Um Ambiente C++ para o Desenvolvimento de Software com Suporte à Evolução Dinâmica não Antecipada

André Felipe de Albuquerque Rodrigues

Dissertação de Mestrado submetida à Coordenadoria do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande - Campus de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Engenharia da Computação

Angelo Perkusich, Dr.
Orientador

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

R696a
2008 Rodrigues, André Felipe de Albuquerque Rodrigues
Um Ambiente C++ para o desenvolvimento de software com suporte à evolução dinâmica não antecipada / André Felipe de Albuquerque Rodrigues Rodrigues. — Campina Grande, 2008.
76 f.

Dissertação (Mestrado em Ciências no Domínio da Engenharia Elétrica) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientadores: Prof. Dr. Ângelo Perkusich.

1. C++ 2. Arcabouço 3. Evolução Dinâmica de Software não Antecipada 4. Desenvolvimento Baseado em Componentes I. Título.

CDU – 004.43 (043)

UM AMBIENTE C++ PARA O DESENVOLVIMENTO DE SOFTWARE COM SUPORTE À EVOLUÇÃO DINÂMICA NÃO ANTECIPADA

ANDRÉ FELIPE DE ALBUQUERQUE RODRIGUES

Dissertação Aprovada em 22.08.2008

ANGELO PERKUSICH, D.Sc., UFCG
Orientador

Maria de Fatima Q. Vieira
MARIA DE FÁTIMA QUEIROZ VIEIRA, Ph.D., UFCG
Componente da Banca

LEANDRO DIAS DA SILVA, D.Sc., UFAL
Componente da Banca

CAMPINA GRANDE - PB
AGOSTO - 2008

Resumo

A evolução de software tem se caracterizado pelo seu alto custo tanto em termos financeiros quanto em termos de tempo de desenvolvimento. O impacto da evolução sobre o código fonte e a arquitetura da aplicação é mais significativo quando as mudanças nos requisitos do software não foram previstas ou antecipadas. O processo de evolução se torna ainda mais complexo em domínios onde as aplicações, devido a razões financeiras ou de segurança, precisam evoluir de forma dinâmica, ou seja, sem interromper a execução.

Dentro desse contexto, apresenta-se um ambiente C++ para o desenvolvimento de software com suporte à evolução dinâmica não antecipada. O ambiente C++ apresentado é composto por um arcabouço C++ denominado CCF e por um servidor de aplicação C++ denominado CCAS. O CCF é um arcabouço de componentes que fornece uma API para o desenvolvimento de aplicações C++ com suporte à evolução dinâmica não antecipada. O servidor CCAS implementa mecanismos para gerenciar a implantação e a execução das aplicações desenvolvidas sobre o CCF. Juntos, o CCF e o CCAS fornecem um ambiente C++ para o desenvolvimento de software com suporte à evolução dinâmica não antecipada totalmente transparente para o desenvolvedor.

O ambiente C++ foi validado através da implementação de uma aplicação para dispositivos móveis. O domínio dos sistemas embarcados foi escolhido para evidenciar o bom desempenho e o baixo consumo de memória das aplicações desenvolvidas utilizando o ambiente C++ proposto. Esses requisitos são de grande importância quando se trata de desenvolvimento de aplicações para dispositivos como celulares e *internet tablets*.

Abstract

Software evolution has been characterized by their huge cost and slow speed of implementation. The impact of software evolution on both the design and the existing code is more significant when the changes in the software requirements have not been planned or anticipated. The evolution process becomes even more complex in domains where applications, due to financial or security reasons, need to evolve dynamically, which means making changes to the software system while it is executing.

In this context, we introduce a C++ environment for dynamic unanticipated software evolution. Such an environment is composed of a C++ framework called CCF and a C++ application server called CCAS. CCF is a framework for developing component based software supporting dynamic unanticipated evolution. CCAS is a server that manages the deployment and execution of applications implemented through the CCF. Together, CCF and CCAS provide a C++ environment for dynamic unanticipated software evolution completely transparent to the developer.

The C++ environment was validated through the implementation of an application for a mobile device. The embedded systems domain was chosen to make clear the performance and low memory consumption of the applications developed using the proposed C++ environment. Such requirements are very important when developing applications to devices like cell phones and compact internet tablets.

Agradecimentos

Em primeiro lugar agradeço a Deus por ter me guiado não só neste momento mas em toda minha vida.

Agradeço a minha esposa Raphaela pelo apoio e por ter me aturado nesses quase seis anos de convivência.

Agradeço a meus pais, Antônio e Denise, e as minhas irmãs, Ana Patricia e Ana Carolina, por tudo que conquistei na vida, sem essas pessoas eu não teria conseguido.

Agradeço a professora Lígia Perkusich pelo grande incentivo em relação ao mestrado, sem ela certamente não estaria escrevendo este agradecimento.

Agradeço aos professores Angelo Perkusich e Hyggo Almeida pela orientação e por toda ajuda prestada para que eu pudesse concluir este trabalho.

Agradeço aos amigos do *Embedded*, em especial a Adrian, Diego, Yuri, Danilo, Zé Luiz, Olympio, Mário, Glauber e Marcos Fábio pelo companheirismo nestes anos de convivência.

Agradeço a todas as pessoas que contribuíram para que eu chegasse até aqui.

Por fim, não vou fazer outro agradecimento mas sim uma dedicatória. Dedico este trabalho a minha filha Beatriz.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Capítulo 1

Introdução

A maioria dos sistemas de software não podem ser completamente especificados e implementados uma única vez e para sempre [23]. Um software desenvolvido para endereçar um problema do mundo real deve ser continuamente adaptado para permanecer satisfatoriamente em uso. Tanto a implementação original quanto as versões subseqüentes são constantemente atualizadas. Tais mudanças são guiadas pelos resultados da execução do software e pelas mudanças do mundo externo. Quase todos os sistemas de software de sucesso estimulam requisições de mudanças e melhorias por parte do usuário. Evolução de software é inevitável, antes ou depois da liberação da versão inicial. O software que não fornece suporte para evolução irá gradualmente cair em desuso.

A evolução de software caracteriza-se pelo seu alto custo tanto em termos financeiros quanto em termos de tempo de desenvolvimento [7]. Um estudo publicado por Bennet Lientz em 1980 apontou a evolução como responsável por 50% do custo total do software [24]. Em um estudo mais recente, realizado por Len Erlikh em 2000, este valor subiu para 90% [14]. Motivadas por este cenário, diversas abordagens de engenharia de software [8, 20, 27] têm sido propostas para reduzir os custos da evolução de software, no entanto todas elas se baseiam na preparação da arquitetura do software para mudanças antecipadas. O problema surge quando uma parte do software que não havia sido preparada para mudanças precisa ser modificada. O impacto da evolução sobre o código fonte e a arquitetura do projeto é mais significativo quando as mudanças nos requisitos do software não foram antecipadas. Uma mudança é considerada não antecipada quando sua implementação não depende de pontos de extensão no código das versões anteriores do software [3].

Para alguns sistemas com o requisito de alta disponibilidade, tais como sistemas de telecomunicação, sistemas bancários, sistemas de hospitais e sistemas militares, não é aceitável interromper a execução do software devido a razões financeiras ou de segurança. Por exemplo, um dos requisitos de um sistema de telecomunicação é ter um tempo máximo

de 2 horas de interrupção em 40 anos de execução [35]. Uma opção para tais sistemas é usar uma infra-estrutura com suporte à Evolução Dinâmica de Software Não Antecipada (EDSNA), o que significa fazer mudanças em qualquer parte do software em tempo de execução.

1.1 Descrição do problema

A complexidade para introduzir um novo código em sistemas que já estão em execução depende muito da linguagem de programação e do ambiente utilizado [12]. Enquanto a inserção de código em tempo de execução é um procedimento não trivial em linguagens que são compiladas para código nativo como C e C++ [19,35], linguagens modernas como Java, SmallTalk e C# permitem a inserção de código em tempo de execução de uma maneira extremamente conveniente e segura. Nessas linguagens modernas, implementações de infra-estruturas com suporte à EDSNA são facilmente encontradas [2,6,29]. No entanto, em linguagens como C e C++ há uma escassez de implementações para tratar EDSNA.

Devido ao crescimento de domínios como sistemas embarcados e sistemas em tempo real, linguagens como C++ vêm ganhando força. Dentro desse contexto, alguns trabalhos em C++ com suporte à EDSNA têm sido propostos. O trabalho apresentado em [11] utiliza uma linguagem interpretada para fornecer suporte à EDSNA. Em [25], a EDSNA é alcançada através da definição de regras de interação. O trabalho proposto em [38] se baseia na arquitetura orientada a serviços [13] para fornecer suporte à EDSNA. No entanto, nenhum desses trabalhos fornece uma infra-estrutura de suporte à EDSNA totalmente transparente para o desenvolvedor, ou seja, além de se preocupar com a regra de negócio da aplicação, o desenvolvedor deve gerenciar os detalhes de implementação da infra-estrutura de EDSNA.

1.2 Objetivo

O objetivo desse trabalho é disponibilizar um ambiente C++ para o desenvolvimento de software com suporte à evolução dinâmica não antecipada. Tal ambiente é composto por um arcabouço de componentes C++ denominado CCF (*C++ Component Framework*) [32] e por um servidor de aplicação C++ denominado CCAS (*C++ Component Application Server*) [31]. O CCF é um arcabouço para o desenvolvimento de software baseado em componentes com suporte à evolução dinâmica não antecipada. Esse arcabouço implementa a Especificação do Modelo de Componentes COMPOR (CMS) [4], fornecendo assim, uma API para o desenvolvimento de aplicações C++ com suporte a

mudanças não antecipadas, mesmo em tempo de execução. A infra-estrutura disponibilizada pelo CCF permite o desenvolvimento de aplicações com suporte à EDSNA de forma transparente para o desenvolvedor, ou seja, o desenvolvedor não precisa gerenciar os detalhes de implementação do CCF. As aplicações implementadas sobre o CCF são implantadas e gerenciadas pelo servidor de aplicação CCAS.

O ambiente C++ é validado através da implementação de uma aplicação para dispositivos móveis. O domínio dos sistemas embarcados foi escolhido para evidenciar o bom desempenho e o baixo consumo de memória das aplicações desenvolvidas utilizando o ambiente C++ proposto. Esses requisitos são de grande importância quando se trata de desenvolvimento de aplicações para dispositivos como celulares e *internet tablets*.

1.3 Relevância

A partir da disponibilização do ambiente C++ apresentado neste trabalho, as aplicações C++ que possuem o requisito de alta disponibilidade terão o tempo de desenvolvimento 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. Além disso, essas aplicações poderão evoluir sem que haja nenhum tipo de planejamento prévio para tal, inclusive em tempo de execução.

A especificação e a arquitetura utilizada na implementação do ambiente C++ introduzido neste trabalho foi implementada anteriormente na linguagem Java [4]. Assim como a implementação C++, a implementação Java disponibiliza um ambiente para desenvolvimento de software com suporte à evolução dinâmica não antecipada. No entanto, o ambiente Java apresenta algumas limitações inerentes à linguagem: o desempenho impactado, pois Java é uma linguagem interpretada; e o consumo "desnecessário" de memória, devido a instalação e execução da máquina virtual Java. O desempenho e o consumo de memória são requisitos de grande importância em domínios tais como o de sistemas embarcados. Dentro desse contexto, o ambiente C++ apresentado neste trabalho fornece mecanismos para implementar aplicações de alto desempenho com suporte à evolução dinâmica não antecipada sem desperdício de memória.

1.4 Estrutura do documento

O restante do documento está organizado da seguinte forma:

- No Capítulo 2, são apresentados os conceitos relacionados ao trabalho proposto que são necessários ao entendimento do restante do documento, assim como, os trabalhos

relacionados.

- No Capítulo 3, apresenta-se a Especificação do Modelo de Componentes COMPOR (CMS) utilizada na implementação do arcabouço de componentes C++ proposto neste trabalho.

- No Capítulo 4, apresentam-se os detalhes de projeto, de implementação e de utilização do arcabouço de componentes C++ denominado CCF (*C++ Component Framework*).

- No Capítulo 5, os detalhes da arquitetura e da implementação do servidor de aplicação C++ denominado CCAS (*C++ Component Application Server*) são apresentados

- No Capítulo 6, apresenta-se a implementação do estudo de caso *DjVu Compactor*, ressaltando a utilização do CCF e CCAS no processo de desenvolvimento.

- No Capítulo 7, são apresentadas as considerações finais e os trabalhos futuros.

Capítulo 2

Fundamentação

Com o intuito de facilitar o entendimento do ambiente C++ proposto, os conceitos relacionados ao Desenvolvimento Baseado em Componentes (DBC) e à Evolução Dinâmica de Software Não Antecipada (EDSNA) utilizados neste documento são descritos neste capítulo. Além disso, os trabalhos relacionados também são apresentados.

2.1 Desenvolvimento Baseado em Componentes

Muitos sistemas apresentam módulos similares que são implementados de forma diferente. O reúso desses módulos, que muitas vezes são idênticos, aumentaria a qualidade do software e a produtividade do desenvolvimento, bem como, reduziria os custos e os esforços de produção. Nesse sentido, o Desenvolvimento Baseado em Componentes (DBC) surge como uma perspectiva de desenvolvimento de software caracterizada pela composição de módulos pré-existentes [18], embora os módulos por si só não sejam suficientes para garantir o reúso do software [34]. Os sistemas definidos através da composição de módulos permitem que sejam adicionadas e removidas partes do software sem a necessidade de sua completa substituição. Dessa maneira, o processo de evolução do software se torna menos custoso.

2.1.1 Componentes

Dentro do contexto de DBC, os módulos reutilizáveis são denominados **componentes**. Os componentes de software são unidades de composição com interfaces contratualmente definidas que são desenvolvidas independentemente e podem ser combinadas por terceiros [36]. Os componentes também podem ser vistos como o estado seguinte de abstração depois de funções e classes [34]. Algumas características inerentes aos componentes são apresentadas a seguir.

- O componente deve implementar a funcionalidade a que ele se propõe de forma clara e específica.
- O componente deve ter uma interface bem definida, que indica como ele pode ser utilizado e conectado a outros componentes. Os detalhes de como o componente é construído deve ser ocultado.
- É necessário que o componente apresente uma documentação indicando o que é exigido para sua utilização e o que ele provê.

Essas características facilitam o processo de combinação de componentes desenvolvidos ou comprados, aumentando assim, a qualidade e a velocidade de desenvolvimento e diminuindo o tempo de entrega do produto ao mercado.

2.1.2 Modelo componentes e arcabouço de componentes

Um componente não pode ser visto de forma completamente independente dos outros componentes com os quais se relaciona e de seu ambiente. O relacionamento entre os componentes deve estar de acordo com um conjunto de padrões e convenções definido pelo **modelo de componentes**. Segundo Bachmann [5], o modelo de componentes define os padrões e convenções impostas aos componentes do sistema, de modo a descrever a função de cada um e como eles interagem entre si. O conjunto de convenções que devem ser definidas pelo modelo de componentes são apresentadas na Tabela 2.1 [18].

Item	**Descrição**
Interfaces	Especificação do comportamento e das propriedades
Metadados	Informações sobre os componentes, interfaces e seus relacionamentos
Interoperabilidade	Comunicação e troca de dados entre componentes de diferentes origens, implementados em diferentes linguagens
Personalização	Interfaces que possibilitam a personalização dos componentes
Composição	Interfaces e regras para combinar componentes no desenvolvimento de aplicações e para substituir e adicionar componentes as aplicações já existentes
Suporte a evolução	Regras e serviços para substituir componentes ou interfaces por versões mais recentes
Empacotamento e utilização	Empacotar implementações e recursos necessários para instalar e configurar componentes

Tabela 2.1: Convenções definidas por um *modelo de componentes*.

Os conceitos de modelo de componentes e arcabouço de componentes são complementares e fortemente acoplados. O arcabouço de componentes implementa o modelo de componentes respeitando e regulando as definições estabelecidas. Segundo Bachmann [5], as funções do **arcabouço de componentes** são: fornecer suporte ao desenvolvimento de componentes e à composição de aplicações; gerenciar os recursos compartilhados pelos componentes; e prover um mecanismo que possibilite a comunicação entre os componentes. Com o arcabouço de componentes, os desenvolvedores de aplicações baseadas em componentes não precisam se preocupar com a implementação de serviços complexos como troca de mensagens, passagem de dados e ligação dos componentes. As relações entre componentes, modelo de componentes e arcabouço de componentes são apresentadas na Figura 2.1.

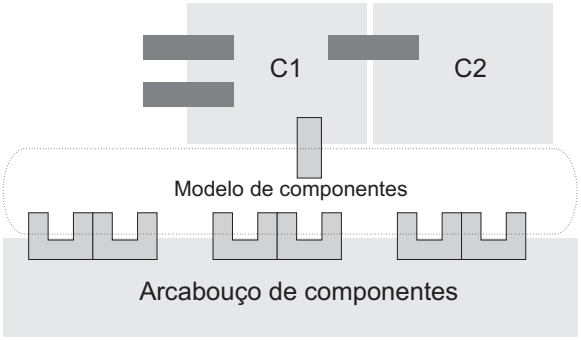

Figura 2.1: Relacionamento entre componentes, modelo de componentes e arcabouço de componentes.

Os componentes são representados por C1 e C2. O modelo de componentes é o intermediário entre os componentes propriamente ditos e o arcabouço de componentes, definindo os tipos de componentes, as formas de interação e os recursos necessários. Por fim, o arcabouço de componentes implementa a infra-estrutura de suporte a comunicação e composição dos componentes.

2.1.3 Relação entre DBC e o trabalho proposto

O foco deste trabalho em relação à DBC é a implementação de um arcabouço de componentes C++ denominado CCF [32], descrito no Capítulo 4. Os padrões e convenções que definem as regras de composição e interação de componentes utilizadas na implementação do CCF estão especificados no modelo de componentes COMPOR, apresentado no Capítulo 3.

2.2 Evolução Dinâmica de Software Não Antecipada

A evolução é a fase de desenvolvimento mais cara de um software em termos financeiros e em termos de tempo [7]. O RISE (*Research Institute for Software Evolution*) [15] define a evolução de software como o conjunto de atividades técnicas e gerenciais que visam garantir que o software continue a atingir seus objetivos de negócio, e da organização para qual foi desenvolvido, de uma maneira viável em termos de custo. A evolução de software é classificada por Lientz e Swanson [24] em quatro tipos:

- **Corretiva:** evolução em virtude de problemas encontrados no software.
- **Adaptativa:** evolução em virtude da adaptação do software a um novo ambiente ou plataforma.
- **Aperfeiçoativa:** evolução em virtude de requisições de melhorias por parte do usuário.
- **Preventiva:** evolução em virtude de futuros problemas que podem vir a acontecer.

2.2.1 Evolução Dinâmica de Software

A Evolução Dinâmica de Software é um tipo de evolução que ocorre sem a interrupção da execução do software. A principal motivação para se desenvolver um software com suporte à evolução dinâmica são as aplicações que precisam se manter em constante evolução em tempo de execução, ou seja, que não podem ter a execução interrompida por razões de segurança ou financeiras. Exemplos bem conhecidos de tais sistemas são serviços Web, sistemas de telecomunicação, sistemas bancários, sistemas de controle de tráfego aéreo e sistemas militares [12].

No caso de sistemas baseados em componentes, as interfaces bem definidas e a independência dos componentes do sistema torna menos complexa a gerência de evolução dinâmica de software.

2.2.2 Evolução de Software Não Antecipada

A Evolução de Software Não Antecipada é um tipo de evolução que não depende de pontos de extensão previamente definidos no software [21]. Os pontos de extensão são ganchos implementados no código fonte da aplicação a fim de facilitar a evolução com base na antecipação de cenários. Porém, prever tais cenários de evolução não é uma tarefa simples. Além disso, quanto mais cenários de evolução são previstos e considerados no projeto, mais complexa se torna a solução inicial [21]. A maioria das complicações

técnicas e gastos relacionados à evolução de software ocorrem quando as mudanças não foram previstas no projeto inicial.

No contexto de desenvolvimento baseado em componentes, mudanças não antecipadas estão relacionadas à alteração de componentes existentes, inserção de novos componentes, novos serviços não previstos e novos módulos na arquitetura.

2.2.3 Evolução Dinâmica de Software Não Antecipada

A Evolução Dinâmica de Software Não Antecipada (EDSNA) é um tipo de evolução que ocorre sem a interrupção da execução do software e que não depende de pontos de extensão previamente definidos. Um dos principais problemas ao implementar uma infra-estrutura com suporte à EDSNA é deixar o processo de evolução transparente para o desenvolvedor final. Nos trabalhos relacionados apresentados posteriormente, o desenvolvedor precisa gerenciar o processo de evolução além de implementar a regra de negócio da aplicação.

2.2.4 Relação entre EDSNA e o trabalho proposto

O foco deste trabalho em relação à EDSNA é fornecer um ambiente C++ para o desenvolvimento de software com suporte à evolução dinâmica não antecipada. O ambiente C++ é composto por um arcabouço de componentes C++ denominado CCF [32] e por um servidor de aplicação C++ denominado CCAS [31]. O CCF e o CCAS são apresentados nos Capítulos 4 e 5, respectivamente.

2.3 Trabalhos relacionados

Nesta subseção, são apresentadas abordagens de ambientes C++ para o desenvolvimento de software com suporte à evolução dinâmica não antecipada.

2.3.1 *Balboa*

O ambiente de composição Balboa [11] é formado por três partes (Figura 2.2): uma linguagem de integração de componentes (CIL); um conjunto de bibliotecas de componentes C++; e um conjunto de interfaces (SLIs) para integrar as duas partes anteriores. O Balboa usa uma linguagem interpretada própria (CIL) tanto para instanciar quanto para conectar os componentes. Depois disso, um componente C++ pode ser manipulado através da interface SLI, a qual é implementada usando a linguagem de definição de interface do Balboa (BIDL), ou interagir diretamente com outros componentes C++. Para implementar a evolução dinâmica, o Balboa utiliza três linguagens diferentes: CIL, C++ e

BIDL. Além disso, o desempenho do Balboa é impactado quando o componente C++ é manipulado através da interface SLI, pois a linguagem BIDL é interpretada.

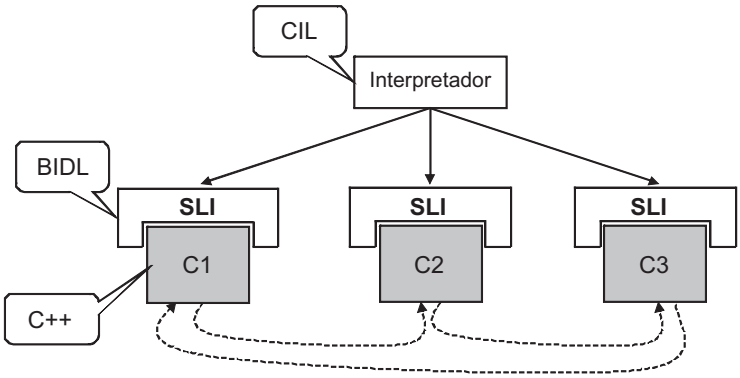

Figura 2.2: Arquitetura do *Balboa*.

2.3.2 *Accord*

O arcabouço de programação para aplicações autônomas chamado *Accord* [25] permite a composição de elementos C++ em tempo de execução através de definição de regras. A arquitetura do *Accord* é apresentada na Figura 2.3. O gerenciador de composição injeta as regras de interação definidas pelo usuário no gerenciador do elemento C++ o qual, então, executa as regras para configurar o elemento e para estabelecer as ligações com outros elementos. Embora o Accord permita a troca de elementos C++ em tempo de execução, os elementos, enquanto estão no processo de substituição, não respondem as requisições nem as armazenam para repassar aos novos elementos. Além disso, composição recursiva não é possível no Accord.

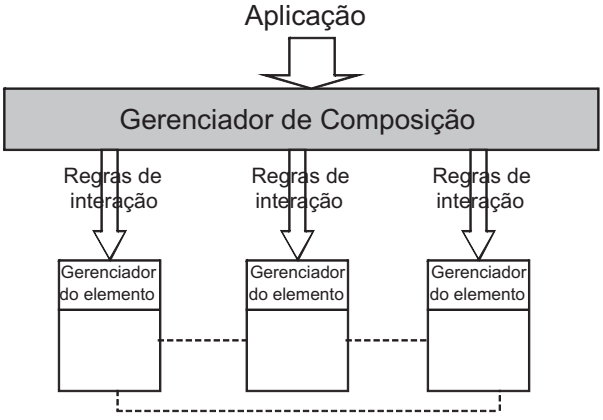

Figura 2.3: Arquitetura do *Accord*.

2.3.3 *A Framework for Live Software Upgrade*

O arcabouço proposto em [38] utiliza a idéia de *proxy* para eliminar as referências entre os módulos de implementação C++. A arquitetura do arcabouço é apresentada na Figura 2.4. O arcabouço é dividido em duas partes. A primeira parte é o serviço de configuração dinâmica, o qual inclui um módulo *proxy*. Os módulos dinâmicos, ou seja, os módulos de implementação, constituem a segunda parte. Os módulos de implementação são os componentes C++, os quais podem ser carregados, descarregados e até mesmo substituídos em tempo de execução. O módulo *proxy* é utilizado como uma porta, encaminhando todas as requisições que chegam para o módulo de implementação correto. Embora os módulos de implementação possam ser substituídos, o processo de substituição só é realizado quando o módulo não tiver requisições pendentes. Além disso, a evolução dinâmica só acontece nos módulos de implementação.

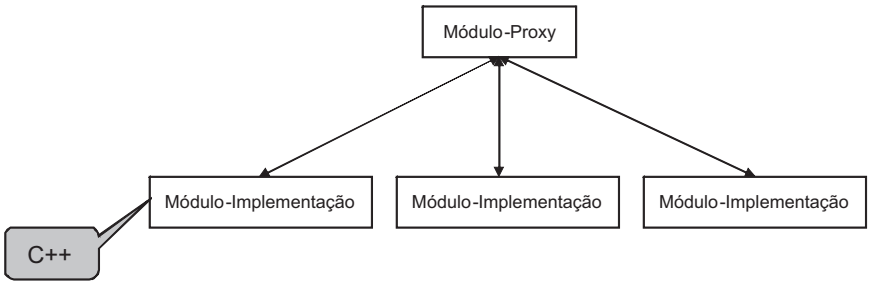

Figura 2.4: Arquitetura do *Framework for Live Software Upgrade*.

2.3.4 Discussão sobre os trabalhos relacionados

O ambiente C++ apresentado neste documento fornece um mecanismo para o desenvolvimento de software com suporte à evolução dinâmica não antecipada. No ambiente proposto, cada aplicação é uma estrutura hierárquica cujo projeto é baseado no padrão Composite [17]. Dessa maneira, as aplicações podem compor elementos complexos a partir de elementos simples, através de composição recursiva [17]. Os elementos podem ser habilitados, desabilitados, carregados, descarregados e até substituídos em tempo de execução. O processo de substituição ocorre instantaneamente, ou seja, o elemento é substituído de imediato. Após a substituição, as novas requisições passam a ser encaminhadas para o novo elemento. Caso haja alguma tarefa em curso no elemento substituído, ela é terminada antes do mesmo ser destruído. Qualquer elemento da hierarquia pode ser substituído. Para implementar as aplicações, o desenvolvedor precisa conhecer apenas a linguagem de programação C++ que é largamente utilizada por desenvolvedores no mundo inteiro facilitando a utilização do ambiente C++ proposto neste documento.

Capítulo 3

COMPOR Component Model Specification

A Especificação do Modelo de Componentes COMPOR (CMS) [4] fornece mecanismos para o desenvolvimento de software baseado em componentes com suporte à evolução dinâmica não antecipada. A idéia da CMS é desacoplar os componentes que provêem as funcionalidades do sistema. A inexistência de referências entre os componentes provedores de funcionalidades facilita a inserção, remoção e atualização dos componentes em tempo de execução. Assim, o impacto da evolução de software sobre uma aplicação desenvolvida utilizando a CMS é menor.

Uma aplicação baseada na CMS possui uma estrutura hierárquica em forma de árvore, onde os componentes provedores de funcionalidades são as folhas da árvore e os componentes estruturais são os galhos da árvore. As folhas da árvore são denominadas **componentes funcionais** e os galhos da árvore são denominados **contêineres**. Um exemplo de uma hierarquia CMS apresentado na Figura 3.1, onde Ct1 e Ct2 são contêineres e X, Y e Z são componentes funcionais. Os componentes funcionais fornecem, através de serviços e eventos, as funcionalidades do sistema e não possuem componentes-filhos. Os contêineres armazenam e gerenciam os serviços providos e os eventos de interesse dos componentes-filhos [4].

Além dos componentes funcionais e dos contêineres, a hierarquia de uma aplicação CMS pode conter dois tipos de entidades: os **componentes personalizáveis** e os **adaptadores**. Os componentes personalizáveis são tipos especiais de componentes funcionais que além de fornecerem as funcionalidades do sistema podem ser modificados através de comportamentos. Os adaptadores são tipos especiais de contêineres que são utilizados para modificar os serviços providos ou os eventos de interesse de componentes funcionais ou de componentes personalizáveis.

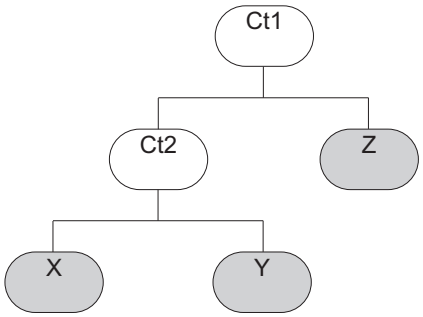

Figura 3.1: Exemplo de uma hierarquia CMS.

3.1 Disponibilização de componentes

Os componentes funcionais ficam disponíveis após a inserção nos contêineres. Portanto, pelo menos um contêiner deve existir: o contêiner raiz. Os contêineres armazenam duas tabelas, uma para os serviços providos e outra para os eventos de interesse dos seus componentes-filhos. Quando um componente é inserido em um contêiner, as tabelas de *serviços providos* e *eventos de interesse* de cada contêiner são atualizadas até a raiz da hierarquia. Na Figura 3.2, ilustra-se o processo de disponibilização de componentes e os seus passos são detalhados a seguir.

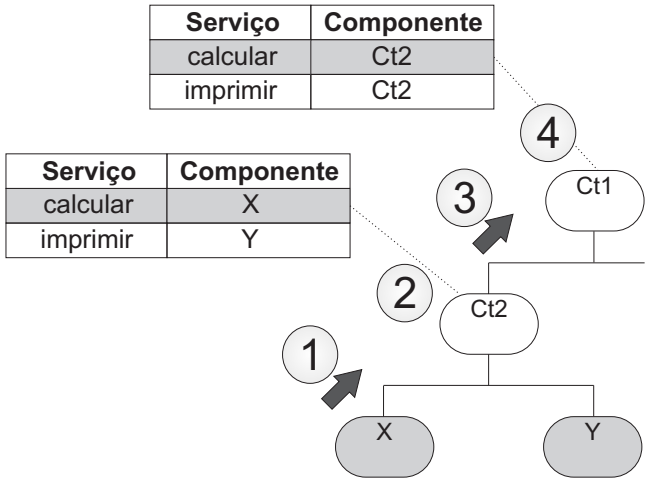

Figura 3.2: Disponibilização de componentes.

1. O componente X que implementa o serviço "calcular" é inserido no contêiner 2.
2. O contêiner 2 atualiza a sua tabela de serviços redefinindo os serviços providos por seus componentes-filhos.
3. O contêiner 2 solicita a seu contêiner-pai a atualização da tabela de serviços providos.
4. O contêiner 1 atualiza a sua tabela de serviços redefinindo os serviços providos por seus componentes-filhos.

Após a disponibilização do componente "X", qualquer componente da hierarquia pode acessar os seus serviços como também notificá-lo quando algum evento de seu interesse ocorrer, mesmo sem haver uma referência explícita para "X". Observe que o componente tem a referência apenas para o seu contêiner-pai.

3.2 Atualização de componentes

A atualização de componentes em uma aplicação CMS consiste em inserir o novo componente na hierarquia e remover o componente antigo. A inserção é realizada antes da remoção para garantir que os serviços providos e eventos de interesse do componente antigo estejam sempre disponíveis. Se a remoção for realizada antes da inserção, os serviços providos e os eventos de interesse do componente antigo podem ficar indisponíveis no intervalo entre a remoção e a inserção. Na Figura 3.3, ilustra-se o processo de atualização de componentes e os seus passos são detalhados a seguir.

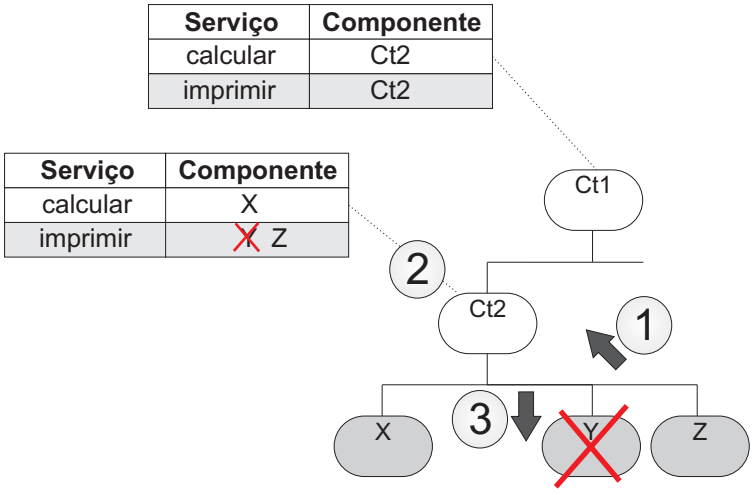

Figura 3.3: Atualização de componentes.

1. O componente Z que implementa o serviço "imprimir" é inserido no contêiner 2.

2. O contêiner 2 atualiza a sua tabela de serviços. O provedor do serviço "imprimir" deixa de ser o componente Y e passa a ser o componente Z. A partir daí, as requisições do serviço "imprimir" são encaminhadas para o componente Z e não mais para o componente Y.

3. O componente Y é removido do contêiner 2. O componente Y só é destivado quando todas as requisições pendentes forem finalizadas.

Vale ressaltar que apenas a tabela de *serviços providos* do contêiner 2 foi atualizada. O restante da hierarquia permanece da mesma forma.

3.3 Personalização de componentes

A personalização de componentes ocorre em um tipo especial de componente funcional: o componente personalizável. O componente personalizável é modificado através de comportamentos que são armazenados nos gerenciadores de comportamentos. Além de implementar os serviços providos e os eventos de interesse, o componente personalizável disponibiliza comportamentos que podem ser ativados ou desativados em tempo de execução. Os comportamentos geralmente são requisitos transversais como *log*, persistência e tratamento de exceção.

O gerenciador de comportamentos tem um papel fundamental na execução de um componente personalizável. Ao receber a requisição de um serviço, o componente personalizável verifica junto ao seu gerenciador de comportamentos quais os comportamentos devem ser invocados, ou seja, os comportamentos que estão ativados. Os comportamentos ativados são invocados antes e depois da execução do serviço. Caso ocorra uma exceção, os comportamentos ativados também são invocados. Na Figura 3.4, ilustra-se uma hierarquia CMS onde o componente K é um componente personalizável.

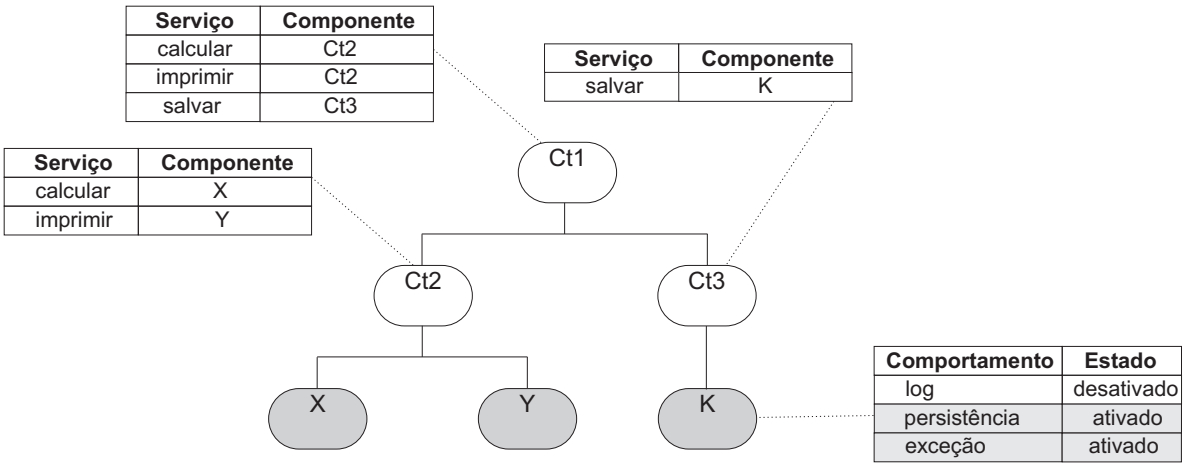

Figura 3.4: Personalização de componentes.

Além de implementar o serviço "salvar", o componente personalizável K disponibiliza os comportamentos *log*, *persistência* e *exceção*. Quando o serviço "salvar" do componente personalizável K é requisitado, os comportamentos *persistência* e *exceção* são invocados antes e depois da execução do serviço "salvar". Caso ocorra uma exceção, os comportamentos *persistência* e *exceção* também são invocados. Por estar desativado, o comportamento *log* não é invocado. É importante lembrar que os componentes personalizáveis têm as mesmas características dos componentes funcionais em termos de interação entre componentes e de composição da hierarquia.

3.4 Adaptação de componentes

Um componente CMS, seja ele funcional ou personalizável, é adaptado através de um tipo especial de contêiner: o adaptador. Os parâmetros, o retorno e as exceções das interfaces que implementam serviços ou eventos do componente podem ser modificadas pelo adaptador. O adaptador se coloca entre o componente adaptado e o seu contêiner-pai podendo modificar apenas alguns serviços ou eventos do componente adaptado. Os serviços ou eventos não modificados são repassados para o componente adaptado. Na Figura 3.5, ilustra-se a adaptação do serviço "salvar" do componente K e os seus passos são detalhados a seguir.

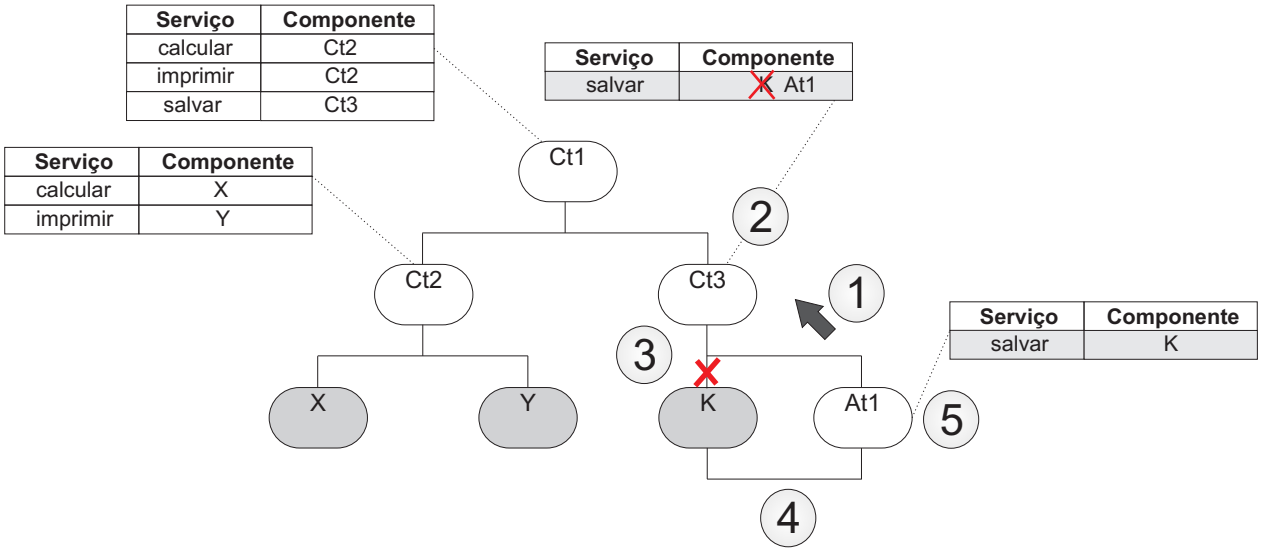

Figura 3.5: Adaptação de componentes.

1. O adaptador 1 que modifica o serviço "salvar" é inserido no contêiner 3.
2. O contêiner 3 atualiza a sua tabela de serviços. O provedor do serviço "salvar" deixa de ser o componente K e passa a ser o adaptador 1. A partir daí, as requisições do serviço "salvar" são encaminhadas para o adaptador 1 e não mais para o componente K.
3. O componente K é removido do contêiner 3.
4. O componente K é adicionado ao adaptador 1.
5. O adaptador 1 atualiza a sua tabela se serviços. Assim, o serviço adaptado "salvar" pode invocar o serviço "salvar" original do componente K.

Um adaptador pode ser visto como um contêiner que possui apenas um componente, funcional ou personalizável, como entidade filha.

3.5 Mecanismos de interação

A interação entre componentes pode ocorrer através de serviços ou de eventos. Na interação baseada em serviços, um componente funcional invoca qualquer serviço provido por outro componente da hierarquia, mesmo que ele pertença a outro contêiner. Já a interação baseada em eventos anuncia mudanças em algum componente funcional para todos os interessados. Em ambos os casos não existem referências entre os componentes.

3.5.1 Interação baseada em serviços

Quando ocorre a requisição de um serviço, o provedor do serviço deve ser encontrado. Assim, a requisição do serviço "salvar" pelo componente "X" inicia uma busca na hierarquia pelo provedor do serviço, componente "K". Este processo é apresentado na Figura 3.6 e os passos são descritos a seguir.

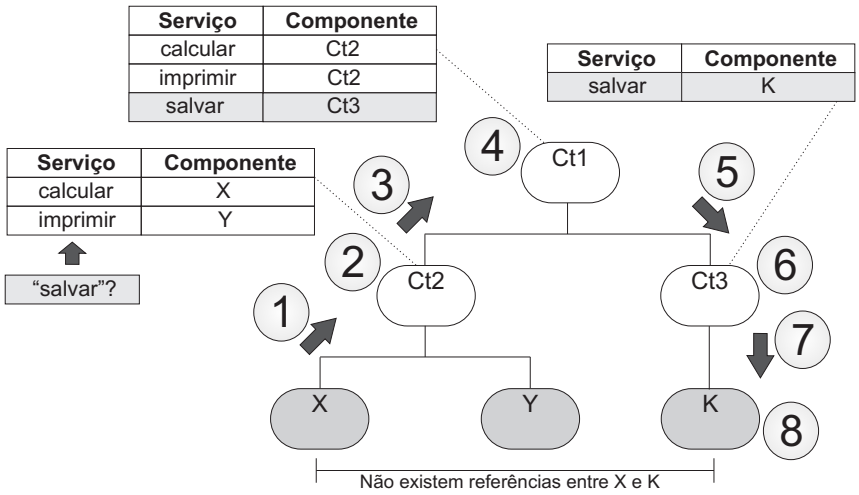

Figura 3.6: Interação baseada em serviços.

1. O componente X solicita a execução do serviço "salvar" ao seu "contêiner-pai".
2. O Contêiner 2 verifica, de acordo com sua tabela de serviços, que nenhum dos seus "componentes-filhos" implementa o serviço "salvar".
3. O contêiner 2 encaminha a solicitação para o seu "contêiner-pai".
4. O Contêiner 1 verifica, de acordo com sua tabela de serviços, que um de seus "componentes-filhos" implementa o serviço "salvar" (Contêiner 3). Para o Contêiner 1, o Contêiner 3 é o componente provedor do serviço.
5. O contêiner 1 encaminha a solicitação para o contêiner 3.

6. O Contêiner 3 não implementa o serviço, mas possui a referência para o real provedor do serviço - componente "K".

7. O Contêiner 3 encaminha a solicitação para o componente "K".

8. O componente "K" executa o serviço "salvar" e retorna o resultado.

É importante observar que não existe referência entre o componente que solicita o serviço ("X") e o componente que o provê ("K"). Assim, o componente que implementa o serviço "salvar" pode ser substituído sem modificar o restante da estrutura.

3.5.2 Interação baseada em eventos

Quando um evento é anunciado por um componente funcional, todos os componentes da hierarquia que têm interesse no evento devem ser notificados. A interação baseada em eventos também é implementada pelos contêineres, dessa maneira não são necessárias referências entre os componentes funcionais. O processo de anúncio de evento é ilustrado na Figura 3.7 e os seus passos são detalhados a seguir.

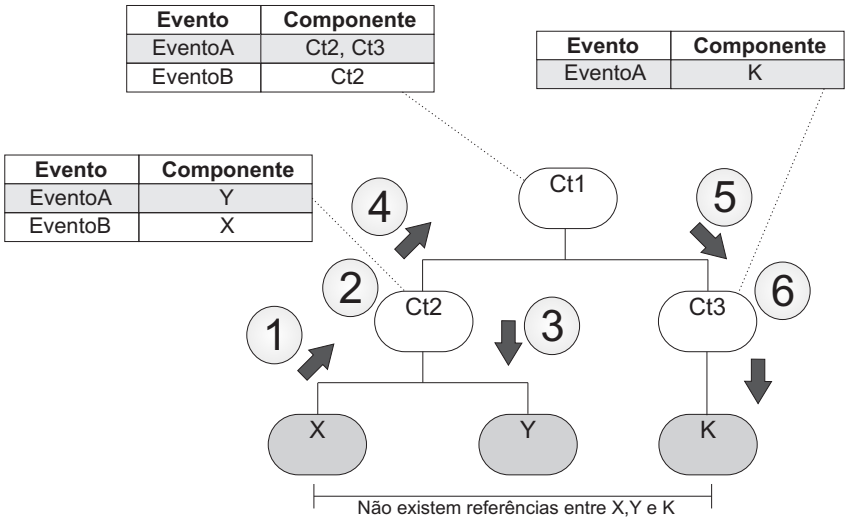

Figura 3.7: Interação baseada em eventos.

1. O componente "X" anuncia o evento "A".

2. O anúncio é recebido pelo seu "contêiner-pai" (Contêiner 2), que verifica em sua tabela de evento se algum de seus "componentes-filhos" tem interesse no evento.

3. O contêiner 2 encaminha o evento para os componentes interessados, nesse caso apenas o componente "Y".

4. O contêiner então repassa o evento para o seu "contêiner-pai" (Contêiner 1).

5. O contêiner 1, de acordo com sua tabela de eventos, encaminha o evento para todos os interessados, exceto para o que anunciou o evento (Contêiner 2). Como o Contêiner 1 é a raiz da hierarquia, não existe "contêiner-pai" para repassar o anúncio. Assim, o evento é encaminhado apenas para o Contêiner 3.

6. O Contêiner 3, de acordo com sua tabela de eventos, encaminha o evento para o Componente "K".

A CMS define as regras de interação e composição de componentes que devem ser seguidas para que o software possa evoluir de forma dinâmica e não antecipada. O software desenvolvido seguindo a especificação CMS permite a adição, remoção, atualização, personalização e adaptação de componentes sem nenhum tipo de planejamento prévio inclusive em tempo de execução. Os mecanismos de interação baseados em serviços e eventos especificados na CMS buscam, através do desacoplamento dos componentes provedores de funcionalidades, facilitar o processo de evolução de software. No próximo capítulo, apresenta-se o arcabouço que implementa a especificação CMS em C++.

Capítulo 4

C++ Component Framework

No capítulo anterior foi apresentada a CMS. Neste capítulo, descreve-se a implementação em C++ da CMS, *C++ Component Framework* (CCF). O CCF é um arcabouço para o desenvolvimento de software baseado em componentes com suporte à evolução dinâmica não antecipada. As aplicações desenvolvidas utilizando o CCF permitem inserir, remover e atualizar componentes em tempo de execução. Além disso, as aplicações CCF não precisam de pontos de extensão no código, ou seja, a evolução pode ocorrer em qualquer parte e a qualquer momento.

4.1 Projeto do CCF

Nesta seção, as classes do CCF e os seus relacionamentos são apresentados utilizando a notação de diagrama de classes da linguagem UML [16]. Os estereótipos utilizados para representar os tipos e estruturas de dados são apresentados na Tabela 4.1. No diagrama de classes da Figura 4.1, apresenta-se uma visão geral do projeto do CCF. Nas subseções a seguir, cada parte do projeto do CCF é detalhada.

Estereótipo	**Descrição**
string	Cadeia de caracteres
vector	Vetor de elementos
map	Mapeamento chave-valor
void*	Ponteiro genérico, ou seja, para qualquer tipo de objeto
member	Método referenciado em tempo de execução através de reflexão
list	Lista de elementos
set	Conjunto de elementos sem repetição
exception	Exceções

Tabela 4.1: Estereótipos referentes a tipos e estruturas de dados do CCF.

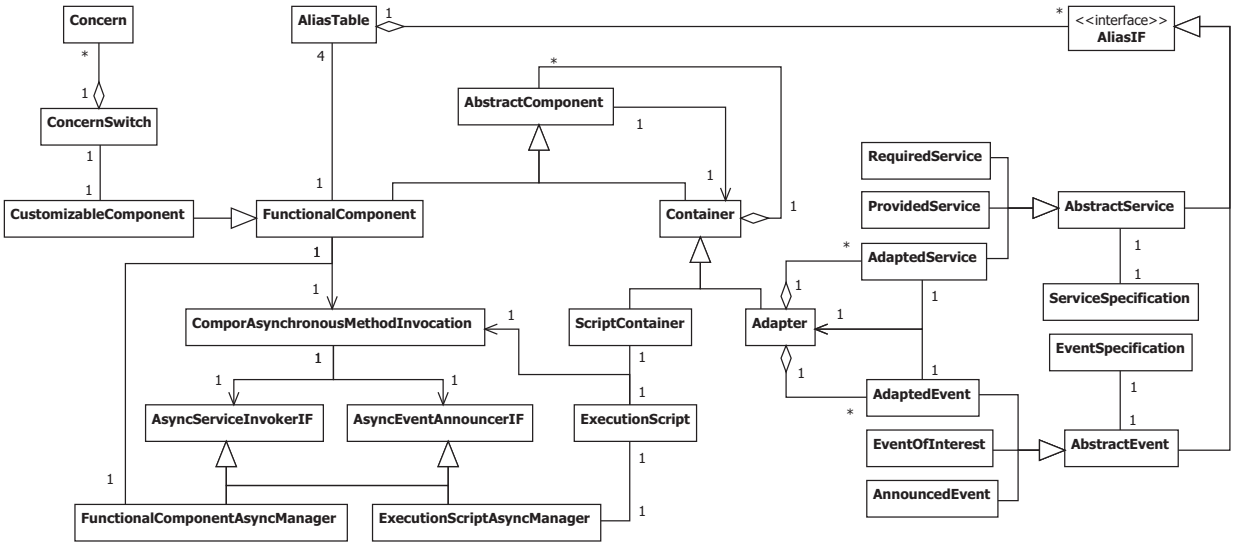

Figura 4.1: Visão geral do projeto do CCF.

4.1.1 Estrutura de componentes

O núcleo da estrutura de componentes do CCF [32] é baseado no padrão de projeto Composite [17], o qual é utilizado para compor objetos em estruturas de árvore a fim de representar hierarquias parte-todo. As classes `AbstractComponent`, `FunctionalComponent` e `Container`, detalhadas no diagrama da Figura 4.2, representam o núcleo da estrutura de componentes do CCF. Qualquer componente da hierarquia CCF possui algum tipo de relação com estas classes. O elo entre a estrutura de componentes e os mecanismos de interação do CCF são implementados pelas classes `AliasTable` e `AliasIF`.

A classe abstrata `AbstractComponent` permite a composição de elementos complexos a partir de elementos simples, através de composição recursiva [17]. Os métodos abstratos da classe `AbstractComponent` são implementados de maneira diferente pelas subclasses `FunctionalComponent` e `Container`. `FunctionalComponent` é superclasse dos componentes funcionais. Os componentes funcionais possuem tabelas (`AliasTable`) de serviços providos, serviços requeridos, eventos de interesse e eventos anunciados. Os elementos dessas tabelas implementam a interface (`AliasIF`). A classe `Container` representa os contêineres da hierarquia. Os contêineres podem armazenar tanto componentes funcionais quanto outros contêineres, ou seja, podem compor elementos complexos a partir de elementos simples.

1. A classe `AbstractComponent`

   A classe `AbstractComponent` define a interface que deve ser implementada por um componente da hierarquia CCF. O atributo `m_container` da classe `AbstractComponent` armazena, caso exista, uma referência para o contêiner-pai. Os métodos `getContainer` e `setContainer` estão relacionados à atualização e recuperação do

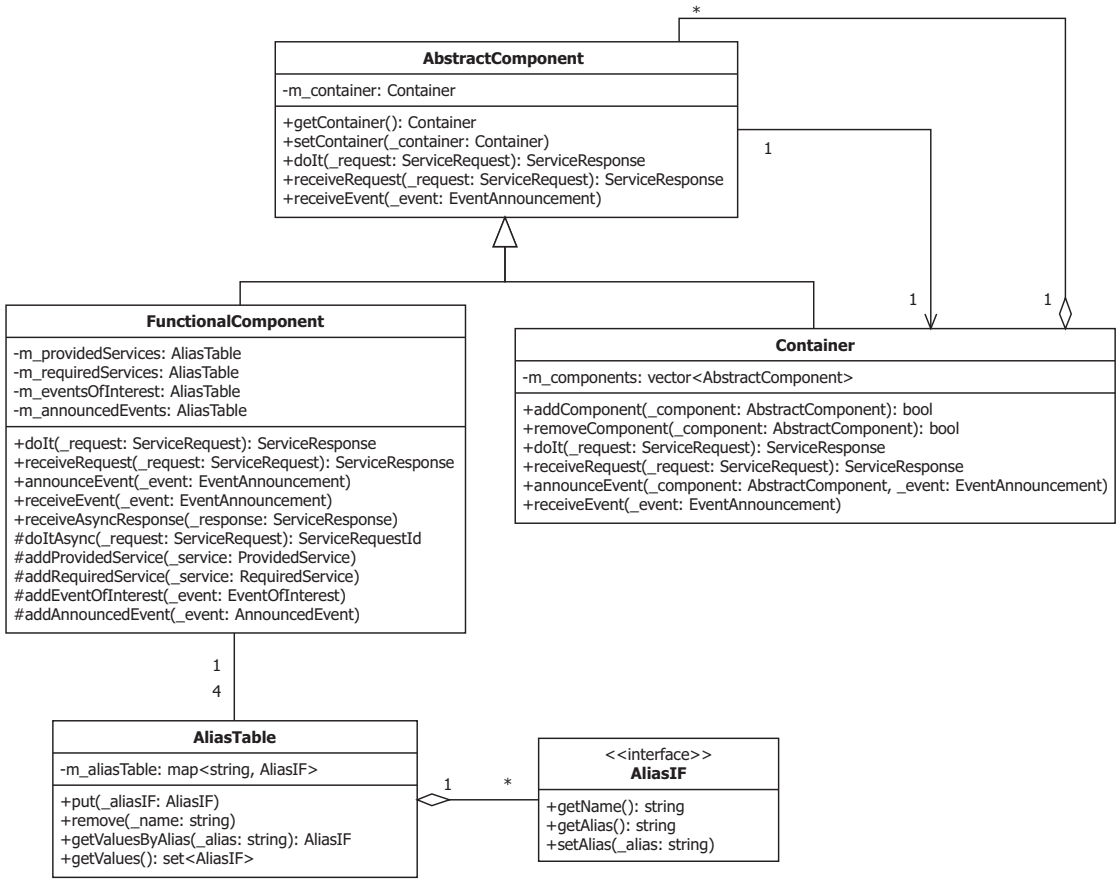

Figura 4.2: Núcleo da estrutura de componentes do CCF.

atributo `m_container`. Os outros métodos são detalhados a seguir.

`doIt(_request: ServiceRequest): ServiceResponse`

Método abstrato. Invoca um serviço, onde `_request` é uma instância da classe `ServiceRequest` que encapsula a requisição do serviço. O retorno é a resposta do serviço encapsulada em uma instância da classe `ServiceResponse`.

`receiveRequest(_request: ServiceRequest): ServiceResponse`

Método abstrato. Recebe uma requisição de serviço, onde `_request` é uma instância da classe `ServiceRequest` que encapsula a requisição do serviço. O retorno é a resposta do serviço encapsulada em uma instância da classe `ServiceResponse`.

`receiveEvent(_event: EventAnnouncement)`

Método abstrato. Recebe um anúncio de evento, onde `_event` é uma instância da classe `EventAnnouncement` que encapsula o anúncio do evento.

2. A classe `FunctionalComponent`

Os atributos da classe `FunctionalComponent` são: `m_providedServices`, tabela `AliasTable` que armazena os serviços providos; `m_requiredServices`, tabela `AliasTable` que armazena os serviços requeridos; `m_eventsOfInterest`, tabela `AliasTable` que armazena os eventos de interesse; `m_announcedEvents`, tabela `AliasTable` que armazena os eventos anunciados; Os elementos das tabelas `AliasTable` citadas implementam a interface `AliasIF`. Os métodos `addProvidedService`, `addRequiredService`, `addEventOfInterest` e `addAnnouncedEvent` adicionam serviços providos, serviços requeridos, eventos de interesse e eventos anunciados, respectivamente. Os outros métodos são detalhados a seguir.

`doIt(_request: ServiceRequest): ServiceResponse`

Implementa o método abstrato definido na classe `AbstractComponent`. Na classe `FunctionalComponent`, o método `doIt` repassa a requisição do serviço para o contêiner-pai através do método `doIt`.

`receiveRequest(_request: ServiceRequest): ServiceResponse`

Implementa o método abstrato definido na classe `AbstractComponent`. Na classe `FunctionalComponent`, o método `receiveRequest` recupera o método do componente que implementa o serviço requisitado e o invoca utilizando mecanismos de reflexão computacional.

`announceEvent(_event: EventAnnouncement)`

Anuncia um evento, onde `_event` é o anúncio do evento. O método `announceEvent` repassa o anúncio do evento para o contêiner-pai através do método `announceEvent`. O anúncio de um evento ocorre de maneira assíncrona.

`receiveEvent(_event: EventAnnouncement)`

Implementa o método abstrato definido na classe `AbstractComponent`. Na classe `FunctionalComponent`, o método `receiveEvent` recupera o método do componente que trata o evento anunciado e o invoca utilizando mecanismos de reflexão computacional.

`receiveAsyncResponse(_response: ServiceResponse)`

Recebe a resposta de uma invocação de serviço assíncrona, onde `_response` é uma instância da classe `ServiceResponse` que encapsula a resposta do serviço. Esse método tem implementação vazia na classe `FunctionalComponent`. As classes que herdam de `FunctionalComponent` devem implementar esse método caso tenham interesse em receber as respostas das requisições de serviço assíncronas.

`doItAsync(_request: ServiceRequest): ServiceRequestId`

Invoca um serviço de maneira assíncrona, onde `_request` é uma instância da classe `ServiceRequest` que encapsula a requisição do serviço. O retorno é o identificador da requisição encapsulado em uma instância da classe `ServiceRequestId`.

2.1. A classe `AliasTable`

O atributo da classe `AliasTable` é uma tabela cujo os índices dos elementos `AliasIF` são os seus apelidos. Os métodos `put`, `remove`, `getValuesByAlias`, `getValues` estão relacionados à recuperação e à atualização dos valores armazenados na tabela.

2.2. A classe `AliasIF`

A classe `AliasIF` deve ser estendida pelas classes que implementam os mecanismos de interação do CCF. Os métodos `getName`, `getAlias` e `setAlias` estão relacionados à recuperação e atualização do nome e do apelido do elemento. As classes `ProvidedService`, `RequiredService`, `EventOfInterest`, `AnnouncedEvent` implementam a interface `AliasIF` e serão detalhadas posteriormente.

3. A classe `Container`

O atributo `m_components` da classe `Container` é uma coleção de componentes-filhos `AbstractComponent`. O método `addComponent` adiciona e o método `removeComponent` remove um componente-filho da coleção. Os outros métodos são detalhados a seguir.

`doIt(_request: ServiceRequest): ServiceResponse`

Implementa o método abstrato definido na classe `AbstractComponent`. Na classe `Container`, o método `doIt` verifica se algum dos componentes-filhos provê o serviço requisitado. Se for encontrado, o método `receiveRequest` do componente provedor do serviço é invocado. Se não for encontrado, o método `doIt` do contêiner-pai é invocado.

`receiveRequest(_request: ServiceRequest): ServiceResponse`

Implementa o método abstrato definido na classe `AbstractComponent`. Na classe `Container`, o método `receiveRequest` encaminha a requisição para o componente-filho que provê o serviço através do método `receiveRequest`.

`announceEvent(_component: AbstractComponent, _event: EventAnnouncement)`

Anuncia um evento, onde `_component` é o componente que anuncia o evento e `_event` é o anúncio do evento. O método `announceEvent` verifica se existe algum componente-filho interessado no evento. Se for encontrado, o anúncio do evento é repassado através do método `receiveEvent`. O método `announceEvent` do contêiner-pai também é invocado para repassar o anúncio aos outros componentes interessados da hierarquia.

`receiveEvent(_event: EventAnnouncement)`

Implementa o método abstrato definido na classe `AbstractComponent`. Na classe `Container`, o método `receiveEvent` repassa o anúncio para o componente-filho interessado no evento através do método `receiveEvent`.

Componentes personalizáveis

Os componentes personalizáveis são componentes que podem ser modificados através de comportamentos. `CustomizableComponent` é a superclasse dos componentes personalizáveis do CCF. Os comportamentos podem ser ativados ou desativados através do gerenciador de comportamentos (`ConcernSwitch`). A classe `Concern` deve ser estendida pelas classes que implementam comportamentos do CCF. Os comportamentos são invocados antes e após a execução de um serviço implementado por um componente personalizável. As classes que compõem a estrutura de componentes personalizáveis do CCF são detalhadas no diagrama da Figura 4.3.

1. A classe `CustomizableComponent`

   O atributo `m_concernSwitch` da classe `CustomizableComponent` é uma instância de `ConcernSwitch` que é responsável por gerenciar os comportamentos do componente personalizável. O método `addConcern` adiciona um comportamento no componente personalizável indicando o estado, ativado ou desativado. O método `receiveRequest` é detalhado a seguir.

   `receiveRequest(_request: ServiceRequest): ServiceResponse`

   Sobrescreve o método implementado na classe `FunctionalComponent`. Na classe `CustomizableComponent`, o método `receiveRequest` executa os métodos `executeBefore` e `executeAfter` da classe `ConcernSwitch` antes e depois de invocar o método `doIt` da classe `FunctionalComponent`. Se alguma exceção for lançada pelo método `doIt` da classe `FunctionalComponent`, o método `executeOnException` da classe `ConcernSwitch` é invocado.

   1.1. A classe `ConcernSwitch`

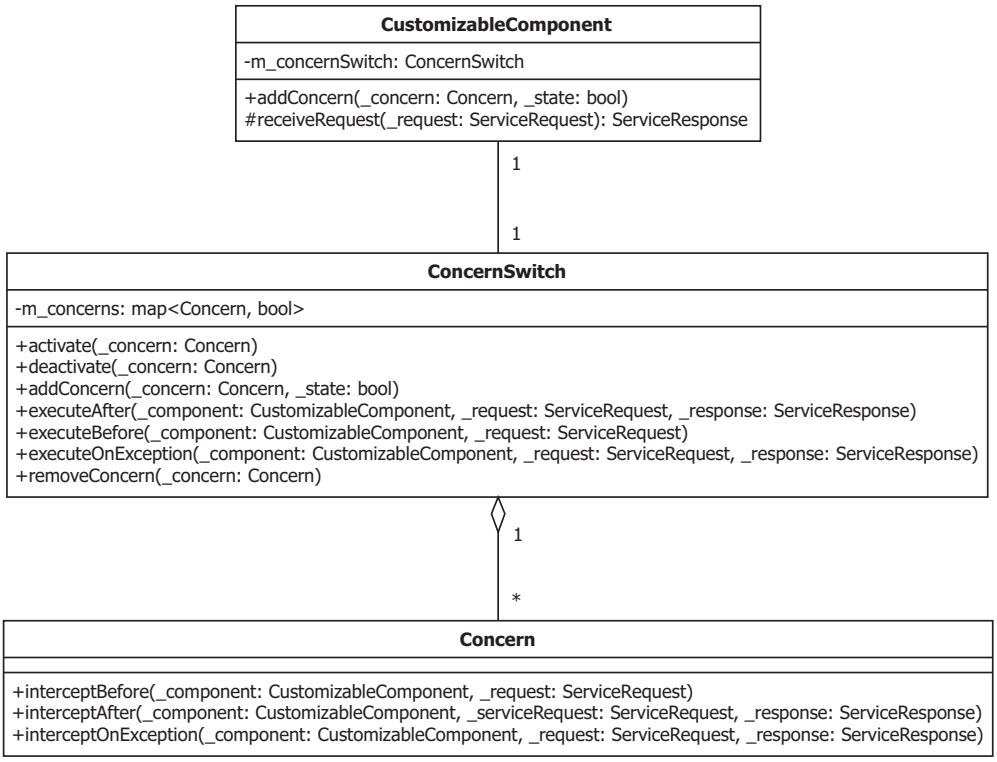

Figura 4.3: Estrutura dos componentes personalizáveis do CCF.

O atributo `m_concerns` da classe `ConcernSwitch` é uma tabela que armazena o comportamento e seu estado, ativado ou desativado. Os métodos `activate`, `deactivate`, `addConcern` e `removeConcern` estão relacionados à atualização da tabela de comportamentos. Os detalhes dos outros métodos são apresentados a seguir.

`executeAfter(_component: CustomizableComponent, _request: ServiceRequest, _response: ServiceResponse)`
: Invoca o método `interceptAfter` de todos os comportamentos (classe `Concern`) ativos da tabela após a execução do serviço, onde `_component` é o componente personalizável que implementa o serviço, `_request` é a requisição do serviço e `_response` é a resposta do serviço.

`executeBefore(_component: CustomizableComponent, _request: ServiceRequest)`
: Invoca o método `interceptBefore` de todos os comportamentos (classe `Concern`) ativos da tabela antes da execução do serviço, onde `_component` é o componente personalizável que implementa o serviço e `_request` é a requisição do serviço.

`executeOnException(_component: CustomizableComponent, _request: ServiceRequest, _response: ServiceResponse)`

Invoca o método `interceptOnException` de todos os comportamentos (classe `Concern`) ativos da tabela caso seja lançada uma exceção na execução do serviço, onde `_component` é o componente personalizável que implementa o serviço, `_request` é a requisição do serviço e `_response` é resposta do serviço que contém a exceção lançada.

1.2. A classe `Concern`

A classe `Concern` define a interface que deve ser implementada pelo comportamento de um componente personalizável. Os métodos são: `interceptBefore`, que implementa o comportamento a ser invocado antes da execução do serviço; `interceptAfter`, que implementa o comportamento a ser invocado após a execução do serviço; e `interceptOnException`, que implementa o comportamento a ser invocado caso seja lançada uma exceção na execução do serviço.

Adaptadores

A classe `Adapter` é um tipo especial de contêiner utilizado para adaptar serviços e eventos implementados em componentes funcionais. Os serviços e os eventos são adaptados através das classes `AdaptedService` e `AdaptedEvent`. As classes que compõem a estrutura de adaptadores do CCF são apresentadas no diagrama da Figura 4.4.

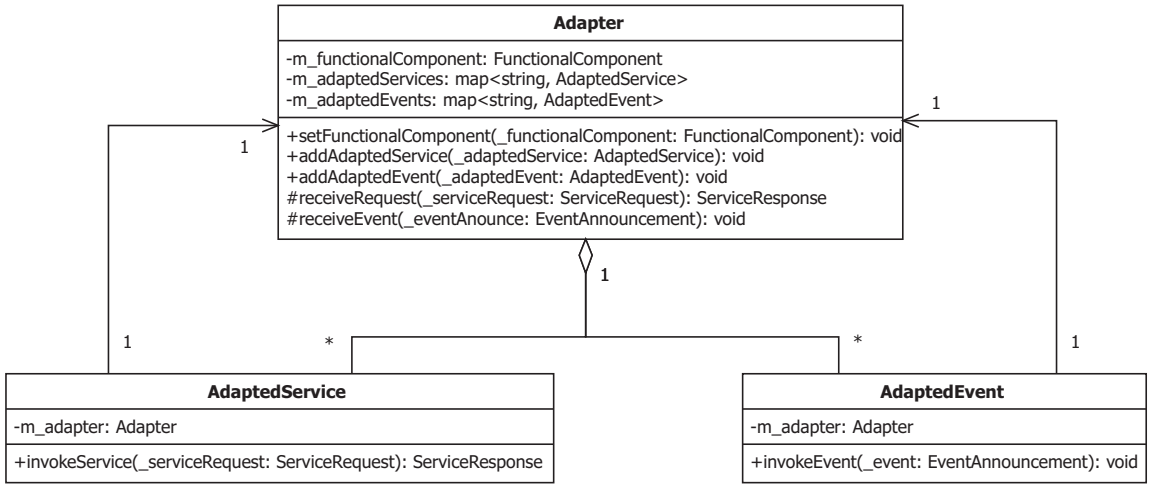

Figura 4.4: Estrutura dos adaptadores do CCF.

1. A classe `Adapter`

Os atributos da classe `Adapter` são: `m_functionalComponent`, referência para o componente funcional que implementa os serviços e eventos que serão adaptados; `m_adaptedServices`, tabela que armazena os apelidos e os serviços adaptados `AdaptedService`; e `m_adaptedEvents`, tabela que armazena os apelidos e os eventos

adaptados `AdaptedEvent`. O método `setFunctionalComponent` atualiza a referência do componente funcional adaptado. `addAdaptedService` e `addAdaptedEvent` adicionam serviços e eventos adaptados, respectivamente. Os outros métodos são detalhados a seguir.

`receiveRequest(_serviceRequest: ServiceRequest): ServiceResponse`

Sobrescreve o método `receiveRequest` da classe `Container`. Na classe `Adapter`, o método verifica se existe alguma adaptação para o serviço requisitado. Se a adaptação for encontrada, o método `invokeService` da classe `AdaptedService` é invocado. Se a adaptação não for encontrada, o método `receiveRequest` do componente funcional adaptado, armazenado em `m_functionalComponent`, é invocado.

`receiveEvent(_eventAnounce: EventAnnouncement): void`

Sobrescreve o método `receiveEvent` da classe `Container`. Na classe `Adapter`, o método verifica se existe alguma adaptação para o evento anunciado. Se a adaptação for encontrada, o método `invokeEvent` da classe `AdaptedEvent` é invocado. Se a adaptação não for encontrada, o método `receiveEvent` do componente funcional adaptado, armazenado em `m_functionalComponent`, é invocado.

1.1. A classe `AdaptedService`

O atributo `m_adapter` é uma referência para o adaptador que armazena o serviço adaptado. O método abstrato `invokeService` deve ser sobrescrito pelas classes que implementam os serviços adaptados.

1.2. A classe `AdaptedEvent`

O atributo `m_adapter` é uma referência para o adaptador que armazena o evento adaptado. O método abstrato `invokeEvent` deve ser sobrescrito pelas classes que implementam os eventos adaptados.

4.1.2 Mecanismos de interação

A interação entre os componentes do CCF pode ocorrer através de serviços ou de eventos. Na interação baseada em serviços, um componente funcional invoca qualquer serviço provido por outro componente da hierarquia. Na interação baseada em eventos, um componente funcional anuncia mudanças para todos os interessados. O diagrama da Figura 4.5 apresenta as classes relacionadas aos mecanismos de interação do CCF.

A interação baseada em serviços é implementada pelas classes `AbstractService`, `ServiceSpecification`, `AdaptedService`, `ProvidedService` e `RequiredService`. A

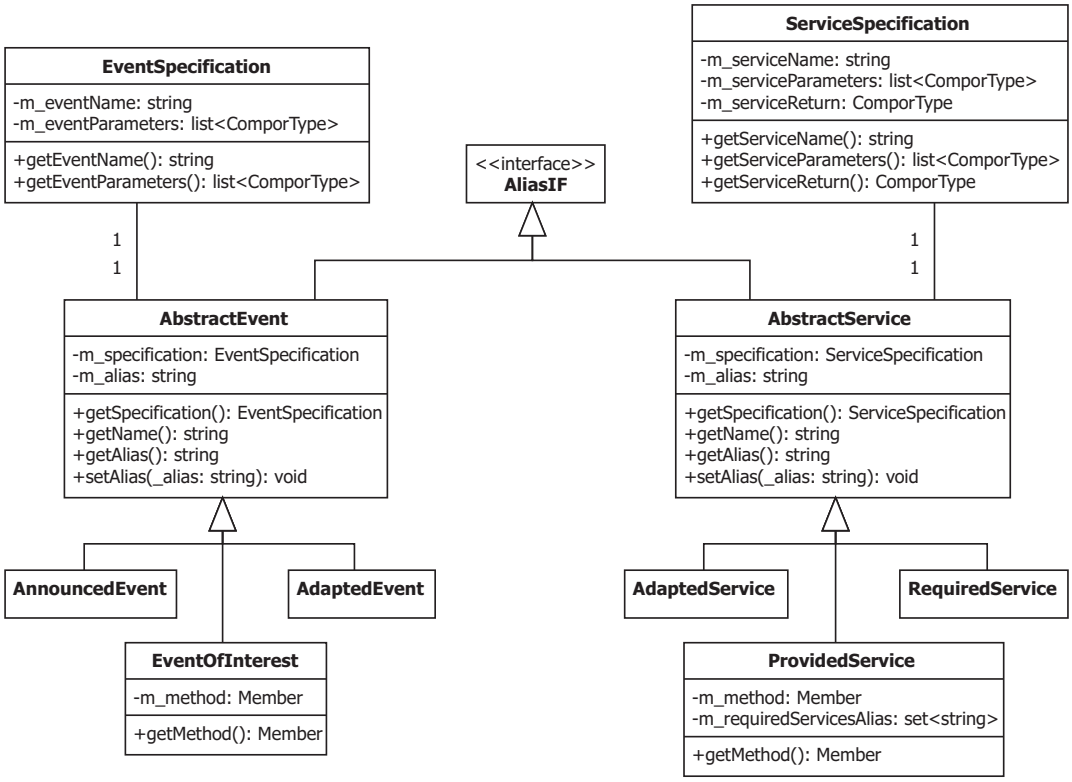

Figura 4.5: Classes relacionadas aos mecanismos de interação do CCF.

classe `AbstractService` é estendida por todas as classes que manipulam serviços no CCF. As especificações dos serviços são instâncias da classe `ServiceSpecification`. As classes `AdaptedService`, `ProvidedService` e `RequiredService` representam os serviços adaptados, providos e requeridos do CCF, respectivamente. As classes `AbstractService`, `ServiceSpecification` e `ProvidedService` são detalhadas a seguir.

1. A classe `AbstractService`

   Os atributos `m_specification` e `m_alias` da classe `AbstractService` armazenam a especificação e apelido do serviço, respectivamente. A especificação do serviço é recuperada através do método `getSpecification`. Os métodos `getName`, `getAlias` e `setAlias` implementam a interface definida pela classe `AliasIF`.

   1.1. A classe `ServiceSpecification`

   Os atributos da classe `ServiceSpecification` armazenam o nome `m_serviceName`, os tipos dos parâmetros `m_serviceParameters` e o tipo de retorno `m_serviceReturn` do serviço. Os tipos são encapsulados em instâncias da classe `ComporType`. Os métodos `getServiceName`, `getServiceParameters`, `getServiceReturn` estão relacionados à recuperação dos atributos da classe.

2. A classe `ProvidedService`

Os atributos da classe `ProvidedService` são uma referência para o método que implementa o serviço `m_method` e uma lista de serviços requeridos `m_requiredServicesAlias`. A referência para o método provedor do serviço `m_method` é armazenada pela classe `ProvidedService` através de reflexão computacional. Quando a requisição do serviço é recebida, o método `getMethod` é invocado pela classe `FunctionalComponent` para recuperar a referência para o método que implementa o serviço provido.

A interação baseada em eventos é implementada pelas classes `AbstractEvent`, `EventSpecification`, `AnnouncedEvent`, `EventOfInterest` e `AdaptedEvent`. A classe `AbstractEvent` é estendida por todas as classes que manipulam eventos no CCF. As especificações dos eventos são instâncias da classe `EventSpecification`. As classes `AnnouncedEvent`, `EventOfInterest` e `AdaptedEvent` representam os eventos anunciados, de interesse e adaptados do CCF, respectivamente. As classes `AbstractEvent`, `EventSpecification` e `EventOfInterest` são detalhadas a seguir.

1. A classe `AbstractEvent`

   Os atributos da classe `AbstractEvent` armazenam a especificação `m_specification` e apelido `m_alias` do evento. A especificação do evento é recuperada através do método `getSpecification`. Os métodos `getName`, `getAlias` e `setAlias` implementam a interface definida pela classe `AliasIF`.

   1.1. A classe `EventSpecification`

   Os atributos da classe `EventSpecification` armazenam o nome `m_eventName` e os tipos dos parâmetros `m_eventParameters` do evento. Os tipos são encapsulados em instâncias da classe `ComporType`. Os métodos `getEventName` e `getEventParameters` estão relacionados à recuperação dos atributos da classe.

2. A classe `EventOfInterest`

   O atributo da classe `EventOfInterest` é uma referência para o método que implementa o evento de interesse `m_method`. A referência para o método que trata o evento de interesse `m_method` é armazenada pela classe `EventOfInterest` através de reflexão computacional. Quando o anúncio do evento é recebido, o método `getMethod` é invocado pela classe `FunctionalComponent` para recuperar a referência para o método que trata o evento de interesse.

A interação entre os componentes do CCF pode ocorrer de maneira assíncrona. A interação baseada em serviços normalmente é síncrona, mas também pode ser assíncrona. A interação baseada em eventos sempre ocorre de maneira assíncrona. As classes apresentadas no digrama da Figura 4.6 estão relacionadas à interação assíncrona do CCF.

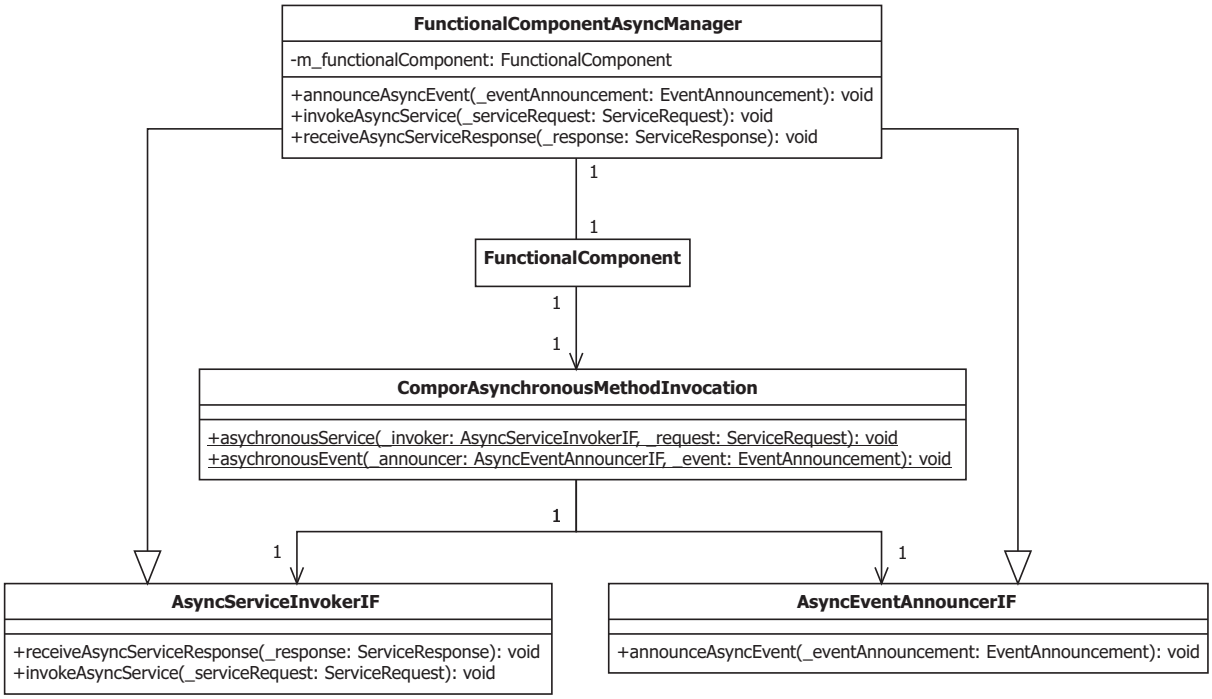

Figura 4.6: Classes relacionadas aos mecanismos de interação assíncrona do CCF.

A classe `ComporAsynchronousMethodInvocation` implementa os métodos estáticos responsáveis pela invocação assíncrona de um serviço e pelo anúncio de um evento. As interfaces `AsyncServiceInvokerIF` e `AsyncEventInvokerIF` implementadas pela classe `FunctionalComponentAsyncManager` gerenciam a invocação assíncrona de serviço e o anúncio de evento, respectivamente. A classe `FunctionalComponentAsyncManager` é responsável por gerenciar os mecanismos de interação assíncrona dos componentes funcionais (`FunctionalComponent`). As classes `ComporAsynchronousMethodInvocation` e `FunctionalComponentAsyncManager` são detalhadas a seguir.

1. A classe `ComporAsynchronousMethodInvocation`

   Os métodos estáticos da classe `ComporAsynchronousMethodInvocation` são apresentados a seguir.

   `asychronousService(_invoker: AsyncServiceInvokerIF, _request: ServiceRequest): void`

   O método `doItAsync` da classe `FunctionalComponent` cria uma nova thread para invocar o método `asychronousService` passando como parâmetro a instância da classe `FunctionalComponentAsyncManager` e a requisição de serviço. `asychronousService` invoca o serviço através do método `invokeAsyncService` da instância passada como parâmetro de `FunctionalComponentAsyncManager`. A resposta do serviço é repassada para a instância de `FunctionalComponentAsyncManager` pelo método `receiveAsyncServiceResponse`.

`asychronousEvent(_announcer: AsyncEventAnnouncerIF, _event: EventAnnouncement): void`

O método `announceEvent` da classe `FunctionalComponent` cria uma nova thread para invocar o método `asychronousEvent` passando como parâmetro a instância da classe `FunctionalComponentAsyncManager` e o anúncio do evento. `asychronousEvent` anuncia o evento através do método `announceAsyncEvent` da instância da classe `FunctionalComponentAsyncManager` passada como parâmetro.

2. A classe `FunctionalComponentAsyncManager`

O atributo `m_functionalComponent` é uma referência para o componente funcional que é gerenciado pela instância de `FunctionalComponentAsyncManager`. Os métodos da classe `FunctionalComponentAsyncManager` que implementam as interfaces `AsyncServiceInvokerIF` e `AsyncEventInvokerIF` são detalhados a seguir.

`announceAsyncEvent(_eventAnnouncement: EventAnnouncement): void`

Invoca o método `announceEvent` do contêiner-pai do componente funcional armazenado em `m_functionalComponent`.

`invokeAsyncService(_serviceRequest: ServiceRequest): void`

Invoca o método `doIt` do componente funcional armazenado em `m_functionalComponent`.

`receiveAsyncServiceResponse(_response: ServiceResponse): void`

Invoca o método `receiveAsyncServiceResponse` do componente funcional armazenado em `m_functionalComponent`.

4.1.3 Execução

A execução de aplicações CCF depende do *script* de execução e do contêiner raíz da hierarquia. As classes relacionadas à execução de aplicações CCF são apresentadas no diagrama da Figura 4.7. A classe `ExecutionScript` é ponto de entrada de uma aplicação CCF. O *script* de execução pode invocar serviços e anunciar eventos para qualquer componente funcional da hierarquia. A classe `ExecutionScriptAsyncManager` implementa o gerenciador de invocação assíncrona de serviço e de anúncio de evento do *script* de execução. A implementação da classe`ExecutionScriptAsyncManager` é similar à da classe `FunctionalComponentAsyncManager`. `ScriptContainer` representa o contêiner raíz da hierarquia. As classes `ScriptContainer` e `ExecutionScript` são detalhadas a seguir.

1. A classe `ScriptContainer`

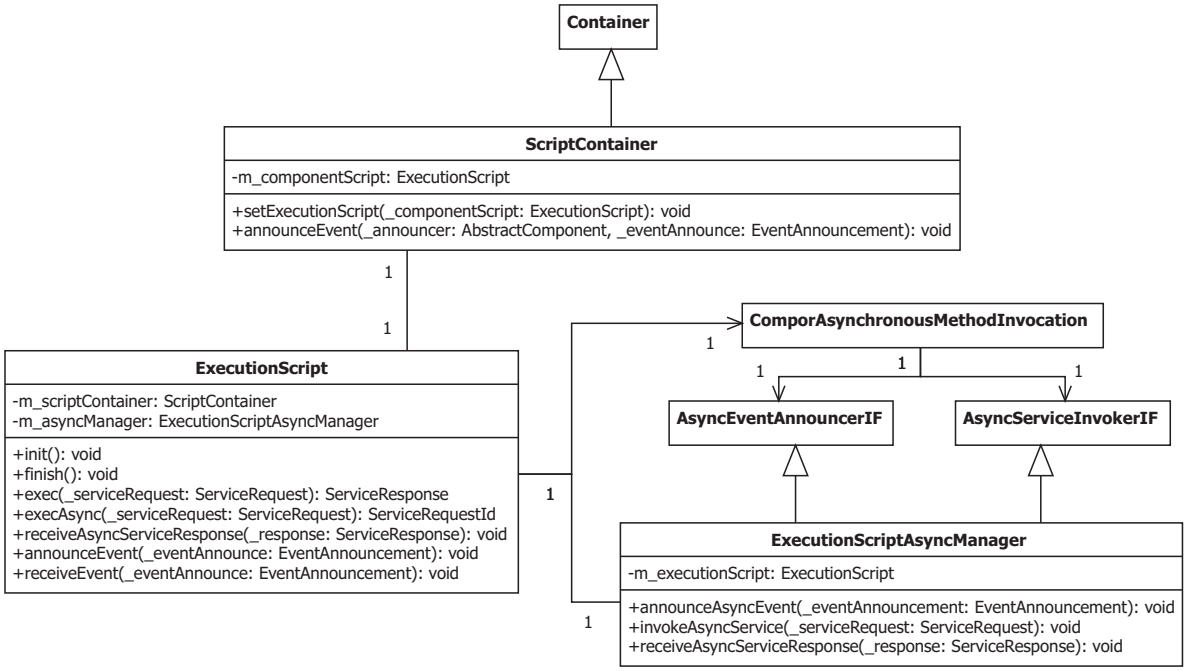

Figura 4.7: Classes relacionadas à execução de aplicações CCF.

O atributo `m_componentScript` armazena a referência para o *script* de execução. O método `setExecutionScript` atualiza a referência do *script* de execução armazenada em `m_componentScript`. O método `announceEvent` é detalhado a seguir.

`announceEvent(_announcer: AbstractComponent, _eventAnnounce: EventAnnouncement): void`

Sobrescreve o método `announceEvent` da classe `Container`. `announceEvent` anuncia o evento para o *script* de execução através do método `receiveEvent` e depois invoca o método `announceEvent` da classe `Container`.

2. A classe `ExecutionScript`

Os atributos são: `m_scriptContainer`, que armazena uma referência para o contêiner principal da aplicação; e `m_asyncManager`, que gerencia os mecanismos de interação assíncrona. Os métodos `init` e `finish` inicializam e interrompem a execução de uma aplicação CCF. Os outros métodos são detalhados a seguir.

`exec(_serviceRequest: ServiceRequest): ServiceResponse`

Invoca o método `doIt` do contêiner raíz `m_scriptContainer`.

`execAsync(_serviceRequest: ServiceRequest): ServiceRequestId`

Invoca o método `asychronousService` da classe `ComporAsynchronousMethodInvocation` em uma nova thread passando como parâmetro a instância da classe `ExecutionScriptAsyncManager` e a requisição do serviço. O serviço é

executado quando o gerenciador de mecanismos de interação assíncrona invoca o método `exec` do *script* de execução.

`receiveAsyncServiceResponse(_response: ServiceResponse): void`

Recebe a resposta da invocação de serviço assíncrona. Este método tem implementação vazia e deve ser sobrescrito caso o *script* de execução precise da resposta do serviço.

`announceEvent(_eventAnnounce: EventAnnouncement): void`

Invoca o método `asychronousEvent` da classe `ComporAsynchronousMethodInvocation` em uma nova thread passando como parâmetro a instância da classe `ExecutionScriptAsyncManager` e o anúncio do evento. O evento é anunciado quando o gerenciador de mecanismos de interação assíncrona invoca o método `announceEvent` do contêiner raíz.

`receiveEvent(_eventAnnounce: EventAnnouncement): void`

Recebe o anúncio de um evento. Este método tem implementação vazia e deve ser sobrescrito caso o *script* de execução precise tratar anúncios de eventos.

4.2 Implementação do CCF

Na seção anterior foram apresentados os detalhes do projeto do CCF. Nesta seção, discute-se a implementação dos principais métodos relacionados à disponibilização de componentes e aos mecanismos de interação do CCF. O mecanismo de reflexão computacional utilizado no CCF também é detalhado.

4.2.1 Disponibilização de componentes

O processo de disponibilização do CCF é implementado pelo método `addComponent` da classe `Container`. O parâmetro do método `addComponent` é uma instância da classe `AbstractComponent` que pode ser um componente funcional ou um contêiner. O método `addComponent` é responsável por atualizar toda a hierarquia de forma que qualquer componente da aplicação possa acessar os serviços providos e anunciar os eventos de interesse do novo componente. O processo de disponibilização do componente X é apresentado na Figura 4.8.

O componente X é adicionado à hierarquia através do método `addComponent` do contêiner 2. O método `addComponent` do contêiner 2 adiciona o componente X à lista de componentes-filhos, atualiza as tabelas de *serviços providos* e *eventos de interesse* através dos métodos `addAllProvidedServices` e `addAllEventsOfInterest` e invoca o método `addComponent` do contêiner 1. O método `addComponent` do contêiner 1 apenas invoca os

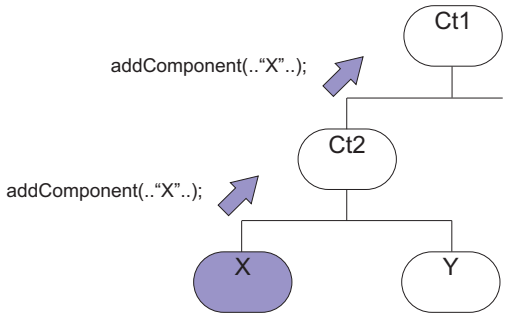

Figura 4.8: Disponibilização de componentes.

métodos `addAllProvidedServices` e `addAllEventsOfInterest` para atualizar as tabelas de *serviços providos* e *eventos de interesse*. A partir daí, qualquer componente da hierarquia pode acessar os serviços e anunciar os eventos do componente X.

4.2.2 Interação baseada em serviços

A interação baseada em serviços é implementada pelos métodos `doIt` e `receiveRequest`. Esses métodos são responsáveis por encontrar o provedor e executar o serviço seguindo a hierarquia. O método `doIt` encaminha a requisição do serviço, no sentido *bottom-up*, até encontrar o contêiner que armazena a referência para o provedor. Quando isso ocorre, o método `receiveRequest` é invocado para repassar a requisição, no sentido *top-down*, até chegar ao componente funcional provedor do serviço (Figura 4.9). Ambos os métodos recebem como parâmetro um objeto `ServiceRequest` que encapsula o nome e os parâmetros necessários para execução do serviço. O retorno dos métodos `doIt` e `receiveRequest` é um objeto `ServiceResponse` que armazena o resultado do serviço, caso a execução ocorra com sucesso, ou a exceção, caso aconteça uma falha.

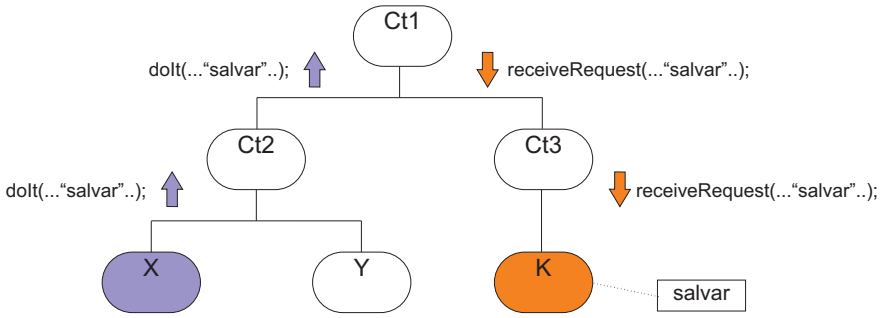

Figura 4.9: Interação baseada em serviços no CCF.

Na interação baseada em serviços assíncrona, o serviço é requisitado através do método `doItAsync`. O método `doItAsync` cria uma nova *thread* para invocar o método `doIt`. O retorno do método `doItAsync` é o identificador da requisição (`ServiceRequestId`). O método `doIt` recebe como parâmetro a requisição do serviço (`ServiceRequest`) e inicia

uma busca em toda hierarquia para encontrar o provedor do serviço. Enquanto isso, a *thread* principal continua sua execução normalmente. Quando a execução do método `doIt` termina, a nova *thread* encaminha a resposta `ServiceResponse` para o componente que requisitou o serviço através do método `receiveAsyncServiceResponse`. Como a resposta `ServiceResponse` possui o identificador da requisição, o componente que invocou o método `doItAsync` pode identificar a qual requisição a resposta pertence.

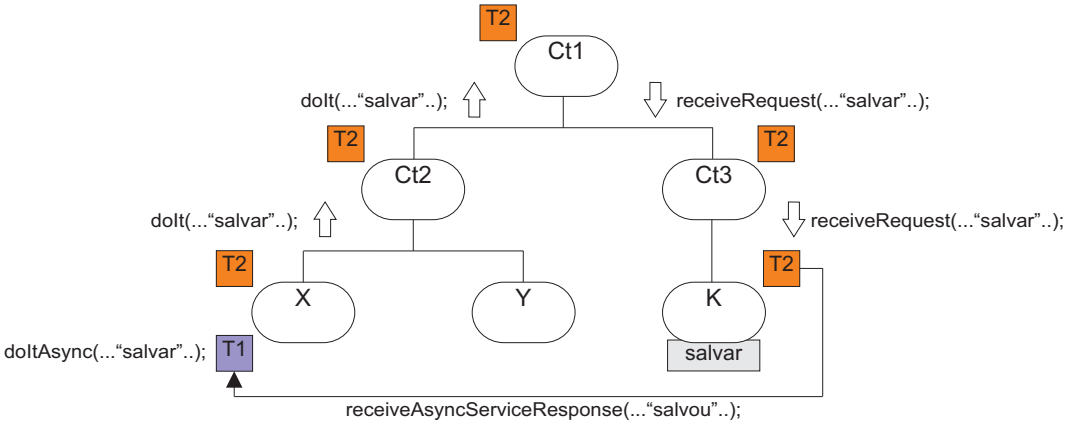

Figura 4.10: Interação baseada em serviços assíncrona no CCF.

A interação de serviços assíncrona é baseada no padrão de projeto *ActiveObject* [37]. O padrão de projeto *ActiveObject* é implementado no CCF através da API Pthreads [28]. A API Pthreads foi escolhida por ser um padrão IEEE POSIX e por apresentar implementações em diversas plataformas, como Linux, Windows, Solaris e etc. O diagrama de seqüencia da Figura 4.11 ilustra a interação de serviços assíncrona no CCF.

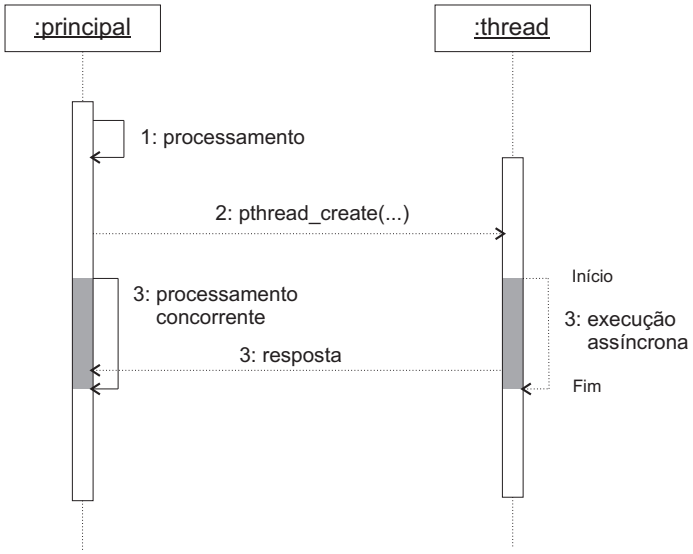

Figura 4.11: Diagrama de seqüencia relacionado à interação de serviços assíncrona no CCF.

4.2.3 Interação baseada em eventos

A interação baseada em eventos é implementada pelos métodos `announceEvent` e `receiveEvent` e sempre ocorre de maneira assíncrona [3]. O método `announceEvent` anuncia o evento encapsulado em um objeto da classe `EventAnnouncement`, no sentido *bottom-up*, para todos os componentes da hierarquia em uma nova *thread*. Quando um contêiner identifica que um componente-filho tem interesse no evento, o método `receiveEvent` é invocado, no sentido *top-down*, até o anúncio chegar ao componente interessado. Não há resposta em um anúncio de evento.

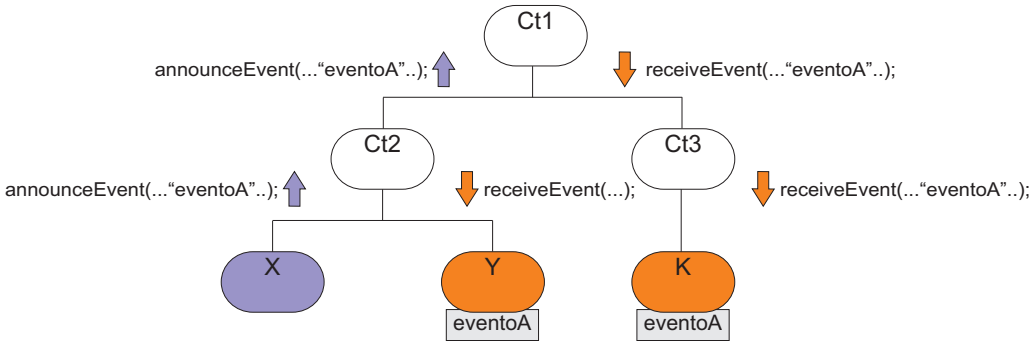

Figura 4.12: Interação baseada em eventos no CCF.

4.2.4 Reflexão computacional

Os métodos que são invocados quando uma requisição de serviço ou um anúncio de evento ocorre são armazenados pela classe `FunctionalComponent` em objetos do tipo `ProvidedService` e `EventOfInterest`, respectivamente [3]. O armazenamento e a recuperação desses métodos são feitos através de reflexão computacional [9]. Como C++ não possui um mecanismo de reflexão nativo, o CCF utiliza a biblioteca Seal Reflex [33]. A biblioteca Seal Reflex possibilita um programa C++ criar um objeto a partir do nome da classe, invocar um método utilizando as informações da sua assinatura, recuperar informações sobre uma classe e etc.

Seal reflex satisfaz as duas funcionalidades requeridas de uma API de reflexão na especificação CMS:

1. A possibilidade de armazenar uma referência para um método de uma classe em um atributo. Um exemplo de como armazenar uma referência para um método em um atributo utilizando a biblioteca Seal Reflex é apresentado na Listagem de Código 4.1. A classe `Reflection` possui um atributo `m_method` do tipo `Member`. `Member` é uma classe de Seal Reflex utilizada para armazenar referências a métodos. O construtor da classe `Reflection` recupera a referência para o método `test` e

armazena na variável `m_method`. O valor da variável `m_method` pode ser recuperado por outras classes através do método `getMethod`.

Código 4.1: Armazenamento de um método em um atributo.

```
1  class Reflection {
2  public:
3      inline Reflection() {
4          Type type = Type::ByName("Reflection");
5          m_method = type.MemberByName("test");
6      }
7      inline virtual ~Reflection() {}
8      inline void test(const string & param) {
9          cout << "Reflexão computacional em C++ " << param << endl;
10     }
11     inline Member & getMethod() { return m_method; }
12
13 private:
14     Member m_method;
15 };
```

2. A possibilidade de invocar o método armazenado em um atributo. Um exemplo de como invocar um método armazenado em um atributo utilizando a biblioteca de reflexão Seal Reflex é apresentado na Listagem de Código 4.2. O atributo que armazena a referência para o método `test` da classe `Reflection` é recuperado na linha 2. A invocação do método `test` é realizada via reflexão computacional através do método `Invoke` da classe `Member` de Seal Reflex, linha 7. O método `Invoke` recebe a referência para objeto que implementa o método `test` e um vetor com os parâmetros que serão utilizados pelo método `test`.

Código 4.2: Invocação de um método armazenado em um atributo.

```
1  Reflection ref;
2  Member method = ref.getMethod();
3  vector<void *> parameters;
4  string param = "funciona!"
5
6  parameters.push_back(&param);
7  method.Invoke(Object(Type(), &ref), parameters);
```

As funcionalidades descritas nas Listagens de Código 4.1 e 4.2 são utilizadas pela classe `FunctionalComponent` para armazenar e invocar os serviços providos e os eventos de interesse, como ilustrado na Figura 4.13. A implementação do método `receiveRequest` da classe `FunctionalComponent` recupera o método definido pelo componente K como implementador do serviço *salvar* e o invoca utilizando o mecanismo de reflexão Seal Reflex. O mesmo procedimento acontece no método `receiveEvent` da classe `FunctionalComponent` quando ocorre uma notificação de evento.

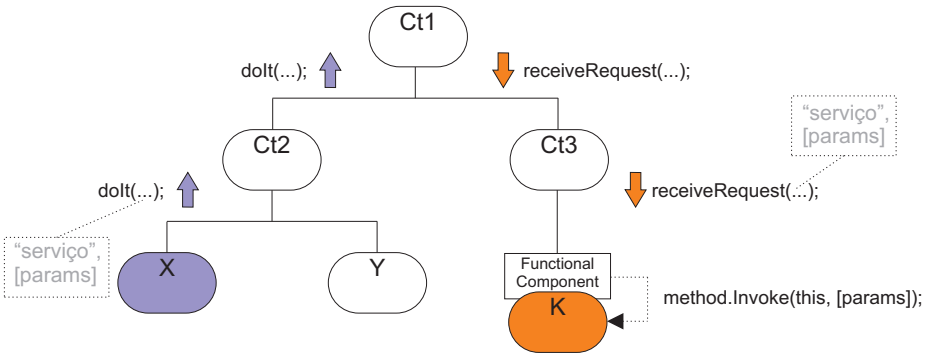

Figura 4.13: Mecanismo de reflexão no CCF.

4.3 Como utilizar o CCF?

A seguir, descreve-se como utilizar a API do arcabouço para construir aplicações CCF. Apenas as principais funcionalidades são descritas. Mais detalhes sobre a API e o código fonte podem ser encontrados no site do CCF (`http://gforge.embedded.ufcg.edu.br/projects/ccf/`).

4.3.1 Criando componentes

Componentes funcionais

Os componentes funcionais são instâncias de classes que herdam de `FunctionalComponent`. Um exemplo de componente funcional é apresentado na Listagem de Código 4.3. O componente funcional `ComponentMinus` invoca e provê serviços e recebe eventos, ou seja, utiliza quase todos os mecanismos de interação disponíveis no CCF. O construtor da classe `ComponentMinus` adiciona os serviços requeridos, serviços providos e eventos de interesse através dos métodos `addRequiredServices`, `addProvidedServices` e `addInterestEvents`, respectivamente.

Código 4.3: Exemplo de um componente funcional.

```
1  class ComponentMinus : public FunctionalComponent {
2  public:
3      inline ComponentMinus(): FunctionalComponent("Componente Menos") {
4          addRequiredServices();
5          addProvidedServices();
6          addInterestEvents();
7      }
8      inline virtual ~ComponentMinus() {}
9      inline int operationMinus(const int & _arg1, const int & _arg2) {
10         vector <void *> param;
11         param.push_back(new int(_arg1));
12         param.push_back(new int(-1 * _arg2));
13         ServiceRequest request("plus", param);
14         ServiceResponse *response = doIt(request);
15         int data = *(int *)response->getData();
16         delete response;
17         return data;
18     }
19     inline void operationDone() {
20         cout << "Operação realizada com sucesso!" << endl;
21     }
22
23 private:
24     inline void addRequiredServices() {
25         ServiceSpecification *spec = new ServiceSpecification("plus");
26         RequiredService *service = new RequiredService(spec);
27         addRequiredService(spec);
28     }
29     inline void addProvidedServices() {
30         Type type = Type::ByName("ComponentMinus");
31         Member method = type.MemberByName("operationMinus");
32         ServiceSpecification *spec = new ServiceSpecification("minus");
33         ProvidedService *service = new ProvidedService(spec, method);
34         addProvidedService(service);
35     }
36     inline void addInterestEvents() {
37         Type type = Type::ByName("ComponentMinus");
38         Member method = type.MemberByName("operationDone");
39         EventSpecification *spec = new EventSpecification("done");
40         EventOfInterest *event = new EventOfInterest(spec, method);
41         addEventOfInterest(event);
42     }
43 };
```

O método `addRequiredServices` adiciona o serviço requerido *plus*. O serviço *plus* é provido por outro componente. O método `operationMinus` é adicionado como implementador do serviço provido *minus* pelo método `addProvidedServices`. A referência para o método que implementa o serviço *minus* é recuperada utilizando a API de reflexão Seal Reflex [33], linhas 30 e 31, e armazenada em uma instância da classe `ProvidedService`. É importante observar que o nome do serviço ou do evento pode ser diferente do nome do método que o implementa.

O método `addAnnouncedEvents` adiciona o evento anunciado *done* pelo componente funcional `ComponentMinus`. O método `operationDone` é adicionado como implementador do evento de interesse *done* pelo método `addInterestEvents`. A referência para o método que implementa o evento de interesse é recuperada também utilizando a API de reflexão Seal Reflex, linhas 37 e 38, e armazenada em uma instância da classe `EventOfInterest`.

O serviço *minus* provido pelo componente `ComponentMinus` subtrai dois números inteiros e retorna o resultado da operação. O método `operationMinus` implementa a subtração de inteiros utilizando o serviço *plus*, linhas 13 e 14, provido por outro componente da hierarquia. O componente `ComponentMinus` também tem interesse em um evento anunciado por outro componente. O método `operationDone` trata o evento de interesse *done* imprimindo uma mensagem de sucesso.

A invocação do serviço *minus* também pode ser realizada de maneira assíncrona através do método `doItAsync`. Nesse caso, a classe do componente funcional que requisitar o serviço através do método `doItAsync` deve sobrescrever o método `receiveAsyncResponse` para receber a resposta do serviço. É importante ressaltar que a execução dos métodos `doIt`, `doItAsync` e `announceEvent` só fazem sentido quando o componente faz parte de uma hierarquia CCF.

Componentes funcionais personalizáveis

Os componentes funcionais personalizáveis são componentes que podem ser modificados através de comportamentos. Os comportamentos devem estender da classe `Concern` como mostra a Listagem de Código 4.4. A classe `LogConcern` adiciona o comportamento de *Log* às requisições de serviços de um componente funcional personalizável. Os métodos `interceptBefore` e `interceptAfter` são invocados antes e após a execução de um serviço. O método `interceptOnException` é invocado caso ocorra uma exceção na execução do serviço.

Código 4.4: Exemplo de um comportamento.

```
1  class LogConcern : public Concern {
2  public:
3      inline LogConcern() : Concern("Log") {}
4          inline virtual ~LogConcern() {}
5      inline void interceptBefore(CustomizableComponent * _component,
6                                  ServiceRequest & _request) {
7          cout << "Log - Antes: " << _request.getServiceName() << endl;
8      }
9      inline void interceptAfter(CustomizableComponent * _component,
10                     ServiceRequest & _request, ServiceResponse * _response) {
11         cout << "Log - Depois: " << _request.getServiceName() << endl;
12     }
13     inline void interceptOnException(CustomizableComponent * _component,
14                     ServiceRequest & _request, ServiceResponse * _response) {
15         cout << "Log - Exceção: " << _request.getServiceName() << endl;
16     }
```

Os componentes funcionais personalizáveis são instâncias de classes que herdam de `CustomizableComponent`. Um exemplo de componente funcional personalizável é apresentado na Listagem de Código 4.5. O construtor da classe `ComponentPlus` adiciona os serviços providos e os eventos anunciados através dos métodos `addProvidedServices` e `addAnnouncedEvents`, respectivamente. O método `addConcern` adiciona o comportamento implementado por `LogConcern` à classe `ComponentPlus`, inicializando-o como um comportamento ativado. A publicação de serviços e eventos deste tipo de componente ocorre da mesma forma que um componente funcional comum.

O serviço *plus* provido pelo componente funcional personalizável `ComponentPlus` através do método `operationPlus` soma dois números inteiros e anuncia o evento *done* para os componentes interessados da hierarquia. Neste exemplo o evento não possui parâmetros. Para eventos com parâmetros, basta passá-los como segundo argumento no construtor de `EventAnnouncement`. O comportamento `LogConcern` é invocado antes e depois da execução do serviço *plus*. Caso ocorra uma exceção na execução do serviço *plus*, o comportamento `LogConcern` também é invocado.

Código 4.5: Exemplo de um componente funcional personalizável.

```
class ComponentPlus : public CustomizableComponent {
public:
    inline ComponentPlus() : CustomizableComponent("Componente Mais") {
        addProvidedServices();
        addAnnouncedEvents();
        addConcern(new LogConcern(), true);
    }
    inline virtual ~ComponentPlus() {}
    inline int operationPlus(const int & _arg1, const int & _arg2) {
        EventAnnouncement *event = new EventAnnouncement("done");
        announceEvent(event);
        return _arg1 + _arg2;
    }

private:
    inline void addProvidedServices() {
        Type type = Type::ByName("ComponentPlus");
        Member method = type.MemberByName("operationPlus");
        ServiceSpecification *spec = new ServiceSpecification("plus");
        ProvidedService *service = new ProvidedService(spec, method);
        addProvidedService(service);
    }
    inline void addAnnouncedEvents() {
        EventSpecification *spec = new EventSpecification("done");
        AnnouncedEvent *event = new AnnouncedEvent(spec);
        addAnnouncedEvent(event);
    }
```

Adaptadores

Os adaptadores são tipos especiais de contêineres utilizados para modificar serviços e eventos implementados por componentes da hierarquia CCF. As adaptações do serviço *minus* e do evento *done* do componente funcional `ComponentMinus` são apresentadas nas Listagens de Código 4.6 e 4.7. Os serviços adaptados devem estender da classe `AdaptedService`. Os eventos adaptados estendem da classe `AdaptedEvent`.

Código 4.6: Exemplo de um serviço adaptado.

```
class MinusAdaptation : public AdaptedService {
public:
    inline MinusAdaptation(): AdaptedService(new ServiceSpecification("minus")) {}
    inline virtual ~MinusAdaptation() {}
    inline ServiceResponse * invokeService(ServiceRequest & _serviceRequest) {
        cout << "Adaptação do serviço minus" << endl;
        return getAdapter()->invokeOriginalService(_serviceRequest);
    }
```

A classe `MinusAdaptation` implementa a adaptação do serviço *minus* do `ComponentMinus`. O construtor da classe `MinusAdaptation` cria a especificação do serviço *minus* e passa para o construtor da classe `AdaptedService`. A adaptação do serviço *minus* é im-

plementada pelo método `invokeService`. O método `invokeService` imprime um texto e invoca o serviço original.

Código 4.7: Exemplo de um evento adaptado.

```
1 class DoneAdaptation : public AdaptedEvent {
2 public:
3     inline DoneAdaptation() : AdaptedEvent(new EventSpecification("done")) {}
4     inline virtual ~DoneAdaptation() {}
5     inline void invokeEvent(EventAnnouncement & _event) {
6         cout << "Adaptação do evento done" << endl;
7         getAdapter()->invokeOriginalEvent(_event);
8     }
```

A classe `DoneAdaptation` implementa a adaptação do evento *done* do `ComponentMinus`. O construtor da classe `DoneAdaptation` cria a especificação do evento *done* e passa para o construtor da classe `AdaptedEvent`. A adaptação do evento *done* é implementada pelo método `invokeEvent`. O método `invokeEvent` imprime um texto e invoca o evento original.

A adaptação do `ComponentMinus` é apresentada na Listagem de Código 4.8. Na linha 1, o adaptador é criado como uma instância da classe `Adapter`. O serviço adaptado `MinusAdaptation` e o evento adaptado `DoneAdaptation` são adicionados à instância do adaptador, linhas 3 e 4. O componente funcional adaptado é definido na linha 5. A partir daí, as requisições ao serviço *minus* e as notificações do evento *done* são direcionadas aos respectivos adaptadores.

Código 4.8: Criação de um adaptador.

```
1 Adapter *adapter = new Adapter("Adaptador Componente Menos");
2
3 adapter->addAdaptedService(new MinusAdaptation());
4 adapter->addAdaptedEvent(new DoneAdaptation());
5 adapter->setFunctionalComponent(componentMinus);
```

4.3.2 Montando aplicações

Uma vez implementados, os componentes devem ser organizados em uma hierarquia CCF. O exemplo de uma hierarquia de aplicação CCF é apresentado na Figura 4.14. A hierarquia é formada por: um *script* de execução que é o ponto de entrada da aplicação; um contêiner raíz; um contêiner menos que armazena o componente menos; e um contêiner mais que armazenar o componente mais.

A construção da hierarquia da aplicação ilustrada na Figura 4.14 é descrita na Listagem de Código 4.9. O contêiner raíz é criado na linha 1. Nas linhas 2 e 3, os contêineres menos e mais são criados como instâncias da classe `Container`. Os componentes funcionais menos e mais são instâncias das classes `ComponentMinus` e `ComponentPlus` (linhas 4 e 5). Uma

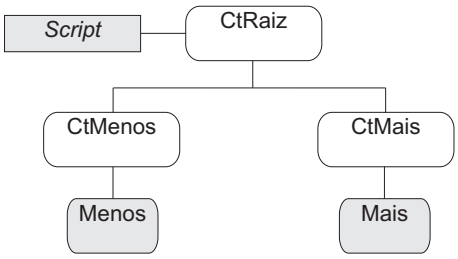

Figura 4.14: Exemplo de uma hierarquia CCF.

vez criados os componentes, deve-se montar a hierarquia. Nas linhas 8 e 9, os componentes menos e mais são inseridos nos contêineres menos e mais, respectivamente. O contêiner raíz adiciona os seus componentes filhos nas linhas 10 e 11, ou seja, os contêineres menos e mais. Agora falta apenas a definição de um *script* de execução para que a aplicação esteja completa. O *script* de execução é criado como uma instância da classe `ExecutionScript` (linha 12) passando como parâmetro o contêiner raíz da aplicação CCF. Observe que o contêiner raíz é uma instância da classe `ScriptContainer` para que seja possível associá-lo a um *script* de execução. Para iniciar a aplicação CCF basta invocar o método `init` da classe `ExecutionScript`. O método `init` inicia o contêiner raíz que por sua vez inicia toda a hierarquia recursivamente.

Código 4.9: Montando a hierarquia da aplicação.

```
1  ScriptContainer *root = new ScriptContainer("Raíz");
2  Container *containerMinus = new Container("Contêiner Menos");
3  Container *containerPlus = new Container("Contêiner Mais");
4  ComponentMinus *componentMinus = new ComponentMinus();
5  ComponentPlus *componentPlus = new ComponentPlus();
6  ExecutionScript *script = NULL;
7
8  containerMinus->addComponent(componentMinus);
9  containerPlus->addComponent(componentPlus);
10 root->addComponent(containerMinus);
11 root->addComponent(containerPlus);
12 script = new ExecutionScript(root);
13 script->init();
```

Após a montagem da aplicação CCF, qualquer componente da hierarquia pode ser atualizado ou removido em tempo de execução sem nenhum tipo de planejamento prévio para tal. Além disso, novos componentes podem ser adicionados sempre que necessário. Assim, a aplicação CCF pode ser modificada em qualquer parte e a qualquer momento, ou seja, a aplicação CCF pode evoluir de forma dinâmica e não antecipada. Uma vez montada, a aplicação CCF deve ser implantada no ambiente de execução. O servidor CCAS, apresentado no próximo capítulo, é responsável por gerenciar a implantação e a execução das aplicações desenvolvidas utilizando o arcabouço CCF.

Capítulo 5

C++ Component Application Server

No capítulo 4 foram apresentados os detalhes do projeto e da implementação do CCF, assim como, um guia de como utilizar a API do CCF para construir aplicações. Uma vez implementada, a aplicação CCF precisa ser preparada para implantação. O processo de implantação engloba a preparação dos componentes, a construção da hierarquia da aplicação e, finalmente, o envio dos componentes e da hierarquia da aplicação para o ambiente de execução. No ambiente de execução a aplicação CCF é montada seguindo a hierarquia especificada.

Os passos de implantação são comuns para todas as aplicações CCF independente da natureza das mesmas. Além disso, algumas etapas realizadas nesse processo não são triviais como, por exemplo, geração das bibliotecas de reflexão, comunicação remota entre processos e carregamento dinâmico de classes. Desta forma, fez-se necessária a implementação do *C++ Component Application Server* (CCAS) [31]. O CCAS é responsável por gerenciar a implantação e a execução das aplicações CCF.

O CCAS elimina toda a replicação e complexidade do código necessário para implantação de novos componentes das aplicações CCF. Assim, os desenvolvedores de aplicações CCF podem evoluir qualquer componente da hierarquia sem escrever nenhuma linha de código para tratar tal evolução. Para o CCAS uma aplicação CCF é uma organização hierárquica de componentes que pode ser evoluída de forma dinâmica e não antecipada. Por evolução entende-se inserção, atualização e remoção de componentes.

5.1 Arquitetura do CCAS

A arquitetura geral do CCAS é ilustrada na Figura 5.1. O CCAS é composto por dois módulos: o cliente e o servidor [3]. O módulo cliente, ou módulo de desenvolvimento, é responsável por preparar e enviar os componentes para o módulo servidor. O módulo servidor, ou módulo de execução, é responsável por implantar os componentes enviados

pelo módulo cliente dinamicamente nas respectivas aplicações CCF.

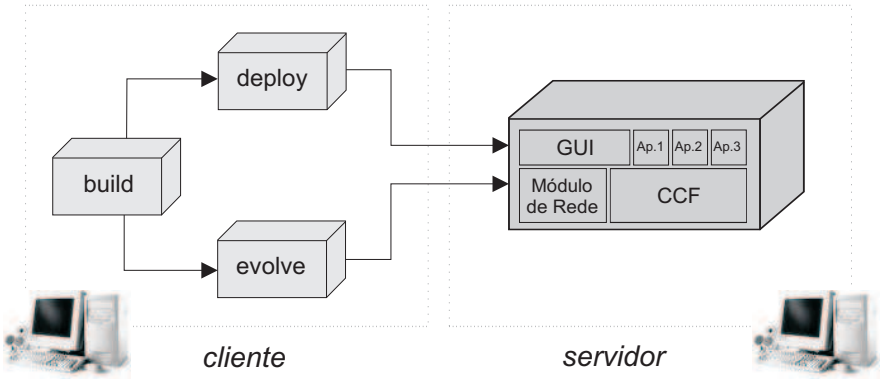

Figura 5.1: Arquitetura geral do CCAS.

5.1.1 Cliente

O módulo cliente é divido em três aplicações (Figura 5.1): a aplicação *build* é responsável por preparar os componentes que serão inseridos em uma hierarquia CCF; a aplicação *deploy* é responsável por implantar uma nova aplicação CCF no módulo servidor; a aplicação *evolve* atualiza a hierarquia de uma aplicação CCF que já se encontra no módulo servidor. Tendo em vista que o módulo servidor pode executar em uma máquina remota, todas as interações entre as aplicações *deploy* e *evolve* com o módulo servidor são feitas através de mecanismos de comunicação remota.

Na Figura 5.2, é apresentada uma analogia das três aplicações do módulo cliente com as etapas de fabricação e entrega de um produto convencional. Na aplicação *build*, os produtos são fabricados e empacotados. Na aplicação *deploy*, os produtos são enviados pela primeira vez para o cliente. Na aplicação *evolve*, os produtos são reenviados para o cliente seguindo um roteiro de entrega pré-definido. O roteiro pode conter instruções como entrega, devolução e substituição de produtos. Na analogia descrita, os produtos seriam os componentes CCF e os clientes seriam as aplicações CCF.

Preparando uma aplicação CCF para implantação

A aplicação *build* recebe como parâmetro um arquivo que contém os caminhos para os arquivos de cabeçalho e as bibliotecas dos componentes e algumas opções de compilação. Um exemplo da estrutura do arquivo de entrada da aplicação *build* é apresentado na Listagem de Código 5.1. O arquivo contém um grupo [ccasBuild] formado pelas quatro chaves apresentadas a seguir.

- ***projectPath*** - caminho para o diretório principal do projeto no módulo cliente

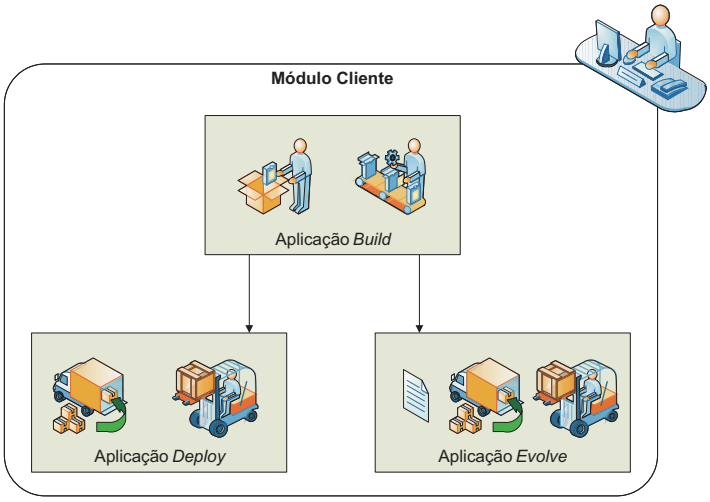

Figura 5.2: Analogia com o módulo cliente.

- ***components*** - pares <arquivo de cabeçalho; biblioteca> de componentes separados por ponto e vírgula
- ***cflags*** - parâmetros necessários para a compilação dos componentes definidos em *components*
- ***libs*** - parâmetros necessários para realizar o *link* dos componentes definidos em *components*

Código 5.1: Exemplo de um arquivo de entrada da aplicação *build*.

```
1 [ccasBuild]
2
3 projectPath=/myProject/path/
4 components=src/component1.h;lib/component1.so;src/component2.h;lib/component2.so;
5 cflags=-I/Dependency/inc
6 libs=-lDependency
```

Implantando uma aplicação CCF

A aplicação *deploy* recebe como parâmetro um XML que descreve a hierarquia da aplicação CCF e um caminho para o diretório onde se encontram as bibliotecas dos componentes que serão implantadas no módulo servidor. Um exemplo da estrutura do XML que descreve a hierarquia de uma aplicação CCF é apresentado na Listagem de Código 5.2. As possíveis *tags* que definem a hierarquia de uma aplicação CCF são descritas a seguir.

- ***ccasFile*** - tag principal do XML que define uma aplicação CCF
- ***serverUrl*** - endereço do módulo servidor

- ***application*** - define uma aplicação CCF, onde o atributo *name* é o nome da aplicação
- ***rootContainer*** - representa o contêiner raíz em uma hierarquia CCF, onde o atributo *name* é o nome do contêiner raíz
- ***container*** - representa um contêiner em uma hierarquia CCF exceto o raíz, onde o atributo *name* é o nome do contêiner
- ***component*** - representa um componente funcional em uma hierarquia CCF exceto um componente funcional personalizável, onde o atributo *name* é o nome do componente, *lib* é a biblioteca do componente e *dictionary* é o dicionário do componente
- ***adapter*** - representa um adaptador em uma hierarquia CCF, onde o atributo *name* é o nome do adaptador, *lib* é a biblioteca do adaptador e *dictionary* é o dicionário do adaptador
- ***customizableComponent*** - representa um componente funcional personalizável em uma hierarquia CCF, onde o atributo *name* é o nome do componente, *lib* é a biblioteca do componente e *dictionary* é o dicionário do componente

Código 5.2: Exemplo do XML que descreve a hierarquia de uma aplicação CCF.

```
<ccasFile>
   <serverUrl>http://localhost:8080</serverUrl>
   <application name="myProject">
     <rootContainer name="rootContainer">
       <container name="myContainer">
         <component name="component1" lib="component1.so" dictionary="component1Dict.so">
           <adapter name="adapter1" lib="adapter1.so"
             dictionary="adapter1Dict.so"></adapter>
         </component>
         <customizableComponent name="component2" lib="component2.so"
           dictionary="component2Dict.so"></customizableComponent>
       </container>
     </rootContainer>
   </application>
</ccasFile>
```

Evoluindo uma aplicação CCF

A aplicação *evolve* recebe os mesmo parâmetros da aplicação *deploy*, ou seja, um XML que descreve a hierarquia da aplicação CCF após a evolução e um caminho para o diretório onde se encontram as bibliotecas dos componentes que serão evoluídos no módulo servidor. A estrutura do XML que descreve a hierarquia de uma aplicação CCF é a mesma

apresentada na Listagem de Código 5.2. Além disso, a aplicação *evolve* recebe como parâmetro um arquivo que contém os comandos necessários para evoluir uma aplicação CCF. O comandos são executados pelo módulo servidor na ordem definida no arquivo, ou seja, da primeira à última linha. Cada linha deve conter um comando. Os comandos possíveis são descritos a seguir.

addContainer <container-path>

Adiciona um contêiner na hierarquia de uma aplicação CCF, onde o parâmetro <container-path> é o caminho completo do contêiner na hierarquia. Exemplo: addContainer rootContainer/myContainer. Os níveis da hierarquia são separados por barra. No exemplo, o contêiner myContainer será adicionado no contêiner raiz rootContainer

addComponent <component-path> <component-lib> <dictionary-lib>

Adiciona um componente na hierarquia de uma aplicação CCF, onde <component-path> é o caminho completo do componente na hierarquia, <component-lib> é o nome da biblioteca do componente e <dictionary-lib> é o nome da biblioteca do dicionário. Exemplo: addComponent rootContainer/myContainer/component1 component1.so component1Dict.so.

addCustomComponent <component-path> <component-lib> <dictionary-lib>

Adiciona um componente personalizável na hierarquia de uma aplicação CCF, onde <component-path> é o caminho completo do componente personalizável na hierarquia, <component-lib> é o nome da biblioteca do componente personalizável e <dictionary-lib> é o nome da biblioteca do dicionário. Exemplo: addCustomComponent rootContainer/myContainer/component2 component2.so component2Dict.so.

addAdapter <adapter-path> <adapter-lib> <dictionary-lib>

Adiciona um adaptador na hierarquia de uma aplicação CCF, onde <adapter-path> é o caminho completo do adaptador na hierarquia, <adapter-lib> é o nome da biblioteca do adaptador e <dictionary-lib> é o nome da biblioteca do dicionário. Exemplo: addAdapter rootContainer/myContainer/component1/adapter1 adapter1.so adapter1Dict.so. No caminho completo passado em <adapter-path> o elemento pai do adaptador é o componente que será adaptado, no exemplo, component1.

remComponent <component-path>

Remove um componente da hierarquia de uma aplicação CCF, onde o parâmetro <component-path> é o caminho completo do componente na hierarquia. Exemplo: remComponent rootContainer/myContainer/component2. Qualquer tipo de componente pode ser removido da hierarquia através desse comando exceto um adaptador. Caso o componente removido seja um contêiner, todos os seus filhos também são removidos da hierarquia.

remAdapter <adapter-path>

Remove um adaptador da hierarquia de uma aplicação CCF, onde o parâmetro <adapter-path> é o caminho completo do adaptador na hierarquia. Exemplo: remAdapter rootContainer/myContainer/component1/adapter1.

activateConcern <component-path> <concern-name>

Ativa um comportamento de um componente personalizável da hierarquia de uma aplicação CCF, onde <component-path> é o caminho completo do componente personalizável e <concern-name> é o nome do comportamento a ser ativado. Exemplo: activateConcern rootContainer/myContainer/component2 concern1.

deactivateConcern <component-path> <concern-name>

Desativa um comportamento de um componente personalizável da hierarquia de uma aplicação CCF, onde <component-path> é o caminho completo do componente personalizável e <concern-name> é o nome do comportamento a ser desativado. Exemplo: deactivateConcern rootContainer/myContainer/component2 concern1.

setAliasEventOfInterest <component-path> <event-alias> <new-event-alias>

Atualiza o *alias* de um evento de interesse de um componente da hierarquia de uma aplicação CCF, onde <component-path> é o caminho completo do componente que contém o evento de interesse, <event-alias> é o *alias* atual do evento de interesse e <new-event-alias> é o novo *alias* do evento de interesse.

setAliasAnnouncedEvent <component-path> <event-alias> <new-event-alias>

Atualiza o *alias* de um evento anunciado por um componente da hierarquia de uma aplicação CCF, onde <component-path> é o caminho completo do componente que anuncia o evento, <event-alias> é o *alias* atual do evento anunciado e <new-event-alias> é o novo *alias* do evento anunciado.

setAliasProvidedService <component-path> <service-alias> <new-service-alias>

Atualiza o *alias* de um serviço provido por um componente da hierarquia de uma aplicação CCF, onde <component-path> é o caminho completo do componente que provê o serviço, <service-alias> é o *alias* atual do serviço provido e <new-service-alias> é o novo *alias* do serviço provido.

setAliasRequiredService <component-path> <service-alias> <new-service-alias>

Atualiza o *alias* de um serviço requerido por um componente da hierarquia de uma aplicação CCF, onde <component-path> é o caminho completo do componente que requer o serviço, <service-alias> é o *alias* atual do serviço requerido e <new-service-alias> é o novo *alias* do serviço requerido.

5.1.2 Servidor

O servidor é subdividido em quatro módulos internos como ilustrado na Figura 5.3: o módulo de rede é responsável por receber o conteúdo dos arquivos que contêm a hierarquia, os componentes e, caso necessário, os comandos de evolução de uma aplicação CCF; o módulo de carregamento dinâmico é responsável por carregar e inserir os componentes nas aplicações CCF seguindo a hierarquia especificada; o CCF é o arcabouço com suporte à evolução dinâmica não antecipada sobre o qual as aplicações são implementadas; GUI é a interface gráfica que permite iniciar ou parar as aplicações CCF implantadas no servidor.

Figura 5.3: Arquitetura do módulo servidor.

Na Figura 5.4, é apresentada uma analogia do módulo servidor com as etapas de recebimento de um produto convencional. No módulo de rede, os produtos são recebidos. No módulo de carregamento dinâmico, os produtos são desempacotados e verificados. No

CCF, os produtos são entregues ao consumidor final. Na analogia, os produtos seriam os componentes e os consumidores finais seriam as aplicações CCF.

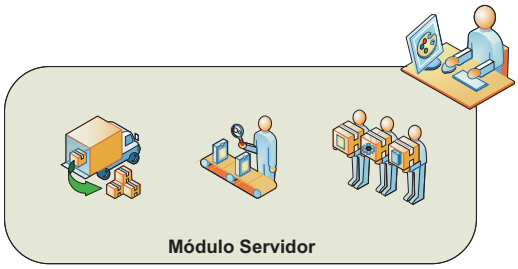

Figura 5.4: Analogia com o módulo servidor.

5.2 Implementação do CCAS

O CCAS utiliza a infra-estrutura fornecida pela biblioteca Seal Reflex [33] tanto no módulo cliente quanto no módulo servidor. Seal Reflex é a biblioteca que habilita reflexão computacional nos códigos do CCF e CCAS. O procedimento de reflexão computacional Seal Reflex necessita de dicionários que contêm informações sobre as classes objetos de reflexão. No CCAS, a aplicação *ccasBuild* do módulo cliente é responsável por gerar os dicionários das classes que serão objetos de reflexão Seal Reflex. Nesse caso, os objetos de reflexão são componentes de aplicações CCF. O módulo de carregamento dinâmico do *ccasServer* utiliza os dicionários gerados no módulo cliente para criar os objetos dos componentes a partir dos nomes das classes utilizando a API de Seal Reflex. Os nomes das classes dos componentes são descritos no XML que contém a hierarquia das aplicações CCF. Os detalhes de implementação dos módulos cliente e servidor são apresentados a seguir.

5.2.1 CCAS cliente

A aplicação *ccasBuild*

O *ccasBuild* é responsável por preparar os componentes para implantação em uma aplicação CCF. Na Figura 5.5, o fluxo de execução da aplicação *ccasBuild* é apresentado. O *ccasBuild* recebe como parâmetro de entrada o arquivo *build.ccas* que contém os componentes que serão preparados para implantação e outras informações necessárias para o processo de preparação. O resultado da execução da aplicação *ccasBuild* é um conjunto de pares <biblioteca componente; dicionário componente> que são armazenados no diretório de saída. Após a execução do *ccasBuild*, os componentes estão prontos para implantação.

A aplicação *ccasBuild* é executada através da linha de comando. A sintaxe é apresentada a seguir.

- **ccasBuild <build-file>**, onde <build-file> é o caminho completo para o arquivo que contém os componentes que serão preparados para implantação e outras informações necessárias ao processo de preparação.

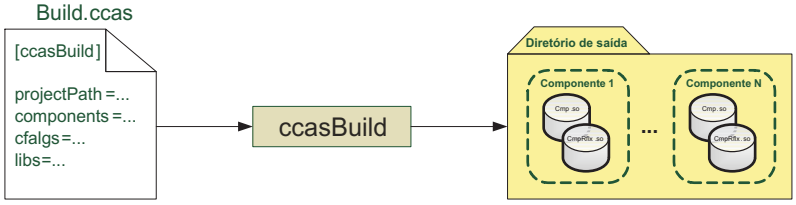

Figura 5.5: Fluxo de execução da aplicação *ccasBuild*.

A preparação de componentes para implantação em uma aplicação CCF é simples, porém trabalhoso. O procedimento utiliza mecanismos da biblioteca Seal Reflex e compilação de código. A ferramenta genreflex [33] de Seal Reflex cria os dicionários com as informações sobre os componentes CCF. Em seguida, as bibliotecas são geradas a partir dos dicionários criados pela ferramenta genreflex. Na Figura 5.6, o processo de preparação para implantação de um componente CCF é apresentado. Os passos são descritos a seguir.

1. A aplicação *ccasBuild* usa a ferramenta genreflex [33] para gerar o dicionário (*MeuComponenteRflx.cpp*) com as informações sobre a classe descrita no arquivo *MeuComponente.h*.

2. A aplicação *ccasBuild* gera a biblioteca compartilhada com as informações do dicionário (*MeuComponenteRflx.so*).

3. As bibliotecas do componente estão prontas para implantação ou evolução.

A aplicação *ccasDeploy*

O *ccasDeploy* é responsável por implantar uma aplicação CCF no módulo servidor. Na Figura 5.7, o fluxo de execução da aplicação *ccasDeploy* é apresentado. Os parâmetros de entrada são um arquivo XML e o caminho para um diretório. O arquivo XML (*application.xml*) descreve a hierarquia da aplicação CCF que será implantada. O diretório armazena os pares <biblioteca; dicionário> dos componentes que fazem parte da aplicação CCF. Quando o *ccasDeploy* é executado, os conteúdos do arquivo *application.xml* e das bibliotecas dos componentes CCF são enviados para o módulo servidor através de XML-RPC [1]. O *ccasDeploy* é executado pela linha de comando. A sintaxe é descrita a seguir.

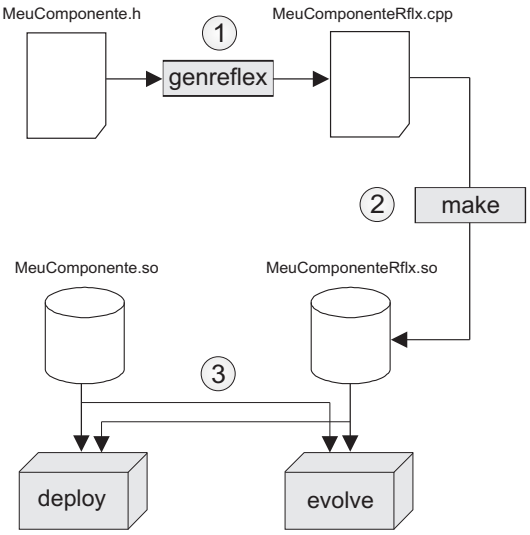

Figura 5.6: Preparação de um componente CCF para implantação.

- **ccasDeploy <app-hierarchy> <component-dir>**, onde <app-hierarchy> é o caminho completo para o arquivo XML que descreve a hierarquia da aplicação a ser implantada e <component-dir> é o caminho completo para o diretório onde se encontram os pares <biblioteca; dicionário> dos componentes CCF gerados pelo *ccasBuild*.

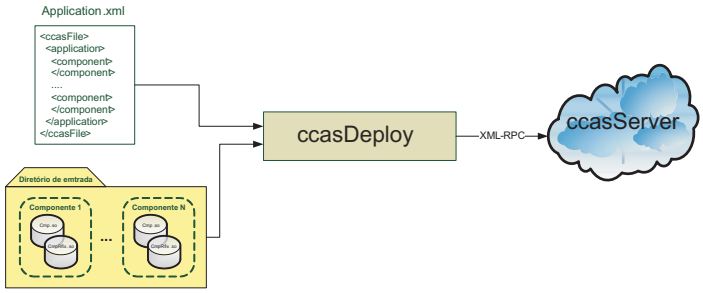

Figura 5.7: Fluxo de execução da aplicação *ccasDeploy*.

A aplicação *ccasEvolve*

O *ccasEvolve* é responsável por evoluir a hierarquia de uma aplicação CCF que já foi implantada no módulo servidor através da aplicação *ccasDeploy*. Na Figura 5.8, o fluxo de execução da aplicação *ccasEvolve* é apresentado. Os parâmetros de entrada são um arquivo XML, o caminho para um diretório e um arquivo de atualização. O arquivo XML (*application.xml*) descreve a hierarquia da aplicação CCF após a evolução. O diretório contém os pares <biblioteca; dicionário> dos componentes CCF que serão utilizados no processo de evolução. O arquivo de atualização (*script.ccas*) descreve os comandos de evolução que serão seguidos para atingir a configuração final da hierarquia. Quando

o *ccasEvolve* é executado, o conteúdo do arquivo *application.xml*, das bibliotecas dos componentes CCF e do arquivo *script.ccas* são enviados para o módulo servidor através de XML-RPC [1]. O *ccasEvolve* é executado pela linha de comando. A sintaxe é descrita a seguir.

- **ccasEvolve <app-hierarchy> <component-dir> <evolution-file>**,
  onde <app-hierarchy> é o caminho completo para o arquivo XML que descreve a hierarquia da aplicação CCF após evolução, <component-dir> é o caminho completo para o diretório onde se encontram os pares <biblioteca; dicionário> dos componentes CCF gerados pelo *ccasBuild* e <evolution-file> é caminho completo para o arquivo que contém os comandos de evolução.

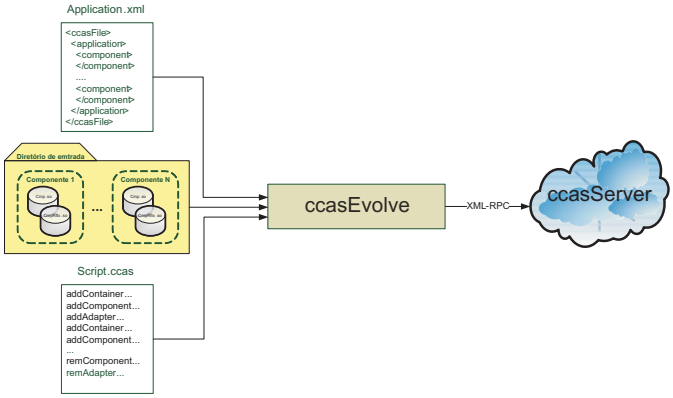

Figura 5.8: Fluxo de execução da aplicação *ccasEvolve*.

5.2.2 CCAS servidor

O *ccasServer* é responsável por gerenciar a implantação e a execução de aplicações CCF. O processo do *ccasServer* executa em modo *background* nas máquinas que receberão requisições de implantação ou evolução a partir do módulo cliente. A arquitetura do *ccasServer* é apresentada na Figura 5.9. Os detalhes de cada módulo são descritos a seguir.

- O módulo de rede implementa um servidor XML-RPC [1] para receber as requisições das aplicações *ccasDeploy* e *ccasEvolve* do módulo cliente.

- O módulo de carregamento dinâmico utiliza a API de reflexão Seal Reflex para carregar os componentes CCF. Em caso de implantação, após carregar os componentes CCF, o módulo de carregamento dinâmico cria uma nova instância do arcabouço CCF para montar a hierarquia descrita no XML. Essa instância, ou seja, a nova aplicação CCF, é iniciada em uma nova *thread* através da API Pthreads [28]. A nova

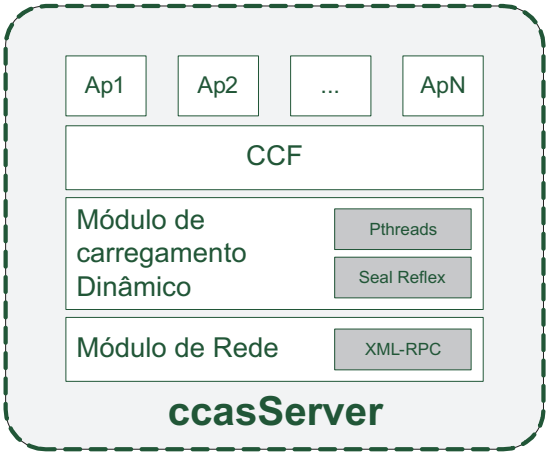

Figura 5.9: Arquitetura do *ccasServer*.

thread é destruída quando o usuário parar a execução da aplicação CCF utilizando a GUI do *ccasServer* ou quando *ccasServer* for finalizado.

- o CCF é o arcabouço utilizado pelo módulo de carregamento dinâmico para compor as aplicações no *ccasServer*.

A interface gráfica do *ccasServer* disponibiliza uma lista das aplicações CCF implantadas. A tela principal do *ccasServer* é ilustrada na Figura 5.10. As únicas operações que podem ser realizadas sobre as aplicações CCF a partir da interface gráfica do *ccasServer* são: iniciar e parar. Qualquer outra operação, seja ela de implantação ou de evolução, deve ser realizada através das aplicações do módulo cliente.

Figura 5.10: Tela principal do *ccasServer*.

O diagrama de classes simplificado do *ccasServer* é ilustrado na Figura 5.11. O *ccasServer* possui duas interfaces de comunicação com o módulo cliente, uma para implantação (`Deploy`) e outra para evolução (`Evolve`). Quando a aplicação *ccasDeploy* é executada, a requisição para implantar uma aplicação CCF chega ao método `execute` da classe `Deploy`. O método `execute` cria uma nova aplicação CCF (`ComporApp`) e invoca o método `parser` da classe `DeployParser`. O `DeployParser` analisa o XML e monta a aplicação CCF (`ComporApp`) de acordo com a especificação. A aplicação CCF (`ComporApp`)

montada é inserida no gerenciador de aplicações CCF (`AppManager`). Quando a aplicação *ccasEvolve* é executada, a requisição para evoluir uma aplicação CCF chega ao método `execute` da classe `Evolve`. O método `execute` recupera a aplicação (`ComporApp`) do gerenciador de aplicações CCF (`AppManager`) e invoca o método `parser` da classe `EvolveParser`. O `EvolveParser` analisa os comandos de evolução e os executa sobre a aplicação CCF (`ComporApp`) seguindo a ordem definida no arquivo.

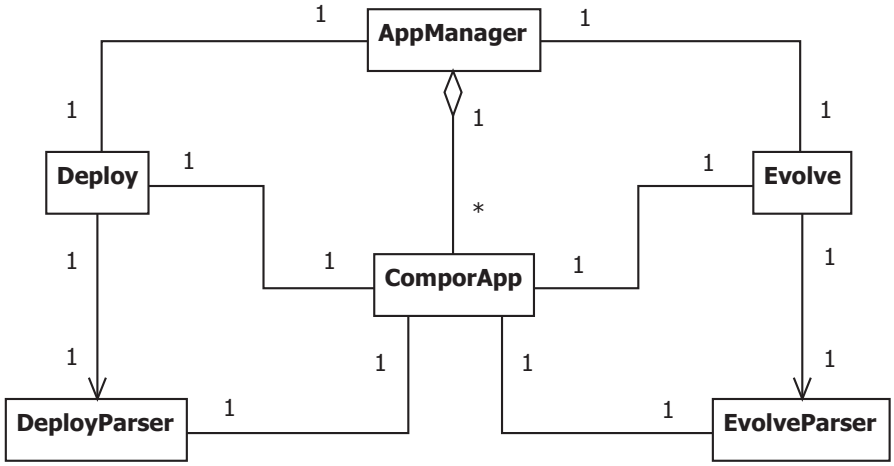

Figura 5.11: Diagrama de classes simplificado do *ccasServer*.

A aplicação *ccasServer* oferece toda a infra-estrutura para gerenciar a execução das aplicações desenvolvidas utilizando o arcabouço CCF. As aplicações *ccasBuild*, *ccasDeploy* e *ccasEvolve* fornecem os mecanismos necessários para preparar, implantar e evoluir as aplicações CCF. Juntas, as aplicações dos módulos cliente e servidor disponibilizam um ambiente completo para gerenciar a implantação e a execução de aplicações desenvolvidas utilizando o arcabouço CCF.

Capítulo 6

Estudo de caso: *DjVu Compactor*

A aplicação denominada *DjVu Compactor* foi desenvolvida utilizando a infra-estrutura disponibilizada pelo CCF e pelo CCAS para fornecer mecanismos de carregamento e descarregamento de algoritmos de compactação e descompactação DjVu sob demanda. DjVu é um formato de compressão de arquivos cujas especificações estão disponíveis sob a licença GPL. O formato foi desenvolvido pelos laboratórios AT&T [22] e, atualmente, uma implementação de código aberto de parte dos algoritmos de compactação e descompactação está disponível no projeto DjVuLibre [10].

6.1 Implementação do *DjVu Compactor*

O *DjVu Compactor* encapsula cada algoritmo implementado no projeto DjVuLibre em um componente CCF possibilitando o carregamento sob demanda de algoritmos DjVu. O projeto DjVuLibre disponibiliza diversas implementações de algoritmos de compactação e descompactação DjVu. O *bzz* e o *cjb2* são dois algoritmos implementados no DjVuLibre. O *bzz* é um compactador e descompactador de propósito geral, similar ao *bzip2*. O *cjb2* é um compactador de imagens preto e branco. A idéia do *DjVu Compactor* é disponibilizar uma infra-estrutura onde o usuário possa carregar apenas os algoritmos DjVu necessários no momento, tendo em vista que nem todos algoritmos são utilizados ao mesmo tempo. O formato DjVu foi escolhido por apresentar uma ótima taxa de compressão e por sua eficiência de transmissão. Se comparado ao formato PDF, os arquivos DjVu são em torno de 3 a 8 vezes menores. Além disso, o usuário não precisa esperar o arquivo DjVu baixar por completo para visualizá-lo, pois apenas a parte do arquivo que esta sendo exibida na tela é necessária.

A hierarquia de componentes da aplicação *DjVu Compactor* é apresentada na Figura 6.1. O *CtRaiz* é o contêiner raíz do *DjVu Compactor*. Os componentes funcionais que implementam os algoritmos DjVu, *Bzz* e *Cjb2*, são armazenados no contêiner *CtAlg*. O

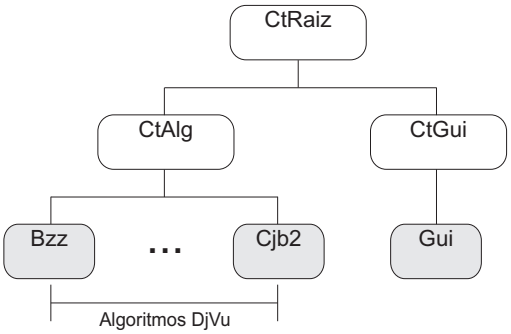

Figura 6.1: Hierarquia de componentes do *DjVu Compactor*.

contêiner *CtGui* armazena o componente funcional que implementa a interface gráfica do *DjVu Compactor*: componente *Gui*. É importante observar que o componente *Gui* também pode evoluir. A interface gráfica do *DjVu Compactor* (componente *Gui*) é implementada utilizando GTK [30]. A tela principal do *DjVu Compactor* é apresentada na Figura 6.2. O usuário pode escolher o algoritmo DjVu a ser utilizado a partir da lista de algoritmos disponíveis passando como parâmetros o caminho completo dos arquivos de entrada e saída.

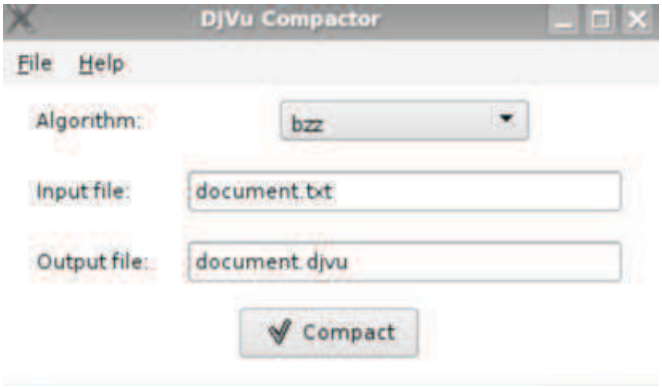

Figura 6.2: Tela principal do *DjVu Compactor*.

O componente *Bzz* disponibiliza o serviço *bzz* que recebe como parâmetros o caminho completo dos arquivos de entrada e saída. Os eventos *addAlgorithm* e *removeAlgorithm* são anunciados quando o componente *Bzz* é iniciado e finalizado, respectivamente. O componente *Cjb2* disponibiliza o serviço *cjb2* que recebe como parâmetros o caminho completo dos arquivos de entrada e saída. Os eventos *addAlgorithm* e *removeAlgorithm* são anunciados quando o componente *Cjb2* é iniciado e finalizado, respectivamente. A inicialização do componente *Gui* verifica quais os algoritmos DjVu estão disponíveis na hierarquia. Para manter a lista de algoritmos DjVu disponíveis atualizada, o componente *Gui* recebe os anúncios dos eventos *addAlgorithm* e *removeAlgorithm*. A disponibilização do componente *Bzz* é apresentada na Figura 6.3. Quando o componente *Bzz* é adicionado,

o evento *addAlgorithm* é anunciado para informar o componente *Gui* sobre o novo algoritmo *bzz*. O componente *Gui* recebe o anúncio *addAlgorithm* do componente *Bzz* e adiciona o algoritmo *bzz* a lista de algoritmos disponíveis. O evento *removeAlgorithm* anunciado quando o componente *Bzz* é removido da hierarquia ocorre da mesma maneira que o evento *addAlgorithm*.

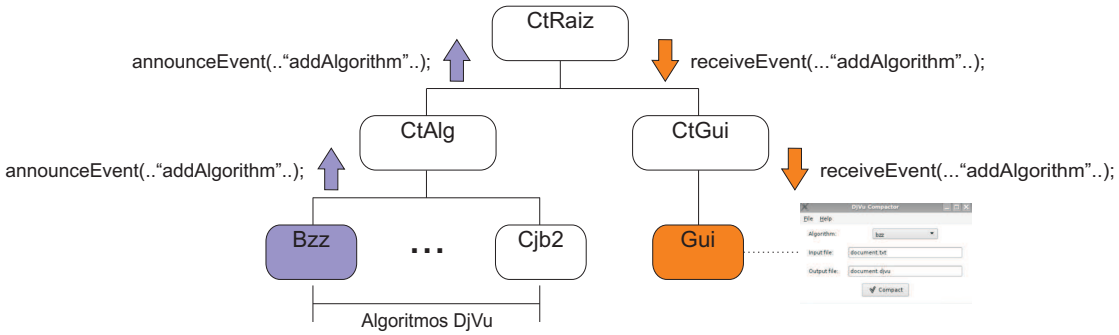

Figura 6.3: Disponibilização do componente *Bzz*.

O componente *Gui* invoca os algoritmos DjVu através da requisição de serviços. A invocação do algoritmo *bzz* é apresentada na Figura 6.4. O método `doIt` encaminha a requisição do serviço *bzz*, no sentido *bottom-up*, até encontrar o contêiner que armazena a referência para o algoritmo *bzz*. A partir daí, o método `receiveRequest` é invocado para repassar a requisição, no sentido *top-down*, ao componente *Bzz* que implementa o algoritmo *bzz*.

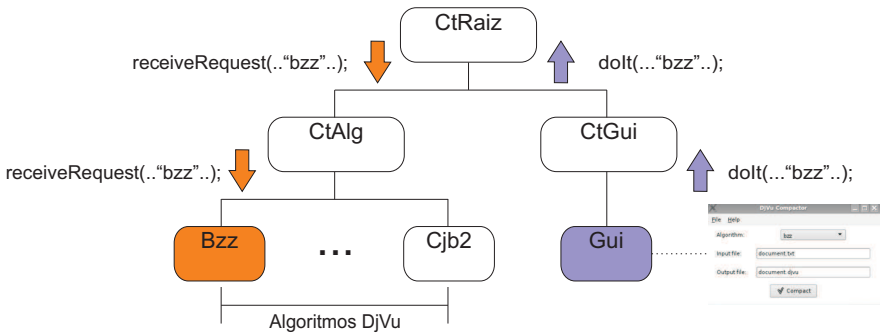

Figura 6.4: Invocação do algoritmo *bzz*.

6.1.1 Utilização do CCF

Os algoritmos DjVu, assim como, a interface gráfica do *DjVu Compactor* são encapsulados em componentes funcionais CCF. A seguir são apresentados os códigos fontes dos componentes `ComponentBzz`, `ComponentCjb2` e `ComponentGUI` que representam o algoritmo Bzz, o algoritmo Cjb2 e a interface gráfica do *DjVu Compactor*, respectivamente.

1. Componente *Bzz*

   A classe `ComponentBzz` que implementa o algoritmo *Bzz* estende de `FunctionalComponent` (Listagem de Código 6.1). O construtor da classe `ComponentBzz` adiciona o serviço provido *bzz* e os eventos anunciados *addAlgorithm* e *removeAlgorithm* através dos métodos `addProvidedServices` e `addAnnouncedEvents`. O serviço provido *bzz* é implementado pelo método `bzz` que utiliza o algoritmo *Bzz* para compactar ou descompactar o arquivo passado como parâmetro. Os métodos `startImpl` e `stopImpl` anunciam os eventos *addAlgorithm* e *removeAlgorithm* quando o componente *Bzz* é iniciado e finalizado, respectivamente.

Código 6.1: Componente *Bzz*.

```
#define ADD_ALG "addAlgorithm"
#define REM_ALG "removeAlgorithm"
#define ALG "bzz"

class ComponentBzz : public FunctionalComponent {
public:
    inline ComponentBzz() : FunctionalComponent("ComponentBzz") {
        addProvidedServices();
        addAnnouncedEvents();
    }
    inline virtual ~ComponentBzz() {}
    inline void bzz(const string & _infile, const string & _outfile){
        bzzDjvu(_infile, _outfile);
    }
    inline void startImpl() {
        vector <void *> param;
        param.push_back(new string(ALG));
        EventAnnouncement *event = new EventAnnouncement(ADD_ALG, param);
        announceEvent(event);
    }
    inline void stopImpl() {
        vector <void *> param;
        param.push_back(new string(ALG));
        EventAnnouncement *event = new EventAnnouncement(REM_ALG, param);
        announceEvent(event);
    }

private:
    inline void addProvidedServices() {
        Type type = Type::ByName("ComponentBzz");
        Member method1 = type.MemberByName(ALG);
        ServiceSpecification *spec1 = new ServiceSpecification(ALG);
        ProvidedService *service1 = new ProvidedService(spec1, method1);
        addProvidedService(service1);
    }
    inline void addAnnouncedEvents() {
        EventSpecification *spec1 = new EventSpecification(ADD_ALG),
                           *spec2 = new EventSpecification(REM_ALG);
        AnnouncedEvent *announcedEvent1 = new AnnouncedEvent(spec1),
                       *announcedEvent2 = new AnnouncedEvent(spec2);
        addAnnouncedEvent(announcedEvent1);
        addAnnouncedEvent(announcedEvent2);
    }
};
```

2. Componente *Cjb2*

   A classe `ComponentCjb2` que implementa o algoritmo *Cjb2* estende de `Customizable-Component` (Listagem de Código 6.3). Assim, o componente *Cjb2* pode ser personalizado por comportamentos que devem estender da classe `Concern`. O comportamento implementado pela classe `GenericConcern`, apresentado na Listagem de Código 6.2, é utilizado para personalizar o componente *Cjb2*. A classe `GenericCon-`

`cern` implementa o comportamento de *Log* onde as mensagens de *Log* são impressas antes e depois da execução do serviço *cjb2* através dos métodos `interceptBefore` e `interceptAfter`. Caso ocorra uma exceção na execução do serviço *cjb2*, o *Log* de exceção é impresso pelo método `interceptOnException`.

Código 6.2: Comportamento *Cjb2*.

```
1  class GenericConcern : public Concern {
2  public:
3      inline GenericConcern(const string & _concernName) :
4          Concern(_concernName) {}
5          inline virtual ~GenericConcern() {}
6      inline void interceptAfter(CustomizableComponent * _component,
7                  ServiceRequest & _request, ServiceResponse * _response) {
8          cout << _component->getName() << ":After" << endl;
9      }
10     inline void interceptBefore(CustomizableComponent * _component,
11                                 ServiceRequest & _request) {
12         cout << _component->getName() << ":Before" << endl;
13     }
14     inline void interceptOnException(CustomizableComponent * _component,
15                 ServiceRequest & _request, ServiceResponse * _response) {
16         cout << _component->getName() << ":Exception" << endl;
17     }
18 };
```

O construtor da classe `ComponentCjb2` adiciona o serviço provido *cjb2*, os eventos anunciados *addAlgorithm* e *removeAlgorithm* e o comportamento de *Log* através dos métodos `addProvidedServices`, `addAnnouncedEvents` e `addConcerns`, respectivamente. O serviço *cjb2* é implementado pelo método `cjb2` que utiliza o algoritmo *Cjb2* para compactar a imagem passada como parâmetro. Os métodos `startImpl` e `stopImpl` anunciam os eventos *addAlgorithm* e *removeAlgorithm* quando o componente *Cjb2* é iniciado e finalizado, respectivamente.

Código 6.3: Componente *Cjb2*.

```
#define ADD_ALG "addAlgorithm"
#define REM_ALG "removeAlgorithm"
#define ALG "cjb2"

class ComponentCjb2 : public CustomizableComponent {
public:
        inline ComponentCjb2() {
        addProvidedServices();
        addAnnouncedEvents();
        addConcerns();
    }
    inline virtual ~ComponentCjb2() {}
    inline void cjb2(const string & _infile, const string & _outfile) {
        cjb2Djvu(_infile.c_str(), _outfile.c_str(), 200);
    }
    inline void startImpl() {
        vector <void *> param;
        param.push_back(new string(ALG));
        EventAnnouncement *event = new EventAnnouncement(ADD_ALG, param);
        announceEvent(event);
    }
    inline void stopImpl() {
        vector <void *> param;
        param.push_back(new string(ALG));
        EventAnnouncement *event = new EventAnnouncement(REM_ALG, param);
        announceEvent(event);
    }

private:
    inline void addProvidedServices() {
        Type type = Type::ByName("ComponentCjb2");
        Member method1 = type.MemberByName(ALG);
        ServiceSpecification *spec1 = new ServiceSpecification(ALG);
        ProvidedService *service1 = new ProvidedService(spec1, method1);
        addProvidedService(service1);
    }
    inline void addAnnouncedEvents() {
        EventSpecification *spec1 = new EventSpecification(ADD_ALG),
                           *spec2 = new EventSpecification(REM_ALG);
        AnnouncedEvent *announcedEvent1 = new AnnouncedEvent(spec1),
                       *announcedEvent2 = new AnnouncedEvent(spec2);
        addAnnouncedEvent(announcedEvent1);
        addAnnouncedEvent(announcedEvent2);
    }
    inline void addConcerns() {
        GenericConcern *log = new GenericConcern("LOG");
        addConcern(log, true);
    }
};
```

3. Componente *Gui*

A classe `ComponentGUI` que implementa a interface gráfica do *DjVu Compactor* estende de `FunctionalComponent` (Listagem de Código 6.4). O construtor da classe

`ComponentGUI` adiciona os eventos de interesse *addAlgorithm* e *removeAlgorithm* através do método `addInterestEvents`. A tela principal do *DjVu Compactor* é criada quando o componente *Gui* é iniciado (método `startImpl`) e destruída quando o componente *Gui* é finalizado (método `stopImpl`). O evento *addAlgorithm* é implementado pelo método `addAlgorithm` que adiciona o algoritmo anunciado à lista de algoritmos disponíveis da tela principal do *DjVu Compactor*. O método `addAlgorithm` também adiciona o algoritmo anunciado à lista de serviços requeridos (método `addRequiredServices`) do componente *Gui*. Assim, o componente *Gui* pode requisitar o serviço que implementa o novo algoritmo DjVu. O evento *removeAlgorithm* é implementado pelo método `removeAlgorithm` que remove o algoritmo DjVu da lista de algoritmos disponíveis da tela principal do *DjVu Compactor*.

Código 6.4: Componente *Gui*.

```
#define ADD_ALG "addAlgorithm"
#define REM_ALG "removeAlgorithm"

class ComponentGUI : public FunctionalComponent {
public:
        inline ComponentGUI() : FunctionalComponent("ComponentGUI") {
        addInterestEvents();
    }
        inline virtual ~ComponentGUI() {}
        inline virtual void startImpl() { startMainScreen(); }
    inline virtual void stopImpl() { stopMainScreen(); }
    inline void addAlgorithm(const string &_algorithm) {
        addInternalAlgorithm(_algorithm, true);
    }
    inline void removeAlgorithm(const string &_algorithm) {
            removeInternalAlgorithm(_algorithm);
    }

private:
    inline void addInterestEvents() {
        EventSpecification *spec1 = new EventSpecification(ADD_ALG),
                           *spec2 = new EventSpecification(REM_ALG);
        EventOfInterest *event1 = new EventOfInterest(spec1, method1),
                        *event2 = new EventOfInterest(spec2, method2);
        Type type = Type::ByName("ComponentGUI");
        Member method1 = type.MemberByName(ADD_ALG);
        Member method2 = type.MemberByName(REM_ALG);
        addEventOfInterest(event1);
        addEventOfInterest(event2);
    }
        inline void addRequiredServices(const string &_service) {
        ServiceSpecification *spec = new ServiceSpecification(_service);
        RequiredService *service = new RequiredService(spec);
        addRequiredService(service);
    }
};
```

6.1.2 Utilização do CCAS

A infra-estrutura fornecida pelo CCAS é utilizada para gerenciar a implantação e a execução do *DjVu Compactor*. A seguir são apresentados os passos para preparar, implantar e evoluir o *DjVu Compactor*.

Preparação

Os componentes do *DjVu Compactor* precisam ser preparados antes de serem implantados no CCAS. A preparação dos componentes do *DjVu Compactor* é realizada através da aplicação *ccasBuild* utilizando a linha de comando a seguir.

ccasBuild djvuCompactor.bld

O conteúdo do arquivo *djvuCompactor.bld* é apresentado na Listagem de Código 6.5. O caminho completo para o diretório raíz que contém o código fonte do *DjVu Compactor* é armazenado na chave *projectPath*. A chave *components* contém os caminhos relativos ao diretório raíz dos pares <arquivo de cabeçalho; biblioteca> dos componentes *Bzz*, *Cjb2* e *Gui* separados por ponto e vírgula. A aplicação *ccasBuild* utiliza as informações contidas no arquivo *djvuCompactor.bld* para preparar os componentes do *DjVu Compactor* e armazená-los no diretório de saída *$HOME/ccas/*.

Código 6.5: Arquivo de entrada da aplicação *ccasBuild* do *DjVu Compactor*.

```
1 [ccasBuild]
2
3 projectPath=${HOME}/djvuCompactor/
4 components=src/components/Bzz/Bzz.h;src/components/Bzz/.libs/Bzz.so;
5              src/components/Cjb2/Cjb2.h;src/components/Cjb2/.libs/Cjb2.so;
6              src/components/Gui/Gui.h;src/components/Gui/.libs/Gui.so
```

Implantação

Uma vez preparados, os componentes do *DjVu Compactor* podem ser implantados no CCAS. Neste exemplo, o *DjVu Compactor* é implantado de acordo com a hierarquia apresentada na Figura 6.5. A hierarquia inicial do *DjVu Compactor* apresenta o contêiner raíz (*CtRaiz*), o contêiner de algoritmos DjVu (*CtAlg*) que, por enquanto, contém apenas o componente *Bzz* e o contêiner de interface gráfica (*CtGui*) que armazena o componente *Gui*.

A implantação do *DjVu Compactor* é realizada através da aplicação *ccasDeploy* utilizando a linha de comando a seguir.

ccasDeploy djvuCompactor.xml $HOME/ccas/

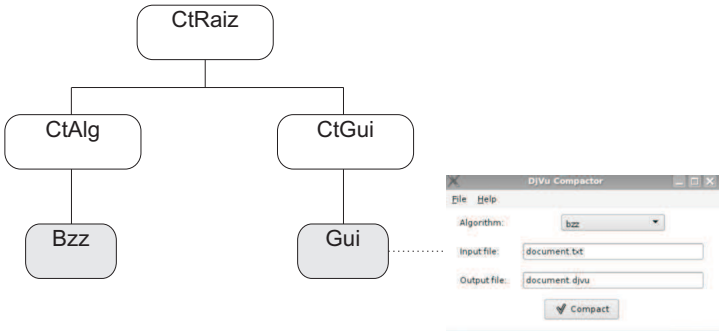

Figura 6.5: Hierarquia inicial do *DjVu Compactor*.

O arquivo *djvuCompactor.xml* é apresentado na Listagem de Código 6.6. O endereço de rede do *ccasServer* é armazenado na *tag serverUrl*. A *tag application* define a aplicação *DjVu Compactor*. O contêiner raíz *CtRaiz* é definido pela *tag rootContainer*. Os filhos *CtAlg* e *CtGui* do contêiner raíz são definidos pela *tag container*. Os contêineres *CtAlg* e *CtGui* armazenam os componentes *Bzz* e *Gui*, respectivamente. A *tag component* define os componentes *Bzz* e *Gui*. O conteúdo do arquivo *djvuCompactor.xml*, assim como, as bibliotecas dos componentes *Bzz* e *Gui* armazenadas no diretório *$HOME/ccas/* são enviados para o *ccasServer*. O *ccasServer* utiliza as informações contidas no arquivo *djvuCompactor.xml* e as bibliotecas dos componentes *Bzz* e *Gui* para montar e iniciar a aplicação *DjVu Compactor*.

Código 6.6: XML que descreve a hierarquia do *DjVu Compactor*.

```
1  <ccasFile>
2     <serverUrl>http://localhost:8080</serverUrl>
3     <application name="DjVuCompactor">
4       <rootContainer name="CtRaiz">
5         <container name="CtAlg">
6           <component name="Bzz" lib="Bzz.so"
7             dictionary="BzzDict.so"></component>
8         </container>
9         <container name="CtGui">
10          <component name="Gui" lib="Gui.so"
11            dictionary="GuiDict.so"></component>
12        </container>
13      </rootContainer>
14    </application>
15 </ccasFile>
```

Evolução

A hierarquia implantada do *DjVu Compactor* pode ser evoluída em qualquer parte e a qualquer momento. Neste exemplo, o *DjVu Compactor* é evoluído de acordo com a hierarquia apresentada na Figura 6.6. O componente *Bzz* é removido do contêiner de algoritmos e o componente *Cjb2* é adicionado ao contêiner de algoritmos. A evolução é

realizada através da aplicação *ccasEvolve* utilizando a linha de comando a seguir.

ccasEvolve djvuCompactor.xml $HOME/ccas/ djvuEvolution.cmd

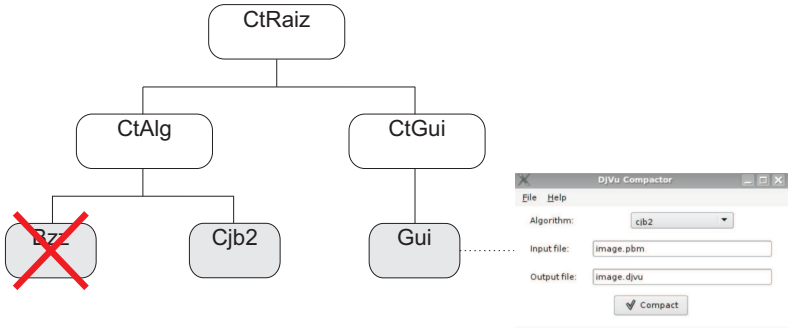

Figura 6.6: Evolução da hierarquia do *DjVu Compactor*.

O conteúdo do arquivo *djvuEvolution.cmd* é apresentado na Listagem de Código 6.7. O arquivo *djvuEvolution.cmd* contém os comandos que devem ser executados pelo *ccasServer* para evoluir a aplicação *DjVu Compactor*. Os comandos são executados na ordem definida no arquivo. O componente *Bzz* é removido do contêiner de algoritmos através do comando *remComponent*. Observe que o caminho completo do componente *Bzz* na hierarquia (*CtRaiz/CtAlg/Bzz*) é passado como parâmetro do comando *remComponent*. O comando *addCustomComponent* adiciona o componente *Cjb2* ao contêiner de algoritmos. *Cjb2* é um componente funcional personalizável, por isso, o comando *addCustomComponent* é utilizado. Por fim, o comportamento de *Log* do componente *Cjb2* é desativado através do comando *deactivateConcern*.

Código 6.7: Comandos de evolução do *DjVu Compactor*.

```
1 remComponent CtRaiz/CtAlg/Bzz
2 addCustomComponent CtRaiz/CtAlg/Cjb2
3 deactivateConcern CtRaiz/CtAlg/Cjb2 LOG
```

O arquivo *djvuCompactor.xml* apresentado na Listagem de Código 6.8 descreve o XML da hierarquia do *DjVu Compactor* após a evolução. O XML permanece o mesmo exceto a hierarquia sob o contêiner de algoritmos *CtAlg*. O componente *Bzz* é substituído pelo componente *Cjb2*. A *tag customizableComponent* é utilizada para representar o componente funcional personalizável *Cjb2*. O conteúdo dos arquivos *djvuCompactor.xml* e *djvuCompactor.cmd*, assim como, a biblioteca do componente *Cjb2* armazenada no diretório *$HOME/ccas/* são enviados para o *ccasServer*. O *ccasServer* utiliza as informações contidas nos arquivos *djvuCompactor.xml* e *djvuCompactor.cmd* e a biblioteca do componente *Cjb2* para evoluir a aplicação *DjVu Compactor*.

Código 6.8: XML que descreve a hierarquia do *DjVu Compactor* após a evolução.

```
<ccasFile>
   <serverUrl>http://localhost:8080</serverUrl>
   <application name="DjVuCompactor">
     <rootContainer name="CtRaiz">
       <container name="CtAlg">
         <customizableComponent name="Cjb2" lib="Cjb2.so"
           dictionary="Cjb2Dict.so"></customizableComponent>
       </container>
       <container name="CtGui">
         <component name="Gui" lib="Gui.so"
           dictionary="GuiDict.so"></component>
       </container>
     </rootContainer>
   </application>
</ccasFile>
```

Após a evolução, a hierarquia do *DjVu Compactor* apresenta apenas um algoritmo DjVu: o *Cjb2*. A interface gráfica do *DjVu Compactor* permanece a mesma, tendo em vista que o componente *Gui* não foi atualizado. É importante ressaltar que qualquer componente da hierarquia do *DjVu Compactor*, inclusive o componente *Gui*, pode ser atualizado em tempo de execução. Além disso, novos componentes DjVu podem ser desenvolvidos e adicionados sempre que necessário.

Capítulo 7

Considerações finais

A evolução de software tem se caracterizado pelo seu alto custo tanto em termos financeiros quanto em termos de tempo de desenvolvimento. O impacto da evolução sobre o código fonte e a arquitetura da aplicação é mais significativo quando as mudanças nos requisitos do software não foram previstas ou antecipadas. O processo de evolução se torna ainda mais complexo em domínios onde as aplicações, devido a razões financeiras ou de segurança, precisam evoluir de forma dinâmica, ou seja, sem interromper a execução.

Neste trabalho foi apresentado um ambiente C++ para o desenvolvimento de software com suporte à evolução dinâmica não antecipada. O ambiente apresentado é composto por um arcabouço C++ para desenvolvimento de software baseado em componentes com suporte à evolução dinâmica não antecipada denominado CCF e por um servidor de aplicação C++ denominado CCAS. O arcabouço CCF implementa a especificação do modelo de componentes COMPOR (CMS), fornecendo assim, uma API para o desenvolvimento de aplicações C++ com suporte a mudanças não antecipadas, mesmo em tempo de execução. O servidor CCAS é responsável por gerenciar a implantação e a execução das aplicações desenvolvidas sobre o CCF. Juntos, o CCF e o CCAS fornecem um ambiente C++ para o desenvolvimento de software com suporte à evolução dinâmica não antecipada totalmente transparente para o desenvolvedor.

A partir da disponibilização do ambiente C++ apresentado neste trabalho, as aplicações C++ que possuem o requisito de alta disponibilidade terão o tempo de desenvolvimento 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. Para implementar as aplicações, o desenvolvedor precisa utilizar apenas a linguagem de programação C++ que é largamente utilizada no mundo inteiro. C++ é uma linguagem compilada de alto desempenho cujo o consumo de memória é baixo se comparada a linguagens interpretadas, pois não há a necessidade de máquinas virtuais ou interpretadores para executar o

código.

Para validar o ambiente C++ introduzido, uma aplicação para compactar e descompactar arquivos DjVu denominada *DjVu Compactor* foi implementada. O *DjVu Compactor* utiliza a infra-estrutura disponibilizada no ambiente C++ apresentado neste trabalho para possibilitar o carregamento sob-demanda de algoritmos de compactação e descompactação DjVu inclusive em tempo de execução. Dessa maneira, o usuário pode carregar apenas os algoritmos DjVu necessários no momento, tendo em vista que nem todos algoritmos são utilizados ao mesmo tempo.

O ambiente C++ apresentado neste trabalho fornece mecanismos para implementar aplicações de alto desempenho com suporte à evolução dinâmica não antecipada sem desperdício de memória. Esses requisitos são de grande importância em domínios tais como o de sistemas embarcados. Dentro desse contexto, o CCF e o CCAS foram portados para a plataforma Maemo [26]. Maemo é uma plataforma aberta baseada em Linux que está disponível em *internet tablets* da Nokia, como o Nokia N810, Nokia N800 e Nokia 770. Com o porte do CCF e do CCAS para o Maemo, é possível desenvolver outras aplicações para tal plataforma com suporte à evolução dinâmica não antecipada.

No que se refere a trabalhos futuros, o ambiente C++ detalhado neste trabalho será utilizado no desenvolvimento de aplicações que possuem o requisito de alta disponibilidade para plataforma Maemo, mais especificamente, serão desenvolvidas aplicações no domínio da computação pervasiva. Uma dessas aplicações será um *middleware* C++ baseado em componentes para fornecer serviços em ambientes pervasivos. Além disso, serão adicionados aspectos de segurança e integridade à implementação do ambiente C++ proposto.

Referências Bibliográficas

[1] *XML-RPC*. http://www.xmlrpc.com - Acessado em 23/07/2008.

[2] OSGi Alliance. Open services gateway initiative. http://www.osgi.org - Acessado em 23/07/2008.

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

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

[5] Felix Bachmann, Len Bass, , Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume ii: Technical concepts of component-based software engineering, 2nd edition. Technical report, SEI Joint Program Office, 2000.

[6] Israel Ben-Shaul, Ophir Holder, and Boris Lavva. Dynamic adaptation and deployment of distributed components in hadas. *IEEE Transactions on Software Engineering*, 27(9):769–787, 2001.

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

[8] Alan W. Brown. *Large-Scale, Component Based Development*. Prentice Hall, 2000.

[9] Walter Cazzola, Robert J. Stroud, and Francesco Tisato. *Reflection and Software Engineering*. Springer, 2000.

[10] DjVuLibre. An open source implementation of djvu. http://djvu.sourceforge.net - Acessado em 23/07/2008.

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

[12] Peter Ebraert, Yves Vandewoude, Theo D'Hondt, and Yolande Berbers. Pitfalls in unanticipated dynamic software evolution. *Proc. of Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05)*, páginas 41–49, Glasgow, Scotland, 2005.

[13] Thomas Erl. *Service-Oriented Architecture(SOA): Concepts, Technology, and Design.* Prentice Hall, 2005.

[14] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[15] Research Institute for Software Evolution. Software evolution group. http://www.dur.ac.uk/RISE - Acessado em 23/07/2008.

[16] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley, 1999.

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

[18] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley, 2001.

[19] M. Hicks. *Dynamic Software Updating.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2001.

[20] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[21] Gunter Kniesel, Joost Noppen, Tom Mens, and Jim Buckley. *Unanticipated Software Evolution.* Springer Berlin / Heidelberg, 2002.

[22] AT&T Labs. http://www.research.att.com - Acessado em 23/07/2008.

[23] Meir M. Lehman and Juan F. Ramil. Software evolution and software evolution processes. *Ann. Softw. Eng.*, 14(1-4):275–309, 2002.

[24] Bennet P. Lientz and E. Burton Swanson. *Software Maintenance Management.* Addison-Wesley, 1980.

[25] Hua Liu and Manish Parashar. Accord: a programming framework for autonomic applications. *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, volume 36, páginas 341–352. IEEE Computer Society, 2006.

[26] Maemo. A linux-based development platform for handheld devices. http://maemo.org - Acessado em 23/07/2008.

[27] Johannes Mayer, Ingo Melzer, and Franz Schweiggert. Lightweight plug-in-based application development. *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, páginas 87–102, London, UK, 2003. Springer-Verlag.

[28] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly & Associates, 1996.

[29] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. *ICSE '98: Proceedings of the 20th international conference on Software engineering*, páginas 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[30] Havoc Pennington. *GTK+ /Gnome Application Development*. Sams, 1999.

[31] Andre Rodrigues, Hyggo Almeida, and Angelo Perkusich. A c++ environment for dynamic unanticipated software evolution. *The 23nd Annual ACM Symposium on Applied Computing*, Fortaleza, Ceará, Brazil, 2008. ACM Symposium on Applied Computing.

[32] Andre Rodrigues, Hyggo Oliveira de Almeida, and Angelo Perkusich. A c++ framework for developing component based software supporting dynamic unanticipated evolution. *The Nineteenth International Conference on Software Engineering and Knowledge Engineering*, páginas 326–331, Boston, MA, USA, 2007. Knowledge Systems Institute Graduate School.

[33] S. Roiser. The SEAL C++ Reflection System. *Computing in High Energy and Nuclear Physics (CHEP) Conference*, Interlaken, Switzerland, 2004.

[34] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 2001.

[35] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, 1993.

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

[37] John Vlissides, James Coplien, and Norman Kerth. *Pattern Languages of Program Design 2.* Addison-Wesley, 1996.

[38] Lizhou Yu, Gholamali C. Shoja, Hausi A. Müller, and Anand Srinivasan. A framework for live software upgrade. *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, páginas 149–158, Washington, DC, USA, 2002. IEEE Computer Society.