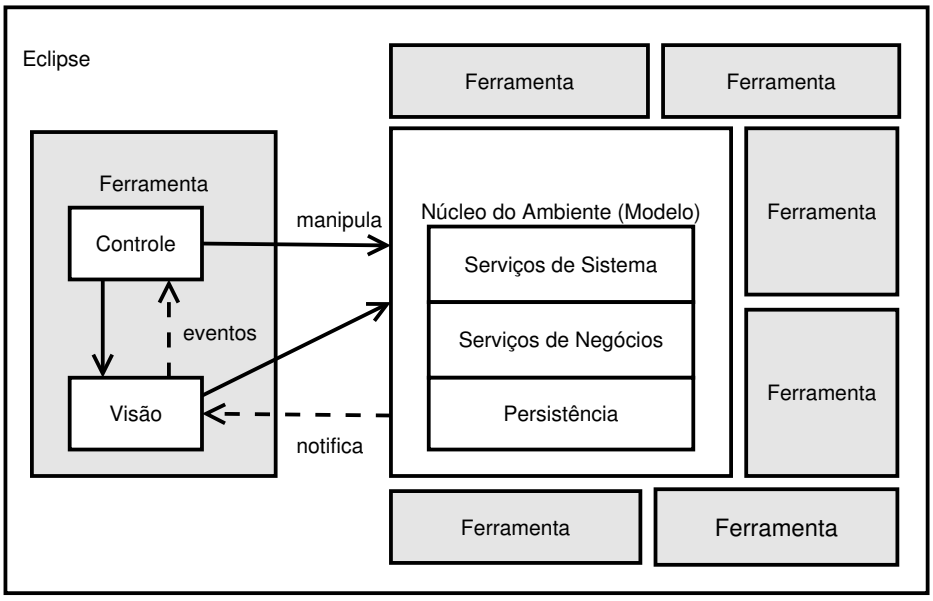

Figura 4.2: Visão lógica da arquitetura do ambiente Bellatrix

4.2 Modelo Conceitual do Negócio

O **modelo conceitual do negócio** é um modelo das informações que existem em um domínio de negócios [15]. Sendo conceitual, ele não é necessariamente um modelo computacional. Esse modelo pode ser representado num diagrama de classes UML que captura classes conceituais e suas relações. Um **modelo de tipos do negócio** pode ser derivado do modelo conceitual. O modelo de tipos é um modelo de especificação, cujo foco é em modelar precisamente as informações do domínio que estão dentro do escopo do sistema computacional.

O modelo conceitual do ambiente Bellatrix deve ser um modelo representando os principais conceitos de DBC e arquiteturas de software, entre eles interfaces, componentes e conectores. Num trabalho anterior [33] foi proposto um modelo conceitual que integra os principais conceitos de arquitetura de software e DBC. Partes desse modelo são apresentados na Figura 4.3 e na Figura 4.4, respectivamente. Uma Arquitetura de Software possui uma ou mais Visões Arquiteturais consistentes entre si. Uma visão possui Elementos Arquiteturais. Um elemento arquitetural possui um Tipo Arquitetural, que pode ser um Conector ou um Componente Arquitetural. Esses elementos ainda possuem Pontos de Conexão e podem estar conectados entre si em uma dada visão. Ainda, um elemento arquitetural pode possuir arquiteturas internas. Um Componente de Software possui uma Especificação e uma Implementação. A especificação do componente é um elemento arquitetural, no contexto de uma arquitetura de componentes. Uma implementação de componente pode

ser um Componente Elementar ou um Componente Composto. O componente composto é formado por outros componentes internos.

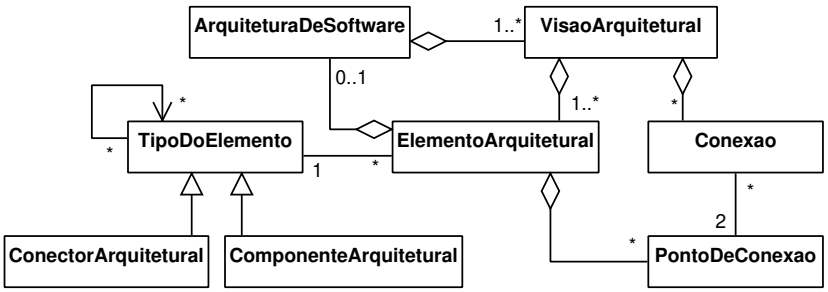

Figura 4.3: Modelo conceitual de arquiteturas de software.

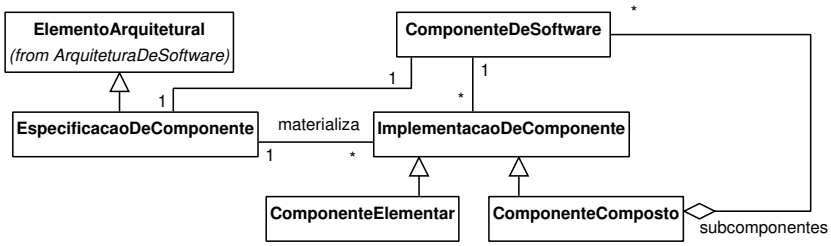

Figura 4.4: Modelo conceitual de DBC.

Entretanto, o modelo conceitual de arquiteturas e DBC não cobre a especificação de interfaces e suas operações, devido à existência prévia de trabalhos detalhados sobre o assunto. Um dos trabalhos mais desenvolvidos nessa área é o modelo CORBA IDL (do inglês *Interface Definition Language*) [52], atualmente um padrão ISO sob número 14750. Para o modelo conceitual do ambiente essas informações são necessárias, de modo que um modelo conceitual derivado da especificação do padrão IDL foi criado e incluído no modelo de componentes e arquiteturas. Esse modelo pode ser visto na Figura 4.5. Uma interface pode conter definições de tipo, constantes, atributos, exceções e operações. Operações podem conter parâmetros — que podem ser de entrada, saída ou entrada e saída — e estar associadas às exceções que elas lançam. As definições de tipo podem ser uniões, enumerações e tipos estruturados. Exceções e tipos de dados estruturados possuem atributos membros em que armazenam informação.

Ainda, é importante incluir os elementos de testes de componentes e sistemas no modelo conceitual, tais como casos de testes e resultados esperados. Rocha [57] e Guerra *et al.* [32] definem como parte de seus trabalhos um modelo para manipulação e persistência de casos de testes. Esse modelo também foi incluído no modelo conceitual e é apresentado na Figura 4.6. Um conjunto de testes possui um ou mais casos de teste. Cada caso de teste faz chamadas de método nas interfaces a serem testadas. Essas chamadas conhecem as

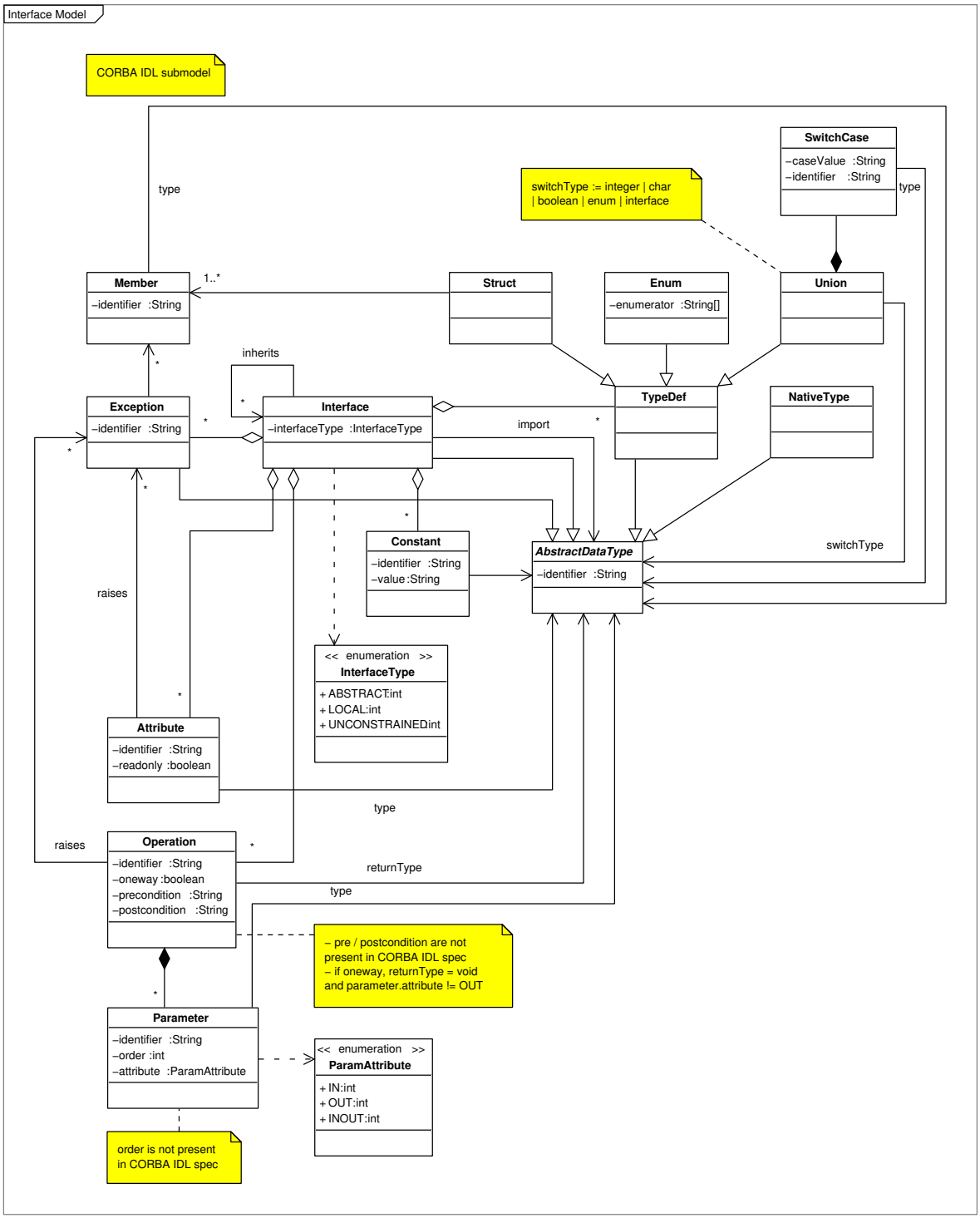

Figura 4.5: Modelo conceitual de interfaces baseado no padrão CORBA IDL.

operações das respectivas interfaces providas. Elas armazenam os argumentos a serem passados e seus valores. Ainda, a chamada conhece os resultados esperados de cada chamada e pode armazenar os resultados de execução do teste obtidos. A chamada pode utilizar stubs de testes que também possuem seus resultados. Esses resultados podem ser normais ou excepcionais.

O modelo conceitual resultante desses trabalhos foi refinado e deu origem ao modelo de tipos do ambiente BELLATRIX, conforme descrito no processo UML Components. Assim, o modelo de tipos possui um escopo bem definido, com relacionamentos e atributos revisados e modelados com foco na implementação do sistema.

Para a construção dos editores gráficos, seria necessário adicionar ao modelo de tipos do ambiente classes e relacionamentos que representassem os elementos gráficos, tais como linhas e conexões, posição na tela, cores, entre outros. Assim, para cada editor gráfico foi criado um sub-modelo independente contendo todos os elementos necessários para seu funcionamento. Esse sub-modelo possui referências uni-direcionais para os elementos conceituais, utilizando o padrão *bridge* [27]. Com isso, pôde-se manter a separação entre os elementos conceituais do modelo e os elementos que representam aspectos gráficos, e conseqüentemente os editores, promovendo organização e o desacoplamento.

4.3 Especificação dos Componentes

O primeiro passo para a especificação dos componentes que formam o componente arquitetural Modelo apresentado na arquitetura inicial do ambiente da Figura 4.2 é a identificação das interfaces que estarão presentes nos componentes das camadas de serviços de sistema e de serviços de negócios. As **interfaces de sistema** foram derivadas dos casos de uso identificados, um para cada caso de uso. Para cada passo de responsabilidade do sistema desses casos de uso, uma operação foi adicionada à interface. Os casos de uso foram identificados de acordo com a Seção 3.3. As **interfaces de negócios** foram criadas a partir da identificação dos tipos principais (*core types*) do modelo de tipos do negócio. Foi criada uma interface de negócio para cada tipo principal. O modelo de tipos do ambiente foi obtido conforme descrito na Seção 4.2.

A partir dessas interfaces, componentes foram identificados e criados. As interfaces de sistema foram agrupadas em componentes seguindo a divisão em ferramentas da proposta inicial do ambiente, apresentada na Seção 1.2. Esses componentes são **componentes de sistema**, localizados na camada de serviços de sistema da arquitetura do ambiente. A Figura 4.7 mostra um exemplo de um componente de sistema, o componente `ArchitectureSystem`, responsável pelos serviços de especificação de arquiteturas e configurações. Para cada interface de negócios foi criado um componente, conforme recomendado pelo processo. Esses componentes são os **componentes de negócios**, lo-

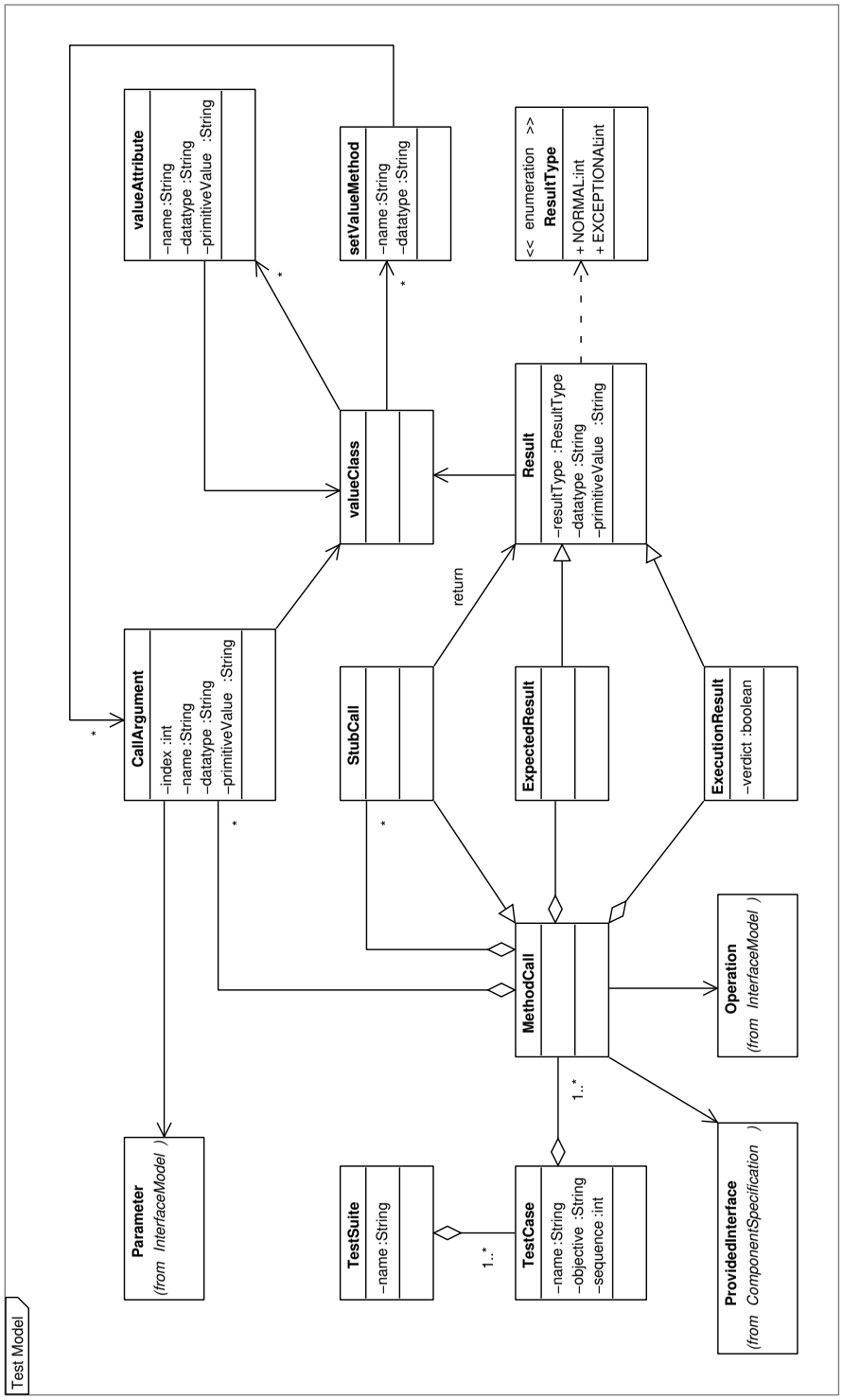

Figura 4.6: Modelo conceitual de testes.

calizados na camada de serviços de negócios da arquitetura mencionada. A Figura 4.8 apresenta um componente de negócios, o componente `ComponentSpecificationMgr`, responsável pela especificação de componentes.

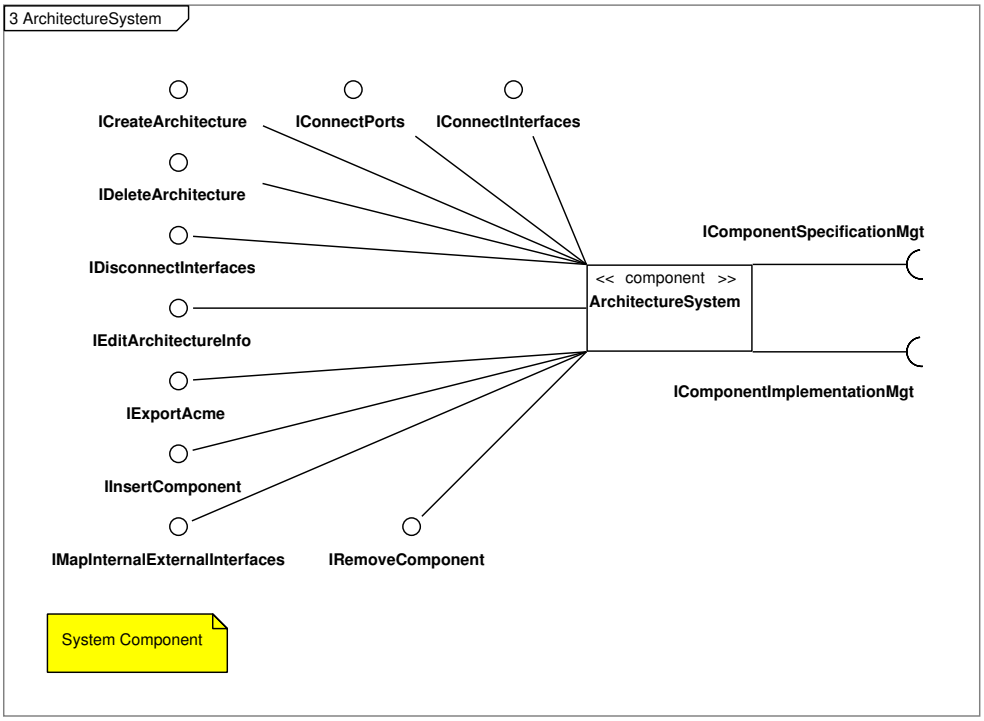

Figura 4.7: Exemplo de componente de sistema.

A partir desses componentes de sistemas e negócios, uma especificação da arquitetura de componentes preliminar foi criada, conforme descrito na Seção 4.4. Foram feitos os diagramas de colaboração para descrever a interação entre as interfaces e detalhar as operações das mesmas. Ainda, a arquitetura de componentes foi refinada.

4.4 Arquitetura de Componentes do Ambiente BELLATRIX

A especificação da **arquitetura de componentes** define um contexto no qual dependências de componentes isolados (interfaces requeridas) são vinculadas a serviços (interfaces providas) de outros componentes [15]. Partindo dos componentes de sistema e negócio identificados, a arquitetura inicial foi montada considerando-se os serviços oferecidos pelos componentes de sistema e quais os tipos de dados dos componentes de negócios eles precisariam manipular. A arquitetura ainda foi refinada durante a etapa de interação entre

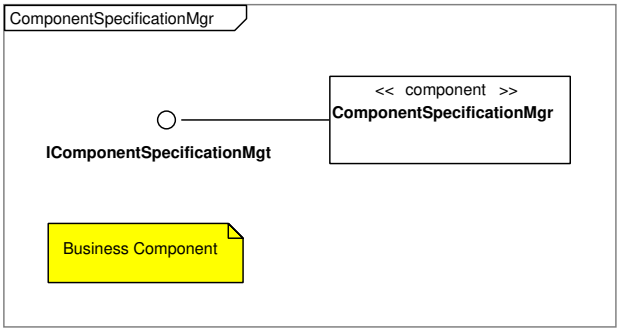

Figura 4.8: Exemplo de componente de negócios.

as interfaces, quando as interfaces foram detalhadas com suas operações e parâmetros. A arquitetura de componentes obtida é apresentada na Figura 4.9. Essa figura mostra o Núcleo do Ambiente, mais especificamente as camadas de serviços de sistema e serviços de negócios. Para fins de clareza, a arquitetura mostrada abstrai as interfaces de sistema (providas pelos componentes de sistema), agrupando-as numa única interface. As interfaces de negócios estão todas representadas. Apesar da figura não explicitar a existência de conectores, eles foram considerados na arquitetura do ambiente, para a conexão entre os componentes.

4.5 Resumo

O projeto do ambiente Bellatrix foi concebido baseado em componentes e seguindo o processo UML Components, visando aumentar sua manutenibilidade e extensibilidade. Esse processo parte de uma arquitetura seguindo o estilo em camadas. Um dos requisitos do ambiente é integração com o IDE Eclipse, cujas tecnologias impõem uma arquitetura seguindo o estilo MVC. Assim, a arquitetura resultante combina ambos os estilos, criando um MVC cujo modelo é organizado em camadas.

O modelo conceitual do negócio é um modelo conceitual das informações que existem em um domínio de negócios. A partir desse modelo, é possível derivar o modelo de tipos do sistema, que é um modelo de especificação, cujo foco é em modelar precisamente as informações do domínio que estão dentro do escopo do sistema computacional. O modelo conceitual do ambiente Bellatrix baseou-se em trabalhos existentes e anteriores, incluindo um modelo que representa conceitos de arquiteturas de software e DBC, um modelo baseado na IDL para representação de interfaces e um modelo para representar casos de testes.

O primeiro passo para a especificação dos componentes do sistema é a identificação das interfaces a serem providas por cada camada. As interfaces de sistema são criadas a

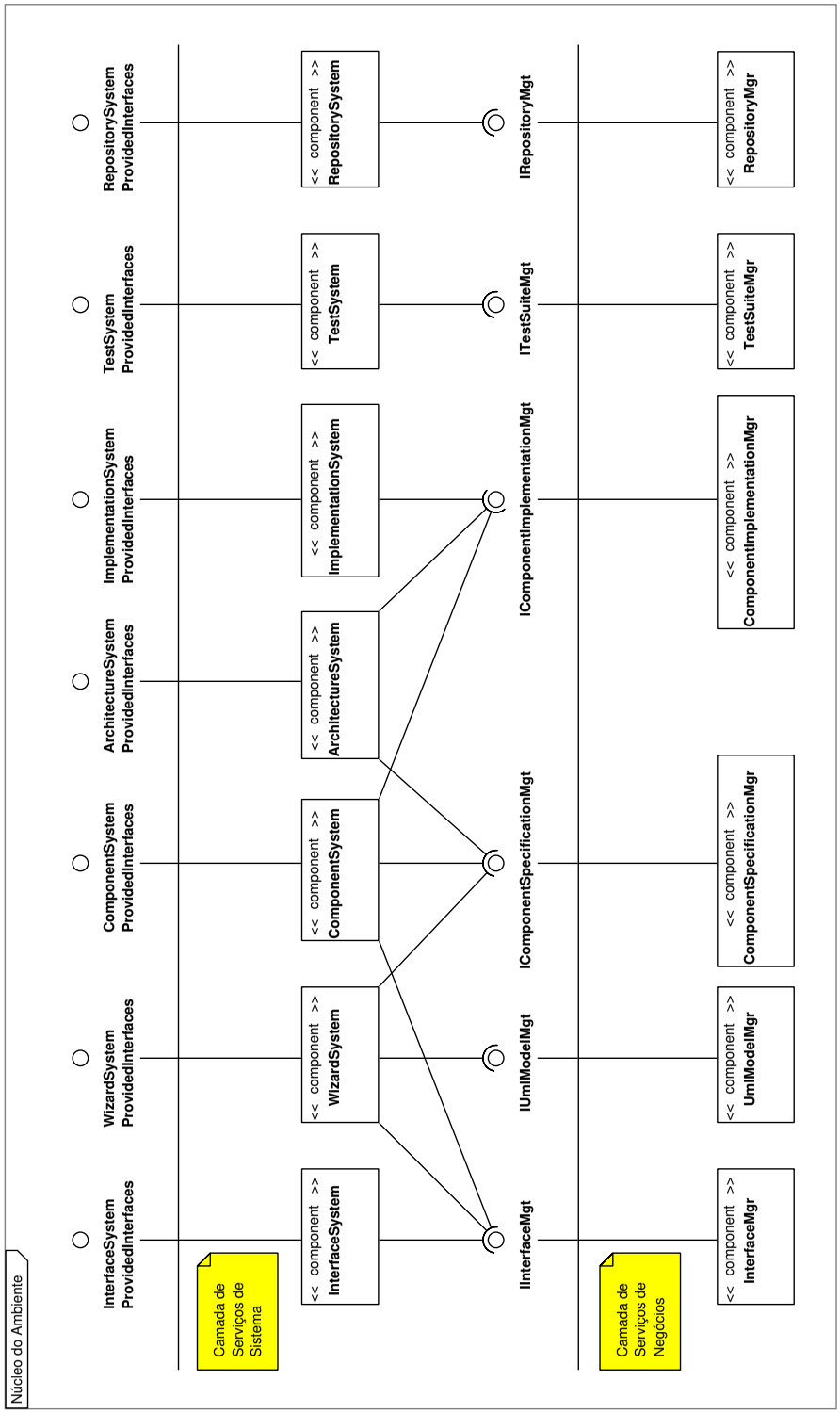

Figura 4.9: Arquitetura de componentes das camadas de sistema e negócios do núcleo do ambiente BELLATRIX.

partir dos casos de uso. As interfaces de negócios são criados a partir dos tipos principais (*core types*) do modelo conceitual. Foram criados componentes a partir dessas interfaces. Os componentes de sistema agruparam as interfaces de acordo com as ferramentas do ambiente descritas na Seção 1.2. Os componentes de negócios foram criados um para cada interface de negócio. Esses componentes foram conectados numa arquitetura de componentes, tomando por base os serviços dos componentes de sistema e quais os tipos do modelo eles manipulariam. As interfaces foram detalhadas com a criação de diagramas de colaboração mostrando a interação entre seus serviços. A arquitetura de componentes também foi refinada a partir dessas informações.

Assim, ao final do projeto do ambiente, possuimos as especificações dos componentes que o compõem e suas interfaces, bem como a arquitetura de componentes e o modelo de tipos do ambiente. O Capítulo 5 descreve como esse projeto foi implementado no ambiente Eclipse.

Capítulo 5

Implementação do Ambiente Bellatrix

Neste capítulo é descrita a estratégia e tecnologias utilizadas para a implementação do ambiente Bellatrix. Foram implementados, seguindo a Figura 1.1 os editores de (1) interfaces, (2) tipos de dados, (3) exceções e (4) componentes, bem como partes de seu Núcleo, conforme descrito nas próximas seções. No contexto de outro projeto de mestrado em andamento [48], foram implementados os editores de (5) arquiteturas, (6) configurações, (7) estilos arquiteturais e (8) gerador de código seguindo a mesma estratégia definida neste capítulo.

A Seção 5.1 descreve a materialização de componentes das camadas de sistema e negócios no ambiente Eclipse, propondo um modelo de componentes para o mesmo. Na Seção 5.2 é mostrada a implementação do modelo de tipos do negócio na camada de persistência do ambiente. A Seção 5.3 apresenta as tecnologias envolvidas e o desenvolvimento da interface gráfica com o usuário do ambiente, utilizando o estilo arquitetural MVC. Na Seção 5.4 é descrita a implementação do gerenciador do processo do ambiente. A Seção 5.5 mostra a estratégia de testes do ambiente e na Seção 5.6 apresentamos um sumário dos assuntos tratados.

5.1 Implementação do Núcleo do Ambiente

Um dos requisitos iniciais do ambiente Bellatrix era a sua integração com a plataforma Eclipse, conforme discutido na Seção 3.2.4. Nas próximas seções são apresentados o modelo de *plug-ins* utilizado pelo Eclipse e a estratégia de materialização dos componentes do núcleo do ambiente Bellatrix no Eclipse, utilizada nas camadas de sistema e negócios do ambiente. Seguindo a Figura 4.9, que mostra a arquitetura de componentes do núcleo do ambiente, foram implementados os seguintes componentes: `InterfaceSystem`,

`ComponentSystem`, `ArchitectureSystem`, `ImplementationSystem`, `InterfaceMgr`, `ComponentSpecificationMgr`, `ComponentImplementationMgr` e seus respectivos conectores.

5.1.1 Modelo de *Plug-ins* do Eclipse

O ambiente Eclipse é formado por uma plataforma e um conjunto de módulos anexáveis chamados *plug-ins*, de modo que a maior parte de suas funções são implementadas por meio de *plug-ins*. Ele pode ser considerado um *framework* de componentes, onde cada componente é um *plug-in*. Esses *plug-ins* podem ser instalados de forma independente, simplesmente copiando seu código binário num diretório especial chamado `plugins` do diretório de instalação do Eclipse [21]. Durante sua inicialização, o Eclipse percorre esse diretório, detectando os *plug-ins* instalados. O **código binário** para instalação de um *plug-in* é um arquivo JAR (*Java Archive*, que permite empacotar várias classes em um único arquivo) contendo as classes que implementam as funcionalidades do *plug-in* e arquivos descritores, chamados `plugin.xml` e `manifest.mf` [44]. Para fins de compatibilidade com versões anteriores do Eclipse, também é possível que a instalação de um *plug-in* seja um diretório contendo um arquivo descritor e um arquivo JAR.

Os **arquivos descritores** são arquivos texto onde o *plug-in* declara à plataforma diferentes informações, tais como seu nome, desenvolvedor, versão, dependências e elementos de interface com o usuário. Essas informações presentes nos descritores agilizam a inicialização do Eclipse, possibilitando que os *plug-ins* sejam instanciados sob demanda. Por exemplo, supondo um *plug-in* que cria um novo item de menu no Eclipse, as informações sobre o item de menu — seu local e o texto que aparece ao usuário — são declarados no arquivo descritor, de modo que podem ser exibidos ao usuário antes da instanciação das classes do *plug-in*, que só ocorre quando o usuário seleciona esse item. Os arquivos descritores possuem vários elementos de informação. Os mais importantes, do ponto de vista da composição, são as seguintes declarações:

1. *Export-Package*, que permite definir quais pacotes do *plug-in* são visíveis externamente;

2. *Import-Package*, que permite definir quais os pacotes são necessários para a execução;

3. *Require-Bundle*, que define quais *plug-ins* são necessários para a execução;

4. *Extension-point*, através do qual são definidos pontos onde um *plug-in* pode ser estendido, por exemplo, definindo interfaces que podem ser implementadas para tal;

5. *Extension*, que define pontos de extensão em que um *plug-in* estende outros *plug-ins*, por exemplo, definindo classes de implementação que o Eclipse instancia automaticamente.

5.1.2 Modelo de Implementação de Componentes Utilizado

Para a implementação das camadas de sistema e negócios, é desejável que seus componentes possam ser implementados aproveitando o modelo de *plug-ins* do Eclipse descrito na Seção 5.1.1 e todas as suas vantagens. Entretanto, os seguintes pontos fracos da estrutura do Eclipse foram identificados [41]: (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.

A fim de eliminar ou minimizar essas deficiências, foi aplicado o modelo COSMOS [62] dentro da estrutura de um *plug-in*. Entretanto, esse modelo precisou ser adaptado pela impossibilidade de se criar o código de instanciação do sistema no ambiente Eclipse. Esse código de instanciação do sistema tem por objetivo a instanciação de todos os componentes e conectores do sistema e a troca das instâncias de implementações das interfaces requeridas entre os componentes. Devido a instanciação sob demanda no Eclipse, não há um local apropriado para tal código. Ainda, utilizá-lo violaria uma das diretrizes do ambiente Eclipse. Por fim, utilizamos os mecanismos do próprio Eclipse para apoiar o modelo.

O resultado desse trabalho é um modelo de implementação de componentes que combina o modelo COSMOS e o modelo de *plug-ins* do Eclipse. Assim, cada componente e conector das camadas de sistema e negócios, apresentados na Figura 4.9 é mapeado em um *plug-in* nesse modelo. Ainda, cada um dos editores e ferramentas do ambiente, apresentados na Figura 1.1, são materializados como *plug-ins*. A Figura 5.1 ilustra um componente nesse modelo, que segue as seguintes diretrizes:

1. Cada componente ou conector é materializado como um *plug-in*, possuindo implantação independente, auto-descrição e encapsulamento garantido pelo Eclipse. Na Figura 5.1, o componente A foi mapeado para o *plug-in* apresentado na parte inferior da figura;

2. Internamente, os componentes e conectores seguem a estrutura do modelo COSMOS, incluindo os modelos de especificação, implementação e conectores, bem como os padrões de projeto definidos pelo modelo. Isso confere conectores e interfaces requeridas aos *plug-ins*;

3. No lugar do código de instanciação do sistema, inviável dentro do Eclipse, foi criado um *plug-in* que contém a configuração do sistema (ConfigurationManager). Esse *plug-*

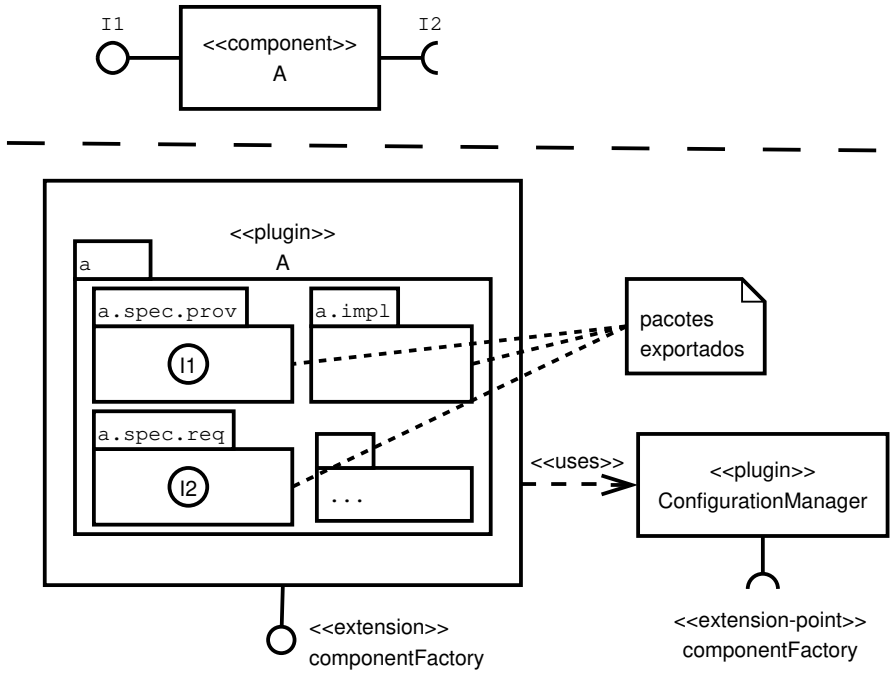

Figura 5.1: Estrutura de um *plug-in* combinado com o modelo COSMOS.

in define uma API à qual todos os componentes acessam de modo a obter instâncias de suas interfaces requeridas. A configuração do sistema pode ser parametrizada através de arquivos de configuração;

4. O *plug-in* de configuração exporta a API utilizada para se obter instâncias das interfaces requeridas e define um *extension-point* que define a interface que os componentes devem implementar para serem instanciados. Esse *extension-point* é identificado como `componentFactory` na Figura 5.1;

5. Um componente declara em seu arquivo descritor que ele exporta sua especificação (interfaces providas e requeridas) e seu código de instanciação (através da classe `ComponentFactory`) e requer o *plug-in* que contém a configuração;

6. Um conector declara em seu arquivo descritor que ele exporta seu código de instanciação (através da classe `ComponentFactory`) e requer os dois componentes que ele conecta e o *plug-in* que contém a configuração;

7. No COSMOS, o gerente (interface `IManager`) de cada componente é responsável por fornecer instâncias das interfaces requeridas. Neste modelo adaptado, se ele não possuir uma dada referência, ele consulta o *plug-in* de configuração, mantendo a instanciação sob demanda.

Uma limitação dessa abordagem é a impossibilidade de se criar arquiteturas de componentes que contenham ciclos de dependências entre componentes. Essa limitação surge de uma limitação do Eclipse, que não permite ciclos na definição do *classpath* de um *plug-in* através dos mecanismos de *Require-Bundle* e *Import-Package*.

5.2 Implementação do Modelo de Tipos

O modelo de tipos de negócio, derivado do modelo conceitual apresentado na Seção 4.2, foi implementado para a construção do ambiente Bellatrix. Essa implementação do modelo contém objetos que são manipulados pelo ambiente durante as atividades de criação e especificação apoiadas, e depois são persistidos. A implementação utilizou a plataforma EMF, descrita na Seção 2.6.

Assim, o modelo de tipos do negócio foi modelado utilizando a ferramenta EclipseUML [55]. Um arquivo XMI do modelo foi exportado e importado no EMF e o código referente ao modelo foi gerado. Esse código foi empacotado como um *plug-in* para o Eclipse. Desse modo, a camada de persistência da arquitetura foi materializada como um *plug-in* e abriga uma implementação do modelo de tipos do negócio, desenvolvido utilizando o EMF.

5.3 Interface Gráfica com Usuário no Eclipse

Para a implementação da interface gráfica com o usuário no Eclipse, em especial os editores gráficos, é necessário a compreensão e o uso das tecnologias de interface gráfica desse ambiente. Buscando maior agilidade, infra-estrutura e integração no uso dessas tecnologias, foi utilizada a plataforma GEF, descrita na Seção 2.6, materializando o mecanismo de MVC (Figura 2.1) da arquitetura do ambiente Bellatrix.

A interface gráfica de cada ferramenta apresentada na Seção 1.2, Figura 1.1, foi materializada como um *plug-in* do Eclipse, utilizando o GEF. A Figura 5.2 mostra a arquitetura de um desses *plug-ins*. Os elementos de visão são construídos em Draw2D, enquanto os elementos de controle utilizam o GEF. Eles se comunicam com o modelo criado utilizando o EMF e as camadas de sistema e negócios. Ainda, eles se comunicam com o editor gráfico, que estende a plataforma Eclipse.

Para a criação dos elementos em Draw2D foi feito um estudo de como cada elemento gráfico, por exemplo representando interfaces e componentes, deveria aparecer ao usuário. Assim, foi criado código utilizando a infra-estrutura do Draw2D que exibisse esses elementos conforme especificado. Esse código em Java define, de maneira geral, a aparência da figura — incluindo retângulos e linhas, com cores e texto — e métodos para alte-

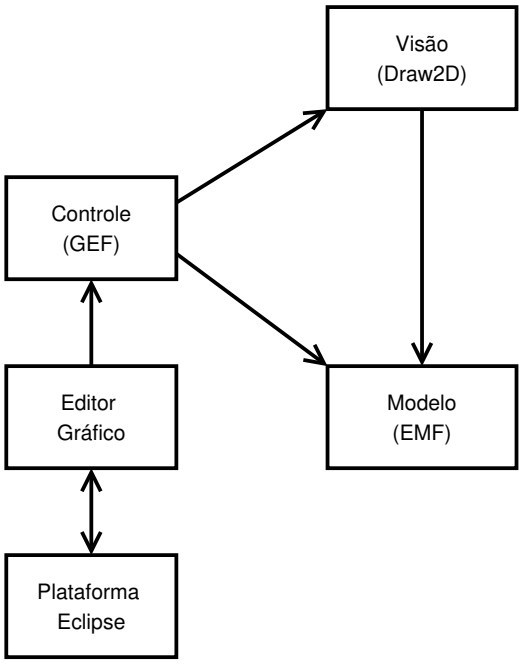

Figura 5.2: Arquitetura de um editor utilizando o GEF.

rar essa aparência, por exemplo, para alterar o nome de uma figura representando um componente.

5.4 Gerenciador do Processo

A arquitetura inicial do ambiente, descrita na Seção 1.2 e mostrada na Figura 4.1, propõe um gerenciador do processo, cuja função é guiar desenvolvedores através do processo de desenvolvimento, indicando tarefas e ferramentas. Para a implementação desse gerenciador de processo, foi utilizada a ferramenta OpenFlow, que utiliza a plataforma Zope para sua execução.

O **Zope** [73] é um servidor de aplicações de código aberto, que permite a construção de sistemas de gerência de conteúdo, *intranets*, portais e aplicações customizadas. Ele é apoiado por várias empresas e desenvolvedores que trabalham no desenvolvimento da plataforma Zope e de aplicações para ela. Esse desenvolvimento é realizado na linguagem de programação Phyton. O Zope contém um banco de dados transacional no qual ele armazena todo o seu conteúdo, incluindo configuração, páginas HTML e *scripts*. Assim, ele permite a edição e manutenção de seu conteúdo através de páginas *web*.

O **OpenFlow** [36] é uma máquina flexível de *workflow*, que possibilita o desenvolvimento de aplicações *web* orientadas a *workflow*. Ele possui suporte a padrões da Internet,

acesso a banco de dados relacionais, é multi-plataforma, baseado na *web* e está disponível para uso como software livre. O OpenFlow permite modelar o processo, definindo tarefas e atividades, que podem ser atribuídas a papéis. Um usuário ou grupo de usuários podem assumir tais papéis e executar as tarefas.

Assim, foi realizada a modelagem do UML Components no OpenFlow, com base nos diagramas de atividades do processo elaborados durante a extração de requisitos do ambiente (Seção 3.3). Assim é possível criar instâncias do processo modelado e executá-los, de acordo com o modelo. Apesar disso, seu uso é limitado. Para explorar melhor a ferramenta, seria necessária a construção de uma aplicação *web* dentro do servidor Zope, interligada com o OpenFlow. Essa implementação não foi realizada devido à necessidade de aprendizado de novas tecnologias e às restrições de tempo.

5.5 Estratégia de Testes para o Desenvolvimento

Nas fases iniciais do desenvolvimento do ambiente Bellatrix foi considerado o uso da ferramenta CBDUnit [32] para os testes de unidade dos componentes que formam as camadas de persistência, sistema e negócios, de modo isolado da camada de interface gráfica. O CDBUnit possui uma interface gráfica própria e pode ser utilizado para testar implementações de componentes que utilizem a linguagem Java e o modelo COSMOS. Entretanto, durante o desenvolvimento do ambiente, essa abordagem se mostrou limitada pois os componentes desenvolvidos possuem dependências com o ambiente Eclipse que deixariam de ser testadas e que dificultam o uso da ferramenta.

Assim, estudou-se a criação de *drivers* e *stubs* de testes com essa finalidade. Esses *drivers* e *stubs* seriam codificados como *plug-ins* para o Eclipse e, assim, poderiam utilizar as dependências do Eclipse. Eles ainda poderiam ser armazenados e mantidos sob controle de versão, de modo a facilitar seu uso em testes de regressão. Entretanto, a criação manual dos *drivers* e *stubs* e a implementação de uma infra-estrutura de testes para a execução automatizada dos casos de testes mostrou-se custosa em termos de tempo e não pôde ser realizada. Os testes da interface gráfica com o usuário também se mostraram complexos sem ferramentas adequadas para tal.

Desse modo, não foi possível a adoção de uma estratégia para testes automatizados dos componentes e da interface gráfica com o usuário. Os testes foram realizados manualmente tendo como base os casos de uso que estavam sendo implementados no momento e o conteúdo dos arquivos gerados pelo ambiente, formato em XMI.

5.6 Resumo

Este capítulo apresentou as estratégias utilizadas na implementação dos diferentes elementos da arquitetura do ambiente BELLATRIX. A materialização dos **componentes de sistema e negócios** utilizou o modelo de *plug-ins* do Eclipse em conjunto com o modelo COSMOS. O Eclipse pode ser considerado uma plataforma para composição, sendo que suas características como *plug-ins* extensíveis (arquiteturas multi-nível), portas explícitas, auto-descrição e encapsulamento são considerados pontos fortes para tal. Entretanto, o modelo de *plug-ins* para o Eclipse possui como pontos fracos a ausência de componentes requeridos obrigatórios e de conectores explícitos. A fim de minimizar esses problemas, ainda aproveitando as características do Eclipse, o modelo COSMOS foi aplicado no interior dos *plug-ins*. A materialização da **camada de persistência**, contendo o modelo de tipos de negócio do ambiente, utilizou o EMF. O EMF, tecnologia do Eclipse, que provê infra-estrutura para a criação de modelos em código Java a partir de uma especificação do mesmo em XMI. A camada de **interface gráfica com o usuário** foi implementada utilizando GEF, concretizando o mecanismo de MVC da arquitetura. O GEF provê uma plataforma para a criação de editores gráficos no Eclipse. Para a concretização do **gerenciador do processo**, utilizou-se a ferramenta OpenFlow. O OpenFlow é uma ferramenta que possibilita a criação de aplicações *web* baseadas em *workflow*, utilizando o servidor de aplicações Zope. O processo UML Components foi modelado e inserido nessa ferramenta. Para os **testes** do ambiente, não foi possível estabelecer uma rotina de testes automatizados para o ambiente, de modo que os testes foram feitos manualmente com base na descrição dos casos de uso e nos arquivos XMI produzidos pelo ambiente.

No Capítulo 6 apresentamos um estudo de caso da implementação do ambiente mostrada nesse capítulo. Esse estudo de caso modela um exemplo de sistema de gerenciamento de reservas de hotel seguindo o processo UML Components.

Capítulo 6

Estudo de Caso: Sistema de Reservas de Hotel

Este capítulo apresenta um estudo de caso que foi realizado para avaliar a especificação e a implementação atual do ambiente Bellatrix, com foco num processo de DBC. A Seção 6.1 apresenta o objetivo do estudo de caso. A Seção 6.2 mostra os materiais e métodos com os quais o estudo de caso foi conduzido. Na Seção 6.3, o sistema de reservas de hotel utilizado é descrito. A Seção 6.4 apresenta sua execução e a Seção 6.5 discute resultados. Por fim, a Seção 6.6 sumariza este capítulo.

6.1 Objetivo

Um estudo de caso foi conduzido com o objetivo de avaliar o ambiente Bellatrix com foco num processo DBC, verificando se todas as funcionalidades desejáveis foram especificadas e se as funcionalidades mínimas para o seu uso seguindo um processo de desenvolvimento foram implementadas, buscando eventualmente identificar novas funcionalidades e deficiências na especificação do ambiente.

6.2 Material e Método

Material:

- O estudo foi realizado pelo autor desse trabalho.
- A especificação de um sistema fictício de reservas de quartos em hotel, apresentada na Seção 6.3, foi utilizada. Essa especificação foi baseada no exemplo presente

no livro *UML Components: a simple process for specifying component-based software* [15].

- O estudo foi conduzido nas dependências do Instituto de Computação da UNICAMP. O ambiente computacional utilizado inclui o sistema operacional Linux Fedora Core 4 e a plataforma Eclipse, com os *plug-ins* do Bellatrix e suas dependências instaladas. Ainda, para a modelagem dos diagramas UML utilizou-se as ferramentas Omondo EclipseUML [55], um *plug-in* para o Eclipse, e Gentleware Poseidon [29], uma aplicação em Java Swing. Foram utilizadas duas ferramentas UML com o objetivo de comparar o uso de tais ferramentas integradas ou não à plataforma Eclipse e avaliar a transferência de seus artefatos ao Bellatrix.

Método:

Para a realização desse estudo, seguiu-se as fases do processo UML Components, gerando os artefatos indicados pelo processo utilizando as ferramentas UML e o ambiente Bellatrix. Foi especificado o sistema de reservas de quartos em hotel. As fases do processo, descritas em maiores detalhes na Seção 2.3, são:

- Fase 1: Levantamento de requisitos, quando os requisitos do sistema são identificados, bem como seus casos de uso e o processo de negócios utilizado.
- Fase 2: Especificação de componentes. Essa fase se subdivide em três subfases:
  - Fase 2.1: Identificação de componentes, quando as interfaces são identificadas. A partir das interfaces, os componentes são modelados e se monta uma primeira arquitetura do sistema.
  - Fase 2.2: Interação de componentes, quando as assinaturas das operações das interfaces são identificadas. Novas operações também são identificadas nesta fase.
  - Fase 2.3: Especificação de componentes, quando se consolida a arquitetura de componentes do sistema, produzindo documentação para cada componente.
- Fase 3: Provisionamento, fase em que os componentes especificados são disponibilizados, seja através de aquisição, reuso ou desenvolvimento.
- Fase 4: Montagem, fase em que a implementação dos componentes são ligadas entre si e com a interface de usuário, formando o sistema final.

6.3 Descrição do Sistema de Reservas de Quartos

Para a realização desse estudo, modelou-se o exemplo presente no livro *UML Components: a simple process for specifying component-based software* [15], um sistema de gerenciamento de reservas de hotel. Esse sistema 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, tipo de quarto e período da estadia para a reserva, bem como registrar a entrada efetiva do hóspede no hotel. Ele ainda está interligado ao sistema de cobranças da cadeia, emitindo pedidos de cobrança das estadias e processando reservas não concretizadas — usualmente chamadas de *no-show*. O diagrama de casos de uso do sistema é apresentado na Figura 6.1.

A seguir, descrevemos brevemente os casos de uso do sistema:

Reservar quarto. O reservante reserva um quarto em um hotel. Para isso, ele precisa informar o hotel, as datas de entrada e saída previstas e o tipo de quarto. O reservante informa seus dados pessoais, caso não seja um cliente cadastrado. Após a realização da reserva um código de reserva é entregue ao reservante.

Registrar entrada no hotel. O hóspede chega ao hotel e pede o quarto que lhe está reservado. Para isso, o hóspede fornece o código de reserva ao atendente. Ele confirma as datas de entrada e saída e o tipo de quarto, registrando a entrada do hóspede no sistema de cobranças.

Alterar reserva. O reservante deseja alterar as informações de uma reserva previamente registrada. Assim, o reservante fornece o código de reserva e informa os dados a serem alterados, entre o hotel, datas de entrada e saída previstas e tipo de quarto. A reserva é alterada e o código de reserva permanece o mesmo.

Cancelar reserva. O reservante deseja cancelar uma reserva feita previamente. Assim, ele fornece o código de reserva e confirma o cancelamento de sua reserva. O cancelamento deve ser feito antes da data de entrada prevista na reserva.

Processar *no-show*. Caso o hóspede não compareça ao hotel na data de sua reserva, ou seja, não registre sua entrada no hotel, uma taxa de *no-show* é cobrada. O administrador de reservas emite os pedidos de cobrança de reservas nessa situação.

Remover clientes antigos. O administrador de reservas pode remover o cadastro de clientes antigos. O sistema deve permitir listar clientes que não utilizaram o hotel num período de tempo a ser determinado pelo administrador.

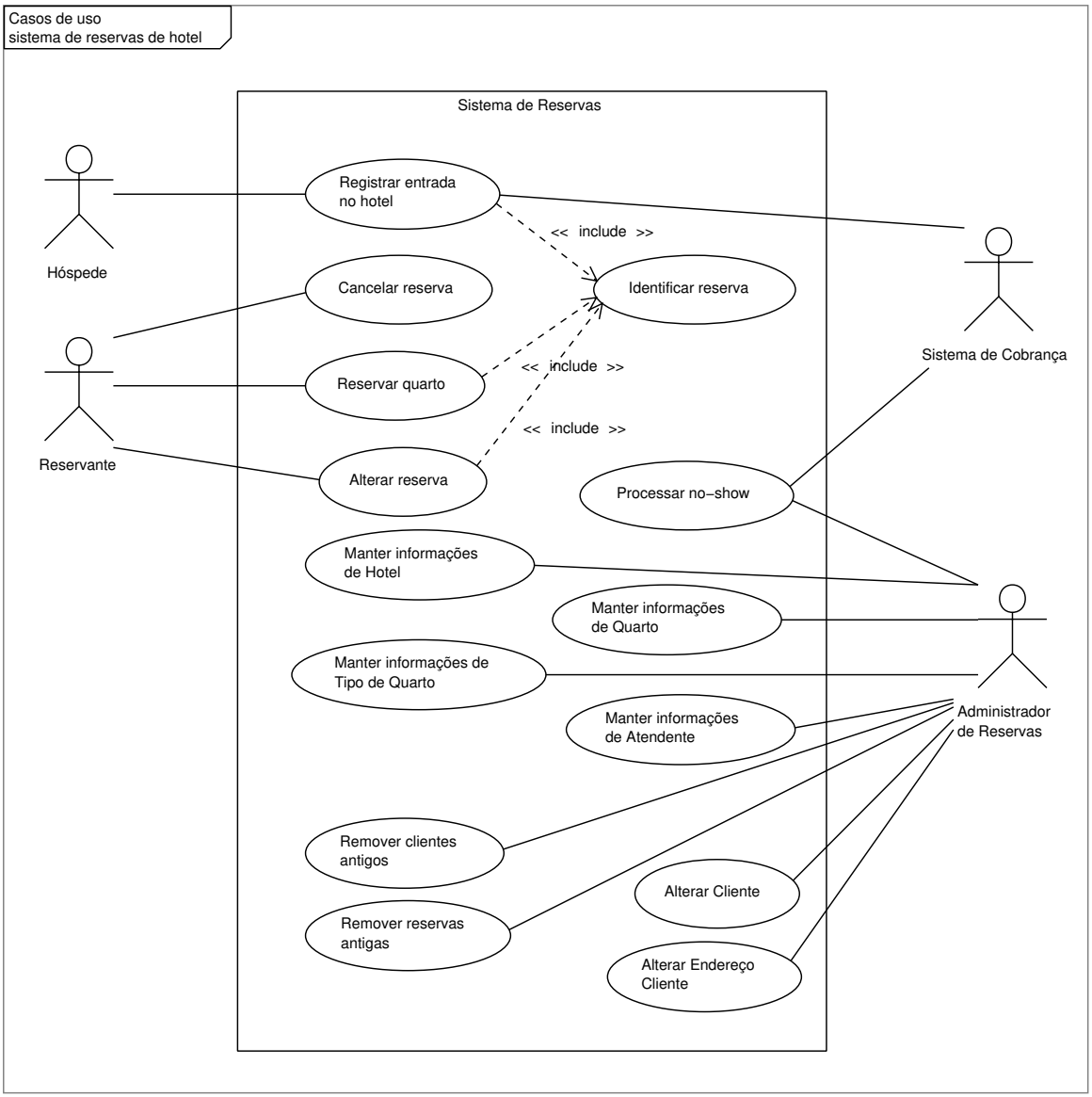

Figura 6.1: Casos de uso do sistema de reservas.

Remover reservas antigas. O administrador de reservas pode remover as informações referentes a reservas antigas já cobradas. O sistema deve permitir listar reservas nessa situação num período de tempo a ser determinado pelo administrador.

Alterar cliente. O administrador de reservas pode alterar dados pessoais dos clientes cadastrados no hotel.

Alterar endereço cliente. O administrador de reservas pode alterar o endereço dos clientes cadastrados no hotel.

Manter informações de hotel. O administrador de reservas pode incluir, alterar e excluir informações registradas sobre os hotéis.

Manter informações de quarto. O administrador de reservas pode incluir, alterar e excluir informações registradas sobre os quartos existentes nos hotéis.

Manter informações de tipo de quarto. O administrador de reservas pode incluir, alterar e excluir informações registradas sobre os tipos de quartos existentes nos hotéis. Tipos de quartos podem incluir tamanho, número de camas, nível de conforto, entre outros.

Manter informações de atendente. O administrador de reservas pode incluir, alterar e excluir informações registradas sobre os atendentes dos hotéis.

Durante o estudo de caso, a modelagem da especificação do sistema, apresentada de forma resumida no livro, foi ampliada de forma a deixá-la mais completa.

6.4 Execução

Conforme mencionado, um sistema de reservas de hotel foi modelado no ambiente Bellatrix e nas ferramentas Eclipse UML e Poseidon, seguindo os passos do processo UML Components. As próximas seções apresentam os principais artefatos por fase modelados no ambiente.

6.4.1 Levantamento de Requisitos (Fase 1)

A fase de levantamento de requisitos inclui atividades de modelagem do processo e dos conceitos do negócio e a modelagem de casos de uso. A modelagem do processo do negócio é feita utilizando-se de diagramas de atividades UML, onde as atividades para a realização das tarefas do negócio são modeladas. O modelo do processo de reservas de hotel foi criado utilizando as ferramentas UML e pode ser visto na Figura 6.2.

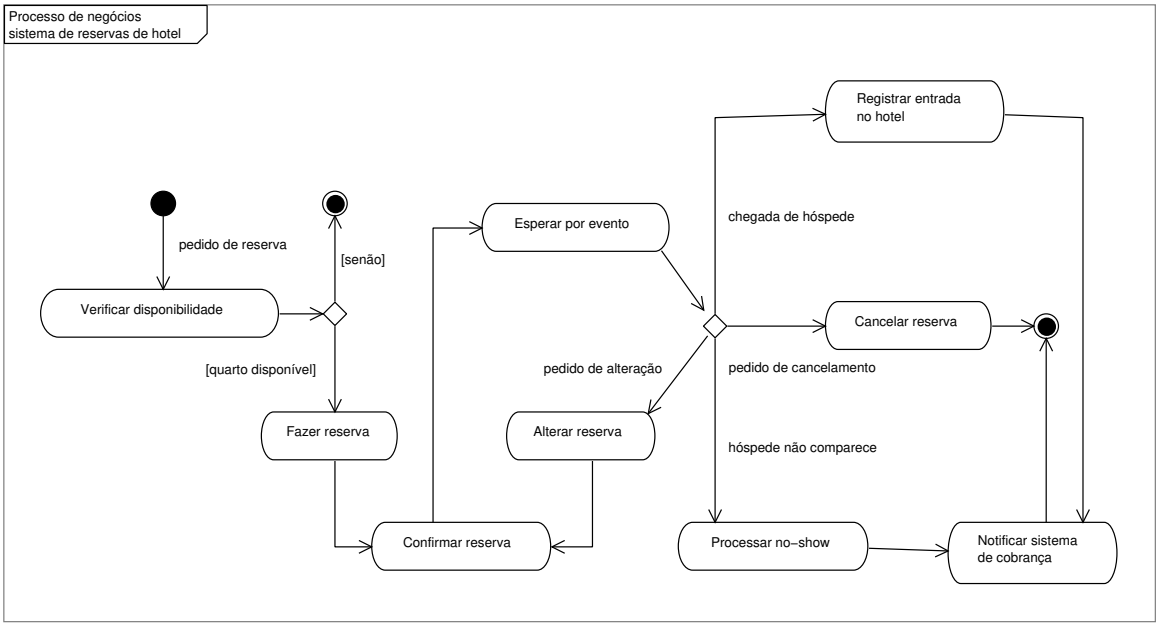

Figura 6.2: Modelo do processo do negócio.

O **modelo conceitual do negócio**, expresso através de um diagrama de classes, representa os principais conceitos e termos do domínio do negócio e o relacionamento entre eles. Seu objetivo é modelar o negócio, sem a preocupação com o escopo do sistema a ser desenvolvido. O modelo conceitual do sistema de reservas de hotel foi criado utilizando as ferramentas UML e pode ser visto na Figura 6.3.

Seguindo o processo, os casos de uso foram descobertos a partir do modelo do processo do negócio. Cada atividade do diagrama de atividades UML (Figura 6.2) é um potencial caso de uso do sistema. Ainda, a partir do modelo conceitual do negócio, levantou-se os atores e principais entidades do sistema. As entidades que precisam ser mantidas no escopo do sistema, por exemplo incluídas ou removidas, precisam de casos de uso para tal. O diagrama de casos de uso do sistema de reservas foi modelado utilizando as ferramentas UML e pode ser visto na Figura 6.1.

Para fins de brevidade, somente dois casos de uso do sistema de reservas são descritos no exemplo do livro *UML Components* utilizado. Assim, os passos dos demais casos de uso foram descritos utilizando como base o formato utilizado no exemplo. As descrições foram criadas em um editor de textos e uma dessas descrições pode ser vista na Tabela 6.1.

6.4.2 Identificação de componentes (Fase 2.1)

Na fase de identificação de componentes, são identificadas as interfaces e os componentes do sistema, bem como uma arquitetura inicial de componentes é definida. O processo

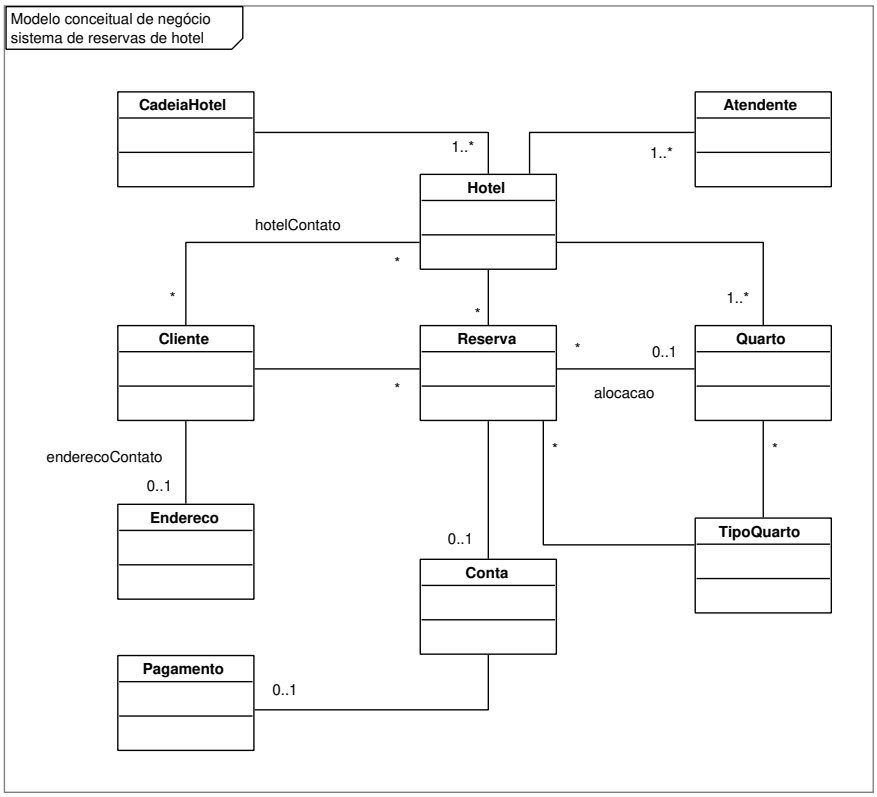

Figura 6.3: Modelo conceitual do negócio.

Tabela 6.1: Descrição de caso de uso Reservar Quarto.

Nome: Reservar Quarto (Make a reservation)

Iniciador: Reservante (ReservationMaker)
Objetivo: Reservar um quarto no hotel
Cenário principal de sucesso:

1. O reservante pede para fazer uma reserva no hotel;
2. O reservante escolhe, em qualquer ordem, o hotel, datas e o tipo do quarto;
3. O sistema informa o preço ao reservante;
4. O reservante confirma a reserva;
5. O reservante informa seu nome e CEP;
6. O reservante informa seu email de contato;
7. O sistema faz a reserva e gera o código da reserva;
8. O sistema informa o código da reserva ao reservante;
9. O sistema cria e envia a confirmação da reserva via email.

Extensões:

3. O tipo de quarto não está disponível
   a. O sistema oferece tipos de quartos alternativos
   b. O reservante escolhe uma das alternativas disponíveis

3b. O reservante rejeita as alternativas
   a. Fim

4. O reservante não confirma a reserva
   a. Fim

6. O cliente já está cadastrado (com base no nome e no CEP)
   a. Fim

UML Components segue uma arquitetura em camadas pré-definida (Figura 2.3) onde as principais camadas são a **camada de sistema**, que provê acesso aos serviços do sistema, e a **camada de negócios**, que contém as regras de negócios.

As interfaces da camada de sistemas têm origem nos casos de uso. Para cada caso de uso é criada uma interface de sistema. Nessas interfaces, para cada passo de responsabilidade do sistema do caso de uso, é criada uma operação. As operações são deixadas sem parâmetros e sem tipo de retorno, que serão descobertos em fases futuras. A modelagem dessas interfaces foi feita utilizando o Editor de Interfaces do Bellatrix. A Figura 6.4 mostra a interface de sistema `IMakeReservation`, referente ao caso de uso `Reservar quarto` (*Make a reservation*).

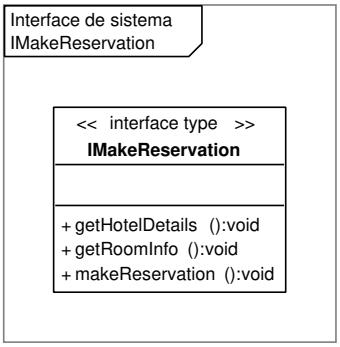

Figura 6.4: Interface de sistema `IMakeReservation`.

No ambiente Bellatrix estão especificados assistentes para importar diagramas de casos de uso em formato XMI e, a partir deles, criar automaticamente as interfaces de sistema. Entretanto, esses assistentes não foram implementados, de modo que as interfaces e operações foram criados manualmente, observando-se o diagrama e a descrição dos casos de uso.

As interfaces da camada de negócios têm origem no modelo conceitual do negócio. Esse modelo é refinado, incluindo atributos, cardinalidades e excluindo elementos e relacionamentos com base no escopo do sistema, criando o **modelo de tipos do negócio**. No modelo de tipos, são identificados os **tipos principais** (do inglês *core types*) do negócio, através de uma extensão do mecanismo de estereótipos de UML. O modelo de tipos do sistema de reservas de hotel foi criado nas ferramentas UML e pode ser visto na Figura 6.5.

Para cada um dos tipos principais do negócio, uma interface de negócios é criada. Essas interfaces são deixadas sem operações, que serão descobertas em fases posteriores.De modo similar às interfaces de sistema, as interfaces de negócios foram criadas utilizando o editor de interfaces do Bellatrix. Ainda, também estão especificados para o ambiente, assistentes para importar diagramas de classes em formato XMI e, a partir das classes identificadas como tipos principais, criar automaticamente as interfaces de negócios. En-

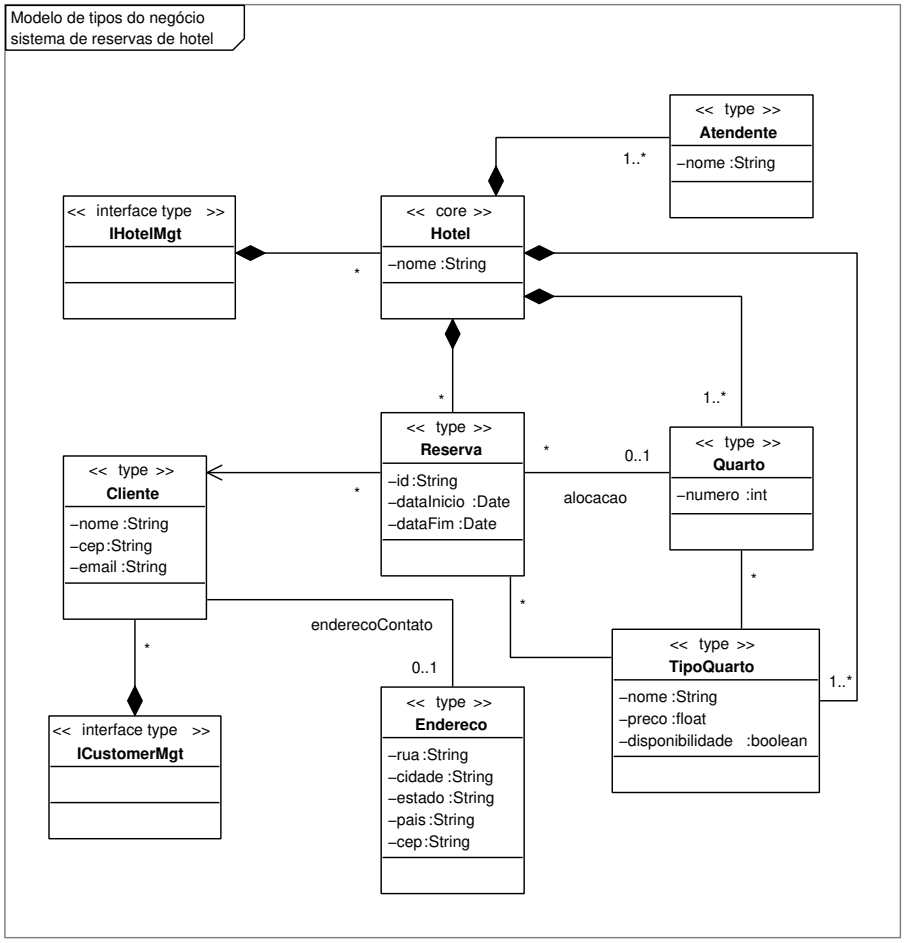

Figura 6.5: Modelo de tipos do negócio.

tretanto, tais assistentes ainda não estão implementados e as interfaces foram criadas manualmente.

Os componentes da camada de sistemas são aqueles que provêem as interfaces de sistema. Esses componentes são criados seguindo critérios de coesão e granularidade das funções de suas interfaces. Já os componentes da camada de negócios provêem as interfaces de negócios. Tais componentes são criados um para cada interface de negócios identificada. A especificação dos componentes foi criada utilizando o Editor de Componentes do ambiente. A Figura 6.6 mostra o componente de negócio `HotelMgr`, que provê a interface `IHotelMgt`.

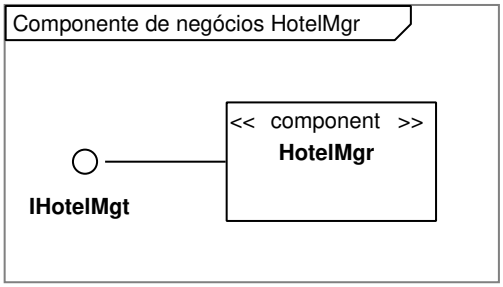

Figura 6.6: Componente de negócio `HotelMgr`.

Uma arquitetura de componentes inicial é criada a partir dos componentes especificados. Essa arquitetura poderá ser refinada nas fases posteriores do processo. A arquitetura foi criada utilizando o Editor de Arquiteturas do Bellatrix. As Figuras 6.7 e 6.8 mostram telas capturadas do editor de componentes e a Figura 6.9 mostra o editor de arquiteturas.

6.4.3 Interação de componentes (Fase 2.2)

Na fase anterior, Identificação de Componentes, foram identificadas as interfaces de sistema e de negócios, bem como os componentes que as provêem. Uma arquitetura de componentes inicial foi estabelecida. As operações das interfaces de sistema foram identificadas, mas não possuem assinatura definida; as interfaces de negócios não possuem operações.

Na fase de Interação de Componentes, as assinaturas das operações das interfaces de sistema serão definidas, as operações das interfaces de negócios serão identificadas e detalhadas e, eventualmente, a arquitetura de componentes é refinada. As operações das interfaces de sistema devem delegar à camada de negócios as atividades relacionadas às regras de negócios. Assim, para cada operação das interfaces de sistema, um diagrama de colaboração UML é criado, de modo a modelar a interação entre os componentes de sistema e de negócios, através de suas interfaces. Em paralelo, a operação da interface de

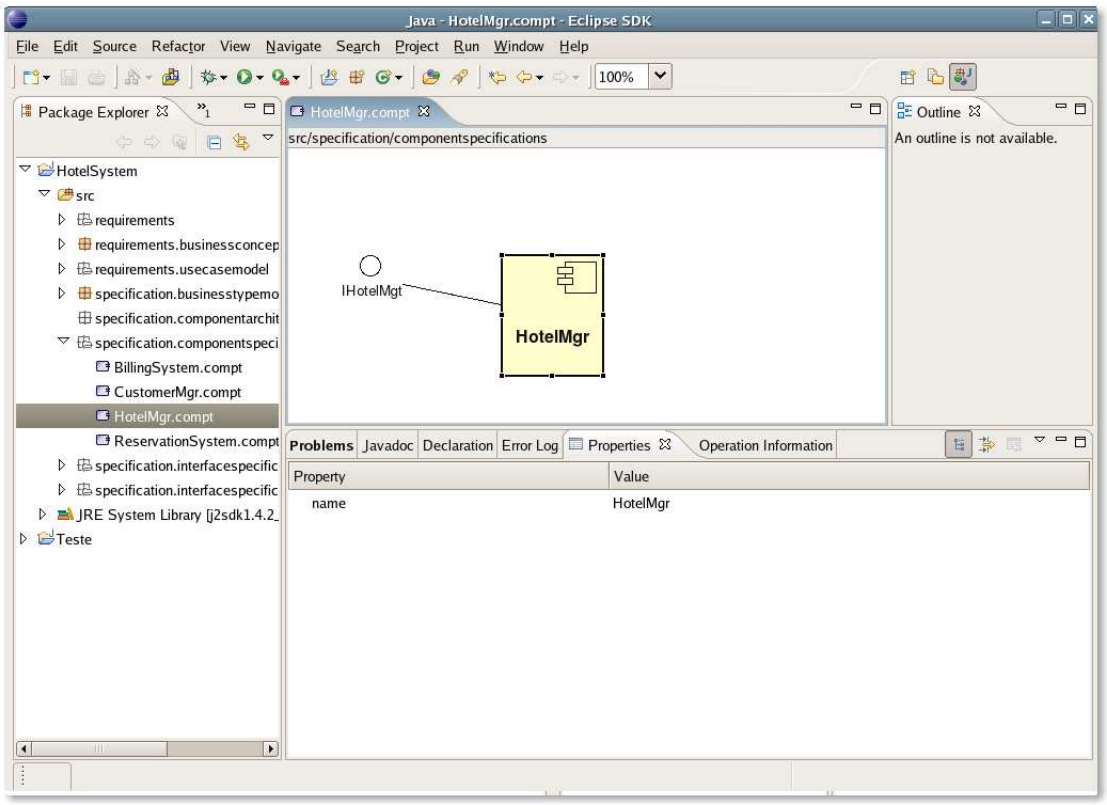

Figura 6.7: Editor de componentes do Bellatrix (1).

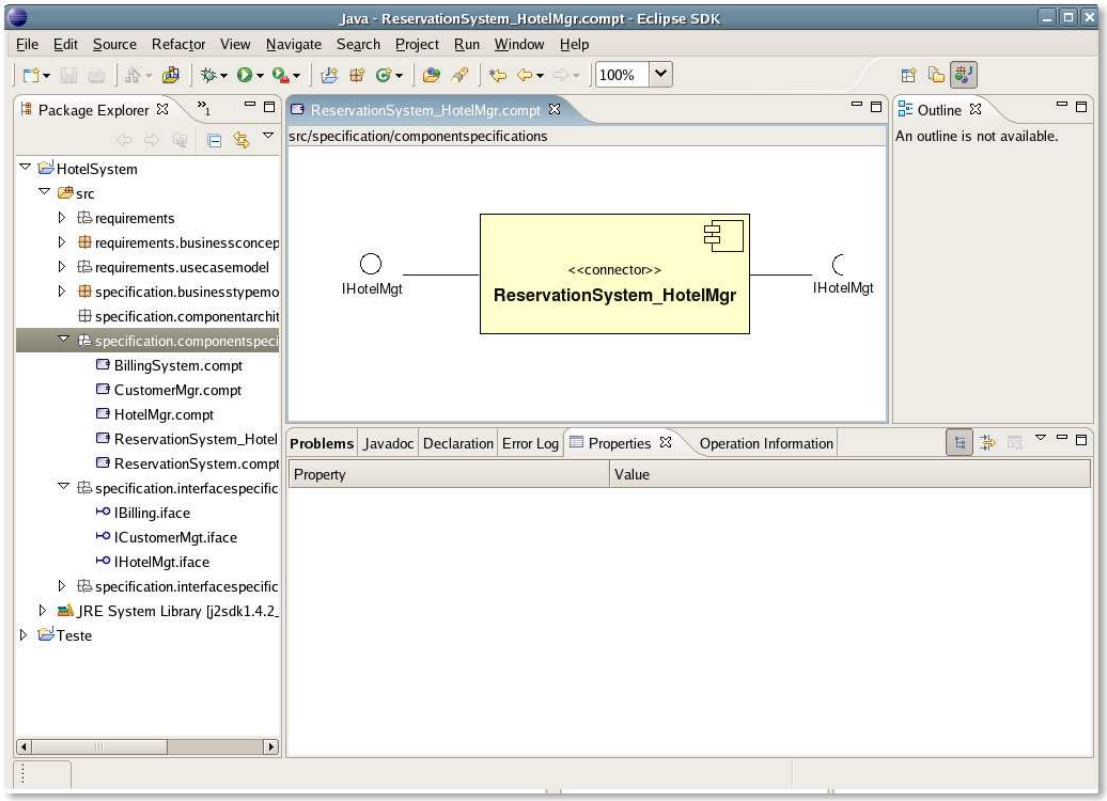

Figura 6.8: Editor de componentes do Bellatrix (2).

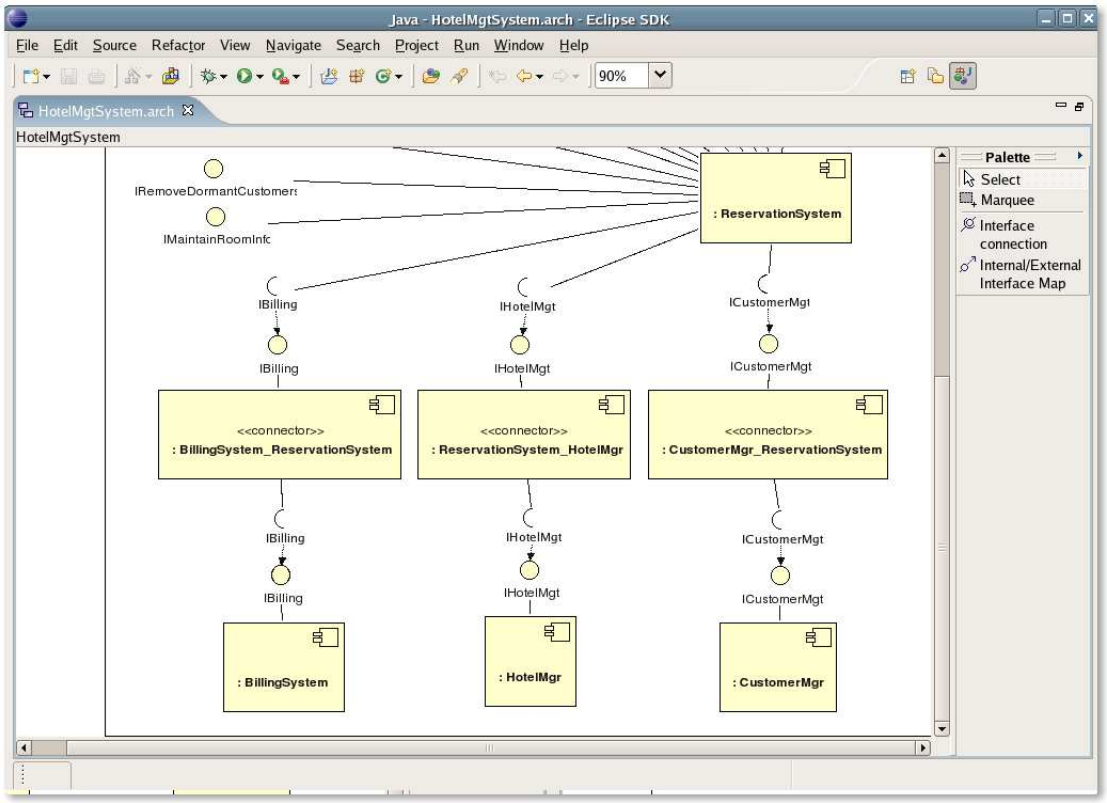

Figura 6.9: Editor de arquiteturas do Bellatrix.

sistema é detalhada, identificando os parâmetros de entrada e saída e seus tipos. Caso necessário, podem ser criados tipos de dados estruturados, como forma de agrupar e organizar as informações dos parâmetros.

Os diagramas de colaboração foram criados nas ferramentas de modelagem UML. Um exemplo desses diagramas podem ser visto na Figura 6.10. Os detalhes das operações podem ser inseridos através do Editor de Interfaces do Bellatrix. Um editor também foi criado para modelar tipos de dados estruturados, que podem ser utilizados nas assinaturas das interfaces. As Figuras 6.11 e 6.12 mostram telas capturadas desses editores.

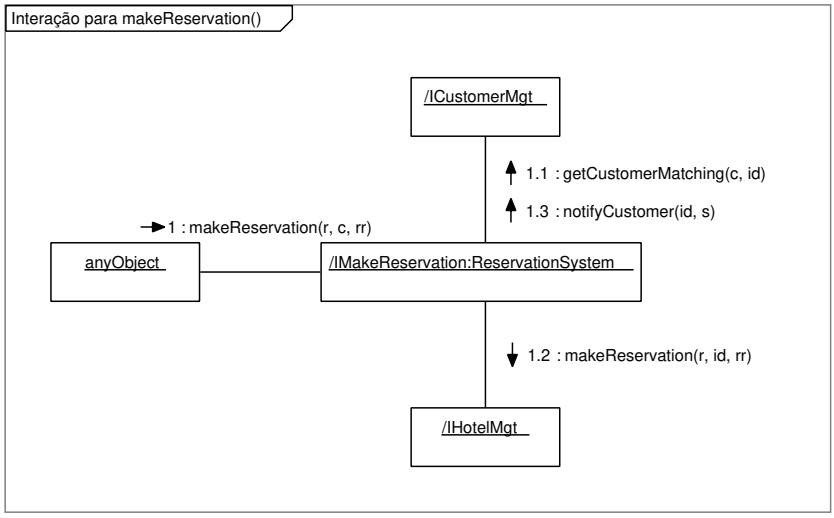

Figura 6.10: Exemplo de diagrama de colaboração para interação de componentes.

6.4.4 Especificação de componentes (Fase 2.3)

A fase de Especificação de Componentes envolve atividades de especificação detalhada das interfaces e dos componentes especificados até o momento, criando diagramas de especificação de interfaces e diagramas de especificação de componentes. Esses diagramas são importantes no contexto do UML Components. Seguindo o processo, as interfaces são especificadas juntamente com o modelo de tipos do negócio e as dependências dos componentes são especificados juntamente com a arquitetura do sistema. Esses dois artefatos possuem escopo de aplicação e, assim, é necessário a criação de uma documentação por interface e por componente, facilitando o futuro reuso de interfaces e componentes individualmente.

O diagrama de especificação de interfaces é criado a partir do modelo de tipos do negócio, e ele contêm os tipos de negócio diretamente manipulados pela interface, os tipos de dados das operações, definidos na fase anterior, e a interface em si. É criado um desses

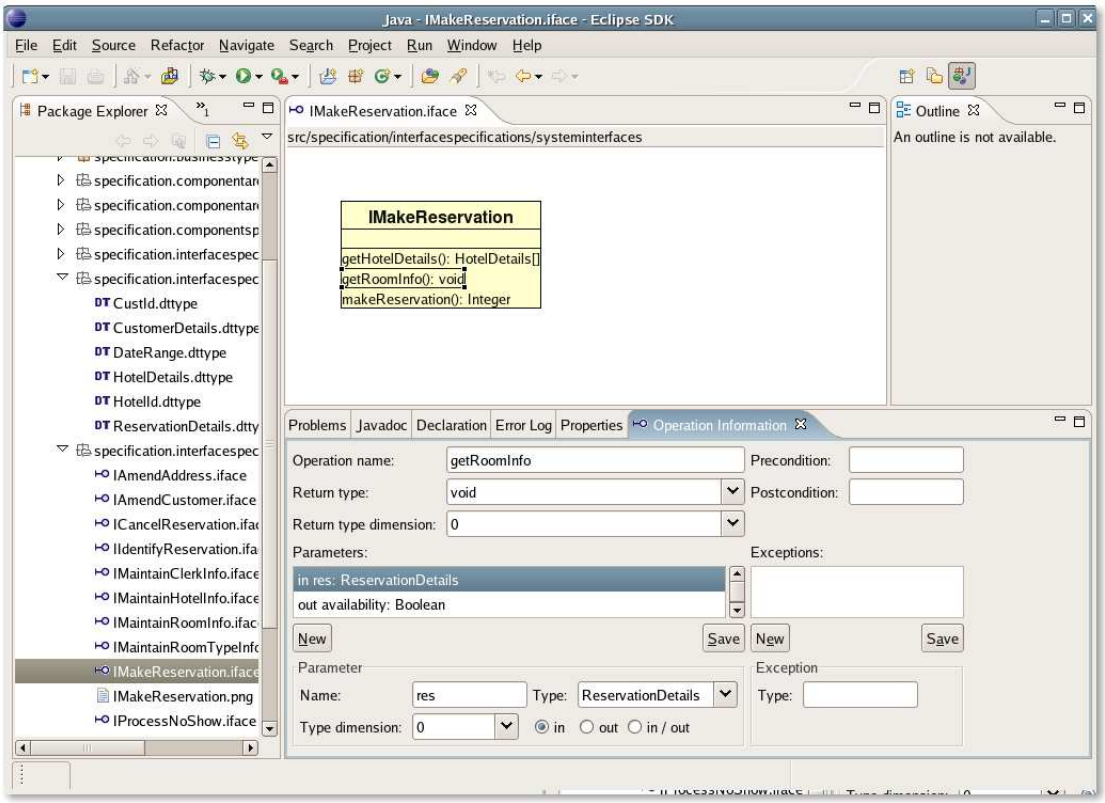

Figura 6.11: Detalhando operações no Bellatrix.

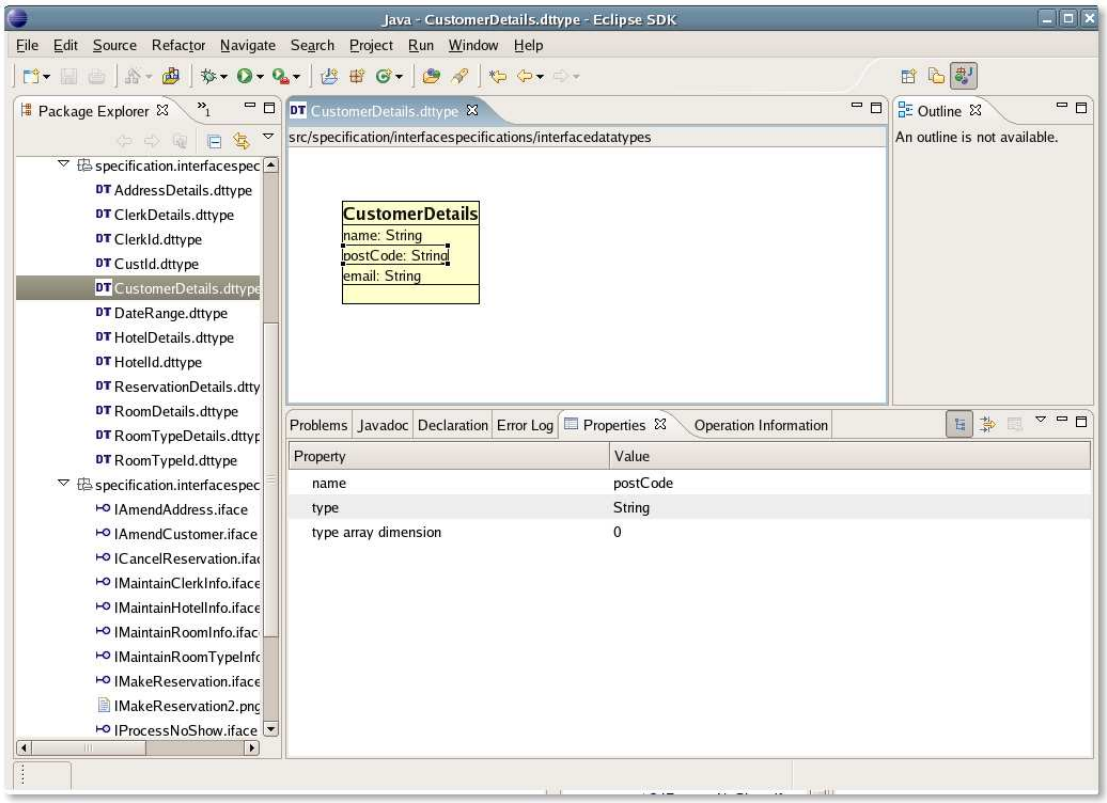

Figura 6.12: Editor de tipos de dados estruturados do Bellatrix.

diagramas por interface. Ainda, para cada operação das interfaces são criadas pré e pós condições em OCL. O diagrama de especificação de componentes é criado a partir do diagrama da arquitetura do sistema, um por componente, e mostra o componente e suas dependências em termos de interfaces — lembrando que o UML Components utiliza o conceito de dependência para isto.

No estudo de caso, não foram criados diagramas de especificação de interfaces, exatamente como propostos no processo, pois a especificação da interface e dos tipos de dados no Bellatrix são registrados em arquivos separados do modelo de tipos do negócio. O modelo de informações das interfaces foi criado em uma ferramenta de modelagem UML de modo similar ao proposto pelo processo, a partir de uma cópia do modelo de tipos do negócio, removendo os tipos fora do escopo da interface. Os diagramas de especificação de componentes não precisaram ser criados pois, diferente do processo, foram representadas as interfaces requeridas de cada componente juntamente com sua especificação.

6.4.5 Provisionamento (Fase 3) e Montagem (Fase 4)

A fase de provisionamento do processo é a etapa em que os componentes especificados são disponibilizados, seja através da aquisição, reuso ou desenvolvimento. Na fase seguinte, a fase de montagem, os componentes são ligados entre si, de acordo com sua arquitetura, e com a interface gráfica de usuário, formando a aplicação final. O processo UML Components não descreve essas fases com a riqueza de detalhes que faz para as fases anteriores.

Assim, como suporte à aquisição e reuso de componentes, estão especificadas funcionalidades de integração com um repositório de componentes, permitindo a busca e obtenção de componentes pré-existentes. Essas funcionalidades não estão implementadas atualmente, entretanto, existem trabalhos em andamento no Instituto de Computação da UNICAMP de desenvolvimento de um repositório de componentes que será futuramente integrado ao Bellatrix.

No sentido do desenvolvimento dos componentes especificados e da atividade de montagem, um conjunto de funcionalidades estão especificadas e implementadas no Bellatrix. Seguindo o estudo de caso, a partir da arquitetura de componentes, foi criada uma configuração utilizando um dos editores do ambiente. Uma configuração, nesse contexto, especifica um conjunto de implementações de componentes — ou componentes concretos — que materializam cada uma das especificações de componentes da arquitetura. Isso permite que, para uma mesma arquitetura, sejam criadas diferentes configurações variando a implementação dos componentes especificados. Uma configuração foi criada utilizando o Editor de Configurações do ambiente. A Figura 6.13 mostra uma tela desse editor.

Apesar do exemplo do sistema de reservas de hotel, conforme descrito originalmente,

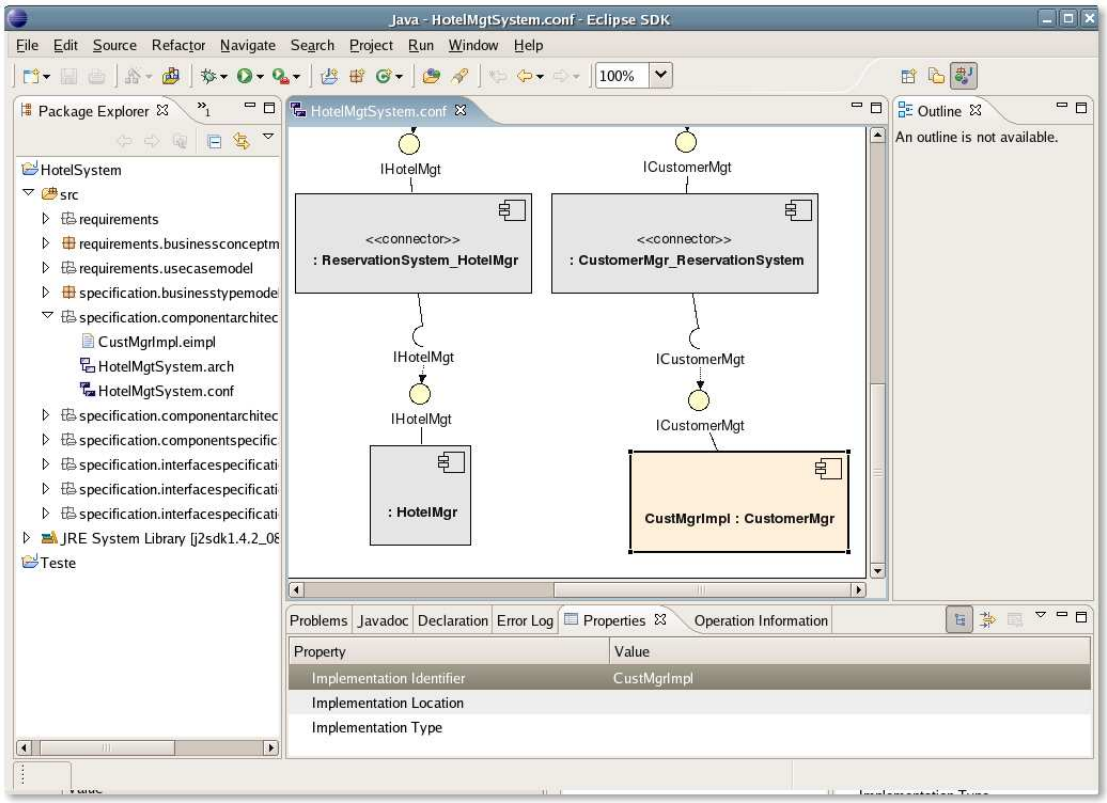

Figura 6.13: Editor de configurações do Bellatrix.

considerar o componente de cobranças *BillingSystem* como um componente externo já disponível, para este estudo de caso foi considerado que todos os componentes deveriam ser desenvolvidos. Nesse sentido, foi utilizado o gerador de código para criar esqueletos de componentes especificados, em conformidade com o modelo de implementação de componentes COSMOS. Dado que os artefatos de especificação criados pelo Bellatrix são persistidos em formato XMI, e a arquitetura interna do ambiente, é possível criar geradores de código para outros modelos de componentes.

6.5 Avaliação

Durante a condução do estudo de caso, foram feitas as seguintes considerações, agrupadas por fase do processo UML Components:

Levantamento de requisitos (Fase 1). A ferramenta EclipseUML, integrada com o ambiente Eclipse, dá total apoio à criação dos diagramas necessários para essa fase. A ferramenta Poseidon também permite o desenho de modo satisfatório dos diagramas mencionados. Uma limitação do EclipseUML é que sua versão gratuita não permite exportar os diagramas em formato de figura. Nenhuma das ferramentas oferece apoio à descrição dos passos dos casos de uso, de modo que essa descrição foi feita utilizando um editor de textos.

Identificação de componentes (Fase 2.1). A atividade de criar as interfaces manualmente, redigitando informações inseridas nas ferramentas de modelagem UML é uma atividade repetitiva que poderia ser amenizada com os assistentes do Bellatrix para importar os diagramas de casos de uso e de classes. Infelizmente tais assistentes estão especificados mas ainda não implementados. Mesmo utilizando somente as ferramentas de modelagem UML, desenvolvendo sem o ambiente Bellatrix, essa atividade envolveria redigitar informações em ambas as ferramentas UML utilizadas, EclipseUML e Poseidon. Como as ferramentas de modelagem UML não possuem suporte à descrição dos passos dos casos de uso, a criação das operações das interfaces de sistemas seria manual de qualquer modo. A ferramenta EclipseUML se mostrou menos adequada para essa fase do processo UML Components, visto que ela apresenta dificuldades para copiar diagramas, como no caso de refinar o modelo conceitual do negócio para o modelo de tipos do negócio, e para definir estereótipos novos, como no caso de identificar e marcar os tipos principais do negócio. Os editores de componentes e arquiteturas do Bellatrix se mostraram satisfatórios para essa fase do processo, permitindo ainda definir interfaces requeridas e conectores arquiteturais, conceitos não presentes no exemplo original. Ainda, o editor

de arquiteturas dá apoio à definição de sub-componentes, permitindo decomposição hierárquica.

Interação de componentes (Fase 2.2). Os diagramas de colaboração foram feitos sem dificuldades nas ferramentas de modelagem UML, apesar de não ser possível modelar mensagens sem um objeto de origem, como apresentado no livro. Os editores do Bellatrix inicialmente não permitiam que os tipos de parâmetros e retorno de operações fossem vetores. Essa deficiência foi corrigida, inclusive no editor de tipos de dados estruturados, e assim todas as operações puderam ser especificadas corretamente.

Especificação de componentes (Fase 2.3). As ferramentas de modelagem UML se mostraram adequadas para a criação do modelo de informações das interfaces, exceto pela dificuldade em se copiar diagramas na ferramenta EclipseUML. O fato dos editores do Bellatrix persistirem os modelos em arquivos separados auxiliou nessa fase, facilitando a criação de especificação separada por interface e por componente. Algumas limitações foram percebidas nesses editores, como a necessidade de se manter o caminho relativo entre os arquivos de especificação no caso de se utilizar um repositório e o apoio ao uso de OCL para as pré-condições e pós-condições, que poderia ser melhorado com maior espaço para modelagem e a criação ou adaptação de um validador para as restrições criadas. Ainda, não é possível exportar as representações gráficas dos editores do Bellatrix para arquivos em formato imagem.

Provisionamento (Fase 3) e Montagem (Fase 4). O gerador de código criou os esqueletos dos componentes e conectores COSMOS, bem como o seu código de instanciação, responsável por instanciar e conectar os componentes em tempo de execução. A operação da camada de sistema `IMakeReservation#makeReservation()` foi implementada para verificar o funcionamento do código gerado.

6.6 Resumo

Este capítulo descreve um estudo de caso conduzido para avaliar o ambiente Bellatrix. Esse estudo foi realizado pelo autor desse trabalho, modelando um sistema de reservas de hotel que consta como exemplo do livro de Cheesman *et al* sobre o processo UML Components. Assim, foi utilizada a arquitetura em camadas, prescrita pelo processo, com as camadas de sistema e negócios. Foram utilizadas as ferramentas de modelagem UML EclipseUML e Poseidon para a criação dos diagramas propostos pelo processo mas não cobertos pelo Bellatrix.

Na fase de levantamento de requisitos, o processo do negócio foi modelado utilizando diagramas de atividades UML. Um modelo conceitual do negócio também foi criado, utilizando diagramas de classes UML. A partir da modelagem do negócio, um diagrama de casos de uso foi criado, bem como a descrição dos mesmos. Os casos de uso foram detalhados em relação ao exemplo original, para deixar o sistema mais completo. As ferramentas de modelagem UML se mostraram satisfatórias para essa etapa, apesar de nenhuma das ferramentas possuir apoio para a descrição dos passos dos casos de uso.

Para a fase de identificação de componentes, as ferramentas UML foram utilizadas para refinar o modelo conceitual do negócio, criando o modelo de tipos do negócio e marcar os tipos principais com estereótipos. Os editores do Bellatrix foram utilizados para criar as especificações das interfaces de sistema e negócios, as especificações de componentes e a arquitetura inicial do sistema, de modo gráfico. A ferramenta EclipseUML mostrou dificuldades para copiar diagramas e impossibilidade para definir estereótipos novos. No Bellatrix, os assistentes para importar os casos de uso e o modelo de tipos do negócio, auxiliando a criação das interfaces, estão especificados mas não foram implementados. Entretanto, como as ferramentas UML não dão apoio à descrição dos passos dos casos de uso, a criação das operações das interfaces de sistema criadas a partir desses passos ainda não poderiam ser automatizados. Os editores possuem conceitos de DBC não utilizados pelo UML Components, como interfaces requeridas e decomposição hierárquica.

Na fase de interação de componentes, as assinaturas das operações foram especificados através da especificação da interação entre os componentes. Essa interação foi modelada em diagramas de colaboração UML. Assim, foram especificadas as operações e seus tipos de dados de parâmetros e retorno nos editores do Bellatrix. As ferramentas UML se mostraram adequadas para essa fase.

Na fase de especificação de componentes, foi criada documentação para as interfaces e componentes especificados, individualmente. Os editores do Bellatrix persistem os modelos em arquivos isolados, o que facilita essa atividade. O modelo de informação das interfaces, parte do modelo de tipos do negócio, foi criado utilizando uma ferramenta UML. Algumas limitações do Bellatrix foram a necessidade de se manter o caminho relativo entre os arquivos de especificação, o baixo apoio a pré e pós condições em OCL e a impossibilidade se se salvar as especificações em formato de imagem.

As fases de provisionamento e montagem são as fases em que os componentes são disponibilizados, via aquisição, reuso ou desenvolvimento e ligados entre si, de acordo com sua arquitetura. O Bellatrix foi utilizado para criar uma especificação da configuração do sistema, ou seja, quais implementações serão utilizadas para atender às especificações de componentes da arquitetura. Ainda, um gerador de código foi utilizado para gerar esqueletos de código de acordo com o modelo de implementação de componentes COSMOS.

No próximo capítulo, outro estudo de caso conduzido é apresentado. Nesse estudo de

caso, um sistema financeiro baseado em componentes foi modelado no ambiente Bellatrix.

Capítulo 7

Estudo de Caso: Sistema Financeiro

Este capítulo apresenta um estudo de caso que foi realizado para avaliar a implementação atual do ambiente Bellatrix, com foco em suas funcionalidades e sua usabilidade. A Seção 7.1 apresenta o objetivo do estudo de caso. Na Seção 7.2 são descritos os materiais e os métodos utilizados na condução do estudo de caso. A Seção 7.3 descreve as funcionalidades do sistema financeiro utilizado, incluindo seus casos de uso. A especificação desse sistema é apresentada na Seção 7.4. A Seção 7.5 apresenta sua execução e a Seção 7.6 discute seus resultados. Por fim, a Seção 7.7 sumariza este capítulo.

7.1 Objetivo

Este estudo de caso foi conduzido com o objetivo de avaliar o ambiente Bellatrix com foco no uso de sua primeira implementação, verificando se as funcionalidades mínimas para sua adoção por usuários finais foram implementadas e também a usabilidade dessas funcionalidades. As funcionalidades presentes nessa primeira implementação do ambiente são os Editores de (1) exceções, (2) tipos de dados (*datatypes*), (3) interfaces, (4) componentes, (5) arquiteturas, (6) configurações, e (7) Gerador de código no modelo COSMOS. Ainda, buscou-se identificar a necessidade da implementação de novas funcionalidades, especificadas ou não, e eventuais deficiências e falhas na implementação atual.

7.2 Material e Método

Material:

- O estudo de caso foi realizado por quatro voluntários, todos alunos de pós-graduação deste Instituto, alguns com experiência profissional na indústria de software. Nenhum dos voluntários havia utilizado anteriormente o ambiente e todos eles conhe-

ciam o processo UML Components. Identificamos os voluntários pelas letras de A até D e os descrevemos a seguir:

  - O voluntário A é aluno de mestrado e possui experiência profissional com desenvolvimento de software.
  - O voluntário B é aluno de mestrado, sem experiência profissional.
  - O voluntário C é aluno de doutorado e atua profissionalmente na indústria de software.
  - O voluntário D é aluno de doutorado, sem experiência profissional.

- A especificação de um sistema financeiro real baseado em componentes, apresentada nas Seções 7.3 e 7.4, foi utilizada. Essa especificação foi modelada por uma empresa de médio porte e apresentada no contexto de outros trabalhos [12, 56].

- Um questionário foi elaborado com base em testes de usabilidade [58] e utilizado para a avaliação do estudo de caso. Esse questionário está presente no Apêndice A e procura levantar informações sobre o perfil do usuário, o uso do ambiente em termos das funcionalidades presentes, a facilidade de aprendizado, produtividade e sistema de ajuda. Ainda, coletam-se sugestões de melhorias e problemas.

- O ambiente computacional utilizado foi o Laboratório de Sistemas Distribuídos (LSD) do Instituto de Computação, UNICAMP.

Método:

Foi disponibilizada em área pública uma instalação do ambiente Bellatrix, a especificação do sistema financeiro e o questionário de avaliação. Os voluntários instalaram e utilizaram o ambiente para modelar o sistema, sem nenhum auxílio direto. Após essa tarefa, eles preencheram o questionário de avaliação.

7.3 Descrição do Sistema Financeiro

O sistema em questão pertence ao domínio de sistemas financeiros bancários, mais especificamente, o controle sobre o uso de talões de cheques. A especificação desse sistema é real e desenvolvida por uma empresa de médio porte que atua na construção de sistemas para a área financeira. Essa especificação foi produzida seguindo o processo MDCE+ [12] — um processo para desenvolvimento de sistemas DBC confiáveis baseado no processo UML Components [15] — e descrita em trabalhos anteriores [56]. Foram especificados para ele seis funcionalidades básicas, mostradas a seguir:

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.** Essa funcionalidade consiste na retirada física de talões de cheque solicitados anteriormente.

3. **Sustação de cheques.** O cliente pode cancelar um determinado número de cheques, em caso de perda, roubo ou outro motivo específico.

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.

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.

Um diagrama contendo os casos de uso identificados para esse sistema é apresentado na Figura 7.1. A descrição dos principais casos de uso é apresentada a seguir:

Requisitar talão de cheque. O usuário registra a solicitação de talão para o cliente. O cliente deve possuir uma conta cadastrada em uma agência aberta. O cliente deve ser o titular da conta e sua conta não pode estar bloqueada para requisição de talão.

Entregar talão de cheque. O operador registra a entrega do talão de cheques ao cliente. O cliente deve possuir uma conta cadastrada em uma agência aberta. Deve existir o talão em estoque e a não pode estar bloqueada para a entrega do talão.

Sustar cheques. O operador registra a sustação de um ou mais cheques. A conta relativa ao cheque deve estar cadastrada em uma agência aberta. Deve ser informado um motivo para a sustação (por exemplo, roubo do malote, perda, roubo, outros) e a situação dos cheques (por exemplo, em branco, preenchido e datado, preenchido e não datado). Os cheques não podem estar previamente sustados.

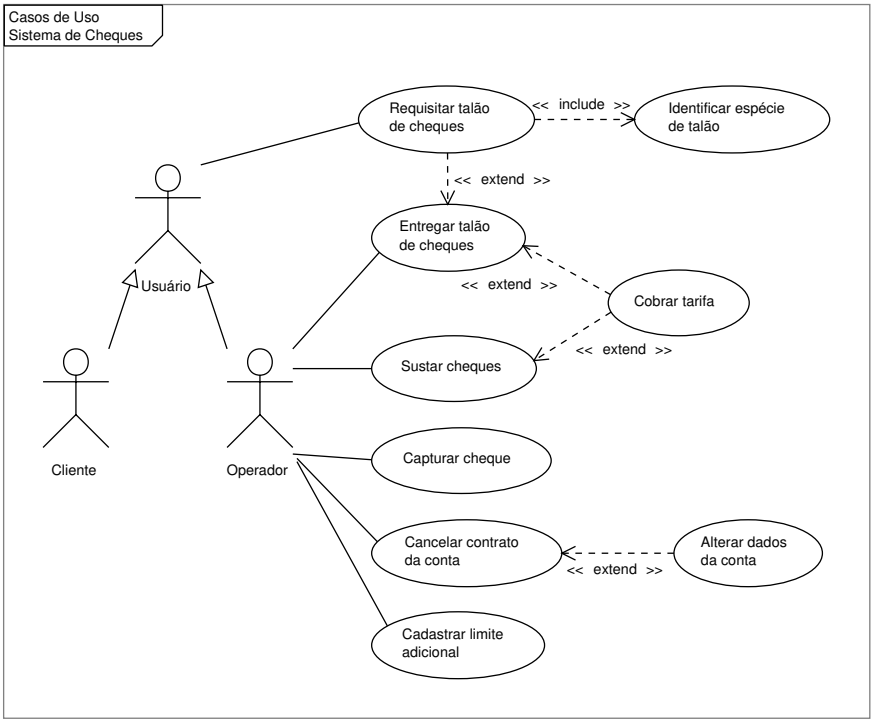

Figura 7.1: Diagrama de casos de uso do sistema.

Capturar cheques. O operador registra os cheques capturados em uma operação de depósito em cheque. A operação de captura é identificada por um número identificador e registra a data do movimento. O cheque deve ter seus dados válidos e não pode ter sido previamente capturado. A instituição financeira que emitiu o cheque deve estar "funcionando" no sistema de compensação bancária.

Cancelar contrato da conta. O operador efetua o cancelamento do contrato da conta. Se a conta foi cadastrada no mesmo dia do cancelamento, efetua-se uma alteração de modalidade e das respectivas taxas para essa conta. A conta em questão deve estar cadastrada e não cancelada em uma agência aberta.

Cadastrar limite adicional. O operador cadastra um limite de crédito adicional para a conta do cliente. A conta em questão deve estar cadastrada em uma agência aberta. O limite deve ser maior que zero e o tipo de conta deve permitir o uso de limite adicional.

7.4 Especificação do Sistema

A arquitetura adotada para o sistema segue restrições impostas pelo ambiente da empresa, como por exemplo o uso de componentes utilitários e de componentes pré-existentes de acesso a dados. A Figura 7.2 apresenta essa arquitetura. Nessa figura, pode-se observar que a arquitetura segue um estilo em camadas relaxado, com camadas utilitárias sendo acessadas pelas camadas da aplicação. A camada de dados contém componentes que oferecem operações de acesso às bases de dados utilizadas. Os componentes da camada de utilitários provêem serviços de menor granularidade e de interesses horizontais, tais como validadores de formulários. Os componentes da camada framework oferecem serviços de maior granularidade, como por exemplo a comunicação com sistemas legados. As camadas de sistema e negócio são análogas às camadas de mesmo nome introduzidas pelo processo UML Components.

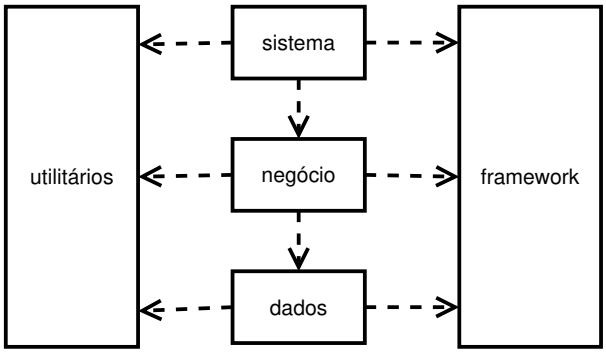

Figura 7.2: Arquitetura do sistema.

As interfaces providas em cada camada da arquitetura do sistema foram descobertas seguindo o processo de desenvolvimento MDCE+. A Figura 7.3 apresenta as interfaces da camada de negócios. Ainda, essas interfaces utilizam tipos de dados estruturados em suas operações. Esses tipos de dados são apresentadas na Figura 7.4.

A especificação das interfaces foi simplificada, com a eliminação de parte das exceções lançadas pelas operações, visando a redução do tamanho da especificação. Foram mantidas apenas as exceções da interface `ISBContaMgrReq`, que foi considerada responsável por requisitos críticos de disponibilidade. A Figura 7.5 apresenta tais exceções.

A identificação dos componentes levou em consideração o reuso de componentes pré-existentes. A Figura 7.6 apresenta os componentes da camada de negócios. Nessa figura, podem ser observados os componentes 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. Ainda, são mostradas as interfaces providas e requeridas de cada um desses componentes.

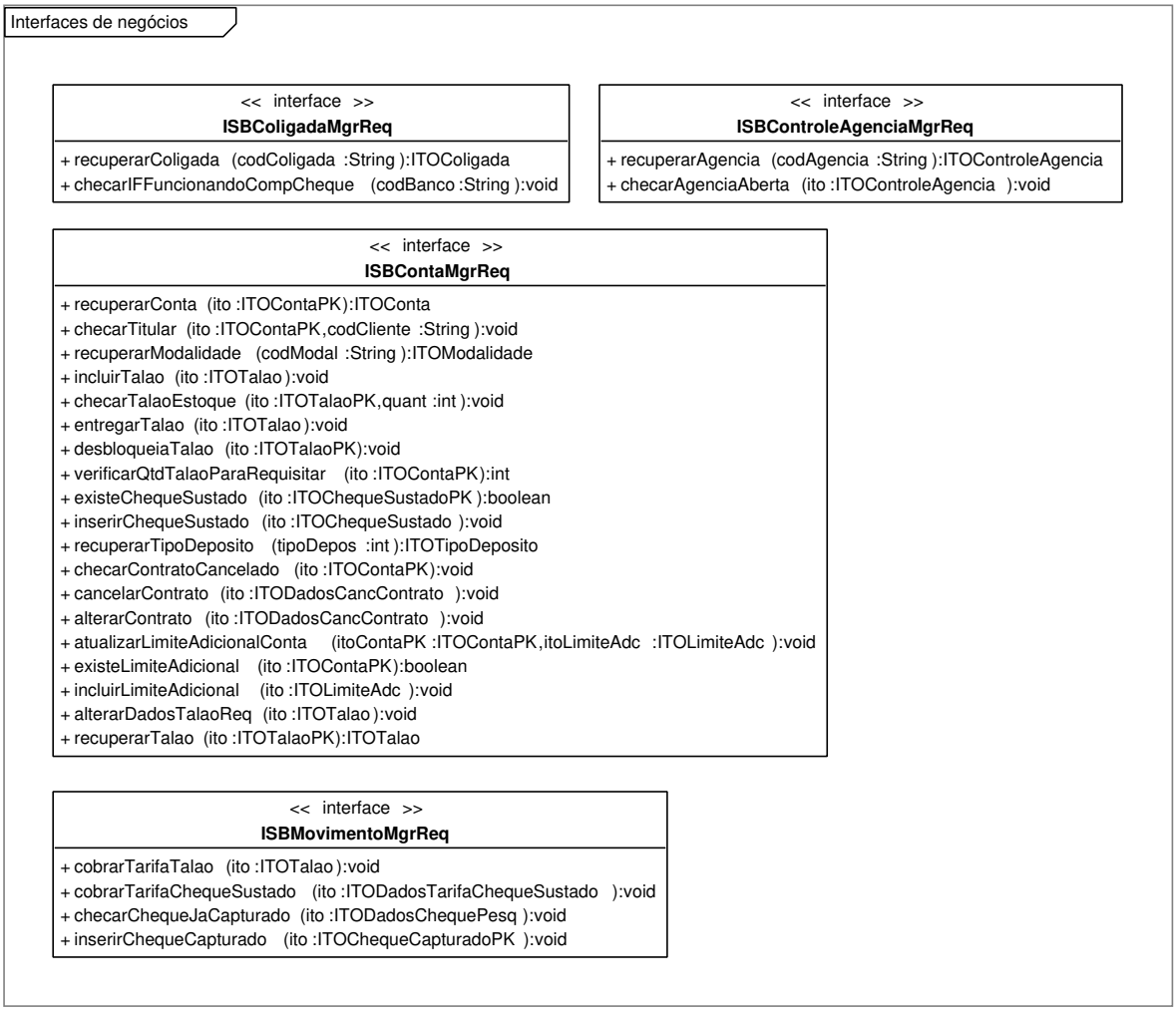

Figura 7.3: Interfaces da camada de negócios.

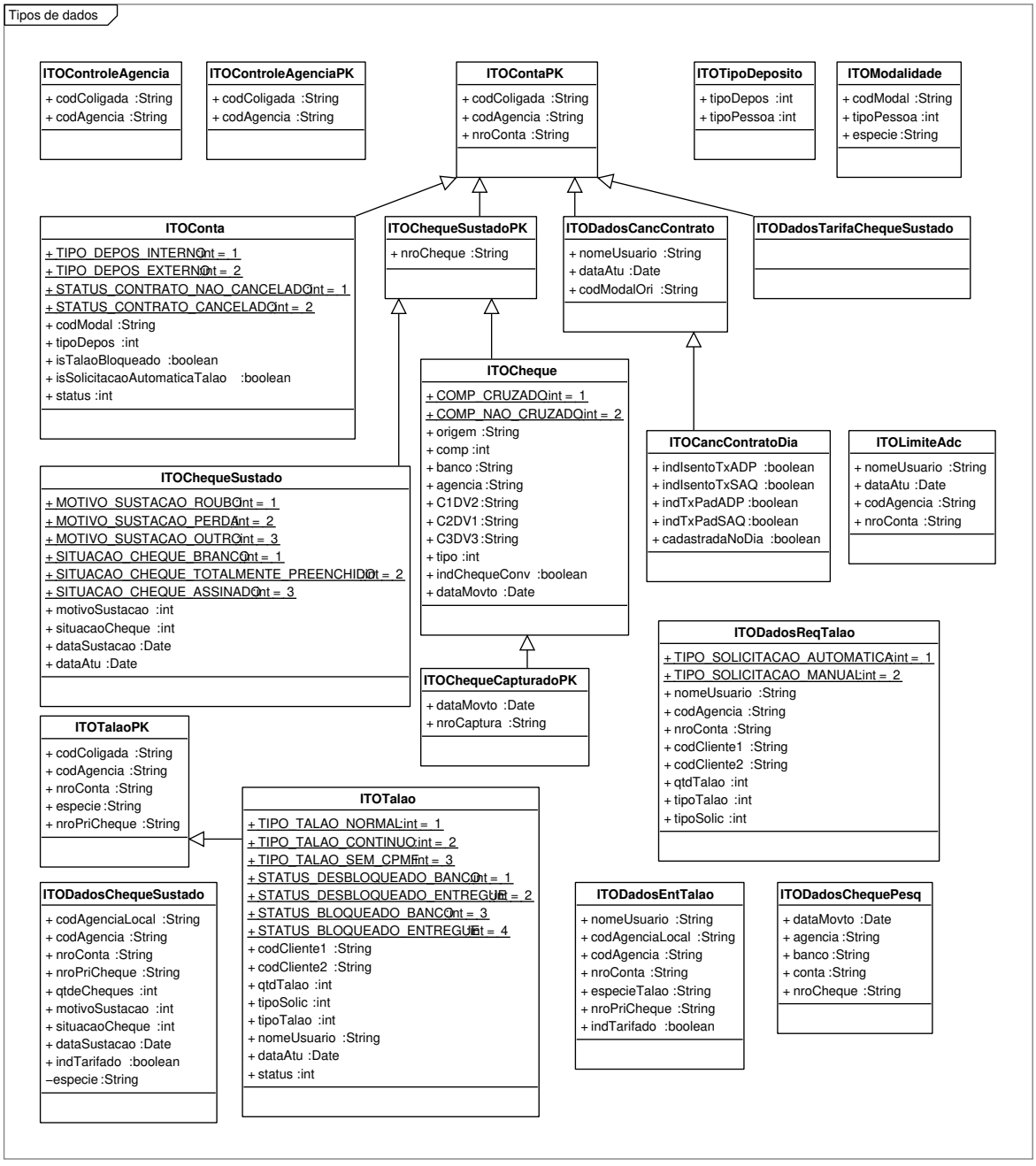

Figura 7.4: Tipos de dados (*datatypes*) do sistema.

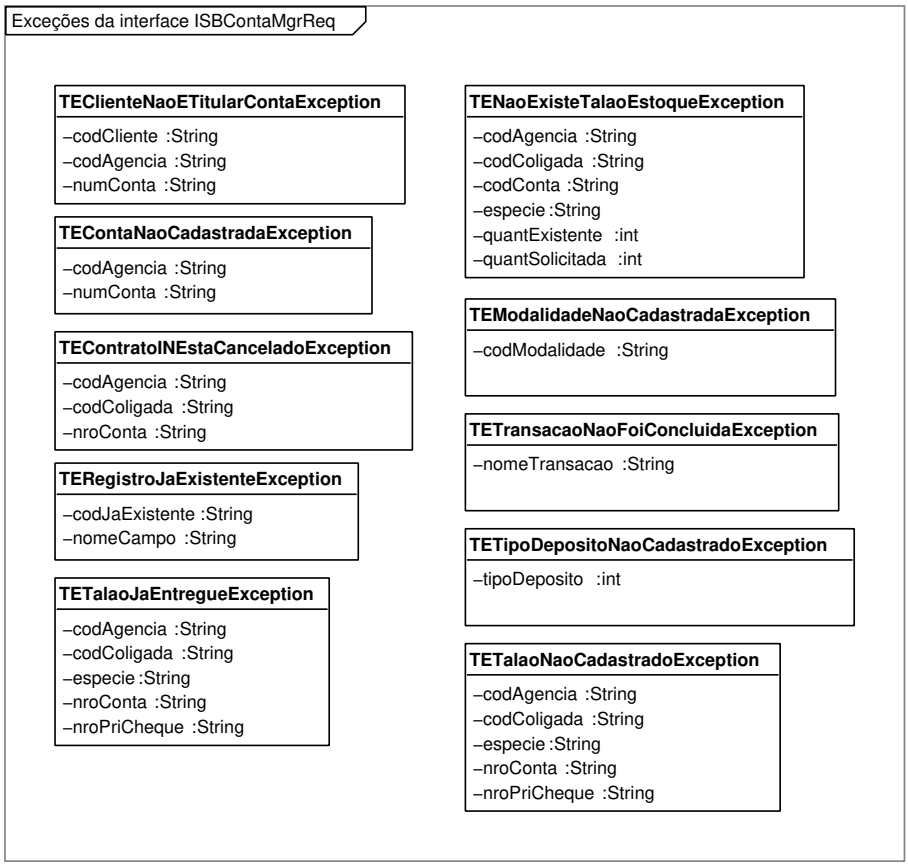

Figura 7.5: Exceções lançadas pela interface `ISBContaMgrReq`.

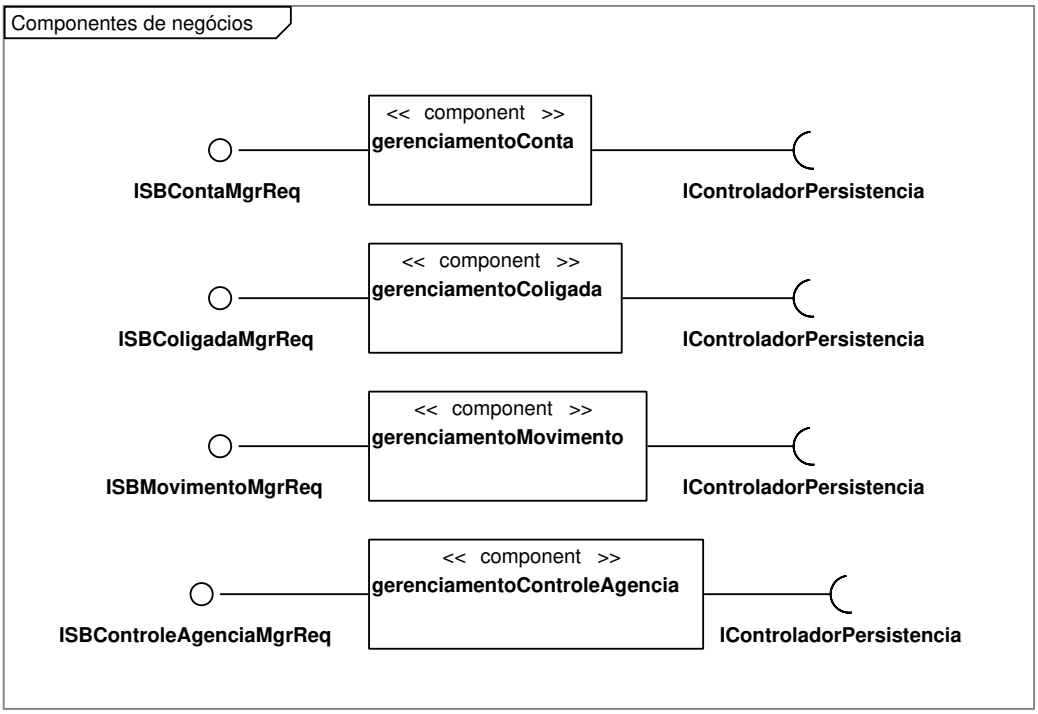

Figura 7.6: Componentes da camada de negócios.

Os componentes do sistema financeiro foram organizados na arquitetura de componentes mostrada da Figura 7.7. Essa arquitetura refina a arquitetura mostrada na Figura 7.2, mostrando os componentes de cada uma das camadas. Por motivo de clareza, a figura abstrai os conectores, que foram criados seguindo 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 componentes `operacoesConta`, `operacoesChequeSustado` e `gerenciamentoConta`. O componente `gerenciamentoConta` foi assim replicado na arquitetura mostrada na Figura 7.7. Os conectores possuem funcionalidades para detectar a falha em um dos componentes e passar a utilizar a réplica. Os componentes `operacoesConta` e `operacoesChequeSustado` possuem uma arquitetura interna que pode ser observada na Figura 7.8. Essa arquitetura interna foi baseada no componente ideal tolerante a falhas [39], onde um sub-componente **normal** concretiza o comportamento normal do componente e outro sub-componente **tratador** concretiza o comportamento excepcional, sendo responsável pelo tratamento de exceções. Outros componentes da arquitetura também possuíam requisitos críticos de qualidade. Entretanto, desconsideramos tais requisitos visando reduzir o tamanho da especificação do estudo de caso.

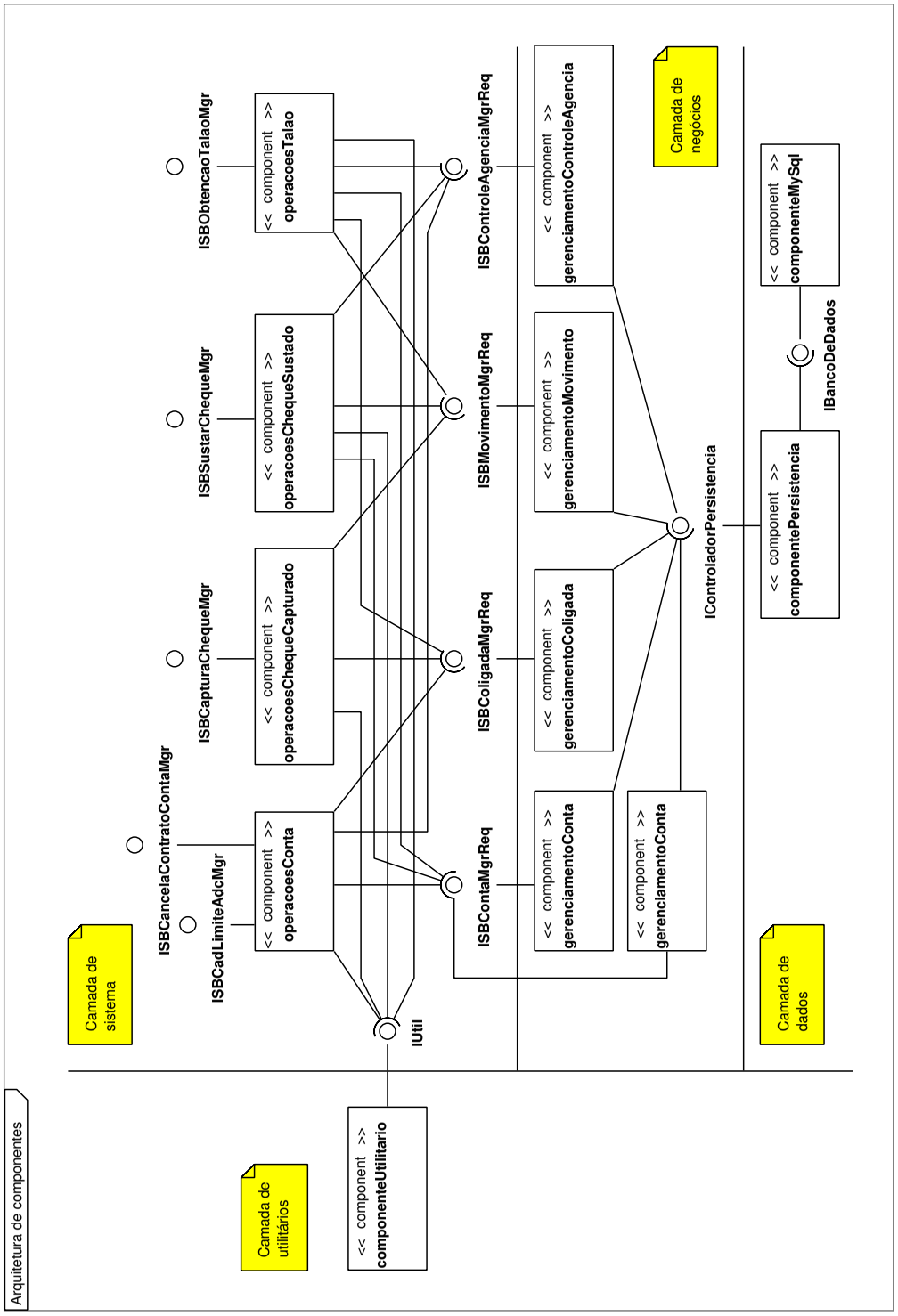

Figura 7.7: Arquitetura de componentes do sistema.

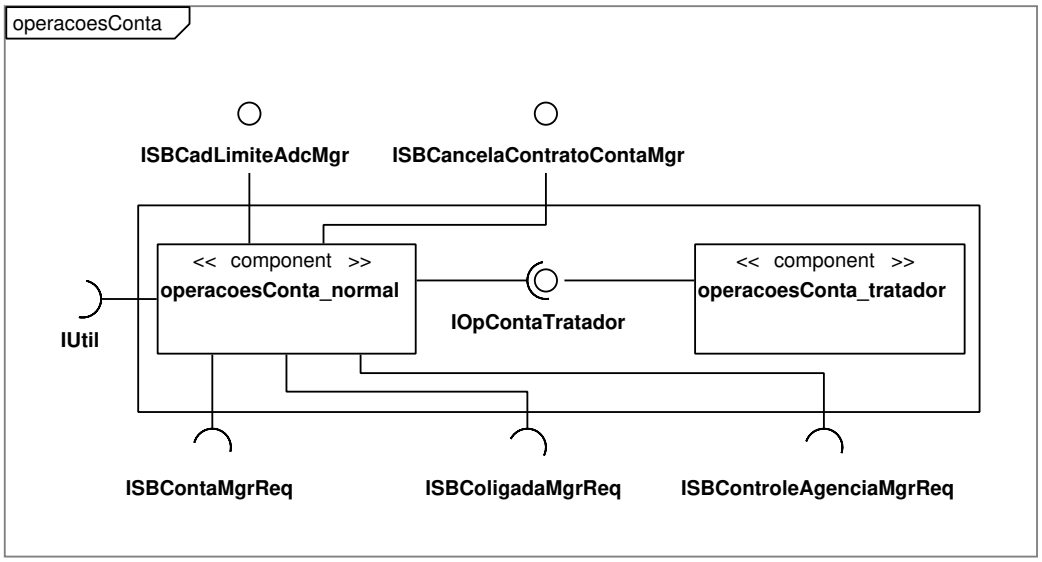

Figura 7.8: Arquitetura interna do componente `operacoesConta`.

7.5 Execução

A partir da especificação apresentada na Seção 7.4 os voluntários utilizaram o ambiente Bellatrix, modelando todo o sistema, e preencheram o questionário de avaliação. A primeira parte deste questionário identificou o perfil do usuário e o seu uso do ambiente e do sistema de ajuda. Essas informações são apresentadas a seguir, utilizando a identificação dos usuários por letras:

- O voluntário A, aluno de mestrado com experiência profissional, utilizou as ferramentas para definir exceções, tipos de dados, interfaces, componentes e arquiteturas, criando 71 artefatos em cerca de 6 horas. Ele se auto-avaliou como conhecedor dos conceitos de DBC envolvidos no uso do Bellatrix e recorreu ao sistema de ajuda (*help*) 3 vezes, nem sempre encontrando a informação que necessitava.

- O voluntário B, aluno de mestrado, utilizou as ferramentas para definir exceções, tipos de dados, interfaces, componentes e arquiteturas e criou 96 artefatos em aproximadamente de 5 horas. Ele se auto-avaliou como conhecedor dos conceitos de DBC envolvidos no uso do ambiente e não utilizou o sistema de ajuda.

- O voluntário C, aluno de doutorado atuando profissionalmente na indústria de software, utilizou todas as ferramentas disponíveis para a modelagem do sistema, inclusive gerando código no modelo COSMOS para o mesmo. Desconsiderando o código gerado, foram criados 92 artefatos em aproximadamente 5 horas e meia. Ele se

auto-avaliou como possuindo os conhecimentos mínimos para o uso do ambiente e utilizou o sistema de ajuda diversas vezes, nem sempre encontrando a informação que necessitava.

- O voluntário D, aluno de doutorado, utilizou todas as ferramentas do ambiente na modelagem do sistema e gerou código no modelo COSMOS para o mesmo. Ele criou 106 artefatos, excluindo o código gerado, e utilizou 4 horas para tal. Ele se auto-avaliou como tendo pleno domínio dos conceitos de DBC para o uso do Bellatrix e recorreu ao sistema de ajuda 2 vezes, considerando o sistema útil para sanar suas dúvidas.

Em seguida, foi coletada a opinião dos voluntários sobre diferentes quesitos do ambiente. Eles atribuíram notas de 1 (péssimo) a 5 (ótimo) para cada um dos quesitos, conforme mostrado na Tabela 7.1.

Tabela 7.1: Notas atribuídas pelos voluntários ao Bellatrix.

Quesito	**usuário A**	**usuário B**	**usuário C**	**usuário D**
Funcionalidades presentes no ambiente	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

7.6 Avaliação

Os voluntários avaliaram o ambiente Bellatrix interessante por abranger os conceitos de DBC, dando apoio à criação de componentes, sub-componentes e configurações, comparado com as ferramentas utilizadas atualmente por eles. Entretanto, críticas foram feitas para a especificação de interfaces, considerada mais fácil de ser realizada na ferramenta Poseidon [29], utilizada por alguns deles. O ambiente foi considerado produtivo para a modelagem da especificação do estudo de caso. Entretanto, devido a limitações técnicas da implementação utilizada, foram identificadas dificuldades para o refinamento sucessivo dos artefatos criados, e mesmo para a correção de alguns deles. Ainda, foi mencionada a impossibilidade de se especificar estilos arquiteturais e modelos de domínio no ambiente Bellatrix, como é possível em ferramentas como o AcmeStudio [59] e Odyssey [11].

Com base nos questionários, foram identificados os seguintes pontos de melhoria no ambiente:

Robustez da ferramenta: Foram encontrados 5 erros no código do ambiente (*bugs*) que causaram que funções não fossem executadas normalmente. O ambiente Eclipse se manteve estável apesar das partes relacionadas ao erro não funcionar adequadamente.

Funcionalidade: Foram encontrados 29 pontos que necessitam melhorias em termos de funcionalidades. Desses pontos, 12 foram sobre inclusão de maior apoio para edição gráfica, tais como operações de *drag and drop* entre editores, posicionamento automático de elementos dentro dos diagramas, e a possibilidade de ocultar conectores numa arquitetura, entre outros. Ainda, outros 6 pontos identificados diziam respeito a problemas editando tipos de dados, como por exemplo a dificuldade para a inclusão de alguns tipos de dados não nativos disponíveis na linguagem Java e a impossibilidade de se definir visibilidade nos atributos dos tipos de dados.

Usabilidade: Foram apresentados pelos voluntários 8 sugestões de melhorias de usabilidade, tais como inclusão de teclas de atalho, alteração de cores e de disposição de campos na tela.

Sistema de ajuda: Foram identificados também 9 pontos do sistema de ajuda que necessitam de atualização e ampliação. Entre esses itens, priorizou-se a correção dos erros no código do ambiente.

7.7 Resumo

Este capítulo apresentou um estudo de caso conduzido para avaliar o ambiente Bellatrix. Esse estudo foi realizado por quatro alunos de pós-graduação do Instituto de Computação da UNICAMP, alguns com experiência profissional, e consistiu em modelar um sistema financeiro real no ambiente. Após essa tarefa, os voluntários preencheram um questionário sobre as funcionalidades e a usabilidade do ambiente.

O sistema financeiro, mais especificamente, controla o uso de talões de cheques, possuindo as seguintes funcionalidades: (1) requisição de talões de cheques; (2) entrega de talões de cheques; (3) sustação de cheques; (4) captura de cheque para compensação; (5) cancelamento do contrato da conta; e (6) cadastramento de limite adicional. Ele foi especificado utilizando o processo de DBC MCDE+, com pouco mais de dez componentes e possuindo requisitos críticos de disponibilidade. Esses requisitos foram abordados utilizando tratadores de exceções e réplicas estruturados na arquitetura do sistema.

Com base nos questionários, no geral observa-se que os voluntários se consideraram conhecedores dos conceitos envolvidos no ambiente. Eles sentiram a necessidade de recorrer

ao sistema de ajuda, obtendo apoio razoável do mesmo. O sistema proposto foi totalmente especificado e gerou em torno de 90 artefatos, levando em torno de 5 horas para ser modelado. Os usuários avaliaram a ferramenta no geral como boa, e diferentes pontos de melhorias em funcionalidades foram identificados, especialmente inclusão de maior apoio gráfico e de especificação de interfaces. Também foram identificados pontos de melhorias de usabilidade, tais como inclusão de teclas de atalho, alteração de disposição de campos na tela e melhoramentos no sistema de ajuda. Ainda, foram identificados defeitos no ambiente que causaram seu mal funcionamento.

No próximo capítulo, são feitas considerações finais sobre este trabalho, incluindo conclusões, contribuições e trabalhos futuros.

Capítulo 8

Conclusões e Trabalhos Futuros

Este trabalho apresentou o ambiente Bellatrix, um ambiente que dá suporte ao DBC com ênfase na arquitetura de software e estende o ambiente integrado de desenvolvimento Eclipse. O ambiente é centrado na arquitetura de software e provê suporte ao processo de DBC. Um dos principais objetivos do Bellatrix é facilitar a transição de um projeto de arquitetura de software para a implementação do sistema, de acordo com o paradigma DBC. Inicialmente foi utilizado o processo de DBC UML Components e o modelo de implementação de componentes COSMOS.

No sentido de especificar o Bellatrix foram desempenhadas atividades de levantamento de requisitos, projeto e implementação do ambiente. Seu desenvolvimento é baseado em componentes e seguiu o processo UML Components. Dentre as atividades de levantamento de requisitos, buscou-se trabalhos que descrevessem requisitos para ambientes DBC e trabalhos relacionados. A modelagem do processo do negócio consistiu num conjunto de diagramas de atividades modelando o processo UML Components e um modelo conceitual do negócio foi utilizado para dar origem aos casos de uso identificados. Foram montados protótipos de interfaces com o usuário.

Durante o projeto do ambiente, inicialmente foram realizados trabalhos para a definição da arquitetura do ambiente. O processo UML Components parte de uma arquitetura seguindo o estilo em camadas. O Eclipse e suas tecnologias impõem uma arquitetura seguindo o estilo MVC. Assim, a arquitetura resultante combina ambos os estilos. O modelo conceitual do ambiente Bellatrix inclui um modelo que representa conceitos de arquiteturas de software e DBC, um modelo baseado na IDL para representação de interfaces e um modelo para representar casos de testes. Seguindo o processo, as interfaces de sistema foram criadas a partir dos casos de uso e as interfaces de negócios foram criadas a partir dos tipos principais do modelo conceitual. Foram criados componentes de sistema e negócios para essas interfaces e esses foram foram conectados numa arquitetura de componentes. As interfaces foram detalhadas com a criação de diagramas de colaboração

entre seus serviços, refinando também a arquitetura de componentes.

Para a implementação dos diferentes elementos da arquitetura do ambiente, foram utilizadas diferentes estratégias. A materialização do núcleo do ambiente, contendo os componentes de sistema e negócios, utilizou o modelo de *plug-ins* do Eclipse em conjunto com o modelo COSMOS. A materialização da camada de persistência, contendo o modelo de tipos de negócio do ambiente, utilizou o EMF, *Eclipse Modeling Framework*. A camada de interface gráfica com o usuário foi implementada utilizando GEF, *Graphical Editing Framework*, concretizando o mecanismo de MVC da arquitetura. Para o gerenciador do processo, utilizou-se a ferramenta OpenFlow.

Foram realizados dois estudos de caso do ambiente. O primeiro, com foco no processo, utilizou um sistema fictício de reservas de quartos em hotéis e seguiu o processo UML Components, utilizando o Bellatrix e ferramentas UML. O segundo, com foco nas funcionalidades e usabilidade, foi realizado por voluntários e utilizou a especificação de um sistema financeiro real.

O ambiente Bellatrix encontra-se especificado e parcialmente implementado. Em particular, nem toda a interface gráfica com o usuário está completa. Ainda, falta a interface com o repositório de componentes, que está em desenvolvimento em outro trabalho em andamento [68]. O código desenvolvido está depositado no repositório CVS do grupo de pesquisas.

As principais dificuldades encontradas se relacionam ao projeto arquitetural do ambiente e sua implementação no Eclipse. Inicialmente, houve uma certa dificuldade para se definir o escopo do ambiente e seus requisitos, visto o desejo de se integrar diversas funcionalidades no mesmo ambiente. Com respeito ao projeto arquitetural, a arquitetura do ambiente está sujeita às restrições impostas pela tecnologia e às camadas indicadas pelo processo UML Components. Uma dificuldade estava na conciliação dessas restrições e o uso das camadas, recomendadas para sistemas de informação, num ambiente interativo. Ainda, outra dificuldade foi encontrada na implementação do ambiente no Eclipse e o estudo de suas tecnologias, em especial o desenvolvimento da interface gráfica com o usuário e dos editores gráficos, devido à sua curva de aprendizado.

8.1 Contribuições

Podemos citar como principais contribuições deste trabalho:

- A descrição geral do ambiente Bellatrix, registrado no artigo R. T. Tomita, F. Castor Filho, P. A. C. Guerra e C. M. F. Rubira. "Bellatrix: um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes". *In: Anais do 4o. Workshop de Desenvolvimento Baseado em Componentes*, 2004.

- A colaboração na criação de um modelo conceitual para o ambiente, 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.

- O levantamento de requisitos para o ambiente, incluindo casos de uso e protótipos de interfaces com o usuário, registrado em R. T. Tomita e C. M. F. Rubira. "Requisitos para um Ambiente de Desenvolvimento Baseado em Componentes e Centrado na Arquitetura de Software". Relatório Técnico IC-05-23, Instituto de Computação / UNICAMP, 2005.

- A especificação do ambiente, incluindo seu projeto arquitetural e especificação de componentes, registrado em R. T. Tomita e C. M. F. Rubira. "Projeto para um Ambiente de Desenvolvimento Baseado em Componentes e Centrado na Arquitetura de Software". Relatório Técnico, Instituto de Computação / UNICAMP, 2006 (a ser publicado).

- A implementação parcial do ambiente, incluindo a codificação do modelo e da interface gráfica com o usuário utilizando tecnologias Eclipse e as funcionalidades básicas do sistema. Todo código desenvolvido está depositado no repositório CVS do grupo de pesquisas e estará disponível publicamente. O conhecimento adquirido de tecnologias Eclipse está documentado em R. T. Tomita e C. M. F. Rubira. "Estudo de tecnologias para o desenvolvimento de *plug-ins* para o ambiente Eclipse". Relatório Técnico, Instituto de Computação / UNICAMP, 2006 (a ser publicado).

8.2 Trabalhos Futuros

Podem ser identificadas limitações e sugestões de alguns trabalhos de pesquisa e atividades como extensões e melhorias do trabalho apresentado nesta dissertação. Com relação às suas funcionalidades, a seguinte lista é apresentada:

- Durante a implementação do Gerenciador de Workflow do ambiente, mostrado na Figura 1.1, o processo UML Components foi modelado na ferramenta OpenFlow, mas seu uso é limitado. Para obter um melhor aproveitamento da ferramenta, a implementação de uma aplicação *web* para o servidor Zope utilizando a máquina de *workflow* seria necessária.

- É possível especificar as interfaces e suas operações em detalhes no ambiente Bellatrix. Entretanto, seu suporte atual a pré-condições e pós-condições escritas em

OCL é limitado. O espaço para a descrição dessas restrições poderia ser ampliado, bem como poderiam ser incluídos apoio à sintaxe e verificação das mesmas.

- Finalizar a implementação da interface gráfica com o usuário no Eclipse, que não pôde ser concluída devido à complexidade e ao tempo disponível. Esta atividade é parte de um trabalho de mestrado em andamento no Instituto [48].

- Em termos de implementação, o *plug-in* que armazena a configuração das ferramentas que compõem o Bellatrix poderia armazená-la utilizando uma ADL, por exemplo xAcme. Isso facilitaria a tarefa de criação dos arquivos que descrevem a configuração interna do ambiente.

- Poderiam ser elaboradas melhorias nas ferramentas para geração de código e incluídas ferramentas de engenharia reversa. Essas funcionalidades possibilitariam um apoio maior a outros modelos de componentes e a ADLs.

- Os testes dos componentes que formam o ambiente Bellatrix não foram feitos de forma automatizada, o que dificulta a realização dos testes, inclusive de testes de regressão. Poderia ser estudado e criado ferramental para os testes unitários de componentes dentro do Eclipse, ou eventualmente, testes unitários de *plug-ins* para Eclipse.

Com respeito à avaliação do ambiente, foram identificados os seguintes items:

- O estudo de caso do sistema de reservas de hotel, apresentado no Capítulo 6, foi realizado com base no processo UML Components. Uma avaliação utilizando outro processo de DBC e um sistema real, de maior complexidade e tamanho deve ser executado.

- No estudo de caso do sistema de reservas de hotel (Capítulo 6) utilizou-se duas ferramentas de modelagem UML, uma integrada ao Eclipse e outra não. Poderiam ser realizados outros estudos para avaliar outras ferramentas UML que sejam adequadas ao uso no processo UML Components, preferencialmente integradas ao Eclipse, como por exemplo a ferramenta SDE [70].

- O estudo de caso do sistema financeiro, apresentado no Capítulo 7, foi realizado por 4 alunos de pós-graduação deste Instituto. Um estudo de caso mais abrangente, com a participação de um número maior de pessoas de perfis diferentes poderia ser realizado.

Sobre a integração do ambiente Bellatrix com outras ferramentas disponíveis, listamos os seguintes pontos:

- Integrar a ferramenta de testes unitários de componentes CBDUnit [32] ao Eclipse. Inicialmente planejada para ser integrada ao Bellatrix, essa ferramenta foi desenvolvida fora do ambiente Eclipse, utilizando a tecnologia Swing. Ainda, seria necessário adequar o mecanismo de execução de testes da ferramenta às restrições impostas pelo Eclipse.

- Integrar um repositório de componentes ao ambiente, construindo a interface gráfica e implementando os componentes especificados para esse propósito. O repositório em si está sendo desenvolvido em um trabalho de mestrado em andamento no Instituto [68].

- Estudar a integração de outras ferramentas de DBC já existentes com o Bellatrix. Poderia ser estudada a integração com *plug-ins* de modelagem de engenharia de domínio (por exemplo, FeaturePlugin [2]) e de implantação específicos para um modelo de componentes (por exemplo, Lomboz [54]).

Apêndice A

Questionário de Avaliação do Ambiente Bellatrix

1. Ferramenta avaliada:
   ( ) Editor de exceções
   ( ) Editor de *datatypes*
   ( ) Editor de interfaces
   ( ) Editor de componentes
   ( ) Editor de arquiteturas
   ( ) Editor de configurações
   ( ) Exportar para COSMOS

2. Como você avalia seus conhecimentos sobre os conceitos de desenvolvimento baseado em componentes (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. Quantos artefatos (arquivos) foram criados ou editados na ferramenta avaliada?

4. 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?

5. 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

6. Você considera a ferramenta do BELLATRIX utilizada melhor, igual ou pior do que a ferramenta utilizada atualmente para desempenhar a mesma tarefa? Qual é essa ferramenta?

7. 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?)

8. Você tem sugestões de melhoria para as funcionalidades? Faltou alguma funcionalidade importante? Qual?

9. Ocorreram erros (*bugs*) no uso da ferramenta? Por favor, descreva-os para que possamos corrigí-los:

Bibliografia

[1] Miren Begoña Albizuri-Romero. A retrospective view of CASE tools adoption. *Software Engineering Notes (ACM SIGSOFT)*, 25(2):46 – 51, March 2000.

[2] Michal Antkiewicz and Krzysztof Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 64 – 72. ACM Press, 2004.

[3] ArgoUML. Argouml project home page. `http://argouml.tigris.org/`, acessado em 05/03/2006.

[4] C. Atkinson et al. *Component-Based Produt Line Engineering with UML*. Addison-Wesley, 2001.

[5] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute at Carnegie-Mellon University, April 2000.

[6] Stefan Van Baelen, David Urting, Werner Van Belle, Viviane Jonckers, Tom Holvoet, Yolande Berbers, and Karel De Vlaminck. Toward a unified terminology for component-based development. In *Proceedings of 14th European Conference on Object-Oriented Programming (ECOOP)*, June 2000.

[7] Robert Balzer and Volker Gruhn. Process-centered software engineering environments: Academic and industrial perspectives. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE2001)*, pages 671 – 673, 2001.

[8] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 2nd edition, 2003.

[9] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[10] Regina M. M. Braga. *Busca e Recuperação de Componentes em Ambientes de Reutilização de Software*. PhD thesis, COPPE - UFRJ, Dezembro 2000.

[11] 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.

[12] 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.

[13] Francois Bronsard, Douglas Bryan, W. (Voytek) Kozaczynski, Edy S. Liongosari, Jim Q. Ning, Asgeir Olafsson, and John W. Wetterstrand. Toward software plug-and-play. In *ACM Symposium on Software Reusability (SSR'97)*, pages 19 – 29. ACM Press, 1997.

[14] 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.

[15] John Cheesman and John Daniels. *UML Components*. Addison-Wesley, 2000.

[16] Milton Cesar Fraga de Souza. Um processo de desenvolvimento baseado em componentes adaptado ao model-driven architecture. Master's thesis, Instituto de Computação — Universidade Estadual de Campinas, 2004.

[17] André Van der Hoek. Capturing product line architectures. In *Proceedings of the 4th International Software Architecture Workshop*, Limerick, Ireland, June 2000.

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

[19] Eclipse Consortium. C/C++ development tools. `http://www.eclipse.org/cdt/`, acessado em 05/03/2006.

[20] Eclipse Consortium. COBOL IDE project. `http://www.eclipse.org/cobol`, acessado em 05/03/2006.

[21] Eclipse Consortium. Eclipse platform technical overview. `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`, acessado em 05/03/2006.

[22] Eclipse Consortium. Eclipse.org main page. `http://www.eclipse.org/`, acessado em 05/03/2006.

[23] Eclipse Consortium. EMF - eclipse modeling framework. `http://www.eclipse.org/emf`, acessado em 05/03/2006.

[24] Eclipse Consortium. GEF - graphical editing framework. `http://www.eclipse.org/gef`, acessado em 05/03/2006.

[25] Alfonso Fuggetta. A classification of case technology. *IEEE Computer*, 26(12):25 – 38, December 1993.

[26] Alfonso Fuggetta. Software process: A roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 25 – 34. ACM Press, 2000.

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

[28] 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.

[29] Gentleware AG. Poseidon for uml. `http://www.gentleware.com`, acessado em 05/03/2006.

[30] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[31] Paulo Asterio de Castro Guerra. *Uma Abordagem Arquitetural Para Tolerância a Falhas Em Sistemas de Software Baseados Em Componentes*. PhD thesis, Instituto de Computação — Universidade Estadual de Campinas, 2004.

[32] Paulo Asterio de Castro Guerra, Carolina Stephan de Araújo, Camila Ribeiro Rocha, and Eliane Martins. CBDUnit — uma ferramenta para testes unitários de componentes. In *Anais do Simpósio Brasileiro de Engenharia de Software (SBES2005), 12a. Sessão de Ferramentas*, 2005.

[33] Paulo Asterio de Castro Guerra, Tiago Moronte, Rodrigo T. Tomita, and Cecília M. F. Rubira. Um modelo conceitual e uma terminologia para o desenvolvimento baseado em componentes e centrado na arquitetura de software. In *Anais do 5o. Workshop de Desenvolvimento Baseado em Componentes (WDBC2005)*, pages 41 – 48, 2005.

[34] IBM Software. Rational Application Developer for WebSphere Software. `http://www-306.ibm.com/software/awdtools/developer/application/`, acessado em 05/03/2006.

[35] IBM Software. Rational Software Modeler. `http://www-306.ibm.com/software/awdtools/modeler/swmodeler/index.html`, acessado em 05/03/2006.

[36] Icube. Openflow: open-source workflow management system. `http://www.openflow.it`, acessado em 05/03/2006.

[37] Institute for Software Research at the University of California at Irvine. ArchStudio: An architecture-based development environment. `http://www.isr.uci.edu/projects/archstudio/`, acessado em 05/03/2006.

[38] Philippe Kruchten. *The Rational Unified Process — An Introduction*. Addison-Wesley, 1999.

[39] P. A. Lee and T. Anderson. *Fault Tolerance - Principles and Practice*, volume Volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag/Wien, second edition, 1990.

[40] D. Lucrédio, E. S. Almeida, C. P. Bianchini, A. F. Prado, and L. C. Trevelin. Orion - a component-based software engineering environment. *Journal of Object Technology*, 3(4):51 – 74, April 2004. Special issue: TOOLS USA 2003.

[41] 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.

[42] Chris Lüer and David S. Rosenblum. WREN - an environment for component-based development. In *Proceedings of Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 207 – 217. ACM Press, 2001.

[43] Chris Lüer, David S. Rosenblum, and André van der Hoek. The evolution of software evolvability. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 134 – 137. ACM Press, 2001.

[44] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform*. Addison-Wesley, 2005.

[45] Microsoft Corporation. Microsoft .net information. `http://www.microsoft.com/net/`, acessado em 05/03/2006.

[46] Microsoft Corporation. Microsoft visual studio developer center. `http://msdn.microsoft.com/vstudio/`, acessado em 05/03/2006.

[47] Nelson Miler. A engenharia de aplicações no contexto da reutilização baseada em modelos de domínio. Master's thesis, COPPE - UFRJ, Julho 2000.

[48] Tiago C. Moronte. Um ambiente para descrição de arquiteturas de software. Proposta de dissertação de mestrado, Instituto de Computação — UNICAMP, Setembro 2004.

[49] Jim Q. Ning. ADE - an architecture design environment for component-based software engineering. In *Proceedings of International Conference on Software Engineering (ICSE'97)*, pages 614 – 615. ACM Press, 1997.

[50] Jim Q. Ning. CBSE research at andersen consulting. In *Proceedings of the 1st International Workshop on Component-Based Software Engineering*, April 1998.

[51] Object Management Group. CORBA Component Model. `http://www.omg.org/technology/documents/formal/components.htm`, acessado em 05/03/2006.

[52] Object Management Group. CORBA Specification. `http://www.omg.org/technology/documents/formal/corba_2.htm`, acessado em 05/03/2006.

[53] Object Management Group. MOF 2.0 / XMI Mapping Specification, v2.1. `http://www.omg.org/technology/documents/formal/xmi.htm`, acessado em 05/03/2006.

[54] ObjectLearn. Lomboz project. `http://www.objectlearn.com/`, acessado em 05/03/2006.

[55] Omondo. EclipseUML. `http://www.omondo.com`, acessado em 05/03/2006.

[56] 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 IC-05-01, Instituto de Computação / UNICAMP, 2005.

[57] Camila Ribeiro Rocha. Um método de testes para componentes tolerantes a falhas. Master's thesis, Instituto de Computação — Universidade Estadual de Campinas, 2005.

[58] Heloísa Vieira da Rocha and Maria Cecília C. Baranauskas. *Design e Avaliação de Interfaces Humano-Computador*. NIED / UNICAMP, 2003.

[59] 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.

[60] Geri Schneider and Jason P. Winters. *Applying Use Cases: A Practical Guide*. Addison-Wesley, 2 edition, 2001.

[61] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and implementations for architectural connections. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, page 2. IEEE Computer Society, 1996.

[62] 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.

[63] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[64] Sun Microsystems. Java 2 platform enterprise edition. `http://java.sun.com/j2ee/`, acessado em 05/03/2006.

[65] Sun Microsystems. Java Remote Method Invocation (Java RMI). `http://java.sun.com/products/jdk/rmi/`, acessado em 05/03/2006.

[66] Sun Microsystems. Javabeans. `http://java.sun.com/products/javabeans`, acessado em 05/03/2006.

[67] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2nd edition, 2002.

[68] Leonardo P. Tizzei, Helder S. Pinho, Paulo A. C. Guerra, and Cecília M. F. Rubira. Um repositório de componentes com suporte à evolução centrada na arquitetura de software. In *Anais do 5o. Workshop de Desenvolvimento Baseado em Componentes (WDBC2005)*, pages 1 – 8, 2005.

[69] 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.

[70] Visual Paradigm. Smart development environment for eclipse. `http://www.visual-paradigm.com/product/sde/`, acessado em 05/03/2006.

[71] W3C Consortium. Extensible markup language (xml) 1.0 (third edition). `http://www.w3.org/TR/REC-xml/`, acessado em 05/03/2006.

[72] Cláudia Maria Lima Werner and Regina Maria Maciel Braga. A engenharia de domínio e o desenvolvimento baseado em componentes. In Itana Maria de Souza Gimenes and Elisa Hatsue Moriya Huzita, editors, *Desenvolvimento Baseado em Componentes*, chapter 3, pages 57 – 104. Editora Ciência Moderna, 2005.

[73] Zope Corporation. Zope community. `http://www.zope.org`, acessado em 05/03/2006.