Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Uma Infra-Estrutura para o Desenvolvimento de Aplicações Corporativas com Suporte à Evolução Dinâmica e Não Antecipada

Marcos Fábio Pereira

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Leandro Dias da Silva e Angelo Perkusich

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

P436i
2009 Pereira, Marcos Fábio.
Uma infra-estrutura para o desenvolvimento de aplicações corporativas com suporte à evolução dinâmica e não antecipada / Marcos Fábio Pereira. ─ Campina Grande, 2009.
61 f. : il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientadores: Prof. Dr. Leandro Dias da Silva, Prof. Dr. Angelo Perkusich.

1. Engenharia de Software. 2. Aplicações Corporativas.. 3. Evolução de Software. I. Título.

CDU – 004.41(043)

"UMA INFRA-ESTRUTURA PARA O DESENVOLVIMENTO DE APLICAÇÕES CORPORATIVAS COM SUPORTE À EVOLUÇÃO DINÂMICA E NÃO ANTECIPADA"

MARCOS FÁBIO PEREIRA

DISSERTAÇÃO APROVADA EM 28.09.2009

LEANDRO DIAS DA SILVA, D.Sc
Orientador(a)

ANGELO PERKUSICH, D.Sc
Orientador(a)

HYGGO OLIVEIRA DE ALMEIDA, D.Sc
Examinador(a)

MARIA DE FÁTIMA QUEIROZ VIEIRA , Ph.D
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Aplicações corporativas têm como principal finalidade auxiliar nas atividades dos diversos setores de uma corporação. Atualmente existe uma grande necessidade por este tipo de aplicação e este número tende a aumentar com o surgimento de novas corporações, além do crescimento das já existentes. Do ponto de vista da Engenharia de Software, uma característica importante destas aplicações é o conjunto comum de requisitos não funcionais que apresentam. Aplicações corporativas devem prover, em geral: distribuição, facilitar a escalabilidade do software; balanceamento de carga e tolerância a falhas, para garantir robustez e alta disponibilidade; segurança, para garantir a proteção dos dados da corporação; serviços transacionais, para garantir a consistência dos dados e nas operações sobre eles; dentre outras funcionalidades. Além destes requisitos, tais aplicações precisam lidar com mudanças constantes nas regras de negócio das corporações. Dada a complexidade das aplicações, tais alterações, em geral, não podem ser previstas em tempo de projeto e normalmente afetam pontos do software que não foram preparados para mudanças. Além disto, durante esta alteração, muitas vezes a aplicação corporativa precisa ser mantida em execução para evitar perdas para a corporação. Sendo assim, tem-se como requisito primordial a possibilidade de evolução nas aplicações de forma dinâmica e não antecipada. Neste trabalho apresenta-se uma infra-estrutura para o desenvolvimento de aplicações corporativas que oferece o suporte à evolução dinâmica e não antecipada. Esta infra-estrutura é uma extensão de um modelo de componentes que oferece suporte nativo à evolução dinâmica e não antecipada, tornando a tarefa de evolução mais eficaz que em soluções já existentes. A validação do trabalho foi realizada através do desenvolvimento de aplicações corporativas a partir da infra-estrutura proposta.

Abstract

Enterprise applications are primarily used to support in the activities of various sectors of a corporation. Currently there is a great need for this type of application with the emergence of new corporations, as well as the expansion of the existing ones. From the perspective of Software Engineering, an important feature of these applications is their common set of non-functional requirements. Enterprise applications must provide, in general: distribution, seeking scalability of the software; load balancing and fault tolerance to ensure robustness and high availability; security measures to ensure the protection of corporate data; transactional services, to ensure consistency in data and operations on them; among other features. In addition to these requirements, such applications must deal with constant changes in the business rules of corporations. Given the complexity of applications, such changes generally can not be predicted at design time and usually affect parts of the software that were not prepared for changes. Moreover, during this change, often the enterprise application needs to be kept running to avoid losses to the corporation. Thus, it is as essential requirement the possibility of dynamic and unanticipated evolution in these applications. This work presents an infrastructure for the development of enterprise applications that provides support for dynamic and unanticipated evolution. This infrastructure is an extension of a component model that has a native support for the dynamic and unanticipated evolution, making the development task more effective than existing solutions. The validation of the work was done through the development of enterprise applications using the proposed infrastructure.

Agradecimentos

Agradeço primeiramente a minha mãe e ao meu irmão, por terem sempre acreditado na minha capacidade e pelo apoio que me deram. Agradeço a todos os amigos que ficaram em Maceió, em especial aos meus "irmãos" Juliana Ribeiro e Luiz Josué. Eles foram pessoas especiais que ajudaram muito nos momentos mais difíceis destes últimos dois anos.

Agradeço a todos os novos amigos que fiz em Campina Grande: Diego Bezerra, Glauber Vinícius (Galubê), Hyggo Almeida, Olympio Cipriano (Mago Bronson), Felipe Pontes (Vigia), André Felipe, Raul Herbster, Walter Guerra, Mateus Máximo, Ádrian Lívio (Gordinho), Danilo Santos (Senhor Burns), José Luís, Thiago Santos, Lorena Maia, Taísa Felix, Taciana Rached, Mário Hozano, Leandro Sales, Ivo Augusto, Thiago Sales, entre outros mais. Graças a eles que este momento merece ser bem lembrado.

Agradeço aos orientadores Leandro Dias e Angelo Perkusich, pelo acompanhamento do trabalho e pela paciência com meus atrasos. Sem eles esta etapa da minha vida não seria concluída.

Conteúdo

Lista de Figuras

Lista de Códigos Fonte

Capítulo 1

Introdução

Aplicações corporativas são tipos de aplicações relacionadas à visualização, manipulação e armazenamento de uma grande quantidade de dados, além da automação dos processos de negócio que manipulam estes dados. Alguns exemplos deste tipo de aplicação incluem sistemas de reservas de passagens, sistemas financeiros, sistemas logísticos e qualquer outro sistema responsável por automatizar os processos de negócio de uma corporação [20].

Grandes empresas possuem uma demanda especial por este tipo de aplicação, dada a grande quantidade de informação da qual dependem para a execução de seus negócios. A confiança na consistência destas informações e a demanda por acesso eficaz são fatores críticos para corporações que desejam ser bem sucedidas em suas áreas de atuação. Por este motivo, estas aplicações possuem um papel fundamental dentro das empresas.

Dada a importância e uso extenso de aplicações corporativas, cada vez mais busca-se uma redução nos custos de desenvolvimento, manutenção e evolução dessas aplicações. Além disto, busca-se uma redução do tempo de resposta destas aplicações, aumentando a velocidade com que a informação chega ao usuário da aplicação. Atualmente, os recursos financeiros destinados a software em grandes corporações são mais demandados pela manutenção e evolução de existentes do que pelo desenvolvimento de novas aplicações [40].

No ciclo de vida de um software, a evolução é a fase mais crítica, com relação a custo e tempo. A evolução do software é necessária nas seguintes situações: reparar problemas de funcionamento; adicionar novas funcionalidades; melhorar funcionalidades existentes (desempenho, extensibilidade, etc); e adaptar-se a um novo ambiente de execução ou plataforma. A evolução de software pode ser definida como o conjunto de atividades que visam

garantir que o software acompanhe a evolução dos objetivos da organização para o qual foi desenvolvido, de uma maneira viável em termos de custo [4].

Algumas características podem tornar a tarefa de evolução de software mais complexa e difícil de ser gerenciada: *Evolução Dinâmica*, que consiste na evolução em tempo de execução, o que é comum em sistemas críticos, nos quais o tempo de interrupção pode gerar custos inaceitáveis; e *Evolução Não Antecipada*, consiste na evolução não prevista em tempo de projeto, ou seja, quando não se utiliza de pontos de extensão previamente definidos. Quando estes dois requisitos ocorrem em conjunto, tem-se a Evolução Dinâmica de Software Não Antecipada (EDSNA) [4].

Em aplicações corporativas as regras de negócio mudam com frequência e tais mudanças não são previstas em tempo de projeto. Além disto, a execução constante da aplicação pode ser um fator essencial. Em aplicações corporativas com estes requisitos é necessária a evolução dinâmica e não antecipada. O escopo deste trabalho consiste na elaboração e desenvolvimento de uma infra-estrutura que ofereça suporte ao desenvolvimento de aplicações corporativas com suporte à evolução dinâmica e não antecipada.

1.1 Descrição do Problema

Algumas das infra-estruturas existentes para o desenvolvimento de aplicações corporativas como *Enterprise Java Bean* EJB [41] e .NET [29] estão entre as que são consideradas estáveis e são frequentemente utilizadas. Apesar de sua popularidade, estas infra-estruturas não possuem suporte à evolução dinâmica de software e não antecipada.

A evolução de software é particularmente importante para as corporações por dois motivos: consome uma grande parte do investimento destinado ao software; e a falha ou atraso na realização da evolução do software implica em oportunidades de negócio perdidas para a corporação [10]. Além desta importância para aplicações corporativas, existe o problema fundamental da evolução de software, que é o fato dos pontos de extensão de uma aplicação não serem completamente antecipados em tempo de projeto [10].

Enquanto infra-estruturas com suporte ao desenvolvimento de aplicações corporativas não oferecem o suporte à evolução dinâmica e não antecipada, existem modelos de componentes com suporte a tal evolução, sendo alguns dos exemplos apresentados neste trabalho

os modelos de componentes COMPOR [5] e Balboa [17; 18]. Porém, estes modelos não oferecem o suporte ao desenvolvimento de aplicações corporativas.

As possíveis soluções para solucionar este problema são: adicionar o suporte à evolução dinâmica e não antecipada em uma infra-estrutura que ofereça o suporte ao desenvolvimento de aplicações corporativas; ou adicionar o suporte ao desenvolvimento de aplicações corporativas em um modelo de componentes com suporte à evolução dinâmica e não antecipada.

Adicionar o suporte à evolução dinâmica e não antecipada em uma infra-estrutura com suporte ao desenvolvimento de aplicações corporativas é a solução mais complexa, uma vez que as infra-estruturas existentes não foram concebidas para este tipo de evolução e a mudança estrutural seria demasiadamente grande. Por este motivo foi escolhida a opção de adicionar o suporte ao desenvolvimento de aplicações corporativas em um modelo de componentes com suporte à evolução dinâmica e não antecipada, pois estes modelos apresentam arquiteturas simples e de fácil entendimento.

1.2 Objetivo

O principal objetivo deste trabalho é a concepção e construção de uma infra-estrutura que forneça o suporte ao desenvolvimento de aplicações corporativas. Tal infra-estrutura, além de prover mecanismos para atender aos requisitos de aplicações corporativas, oferece suporte à evolução dinâmica e não antecipada.

A infra-estrutura proposta foi concebida como uma extensão da especificação do modelo de componentes COMPOR [5], acoplando mecanismos inerentes à natureza de aplicações corporativas, como distribuição, transações e persistência de dados. Para execução das aplicações, foi desenvolvido um ambiente de execução, baseado na arquitetura do ambiente de execução do COMPOR [5].

Para validação do trabalho, foram desenvolvidas duas aplicações em diferentes domínios, para avaliar diferentes pontos da infra-estrutura proposta. Estas aplicações executam sobre o ambiente de execução desenvolvido, permitindo abordar todos os aspectos do trabalho.

1.3 Relevância

Atualmente as soluções de infra-estrutura para o desenvolvimento de aplicações corporativas não oferecem suporte à evolução não antecipada, sendo necessário um refatoramento extenso na aplicação para evoluções deste tipo. Este problema ocorre apesar da imprevisibilidade natural em aplicações corporativas. Além disto, os modelos de componentes que oferecem suporte à evolução não antecipada não oferecem suporte ao desenvolvimento de aplicações corporativas.

Outro ponto relevante é a simplicidade da especificação. Como será descrito ao longo deste documento, a infra-estrutura foi especificada baseada em componentes, ou seja, cada módulo da infra-estrutura pode ser utilizado sob demanda, evitando torná-la complexa do ponto de vista do desenvolvedor. É possível desenvolver uma aplicação apenas com suporte à distribuição, ou que contemple distribuição e persistência. Desta forma, pode-se utilizar o mesmo modelo, para diferentes aplicações, mantendo a simplicidade de sua arquitetura.

Por fim, este trabalho dá continuidade a um projeto multi-institucional, iniciado na Universidade Federal de Campina Grande - o projeto COMPOR (www.compor.net). Espera-se que a infra-estrutura proposta possa servir de base para novos trabalhos na área de evolução dinâmica e não antecipada, bem como para o desenvolvimento de aplicações corporativas com suporte a esta característica.

1.4 Organização do Documento

Este documento está organizado da seguinte forma, no Capítulo 2 são apresentados conceitos considerados fundamentais para o entendimendo do leitor sobre a infra-estrutura proposta. No Capítulo 3 são apresentados alguns trabalhos relacionados à solução proposta, sendo discutido ao final deste capítulo os pontos negativos e positivos da solução proposta com relação a estes trabalhos pesquisados. O capítulo 4, apresenta em detalhes, o modelo de componentes COMPOR. Este modelo foi utilizado como base para o desenvolvimento da infra-estrutura proposta neste trabalho. O conhecimento deste modelo é essencial para o entendimento da infra-estrutura apresentada. Nos Capítulos 5 e 6 é apresentado, em detalhes, o resultado deste trabalho. O Capítulo 5 trata sobre a extensão para aplicações corporati-

vas, especificada sobre o modelo de componentes COMPOR. Enquanto que no Capítulo 6 é apresentado o ambiente de execução implementado, responsável pelo gerenciamento do ciclo de vida das aplicações desenvolvidas, usando a extensão apresentada no Capítulo 5. No Capítulo 7 o trabalho é validado através da apresentação de dois estudos de caso onde a infra-estrutura desenvolvida foi utilizada. O Capítulo 8 apresenta as considerações finais sobre a infra-estrutura desenvolvida e propõe trabalhos futuros.

Capítulo 2

Fundamentação

Neste capítulo são apresentados os conceitos de Aplicações Corporativas, Desenvolvimento Baseado em Componentes e Evolução Dinâmica de Software Não Antecipada. Estes conceitos são a base do trabalho descrito neste documento e o entendimento deles é considerado essencial para a compreensão da infra-estrutura proposta.

2.1 Aplicações Corporativas

Aplicações corporativas são aplicações que tratam da visualização, manipulação e armazenamento de uma grande quantidade de dados e dos processos de negócio que interagem com esta informação [20]. Alguns exemplos de domínios de atuação de aplicações corporativas são: sistemas de reservas, sistemas financeiros, sistemas logísticos, sistemas de comércio eletrônico, entre outros que operam em corporações. Aplicações corporativas possuem seus próprios desafios e soluções, e estes são diferentes dos sistemas embarcados, sistemas de telecomunicação, ou aplicações *desktop* [20].

As aplicações corporativas, também conhecidas como sistemas de informação, precisam ser: seguras, para proteger usuários e a corporação; escaláveis, para garantir que os usuários simultaneamente tenham vantagem de diferentes serviços; e confiáveis, para garantir a consistência dos processos transacionais. Alguns exemplos de aplicações corporativas são apresentados abaixo:

- **Alfresco Community** - É uma aplicação corporativa Open Source que atua na gestão de imagens, documentos, informações corporativas e conteúdo web [1].

- **Compiere** - Consiste em um ERP (Enterprise Resource Planning) para propósitos gerais, com algumas funções de CRM (Customer Relationship Management). Possui controle da parte financeira, recursos humanos, folha de pagamento, gestão de inventário, assim como vendas e relatórios [14].

- **Liferay Social Office** - É uma solução colaborativa social para a corporação. Consiste em uma área de trabalho virtual voltada à comunicação e construção de grupos coesos, que visa poupar tempo em encontrar informações desejadas no ambiente corporativo [26].

A lista de aplicações corporativas existentes é extensa, sendo praticamente impossível listar todas neste trabalho. Para cada domínio existente, dezenas de aplicações corporativas foram desenvolvidas. Apesar de existirem aplicações corporativas para diversos domínios, um conjunto comum de características foi identificado entre todas, sendo que este conjunto determinou um modelo de desenvolvimento para aplicações corporativas independente do domínio. Este modelo foi a base para o desenvolvimento de *middlewares* para o auxílio no desenvolvimento de aplicações corporativas.

Um *middleware* voltado ao desenvolvimento de aplicações corporativas deve se encarregar das operações básicas que são realizadas por qualquer aplicação corporativa, independente do domínio da aplicação. O uso de um *middleware* permite ao desenvolvedor se concentrar na lógica de negócio do domínio da aplicação, sendo as tarefas básicas como segurança, gerenciamento de ciclo de vida, transação, persistência entre outras, responsabilidade do *middleware*. Entre os *middlewares* mais conhecidos merecem destaque o JBoss (Red Hat) [22], WebSphere (IBM) [47], Geronimo (Apache) [9] e WebLogic (BEA) [35].

2.2 Desenvolvimento Baseado em Componentes

O principal objetivo do desenvolvimento baseado em componentes é tornar o desenvolvimento de software um processo de junção de partes menores, conhecidas como componentes [15]. Segundo Szyperski [45], um componente de software é uma unidade de composição com *interfaces* especificadas de forma *contratual*, podendo ser desenvolvida de forma independente e sujeita à composição por terceiros.

Para um melhor entendimento da definição de componentes apresentada, dois outros conceitos são necessários: interface e contrato. A interface de um componente pode ser definida como a especificação do seu ponto de acesso. Um componente é uma unidade com conteúdo encapsulado por trás de uma interface. A interface provê uma separação explícita entre o lado interno e externo de um componente, definindo o que ele implementa mas escondendo como ele é implementado [4].

Um contrato representa a descrição de interface, ou seja, a especificação do comportamento de um componente. Um contrato lista as restrições que o componente deverá manter (invariante). Para cada operação do componente, o contrato também lista as restrições que precisam ser satisfeitas pelo cliente (pré-condições) e aquelas que o componente promete estabelecer como retorno (pós-condições). A pré-condição, o invariante e a pós-condição constituem a especificação do comportamento de um componente [4].

No contexto deste trabalho, serão considerados modelos de componentes que oferecem suporte ao desenvolvimento de aplicações corporativas. Os principais modelos que oferecem este suporte e, mais utilizados atualmente são: EJB [43] (Enterprise Java Beans), consiste na especificação de um modelo de componentes da Sun Microsystems [33] que oferece suporte à segurança, transação, persistência, distribuição, etc; .NET [28], uma iniciativa da Microsoft [32] que é uma plataforma única para desenvolvimento e execução de sistemas e aplicações.; OSGi [2], uma especificação para programacão orientada a serviços utilizando a linguagem Java; entre outros modelos. Estes modelos são apresentados em detalhes no Capítulo 3.

2.3 Evolução de Software Não Antecipada

Evolução é a fase mais crítica, com relação a custo e tempo, do ciclo de vida de um software. A evolução de software é necessária nas seguintes situações: reparar problemas de funcionamento; adicionar novas funcionalidades; melhorar funcionalidades existentes (desempenho, extensibilidade, etc); adaptar-se a um novo ambiente de execução ou plataforma; e prevenir potenciais problemas que possam vir a ocorrer no software.

Define-se como evolução de software, o conjunto de atividades que visam garantir que o software continue a atingir seus objetivos de negócio, e da organização para o qual foi

desenvolvido, de uma maneira viável em termos de custo [4]. Dois tipos de evolução tornam esta fase mais complexa e difícil de ser gerenciada, eles serão descritos nas seções a seguir.

2.3.1 Evolução Dinâmica de Software

As atividades de evolução de software podem ocorrer em diferentes fases do seu ciclo de desenvolvimento, as quais podem ser classificadas em três categorias principais: em tempo de compilação; em tempo de carregamento; e em tempo de execução. Estas três categorias de evolução, descritas a seguir, estão relacionadas com o momento em que ocorrem em um sistema de software [30].

- **Tempo de compilação:** A mudança do software está relacionada ao código fonte do sistema. Conseqüentemente, o software necessita ser recompilado para que a mudança se torne disponível. Utiliza-se também o termo evolução estática para designar este tipo de evolução.
- **Tempo de carregamento:** A mudança ocorre quando os elementos do sistema são carregados dentro de um sistema executável. O sistema não necessita ser recompilado mas precisa ser reiniciado para que a mudança se torne disponível.
- **Tempo de execução:** A mudança ocorre durante a execução do sistema.

A partir da classificação das três categorias de evolução, tem-se que a evolução dinâmica é um tipo de evolução que ocorre sem a interrupção da execução do software.

2.3.2 Evolução Não Antecipada

Durante o projeto de um software é possível identificar possíveis cenários de evolução e elaborar o projeto com pontos de extensão. Contudo, existem cenários de evolução que não podem ser identificados em tempo de projeto. Além disso, quanto mais cenários de evolução são previstos e considerados no projeto, mais complexa se torna a solução inicial, levando a um projeto mais flexível do que demandam os seus requisitos e aumentando o custo e o tempo de desenvolvimento [25].

Esse tipo de evolução relacionada a mudanças para as quais o software não foi preparado durante o projeto e implementação inicial do software é denominada Evolução Não Anteci-

pada. Sem suporte à evolução não antecipada, mudanças não previstas geralmente forçam desenvolvedores a realizar modificações no código e no *design* do projeto.

Quando os requisitos de evolução dinâmica e não antecipada ocorrem em conjunto, tem-se a Evolução Dinâmica de Software Não Antecipada (EDSNA) [4]. Existem diversos trabalhos com suporte à EDSNA, alguns deles são apresentados no Capítulo 3. Porém o modelo utilizado como base para a infra-estrutura proposta foi o *COMPOR Component Model Specification* (COMPOR-CMS) [6].

Capítulo 3

Trabalhos Relacionados

Nesta seção são apresentados alguns trabalhos relacionados, considerados relevantes para comparação com o trabalho proposto. São modelos de componentes com algum suporte, ou nenhum, ao desenvolvimento de aplicações corporativas.

3.1 Balboa

Balboa [17; 18] é um ambiente de composição de software, inicialmente projetado para componentes de hardware, com suporte à evolução dinâmica não antecipada de aplicações desenvolvidas em C++. É composto de três partes principais: uma linguagem de integração de componentes; um conjunto de bibliotecas de componentes C++; e um conjunto de interfaces que estabelecem a ligação entre as duas primeiras partes.

O suporte à EDSNA ocorre através da manipulação da linguagem de *script* que integra os componentes, podendo ser editado em tempo de execução e refletido dinamicamente na aplicação. Não existe um suporte a aplicações corporativas neste modelo, sendo necessário ao desenvolvedor da aplicação especificar e implementar as características de infra-estrutura caso seja necessário.

3.2 Beanome

Beanome [12; 13] adiciona uma camada sobre a arquitetura padrão de OSGi [2] para prover funcionalidades similiares àquelas encontradas em modelos de componentes, de forma que

aplicações possam ser construídas com base na montagem de instâncias de componentes.

O Beanome provê suporte ao gerenciamento da execução de aplicações através de um *middleware* composto por dois principais módulos: fábricas de componentes e registro de componentes. O registro mantém a lista de todas as fábricas disponíveis. As fábricas estão associadas aos tipos de componentes para criar instâncias dos mesmos. Este arcabouço é implementado como um *bundle* OSGi [2].

Com relação ao desenvolvimento de aplicações corporativas, este modelo de componentes apresenta suporte à distribuição e segurança já contemplados pelo padrão OSGi, outras características não são contempladas pelo modelo.

3.3 Mobile Application Server (MAS)

MAS é um middleware para o desenvolvimento de aplicações wireless Java. O principal problema no desenvolvimento de aplicações móveis abordado é a criação de novos serviços de forma rápida e eficiente. MAS provê um conjunto de componentes que oferecem funcionalidades consideradas comuns no desenvolvimento de aplicações móveis:

- Gerenciamento de perfil de usuário;
- Manipulação de dados sobre localização;
- Detecção de dispositivo;
- Adaptação de conteúdo;
- Funções e-wallet;
- API para criação de aplicações de terceiros;
- Abstração sobre detalhes da infra-estrutura do dispositivo por parte dos desenvolvedores de aplicações móveis;

Por causa da variedade de dispositivos usados pelas corporações, os componentes que oferecem as funcionalidades de infra-estrutura possuem diferenças entre implementações para dispositivos distintos. Para resolver este problema o MAS oferece uma instalação sob

demanda desses componentes, de acordo com o dispositivo onde a aplicação está sendo executada. Com estas características o MAS pode ser utilizado para o desenvolvimento de aplicações corporativas móveis.

Após identificar um conjunto de requisitos necessários no desenvolvimento de aplicações móveis, foi desenvolvida a arquitetura e a plataforma MAS que atende a cinco requisitos: modular, flexível, evolucionário, independente de conteúdo e aberto.

3.4 Open Services Gateway Initiative (OSGi)

OSGi é uma especificação para a programação orientada a serviços utilizando a linguagem Java [2]. Esta especificação provê um ambiente orientado a componentes para serviços distribuídos. Ela foi definida e é mantida pela *OSGi Alliance*, uma corporação independente e sem fins lucrativos com foco na produção de tecnologia relacionada à interoperabilidade de aplicações e serviços baseada em OSGi como plataforma de integração de componentes.

Em OSGi existe suporte à evolução dinâmica porém não existe suporte à evolução não antecipada, pois existe referência explícita entre componentes a nível de codificação.

Com relação a aplicações corporativas, a abordagem do OSGi provê suporte à segurança e distribuição, sendo distribuição a maior motivação para criação dessa especificação.

3.5 Software Engineering for Embedded Systems using a Component Oriented Approach (SEESCOA)

SEESCOA [46] é um projeto desenvolvido por uma universidade da Bélgica que tem como objetivo estudar a aplicação da abordagem de componentes para o desenvolvimento de sistemas embarcados com suporte à EDSNA. Um dos pontos fortes dessa abordagem é a definição formal do modelo aliada a uma ferramenta de apoio ao desenvolvimento e à execução do sistema desenvolvido.

A comunicação entre componentes ocorre via troca de mensagens assíncronas. Todo componente possui um conjunto de portas para enviar essas mensagens. Cada porta possui seu próprio protocolo de comunicação, descrito em um arquivo XML que descreve a porta.

O ambiente de execução dos componentes é em Java e foi implantado em dispositivos embarcados executando sobre a máquina virtual Kaffe (KVM) [23]. A implementação mostrou-se eficiente na execução de dispositivos embarcados, mas não há nenhum suporte definido com relação ao desenvolvimento de aplicações corporativas.

3.6 Enterprise JavaBeans (EJB)

Enterprise JavaBean (EJB) [42] é a especificação de um modelo de componentes da Sun Microsystems. Aplicações que são desenvolvidas seguindo este modelo podem ser implantadas em servidores de aplicação que seguem a especificação J2EE, também da Sun Microsystems.

O modelo de componentes Enterprise JavaBean não define um modelo de evolução não antecipada ou dinâmica, contudo, a especificação J2EE para servidores de aplicação define a evolução dinâmica como um de seus requisitos básicos. Consequentemente, como aplicações desenvolvidas usando o modelo de componentes EJB são destinadas à execução sobre servidores de aplicação J2EE, considera-se que o modelo EJB oferece o suporte à evolução dinâmica de software.

O modelo EJB foi originalmente elaborado para oferecer suporte ao desenvolvimento de aplicações corporativas. Logo, em sua lista de características constam a maioria dos serviços requeridos em aplicações deste tipo: distribuição, segurança, transações, persistência, etc.

O modelo EJB ou a especificação J2EE para servidores de aplicação não oferecem suporte nativo à evolução não antecipada de software. Alguns trabalhos [38] foram realizados, propondo uma modificação no modelo EJB para a inclusão do suporte a este tipo de evolução. Porém, esta modificação consiste em uma alteração nos *Class Loaders* da linguagem Java, sendo uma tarefa difícil e que adiciona uma carga de processamento indesejada à execução do modelo.

3.7 .NET Framework

A tecnologia .NET [31] é uma iniciativa da empresa Microsoft que visa uma plataforma única para o desenvolvimento e execução de aplicações. Uma aplicação que executa sobre a plataforma .NET pode ser desenvolvida em qualquer linguagem que possua um compilador

para a *Common Language Runtime* (CLR). O CLR é o ambiente responsável por todos os aspectos relacionados à execução da aplicação [28].

A plataforma .NET, com o auxílio da CLR, oferece ao desenvolvedor os princípios básicos do desenvolvimento orientado a componentes, como compatibilidade entre cliente e componente, separação da interface e implementação, transparência sobre a localização do objeto, gerenciamento de concorrência, gerenciamento de memória, segurança e independência de linguagem [28].

Com idéia semelhante a da plataforma Java, o programador deixa de escrever código para um sistema ou dispositivo específico e passa a escrever para a plataforma .NET. Porém, o código desenvolvido não necessita ser escrito na linguagem C#, a linguagem nativa da plataforma .NET, mas pode ser escrito em qualquer linguagem, desde que exista um compilador desta linguagem para a CLR.

A plataforma .NET oferece suporte à evolução dinâmica através do versionamento de componentes. Com o controle de versionamento, componentes iguais de versões diferentes podem coexistir no mesmo ambiente de execução sem impacto algum na aplicação como um todo. Um componente de versão mais nova pode ser implantado sem a necessidade de parar a execução da aplicação, já que os componentes continuam referenciando a versão anterior do componente. Contudo, a plataforma .NET não oferece suporte à evolução não antecipada, pois assim como no modelo EJB, existe um forte acoplamento entre os componentes da plataforma.

3.8 Discussão sobre os trabalhos relacionados

Para o desenvolvimento de aplicações corporativas, EJB [42] e .NET [28] são as plataformas mais utilizadas. Desenvolvedores usando uma dessas plataformas economizam tempo de desenvolvimento por não se preocuparem com codificação de diversos serviços que já são disponibilizados pela plataforma usada. Estes serviços incluem: segurança, persistência, distribuição, balanceamento de carga, gerenciamento de transação, controle de cache entre outros. Contudo, EJB e .NET não oferecem suporte adequado para evolução não antecipada de software. Isto ocorre principalmente por causa do alto acoplamento existente entre componentes destas plataformas.

A proposta deste trabalho é desenvolver uma infra-estrutura para o desenvolvimento de aplicações corporativas com suporte à evolução dinâmica e não antecipada. A diferença entre a infra-estrutura apresentada neste trabalho e as soluções apresentadas neste capítulo é que as soluções apresentadas oferecem somente uma das duas características nativamante, a evolução dinâmica e não antecipada ou o suporte ao desenvolvimento de aplicações corporativas.

Como apresentado no Capítulo 1, a evolução dinâmica e não antecipada é essencial em alguns tipos de software, especialmente em aplicações corporativas onde as regras de negócio mudam com frequência e a antecipação dessas mudanças em tempo de projeto nem sempre é possível.

O ponto fraco da infra-estrutura proposta neste trabalho em relação às soluções mais utilizadas atualmente está no desempenho de execução de aplicações. Solucões estáveis como EJB e .NET recebem refatoramento constante em seu código, o que permite um ganho de desempenho quando comparadas com versões anteriores da mesma infra-estrutura.

Capítulo 4

Modelo de Componentes Compor

Como apresentado anteriormente, a evolução dinâmica e não antecipada de software é essencialmente necessária em aplicações corporativas. Como proposta ao desenvolvimento de software com suporte à evolução dinâmica e não antecipada, foi definido o modelo de componentes COMPOR - Componente Model Specification (CMS) [5]. Este modelo de componentes permite a mudança de qualquer parte do software, somente removendo e adicionando componentes, mesmo em tempo de execução. Além disso, a evolução não antecipada é possível graças ao baixo acoplamento existente entre os componentes na CMS.

Contudo, a CMS não oferece suporte ao desenvolvimento de aplicações corporativas nativamente. Quando está desenvolvendo software utilizando este modelo de compoentes, o desenvolvedor precisa implementar qualquer característica necessária à aplicação corporativa (segurança, distribuição, transação, etc) e incorporá-la ao modelo. É importante deixar claro que "incorporar ao modelo" significa tornar a característica transversal e torná-la válida para qualquer componente de uma aplicação CMS.

O foco deste capítulo é apresentar a CMS ao leitor e explicar o seu funcionamento. Apresenta-se também um arcabouço orientado a objeto que implementa os conceitos da CMS e serve de base para a sua implementação em qualquer linguagem orientada a objeto.

4.1 Component Model Specification (CMS)

Na especificação da CMS, um sistema baseado em componentes consiste na composição de dois tipos de entidades: componentes funcionais e contêineres. Componentes funcionais

são entidades de software que implementam as funcionalidades específicas da aplicação, tornando essas funcionalidades disponíveis na forma de serviços e eventos. Um componente funcional não é composto de outros componentes, isto é, ele não possui componentes filhos. Componentes funcionais são os únicos elementos de software na CMS que implementam funcionalidades específicas do domínio da aplicação.

Contêineres são entidades que não implementam funcionalidades específicas do domínio da aplicação. Um contêiner controla o acesso de serviços e eventos aos seus componentes filhos, que podem ser componentes funcionais ou outros contêineres. Componentes funcionais tornam-se acessíveis quando inseridos dentro de um contêiner. É necessário existir ao menos um contêiner específico, chamado de *root container*, para poder executar a aplicação. Após inserir um novo componente em um contêiner, as tabelas de serviços e eventos para cada contêiner são atualizadas até o contêiner raiz da hierarquia. Na Figura 4.1 apresenta-se uma hierarquia definida na CMS, com suas tabelas de serviços providos e eventos de interesse [4].

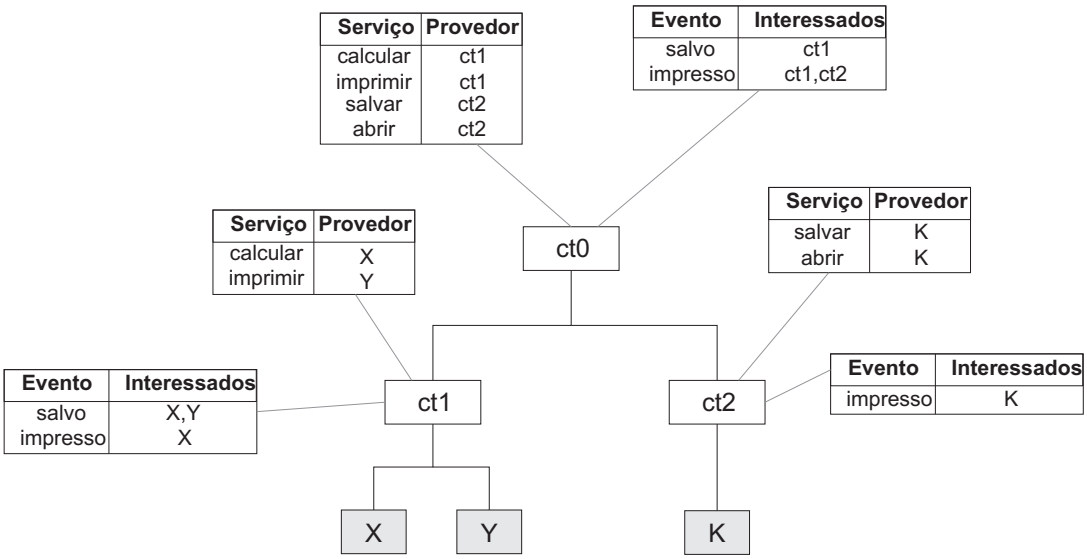

Figura 4.1: Hierarquia com tabela de serviços providos e eventos de interesses na CMS

O modelo de interação é baseado em serviços providos e eventos de interesse. No caso de serviços providos qualquer componente pode invocar um serviço de outro componente, mesmo quando o componente pertence a outro contêiner. A interação baseada em eventos ocorre através do anúncio de uma mudança de estado em um dado componente para todos os demais componentes interessados nesse evento. Em ambos os casos não há referência explícita entre os componentes.

Dada a hierarquia apresentada na Figura 4.1 e supondo que o componente *K* requisita a execução de um serviço "calcular". Então a interação baseada em serviços ocorre da seguinte forma:

1. O componente *K* requisita a execução do serviço "calcular" ao seu contêiner pai;
2. Como o contêiner *ct2* não conhece quem implementa o serviço requisitado, ele encaminha a requisição ao seu contêiner pai *ct0*;
3. *ct0* possui o serviço "calcular" em sua tabela de serviços providos e encaminha a requisição para o contêiner *ct1*, de acordo com as informações em sua tabela de serviços;
4. O contêiner *ct1* sabe que quem implementa o serviço "calcular" é o componente *X*;
5. Após a execução do serviço "calcular" pelo componente funcional *X*, a resposta para a execução percorre o caminho contrário até chegar ao componente que originou a requisição, neste caso o componente *K*;

Usando a mesma hierarquia da Figura 4.1, pode ser apresentada a interação baseada em eventos. Quando um evento é anunciado por um dado componente funcional, todos os componentes que possuem interesse nesse evento são notificados. Supondo a ocorrência de um evento "impresso" no componente *Y*, os passos para a notificação aos interessados ocorrem na forma descrita abaixo:

1. Ao executar o serviço "imprimir", o componente *Y* anuncia a ocorrência do evento "impresso";
2. O anúncio é recebido diretamente pelo seu contêiner pai. Este verifica em sua tabela de eventos de interesse se existe algum componente filho interessado;
3. Como existe um componente filho *X* interessado neste evento, o anúncio é encaminhado para ele. Caso este componente filho seja um contêiner, o anúncio é encaminhado até atingir o componente funcional realmente interessado na ocorrência do evento;

4. Após a verificação da tabela de eventos de interesse e possivelmente o anúncio destes eventos para seus componentes filhos, o contêiner encaminha o anúncio do evento para o seu contêiner pai *ct0*;

5. Os passos 2 e 3 são executados continuamente até chegar ao conteiner raiz;

4.2 Generic Component Framework (GCF)

Para que a CMS possa ser implementada em diversas linguagens de programação foi especificado um arcabouço em conjunto com a CMS, independente de linguagem denominado *Generic Component Framework* (GCF) [4], este arcabouço representa um modelo orientado a objeto que implementa a CMS. Com base nesse arcabouço foi possível implementar a CMS em linguagens de programação como Java [6], C++ [37], C# [19] e Python [11].

O projeto da GCF é baseado no padrão *Composite*[21], que é utilizado para permitir a composição de contêineres e componentes funcionais como definido pela CMS. A interação de serviços e eventos ocorre da mesma forma como apresentado na CMS. Na Figura 4.2 são ilustradas as principais classes do GCF [4].

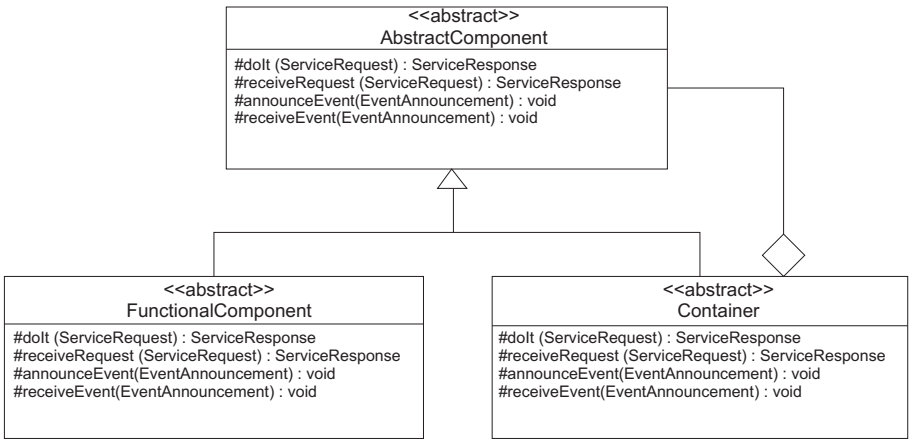

Figura 4.2: Principais classes do GCF.

A classe abstrata *AbstractComponent* mostrada na Figura 4.2 é utilizada como base para a implementação de componentes funcionais e contêineres. Na classe *AbstractComponent* estão definidas as assinaturas de quatro métodos essenciais para o correto funcionamento da aplicação, de acordo com a especificação da CMS, como descrito na Seção 4.1. Os métodos da GCF relacionados com a interação de serviços e eventos da CMS são:

1. **doIt** - É o método responsável por encaminhar requisições de serviços de um componente para seu componente pai.

2. **receiveRequest** - É responsável por encaminhar a requisição para o componente filho que a implementa, quando este é encontrado na hierarquia.

3. **announceEvent** - Método chamado quando um componente deseja anunciar a ocorrência de um evento.

4. **receiveEvent** - O método executado pelos componentes interessados na ocorrência de um evento.

A implementação destes métodos ocorre de forma diferenciada entre componentes funcionais e contêineres. Por exemplo, o método *receiveRequest* em contêineres somente repassa esta chamada para o componente filho que implementa o serviço, que pode ser um componente funcional ou outro contêiner. Enquanto que em componente funcionais, o método *receiveRequest* executa diretamente o método relacionado ao serviço e que foi implementado pelo desenvolvedor do componente.

Como toda a extensão feita sobre a CMS para o desenvolvimento de aplicações corporativas ocorreram sobre a GCF, o conhecimento das classes da GCF apresentadas na Figura 4.2, bem como a sua relação com as interações de serviços e eventos da CMS, são essenciais para o correto entendimento deste trabalho.

Capítulo 5

Infra-Estrutura para Aplicações Corporativas

A GCF atende os requisitos de evolução dinâmica e não antecipada citados anteriormente neste trabalho. Porém, estas características não são suficientes no desenvolvimento de aplicações corporativas. Aplicações precisam ser concluídas cada vez em menos tempo e com maior qualidade, em aplicações corporativas estes dois requisitos (tempo e qualidade) podem ser alcançados através do reuso de código comum em diversas aplicações.

Em aplicações corporativas o código comum consiste nas características de infra-estrutura utilizadas. Uma aplicação corporativa deve ser distribuída, para garantir escalabilidade do software, balanceamento de carga e tolerância a falhas; ser segura, para garantir a confidenciabilidade das informações e proteger os usuários; permitir serviços transacionais, para garantir a consistência nos dados e operações.

Este conjunto de requisitos de uma aplicação corporativa define as características de infra-estrutura necessárias ao modelo de componentes que se proponha a desenvolver aplicações deste tipo. Em modelos de componentes como EJB e C#, o ambiente de execução é a entidade de software responsável pela implementação e controle destas características.

Fornecer a infra-estrutura de uma aplicação, para que possa ser reutilizada pelo desenvolvedor, tornou-se uma exigência aos modelos de componentes para aplicações corporativas. A infra-estrutura de uma aplicação corporativa consiste na implementação de diversas características, as principais são:

- Persistência: Armazenamento e recuperação das informações da aplicação em um repositório de dados.
- Segurança: Autenticação e autorização de usuários (ou outras aplicações) no acesso aos serviços disponibilizados pela aplicação.
- Distribuição: Deve ser possível para a aplicação disponibilizar ou acessar funcionalidades remotamente.
- Transação: A aplicação deve controlar a execução de um determinado conjunto de operações para que sejam executadas como uma única transação.
- Balanceamento de carga: Consiste na distribuição da carga entre instâncias da aplicação. Os critérios para a distribuição de carga podem ser os mais variados: uso de espaço em disco; uso de processamento; largura de banda na rede; etc.
- Tolerância a falhas: Deve ser possível para a aplicação se recuperar de falhas não esperadas.
- Disponibilização na WEB: A aplicação deve prover acesso a seus serviços via WEB.

Neste trabalho foram selecionadas quatro características dentre as previamente citadas (segurança, distribuição, transação e balanceamento de carga). Estas características foram selecionadas levando-se em conta as características mais comuns e requisitadas em infra-estrutras existentes. Além disto, o tempo disponível para a execução deste trabalho foi relevante na escolha das características.

As características foram especificadas na forma de diagramas de classe como extensões da GCF. Como foi dito na Seção 4.2 a GCF consiste em um arcabouço orientado a objeto que serviu de base para a implementação da CMS em diversas linguagens, dentre as implementações uma foi em Java e é conhecida como *Java Component Framework - JCF* [6]. Esta foi a implementação utilizada para testar e validar a infra-estrutura proposta neste trabalho.

Como estas características foram especificadas como extensões da GCF, a CMS foi mantida inalterada e sua característica de evolução dinâmica e não antecipada foi mantida. A extensão que oferece suporte a transação foi especificada e implementada em um trabalho de

conclusão de curso [16], enquanto distribuição, segurança e balanceamento de carga foram especificadas e implementadas no contexto deste trabalho.

Um requisito básico e intuitivo em aplicações corporativas é que estas características devem coexistir no mesmo ambiente de execução. Por exemplo, o uso da característica de segurança não deve inviabilizar a utilização de outras características, como transação ou distribuição. No trabalho apresentado neste documento, foi necessária uma revisão da especificação das características de infra-estrutura com o intuito de garantir esta coexistência. Esta revisão possibilitou a identificação de uma pós-condição que deve ser verificada em qualquer característica que venha a ser especificada e implementada sobre a CMS. Esta pós-condição é enunciada abaixo.

"No que diz respeito a manipulação de requisições de serviços, toda a característica de infra-estrutura deve receber como entrada e gerar como saída uma requisição de serviço do mesmo tipo."

Esta regra se aplica também ao anúncio de eventos. Sendo assim, nenhuma característica de infra-estrutura deve manipular diretamente a hierarquia para requisitar um serviço ou anunciar um evento. Considerando esta pós-condição, o procedimento de inserção e remoção de características de infra-estrutura de uma aplicação é apresentado nas Figuras 5.1, 5.2 e 5.3.

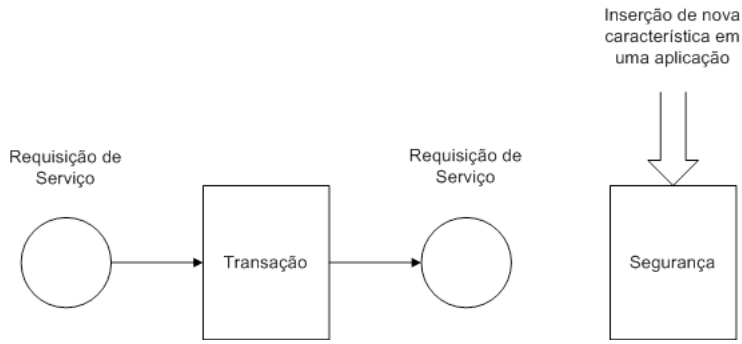

Figura 5.1: Característica de transação

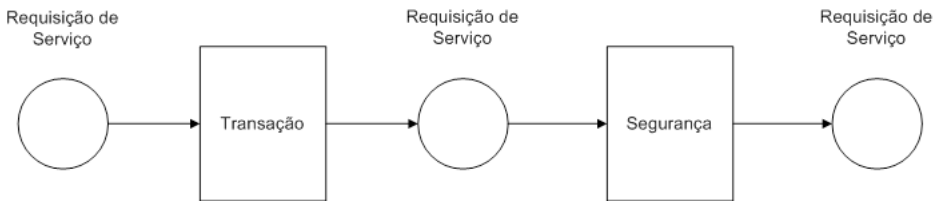

Figura 5.2: Características de transação e segurança

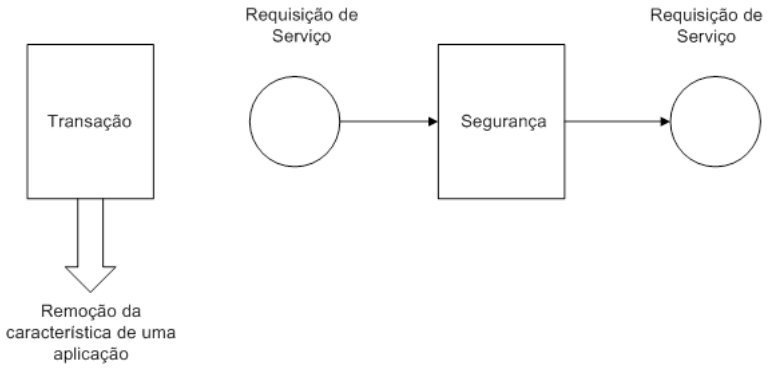

Figura 5.3: Característica de segurança

Na Figura 5.1 é apresentado o fluxo de uma requisição de serviço em uma aplicação que possui somente a característica de transação. Seguindo a pós-condição apresentada anteriormente, esta característica deve receber uma requisição de serviço como entrada e gerar também uma requisição de serviço em sua saída. Na Figura 5.2 é mostrado como ocorre o funcionamento após a inserção da característica de segurança na aplicação. Como qualquer característica recebe como entrada e gera como saída o mesmo tipo de requisição de serviço, elas podem ser organizadas em sequência. Na Figura 5.3 é apresentado como a remoção de uma característica não deve interferir no funcionamento das características restantes.

No decorrer deste capítulo serão apresentadas as características especificadas e implementadas na infra-estrutura proposta neste trabalho. Cada uma delas foi revisada para atender a pós-condição apresentada anteriormente, garantindo assim a coexistência entre as mesmas.

5.1 Distribuição

Distribuição é uma característica básica em aplicações corporativas, pois pode trazer aumento de desempenho, economia na escala do software, além de recuperação e compartilhamento de recursos. Em um software distribuído cada módulo pode estar localizado em um diferente computador da rede. A comunicação entre esses módulos é normalmente baseada em troca de mensagens.

Na característica de distribuição que foi especificada e implementada neste trabalho, a entidade de software responsável pela troca de mensagens entre os módulos distribuídos

são contêineres da CMS. Cada contêiner é responsável por enviar requisições de serviços e anúncios de eventos para seus componentes filhos distribuídos. A Figura 5.4 ilustra o mecanismo de distribuição, que utiliza o padrão de projeto *Proxy* [21].

O mecanismo de distribuição que foi implementado é uma extensão do diagrama de classes da GCF. Desta forma a simplicidade do modelo de componentes foi mantida já que não existe dependência da CMS com o mecanismo de distribuição descrito.

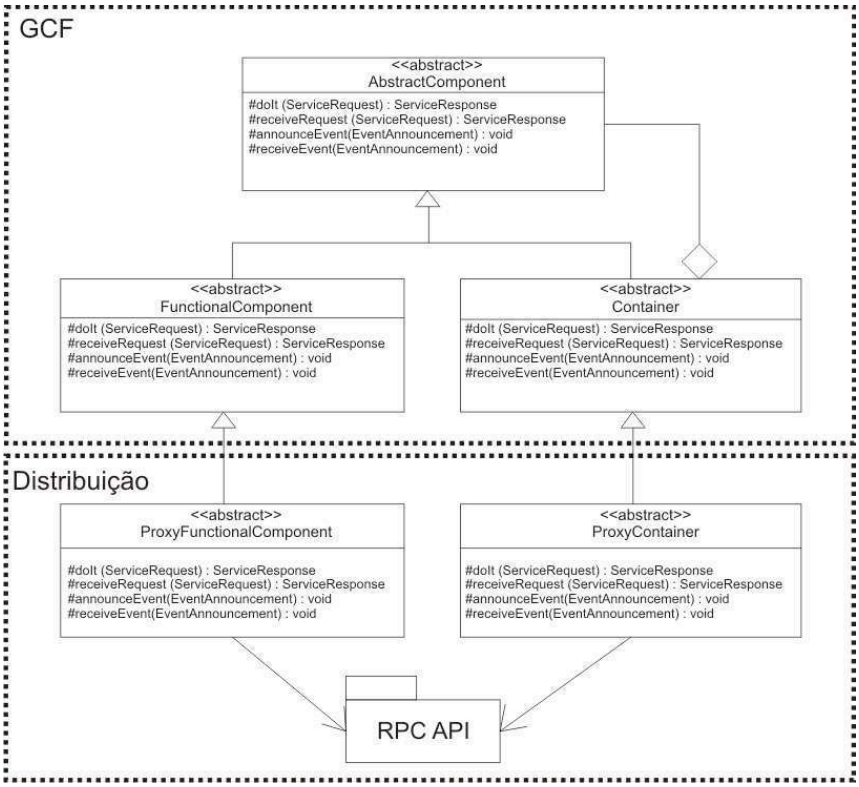

Figura 5.4: Diagrama de classes da arquitetura de distribuição

No mecanismo de distribuição apresentado, os contêineres que acessam componentes remotos possuem filhos que são na verdade entidades representativas para componentes remotos, utilizando o padrão *Proxy* [21] para isto. Estas entidades representativas são responsáveis pelo acesso a rede.

5.1.1 Implementação e funcionamento

O funcionamento da característica de distribuição será apresentado através do exemplo da Figura 5.5. Para iniciar o desenvolvimento de uma aplicação distribuída, o desenvolvedor deve

publicar a parte desejada da hierarquia de componentes em cada *host* onde se deseja hospedar componentes da aplicação, a configuração do mecanismo de distribuição é executado através dos seguintes passos em cada um dos *hosts*:

1. No *host 192.168.10.6*, o desenvolvedor deve criar uma instância da classe `ProxyFunctionalComponent` (`ProxyA`). Esta instância é recuperada do *host 192.168.10.1* através de uma chamada remota de procedimento e é um *proxy* para o componente real que está localizado no *host 192.168.10.1*. Note que o proxy é um filho do contêiner localizado no *host 192.168.10.6*. e funciona como uma entidade representativa para o componente real (A) localizado no *host 192.168.10.1*

2. No *host 192.168.10.1* o componente real é adicionado como um filho de `ProxyCont1`, que é uma instância de `ProxyContainer` recuperada do *host 192.168.10.6* através de uma chamada remota de procedimento. A instância `ProxyCont1` é uma entidade representativa para o contêiner `Cont1` localixado no *host 192.168.10.6*

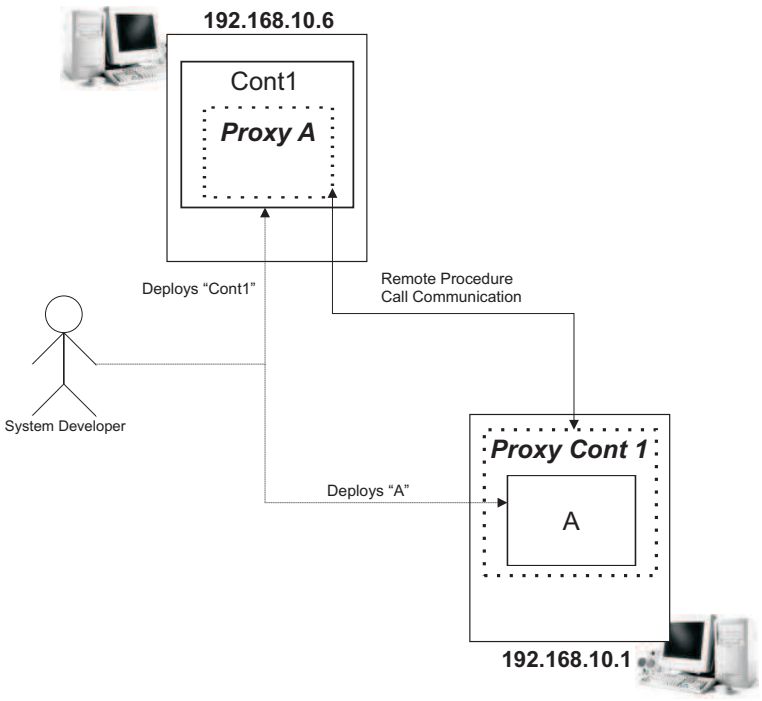

Figura 5.5: Publicação e funcionamento do mecanismo de distribuição.

As instâncias das classes `ProxyFunctionalComponent` (`ProxyA`) e `ProxyContainer` (`ProxyCont1`) são registradas em cada *host*. Esta operação é necessária porquê este registro possui informações sobre como o proxy acessa seus componentes remotos.

Na arquitetura distribuída, o modelo de componentes troca requisições de serviços e anúncios de eventos entre dois *hosts* de uma forma transparente. Para implementar a comunicação via rede, o mecanismo de distribuição implementado sobre a JCF utiliza a implementação da Apache [7] do protocolo XML-RPC [48].

5.2 Segurança

O mecanismo de segurança é essencial pois garante o controle do acesso aos provedores de serviços da aplicação por parte dos clientes. Um serviço de segurança satisfatório deve garantir autenticação de clientes e autorização destes clientes na execução de determinados serviços.

Como mostrado no Capítulo 4, na CMS um apelido é usado para identificar unicamente serviços e eventos com o mesmo nome em diferentes componentes. Contudo, esta estratégia introduz um problema de segurança no modelo. Por exemplo, é possível inserir um provedor *X* entre um outro provedor *Y* e seus clientes com o objetivo de interceptar as requisições de clientes para *Y*. Isto pode representar uma forma intrusiva de fazer algo indesejável no sistema, dado que o provedor inserido *X* pode ser visto como um intruso.

Como esta falha de segurança não é tratada pelo modelo de componentes, a GCF deve provêr formas de manipular políticas de segurança para os modelos de interação e publicação. Estas políticas devem ser satisfeitas quando algum serviço é requisitado ou um evento é anunciado, bem como quando um componente é inserido ou removido de um contêiner. Esta infra-estrutura de segurança foi implementada sobre a JCF usando programação orientada a aspectos, com AspectJ [24]. Aspectos permitem ocultar a complexidade do mecanismo de segurança do desenvolvedor e também simplificar o desenvolvimento de sistemas sem estes requisitos de segurança.

5.2.1 Implementação e funcionamento

O funcionamento do mecanismo de segurança é apresentado na Figura 5.6. A extensão da GCF para suporte a este mecanismo de segurança é apresentado na Figura 5.7. Nos passos abaixo é mostrado como o desenvolvedor pode começar a utilizar esta característica em uma aplicação desenvolvida sobre a CMS.

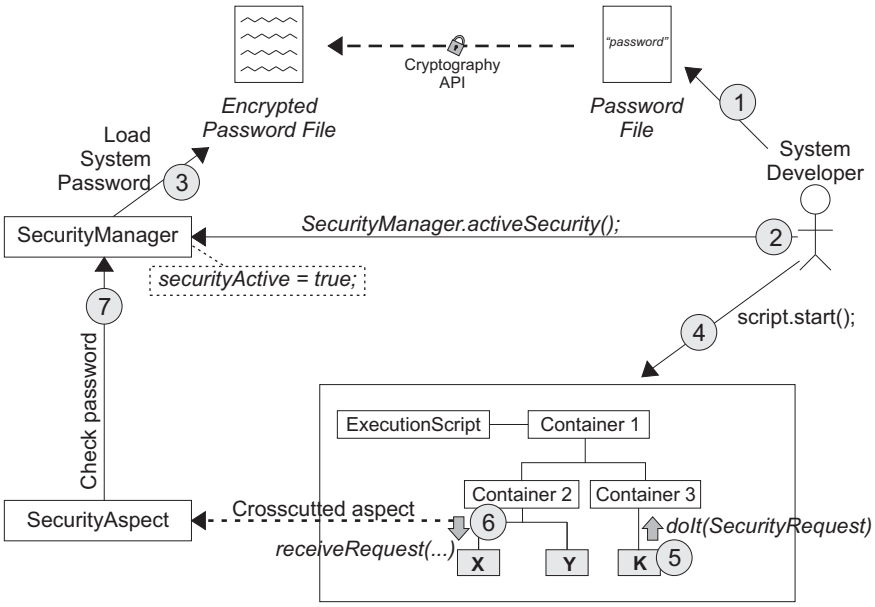

Figura 5.6: Arquitetura de segurança orientada a aspecto.

1. O desenvolvedor da aplicação cria um arquivo ".security" contendo a senha para acesso ao sistema e um arquivo ".policies" contendo as políticas de acesso aos serviços. Estes arquivos são criptografados por questões de segurança, para que somente a infraestrutura possa reconhecer seus conteúdos.

2. O mecanismo de segurança deve primeiramente ser ativado invocando o método `activeSecurity()` da classe `SecurityManager`. Esta operação define que todas as execuções de serviços, anúncios de eventos e adições de componentes devem ser autenticadas e autorizadas.

3. A instância da classe `SecurityManager` recupera a senha e a informação de políticas de acesso criadas no passo 1 e armazena em memória.

4. Após iniciar o contêiner raiz, todos os componentes da aplicação são iniciados e a aplicação pode então iniciar sua execução como uma sequência de invocações de serviços e anúncios de eventos.

5. Quando um componente qualquer requesita a execução de um serviço e o mecanismo de segurança está ativado, o componente requisitante deve encaminhar uma instância de `ServiceRequest` como parâmetro, contendo a senha do sistema entre suas propriedades.

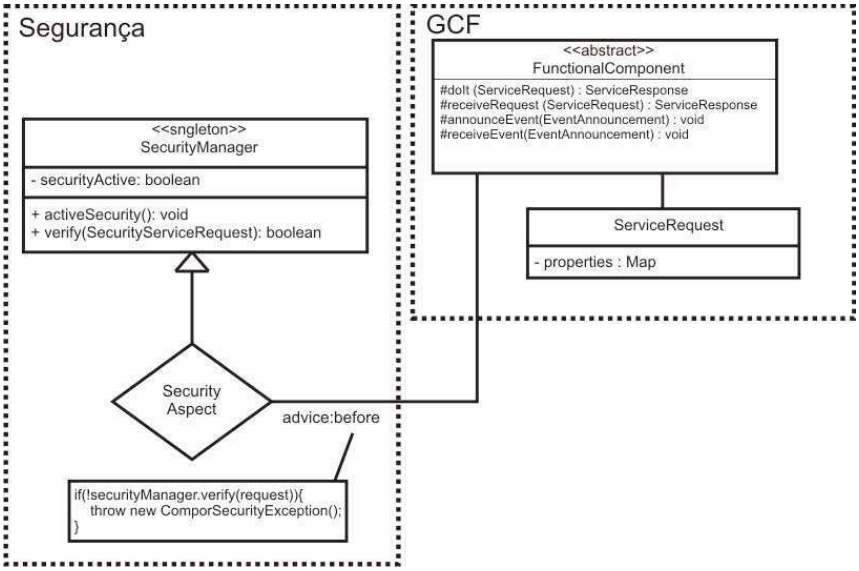

Figura 5.7: Diagrama de classes da arquitetura de segurança

6. O componente provedor do serviço recebe a requisição via o método `receiveRequest`, então o aspecto `SecurityAspect` intercepta a invocação do método e solicita ao `SecurityManager` que autentique e autorize o componente para acesso ao serviço.

7. `SecurityManager` verifica a validade da senha e permite a execução do serviço. Em outro caso, uma exceção do tipo `ComporSecurityException` ocorre.

5.3 Transação

Serviços transacionais são um conjunto de serviços que devem ser executados atomicamente, ou seja, todos devem ser executados ou nenhum deles. A existência de transações é uma característica essencial em aplicações corporativas pois garante a consistência das operações realizadas. Existem diversos protocolos que podem ser implementados para o controle de transações. Neste trabalho foi especificado sobre a GCF e implementado o protocolo *two-phase commit*, um popular protocolo usado para garantir consenso entre todos os membros participantes de uma transação [27]. No restante desta seção será apresentado este protocolo e como o controle de transação ocorre com a extensão especificada.

Assim como no mecanismo de segurança, a abordagem orientada a aspecto foi utilizada,

permitindo a separação do mecanismo de transação e o desenvolvimento de sistemas sem este mecanismo somente pela remoção do aspecto responsável. Com isto, a simplicidade do modelo CMS foi mantida, desde que ela não depende do mecanismo de transação. O diagrama de classe referente a extensão realizada na GCF pode ser visto na Figura 5.8

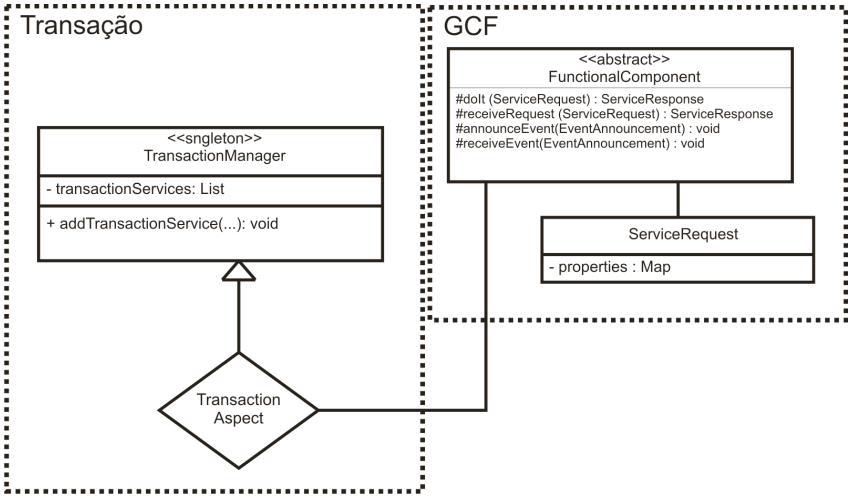

Figura 5.8: Diagrama de classes da arquitetura de transação

5.3.1 O Protocolo *Two-Phase Commit*

O protocolo *two-phase commit* define um coordenador que é responsável por gerenciar a transação. Na primeira fase do protocolo, os participantes devem enviar mensagens de inicialização (*init*) da transação ao coordenador. Estas mensagens possuem a informação de que a operação pode ou não ser executada pelo participante da transação. Assim que o coordenador recebe a resposta de todos os participantes, inicia-se a segunda fase do protocolo, onde o coordenador decide se a transação deve ser concluída com sucesso (*commit*) ou finalizada por uma falha de um ou mais participantes (*rollback*). Após a decisão uma mensagem de *commit* ou *rollback* é enviada a todos os participantes.

5.3.2 Implementação e funcionamento

A Figura 5.9 ilustra o componente *K* requisitando a execução de dois serviços. O primeiro (*withdraw*) é implementado pelo componente *X*, o segundo (*deposit*) é implementado pelo componente *Y*. Como a operação composta pela execução destes dois serviços em conjunto

representa uma transferência bancária, esta operação deve ser atômica (ou indivisível), o que significa que o dinheiro ou move de uma conta para a outra, ou permanece na primeira conta. No restante desta seção serão demonstrados os passos do funcionamento do mecanismo de transação, tomando como exemplo o caso de transferência bancária:

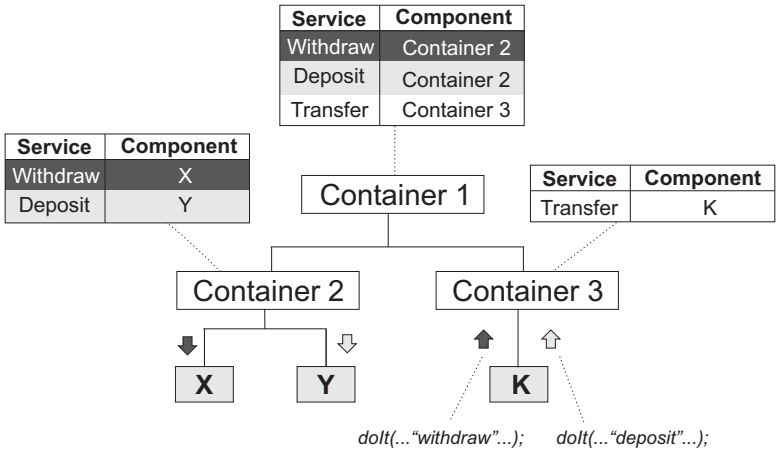

Figura 5.9: Necessidade de operação atômica.

1. Antes de tudo, é responsabilidade do desenvolvedor do serviço a implementação das três operações possíveis no protocolo *two-phase commit* como descrito na Seção 5.3.1: *init*, o desenvolvedor deve verificar se a operação poderá ser executada ou não e sinalizar ao coordenador da transação; *commit*, onde a operação é executada normalmente; *rollback*, onde a operação é desfeita e qualquer tarefa executada na operação *init* também deverá ser desfeita. Estas três operações devem ser configuradas como serviços normais da CMS.

2. O desenvolvedor do componente deve criar algumas propriedades adicionais para configurar os serviços que participam de uma transação, no exemplo apresentado os serviços participantes são *withdraw* e *deposit*. Estas propriedades criadas pelo desenvolvedor são: *transaction:active*, que informa ao mecanismo de transação para considerar este serviço como participante de uma transação; *transaction:init*, contém o apelido do serviço que implementa a operação *init* apresentada no passo 1; *transaction:commit*, contém o apelido do serviço que implementa a operação *commit* apresentada no passo 1; e *transaction:rollback*, contém o apelido do serviço que implementa a operação *rollback* apresentada no passo 1.

3. Antes de cada serviço ser executado pelo componente provedor, dada uma requisição de um cliente, o aspecto de transação *Transaction Aspect* mostrado na Figura 5.8 é disparado. Este aspecto executa os seguintes passos: verifica a existência da propriedade *transaction:active*; dispara o serviço definido na propriedade *transaction:init*; e após receber a resposta de todos os serviços participantes da transação, executa o serviço apropriado (*commit* ou *rollback*) de todos os participantes. Se a requisição não possui a propriedade *transaction:active* o serviço é executado normalmente.

Como a definição sobre qual operação (*init*, *commit* ou *rollback*) deve ser executada pelo provedor do serviço é garantida pela infra-estrutura, a consistência da operação atômica também é garantida.

Para garantir a conscistência de dado em caso de problemas de hardware, cada transação é registrada. Quando o sistema volta a funcionar, o aspecto anexado à sua inicialização recupera o estado anterior ao problema lendo o registro das transações.

5.4 Balanceamento de Carga

O balanceamento de carga consiste na distribuição de processamento entre dois ou mais computadores conectados. A principal finalidade em se utilizar balanceamento de carga é o aumento de desempenho em serviços que são constantemente requisitados na aplicação. Isto ocorre através da replicação do componente responsável pelo serviço em dois ou mais computadores. Além disto, uma entidade de hardware ou software é responsável por selecionar a instância do componente que deve executar o serviço, dada uma requisição.

O balanceamento de carga é uma característica essencial em aplicações corporativas de uso intenso. Aplicações que sofrem diversas requisições simultâneas para processamento de uma mesma tarefa se encaixam neste caso. O ideal nestas situações é que o administrador da aplicação possa adicionar novos computadores que auxiliem no processamento da tarefa, sem precisar alterar seu próprio código para se adequar ao novo computador.

Como dito anteriormente, deve existir alguma entidade de software ou hardware responsável por selecionar qual computador deve executar um serviço. Na GCF o módulo de software que intuitivamente recebe esta responsabilidade é o `Container`, por ser ele o responsável por enviar requisições de serviços a seus componentes filhos. Porém, uma

característica na especificação do `Container` da CMS torna inviável utilizá-lo como balanceador de carga sem que seja feita uma alteração em seu código. O `Container` mantém uma relação um-para-um entre um serviço e o componente filho que o executa, sendo que em um balanceador de carga deve ser possível encaminhar uma mesma requisição de serviço para mais de um componente.

A partir desta limitação foi que surgiu a necessidade de estender a GCF, definindo a classe `LoadBalancingContainer` como pode ser visto no diagrama apresentado na Figura 5.10. Este contêiner executa as mesmas tarefas da classe `Container`, contudo ele mantém uma relação de um-para-muitos entre um serviço e os diversos componentes filhos que podem executá-lo. Além de ser responsável por selecionar um destes componentes quando for necessário.

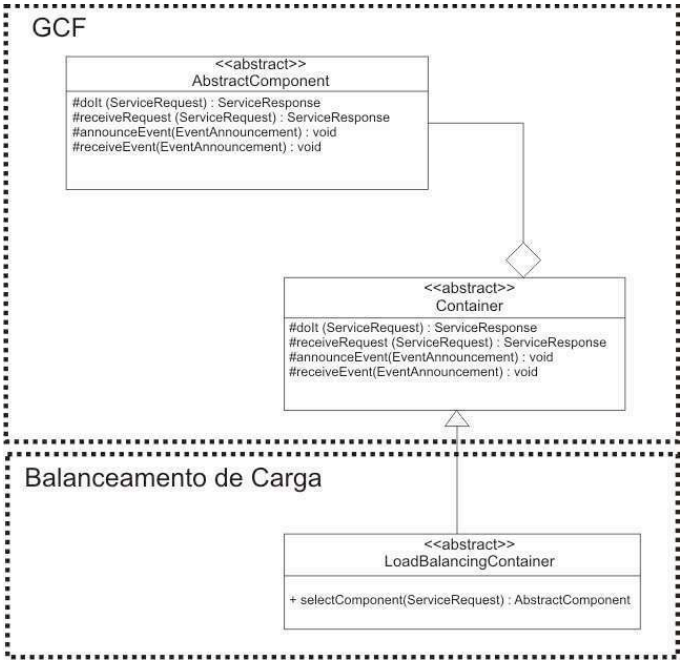

Figura 5.10: Diagrama de classes da arquitetura de balanceamento de carga

O método abstrato *selectComponent* apresentado na Figura 5.10 é um ponto de extensão da especificação. Este método representa o algoritmo utilizado para seleção do componente adequado quando uma requisição de serviço chega ao `LoadBalancingContainer`. Neste trabalho foi implementado o algoritmo de escalonamento *Round Robin* para prova de conceito, porém algoritmos mais complexos podem ser implementados sem maiores dificuldades. No algoritmo de escalonamento *Round Robin* um cliente solicita a execução de um

serviço ao roteador do mecanismo de balanceamento de carga, esta requisição é então encaminhada ao provedor do serviço que se encontra ociosa há mais tempo [39].

A utilização desta característica de balanceamento de carga em conjunto com a característica de distribuição apresentada na Seção 5.1, possibilita uma solução de balanceamento de carga completa, com cada componente filho de `LoadBalancingContainer` sendo uma instância de `ProxyContainer`, que são entidades representativas para os componentes que estão localizados em outros computadores.

5.4.1 Implementação e funcionamento

A utilização deste mecanismo de balanceamento de carga é semelhante a contêineres normais da GCF. O desenvolvedor da aplicação cria instâncias de componentes que implementam o mesmo serviço e adiciona estes componentes como filhos de uma instância da classe `LoadBalancingContainer`. A diferença é que o mecanismo de balanceamento de carga, ao invés de sobrescrever o componente responsável pela execução de um serviço, irá manter uma lista de componentes que podem executar este serviço.

5.5 Conclusão

Como apresentado no início deste capítulo, as características de infra-estrutura implementadas em modelos de componentes que se proponham ao desenvolvimento de aplicações corporativas são definidas pela regularidade com que estas características são exigidas em aplicações de domínios distintos. Atualmente as características apresentadas neste capítulo, e que foram especificadas e implementadas neste trabalho, estão incluídas no conjunto das características mais requisitadas atualmente por aplicações corporativas.

Contudo, a exigência de uma nova característica em aplicações corporativas pode se tornar comum, inserindo esta característica no conjunto de características desejáveis em modelos de componentes para aplicações corporativas, como a integração com aplicações móveis por exemplo. Além disso, características como persistência de dados, disponibilização de aplicações na internet e etc, são atualmente requisitadas pela maioria aplicações corporativas mas não foram especificadas e implementadas neste trabalho.

As características previstas em aplicações corporativas não foram completamente especificadas por não se tratar do foco deste trabalho a disponibilização de uma infra-estrutura completa para o desenvolvimento de aplicações corporativas, mas sim, a disponibilização de uma infra-estrutura para o desenvolvimento de aplicações corporativas com suporte à evolução dinâmica e não antecipada. Novas características podem vir a ser especificadas e implementadas utilizando as recomendações apresentadas no início deste capítulo.

Capítulo 6

Ambiente de Execução

Um ambiente de execução é uma entidade de software sobre a qual aplicações desenvolvidas especificamente para este ambiente podem ser executadas e gerenciadas. A principal funcionalidade de um ambiente de execução é receber comandos de entrada via uma interface, para serem direcionados às aplicações gerenciadas por ele. Os seguintes tipos de softwares são considerados ambientes de execução: Sistemas operacionais, Sistemas gerenciadores de banco de dados, Servidores de aplicação e etc.

Neste trabalho, os ambientes de execução são utilizados como plataforma para a execução de aplicações corporativas desenvolvidas com a extensão apresentada no Capítulo 5. Estes tipos de ambientes de execução são chamados de servidores de aplicação e estão localizados na camada do meio em uma arquitetura de três camadas orientada a servidor, como ilustrada na Figura 6.1.

Neste capítulo será apresentado a especificação, a implementação e o funcionamento do ambiente de execução (servidor de aplicação) responsável pelo gerenciamento de aplicações corporativas desenvolvidas com a extensão da CMS proposta neste trabalho. Este servidor de aplicação recebeu o nome de *Component Application Server - CAS*. Além disto, será apresentado também neste capítulo um cliente em linha de comando para acesso remote ao CAS.

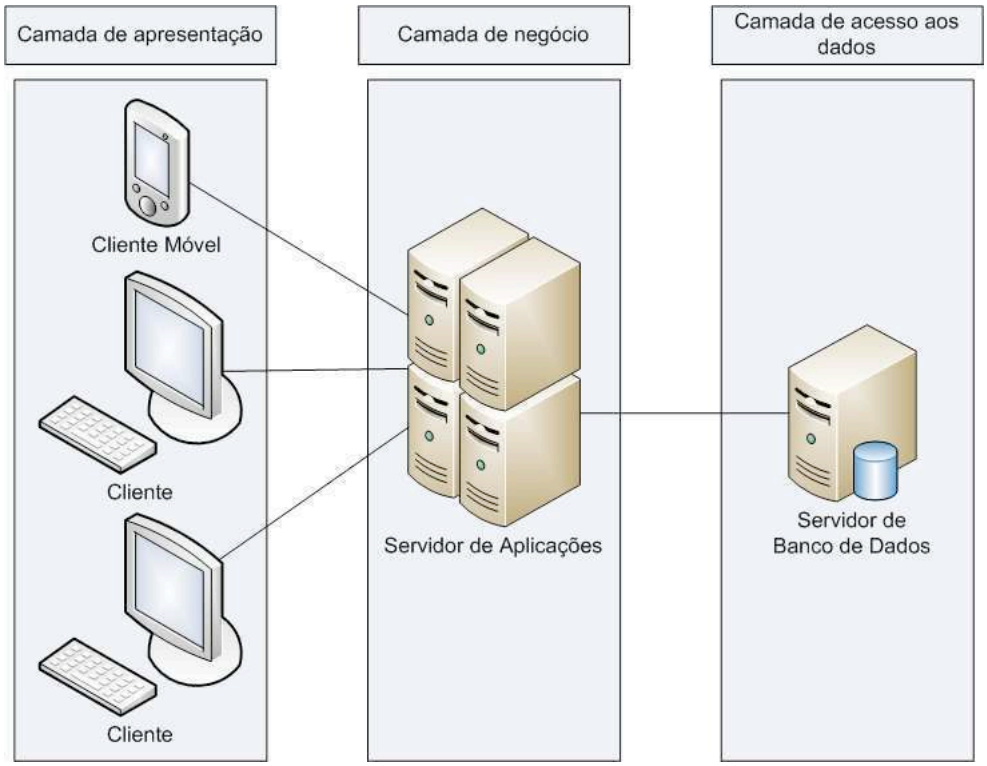

Figura 6.1: Arquitetura de desenvolvimento em três camadas

6.1 Especificação

O CAS é um ambiente de execução para aplicações corporativas desenvolvidas a partir da CMS, portanto é um servidor de aplicação cuja especificação é independente de linguagem. A sua especificação foi definida em duas etapas: inicialmente um diagrama de componentes foi elaborado, onde uma interface para acesso externo e a comunicação do CAS com a aplicação CMS foram definidos; na etapa seguinte foi definido um diagrama de classes baseado no diagrama de componentes resultante da primeira etapa, este diagrama de classes serve de base para implementação do CAS em uma linguagem de programação orientada a objeto.

O diagrama de componentes resultante da primeira etapa da especificação pode ser visto na Figura 6.2 e o diagrama de classes a ser utilizado como base para a implementação do CAS é mostrado na Figura 6.3.

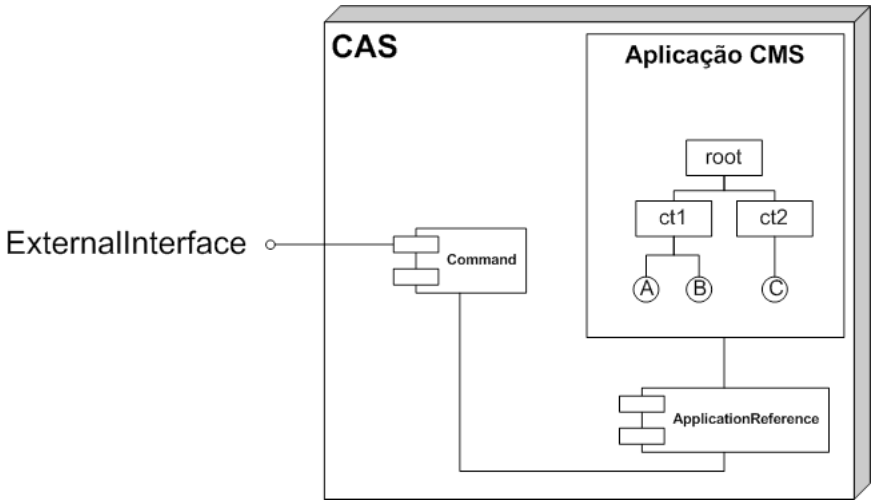

Figura 6.2: CAS - Diagrama de componentes

6.1.1 Interface para acesso externo

Como foi apresentado na Seção 6.1, uma interface para acesso ao CAS foi definida durante a especificação. Esta interface é composta por um conjunto de comandos que podem ser executados em um CAS e afetam diretamente a aplicação CMS gerenciada, configurando a hierarquia de componentes e iniciando a aplicação. Nesta seção serão apresentados estes comandos, com seus respectivos parâmetros e responsabilidades.

- **addComponent(parentName,componentClassName,componentName)** - Responsável por adicionar um novo componente funcional na aplicação gerenciada.
  - **parentName** - Nome do contêiner pai do novo componente, este contêiner deve existir previamente na aplicação gerenciada. Caso seja informado "null" o componente é somente criado para posterior adição em um contêiner ou proxy contêiner.
  - **componentClassName** - Referência para o arquivo físico onde o componente está definido.
  - **componentName** - Nome que o novo componente deve receber. Consiste em um nome único dentro da hierarquia.
- **addContainer(parentName,containerName)** - Responsável por adicionar um novo contêiner na aplicação gerenciada.

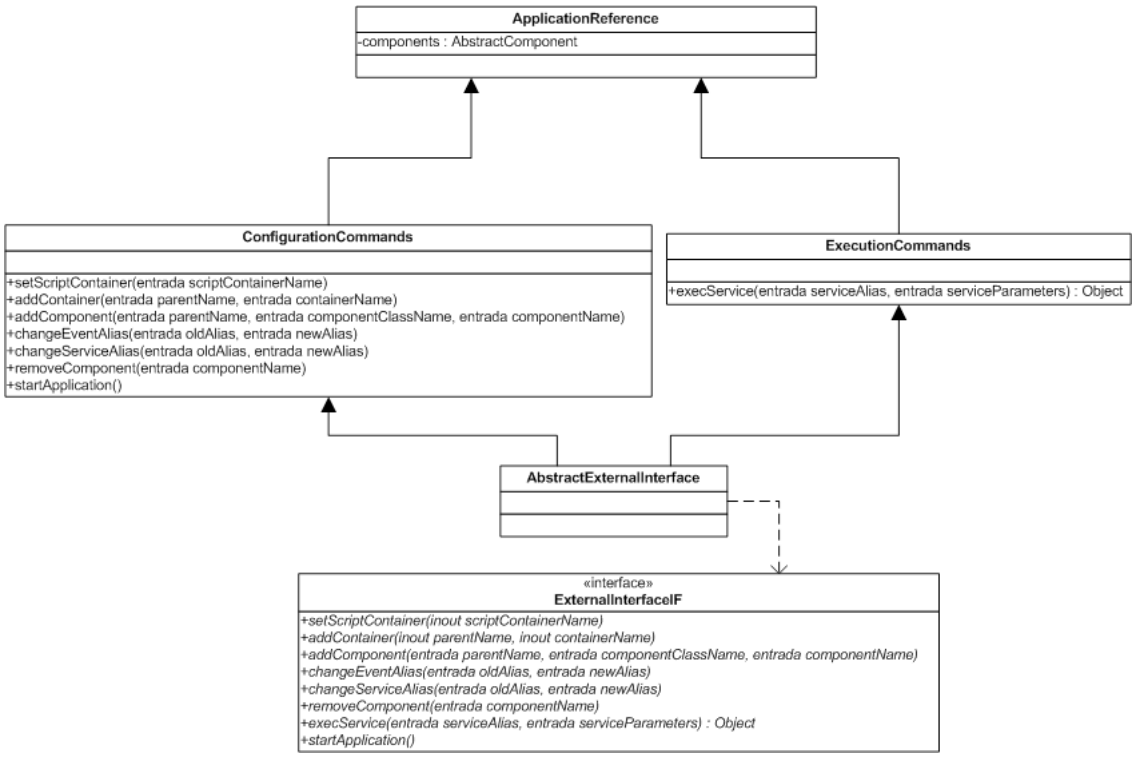

Figura 6.3: CAS - Diagrama de classes

  - **parentName** - Nome do contêiner pai do novo contêiner, este contêiner deve existir previamente na aplicação gerenciada. Caso seja informado "null" o componente é somente criado para posterior adição em um contêiner ou proxy contêiner.
  - **containerName** - Nome que o novo contêiner deve receber.

- **setScriptContainer(scriptContainerName)** - Defini o *Script Container* da aplicação, este é um contêiner especial que atua como a porta de entrada da aplicação CMS para clientes externos. Estes clientes acessam a aplicação através do *Execution Script* definido no *Script Container*.

  - **scriptContainerName** - Nome do *Script Container*.

- **changeServiceAlias(oldAlias,newAlias)** - Altera o apelido de um serviço definido na aplicação.

  - **oldAlias** - Apelido atual do serviço.
  - **newAlias** - Novo apelido do serviço.

- **changeEventAlias(oldAlias,newAlias)** - Altera o apelido de um evento definido na aplicação.
  - **oldAlias** - Apelido atual do evento.
  - **newAlias** - Novo apelido do evento.
- **removeComponent(componentName)** - Remove um componente ou contêiner da aplicação.
  - **componentName** - Nome do componente ou contêiner a ser removido.
- **execService(serviceAlias,serviceParameters)** - Executa um serviço existente na aplicação.
  - **serviceAlias** - Alias do serviço a ser executado.
  - **serviceParameters** - Lista de parâmetros do serviço a ser executado.
- **startApplication** - Informa que a aplicação está configurada e pronta para ser executada. Este comando deve ser executado antes de qualquer chamada ao comando **execService**.
- **addProperty(serviceAlias,property,value)** - Adiciona uma propriedade ao serviço informado.
  - **serviceAlias** - Alias do serviço a ser manipulado.
  - **property** - Nome da nova propriedade adicionada.
  - **value** - Valor da nova propriedade adicionada.
- **removeProperty(serviceAlias,property)** - Remove uma propriedade do serviço informado.
  - **serviceAlias** - Alias do serviço a ser manipulado.
  - **property** - Nome da propriedade removida.

6.2 Implementação

Nesta seção é apresentada a implementação da especificação apresentada na Seção 6.1. Além disto, foi implementado um módulo cliente para interação com o CAS. Estas implementações foram importantes para validação da solução proposta. Nas seções seguintes será chamado de CAS-Server a implementação do CAS apresentada anteriormente e de CAS-Client o módulo cliente que acessa o CAS-Server através da interface disponibilizada.

Para a implementação de ambos, do CAS-Server e do CAS-Client, foi utilizada a tecnologia OSGi [2]. Como apresentado na Seção 3.4, esta tecnologia consiste na especificação de um modelo de componentes e um ambiente de execução que oferece suporte à evolução dinâmica. Sendo assim, é possível atualizar um CAS-Server ou um CAS-Client sem suspender o uso da aplicação.

6.2.1 CAS-Server

O CAS-Server é a entidade de software que implementa a especificação do CAS como apresentada na Seção 6.1. O CAS-Server foi implementado como um *bundle* OSGi, podendo assim ser reconfigurado sem suspender a aplicação mantida por ele. *Bundles* são componentes na nomenclatura OSGi.

Um requisito do CAS-Server é que o mesmo possa ser gerenciado remotamente, para permitir que a partir de um único ponto da rede o cliente possa ter acesso às diversas instâncias do CAS-Server e configurar sua aplicação distribuída. Para permitir isto a interface do CAS-Server foi implementada usando a tecnologia *R-OSGi* [36], que é uma extensão da arquitetura OSGi para permitir o acesso a serviços remotos de forma transparente.

Outras formas de acesso a *bundles* distribuídos foram avaliadas, a utilização de *Web Services* facilitou a implementação do CAS-Client, sendo possível a utilização de qualquer cliente *Web Service* para a configuração da aplicação distribuída. Contudo, a especificação OSGi está recebendo atualizações para permitir a comunicação entre *bundles* distribuídos [3] e o projeto *R-OSGi* implementa esta atualização da especificação, por este motivo foi a tecnologia escolhida neste trabalho.

6.2.2 CAS-Client

Como foi dito na Seção 6.2.1, para a implementação da interface de acesso do CAS-Server foi utilizado o projeto *R-OSGi*. Como o projeto *R-OSGi* provê a comunicação entre *bundles* OSGi, O CAS-Client foi implementado como um *bundle* OSGi, assim como o CAS-Server. Desta forma, qualquer comunicação entre um CAS-Client e vários CAS-Server ocorre de forma transparente para o desenvolvedor.

O CAS-Client reconhece a interface descrita na Seção 6.1.1 e pode enviar estes comandos a um CAS-Server conectado. Neste trabalho foi implementada uma interface em linha de comando para o CAS-Client, esta interface recebe o conteúdo digitado pelo usuário, faz um parser deste conteúdo e envia o comando adequado ao CAS-Server conectado.

Capítulo 7

Estudos de Caso

Neste capítulo são apresentadas duas aplicações, cujo objetivo é validar a infra-estrutura desenvolvida neste trabalho: A primeira aplicação é um gerenciador de transferência bancária com suporte a distribuição e controle de transações; A segunda é um transcodificador de vídeo remoto com suporte a balanceamento de carga. Ambas as aplicações foram desenvolvidas sobre a infra-estrutura apresentada no Capítulo 5 e o ambiente de execução apresentado no Capítulo 6 foi utilizado para gerenciar e evoluir as aplicações.

7.1 Gerenciamento de transações bancárias

Uma aplicação para gerenciamento de transações bancárias foi desenvolvida utilizando a infra-estrutura proposta. O principal objetivo desta implementação foi a validação do funcionamento das diversas características de infra-estrutura em conjunto. Alguns requisitos foram previamente determinados para a aplicação e este conjunto de requisitos definiram quais características de infra-estruturas seriam necessárias. Estes requisitos são listados a seguir.

1. A aplicação deve suportar um controle de transação na operação de transferência bancária com o objetivo de garantir que a quantia é transferida de uma conta para outra ou toda a operação é cancelada.

2. Os componentes responsáveis pelas operações bancárias (saque e depósito) e a aplicação cliente estão em computadores distintos e devem se comunicar através da

rede.

3. As agências bancárias precisam realizar uma autenticação e solicitar autorização para a realização de suas tarefas.

7.1.1 Implementação

Com base nestes requisitos foi detectado que as características de segurança, distribuição e transação são necessárias nesta aplicação. Foi então compilada uma versão da JCF com o suporte a tais características e a aplicação de transferência bancária foi implementada sobre esta versão da JCF. A Figura 7.1 representa a hierarquia de componentes desta aplicação e sua arquitetura distribuída. No restante desta seção será apresentado em detalhes a implementação da aplicação, os componentes responsáveis pelos serviços *transfer*, *deposit* e *withdraw*, além da comunicação entre os componentes da aplicação através da característica de distribuição disponibilizada pela infra-estrutura.

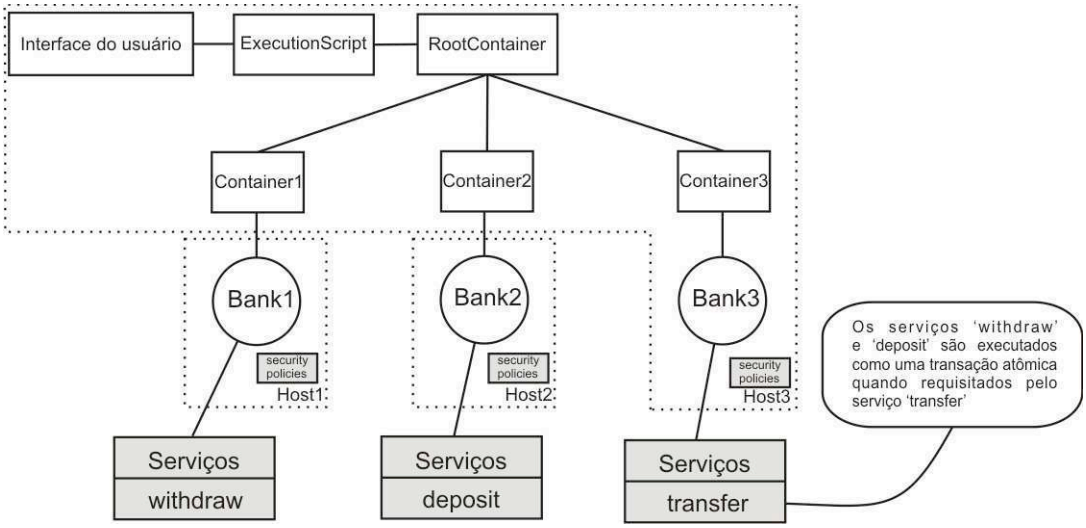

Figura 7.1: Arquitetura do gerenciador de transações financeiras

Componente executor da transferência

No computador de nome *Host3* encontra-se a interface gráfica apresentada na Figura 7.2, juntamente com o componente responsável pelo serviço de transferência bancária *transfer* (*Bank3*). Este serviço segue a implementação de serviços transacionais como descrito na

Seção 5.3. O trecho de código referente a configuração destes serviços transacionais é apresentado na Listagem de Código 7.1. O desenvolvedor precisa criar as requisições participantes da transação e executá-las através do método *init* da classe *TransactionalServices*. Como os serviços serão executados pela hierarquia de componentes, a propriedade de segurança precisa ser configurada pelo desenvolvedor através da chamada ao método *addProperty* da classe *ServiceRequest*, configurando o valor da propriedade *security:password* com a senha da aplicação.

Código Fonte 7.1: Código do serviço *transfer* do componente *Bank3*

```
1  public void transfer(Money money) {
2    /*Cria o serviço de saque*/
3    ServiceRequest withdrawService = new ServiceRequest("withdraw", new
         Object[]{money});
4    /*Ativa a requisição como participante de uma transação*/
5    withdrawService.addProperty("transaction", "true");
6    /*Configura a senha da aplicação, parâmetro necessário para a
         característica de segurança*/
7    withdrawService.addProperty("security:password","compor-pass");
8
9    /*Cria o serviço de depósito*/
10   ServiceRequest depositService = new ServiceRequest("deposit", new
         Object[]{money});
11   /*Ativa a requisição como participante de uma transação*/
12   depositService.addProperty("transaction", "true");
13   /*Configura a senha da aplicação, parâmetro necessário para a
         característica de segurança*/
14   depositService.addProperty("security:password","compor-pass");
15
16   /*Cria o conjunto de serviços transacionais e adiciona os serviços '
         withdraw' e 'deposit'*/
17   TransactionalServices transactionalServices = new TransactionalServices
         ();
18   transactionalServices.addTransactionalService(withdrawService);
19   transactionalServices.addTransactionalService(depositService);
20   transactionalServices.init(this);
21 }
```

Figura 7.2: Tela da aplicação gerenciadora de transações financeiras

Componentes funcionais (*Bank1* e *Bank2*)

Os componentes *Bank1* e *Bank2* implementam os serviços participantes da transferência bancária citada anteriormente. Eles fazem parte de uma transação e por este motivo a implementação das operações *init*, *commit* e *rollback* é necessária em ambos os componentes. O código de um deles (*Bank1*) é apresentado na Listagem de Código 7.2. O código do componente *Bank2* é similar e por este motivo não foi apresentado. Além disto, alguns serviços utilizados pela interface gráfica foram omitidos para facilitar o entendimento da implementação da característica de transação.

Código Fonte 7.2: Código do serviço *deposit* do componente *Bank1*

```
1  public class BankBD1 extends FunctionalComponent {
2    private double moneyInBank;
3
4    public BankBD1() {
5      super("Bank1");
6      defineProvidedServices();
7    }
8
9    private void defineProvidedServices() {
10     try {
11       //Aqui foram declarados os 4 serviços deste componente:
12       //  - O serviço "deposit"
13       //  - Os serviços referentes as 3 operações utilizadas
14       //      dentro de uma transação ("depositInit",
```

```
15          //      "depositCommit" e "depositRollback").
16        } catch (NoSuchMethodException e) {
17          e.printStackTrace();
18        }
19      }
20
21      //Operação de commit, efetiva a transação
22      public void depositCommit(Money moneyObject) throws Throwable {
23        moneyInBank = moneyInBank + moneyObject.getMoneyValue();
24      }
25
26      //Operação de rollback, não executa nada neste caso
27      public void depositRollback(Money moneyObject) throws Throwable {
28        System.out.println("ROLLBACK: deposit");
29      }
30
31      //Se o limite de depósito for ultrapassado,
32      //a exceção ocorre e a operação de rollback deverá ocorrer
33      public void depositInit(Money moneyObject) throws Throwable {
34        if(moneyObject.getMoneyValue() > 5000)
35          throw new Exception("Não pode depositar mais que 5000");
36      }
37
38      //O serviço executado caso a característica
39      //de transação não seja ativada.
40      public void deposit(Money moneyObject) {
41        moneyInBank = moneyInBank + moneyObject.getMoneyValue();
42      }
43  }
```

Configuração dos componentes distribuídos

Até o momento foi apresentado como foram implementadas as características de transação e segurança no caso de transferência bancária. Falta apresentar como a característica de distribuição foi utilizada na configuração da arquitetura mostrada na Figura 7.1.

Na Seção 5.1 foi mostrado as etapas necessárias para que o desenvolvedor da aplicação

configure uma hierarquia distribuída. Nesta aplicação de transferência bancária, o código referente a execução destas etapas no *Host1* é mostrado na Listagem de Código 7.3. A primeira tarefa do desenvolvedor é iniciar o componente em uma porta TCP para que o mesmo possa ser referenciado pelo restante da hierarquia. Em seguida o desenvolvedor inicia o componente *bank1* e o adiciona em uma instância de *ProxyContainer*. Esta instância possui informações sobre o ip, a porta e o identificador do contêiner real do componente *bank1*.

Código Fonte 7.3: Configuração da distribuição em *Host1*

```
1  //Criação e inicialização do host
2  Registry.getInstance().addServer(5567);
3  Registry.getInstance().startServer(5567);
4  //Criação do componente funcional "Bank1"
5  bank1 = new BankBD1();
6  //Criação do "Proxy" que referencia o contêiner
7  //real deste componente que se encontra no "Host3"
8  new ProxyContainer(bank1, "W", "127.0.0.1", "5566");
9  //Inicialização do componente
10 bank1.start();
```

Um código equivalente ao mostrado nesta listagem é executado nos demais computadores participantes da hierarquia distribuída. O código executado no *Host3* se refere ao contêiner real do componente *bank1* e é apresentado na Listagem de Código 7.4.

Código Fonte 7.4: Configuração da distribuição em *Host3*

```
1  //Criação e inicialização do host
2  Registry.getInstance().addServer(5566);
3  Registry.getInstance().startServer(5566);
4
5  //Criação dos contêineres participantes da hierarquia e que se
       encontram neste host
6  //Inserção destes contêineres no contêiner raiz da aplicação.
7  Container2 cont2 = new Container2();
8  Container1 cont1 = new Container1();
9
10 scriptContainer.addComponent(cont1);
11 scriptContainer.addComponent(cont2);
12 scriptContainer.addComponent(new Container3());
```

```
13
14  //Criação dos "Proxy" que referenciam os componentes reais nos
15  //hosts remotos
16  new ProxyFunctionalComponent(cont1, "W", "127.0.0.1", "5567");
17  new ProxyFunctionalComponent(cont2, "Y", "127.0.0.1", "5568");
```

Utilização do CAS para construção da aplicação

Os códigos mostrados nas listagens 7.3 e 7.4 dizem respeito a construção da hierarquia de componentes da aplicação. Este código pode ser automatizado com a utilização do CAS-Server descrito no Capítulo 6. A utilização do CAS-Server permite também uma configuração posterior a inicialização da aplicação, inserindo ou removendo componentes e contêineres em tempo de execução. Nesta seção será apresentado como CAS-Server foi utilizado para gerenciar a execução da aplicação de transferência bancária.

Como citado no Capítulo 6, o CAS-Server foi implementado como um *bundle* OSGi [2] e precisa ser implantado em um *framework* OSGi para o seu funcionamento. O *framework* utilizado foi o Felix [8] da Apache Foundation [7]. Este *framework* é a implementação de referência da especificação OSGi atualmente.

Após a inicialização do Felix e da implantação do ambiente de execução CAS-Server, o desenvolvedor pode acessá-lo através de um CAS-Client. O CAS-Client implementado neste trabalho é uma interface em linha de comando através da qual o desenvolvedor pode manipular a aplicação gerenciada pelo CAS-Server.

Na Listagem de Código 7.5 é apresentada a sequência de comandos enviados pelo CAS-Client ao CAS-Server, estes comandos são equivalentes ao código mostrado na Listagem 7.3. A sequência de comandos das demais intâncias de CAS-Server da aplicação é similar e por este motivo não foram mostrados.

Código Fonte 7.5: Comandos enviados ao CAS-Server

```
1  //Criação do componente funcional "Bank1"
2  addComponent null,net.compor.applications.bank.hostbank1.BankBD1,bank1
3  //Criação do "Proxy" que referencia o contêiner
4  //real deste componente que se encontra no "Host3"
5  addProxyContainer bank1,W,127.0.0.1,5566
6  //Criação e inicialização do host e inicialização do componente
```

```
7 startApplication
```

7.1.2 Cenário de evolução

No estudo de caso apresentado surgiu o requisito de reportar ao usuário ocorrências de saque. O relatório gerado deveria ser enviado via e-mail para uma conta específica. Este componente, responsável por realizar o saque e enviar o e-mail ao usuário proprietário da conta, foi especificado e implementado previamente, sendo necessário somente sua utilização na aplicação existente.

Como este componente foi implementado de forma independente, ele não seguiu qualquer interface prevista pelo componente responsável por realizar a transferência bancária. Caracterizando o cenário de evolução não antecipada. Graças ao baixo acoplamento da CMS, foi possível inserir este novo componente na hierarquia sem realizar qualquer alteração no componente funcional *Bank3*.

Assim que este componente foi inserido na hierarquia e configurado para responder pelo serviço *withdraw*, o envio de e-mail ao usuário passou a ser realizado com sucesso.

7.2 Transcodificador distribuído - M-Transcoder

A aplicação M-Transcoder tem como objetivo distribuir o processo de transcodificação de vídeo, aplicando um processo de balanceamento de carga para otimizar o processo. No contexto desta aplicação, a infra-estrutura proposta foi utilizada para que a aplicação cliente requisite serviços de transcodificação remota de forma transparente para o usuário. Além disto, a infra-estrutura tem a capacidade de selecionar um computador com mais recursos disponíveis para realizar esta tarefa sem que o usuário tome conhecimento desta seleção.

7.2.1 Implementação

Com base nestes requisitos foi detectado que as características de distribuição e balanceamento de cargas são necessárias no desenvolvimento desta aplicação e uma versão da JCF com estas características foi compilada.

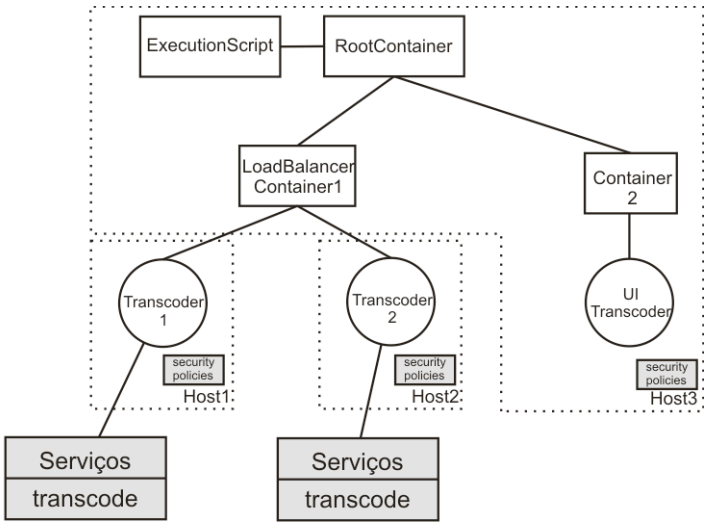

Figura 7.3: Arquitetura do M-Transcoder

Componentes responsáveis pela transcodificação

Os componentes *Transcoder1* e *Transcoder2* são responsáveis pela transcodificação do vídeo. Eles foram implementados utilizando a biblioteca Xuggler [49]. Xuggler é uma biblioteca em Java que o desenvolvedor pode utilizar para manipular arquivos de vídeo em suas aplicações. Na Listagem 7.6 é apresentado o código do componente *Transcoder1*. O código do componente *Transcoder2* é similar e por este motivo não foi mostrado.

Código Fonte 7.6: Código do serviço *transcode* do componente *Transcoder1*

```
1  public class Transcoder extends FunctionalComponent {
2    public Transcoder(String name) {
3      super(name);
4      //Definição do método "transcode" como
5      //o serviço "transcode" do componente Transcoder
6    }
7
8    public void transcode(String sourceUrl, String destinationUrl) {
9      //Código específico da biblioteca Xuggler para transcodificação
10     //do vídeo
11     IMediaReader reader = ToolFactory.makeReader(sourceUrl);
12     reader.addListener(ToolFactory.makeWriter(destinationUrl, reader));
13     while (reader.readPacket() == null)
14       do {} while(false);
```

```
15    }
16 }
```

Componente de interface com o usuário

O componente *UI Transcoder* é responsável pela interface com o usuário e por enviar requisições de transcodificação à hierarquia de componentes. A tela da aplicação é mostrada na Figura 7.4. Ao clicar em "transcodificar" a aplicação cria uma requisição ao serviço *transcode* e submete para a hierarquia através do código mostrado na Listagem 7.7.

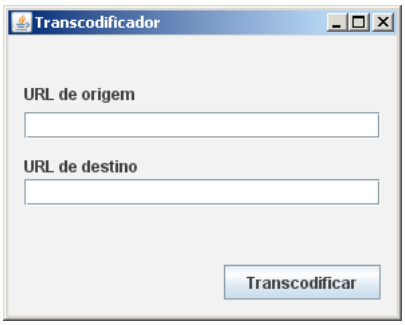

Figura 7.4: Tela da aplicação de transcodificação de vídeo

Código Fonte 7.7: Execução do serviço *trancode*

```
1  /*Cria o serviço de transcodificação*/
2  ServiceRequest transcodeService = new ServiceRequest("transcode", new
       Object[]{sourceURL,destinationURL});
3  /*Configura a senha da aplicação, parâmetro necessário para a
       característica de segurança*/
4  transcodeService.addProperty("security:password","compor-pass");
5  /*Solicita a execução do serviço de transcodificação*/
6  this.doIt(transcodeService);
7  }
```

A requisição ao serviço de transcodificação trafega na hierarquia até chegar ao contêiner responsável pelo balanceamento de carga. Como mostrado na Seção 5.1, o *LoadBalancer-Container* possui referência para uma lista de componentes que implementam um mesmo serviço e sabe como selecionar o próximo componente a atender a requisição de um serviço. No caso da aplicação de transcodificação, o contêiner responsável pelo balanceamento de

carga possui instâncias de *ProxyFunctionalComponent* como seus filhos. Desta forma foi possível integrar a solução de balanceamento de carga com a solução de distribuição e permitir que a aplicação selecione diferentes computadores para executar um mesmo serviço de forma alternada.

Um ponto forte da arquitetura desta aplicação é que mais componentes de transcodificação podem ser adicionados em outros computadores sem que a aplicação tenha a sua execução suspensa para isto. Implementando a mesma biblioteca utilizada anteriormente ou substituindo por outra. Para isto o desenvoledor precisa de um CAS-Client e ter conhecimento do endereço dos CAS-Servers para enviar os comandos necessários. O mecanismo de balanceamento de carga automaticamente irá passar a encaminhar requisições de transcodificação ao novo componente adicionado.

7.2.2 Cenário de evolução

A evolução não antecipada se tornou necessária neste caso de uso quando a realização de conversão de arquivos de audio foi exigida. Um componente chamado *TranscoderAudio-Video* foi implementado e substituiu o componente *Transcoder*. O código da listagem 7.7 do componente *UI Transcoder* permaneceu inalterado, pois não é feita referência direta ao componente *Transcoder*.

Capítulo 8

Considerações finais

A evolução de software é a fase do desenvolvimento com maior custo, tanto em termos financeiros quanto em termos de tempo de desenvolvimento. Quando a evolução do software não pode ser prevista em tempo de projeto, o impacto desta evolução sobre o código fonte e a arquitetura da aplicação são ainda mais significativas. Além do fator de evolução não antecipada em aplicações corporativas, o processo de evolução se torna ainda mais complexo em domínios onde as aplicações, devido a razões financeiras ou de segurança, precisam evoluir sem que haja uma interrupção da execução.

Neste trabalho foi apresentada uma infra-estrutura para o desenvolvimento de aplicações corporativas com suporte à evolução dinâmica e não antecipada. Com esta infra-estrutura o desenvolvedor pode se concentrar nas regras de negócio da aplicação e deixar por conta da infra-estrutura o controle de características comuns ao desenvolvimento de aplicações corporativas, como o controle de transações, autenticação e autorização de componentes, distribuição e balanceamento de carga.

Além de permitir um desenvolvimento concentrado nas regras de negócio da aplicação, a infra-estrutura permite um gerenciamento da execução da aplicação com o uso do CAS-Server. O CAS-Server permite ao desenvolvedor um controle total sobre a aplicação gerenciada por ele, permitindo adição e remoção de componentes, execução de serviços, mudança de nomes dos serviços providos pela aplicação e configuração do acesso remoto a outras instâncias do CAS-Server.

A partir da disponibilização da infra-estrutura apresentada neste trabalho, as aplicações corporativas que possuem o requisito de alta disponibilidade terão o tempo de desenvol-

vimento potencialmente reduzido, pois os esforços de implementação serão direcionados exclusivamente para as regras de negócio das próprias aplicações. Sendo assim, espera-se um aumento na produtividade e uma redução no custo de desenvolvimento. Já que as características comuns em aplicações corporativas são gerenciadas pela infra-estrutura.

Apesar da grande quantidade de desdobramentos futuros para este trabalho, a principal limitação atual e desafio futuro é o suporte à aplicações móveis. Aplicações móveis corporativas estão ganhando importância significativa no mercado. Em julho de 2009 a Motorola divulgou uma pesquisa onde constatou a crescente importância de aplicações móveis para corporações [34]. Além disto, grandes empresas estão firmando parcerias para simplificar o desenvolvimento de aplicações móveis corporativas. Um exemplo é a parceria firmada entre Sybase e Samsung [44].

Esta evolução nas aplicações corporativas exige uma revisão no processo de desenvolvimento, além de uma revisão das infra-estruturas disponíveis para o desenvolvimento e gerenciamento destas aplicações. Esta evolução guia os trabalhos futuros deste trabalho, onde a infra-estrutura apresentada será revisada e novas características serão adicionadas para que a infra-estrutura possa ser utilizada no desenvolvimento de aplicações móveis corporativas.

Referências Bibliográficas

[1] Alfresco. Open source enterprise content management system (cms) by alfresco, April 2009. http://www.alfresco.com/.

[2] OSGi Alliance. Open Services Gateway initiative. http://www.osgi.org - Acessado em 25/02/2009.

[3] OSGi Alliance. Open Services Gateway initiative - Specifications / Draft. http://www.osgi.org - Acessado em 25/02/2009.

[4] Hyggo Almeida. *Infra-estrutura Baseada em Componentes para o Desenvolvimento de Software com Suporte à Evolução Dinâmica Não Antecipada*. PhD thesis, Universidade Federal de Campina Grande, 2007.

[5] Hyggo Almeida, Glauber Ferreira, Emerson Loureiro, Angelo Perkusich, and Evandro Costa. A Component Model to Support Dynamic Unanticipated Software Evolution. In *Proceedings of International Conference on Software Engineering and Knowledge Engineering*, volume 18, pages 262–267, San Francisco, USA, 2006.

[6] Hyggo Almeida, Angelo Perkusich, Evandro Costa, Glauber Ferreira, Emerson Loureiro, Loreno Oliveira, and Rodrigo Paes. A Component Based Infrastructure to Develop Software Supporting Dynamic Unanticipated Evolution. In *Anais do XX Simpósio Brasileiro de Engenharia de Software*, pages 145–160, Florianópolis, SC, Brasil, 2006.

[7] Apache. Apache software foundation, August 2007. http://www.apache.org/.

[8] Apache. Apache felix, March 2009. http://felix.apache.org/.

[9] Apache. Apache geronimo, April 2009. http://geronimo.apache.org/.

[10] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM.

[11] Yguaratã Cavalcanti, Hyggo Almeida, and Evandro Costa. Um Arcabouço Open Source em Python para DBC com Suporte à Evolução Dinâmica não Antecipada. In *Proceedings of VIII Workshop de Software Livre*, Porto Alegre - RS, 2007.

[12] H. Cervantes. Beanome: a Component Model for Extensible Environments. http://www.adele.imag.fr/BEANOME - Acessado em 10/09/2007.

[13] H. Cervantes, D. Donsez, and R. Hall. Dynamic Application Frameworks using OSGi and Beanome. In *Proceedings of International Symposium and School on Advanced Distributed Systems*, Lecture Notes in Computer Science, pages 1–10, Guadalajara, Mexico, 2002.

[14] Compiere. Compiere open souce erp and crm business solution, April 2009. http://www.compiere.com/.

[15] Ivica Crnkovic. Component-based Software Engineering - New Challenges in Software Development. In *Software Focus*, volume 4, pages 127–133. Wiley, 2001.

[16] Márcio de Medeiros Ribeiro. Desenvolvimento de uma Infra-estrutura de Transações para o Arcabouço de Componentes COMPOR. Monografia de Conclusão de Curso de Graduação, Universidade Federal de Alagoas, Maceió, Alagoas, 2006.

[17] Frederic Doucet, Sandeep Shukla, Rajesh Gupta, and Masato Otsuka. An Environment for Dynamic Component Composition for Efficient Co-Design. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 736–743, Paris, França, 2002. IEEE Computer Society.

[18] Frederic Doucet, Sandeep Shukla, Masato Otsuka, and Rajesh Gupta. BALBOA: a Component-based Design Environment for System Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(12):1597–1612, 2003.

[19] Jardel Ferreira e Fabrício Barros. Uma Implementação em CSharp do Modelo de Componentes COMPOR. Monografia de Conclusão de Curso de Especialização, Centro de Estudos Superiores de Maceió, Maceió, Alagoas, 2006.

[20] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

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

[22] JBoss. Community driven open source middleware - jboss community, April 2009. http://www.jboss.org/.

[23] Kaffe. Kaffe Virtual Machine. http://www.kaffe.org - Acessado em 01/02/2008.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[25] Gunter Kniesel, Joost Noppen, Tom Mens, and Jim Buckley. 1st Int. Workshop on Unanticipated Software Evolution. In *ECOOP Workshop Reader*, volume 2548 of *LNCS*. Springer Verlag, 2002.

[26] Liferay. Liferay social office - liferay, April 2009. http://www.liferay.com/.

[27] M. Little, J. Maron, and G. Pavlik. *Java Transaction Processing*. Prentice Hall PTR, 1 edition, 2004.

[28] Juval Lowy. *COM and .NET Component Services: Migrating from COM+ to .NET*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[29] Juval Löwy. *Programming.NET Components*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[30] Tom Mens, Jim Buckley, Awais Rashid, and Matthias Zenger. Towards a taxonomy of software evolution. In *Workshop on Unanticipated Software Evolution*, 2002.

[31] Microsoft. .net framework developer center, August 2007. http://msdn.microsoft.com/netframework/.

[32] Microsoft. Microsoft corporation, May 2009. http://www.microsoft.com/.

[33] Sun Microsystems. Java.sun.com - The Source for Java Developers. http://java.sun.com/ - Acessado em 01/02/2008.

[34] Motorola. Motorola media center, July 2009. http://mediacenter.motorola.com/content/Detail.aspx?ReleaseID=11630&NewsAreaID=2.

[35] Oracle. Oracle and bea systems, April 2009. http://www.oracle.com/bea/index.html.

[36] R-OSGi. Transparent OSGi remote extension for distributed services. http://r-osgi.sourceforge.net/ - Acessado em 25/02/2009.

[37] André Rodrigues, Hyggo Almeida, and Angelo Perkusich. A C++ Framework for Developing Component Based Software Supporting Dynamic Unanticipated Evolution. In *The Nineteenth International Conference on Software Engineering and Knowledge Engineering*, pages 326–331, Boston, USA, 2007.

[38] Yoshiki Sato and Shigeru Chiba. Negligent class loaders for software evolution. In *RAM-SE*, pages 53–58, 2004.

[39] Milan E. Soklic. Simulation of load balancing algorithms: a comparative study. *SIGCSE Bull.*, 34(4):138–141, 2002.

[40] Ian Sommerville. *Software Engineering (7th Edition) (International Computer Science Series)*. Addison Wesley, May 2004.

[41] Rima Patel Sriganesh, Gerald Brose, and Micah Silverman. *Mastering Enterprise JavaBeans 3.0*. John Wiley & Sons, Inc., New York, NY, USA, 2006.

[42] SUN. Jsr-000220 enterprise javabeans 3.0 final release, May 2009. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html.

[43] SUN. Specifications, May 2009. http://java.sun.com/products/ejb/docs.html.

[44] Sybase. Sybase e samsung sds, July 2009. http://www.sybase.com.br/detail?id=1064623.

[45] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[46] Yves Vandewoude and Yolande Berbers. Supporting run-time evolution in seescoa. *J. Integr. Des. Process Sci.*, 8(1):77–89, 2004.

[47] WebSphere. Ibm software websphere, April 2009. http://www-01.ibm.com/software/br/websphere/.

[48] XML-RPC. Xml-rpc home page, August 2007. http://www.xmlrpc.com/.

[49] Xuggle. Xuggle xuggler, April 2009. http://www.xuggle.com/.