# Infra-estrutura Baseada em Componentes para o Desenvolvimento de Software com Suporte à Evolução Dinâmica Não Antecipada

Hyggo Oliveira de Almeida

Tese de Doutorado submetida ao Programa de Pós-Gradução em Engenharia Elétrica da Universidade Federal de Campina Grande - Campus de Campina Grande como parte dos requisitos necessários para obtenção do grau de Doutor em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Engenharia da Computação

Angelo Perkusich, D.Sc.

Orientador

Campina Grande, Paraíba, Brasil

# Infra-estrutura Baseada em Componentes para o Desenvolvimento de Software com Suporte à Evolução Dinâmica Não Antecipada

Hyggo Oliveira de Almeida

*Tese de Doutorado apresentada em outubro de 2007*

Angelo Perkusich, D.Sc.

Orientador

Jorge Cesar Abrantes de Figueiredo, D.Sc.

Presidente da banca

Carlos José Pereira de Lucena, Ph.D.

Examinador

Julio Cesar Sampaio do Prado Leite, Ph.D.

Examinador

Claudia Maria Lima Werner, D.Sc.

Examinador

Dalton Dario Serey Guerrero, D.Sc.

Examinador

Campina Grande, Paraíba, Brasil, outubro de 2007

*À Clarinha*

*“Com talento e um pouco de sorte e saúde,*

*meu véio, eu tiro onda... não vivo à sombra!”*

Jorge Aragão

# Agradecimentos

A Deus, por sempre me proporcionar mais do que realmente mereço.

Aos meus pais e irmão, por me apoiarem em todos os momentos da minha vida, sendo o doutorado apenas mais um destes momentos.

Ao meu orientador e amigo Angelo, pela liberdade que me foi dada para expor e colocar em prática as minhas idéias, pelo incentivo e apoio técnico na realização deste trabalho e pelas proveitosas conversas que tornaram válido, verdadeiramente, o doutorado.

Aos meus colegas do Laboratório Embedded, pela colaboração incondicional nos diversos aspectos deste trabalho. Em especial, a Galuteta e Memesso.

Aos meus colegas do Departamento de Sistemas e Computação pela paciência e compreensão em relação à conclusão desta tese.

Ao Professor Evandro Costa, do Instituto de Computação da UFAL e seus alunos, que colaboraram de forma decisiva para o sucesso deste trabalho.

Aos meus amigos de copo, indispensáveis à sobrevivência do meu bom humor. Em especial, àqueles que comparecem aos sábados no pagode do Idem no Siri Maluco.

À minha noiva Clara, pelo apoio, compreensão e, acima de tudo, paciência perante minha mania egoísta de trabalhar.

Aos professores e funcionários da COPELE/ DEE.

À CAPES e CNPq pelo apoio financeiro.

# Resumo

As atividades relacionadas à evolução têm sido apontadas como fatores de grande impacto sobre o custo e o tempo inerentes ao processo de engenharia de sistemas de software. O impacto causado por tais atividades de evolução é maior quando as mudanças de requisitos a serem contempladas em um software existente não são previstas, ou antecipadas, durante o projeto inicial do sistema. Este tipo de evolução torna-se ainda mais complexo em determinados domínios de aplicação, nos quais, por razões financeiras ou de segurança, a evolução deve ser realizada dinamicamente, ou seja, sem que a execução do software seja interrompida.

Neste trabalho apresenta-se uma infra-estrutura para o desenvolvimento de software com suporte à evolução dinâmica não antecipada. Mais especificamente, introduz-se um modelo de composição de componentes, arcabouços e um conjunto de ferramentas que permitem o desenvolvimento de software com suporte a mudanças não previstas nos seus requisitos iniciais. Apresenta-se um arcabouço genérico para a implementação da especificação de componentes, assim como, a implementação deste arcabouço nas linguagens Java, Python, C++ e C Sharp, e a extensão deste arcabouço para a construção de aplicações corporativas. Propõe-se também um modelo para análise de desempenho de aplicações desenvolvidas utilizando a infra-estrutura. As ferramentas desenvolvidas para o suporte ao desenvolvimento de componentes e composição, análise de desempenho e execução de aplicações também são descritas.

Apresenta-se também a utilização do método formal Alloy na definição de um mecanismo que possibilita que o desenvolvedor verifique se um dado cenário de evolução não antecipada satisfaz a corretude da especificação formal do sistema. Para guiar o desenvolvedor na utilização da infra-estrutura, descreve-se um processo de desenvolvimento de software com suporte à evolução dinâmica não antecipada. Por fim, a validação do trabalho foi realizada através do desenvolvimento de várias aplicações nos contextos de computação pervasiva, sistemas multi-agentes e comunidades virtuais móveis.

# Abstract

Software evolution has been pointed out as an activity of great impact on the total cost and time of the software engineering process. Such an impact is more significative when requirement changes have not been predicted, or anticipated, during the initial software design. Managing this kind of evolution is more complex in some application domains in which software changes must be performed without stopping the system execution, due to financial or safety reasons.

This work presents an infrastructure for developing software with support to dynamic unanticipated evolution. More specifically, we propose a component model, software frameworks and a set of tools that allow developing software with support to unpredicted changes. It is presented a generic framework to implement the component model specification as well as its implementation in Java, Python, C++ and C Sharp. Also, an extension of the generic framework for developing enterprise applications is presented. Also, we introduce a model to analyze the performance of applications developed with the infrastructure. Tools constructed to support the development of components and the composition, performance analisys, and execution of applications are also described.

We also present the application of the Alloy formal method to specify a mechanism to allow developers to verify if a given unanticipated evolution scenario will impact the system specification correctness. To guide developers in using the proposed infrastructure, we present a process to develop software with support to dynamic unanticipated evolution. Finally, several applications of the proposed infrastructure in the context of pervasive computing, multi-agent systems and mobile virtual communities are presented.

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Lista de Listagens de Código

# Lista de Acrônimos

**API** *Application Programming Interface*

**BIDL** *Balboa Interface Definition Language*

**CAS** *Component Application Server*

**CCF** *C++ Component Framework*

**CCM** *CORBA Component Model*

**CCT** *Component Composition Tools*

**CDC** *Connected Device Configuration*

**CDE** *Component Development Environment*

**CDT** *C++ Development Tools*

**CIL** *Component Integration Language*

**CMS** *Component Model Specification*

**CNPq** Conselho Nacional de Desenvolvimento Científico e Tecnológico

**CVM** Comunidades Virtuais Móveis

**COM** *Component Object Model*

**CORBA** *CORBA*

**COTS** *Commercial Of The Shelf*

**DBC** Desenvolvimento Baseado em Componentes

**DCOM** *Distributed Component Object Model*

**EDSNA** Evolução Dinâmica de Software Não Antecipada

**EJB** *Enterprise Java Beans*

**GCF** *Generic Component Framework*

**GUI** *Graphical User Interface*

**JAR** *Java ARchive*

**JCAS** *Java Component Application Server*

**JCF** *Java Component Framework*

**JDT** *Java Development Tools*

**JIT** *Just-In-Time*

**JSF** *Java Server Faces*

**JSP** *Java Server Pages*

**JVM** *Java Virtual Machine*

**KVM** *Kaffe Virtual Machine*

**OMG** *Object Management Group*

**OSE** *Online Software Evolution*

**OSGi** *Open Service Gateway initiative*

**MIT** *Massachusetts Institute of Technology*

**MVC** Modelo-Visão-Controlador

**PCC** *Plug-in* de Ciência de Contexto

**PDN** *Plug-in* de Descoberta de Nós

**PKUAS** *Peking University Application Server*

**POA** Programação Orientada a Aspectos

**PPS** *Plug-in* de Provisão de Serviços

**PyCF** *Python Component Framework*

**PyDev** *Python Development Environment*

**PSP** *Python Server Pages*

**RPC** *Remote Procedure Call*

**RUP** *Rational Unified Process*

**SBC** Sistema Baseado em Componentes

**SEESCOA** *Software Engineering for Embedded Systems using a Component Oriented Approach*

**SLI** *Split-Level Interfaces*

**STL** *Standard Template Library*

**SMA** Sistema Multi-Agentes

**TDD** *Test-Driven Development*

**UML** *Unified Modeling Language*

**UPnP** *Universal Plug and Play*

**VDPI** Validação de Dependências, Preparação e Implantação

**XML** *eXtensible Markup Language*

**XP** *eXtreme Programming*

# Capítulo 1

# Introdução

Evolução de software é definida 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 a qual foi desenvolvido, de uma maneira viável em termos de custo [1]. De um ponto de vista técnico, tais atividades ocorrem após a entrega da versão inicial do software e envolvem, em geral, correções de problemas no funcionamento (*bugs*), assim como a adição, mudança ou remoção de funcionalidades [2], sendo estas últimas diretamente relacionadas às potenciais mudanças nos requisitos iniciais do software ao longo do seu ciclo de vida.

O crescente investimento científico e industrial na concepção de teorias, técnicas e ferramentas para suporte à evolução de software é devido ao forte impacto das atividades inerentes à evolução sobre o custo total do software. Em 1980, um estudo publicado por Bennet Lientz determinou que a evolução era responsável por 50% do custo total do software [3]. Em estudo mais recente, realizado por Len Erlikh em 2000, este valor subiu para 90% [4]. Vários outros estudos foram realizados, em diferentes contextos, tendo o valor sempre oscilado entre 50% e 90% do custo total do software [5, 6, 7, 8, 9, 10, 11]. Mesmo considerando as possíveis margens de erro relacionadas aos experimentos realizados, é importante observar que os números são bastante elevados.

Motivadas por este cenário, diversas abordagens de engenharia vêm sendo propostas para reduzir o custo de evolução do software. Uma delas é a engenharia de requisitos, que visa obter uma descrição mais fiel dos requisitos a serem contemplados pelo software para que este satisfaça as necessidades dos seus usuários [12]. O argumento utilizado é que quanto mais fiel for tal descrição, menor será a probabilidade de mudanças no futuro e, conseqüentemente, menor será o

custo de evolução.

Uma outra vertente está relacionada aos estudos sobre evolução arquitetural e de projeto de software, que partem do princípio de que o software, em algum momento, será alterado, independente de quão eficiente for a aplicação da engenharia de requisitos. Este princípio tem respaldo no fato de que as arquiteturas de software, em geral, possuem em média um ciclo de vida de 12 a 30 anos [13]. Durante este tempo, considerando a velocidade com que as organizações, empresas e negócios mudam atualmente, dificilmente não haverá mudanças de requisitos. Portanto, a idéia é preparar a arquitetura e o projeto do software para mudanças, pois inevitavelmente elas um dia ocorrerão [14]. É esta última vertente que é adotada no contexto deste trabalho.

Esta idéia de projetar software com flexibilidade visando minimizar o impacto gerado por alterações futuras não é recente. Bons desenvolvedores, em grandes empresas, já empregavam técnicas de modularização e encapsulamento ainda no domínio da programação estruturada, visando facilitar o processo de evolução [15].

Entretanto, a popularização desta filosofia "pró-flexibilidade" aconteceu com a publicação em inglês do livro "Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos", em 1995 [16]. Neste livro descreve-se um catálogo de padrões de projeto orientado a objetos com soluções elegantes e flexíveis, tornando-se indispensável para programadores, projetistas e gerentes de desenvolvimento de software que buscavam atender à enorme demanda por software de qualidade, em cada vez menos tempo, da década de 90 e início deste novo milênio. Esta iniciativa desencadeou uma "febre" pela concepção de padrões em diversos setores da Engenharia de Software: análise, teste, negócio, modelagem, dentre outros.

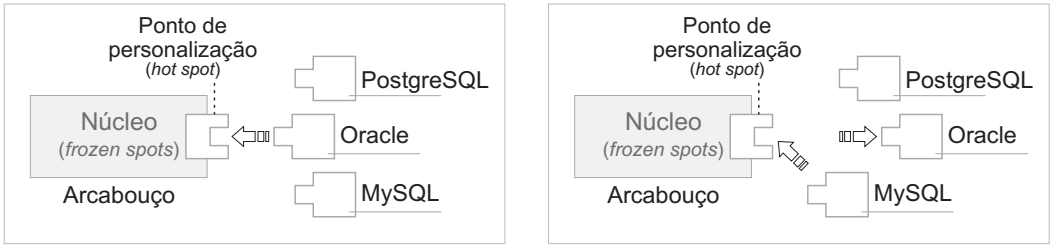
De fato, a idéia conceitual por trás do uso de padrões de projeto está em tornar o projetista de software apto a fazer suposições, *a priori*, sobre as possíveis mudanças de projeto que ocorrerão ao longo do ciclo de vida do software. Sendo assim, preparando o projeto para mudanças, estas trarão menos impacto quando ocorrerem, reduzindo os custos de evolução.

Outra iniciativa com foco em reutilização que também provê suporte à evolução de software são os arcabouços de aplicação (*application frameworks*) [17]. Um arcabouço de aplicação, ou simplesmente arcabouço, é um esqueleto de aplicação, semi-completo, reutilizável, que pode ser especializado ou composto para produzir aplicações específicas.

Apesar da principal motivação para a utilização de arcabouços ser a reutilização de uma estrutura base para o desenvolvimento de várias aplicações de um dado domínio, o projeto do arca-

bouço, em geral, também provê grande flexibilidade para evolução. Isto ocorre porque os pontos de extensão do arcabouço (*hot spots*), ou ganchos (*hooks*), são definidos com base em suposições ou observações sobre diferentes utilizações do mesmo para a construção de aplicações específicas. A flexibilidade necessária para permitir tal extensão também pode ser utilizada para facilitar possíveis alterações futuras. Sendo assim, ao se construir uma aplicação utilizando um arcabouço de software, esta estará preparada para um conjunto de potenciais mudanças, determinadas pelos pontos de extensão do arcabouço.

Por exemplo, considere um arcabouço para conexão e acesso a banco de dados (Figura 1.1). O projeto do arcabouço deve prover mecanismos para que este possa ser estendido para uma dada aplicação, como um gerenciador de banco de dados específico (Oracle [18], MySQL [19] ou PostgreSQL [20], por exemplo). Mas o núcleo do arcabouço (*frozen spots*) e, conseqüentemente, a aplicação que fará uso do mesmo estarão desacoplados do gerenciador escolhido. Sendo assim, da mesma forma que é possível definir qual gerenciador utilizar no momento de estender o arcabouço, torna-se mais simples alterá-lo, se necessário. Portanto, caso haja um cenário de evolução em que o gerenciador necessite ser modificado, o impacto desta mudança sobre o projeto existente será reduzido.




(a) Construção da aplicação. (b) Evolução da aplicação.

Figura 1.1: Arcabouços de software promovem flexibilidade e suporte à evolução.

Seguindo esta mesma linha de raciocínio, foram concebidos os conceitos relacionados ao Desenvolvimento Baseado em Componentes (DBC) [21]. DBC tem como objetivo, em primeira instância, o desenvolvimento de software a partir de componentes pré-concebidos, sendo estes componentes reutilizáveis em várias aplicações e facilmente personalizáveis para produzir novas funcionalidades [22].

A característica de reutilização é ainda mais forte no caso de componentes de prateleira (*Com-*

*mercial Off-The-Shelf*) [23]. Isto ocorre porque componentes de prateleira são artefatos de software caixa-preta, sendo seus serviços conhecidos apenas pela interface [24]. Sendo assim, além da reutilização, o fraco acoplamento entre os componentes de prateleira reduz o impacto da evolução do software, permitindo que componentes existentes possam ser trocados por outros componentes que possuam a mesma interface e sigam o mesmo contrato. O projeto de um software baseado em componentes de prateleira é similar ao de um arcabouço de aplicação caixa-preta [17]. De fato, estes arcabouços são utilizados, em geral, para gerenciar a composição e a interação entre tais componentes [25].

Diversas novas abordagens de engenharia de software vêm sendo propostas e são discutidas ao longo deste documento, tais como a programação orientada a aspectos [26], arquiteturas orientadas a serviços [27], arquiteturas baseadas em *plug-ins* [28], arquiteturas orientadas a agentes [29], dentre outras. Tais abordagens foram concebidas com propósitos diversos, mas também possuem algum tipo de suporte à evolução do software.

É dentro deste contexto de suporte à evolução de sistemas de software que se insere este trabalho, tendo em vista a redução do impacto da mudança de requisitos sobre o software existente. A problemática que motivou a elaboração da tese, assim como os objetivos e a relevância do trabalho são apresentados a seguir.

## 1.1 Problemática

À primeira vista, o "arsenal" de abordagens existentes parece ser suficiente para reduzir a grande parcela do custo total do software atribuída às atividades relacionadas à evolução. Qual seria então o problema? O foco do problema de tais abordagens pode ser resumido em uma característica fundamental: antecipação. Todas as abordagens mencionadas anteriormente se baseiam na antecipação de cenários de evolução, preparando o projeto do software para tal evolução e, conseqüentemente, reduzindo o impacto quando ocorrem mudanças.

O grande problema é que não se pode prever todos os cenários de evolução possíveis. Pelo contrário, quanto mais cenários de evolução são previstos e considerados no projeto, mais complexa se torna a solução inicial, levando a um projeto mais flexível do que demandam os seus requisitos e aumentando o custo e o tempo de desenvolvimento. Este fenômeno é denominado *over-engineering* e tem sido apontado como um dos responsáveis pela baixa produtividade nos

projetos de desenvolvimento de software [30].

É simples observar que o impacto da evolução sobre o projeto e o código existentes é mais significativo quando as mudanças nos requisitos de software não foram antecipadas. Por definição, “evolução de software não antecipada não é algo para o qual é possível preparar-se durante o projeto de um sistema de software” [31]. Sendo assim, uma mudança é considerada não antecipada quando sua implementação não depende de ganchos codificados em versões anteriores do software [32]. Portanto, o problema surge quando uma parte do software que não havia sido preparada para mudanças precisa ser alterada (Figura 1.2). Para complicar o trabalho do desenvolvedor, tais mudanças são vistas como possíveis e, muitas vezes, até simples, pelos clientes de software. O cenário apresentado na Figura 1.2, por exemplo, seria inconcebível em outras engenharias, mas é considerado possível pelos clientes da Engenharia de Software, devido ao fato de que o artefato é lógico e não físico.

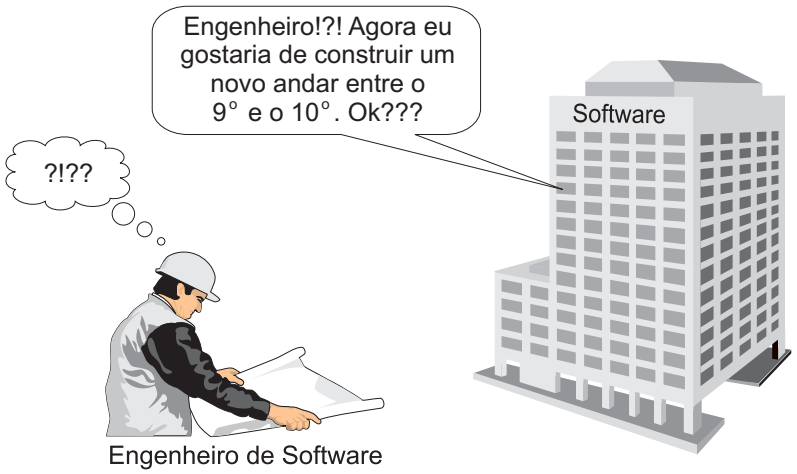



Figura 1.2: Evolução de Software Não Antecipada.

O problema se torna ainda mais complexo ao se considerar o cenário atual de desenvolvimento de software. A proliferação de aplicações baseadas em Internet, as quais possuem característica de acesso global, torna inviável a especificação de horários de funcionamento do software – madrugada no Brasil é tarde no Japão – e exige execução 24 horas por dia, 7 dias na semana (24/7). Isto significa que as atividades de evolução devem ser gerenciadas dinamicamente para evitar perdas financeiras. Um bom exemplo pôde ser observado com um grande site de leilões que, por problemas de infra-estrutura, teve seu sistema interrompido por 22 horas, o que representou um prejuízo de cerca de US$ 3 milhões [33].

Este cenário não é restrito apenas a aplicações baseadas na Internet. Em vários outros domínios a execução do software não pode ser interrompida, seja por motivos financeiros ou de segurança, tais como telecomunicações, sistemas bancários, sistemas hospitalares, sistemas militares, etc. Há ainda domínios em que a própria natureza dinâmica das aplicações exige uma gerência de evolução em tempo de execução. Um exemplo desses domínios é a Computação Pervasiva [34], segundo a qual novos serviços devem ser carregados por demanda para dispositivos móveis, de acordo com as necessidades definidas em um perfil de usuário, mas sem interferência direta do mesmo.

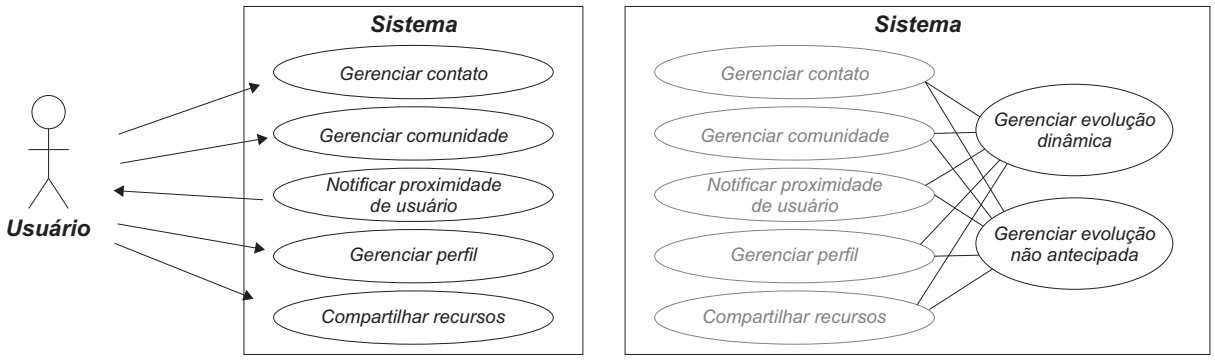
A gerência de evolução dinâmica não antecipada pode ainda ser considerada importante mesmo em aplicações *desktop*, no contexto de atualizações e correções de problemas de funcionamento e de segurança. Grande parte dos sistemas de software, incluindo sistemas operacionais, necessitam ser reiniciados quando ocorrem alterações nos módulos existentes ou novos módulos são adicionados. Este problema poderia ser resolvido com uma infra-estrutura de suporte à Evolução Dinâmica de Software Não Antecipada (EDSNA).

Mas como prover suporte à EDSNA sem causar *over-engineering*? A resposta é simples: uma infra-estrutura de suporte à EDSNA não deve requerer dos desenvolvedores que estes antecipem mudanças e especifiquem qual parte do software poderia evoluir no futuro. Qualquer parte do software deve suportar evolução e os desenvolvedores não devem ter que gerenciar nem implementar os mecanismos que permitem tal evolução.

Nenhuma das abordagens existentes foi concebida com o objetivo descrito. De fato, é possível combinar um conjunto de técnicas para adaptar tais abordagens e adicionar um suporte à EDSNA. Mas, ainda assim, alguns problemas se mantêm em aberto. Para entender tais problemas, considere o exemplo de uma aplicação para computação pervasiva, mais especificamente, um software de acesso a comunidades virtuais móveis.

Na Figura 1.3(a), são ilustrados os requisitos funcionais desta aplicação, a qual representa um dos estudos de casos da infra-estrutura aqui proposta e é detalhada no Capítulo 12. Do ponto de vista do usuário, estas são as funcionalidades do sistema e, conseqüentemente, deveriam ser também as funcionalidades principais do ponto de vista do desenvolvedor. Porém, com um suporte à evolução dinâmica não antecipada, tem-se novos requisitos a serem implementados, do ponto de vista do desenvolvedor, como ilustrado na Figura 1.3(b).

O grande problema das abordagens existentes é que elas deixam os requisitos de evolução di-




(a) Requisitos funcionais do sistema.

(b) Requisitos de evolução dinâmica não antecipada.

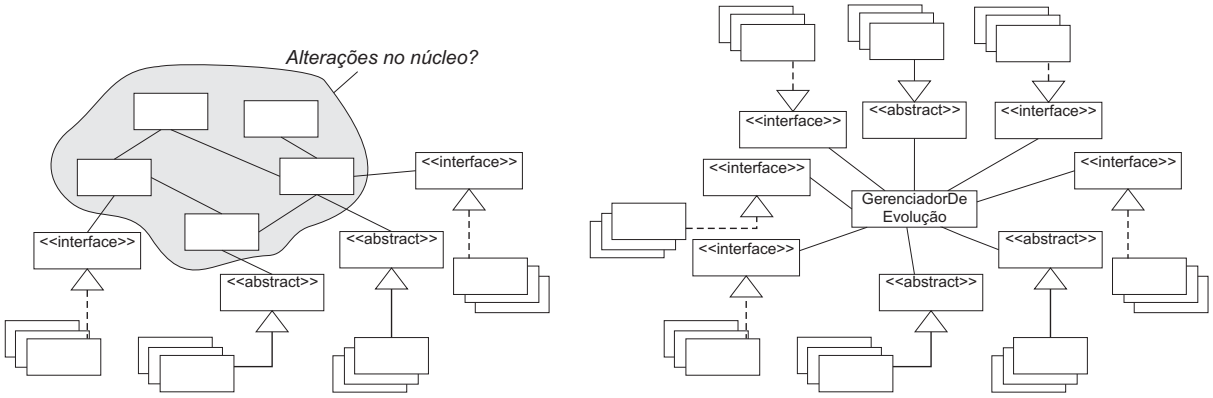
Figura 1.3: Requisitos da aplicação de acesso a comunidades virtuais móveis.

nâmica não antecipada sob gerência do desenvolvedor, o que necessariamente terá que fazer parte do cronograma e custo de desenvolvimento. Ou seja, além de implementar a aplicação, o desenvolvedor terá que preparar todas as suas funcionalidades para serem evoluídas dinamicamente, já que não é possível prever quais irão evoluir.

Por exemplo, considere a utilização de arcabouços de aplicação. O princípio fundamental na idéia de arcabouços é a separação entre as partes mutáveis e imutáveis de um projeto de software. As partes mutáveis são extensíveis, as partes imutáveis não são. Mas em se tratando de evolução não antecipada, o que fazer caso uma parte que havia sido definida como imutável precise ser alterada (Figura 1.4(a))?

Uma possível solução seria desenvolver toda a aplicação como um arcabouço de software, com pontos de extensão para todas as funcionalidades. Acoplando este projeto flexível a um mecanismo de reflexão computacional [35] e carregamento dinâmico de classes [36], seria possível fazer com que a aplicação pudesse ser evoluída dinamicamente.

No entanto, o projeto de um arcabouço é mais complexo que um projeto de uma aplicação específica. Projetar de forma extensível não é uma tarefa trivial e tornar todo o projeto da aplicação extensível é ainda mais complicado. Além disso, seria necessário que o desenvolvedor gerenciasse os mecanismos de evolução dinâmica, que não estariam presentes, *a priori*, nos requisitos da aplicação. Em resumo, para desenvolver a aplicação de comunidades virtuais móveis, com suporte à evolução dinâmica não antecipada, o desenvolvedor precisaria entender conceitos de arcabouços de software, reflexão e carregamento dinâmico de classes, implementando e gerenciando a execução destas funcionalidades (Figura 1.4(b)).




(a) Arcabouço de software.

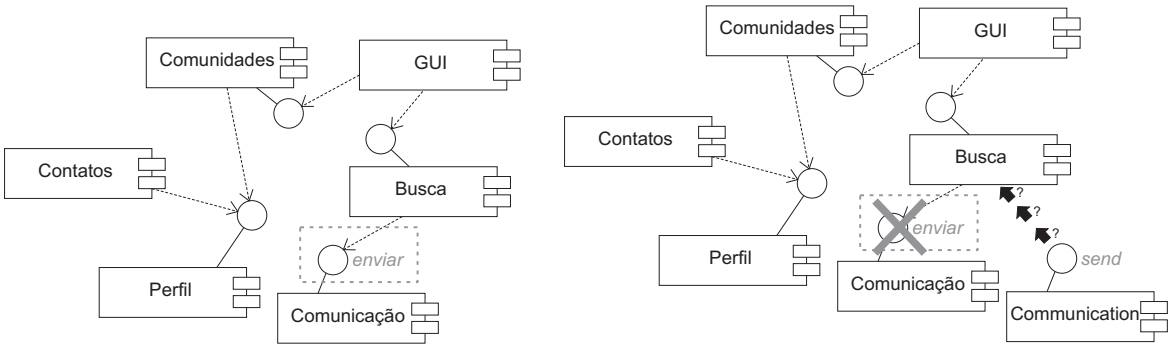
(b) Arcabouço de software com suporte à evolução dinâmica não antecipada.

Figura 1.4: Usando arcabouços para suporte à evolução dinâmica não antecipada.

O mesmo raciocínio elaborado para o contexto de arcabouços pode ser aplicado ao desenvolvimento baseado em *plug-ins* [28]. De acordo com esta abordagem, uma aplicação deve conter um núcleo, de preferência mínimo, e um conjunto de *plug-ins* que implementam as funcionalidades do sistema [37]. Algumas aplicações baseadas em *plug-ins* já contemplam a evolução dinâmica *antecipada*, pois apenas implementam mecanismos para a evolução dinâmica dos *plug-ins* mas não das funcionalidades do núcleo. Um exemplo recente da utilização desta abordagem é a plataforma Eclipse [38].

Outra potencial abordagem para atacar o problema do suporte à EDSNA é o Desenvolvimento Baseado em Componentes (DBC). Porém, apesar de promover flexibilidade de projeto para facilitar mudanças antecipadas, como mencionado anteriormente, o mesmo não ocorre para a evolução não antecipada, ao menos se consideradas as abordagens existentes. Isto ocorre tanto para abordagens mais comerciais voltadas para aplicações corporativas, tais como *Enterprise Java Beans* (EJB) [39] e *CORBA Component Model* (CCM) [40], quanto para abordagens mais voltadas para evolução dinâmica, tais como *Hadas* [41].

O problema com tais abordagens de DBC é que a conexão entre os componentes é flexível mas estritamente ligada aos contratos de interface estabelecidos entre os componentes. Sendo assim, a substituição de um componente por outro deve continuar respeitando o mesmo contrato, o que torna complicada a mudança na interface do componente sem impactar os demais, principalmente em tempo de execução (Figura 1.5).




(a) Software baseado em componentes com interfaces bem definidas...

(b) ...mas alterações se restringem à especificação da interface.

Figura 1.5: Usando componentes para suporte à evolução dinâmica não antecipada.

No contexto de arquiteturas baseadas em serviços, a principal abordagem a ser considerada é a plataforma de serviços da *Open Service Gateway initiative* (OSGi), que tem como objetivo prover um ambiente orientado a serviços, oferecendo mecanismos padronizados para gerenciar o ciclo de vida de aplicações desenvolvidas em Java, independente da plataforma alvo [42].

De acordo com OSGi, uma aplicação é composta por um conjunto de serviços, denominados *bundles*, que são registrados e posteriormente acessados por outros *bundles*. Os *bundles* podem ser inseridos, removidos e alterados dinamicamente, através de ferramentas que implementam a especificação OSGi, tais como Oscar [43] e Knoplerfish [44].

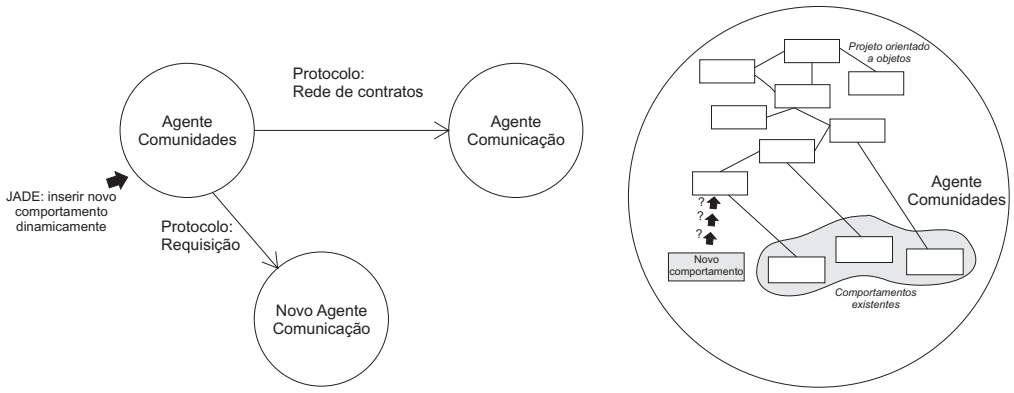
Apesar do suporte à evolução dinâmica provido por OSGi, não há mecanismos que permitam a evolução não antecipada. Isto ocorre porque a ligação entre os *bundles* é feita diretamente no código, inclusive apontando versões específicas dos serviços a serem utilizados. Se por um lado esta característica permite que várias versões de um mesmo *bundle* executem ao mesmo tempo, exige que a cada nova versão de *bundle* seus dependentes sejam também alterados. Este problema é semelhante ao acoplamento entre os componentes nas abordagens existentes para DBC.

No caso da Engenharia de Software Orientada a Agentes [45], tem-se uma abordagem para a construção de sistemas de software complexos com base em um conjunto de entidades autônomas, denominadas agentes, que interagem para prover as funcionalidades do sistema [46].

A utilização de protocolos de alto nível [47] para definir a interação entre os agentes torna as entidades do sistema extremamente desacopladas, facilitando a evolução do sistema através da entrada e saída de agentes. No entanto, apesar da flexibilidade no nível de interação, a implemen-

tação interna de cada um dos agentes é, em geral, realizada utilizando uma linguagem orientada a objetos. Isto ocorre, por exemplo, com a ferramenta Jade [48], uma das mais utilizadas na área.

Sendo assim, no nível da implementação do agente, tem-se os mesmos problemas citados anteriormente. Considerando que a implementação do tratamento dado a cada mensagem definida no protocolo é realizada dentro do agente, o nível de interação também acaba sendo comprometido (Figura 1.6).




(a) Software orientado a agentes com JADE: novo comportamento para novo protocolo adicionado dinamicamente...

(b) ... mas o projeto orientado a objetos interno do agente desconhece o novo comportamento.

Figura 1.6: Usando agentes para suporte à evolução dinâmica não antecipada.

Este mesmo problema ocorre com abordagens de computação distribuída baseada em serviços, como *Web Services* [49], CORBA *Trading Services* [50] e Jini [51]. Apesar do fraco acoplamento entre os serviços, a implementação dos mesmos e do código de aplicação que compõe tais serviços não possui a mesma flexibilidade. Uma solução seria implementar todas as funcionalidades da aplicação como serviços, o que seria impraticável para o desenvolvedor dado o nível de complexidade das abordagens.

A solução apontada como a mais promissora para lidar com a evolução dinâmica não antecipada [2] é a Programação Orientada a Aspectos (POA) [26]. POA promove a separação de interesses através da definição de entidades denominadas aspectos que implementam requisitos comuns aos vários módulos das aplicações, denominados requisitos transversais. Com o suporte de ferramentas como AspectJ [52] é possível combinar códigos de aspectos e de classes para construir uma aplicação, mas ainda mantendo a separação de interesses em tempo de desenvolvimento.

Desta forma, o código dos objetos se mantém desacoplado do código de aspectos e, portanto, não é afetado por mudanças nos aspectos. Sendo assim, novos aspectos podem ser adicionados, implementando novas funcionalidades, sem alterar o código existente.

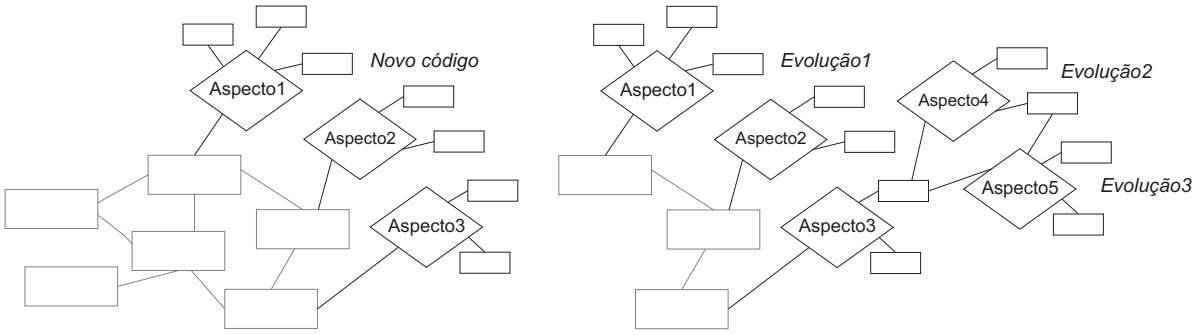
Além disso, o conceito de aspectos dinâmicos [53] é bem adequado para o suporte necessário à evolução dinâmica. Com base neste conceito, e nas várias ferramentas disponíveis que o implementam [53], é possível combinar aspectos ao código dinamicamente. Assim, novas funcionalidades não previstas anteriormente podem ser adicionadas mesmo sem o código fonte da aplicação, ou seja, em tempo de carregamento de classes.

Porém, as abordagens existentes para a programação orientada a aspectos dinâmicos possuem dois problemas. O primeiro deles está relacionado à infra-estrutura de evolução dinâmica. O conceito de aspectos dinâmicos permite que aspectos, definidos geralmente utilizando linguagens declarativas, sejam combinados a um código já compilado, mas não permite que novos aspectos sejam carregados dinamicamente após a aplicação iniciar sua execução [53]. Sendo assim, o carregamento dos aspectos é dinâmico no sentido de combinar-se ao código orientado a objetos em tempo de carregamento de classes, não no sentido de evoluir a aplicação enquanto esta executa.

Ainda que o problema supracitado seja resolvido por alguma ferramenta, ainda resta o segundo problema: gerência da evolução. Na medida em que a aplicação evolui, novos aspectos vão sendo adicionados e combinados ao código existente, inserindo novas funcionalidades no código orientado a objetos. Após alguns cenários de evolução, o entendimento e uma nova evolução no código tornam-se impraticáveis, principalmente considerando-se que o código definido nos aspectos também pode evoluir (Figura 1.7).

Linguagens de programação de *script*, independentemente do paradigma utilizado, também poderiam ser apontadas como possíveis soluções para o problema da EDSNA. Isto ocorre porque algumas destas linguagens, tais como Ruby [54] e Python [55], permitem a inserção dinâmica de código. Isto significa, por exemplo, que é possível criar uma estrutura condicional onde, caso a condição seja satisfeita, um novo código seja adicionado à aplicação. No caso de Python, pode-se alterar um programa em execução, mas na medida em que o software evolui, mais difícil fica de gerenciá-lo, pois tem-se uma gerência de evolução a nível de código. Neste caso, tem-se o mesmo problema da solução orientada a aspectos.

Apesar desta discussão estar considerando os paradigmas de uma forma genérica, ela é fundamentada em uma análise consistente de vários trabalhos existentes que se baseiam nos conceitos




(a) Software orientado a aspectos com código adicionado dinamicamente...

(b) ... mas após algumas evoluções, torna-se de difícil gerência.

Figura 1.7: Usando aspectos para suporte à evolução dinâmica não antecipada.

definidos nestes paradigmas. Sendo assim, o problema não está na definição dos paradigmas e sim na implementação destes conceitos não motivada pela necessidade de EDSNA. De fato, vários destes trabalhos possuem alguma característica que pode facilitar o suporte à EDSNA, o que será discutido neste documento. Porém nenhum deles provê este suporte de forma transparente para o desenvolvedor, pois não foram concebidos com este objetivo, convergindo em algum momento para os problemas apresentados anteriormente.

Além disso, as diversas soluções existentes, quando provêem algum tipo de suporte, o fazem para linguagens e contextos específicos. Por exemplo, a especificação OSGi é baseada nos conceitos de Java e está fortemente relacionada aos mecanismos de carregamento de classes desta linguagem. Um outro exemplo é que apenas algumas linguagens, como AspectJ, que é específica para aplicações desenvolvidas em Java, possuem suporte a aspectos dinâmicos. Enfim, os mecanismos que promovem o suporte à evolução dinâmica, além de apresentarem os problemas mencionados anteriormente, são, na maioria das vezes, dependentes de linguagem.

## 1.2 Objetivos da Tese

### 1.2.1 Objetivo Principal

Neste trabalho, tem-se como objetivo principal a concepção e o desenvolvimento de uma infraestrutura baseada em componentes para a engenharia de software com suporte à evolução dinâmica não antecipada. Mais especificamente, propõe-se um modelo de composição de componen-

tes, um processo de desenvolvimento, arcabouços de software e um conjunto de ferramentas que permitam o desenvolvimento de software considerando os requisitos descritos a seguir.

- **Transparência para o desenvolvedor –** Deve-se prover um modelo de desenvolvimento de software que permita ao desenvolvedor, apenas através da utilização deste modelo, desenvolver software que suporte evolução dinâmica não antecipada. Em momento algum, o desenvolvedor deverá apontar eventuais pontos de mudança no software sendo construído. Os mecanismos que permitam a evolução dinâmica não antecipada devem ser transparentes para o desenvolvedor.

- **Independência de linguagem e plataforma –** A infra-estrutura proposta não deve ser dependente de linguagem de implementação ou plataforma específica para a sua execução. Sendo assim, espera-se que seja possível aplicar tal infra-estrutura a diferentes contextos e plataformas de execução, considerando diferentes linguagens de programação, incluindo linguagens populares e de grande abrangência no processo de desenvolvimento de software atualmente, tais como Java [56], C++ [57], CSharp [58] e Python [55].

- **Infra-estrutura baseada em componentes –** Como forma de reaproveitar recentes esforços e avanços no contexto de Engenharia de Software, principalmente aqueles relacionados à flexibilidade, modularidade e reutilização de software em busca de maior produtividade e menor tempo de desenvolvimento, a infra-estrutura deve estar de acordo com o paradigma de desenvolvimento baseado em componentes, mais especificamente, componentes de prateleira [24].

- **Suporte à modelagem e verificação formal acoplado ao processo de evolução –** Considerando a existência de uma especificação formal inicial que é seguida pelo sistema, deve ser possível verificar se um dado cenário de evolução, tal como adição, remoção ou mudança de componente, vai de encontro à corretude da especificação. Este mecanismo deve estar acoplado ao processo de evolução e deve ser automatizável para possibilitar que o desenvolvedor realize a verificação ainda em tempo de projeto e com auxílio de ferramentas.

- **Comprometimento com a prática do processo em ambientes de desenvolvimento de software –** É imprescindível que a infra-estrutura proposta disponha de um suporte ferra-

mental para ser aplicada em ambientes de desenvolvimento de software. Todas as atividades devem ser auxiliadas por ferramentas computacionais que automatizem as atividades de desenvolvimento. Um modelo teórico que não possa ser automatizado ou inserido no ambiente de desenvolvimento utilizado para programação, por exemplo, não é considerado um modelo interessante no contexto deste trabalho.

### 1.2.2 Objetivos Específicos

Considerando os requisitos descritos anteriormente, pode-se dividir o objetivo geral deste trabalho nos objetivos específicos descritos a seguir.

1. **Definição de um modelo de composição de componentes –** Especificar um modelo de componentes, aqui denominado CMS (*Component Model Specification*), para o desenvolvimento de software baseado em componentes com suporte à evolução dinâmica não antecipada. Este modelo inclui diretrizes de adição e remoção de componentes, composição de aplicações baseadas em componentes e interação entre componentes da aplicação.

2. **Desenvolvimento de arcabouços de componentes –** Desenvolver arcabouços de software que implementam a especificação definida anteriormente. Estes arcabouços devem ser implementados utilizando diferentes linguagens, mais especificamente, Java, C++, CSharp e Python, para validar a genericidade e a abrangência do modelo de componentes. Desenvolvedores de software devem utilizar estes arcabouços para desenvolver aplicações para plataformas específicas, escolhendo a linguagem adequada de acordo com os requisitos de tais aplicações. Além disso, deve-se definir um projeto de arcabouço de software independente de linguagem, orientado a objetos, a ser seguido pelas implementações de arcabouços supracitadas. Este arcabouço genérico pode ser utilizado como base para novas implementações do modelo em outras linguagens orientadas a objetos.

3. **Desenvolvimento de ambientes para composição e execução de aplicações –** Desenvolver ambientes de software para o auxílio à composição de aplicações de acordo com a CMS e para o auxílio à execução de aplicações desenvolvidas utilizando os arcabouços de software. O ambiente de composição de software deve ser construído sobre a plataforma Eclipse [59], de forma independente da linguagem de implementação dos componentes. A

escolha de Eclipse como plataforma-base para o ambiente deve-se ao suporte desta plataforma para a construção de ambientes de desenvolvimento, incluindo interface gráfica, *wizards*, internacionalização, gerenciamento de janelas, visões e perspectivas, dentre outras características que facilitam a construção de tais ambientes. Os ambientes de execução, específicos para cada linguagem, devem ser construídos em separado e funcionam como servidores de aplicação.

4. **Técnica para verificação formal de software diante de cenários de evolução –** Definir uma técnica formal, com base na linguagem Alloy [60], para a verificação formal de especificações de software baseado na CMS, buscando identificar possíveis problemas de corretude após cenários de evolução. As justificativas para a escolha de Alloy estão diretamente relacionadas à conformidade com os conceitos de orientação a objetos – o que pode facilitar sua utilização por parte de desenvolvedores – e a possibilidade de acoplamento da ferramenta de verificação ao ambiente de composição – o que possibilita a automatização do processo.

5. **Definição de um processo de desenvolvimento de software –** Definir um processo para guiar o desenvolvedor na utilização dos conceitos definidos na CMS durante as fases de desenvolvimento de software.

6. **Desenvolvimento de aplicações-piloto –** Desenvolver aplicações-piloto, em diferentes domínios para validar os diversos aspectos do modelo de componentes, arcabouços e ambientes de desenvolvimento.

## 1.3 Relevância do Tema e da Tese

Evolução de software é um tema relevante no contexto de Engenharia de Software devido ao impacto da evolução sobre o custo e tempo total de desenvolvimento. Esta foi a motivação para a concepção dos conceitos de encapsulamento, modularização, padrões, arcabouços etc. E esta ainda continua sendo a motivação para novos paradigmas, como a programação orientada a aspectos, que visa a separação de interesses para, dentre outras coisas, facilitar o processo de evolução do software.

No caso do tema específico tratado neste trabalho, a evolução dinâmica de software não antecipada é cada vez mais relevante devido às características inerentes às aplicações desenvolvidas atualmente e já citadas anteriormente, tais como mudança freqüente de requisitos e funcionamento 24/7. Estas características, aliadas à reutilização, tornam o tema ainda mais relevante para o contexto de desenvolvimento baseado em componentes.

No que diz respeito à relevância do trabalho, em se tratando de uma tese em Engenharia de Software, o autor considera indispensáveis três principais requisitos que foram contemplados e que reforçam a relevância da tese. Estes requisitos serviram como motivação para a realização do trabalho.

O primeiro deles é a consistência teórica. A infra-estrutura proposta foi concebida a partir de um modelo conceitual, fundamentado nos conceitos do desenvolvimento baseado em componentes. Tal modelo foi descrito de forma rigorosa e não-ambígua, permitindo um melhor entendimento e futuros investimentos no arcabouço teórico proposto.

O segundo requisito é a contribuição científica. Diversos trabalhos relacionados foram estudados antes da concepção do modelo conceitual e da infra-estrutura propostos. A partir deste estudo, identificou-se o problema enunciado anteriormente e, a partir de então, foram elencadas as possíveis soluções para o problema, o que culminou com a definição deste trabalho. Até o momento da escrita deste documento, não foram encontrados trabalhos com as características aqui propostas, o que reforça o caráter de originalidade e a contribuição científica, a qual já vem sendo respaldada pela comunidade através da publicação de vários artigos em veículos relevantes da área.

O terceiro requisito é o potencial prático. Toda a infra-estrutura de software desenvolvida no contexto deste trabalho tem como único objetivo demonstrar que a abordagem é viável e praticável. Um modelo de componentes sem arcabouços, arcabouços sem ferramentas e ferramentas sem aplicações seriam cenários que tornariam as reais contribuições deste trabalho apenas suposições. O compromisso com a utilização dos conceitos para construir mecanismos que possam ser aplicados na indústria de software tornam o trabalho relevante em termos práticos.

Por fim, a partir deste trabalho, deu-se origem a um projeto multi-institucional denominado COMPOR - SOFTWARE COMPOSITION (`http://www.compor.net`), que tem por objetivo a concepção de infra-estruturas de desenvolvimento de software com suporte à evolução dinâmica não antecipada. Com base nos resultados deste projeto, vários outros projetos relacionados foram criados, tanto no nível de pesquisa [61, 62, 63, 64, 65] quanto no nível de desenvolvimento [66]. Da

mesma forma, várias dissertações de mestrado [67, 68, 69] e trabalhos de conclusão de curso de graduação e especialização [70, 71, 72, 73, 74, 75, 76, 77, 78] foram desenvolvidos, destacando também a relevância do trabalho em termos institucionais.

## 1.4 Resumo das Contribuições

Um resumo das principais contribuições do trabalho no contexto de desenvolvimento de software com suporte à evolução dinâmica não antecipada é descrito a seguir:

- **Modelo de composição de componentes –** Definição e formalização de uma especificação de modelo de componentes para a construção de software com suporte à evolução dinâmica não antecipada.
- **Arcabouços de componentes –** Definição de um projeto genérico de arcabouço de software para o desenvolvimento de aplicações de acordo com a especificação de componentes proposta, com implementações em Java, Python, C++ e CSharp e uma extensão para a construção de aplicações corporativas.
- **Técnica para verificação formal –** Definição de uma técnica para a verificação de propriedades do sistema diante de cenários de evolução baseada na linguagem Alloy e utilizando a ferramenta *Alloy Analyzer* para automatizar o processo.
- **Modelo para análise de desempenho –** Concepção de um modelo analítico para avaliação de desempenho de aplicações baseadas na especificação de componentes proposta, com o objetivo de auxiliar o desenvolvedor na identificação da arquitetura mais adequada para a sua aplicação, considerando requisitos de coesão funcional, flexibilidade e desempenho.
- **Ambientes para composição e execução de aplicações –** Construção de ambientes de desenvolvimento de componentes, composição e execução de aplicações baseadas na CMS.
- **Processo de desenvolvimento –** Definição de um processo de desenvolvimento para guiar o desenvolvedor na construção de software com suporte à evolução dinâmica não antecipada.
- **Aplicações-piloto –** Desenvolvimento de diversas aplicações utilizando a infra-estrutura proposta como forma de validar o aspecto de evolução dinâmica não antecipada.

## 1.5 Estrutura do Documento

O restante deste documento está organizado da seguinte forma:

- No Capítulo 2, são apresentados alguns conceitos relacionados ao trabalho que são necessários ao entendimento do restante do documento.
- No Capítulo 3, são apresentados os trabalhos relacionados à infra-estrutura proposta.
- No Capítulo 4, apresenta-se a especificação do modelo de composição de componentes para a construção de software com suporte à evolução dinâmica não antecipada.
- No Capítulo 5, apresentam-se os arcabouços de componentes desenvolvidos para a construção de software com suporte à evolução dinâmica não antecipada de acordo com a especificação descrita no Capítulo 4, incluindo um projeto genérico de arcabouço independente de linguagem.
- No Capítulo 6, são apresentadas extensões do projeto do arcabouço genérico descrito no Capítulo 5 para a construção de aplicações corporativas.
- No Capítulo 7, apresenta-se a técnica formal, com base na linguagem Alloy, para a verificação de corretude em sistemas de software baseados na CMS, diante de cenários de evolução.
- No Capítulo 8, apresenta-se um modelo analítico para avaliação de desempenho do software construído com base na CMS.
- No Capítulo 9, apresenta-se o ambiente de composição e desenvolvimento de aplicações baseadas na CMS, construído como um conjunto de *plug-ins* para a plataforma Eclipse.
- No Capítulo 10, descreve-se uma arquitetura de plataforma para a disponibilização de componentes e execução de aplicações, assim como sua implementação na linguagem Java.
- No Capítulo 11, apresenta-se um processo de desenvolvimento para a construção de aplicações com suporte à evolução dinâmica não antecipada utilizando a infra-estrutura proposta.
- No Capítulo 12, são apresentadas as aplicações desenvolvidas utilizando a infra-estrutura proposta.

- No Capítulo 13, são apresentadas as conclusões e as perspectivas futuras deste trabalho.
- No Apêndice A, apresenta-se um detalhamento do projeto genérico de arcabouço de componentes descrito resumidamente no Capítulo 5.
- No Apêndice B, são detalhadas os arcabouços específicos de linguagem que implementam a CMS, os quais foram descritos resumidamente no Capítulo 5.

# Capítulo 2

# Fundamentação

Neste capítulo são apresentados conceitos relacionados ao Desenvolvimento Baseado em Componentes, Evolução Dinâmica de Software Não Antecipada e Alloy. Estes conceitos são considerados fundamentais para o entendimento do restante do documento. Além disso, delimita-se o escopo deste trabalho em relação a cada um destes conceitos.

## 2.1 Desenvolvimento Baseado em Componentes - DBC

Nesta seção são apresentados conceitos relacionados ao Desenvolvimento Baseado em Componentes (DBC), destacando as principais entidades definidas neste paradigma e, em seguida, descrevendo um tipo específico de componente, denominado componente de prateleira. São também apresentadas as fases inerentes ao ciclo de desenvolvimento de software baseado em componentes. Por fim, descreve-se como os conceitos de DBC são aplicados a este trabalho.

### 2.1.1 Componente, Modelo de Componentes e Arcabouço de Componentes

O Desenvolvimento Baseado em Componentes é caracterizado pela composição de aplicações com base na reutilização de unidades de software pré-existentes [22]. Através deste processo de reutilização e montagem, tem-se um potencial aumento na qualidade da aplicação e na produtividade do desenvolvimento, diminuindo o tempo e o custo de produção [21].

Dentro do contexto de DBC, o principal conceito é o de *componente*. Várias definições de componente podem ser encontradas na literatura [79, 80, 81, 82, 22, 83]. Dentre elas, utiliza-se

neste trabalho a definição de Clemens Szyperski [82] (Definição 2.1).

**Definição 2.1 (Componente)** Um componente de software é uma unidade de composição com interfaces especificadas de forma contratual, podendo ser desenvolvida de forma independente e sujeita à composição por terceiros.

Segundo a Definição 2.1, dois outros conceitos são utilizados para definir componente: interface e contrato. A interface de um componente pode ser definida como a especificação do seu *ponto de acesso*. Um componente é uma unidade com conteúdo encapsulado por trás de uma interface. A interface provê uma separação explícita entre o lado interno e externo de um componente, definindo *o que* ele implementa mas escondendo *como* ele é implementado.

De acordo com a abordagem utilizada, a interface pode trazer descrições de serviços, eventos, métodos, funções, etc. Independente da nomenclatura utilizada, eles podem ser utilizados desde que sejam respeitadas as restrições do contrato. Um componente pode ter múltiplas interfaces, cada uma representando diferentes serviços oferecidos pelo componente.

Um contrato representa a descrição de interface, ou seja, a especificação do comportamento de um componente [84]. Um contrato lista as restrições que o componente deverá manter (*invariante*). Para cada operação do componente, o contrato também lista as restrições que precisam ser satisfeitas pelo cliente (pré-condições) e aquelas que o componente promete estabelecer como retorno (pós-condições). A pré-condição, o invariante e a pós-condição constituem a especificação do comportamento de um componente.

Como mencionado anteriormente, o processo de desenvolvimento baseado em componentes é fundamentado na *junção de pedaços* de software [22]. Para que isso ocorra, é necessário que haja uma especificação de como os componentes devem ser compostos e devem interagir entre si. Esta especificação de composição e interação entre componentes é denominada *modelo de componentes*. Apesar das diferentes características atribuídas a modelos de componentes por diferentes autores, há um consenso sobre a definição de modelo de componentes, como descrito na Definição 2.2, baseada em [25].

**Definição 2.2 (Modelo de Componentes)** Um modelo de componentes é um conjunto de padrões e convenções bem definidos para a construção de componentes e composição de aplicações baseadas em componentes.

Se por um lado um modelo de componentes especifica padrões e convenções para a construção de componentes, a infra-estrutura de suporte que impõe tais padrões, no nível de software, é denominada arcabouço de componentes (*component framework*) [25]. Um arcabouço de componentes gerencia os recursos compartilhados pelos componentes e provê os mecanismos que permitem a interação entre eles (Definição 2.3).

**Definição 2.3 (Arcabouço de Componentes)** Um arcabouço de componentes é uma infra-estrutura de software que implementa um modelo de componentes, fornecendo suporte ao desenvolvimento de componentes e à composição de aplicações e gerenciando a interação entre os componentes.

Em um nível mais pragmático, arcabouços de componentes fornecem uma API (*Application Programming Interface*) para o desenvolvimento de componentes, seja por herança de classes e sobreposição de métodos (arcabouços caixa-branca) ou por composição de módulos (arcabouços caixa-preta). As diversas tecnologias de componentes, dentre as mais famosas EJB [39] e CCM [85], possuem arcabouços que utilizam herança para a construção de novos componentes e composição para a integração entre os componentes (arcabouços caixa-cinza). Além disso, arcabouços vêm em geral acompanhados de uma infra-estrutura de gerência do ciclo de vida dos componentes e execução da aplicação, denominada *middleware*.

### 2.1.2 Componentes de Prateleira

Existem diversas definições para componentes de prateleira, ou COTS (*Commercial Off-The-Shelf*). Com base nas definições apresentadas em [86, 87, 88, 24, 89], define-se componentes de prateleira no contexto deste trabalho (Definição 2.4).

**Definição 2.4 (Componente de Prateleira)** Um componente de prateleira é um componente de software desenvolvido por terceiros, não destinado a uma aplicação específica, com código fonte não disponível e que pode ser comprado ou licenciado.

Ao contrário do paradigma tradicional de desenvolvimento de componentes *in-house*, o desenvolvimento de aplicações usando componentes COTS considera que a maior parte das suas funcionalidades devem ser providas por componentes desenvolvidos por terceiros. Essa característica

torna potencialmente o desenvolvimento mais rápido, com esforço reduzido e maior qualidade. Uma vez que os produtos já estão disponíveis e não há recursos requeridos para o desenvolvimento, tem-se também um custo reduzido [23].

As funcionalidades são implementadas pelos componentes desenvolvidos por terceiros e, desta forma, o esforço maior está sobre a integração dos componentes. Este esforço é, em geral, maior que o necessário à integração de componentes desenvolvidos *in-house*. Isto ocorre pois os componentes de prateleira foram desenvolvidos considerando um domínio e não uma aplicação específica e, portanto, necessitam ser personalizados e/ou adaptados. Sendo assim, um modelo de componentes com suporte à COTS deve possuir mecanismos de adaptação e personalização dos componentes sem a necessidade de alteração do código fonte, uma vez que os COTS são caixa-preta.

### 2.1.3 Ciclo de Desenvolvimento Baseado em Componentes

Existem duas principais vertentes referentes ao ciclo de desenvolvimento baseado em componentes: o desenvolvimento *de* componentes e o desenvolvimento de aplicações *com* componentes. O ciclo básico de desenvolvimento baseado em componentes é ilustrado na Figura 2.1. Os passos de 1 a 6 dizem respeito à engenharia de aplicações. Considerando que todos os componentes necessários a um dado sistema já tenham sido desenvolvidos e disponibilizados em um repositório de componentes, o esforço de desenvolvimento é direcionado à adaptação e integração [90].

Se durante a identificação dos componentes necessários ao sistema percebe-se que não há um componente que implemente uma determinada funcionalidade requerida, este componente deve ser desenvolvido (passo 3.1 na Figura 2.1). Inicia-se então um ciclo de desenvolvimento do componente independente do ciclo da aplicação. Este componente não deve ser implementado apenas para esta aplicação e sim de forma que possa ser reutilizado em outras aplicações do mesmo domínio. Após a conclusão do ciclo de desenvolvimento do componente, este deve ser disponibilizado no repositório para poder ser reutilizado no contexto do ciclo de desenvolvimento da aplicação.

Seguindo este processo, tem-se um repositório de componentes cada vez mais completo e, na medida em que os componentes vão sendo reutilizados e atualizados, aumenta-se também a qualidade da implementação dos mesmos. Com o tempo, o esforço de desenvolvimento vai sendo

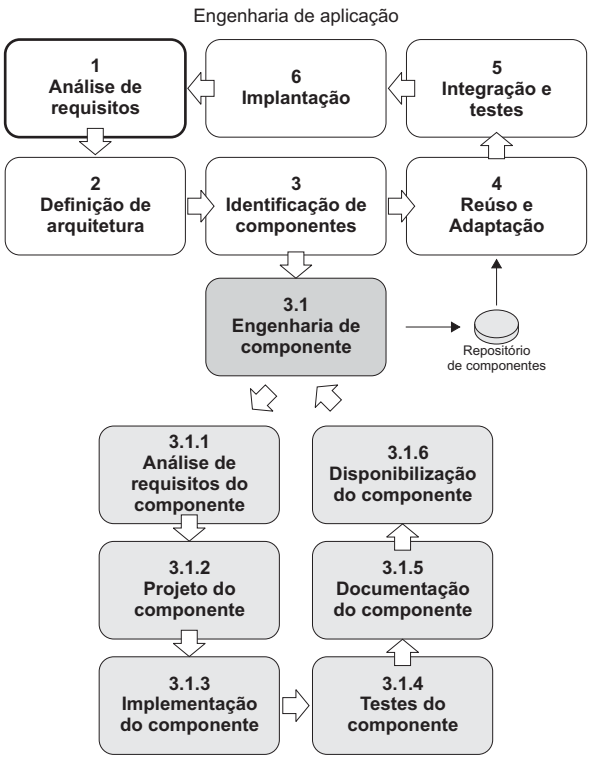



Figura 2.1: Ciclo de desenvolvimento baseado em componentes.

reduzido e, potencialmente, o tempo e o custo gastos no desenvolvimento de aplicações também.

### 2.1.4 DBC no Escopo do Trabalho

No escopo deste trabalho considera-se o ciclo padrão de DBC, com fortes características relacionadas a componentes de prateleira: adaptação e personalização. Vários outros investimentos ainda se mantêm em aberto, dentre eles: certificação, qualidade, recuperação automática ou auxiliada de componentes do repositório, etc.

O foco do trabalho aqui proposto está definido sobre a concepção de um modelo de componentes e um conjunto de mecanismos de engenharia que permitam a aplicação do mesmo, visando prover suporte à evolução dinâmica não antecipada. Apesar de entender a necessidade em abordar outras características, manteve-se o foco a fim de desenvolver uma infra-estrutura potencialmente utilizável, desde a concepção do software até a sua execução e evolução.

Acredita-se que, com a base conceitual e ferramental proposta, torna-se mais simples contemplar em trabalhos futuros outras características importantes de desenvolvimento baseado em componentes que ainda não são abordadas atualmente, tais como qualidade, certificação e recu-

peração automática de componentes de um repositório.

## 2.2 Evolução Dinâmica de Software Não Antecipada

Nesta seção são apresentados os conceitos relacionados à Evolução Dinâmica de Software Não Antecipada (EDSNA) e a aplicação destes conceitos no contexto deste trabalho. Como forma de facilitar o entendimento da EDSNA, define-se, inicialmente, evolução de software, evolução dinâmica de software e evolução de software não antecipada, destacando a motivação e as características de cada uma delas. Define-se, por fim, a EDSNA, com base nas definições anteriores.

### 2.2.1 Evolução de Software

Evolução é a mais longa e cara das fases inerentes ao ciclo de vida de um software [91]. Dentre as atividades referentes à evolução, incluem-se: manter-se em evolução continuamente para reparar problemas de funcionamento; adicionar novas funcionalidades; melhorar funcionalidades existentes (desempenho, extensibilidade, etc); adaptar-se a um novo ambiente de execução ou plataforma; e prevenir potenciais problemas [92]. Estas atividades foram classificadas por Lientz e Swanson [3] de acordo com os tipos de evolução descritos a seguir.

- **Corretiva -** evolução com atividades ditas como reativas, pois estão relacionadas à requisição do usuário para a correção de problemas funcionais ou não funcionais, como desempenho, por exemplo.
- **Adaptativa -** evolução com atividades relacionadas à adaptação do sistema a um novo ambiente ou plataforma de execução.
- **Aperfeiçoativa -** evolução com atividades relacionadas a satisfazer melhor os requisitos do usuário, seja para melhorar o desempenho, implementar novos requisitos ou aumentar a facilidade de manutenção/evolução (*maintainability*/*evolvability*).
- **Preventiva -** evolução com atividades relacionadas a preparar o sistema para potenciais mudanças futuras.

Os tipos de evolução acima descritos foram posteriormente refinados em [93] para representar o propósito da mudança. Porém, de uma forma geral, estes quatro tipos são representativos dos demais definidos em [93].

Ao observar a descrição dos tipos de evolução nota-se que o foco está sobre a manutenção da satisfação dos requisitos do usuário. Partindo deste princípio, com base na definição descrita em [1], tem-se a definição de evolução de software no contexto deste trabalho (Definição 2.5).

**Definição 2.5 (Evolução de Software)** Evolução de software é 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.

### 2.2.2 Evolução Dinâmica de Software

As atividades de evolução de software podem ocorrer em diferentes fases do seu ciclo de desenvolvimento, as quais podem ser classificadas em três categorias principais: em tempo de compilação (*compile-time*); em tempo de carregamento (*load-time*); e em tempo de execução (*run-time*). Estas três categorias de evolução, descritas a seguir, estão relacionadas a *quando* uma mudança é incorporada a um sistema de software (*time of change*) [94].

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

A partir da classificação das três categorias de evolução, tem-se a definição de evolução dinâmica de software no contexto deste trabalho (Definição 2.6).

**Definição 2.6 (Evolução Dinâmica de Software)** A Evolução Dinâmica de Software é um tipo de evolução que ocorre sem a interrupção da execução do software.

Na literatura, outros termos são utilizados para designar evolução dinâmica de software, tais como, *evolução de software em tempo de execução* (*run-time software evolution* [95]) e *evolução on-line de software* (*on-line software evolution* [96]). Outros termos relacionados também encontrados na literatura são: *atualização dinâmica de software* (*dynamic software updating* [97]), *software reconfigurável dinamicamente* (*dynamically reconfigurable software* [98]), *modificação de programas on-the-fly* (*on-the-fly program modification* [99]) e *composição dinâmica de software* (*dynamic software composition* [100]).

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 tais como de segurança ou financeiras. De acordo com um relatório do Yankee Group [101], a perda financeira pode chegar a até US$2,6 milhões em bancos e, em corretoras de bolsa de valores, a até US$4,5 milhões por hora de interrupção de sistema (*downtime*). Historicamente, a perda pôde ser observada em um grande site de leilões que teve seu sistema interrompido por 22 horas, com prejuízo de cerca de US$ 3 milhões [33].

Exemplos bem conhecidos de tais sistemas são serviços Web, sistemas de telecomunicação, sistemas bancários, sistemas de controle de tráfego aéreo e sistemas militares [2]. Além destes exemplos, o surgimento de novos paradigmas, tais como *grids* computacionais [102, 103], computação orientada a serviços [27], computação pervasiva [34] e computação autônoma [104], tem estendido o escopo de sistemas de software ininterruptos.

Um dos principais problemas inerentes à evolução dinâmica é a manutenção do funcionamento. Uma vez que a evolução ocorre durante a execução do sistema, não é possível realizar testes exaustivos e iterações de validação mediante os cenários de evolução em um ambiente controlado de homologação [105]. As alterações decorrentes da evolução serão automaticamente refletidas no sistema em produção e utilizadas por seus usuários.

Por isso, sistemas que permitem evolução dinâmica devem possuir mecanismos para garantir que a integridade do sistema seja preservada para que não haja problemas no seu funcionamento [106]. No caso de um sistema baseado em componentes, deve-se garantir que a evolução em um componente não irá causar problemas nos outros componentes, na arquitetura do sistema como um todo e nas dependências entre os componentes. Da mesma forma, a arquitetura do sistema pode evoluir, funcionalidades podem se tornar desnecessárias e seus componentes pro-

vedores podem ser removidos. Em qualquer destes cenários, deve-se buscar manter um sistema funcional e validado do ponto de vista do usuário.

Algumas soluções para evolução dinâmica no contexto de desenvolvimento baseado em componentes vêm sendo propostas, como discutido no Capítulo 3. Em sistemas baseados em componentes, as dependências entre os componentes são estabelecidas através de interfaces bem definidas e, portanto, existe uma maior independência entre os diversos componentes do sistema. Em tempo de execução, cada componente também possui seu próprio ciclo de vida. Estas características tornam mais simples a gerência de evolução dinâmica baseada em componentes, se comparada aos paradigmas estruturado e orientado a objetos [96].

### 2.2.3 Evolução de Software Não Antecipada

Durante as fases de projeto e implementação de um software é possível antecipar potenciais cenários de evolução. Com base nessa previsão de evolução, podem ser criados pontos de extensão (ganchos) para acomodar as mudanças no software caso estas ocorram [107].

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, levando a um projeto mais flexível do que demandam os seus requisitos e aumentando o custo e o tempo de desenvolvimento [32].

Evidentemente, dada a ausência de preparação de um projeto de software para mudanças, tem-se um impacto muito maior sobre o software quando tal mudança ocorre. A maioria das complicações técnicas e gastos relacionados à evolução de software ocorrem quando as mudanças não foram previstas no projeto inicial do mesmo.

Esse tipo de evolução relacionada a mudanças para as quais o software não foi preparado durante o projeto e implementação inicial do software é denominada Evolução Não Antecipada. Sem suporte à evolução não antecipada, mudanças não previstas geralmente forçam desenvolvedores a realizar modificações no código e no projeto existentes [31].

Com base nestas características e na definição descrita em [32], tem-se a definição de evolução de software não antecipada (Definição 2.7).

**Definição 2.7 (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.

No contexto de desenvolvimento baseado em componentes, mudanças não antecipadas estão relacionadas à alteração de componentes existentes, adição de novos serviços não previstos, adição de novos módulos arquiteturais contendo conjuntos de componentes pré-existentes, dentre outras possíveis alterações neste nível de granularidade. No caso da inserção de novos componentes, deve-se considerar a possibilidade de fazer com que seus serviços sejam acessados pelos componentes existentes.

### 2.2.4 Evolução Dinâmica de Software Não Antecipada - EDSNA

A Evolução Dinâmica de Software Não Antecipada, como o próprio nome sugere, reúne as características dos dois tipos de evolução discutidos anteriormente. Com base nas definições 2.6 e 2.7, tem-se a definição de evolução dinâmica de software não antecipada (Definição 2.8).

**Definição 2.8 (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 no mesmo.

Gerenciar a evolução dinâmica é possível através de técnicas como carregamento dinâmico de classes e reflexão computacional. Um exemplo bem conhecido de uma arquitetura baseada em *plug-ins* que provê evolução dinâmica antecipada é a plataforma Eclipse [59]. A antecipação é definida na interface dos *plug-ins*, sendo possível carregar *plug-ins* que implementam esta interface dinamicamente. Porém, o núcleo da plataforma, que não contém pontos de extensão pré-definidos não pode ser evoluído dinamicamente. Nestes casos, uma infra-estrutura para evolução dinâmica não antecipada é necessária.

O processo de evolução dinâmica não antecipada definido em [2], adaptado para o contexto de desenvolvimento baseado em componentes, possui os passos descritos a seguir.

1. **Atividades *off-line*** - Antes de ocorrer a atualização do software, o componente a ser inserido ou alterado deve ser implementado. As atividades *off-line* se iniciam com a localização de todas as estruturas afetadas por uma dada mudança, ou seja, deve-se encontrar as dependências do componente sendo evoluído. O novo componente é então implementado ou

reutilizado e adaptado de acordo com os requisitos do sistema. Por fim, o comportamento do componente a ser inserido no sistema deve ser verificado juntamente com a especificação do sistema para identificar problemas na corretude durante a evolução. Pode-se também realizar o teste de integração através da implantação do componente em uma plataforma de homologação, com uma cópia do sistema em execução.

2. **Implantação do novo componente no sistema** - A complexidade de introduzir um novo componente no sistema em execução depende da linguagem de programação e da plataforma de componentes utilizada como base. Essa atividade é mais facilmente gerenciada em linguagens como Java e CSharp que em linguagens compiladas para código nativo, como C, por exemplo. Em caso de abordagens que permitem várias instâncias de um determinado componente, seguindo uma extensão do paradigma orientado a objetos, deve-se considerar também a transferência de estado do componente antigo para o novo. Na infra-estrutura proposta neste trabalho, a transferência de estado não é abordada.

3. **Bloquear requisições dos componentes afetados** - Uma vez que a evolução do software em execução pode resultar em inconsistências que podem levar a situações de erro, deve-se preservar a consistência bloqueando as requisições dos componentes da aplicação que são afetados pela mudança durante o processo de evolução. Uma outra alternativa, que foi implementada pela infra-estrutura proposta neste documento, é manter a versão antiga funcional até que a nova versão do componente esteja pronta para receber requisições.

4. **Evolução dos componentes afetados pela mudança** - A inserção ou alteração do componente durante o processo de evolução pode requerer a mudança nos componentes dependentes. Esta evolução, *a priori*, segue o mesmo processo aqui descrito, causando um efeito cascata. Vários componentes podem ser alterados e a consistência do sistema deve sempre ser mantida.

5. **Verificação *on-line*** - Uma vez que o componente está inserido e as dependências resolvidas, pode-se avaliar um número de condições e invariantes sobre a nova versão do sistema. Caso alguma propriedade desejada do sistema não seja verificada, deve-se desfazer o processo de evolução (*rollback*). Este passo do processo não é contemplado pela abordagem proposta neste documento.

6. **Reativação dos componentes afetados** - Caso os componentes dependentes tenham sido desativados ou bloqueados no passo 3, deve-se reativá-los.

Um dos principais problemas ao prover uma infra-estrutura para EDSNA é a transparência para o desenvolvedor. O uso de técnicas que escondam a complexidade da execução dos passos descritos anteriormente é imprescindível ao sucesso da abordagem. Tal infra-estrutura deve vir acompanhada de uma técnica para o desenvolvimento de software, escondendo do desenvolvedor os mecanismos através dos quais o software, após construído, possa ser evoluído dinamicamente, mesmo considerando mudanças não antecipadas. Prover esse nível de transparência não tem sido a tônica dos diversos trabalhos relacionados a este tema, como descrito no Capítulo 3, fazendo com que o desenvolvedor precise gerenciar este processo de evolução no nível de implementação.

### 2.2.5 EDSNA no Escopo do Trabalho

O foco deste trabalho em relação à EDSNA está sobre a evolução funcional e estrutural do software. Tem-se como objetivo prover mecanismos para o desenvolvimento de software baseado em componentes com suporte transparente à evolução dinâmica não antecipada. Como descrito no Capítulo 4, os componentes neste trabalho são entidades funcionais, com um conjunto de serviços e eventos a serem providos a outros componentes da aplicação. Não são considerados componentes como instâncias, assim como na orientação a objetos. Sendo assim, não há suporte direto à transferência de estado dos componentes.

Uma outra característica não contemplada pela abordagem é a evolução de dados na aplicação. Caso o modelo de dados do sistema tenha que ser evoluído dinamicamente, o desenvolvedor não encontrará suporte na infra-estrutura proposta para realizar esta tarefa.

Dos cenários de evolução definidos anteriormente, não são contemplados aqueles relacionados à mudança de plataforma, ou seja, alterar a plataforma (sistema operacional ou hardware) na qual o software está implantado enquanto este se encontra em execução (evolução dinâmica adaptativa).

Por fim, a análise de corretude do software em tempo de execução também não é contemplada. Considera-se que essa análise deve ser feita antes da evolução do software, em tempo de (re)projeto, para evitar comportamentos de execução indesejados. Uma vez evoluído, desfazer a mudança se torna extremamente complicado e não foi considerado como um foco para este trabalho.

## 2.3 Alloy

Nesta seção são apresentados conceitos relacionados ao método formal Alloy como uma solução para a especificação e análise formal de software. Em seguida, descreve-se a aplicação destes conceitos no contexto deste trabalho.

### 2.3.1 O Método Formal Alloy

Alloy é um método formal proposto por Daniel Jackson do *Massachusetts Institute of Technology* (MIT) aplicado à especificação e análise de software. A concepção de Alloy foi motivada pela não disseminação da aplicação de métodos formais na indústria de software. De acordo com Jackson, este problema possui duas causas principais: sintaxe matemática, que intimidam os projetistas de software; e ausência de ferramentas, que impedem a produtividade do processo [60].

A proposta de Alloy é reunir o melhor dos mundos, buscando na especificação formal a idéia de uma notação expressiva e precisa, porém leve [108], e provendo uma ferramenta de análise completamente automática, com resultados imediatos ao desenvolvedor. Alloy tem sido utilizado em vários domínios específicos, tais como arcabouços arquiteturais [109], protocolos de Internet móvel [110], filtragem de mensagens [111], etc.

A linguagem Alloy é baseada em Z [112], de onde foram selecionadas as características essenciais de modelagem de objetos, incorporando algumas construções de outras notações mais recentes [111]. A seguir são descritos os principais conceitos relacionados à linguagem Alloy:

- **Átomos e Relacionamentos -** Um átomo é uma entidade que possui um tipo, é indivisível, imutável e não interpretável. Alloy usa átomos para modelar algo do mundo real ou alguma propriedade do sistema. Relacionamentos agrupam átomos em tuplas, que são seqüências ordenadas de átomos. Relacionamentos também possuem tipo e podem conter uma ou mais tuplas de átomos de tipos específicos, ou nenhuma tupla (relacionamento vazio). É importante notar que Alloy não diferencia entre escalas (um átomo), tuplas (um conjunto ordenado de átomos) e um conjunto de tuplas (um relacionamento). Eles são tratados da mesma forma.

- **Assinaturas -** Uma assinatura consiste de um tipo básico e um conjunto de átomos. Assinaturas são declaradas com a palavra chave `sig` e criam um tipo implícito que não pode ser

referenciado explicitamente. Por exemplo, a instrução descrita na Listagem de Código 2.1 cria assinaturas do tipo `Name` e `Address` (linha 1) e `Book` (linha 3) [60]. Assinaturas podem também possuir campos, como em classes da orientação a objetos. Na linha 4 da Listagem de Código 2.1, declara-se um atributo pertencente à `Book` denominado `addr` que mapeia um nome para um único endereço. Esta unicidade é definida por `lone`.

Listagem de Código 2.1: Sintaxe da linguagem Alloy para *assinaturas*.

```
1 sig Name, Address {}
2
3 sig Book{
4     addr: Name -> lone Address
5 }
```

- **Fatos -** Um fato cria uma restrição nos relacionamentos que limita os seus possíveis valores. Um fato deve ser sempre avaliado como verdadeiro. Por exemplo, na Listagem de Código 2.2, cria-se um fato que determina que todo livro deve possuir um endereço associado.

Listagem de Código 2.2: Sintaxe da linguagem Alloy para *fatos*.

```
1 fact {
2     all b:Book | one b.addr
3 }
```

- **Assertivas -** Assertivas são instruções que devem ser verdadeiras sobre o sistema. Elas servem como verificadores que asseguram que o sistema está se comportando corretamente. Por exemplo, na Listagem de Código 2.3, descreve-se uma assertiva com a mesma expressão do fato apresentado anteriormente. Para verificar uma assertiva e obter contra-exemplos da mesma, utiliza-se o comando `check` (linha 4).

Listagem de Código 2.3: Sintaxe da linguagem Alloy para *assertivas*.

```
1 assert test{
2     all b:Book | one b.addr
3 }
4 check test
```

- **Predicados -** Da mesma forma que se utiliza assertivas para encontrar contra-exemplos, pode-se obter instâncias de uma dada especificação através de predicados. Por exemplo, na

Listagem de Código 2.4, descreve-se uma assertiva com a mesma expressão do fato apresentado anteriormente. Para encontrar instâncias de um predicado, utiliza-se o comando `run` (linha 5).

Listagem de Código 2.4: Sintaxe da linguagem Alloy para *predicados*.

```
1 pred test(b:Book){
2     all b:Book | one b.addr
3 }
4
5 run test
```

- **Funções -** Uma função é uma fórmula reutilizável que pode ser aplicada a um conjunto de parâmetros. Em Alloy, convenções determinam que o segundo parâmetro seja o valor de retorno, mas múltiplos parâmetros também podem ser usados como valor de retorno, se necessário. Funções são similares a funções em linguagens de programação funcionais, com exceção de que, em Alloy, elas são utilizadas como mecanismos de transição de um estado a outro.

- **Operadores -** Muitos dos operadores em Alloy estão fortemente relacionados com a lógica de primeira ordem. Existem três tipos de operadores em Alloy: lógicos, quantificadores e de conjuntos. Estes operadores são utilizados para especificar fatos, assertivas, predicados e funções.

Utilizando os conceitos definidos acima e a sintaxe de Alloy é possível especificar um software e as propriedades que devem ser contempladas por tal especificação. A especificação simples apresentada anteriormente é válida e completa. Evidentemente que existem várias outras construções da linguagem mas, de uma forma geral, elas apenas refinam os conceitos apresentados anteriormente. Para um estudo aprofundado sobre Alloy, recomenda-se a leitura do livro de Daniel Jackson [60].

A especificação descrita anteriormente pode ser analisada através de uma ferramenta Java, de código aberto, desenvolvida pelo *Software Design Group* do (MIT) denominada *Alloy Analyzer* [113]. A ferramenta implementa a geração de instâncias de invariantes, a simulação da execução de operações e a verificação de propriedades definidas em relação a uma dada especificação.

A ferramenta é implementada na linguagem Java e se encontra atualmente na versão 4.0, podendo ser obtida em `http://alloy.mit.edu`.

### 2.3.2 Alloy no Escopo do Trabalho

Alloy não será utilizada como base formal para o trabalho aqui proposto. Utiliza-se Alloy apenas como ferramenta para possibilitar a especificação e verificação de propriedades desejadas no sistema para analisar se um dado cenário de evolução irá impactar a corretude do sistema.

Durante o desenvolvimento da aplicação, o desenvolvedor deve definir um modelo Alloy para o sistema e um conjunto de propriedades que devem ser verificadas. Evidentemente, apenas as partes críticas do sistema devem ser modeladas, visando uma maior eficiência no processo.

Ao adicionar um componente, pode-se então verificar se o modelo do componente, mesclado ao modelo da aplicação, mantém a corretude das propriedades iniciais. Este processo é automatizado através da utilização do *Alloy Analyzer*, que verifica a corretude das propriedades de acordo com todos os modelos, mesclados em um modelo único. Por fim, o resultado da análise é retornado ao desenvolvedor, indicando que as propriedades foram verificadas ou os cenários em que as propriedades não foram verificadas (contra-exemplos).

A motivação para a utilização de Alloy é definida em dois níveis, conceitual e de ferramenta, com foco na inserção da especificação e verificação formal no processo de desenvolvimento de software. Conceitualmente, o fato da linguagem ter características similares à abordagem orientada a objetos permite uma maior familiaridade de desenvolvedores de software com seu paradigma. Outros trabalhos [114, 115, 116] foram desenvolvidos utilizando Redes de Petri Coloridas [117] como formalismo. Porém, não foi encontrada uma maneira eficaz de inserir a modelagem e a verificação do software no processo de desenvolvimento. Neste caso, a mudança de paradigma e de ambiente de desenvolvimento dificultam a utilização do método formal no processo de construção de software.

No nível de ferramenta, o *Alloy Analyzer* é leve, de fácil extensão e de código aberto. Sendo assim, é possível realizar alterações que facilitem a integração deste com ferramentas de desenvolvimento de software, escondendo a complexidade do processo de verificação do desenvolvedor e, potencialmente, aumentando a produtividade. Além disso, a descrição da especificação de um sistema é textual, facilitando a manipulação da especificação e das propriedades a serem verifica-

das.

Neste contexto de ferramentas, em [118] foi desenvolvido um *plug-in* para a plataforma Eclipse para anotação e verificação de propriedades em código Java. Para isso, utilizou-se a linguagem Promela [119] e o verificador de modelos Spin [120], integrado ao *plug-in* do Eclipse. Neste caso, ocorreu o mesmo problema de mudança de paradigma na linguagem, dificultando o trabalho do desenvolvedor. Com a utilização de uma linguagem orientada a objetos como base para a especificação formal e a integração do *Alloy Analyzer* às ferramentas de desenvolvimento, espera-se inserir o passo de verificação formal ao processo sem trazer perdas significativas à produtividade do desenvolvedor.

# Capítulo 3

# Trabalhos Relacionados

Neste capítulo são apresentadas as abordagens correlatas à infra-estrutura proposta neste trabalho, agrupadas de acordo com o tipo de arquitetura e paradigma utilizados. A seguir, são apresentados os tópicos de comparação entre as abordagens. Tem-se, então, uma breve descrição das principais características de cada abordagem e da sua adequação aos requisitos definidos como tópicos de comparação. Por fim, os resultados da comparação são descritos em um quadro comparativo e discutidos em relação à infra-estrutura proposta neste trabalho.

## 3.1 Tópicos de Comparação

A seguir são apresentados os tópicos de comparação dos trabalhos relacionados, os quais se baseiam na motivação para uma infra-estrutura de suporte à EDSNA descrita no capítulo de introdução deste documento.

1. **Suporte à evolução dinâmica não antecipada -** Qual o suporte da abordagem à evolução não antecipada de software? Quais cenários de evolução não antecipada são contemplados (inserção, mudança e remoção de módulos, redefinição arquitetural, etc)? Qual o nível de transparência do processo para o desenvolvedor, ou seja, em que nível o desenvolvedor deve apontar ou não os possíveis pontos de extensão? Qualquer parte da aplicação desenvolvida pode ser evoluída ou apenas determinados módulos? A evolução pode ocorrer dinamicamente? Há suporte ao versionamento de módulos para garantir uma evolução gradativa, ou seja, há suporte à manutenção e co-existência de módulos de mesma funcionalidade?

2. **Suporte à composição de aplicações -** Existe um modelo ou técnica para disciplinar a organização dos módulos e composição da aplicação? Existe encapsulamento das funcionalidades em módulos coesos e fracamente acoplados para facilitar o processo de evolução da aplicação ou a evolução deve ser gerenciada no nível de linguagem/código? Há suporte à composição recursiva, ou seja, é possível compor novas aplicações com base em aplicações existentes?

3. **Especificação formal do modelo/técnica de desenvolvimento -** Há uma descrição formal, não ambígua, do modelo ou técnica de desenvolvimento?

4. **Suporte para aplicações corporativas -** Há suporte ao desenvolvimento de aplicações corporativas? Se sim, quais características são contempladas (transação, web, segurança, persistência, distribuição, etc)?

5. **Processo de desenvolvimento -** Há um processo/método para guiar o desenvolvedor na utilização da abordagem? A gerência de evolução dinâmica não antecipada é transparente em relação à aplicação do processo ou é explicitamente definida como uma fase a ser executada pelo desenvolvedor?

6. **Suporte à verificação formal no processo de evolução -** Há suporte à verificação formal para identificar possíveis impactos da evolução sobre a corretude da especificação do sistema? Esse suporte está vinculado ao processo de desenvolvimento?

7. **Ferramentas de auxílio ao desenvolvimento e à execução -** Existem arcabouços, ferramentas ou ambientes para permitir o desenvolvimento de software utilizando a abordagem proposta ou apenas uma descrição arquitetural/conceitual da mesma? Há ferramentas de suporte à verificação formal, composição da aplicação, disponibilização de módulos/componentes, etc? Existem ferramentas de apoio à execução de aplicações utilizando a abordagem proposta, ou seja, é possível desenvolver novas aplicações e disponibilizá-las em uma infra-estrutura de execução ou servidor de aplicação?

8. **Independência de linguagem e plataforma -** A abordagem é dependente de linguagem ou plataforma específica, ou seja, depende dos mecanismos e características de uma linguagem de programação ou de uma plataforma de hardware ou sistema operacional?

É importante observar que, na comparação realizada, não foram considerados apenas os trabalhos correlatos que preenchem todos os requisitos descritos acima. Da mesma forma, não foram apenas consideradas as abordagens baseadas em componentes. Todos os trabalhos encontrados na literatura relacionados ao desenvolvimento de aplicações com algum tipo de suporte à composição e à evolução de aplicações foram considerados. Esta foi a melhor forma encontrada para estabelecer um estado da arte na área na qual se insere o trabalho descrito neste documento.

## 3.2 Descrição dos Trabalhos

A seguir são descritos os trabalhos relacionados considerados relevantes para comparação com a infra-estrutura proposta, em ordem alfabética. Tal descrição se baseia em publicações de diversas fontes (revistas, conferências, livros, teses e dissertações, etc) encontradas na literatura. Uma vez que a lista apresentada não é exaustiva e considerando a potencial evolução destes trabalhos, sempre que se fizer menção à ausência de uma característica em uma dada abordagem, entenda-se que *não foram encontrados na literatura investimentos por parte dos próprios autores do trabalho ou de outros autores* para prover tal característica.

### 3.2.1 Balboa

Balboa [121, 122] é um ambiente de composição de software, inicialmente projetado para componentes de hardware, com suporte à evolução dinâmica não antecipada de aplicações desenvolvidas em C++. A arquitetura do ambiente é composta de três partes principais: uma linguagem de integração de componentes (CILL - *Component Integration Language*); um conjunto de bibliotecas de componentes C++; e um conjunto de interfaces, denominadas *Split-Level Interfaces* (SLIs), que estabelecem a ligação entre as duas primeiras partes.

Balboa utiliza uma CIL personalizada para instanciar e conectar componentes. As SLIs funcionam como conectores, provendo o nível de indireção necessário à evolução dinâmica não antecipada. Uma vez instanciados e conectados, os serviços de um componente C++ podem ser acessados diretamente ou através de um adaptador SLI (*wrapper*), que é implementado utilizando uma linguagem específica, denominada BIDL (*Balboa Interface Definition Language*) [122].

A seguir são apresentadas as características desta abordagem em relação aos tópicos de com-

paração descritos anteriormente:

1. **Suporte à evolução dinâmica não antecipada -** A abordagem provê suporte à evolução dinâmica não antecipada de aplicações. Isto é realizado através da linguagem de *script* de integração entre os componentes que, em tempo de execução, possibilita a inserção, remoção e alteração de componentes e dependências previamente definidas. Não é necessário especificar *a priori* os pontos de extensão da aplicação sendo desenvolvida.

2. **Suporte à composição de aplicações -** O modelo de composição está diretamente relacionado ao conceito de componentes e linguagem de *script* de integração (*glue code*). Tem-se um conjunto de componentes implementados em uma linguagem (C++) e uma linguagem de *script* que realiza as dependências entre os componentes e define a aplicação. A princípio, pode-se implementar a composição de aplicações, apesar dessa característica não ser provida explicitamente pela abordagem. Por isso, a composição de aplicações fica a cargo do desenvolvedor que deve gerenciar as dependências explicitamente.

3. **Especificação formal do modelo/técnica de desenvolvimento -** Não há especificação formal da técnica de composição de software.

4. **Suporte para aplicações corporativas -** Não há suporte ao desenvolvimento de aplicações corporativas.

5. **Processo de desenvolvimento -** Não há processo de desenvolvimento vinculado à utilização da abordagem.

6. **Suporte à verificação formal no processo de evolução -** Não há suporte à verificação formal vinculado à abordagem.

7. **Ferramentas de auxílio ao desenvolvimento e à execução -** O ambiente Balboa de composição possui suporte para gerenciar a evolução das aplicações desenvolvidas. Para o desenvolvimento da aplicação, tem-se as linguagens de *script* para a integração dos componentes. Estas linguagens são interpretadas para a execução da aplicação, o que tem sido foco de críticas em relação ao desempenho da abordagem [122].

8. **Independência de linguagem e plataforma -** A abordagem é dependente da linguagem de programação C++ para o desenvolvimento dos componentes. Porém, a arquitetura poderia ser implementada em outras linguagens, uma vez que não utiliza nenhuma característica específica de C++. Por outro lado, tem-se um acoplamento com as linguagens utilizadas como *script* de integração entre os componentes, tais como as SLIs e BIDL.

### 3.2.2 Beanome

Beanome [123, 124] adiciona uma "fina" camada sobre a arquitetura padrão de OSGi [42] para prover funcionalidades similares àquelas encontradas em modelos de componentes, de forma que aplicações possam ser construídas com base na montagem de instâncias de componentes. Beanome define um modelo de componentes e um arcabouço de software que implementa tal modelo [123].

O modelo de componentes Beanome é baseado no conceito de *tipos de componentes*. Estes tipos são identificados por um nome único e possuem um número de versão associado. A estrutura de um tipo de componente é definida em um arquivo XML, o qual é denominado descritor do componente. Vários componentes podem ser descritos em um único arquivo descritor. Cada tipo de componente é utilizado como modelo para criação de instâncias do mesmo. Para isso, no descritor do tipo de componente, são definidas as classes que implementam as funcionalidades do componente, assim como suas dependências.

O arcabouço Beanome provê a funcionalidade necessária à implementação do modelo de componentes e gerência de execução de aplicações. De fato, tem-se uma infra-estrutura de suporte à execução (*middleware*) composta por dois principais módulos: fábricas de componentes e registro de componentes. O registro mantém a lista de todas as fábricas disponíveis. As fábricas estão associadas aos tipos de componentes para criar instâncias dos mesmos. Este arcabouço é implementado como um *bundle* OSGi [42]. Da mesma forma, todos os componentes devem ser registrados como *bundles*, em um processo similar ao que ocorre na implementação do *Java Component Application Server* (JCAS), como mencionado no Capítulo 10.

Beanome pode ser apontado como uma evolução do arcabouço OSGi em termos de facilidade de configuração e definição de dependências. Além disso, torna o desenvolvimento baseado em OSGi mais voltado para o mundo de componentes, em contraponto à característica padrão de

orientação a serviços. Porém, exceto pelo modelo de composição mais bem definido, as características desta abordagem em relação aos tópicos de comparação descritos anteriormente continuam as mesmas de OSGi (ver Seção 3.2.9), como ilustrado a seguir:

1. **Suporte à evolução dinâmica não antecipada -** Há suporte à evolução dinâmica, porém não há suporte à evolução não antecipada. Isto ocorre porque as dependências entre os componentes são definidas explicitamente no nível de codificação. Com a utilização do XML como descritor das dependências, tem-se a possibilidade de transferir as alterações do código Java para o código XML. Mesmo assim, uma alteração nos serviços dos componentes causa alterações em outros serviços e, esta reação em cascata, deve ser gerenciada pelo desenvolvedor alterando os arquivos descritores dos componentes. Ou seja, o processo de evolução e (re)disponibilização dos componentes não é transparente.

2. **Suporte à composição de aplicações -** O suporte à composição de aplicações, principalmente no estabelecimento de dependências, pode ser considerado uma evolução em relação à OSGi. Componentes são descritos com base em descritores XML, assim como suas dependências e ligações com classes de implementação. Os componentes são disponibilizados todos no mesmo nível, como bundles OSGi e, portanto, não há suporte transparente à composição recursiva. O versionamento é garantido pelo próprio mecanismo de OSGi, externalizado para o descritor do XML do componente.

3. **Especificação formal do modelo/técnica de desenvolvimento -** Não há especificação formal da técnica de desenvolvimento.

4. **Suporte para aplicações corporativas -** Seguindo o modelo OSGi, tem-se suporte à distribuição e segurança já contemplado pela abordagem.

5. **Processo de desenvolvimento -** Não há processo de desenvolvimento vinculado à abordagem.

6. **Suporte à verificação formal no processo de evolução -** Não há suporte à verificação formal vinculado ao processo de evolução.

7. **Ferramentas de auxílio ao desenvolvimento e à execução -** Pode-se utilizar as ferramentas

de suporte à implantação de componentes e execução de aplicações que implementam a especificação OSGi, tais como Oscar [43] e Knoplerfish [44].

8. **Independência de linguagem e plataforma -** A abordagem é dependente da especificação OSGi e, conseqüentemente, dos mecanismos necessários para a sua implementação os quais são providos pela linguagem Java.

### 3.2.3 C2

A abordagem apresentada em [125, 126, 95] baseia-se na definição clássica de componentes e conectores para prover suporte à evolução dinâmica no nível arquitetural. A definição da abordagem emergiu de um estudo da evolução dinâmica de software em contextos específicos [126] e tem foco na utilização do estilo arquitetural híbrido de camadas, eventos e troca de mensagens, denominado C2. No estilo arquitetural C2, toda a comunicação entre os componentes é realizada via conectores, assim minimizando a dependência e facilitando atividades de evolução. O estilo também impõe restrições de topologia: todo componente tem um lado "superior" e "inferior", com uma única porta de comunicação em cada lado. Um conector C2 também tem os lados "superior" e "inferior", mas o número de portas de comunicação é determinado pela quantidade de componentes ligados a ele. Sendo assim, um conector pode acomodar inúmeros componentes ou outros conectores. Toda comunicação entre componentes é feita assincronamente, através de troca de mensagens mediada pelos conectores.

A seguir são apresentadas as características desta abordagem em relação aos tópicos de comparação descritos anteriormente:

1. **Suporte à evolução dinâmica não antecipada -** A abordagem provê suporte à evolução dinâmica não antecipada. Uma vez que não há restrições quanto à implementação dos componentes, qualquer parte do software, implementada como um componente, pode ser evoluída dinamicamente. Tem-se suporte à inserção, alteração e remoção de módulos, porém a reorganização arquitetural não é explicitamente abordada, assim como o versionamento dos componentes.

2. **Suporte à composição de aplicações -** A técnica para disciplinar a composição dos componentes segue a definição clássica de componentes e conectores: conectores estabelecem

a dependência e a comunicação entre componentes. Além disso, a interconexão de conectores permite uma "simulação" de arquitetura hierárquica, como na *Component Model Specification* (CMS), apresentada no Capítulo 4. Isto torna possível a transparência de funcionalidades de um subconjunto de componentes, vistos como uma fachada única através dos conectores, além de permitir uma solução alternativa à composição recursiva. Porém, esta característica não é transparente ao desenvolvedor, uma vez que o estilo C2 não foi concebido com esse intuito.

3. **Especificação formal do modelo/técnica de desenvolvimento -** Não há especificação formal do modelo de composição de componentes e conectores.

4. **Suporte para aplicações corporativas -** Não há suporte às características de aplicações corporativas.

5. **Processo de desenvolvimento -** Não há processo de desenvolvimento vinculado à abordagem.

6. **Suporte à verificação formal no processo de evolução -** Não há suporte direto à verificação formal vinculado às atividades de evolução. Porém, a arquitetura da ferramenta *ArchStudio* [95] prevê a integração com ferramentas externas para este fim.

7. **Ferramentas de auxílio ao desenvolvimento e à execução -** Foi implementado um arcabouço Java que provê uma API para o desenvolvimento de aplicações no estilo C2 [127]. Além disso, tem-se uma ferramenta denominada *ArchStudio* [95], composta por módulos gráficos para a especificação e evolução dinâmica dos componentes da arquitetura de um sistema.

8. **Independência de linguagem e plataforma -** O arcabouço e ferramentas desenvolvidas são utilizados especificamente para aplicações Java. Porém, o estilo arquitetural é independente de linguagem.

### 3.2.4 Chisel

Chisel [128] é um arcabouço para o desenvolvimento de aplicações com suporte à adaptação de serviços, considerando controle baseado em políticas e ciência de contexto. Tem como cenário de

motivação e aplicação a necessidade de adaptação em ambientes pervasivos, nos quais os cenários de evolução são imprevisíveis.

Chisel possui uma linguagem própria para a especificação de regras de adaptação com base nas informações de recursos presentes no ambiente e da própria aplicação sendo executada. Para garantir a evolução das regras e da aplicação, utiliza-se o arcabouço Iguana/J [129], que é um arcabouço baseado em reflexão e meta-objetos para a evolução dinâmica e não antecipada de software (ver Seção 3.2.7).

Uma vez que Chisel é apenas uma camada de suporte à auto-adaptação (*self-adaptation*) da aplicação e que o arcabouço Iguana/J é quem provê suporte à evolução dinâmica não antecipada, Chisel compartilha dos mesmos problemas de Iguana/J em relação aos tópicos de comparação. A principal exceção é que, com a linguagem de definição de regras, tem-se um mecanismo de mais alto nível para gerenciar a evolução do software e, também, suporte à característica de autonomia da aplicação.

A seguir são apresentadas as características desta abordagem em relação aos tópicos de comparação descritos anteriormente:

1. **Suporte à evolução dinâmica não antecipada -** Utilizando o arcabouço Iguana/J como base, há suporte à evolução dinâmica não antecipada de software. Tem-se um nível mais alto de gerência de evolução devido à especificação da linguagem do Chisel, porém isso é válido apenas para adaptações antecipadas pois, para novos tipos, é necessário baixar ao nível do Iguana/J e gerenciar a evolução diretamente no código. Além disso, tem-se o mesmo problema de Iguana/J: na medida em que o software evolui, torna-se mais difícil gerenciar novas evoluções no código devido ao espalhamento de meta-objetos.

2. **Suporte à composição de aplicações -** Não há suporte à composição de aplicações. De fato, esta característica tem incidência sobre o arcabouço de suporte, neste caso, Iguana/J, que também não tem suporte à composição de aplicações.

3. **Especificação formal do modelo/técnica de desenvolvimento -** Não há especificação formal da técnica de desenvolvimento.

4. **Suporte para aplicações corporativas -** Não há suporte ao desenvolvimento de aplicações corporativas.

5. **Processo de desenvolvimento -** Não há processo de desenvolvimento vinculado à abordagem.

6. **Suporte à verificação formal no processo de evolução -** Não há suporte à verificação formal vinculado ao processo de evolução.

7. **Ferramentas de auxílio ao desenvolvimento e à execução -** Não há ferramentas de auxílio ao desenvolvimento. Deve-se utilizar para a execução o ambiente de execução de Chisel que funciona sobre a extensão da JVM do Iguana/J.

8. **Independência de linguagem e plataforma -** O modelo de adaptação é independente de linguagem e plataforma, podendo inclusive ser utilizado como suporte à auto-adaptação na CMS. Porém, o modelo de evolução de software não antecipada é dependente do Iguana/J e, como visto na Seção 3.2.7, dos mecanismos da máquina virtual Java.

### 3.2.5 DAS

Na abordagem descrita em [130], propõe-se um modelo de desenvolvimento de software com suporte à auto-adaptação dinâmica denominado DAS e suas linguagens de descrição denominadas LEAD [131] e LEAD++ [132]. Um sistema de software baseado em DAS possui uma arquitetura de meta-nível [133], cujo mecanismo básico para auto-adaptação são os *procedimentos adaptáveis*. Um procedimento adaptável é uma variação de um procedimento (ou função) que possui um conjunto de métodos a serem selecionados na auto-adaptação. Estes métodos são invocados através de reflexão computacional, com base em estratégias de seleção de métodos definidas pelo desenvolvedor.

Este modelo foi implementado inicialmente como uma linguagem procedimental denominada LEAD [131] e depois como uma linguagem orientada a objetos estendendo a linguagem Java [132]. Em DAS, tem-se a definição de objetos de ambiente e objetos de eventos. Os primeiros armazenam o estado da aplicação e do ambiente no qual ela está inserida. Os objetos de evento, por sua vez, representam gatilhos que monitoram os objetos de estado e, de acordo com estes, invocam os métodos dos procedimentos adaptáveis.

Apesar de DAS não representar uma abordagem para evolução não antecipada de software, ela foi considerada uma abordagem relacionada por sua característica de evolução dinâmica. As-

sim como várias outras abordagens inseridas na área de software adaptativo (ou auto-adaptável), tem-se um conjunto de possíveis adaptações previstas *a priori*, as quais podem ser realizadas dinamicamente. Uma das abordagens que provê a evolução não antecipada na adaptação do software é Chisel [128] (ver Seção 3.2.4).

A seguir são apresentadas as características desta abordagem em relação aos tópicos de comparação descritos anteriormente:

1. **Suporte à evolução dinâmica não antecipada -** Há suporte à evolução dinâmica mas não há suporte à evolução não antecipada. É possível definir sob quais condições de estado determinados procedimentos são executados. Porém não é possível definir novos procedimentos em tempo de execução.

2. **Suporte à composição de aplicações -** Não há suporte à composição de aplicações provido pela abordagem. Porém, a abordagem poderia ser acoplada a um modelo de componentes e utilizar o mesmo conceito de monitoração e adaptação baseada em eventos.

3. **Especificação formal do modelo/técnica de desenvolvimento -** Não há especificação formal do modelo de desenvolvimento.

4. **Suporte para aplicações corporativas -** Não há suporte ao desenvolvimento de aplicações corporativas.

5. **Processo de desenvolvimento -** Não há processo de desenvolvimento vinculado à abordagem.

6. **Suporte à verificação formal no processo de evolução -** Não há suporte à verificação formal no processo de evolução.

7. **Ferramentas de auxílio ao desenvolvimento e à execução -** Não há ferramentas de auxílio ao desenvolvimento de software utilizando a abordagem. Há linguagens para o desenvolvimento procedimental e orientado a objetos e uma infra-estrutura para a execução e adaptação das aplicações desenvolvidas.

8. **Independência de linguagem e plataforma -** A abordagem DAS é independente de linguagem e plataforma, apesar de a linguagem orientada a objetos ter sido implementada como uma extensão de Java.

### 3.2.6 Hadas

HADAS (*Heterogeneous Autonomous Distributed Abstraction System*) [134, 41] é uma abordagem baseada em modelos distribuídos de objetos e tem foco na interoperabilidade entre componentes distribuídos. De acordo com o modelo de componentes definido em HADAS, o desenvolvedor deve especificar um conjunto de comportamentos *fixos* e um conjunto de comportamentos *extensíveis* para os componentes de sua aplicação. Apenas os comportamentos *extensíveis* podem ser dinamicamente alterados.

Cada componente possui todas as funcionalidades necessárias para a sua operação, incluindo aquelas relacionadas à evolução. Esta característica de auto-suficiência (*self-sufficiency*) é implementada como um conjunto de meta-métodos, definidos na *parte fixa* do componente para a manipulação de sua própria estrutura, a serem utilizados quando a evolução for necessária. Por exemplo, os meta-métodos para adição e remoção de métodos são `addMethod` e `removeMethod`.

O modelo de componentes do HADAS foi implementado em Java, utilizando-se de seus mecanismos de reflexão, carregamento dinâmico de classes, independência de plataforma e invocação remota de método, sendo este último necessário à implementação da arquitetura distribuída. Em primeira instância, HADAS foi concebido para dar suporte à interoperabilidade em aplicações distribuídas, sendo a capacidade de evolução dinâmica apenas uma das suas características adicionais. Isto explica a ausência de um processo de desenvolvimento para guiar o desenvolvedor nas atividades de evolução dinâmica de software não antecipada.

A arquitetura distribuída definida por HADAS possui dois conceitos principais: site e embaixador (*ambassador*). Um *site* é uma entidade representativa de um nó de rede, no qual embaixadores podem ser implantados e disponibilizados. Um *embaixador* é um representante de um determinado componente do sistema. Ele funciona como uma porta de acesso aos seus componentes, assim como os contêineres na CMS. Este nível de indireção permite a atualização de componentes mesmo em tempo de execução, bastando para isso atualizar o embaixador correspondente.

Em [135], apresenta-se uma evolução de HADAS em relação aos tipos de comportamentos definidos. O tipo *extensível* é particionado em *mutável*, *opcional* e *adicional*. Os tipos *adicional* e *opcional* estão relacionados ao acoplamento de novos componentes em tempo de execução, enquanto o tipo *mutável* é equivalente ao *extensível*. No contexto de evolução dinâmica não

antecipada, não há contribuição por parte desta extensão do HADAS, uma vez que, *a priori*, a evolução não pode ser prevista e assim, todos os componentes deveriam ser *extensíveis* [41] ou *mutáveis* [135].

A seguir são apresentadas as características desta abordagem em relação aos tópicos de comparação descritos anteriormente:

1. **Suporte à evolução dinâmica não antecipada -** HADAS provê suporte à evolução dinâmica não antecipada para todas as partes definidas como *extensíveis* em seus componentes. O desenvolvedor deve especificar tais partes *extensíveis*, o que torna o processo de evolução pouco transparente. Uma solução alternativa seria definir todas as funcionalidades dos componentes como extensíveis. Não há suporte direto para o versionamento dos módulos mas é possível publicar componentes de métodos iguais com nomes diferentes. Enfim, o suporte é provido mas não de forma transparente ao desenvolvedor.

2. **Suporte à composição de aplicações -** Tem-se um modelo de componentes vinculado ao desenvolvimento com base nas entidades *componente*, *site* e *embaixador*. As funcionalidades são encapsuladas nos componentes e a evolução pode ser realizada tanto no nível de serviços desses componentes (métodos) quanto no nível de embaixadores (alterando componentes, por exemplo). A estrutura de componentes não é hierárquica e não há composição recursiva, sendo assim, a composição de aplicações tem que ser realizada na granularidade de componentes.

3. **Especificação formal do modelo/técnica de desenvolvimento -** Não há formalização da técnica de desenvolvimento proposta.

4. **Suporte para aplicações corporativas -** Há suporte à distribuição, segurança e persistência. Estas funcionalidades são implementadas pelos embaixadores, o que as torna transparentes para o desenvolvedor.

5. **Processo de desenvolvimento -** Não há processo de desenvolvimento para suporte à aplicação da abordagem.

6. **Suporte à verificação formal no processo de evolução -** Não há suporte à verificação formal no processo de evolução.

7. **Ferramentas de auxílio ao desenvolvimento e à execução -** HADAS possui um ambiente de programação que consiste de um conjunto de ferramentas gráficas integradas para auxiliar o desenvolvedor nas atividades de construção de componentes e desenvolvimento e execução de aplicações. Tal ambiente foi implementado em Java, o que permite a integração com navegadores web através de *applets*.

8. **Independência de linguagem e plataforma -** A abordagem é independente de linguagem, apesar de apenas uma versão Java ter sido implementada. Contudo, os mecanismos necessários à implementação do modelo HADAS são providos também em outras linguagens, tais como reflexão, carregamento dinâmico de classes e invocação de método remota. Sendo assim, pode-se argumentar que HADAS é independente de linguagem e plataforma.

### 3.2.7 Iguana

Iguana é um modelo de programação reflexiva [136, 137, 129] desenvolvido como uma extensão para linguagens orientadas a objetos. Iguana provê um arcabouço para a definição de meta-tipos, implementados como meta-objetos [35], associados com os objetos sem alterar os seus tipos. Ele foi inicialmente introduzido como um modelo de linguagem independente para ser incorporado a linguagens de programação de alto nível, tais como C++ [136]. O suporte à adaptação dinâmica e não antecipada existe apenas na versão em Java, denominada Iguana/J [137].

Iguana provê suporte à programação de meta-nível, permitindo a definição de que partes do modelo de objetos devem ser inspecionadas e, então, passíveis de adaptação, mesmo em tempo de execução. Iguana/J permite, inclusive, que tipos não previstos durante o desenvolvimento sejam adicionados ao sistema através de uma linguagem própria, baseada em Java. Para isso, a máquina virtual Java (JVM) foi estendida utilizando a interface JIT (*Just-In-Time*).

Do ponto de vista de projeto e implementação de software, esta abordagem possui características similares às abordagens orientadas a aspectos, principalmente em relação à definição de pontos de inspeção e alteração/inserção de código. Além disso, pode-se utilizar Iguana na implementação de requisitos não funcionais, objetivando uma maior separação de interesses e evitando espalhamento e baixa coesão do código, que também são motivações para o uso de aspectos.

A seguir são apresentadas as características desta abordagem em relação aos tópicos de comparação descritos anteriormente:

1. **Suporte à evolução dinâmica não antecipada -** Há suporte à evolução dinâmica não antecipada de software. Porém, esta evolução é gerenciada no nível de código, através da especificação de meta-objetos e definição de novas funcionalidades a serem inseridas em tempo de execução. Não há transparência desse processo para o desenvolvedor. Além disso, assim como nas abordagens orientadas a aspectos, na medida em que a aplicação evolui, novos meta-objetos vão sendo adicionados e combinados ao código existente. Após alguns cenários de evolução, o entendimento e uma nova evolução no código tornam-se atividades complexas, principalmente levando-se em conta a potencial evolução do código definido nos próprios meta-objetos.

2. **Suporte à composição de aplicações -** Não há suporte à composição de aplicações. Pode-se considerar a aplicação da linguagem definida por Iguana como uma técnica de extensão da orientação a objetos. Ainda assim, não há a noção explícita de módulos ou componentes. Uma possível solução seria utilizar a idéia de arcabouços baseados em componentes que se integram ao conceito de aspectos, como o JAsCo [138, 139].

3. **Especificação formal do modelo/técnica de desenvolvimento -** Considerando a extensão da linguagem Java, tem-se uma especificação formal da linguagem de programação Iguana/J, porém, não do modelo de programação genérico de Iguana.

4. **Suporte para aplicações corporativas -** Não há suporte ao desenvolvimento de aplicações corporativas.

5. **Processo de desenvolvimento -** Não há processo de desenvolvimento vinculado à abordagem.

6. **Suporte à verificação formal no processo de evolução -** Não há suporte à verificação formal vinculado ao processo de evolução.

7. **Ferramentas de auxílio ao desenvolvimento e à execução -** Não há ferramentas de auxílio ao desenvolvimento. Para a execução, tem-se uma extensão da máquina virtual Java para permitir a interpretação da extensão da linguagem, o Iguana/J.

8. **Independência de linguagem e plataforma -** O modelo de programação de meta-objetos é independente de linguagem e foi inicialmente desenvolvido em C++, como mencionado

anteriormente. Porém, apenas Iguana/J permite adaptação dinâmica e, portanto, o modelo é dependente do mecanismo de extensão da JVM via JIT.

### 3.2.8 *Online Software Evolution* (OSE)

No trabalho descrito em [140] e, posteriormente, em [96], propõe-se uma abordagem baseada em componentes para a evolução dinâmica (*online*) de software. A implementação da abordagem é baseada em características presentes na linguagem Java, tais como reflexão computacional e o mecanismo de carregamento de classes. Utiliza-se um servidor de aplicação baseado em J2EE [141], denominado PKUAS (*Peking University Application Server*) [140]. Através da interface Web deste servidor, é possível atualizar os componentes de uma determinada aplicação.

Os componentes desenvolvidos seguem o modelo de *Enterprise Java Beans* [39]. A evolução nas aplicações é realizada no nível de interface e implementação dos componentes, o que se reflete em métodos e classes Java. Devido às restrições da máquina virtual de Java em relação ao carregamento de uma nova implementação para um método ou classe já carregada, o carregador de classes Java (*class loader*) teve que ser estendido.

A seguir são apresentadas as características desta abordagem em relação aos tópicos de comparação descritos anteriormente:

1. **Suporte à evolução dinâmica não antecipada -** Atividades de evolução relacionadas à remoção de componentes ou de operações das suas interfaces não são consideradas. A motivação para esta restrição é que, segundo os autores deste trabalho, a evolução deve manter a promessa de serviços dos componentes do ponto de vista de seus clientes. A nova versão de um componente deve ser compatível com a antiga. Por isso, considera-se apenas a adição e a atualização de implementações e a adição de novas operações às interfaces dos componentes.

   O desenvolvedor não tem que apontar antecipadamente as partes do software que poderão ser evoluídas, ou seja, qualquer parte da aplicação pode ser alterada em tempo de execução. Não há suporte ao versionamento de módulos, porém, mantém-se a consistência no processo de evolução, uma vez que, durante as atividades de evolução, as requisições para o componente a ser alterado são bloqueadas.

2. **Suporte à composição de aplicações -** O modelo para composição de componentes e aplicações é o *Enterprise Java Beans* [39]. Com isso, tem-se encapsulamento das funcionalidades para facilitar o processo de evolução de componentes. Porém, as alterações são sempre realizadas no nível de operações e interfaces das classes que implementam os componentes. Sendo assim, a evolução é gerenciada, do ponto de vista do desenvolvedor, em um nível de linguagem (método, classe...). A composição recursiva não é inerente a esta técnica de desenvolvimento e não são propostos mecanismos para implementá-la usando abordagens alternativas.

3. **Especificação formal do modelo/técnica de desenvolvimento -** Não há formalização da técnica de desenvolvimento proposta. Contudo, pode-se considerar a definição de *Enterprise Java Beans* como base para a composição de aplicações, apesar da falta de extensão deste modelo para características de evolução.

4. **Suporte para aplicações corporativas -** Uma vez que se baseia em *Enterprise Java Beans*, a abordagem herda todas as ferramentas de suporte para aplicações corporativas da plataforma J2EE [141] (transações, distribuição, segurança, persistência, balanceamento de carga, etc). Este trabalho pode ser apontado como o mais completo provedor de suporte para aplicações corporativas dentre as abordagens aqui discutidas.

5. **Processo de desenvolvimento -** Não foram encontrados na literatura investimentos relacionados à concepção de processos/métodos de desenvolvimento de software para guiar o desenvolvedor na utilização da abordagem proposta. Para tanto, processos de desenvolvimento baseados na tecnologia de *Enterprise JavaBeans* podem ser estendidos para contemplar características de evolução.

6. **Suporte à verificação formal no processo de evolução -** Não há suporte para a especificação e verificação formal no processo de evolução.

7. **Ferramentas de auxílio ao desenvolvimento -** Não há ferramentas gráficas para auxílio ao desenvolvimento especificamente direcionadas ao desenvolvimento de aplicações com suporte à evolução dinâmica não antecipada. Porém, do ponto de vista de construção das aplicações, podem ser consideradas ferramentas para o desenvolvimento de aplicações cor-

porativas baseadas em J2EE, tais como Eclipse [38] e NetBeans [142]. Em relação a arcabouços de desenvolvimento, o arcabouço disponibilizado por *Enterprise Java Beans* deve ser utilizado. Em relação à execução das aplicações, como mencionado anteriormente, foi desenvolvido um servidor de aplicação baseado em J2EE, denominado PKUAS, para a execução e gerência de evolução de aplicações.

8. **Independência de linguagem e plataforma -** A abordagem foi implementada especificamente para a linguagem Java, com base nos seus mecanismos de reflexão e carregamento de classes. Além disso, tal implementação funciona apenas no servidor PKUAS, cujo mecanismo de carregamento de classes estende o mecanismo padrão de Java e J2EE.

### 3.2.9 OSGi

OSGi (*Open Services Gateway initiative*) é uma especificação para a programação orientada a serviços utilizando a linguagem Java [42]. OSGi provê um ambiente orientado a componentes para serviços distribuídos. Esta especificação foi definida e é mantida pela *OSGi Alliance*, uma corporação independente sem fins lucrativos com foco na produção de tecnologia relacionada à interoperabilidade de aplicações e serviços baseada em OSGi como plataforma de integração de componentes.

OSGi provê suporte à gerência do ciclo de vida de componentes de software, denominados *bundles*, e seus serviços. A forma como a especificação é descrita está diretamente relacionada às características da linguagem Java. Por exemplo, um serviço OSGi corresponde a um arquivo JAR (*Java ARchive*) de Java, contendo código e outros recursos. Um serviço publicado por um determinado *bundle* somente estará disponível para ser utilizado por outro *bundle* caso tenha sido registrado anteriormente na plataforma de serviços, com base na nomenclatura de pacotes e classes Java.

Bundles e serviços podem ser instalados, atualizados e removidos dinamicamente, utilizando ambientes de execução que implementam a especificação OSGi, tais como Oscar [43] e Knopflerfish [44].

A seguir são apresentadas as características desta abordagem em relação aos tópicos de comparação descritos anteriormente:

1. **Suporte à evolução dinâmica não antecipada -** Há suporte à evolução dinâmica, porém

não há suporte à evolução não antecipada, pois as dependências entre os componentes são definidas explicitamente no nível de codificação, ao referenciar serviços pelo nome e versão. Ao alterar um serviço e implantar uma nova versão do mesmo, deve-se alterar o código dos *bundles* que utilizam aquele serviço.

2. **Suporte à composição de aplicações -** OSGi provê um modelo de composição baseado em *bundles* e serviços. Os *bundles* são disponibilizados todos no mesmo nível e, portanto, não há suporte transparente à composição recursiva. Há suporte à versionamento dos *bundles* OSGi.

3. **Especificação formal do modelo/técnica de desenvolvimento -** Não há especificação formal da técnica de desenvolvimento.

4. **Suporte para aplicações corporativas -** Esse é um dos pontos fortes da abordagem, principalmente ao que diz respeito à segurança e distribuição, sendo a última uma das grandes motivações para a concepção da abordagem.

5. **Processo de desenvolvimento -** Não há processo de desenvolvimento vinculado à abordagem.

6. **Suporte à verificação formal no processo de evolução -** Não há suporte à verificação formal vinculado ao processo de evolução.

7. **Ferramentas de auxílio ao desenvolvimento e à execução -** Possui ferramentas de suporte à execução e evolução dinâmica de aplicações, tais como Oscar [43] e Knoplerfish [44].

8. **Independência de linguagem e plataforma -** A abordagem é dependente dos mecanismos de carregamento de classes e reflexão da linguagem Java. Não há esforços para a implementação de OSGi em outras linguagens.

### 3.2.10 SEESCOA

SEESCOA (*Software Engineering for Embedded Systems using a Component Oriented Approach*) é um projeto desenvolvido por universidades da Bélgica que, como sugerido no próprio nome, tem como objetivo estudar a aplicação da abordagem de componentes para o desenvolvimento

de sistemas embarcados. No contexto desse projeto, propôs-se uma arquitetura de componentes para desenvolver sistemas embarcados com suporte à reconfiguração dinâmica. Um dos pontos fortes desta abordagem é a definição formal do modelo de componentes aliada a uma ferramenta de apoio ao desenvolvimento (CCOM [143]) e à execução do sistema desenvolvido. O foco da abordagem é sobre o suporte à evolução funcional, não abordando alterações estruturais.

A arquitetura de componentes SEESCOA é baseada no conceito de componentes e portas, de forma similar à abordagem C2 [95] (ver Seção 3.2.3). Os componentes são definidos em vários níveis (sintático, semântico, sincronização e qualidade de serviço) através dos quais realiza-se a reutilização dos mesmos.

A comunicação entre componentes ocorre via troca de mensagens assíncronas. Para enviar mensagens, todo componente possui um conjunto de portas. Cada porta é descrita através de um arquivo XML que determina o protocolo de comunicação utilizado pela mesma. Portas com protocolos compatíveis podem ser conectadas utilizando *conectores* e, assim, trocar mensagens. Utilizando as portas como mediadores da comunicação entre componentes facilita-se a evolução dinâmica não antecipada, uma vez que a alteração ou adição de um componente pode ser gerenciada através da reorganização das portas.

A seguir são apresentadas as características desta abordagem em relação aos tópicos de comparação descritos anteriormente:

1. **Suporte à evolução dinâmica não antecipada -** Há suporte à evolução dinâmica não antecipada do ponto de vista funcional do sistema. Portas podem ser re-organizadas e conectadas a novos componentes em tempo de execução, ou seja, novas funcionalidades podem ser inseridas e componentes existentes podem ser alterados. Não é necessário definir previamente os pontos de extensão do sistema, apesar de que alterações nas portas devem ser refletidas em seus descritores XML, onde está determinado que tipo de conexões são possíveis. Não há suporte direto ao versionamento de componentes. Para prover esta funcionalidade, uma solução alternativa seria criar portas com protocolos iguais e papéis de protocolos com nomes diferentes.

2. **Suporte à composição de aplicações -** Há um modelo de composição bem definido com base em componentes, portas e conectores. Todos os componentes estão em um mesmo nível arquitetural, ou seja, não há composição hierárquica ou suporte à composição de apli-

cações. As atividades de evolução são gerenciadas através da reconfiguração de portas em nível arquitetural e, eventualmente, em descritores XML.

3. **Especificação formal do modelo/técnica de desenvolvimento -** Não há uma especificação formal do modelo de desenvolvimento. Porém, tem-se uma formalização na descrição dos componentes, através de pré- e pós-condições e diagramas de seqüência de mensagens temporizados.

4. **Suporte para aplicações corporativas -** Não há suporte ao desenvolvimento de aplicações corporativas.

5. **Processo de desenvolvimento -** Não há processo de desenvolvimento vinculado à abordagem proposta.

6. **Suporte à verificação formal no processo de evolução -** Não há suporte à verificação formal vinculado ao processo de evolução.

7. **Ferramentas de auxílio ao desenvolvimento e à execução -** Tem-se um ambiente de suporte ao desenvolvimento de sistemas embarcados denominado CCOM [143]. O ambiente de execução dos componentes é implementado em Java e foi implantado em dispositivos embarcados executando sobre a máquina virtual Kaffe (KVM) [144].

8. **Independência de linguagem e plataforma -** A arquitetura de componentes de SEESCOA é independente de linguagem, apesar da única implementação disponível ser em Java.

### 3.2.11 Abordagens Formais

Algumas abordagens existentes estão relacionadas com o trabalho aqui proposto do ponto de vista formal. Apesar de tais trabalhos não abordarem diretamente evolução não antecipada de software eles possuem como objetivo verificar se um dado cenário de evolução de software vai de encontro à corretude da especificação do mesmo.

Em [145], por exemplo, propõe-se a utilização da linguagem Linda para especificar o comportamento de componentes do sistema, incluindo atributos internos e a interação com outros

componentes. A verificação das propriedades é baseada na noção de compatibilidade entre a especificação dos componentes e a do sistema em execução, assim como na composição de especificações apresentada no Capítulo 7. O principal problema da abordagem é que não há ferramenta para automatizar o processo de verificação. Além disso, não foram encontradas referências relativas à inserção da linguagem no processo de desenvolvimento de software integrada a outras linguagens de programação, sejam imperativas, funcionais ou lógicas.

Em [146], apresenta-se a abordagem Proteus. Trata-se de um formalismo para atualização dinâmica de software para linguagens baseadas em C. Programas em Proteus especificam dados e funções cujos nomes são mapeados para funções reais no programa em C. Dessa forma, pode-se utilizar este mapa de nomes para alterar e adicionar novas funcionalidades dinamicamente, respeitando as restrições e verificando as propriedades desejadas. Há também suporte à verificação automatizada. Proteus é indicado para atualização dinâmica de baixo nível, pois as alterações ocorrem a nível de código (funções, dados, ponteiros, etc).

No nível de modelagem, em [147] apresenta-se uma abordagem baseada em modelos para a especificação de sistemas legados com o objetivo de garantir corretude da especificação em cenários de evolução de tais sistemas. Em [148], utiliza-se uma extensão de linguagem de descrição de interfaces para especificar formalmente as interações entre os módulos do sistema e verificar o impacto da evolução do software na corretude do mesmo. Por fim, em [115], utiliza-se um arcabouço desenvolvido em redes de petri coloridas hierárquicas para a especificação e verificação de sistemas baseados em componentes. Nestas abordagens, as atividades de evolução sobre o modelo não são refletidas no software em execução, ou seja, restringem-se apenas ao nível de modelo.

O trabalho apresentado em [149] possui características similares à abordagem proposta. Tem-se um arcabouço de componentes para evolução dinâmica denominado OpenRec e um suporte baseado na linguagem Alloy [60] para verificação de propriedades do sistema. Assim como na abordagem proposta nesta tese, há suporte à automatização utilizando a ferramenta *Alloy Analyzer* [113]. A principal diferença entre as abordagens está na especificação da dependência entre os componentes. Com OpenRec, a ligação entre os componentes não é especificada através da definição de serviços e eventos, como no trabalho aqui proposto. Por fim, não há menção à inserção do método no processo de desenvolvimento baseado em componentes, em especial, componentes de prateleira.

### 3.2.12 Outros Trabalhos

Além dos trabalhos citados anteriormente, vários outros foram estudados e analisados [150, 151, 152, 153, 154, 155, 156, 157, 158], com menor nível de similaridade com o trabalho proposto. Por exemplo, vale ressaltar as tecnologias de componentes conhecidas e utilizadas na indústria, tais como: *JavaBeans* [159] e *Enterprise Java Beans* [39] da *Sun Microsystems*; *CORBA Component Model* [85] do *Object Management Group* (OMG) [160]; e o modelo de componentes da *Microsoft Corporation* que seguiu a evolução COM [161], DCOM [162] e, atualmente, .NET [163]. Estas tecnologias têm sido utilizadas com sucesso para a construção de aplicações corporativas, provendo infra-estrutura robusta de desenvolvimento e execução para tais aplicações. Contudo, tais modelos não foram concebidos para dar suporte à evolução dinâmica não antecipada. Em alguns casos, a tecnologia provê mecanismos e serviços para realizar mudanças em tempo de execução [50, 164] mas, uma vez que não possui arcabouço conceitual voltado a esta característica, tornam difícil a gerência de evolução e a transparência do processo para o desenvolvedor. No caso específico de EJB, uma possível integração com a abordagem OSGi pode significar um avanço considerável na concepção de um modelo Java padrão para evolução dinâmica de software.

O mesmo ocorre em tecnologias orientadas a serviços, tais como *Web Services* [165] e *Jini* [51]. Estas tecnologias, apesar de promover baixo acoplamento, não permitem evolução dinâmica não antecipada dos serviços providos. Uma solução alternativa no contexto de *web services* foi desenvolvida no contexto de um trabalho de conclusão de curso sob orientação do autor deste documento. Neste trabalho, propôs-se um suporte à atualização dinâmica de *web services* utilizando OSGi [74]. Contudo, não se tem uma técnica para o desenvolvimento de aplicações, e sim, uma solução alternativa para uma aplicação específica.

Diversas tecnologias orientadas a aspectos, tais como AspectWerkz [166], PROSE [167] e *Nanning Aspects* [168], com suporte às características de evolução dinâmica também têm sido propostas. Porém, tais abordagens possuem dois problemas principais: infra-estrutura de evolução dinâmica e gerência da evolução. O primeiro diz respeito ao fato de que os aspectos podem ser carregados dinamicamente contendo código relacionado à evolução. Porém, não há como carregar novos aspectos, não previstos em projeto, uma vez que o sistema está sendo executado. Ainda que isso seja possível, tem-se o problema da gerência da evolução. Na medida em que a aplicação evolui, novos aspectos vão sendo adicionados e combinados ao código existente. Após alguns

cenários de evolução, o entendimento e uma nova evolução no código tornam-se impraticáveis, principalmente considerando-se que o código definido nos aspectos também pode evoluir.

## 3.3 Considerações Finais

Neste capítulo foram apresentadas as principais abordagens relacionadas à infra-estrutura proposta neste trabalho. Na Tabela 3.1, apresenta-se o quadro comparativo de tais abordagens, considerando os tópicos de comparação previamente descritos. Observando este quadro, percebe-se que nenhuma delas possui um processo de desenvolvimento associado ou suporte à verificação formal vinculado à utilização da abordagem. Poucas abordagens possuem suporte à composição de aplicações e especificação formal da técnica de desenvolvimento. O foco principal das abordagens tem sido a construção de ferramentas para o desenvolvimento de software com suporte para a evolução dinâmica, não antecipada ou, em alguns casos, para ambas.

| Abordagem/Tópico | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Balboa | ● | ● | | | | | ● | |
| Beanome | | ● | | ● | | | ● | |
| C2 | ● | ● | | | | | ● | ● |
| Chisel | ● | | ● | | | | ● | ● |
| DAS | | | ● | | | | ● | ● |
| Hadas | ● | ● | | ● | | | ● | ● |
| Iguana | ● | | ● | | | | ● | ● |
| OSE | ● | ● | | ● | | | ● | |
| OSGi | ● | | | ● | | | ● | |
| SEESCOA | ● | | | | | | ● | ● |

Tabela 3.1: Quadro comparativo das abordagens correlatas.

Dada a existência de vários investimentos na área, a principal pergunta a ser respondida é: por que não aproveitar os esforços de alguma destas abordagens e adicionar o suporte necessário para contemplar os requisitos de evolução dinâmica não antecipada definidos como objetivos deste trabalho? De fato, apesar de algumas características já terem sido contempladas por tais aborda-

gens, a principal delas ainda se mantém em aberto: transparência para o desenvolvedor. Nenhuma destas abordagens foi concebida visando fornecer ao desenvolvedor uma técnica para desenvolver software de forma que, apenas pela utilização desta técnica, este software tenha suporte para evolução dinâmica não antecipada.

Outra característica importante não abordada pelos outros trabalhos é a possibilidade de composição recursiva, que permite a composição de aplicações seguindo a mesma técnica de composição de componentes. O modelo de composição proposto nesta tese permite versionamento, adaptação e personalização de execução de forma simples, seguindo o próprio mecanismo de interação dos componentes. Além disso, cada uma das características acessórias à especificação de componentes pode ser utilizada de acordo com a demanda do desenvolvedor. Isto pode ser observado no suporte para aplicações corporativas, cujos módulos (persistência, segurança, distribuição, etc) podem ser utilizados independentemente, de acordo com os requisitos da aplicação.

Por fim, do ponto de vista de ferramentas, a infra-estrutura proposta provê suporte ao desenvolvedor desde a construção de componentes e composição de aplicações até a execução e evolução dinâmica das aplicações. A iniciativa de projeto genérico de arcabouço de software que implementa a especificação de componentes não é encontrada em nenhum dos trabalhos relacionados. Na verdade, ao descrever na Tabela 3.1 que determinadas abordagens são independentes de linguagem, considera-se uma visão otimista uma vez que não são dados indícios no nível de projeto de que elas realmente poderiam ser implementadas em outras linguagens. Neste trabalho tem-se diversas implementações da especificação de componentes (Java, C++, Python e CSharp), que são linguagens com diferentes características, APIs e ambientes de execução.

# Capítulo 4

# Especificação do Modelo de Componentes

Neste capítulo apresenta-se a especificação do modelo de componentes (*Component Model Specification* - CMS) para o desenvolvimento de software com suporte à evolução dinâmica não antecipada. Ao utilizar a CMS, o desenvolvedor estará construindo software com suporte à evolução dinâmica não antecipada mas não terá que especificar *a priori* que parte do sistema poderá ser evoluída no futuro. Espera-se do desenvolvedor apenas que, diante de um conjunto de requisitos, utilize os conceitos da CMS para desenvolver o software (ver Figura 4.1). Os mecanismos que permitem a evolução são de responsabilidade da CMS e, evidentemente, do restante da infra-estrutura descrita nos próximos capítulos [169].

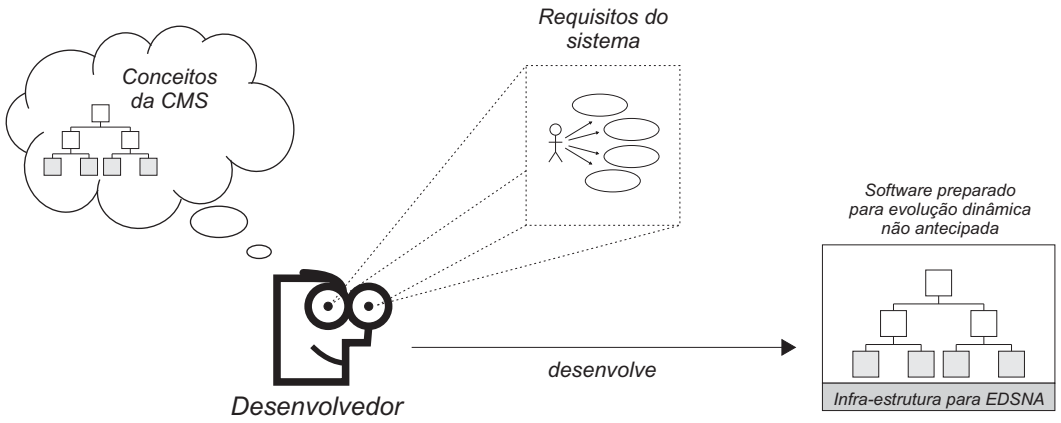



Figura 4.1: Desenvolvimento de software utilizando a CMS.

A idéia chave por trás da CMS é o desacoplamento entre os provedores de funcionalidades. Quando há referências diretas entre provedores, tem-se um maior impacto no software existente quando o mesmo evolui. Isto ocorre porque ao se alterar um determinado módulo do software,

todos os outros que possuem referência direta às funcionalidades do mesmo necessitam ser atualizados [170]. A CMS tem base no princípio de que a inexistência de referências diretas entre provedores de funcionalidades tem como conseqüência uma maior flexibilidade na inserção, remoção e alteração destes provedores, inclusive em tempo de execução.

A seguir apresenta-se a definição do núcleo da CMS, destacando seus mecanismos de composição e interação entre componentes. Depois disto, são discutidos aspectos pragmáticos da utilização da CMS, os quais estão relacionados à reutilização, adaptação, personalização e execução.

## 4.1 Definição do Núcleo da CMS

A seguir são apresentadas as definições relacionadas ao núcleo da CMS utilizando Teoria dos Conjuntos [171]. Para facilitar o entendimento de tais definições, dada uma tripla $A_i = \langle X, Y, Z \rangle$, denotar-se-á por $[X_{A_i}]$ o elemento $X$ da tripla, seja $X$ um elemento simples ou outro conjunto. Caso $X$ seja um conjunto, denotar-se-á por $[X_{A_i}]_j$ o elemento $x_j \in [X_{A_i}]$.

### 4.1.1 Sistema Baseado em Componentes

Na CMS, um Sistema Baseado em Componentes (SBC) é definido como descrito na Definição 4.1. Tem-se uma estrutura baseada no conceito de árvore enraizada [172], com dois tipos de entidades: componentes funcionais e contêineres, as quais serão definidas posteriormente. Os contêineres podem ser representados pelos ramos ou folhas da árvore, sendo a raiz da árvore denominada *contêiner raiz*. Já os componentes funcionais são representados apenas pelas folhas da árvore.

**Definição 4.1 (Sistema Baseado em Componentes)** De acordo com a CMS, um Sistema Baseado em Componentes, SBC, é definido como

$$SBC = \langle R, F, cr \rangle,$$

onde:

- $cr$ é um contêiner denominado **contêiner raiz**, sendo $cr \in R$;

- $R$ é o conjunto de todos os contêineres do sistema sendo $R \neq \emptyset$. O conjunto $R$ não pode ser vazio pois deve conter pelo menos o contêiner raiz;

- $F$ é o conjunto de todos os componentes funcionais do sistema.

Na Figura 4.2, ilustra-se a representação gráfica utilizada para descrever a arquitetura de um sistema baseado em componentes no restante deste documento. De acordo com esta figura, utiliza-se um retângulo branco para a representação gráfica de contêiner e um retângulo cinza para a representação gráfica de componente funcional. A profundidade da árvore é determinada de cima para baixo, sendo assim, os níveis da árvore são ordenados nesse sentido e o contêiner raiz é sempre o elemento no topo da hierarquia.

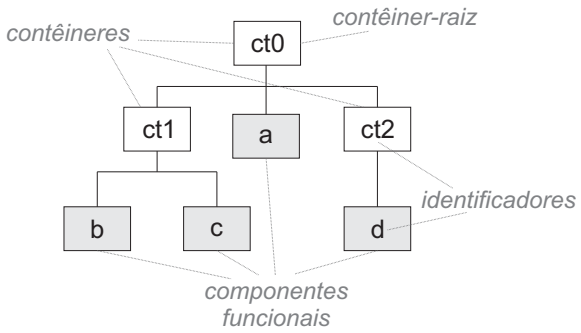



Figura 4.2: Representação gráfica de uma arquitetura de sistema baseado em componentes de acordo com a CMS.

Na Definição 4.1, os conceitos de *componente funcional* e *contêiner* foram utilizados para definir um sistema baseado em componentes. Estes conceitos possuem diferentes conotações na CMS: *contêineres* são entidades estruturais e *componentes funcionais*, como o próprio nome sugere, são entidades funcionais. Isto significa que a organização dos contêineres na hierarquia não interfere nas funcionalidades do sistema, as quais são implementadas e disponibilizadas apenas pelos componentes funcionais através de *serviços* e *eventos*, conforme descrito na Definição 4.2.

**Definição 4.2 (Componente funcional)** Um componente funcional, cf, é um artefato de software que implementa uma ou mais funcionalidades inerentes ao sistema e as disponibiliza em forma de serviços e/ou eventos. Para isso, depende de serviços e eventos disponibilizados por outros componentes funcionais. Sendo assim,

$$cf = \langle id, SP, SR, EA, EI \rangle,$$

onde:

- $id$ é o identificador do componente;
- $SP = \langle sp_0, sp_1, ..., sp_n \rangle$ é o conjunto de serviços providos, sendo $n \in \mathbb{N}$ e $sp_i = \langle ids, fns \rangle$, onde $ids$ é uma cadeia de caracteres que identifica o serviço e $fns$ representa a operação que implementa a execução do serviço $sp_i$. Denota-se por $fns(cf, ids, P) = R$, a operação que executa o serviço $sp_i$ tal que $[id_{sp_i}] = ids$, provido pelo componente $cf$, com parâmetros de entrada $P$ e resultado $R$;
- $SR = \langle sr_0, sr_1, ..., sr_m \rangle$ é o conjunto de serviços requeridos, sendo $m \in \mathbb{N}$ e $sr_m$ é uma cadeia de caracteres que identifica o serviço;
- $EA = \langle ea_0, ea_1, ..., ea_k \rangle$ é o conjunto de eventos anunciados, sendo $k \in \mathbb{N}$ e $ea_k$ é uma cadeia de caracteres que identifica o evento;
- $EI = \langle ei_0, ei_1, ..., ei_p \rangle$ é o conjunto de eventos de interesse do componente, sendo $p \in \mathbb{N}$ e $ei_i = \langle ide, fne \rangle$, onde $ide$ é uma cadeia de caracteres que identifica o evento e $fne(x)$ representa a operação que implementa a recepção do evento $ei_p$ . Denota-se por $fne(cf, ide, P)$, a operação que implementa a recepção do evento $ei_i$, tal que $[id_{ei_i}] = ide$, de interesse do componente $cf$, com parâmetros de entrada $P$.

A denominação “funcional” dada aos componentes da CMS foi motivada pelas diferenças entre tais componentes e os de outras tecnologias relacionadas, tais como EJB [39] e CCM [85]. De acordo com estas tecnologias, um componente pode representar entidades de negócio, contendo estado e um conjunto de operações inerentes, de forma semelhante a objetos na orientação a objetos.

Na CMS, utiliza-se uma definição mais tradicional de componente: interface bem definida e encapsulamento das funcionalidades. Por isso, dada a abrangência de tais abordagens na indústria, decidiu-se enfatizar o aspecto estritamente funcional do componente da CMS através da denominação *componente funcional*.

Vale ressaltar que isto não significa que os componentes funcionais na CMS não podem armazenar estado (*stateless*). Eles podem armazenar estado, mas este serve apenas como auxílio à

execução das funcionalidades (serviços e eventos) e não para identificar/instanciar entidades do modelo de negócio.

Uma vez esclarecida a motivação para a nomenclatura dos componentes funcionais[1], é importante detalhar os dois principais conceitos inerentes à definição de tais componentes: *serviços* e *eventos*. *Serviços* são implementações de funcionalidades providas por um componente e acessíveis por quaisquer outros da hierarquia. Para prover seus serviços, um dado componente pode requerer os serviços de outros componentes, o que é denominado *dependência de serviços* (ver Definição 4.3). Cada serviço possui um retorno da execução, de forma similar a uma função na programação estruturada ou a um método na programação orientada a objetos. Um dado serviço só pode ser implementado por um único componente, o que é determinado pelo identificador do serviço. Sendo assim, não pode haver dois componentes funcionais provendo serviços com mesmo identificador.

**Definição 4.3 (Dependência de Serviços)** Uma relação de dependência entre um serviço provido (sp) por um dado componente (cfa) e um serviço requerido (sr) por um dado componente (cfb), denotada por $dep_s(sp, sr)$, é definida se e somente se $[id_{sp}] = sr$, ou seja

$$dep_s(sp, sr) \Rightarrow ([id_{sp}] = sr)$$

Eventos são anunciados por componentes funcionais para a notificação de mudança de seu estado após algum processamento interno, como a execução de um serviço, por exemplo. A notificação de um evento não possui retorno. Para um dado evento, podem existir vários outros componentes interessados, os quais devem ser notificados independentemente da sua posição na árvore de componentes. Isto é denominado *dependência de eventos* (Definição 4.4).

**Definição 4.4 (Dependência de Eventos)** Uma relação de dependência entre um evento anunciado (ea) por um dado componente (cfa) e um evento de interesse (ei) de um dado componente (cfb), denotada por $dep_e(ea, ei)$, é definida se e somente se $ea = [id_{ei}]$, ou seja

$$dep_e(ea, ei) \Rightarrow (ea = [id_{ei}])$$

Um sistema pode ser simplesmente orientado a serviços ou orientado a eventos. A CMS não restringe o uso de uma abordagem específica, sendo assim, dependendo dos requisitos de desenvolvimento do sistema pode-se utilizar uma ou outra abordagem, ou ainda uma terceira híbrida.

[1] A partir deste ponto, ao se mencionar *componente* no contexto da CMS, entenda-se *componente funcional*.

Independente da abordagem escolhida, todas as dependências de serviços e/ou eventos devem ser contempladas para que o sistema possa ser executado, o que é aqui denominado de *Dependência Completa* (ver Definição 4.5).

**Definição 4.5 (Dependência Completa)** Dado um SBC, a relação de dependência entre os seus componentes funcionais é dita **completa** se e somente se

$$\forall cf_i \in F, \forall sr_j \in [SR_{cf_i}], \exists cf_k \in F, \exists sp_l \in [SP_{cf_k}] \mid dep_s(sp_l, sr_j) \wedge$$
$$\forall cf_m \in F, \forall ei_n \in [EI_{cf_m}], \exists cf_o \in F, \exists ea_p \in [EA_{cf_o}] \mid dep_e(ea_p, ei_n)$$

onde:

- $0 \leq i \leq |F|$, $0 \leq j \leq |[SR_{cf_i}]|$, $0 \leq k \leq |F|$, $0 \leq l \leq |[SP_{cf_k}]|$;
- $0 \leq m \leq |F|$, $0 \leq n \leq |[EI_{cf_m}]|$, $0 \leq o \leq |F|$, $0 \leq p \leq |[EA_{cf_o}]|$;
- $F$ é o conjunto de todos os componentes funcionais do sistema.

Uma vez detalhada a entidade funcional de um sistema baseado na CMS, pode-se agora detalhar a sua entidade estrutural: o contêiner (ver Definição 4.6). A estrutura do sistema é determinada pelos contêineres e interfere nos seus aspectos não funcionais, tais como: coesão, facilidade na composição de sistemas e desempenho.

Além disso, o contêiner implementa a mediação da interação entre as suas entidades filhas que podem ser tanto componentes funcionais quanto outros contêineres. Os modelos de interação baseada em serviços e eventos, que se utilizam dos contêineres para resolver as dependências entre os componentes funcionais, são descritos posteriormente.

**Definição 4.6 (Contêiner)** Um contêiner, ct, é um artefato de software que não implementa funcionalidades específicas do sistema e gerencia o acesso às suas entidades filhas, representadas pelos seus nós descendentes imediatos na árvore de componentes. Sendo assim,

$$ct = \langle id, CF, CT, SP, EI, FSP, FEI \rangle,$$

onde:

- $id$ é o identificador do contêiner;

- $CF = \langle cf_0, cf_1, cf_2, ..., cf_n \rangle$ é o conjunto de componentes que são filhos imediatos do contêiner, sendo $[SP_{cf_i}]$ e $[EI_{cf_i}]$ definidos, respectivamente, como os conjuntos de serviços providos e eventos de interesse de um componente $cf_i \in CF$;

- $CT = \langle ct_0, ct_1, ct_2, ..., ct_n \rangle$ é o conjunto de contêineres que são filhos imediatos do contêiner, sendo $[SP_{ct_i}]$ e $[EI_{ct_i}]$ definidos, respectivamente, como os conjuntos de serviços providos e eventos de interesse dos filhos imediatos de um contêiner $ct_i \in CT$;

- $CF \cup CT = EF$ é o conjunto de todas as entidades filhas imediatas do contêiner;

- $SP = (\bigcup_{i=0}^{|CF|} [SP_{cf_i}]) \cup (\bigcup_{i=0}^{|CT|} [SP_{ct_i}])$ é o conjunto de serviços providos pelas entidades filhas (componentes e outros contêineres) do contêiner;

- $EI = (\bigcup_{i=0}^{|CF|} [EI_{cf_i}]) \cup (\bigcup_{i=0}^{|CT|} [EI_{ct_i}])$ é o conjunto de eventos de interesse das entidades filhas (componentes e outros contêineres) do contêiner;

- $FSP : SP \rightarrow EF$ é uma função que mapeia um serviço provido na entidade filha que o provê, sendo a dupla $\langle sp_i, ef \rangle$ definida se e somente se $sp_i \in [SP_{ef}]$, denotando-se $prov(ct, [id_{sp_i}]) = ef$. Quando a dupla é definida, denota-se $prov_{def}(ct, [id_{sp_i}])$;

- $FEI : EI \rightarrow 2^{EF}$ é uma função que mapeia um evento de interesse nas entidades filhas interessadas no mesmo, sendo a dupla $\langle ei_i, efs \rangle$ definida se e somente se $\forall x \in efs, ei_i \in [EI_x]$, denotando-se $inter(ct, [id_{ei_i}]) = efs$;

- $ct$ é definido como o contêiner pai de toda entidade filha $x \in EF$, denotado por $pai(x) = ct$. Uma vez que o contêiner raiz do sistema não possui um contêiner pai, $pai(cr)$ não é definido.

Na Figura 4.3 são ilustrados seis exemplos de arquiteturas baseadas na CMS. Os conjuntos de todos os contêineres (R) e componentes funcionais (F) para cada arquitetura são ilustrados utilizando Diagramas de Venn [171]. Para todos os exemplos $cr = ct0$ e $pai(ct0)$ não é definido. Algumas propriedades específicas de cada exemplo são descritas a seguir.

- **Exemplo 1:** pai(a) = ct0, pai(b) = pai(c) = ct1, pai(d) = ct2, pai(ct1) = pai(ct2) = ct0.

- **Exemplo 2:** pai(a) = pai(b) = ct1, pai(c) = ct2, pai(ct1) = pai(ct2) = ct0.

- **Exemplo 3:** pai(ct1) = pai(ct2) = ct0. Neste exemplo, o sistema não possui componentes funcionais.

- **Exemplo 4:** pai(a) = ct5, pai(b) = ct6, pai(ct5) = ct3, pai(ct6) = ct4, pai(ct3) = ct1, pai(ct4) = ct2,

  pai(ct1) = pai(ct2) = ct0.

- **Exemplo 5:** pai(a) = pai(b) = pai(c) = pai(d) = pai(e) = pai(f) = ct0.

- **Exemplo 6:** Neste exemplo, assim como no Exemplo 3, o sistema não possui componentes funcionais.

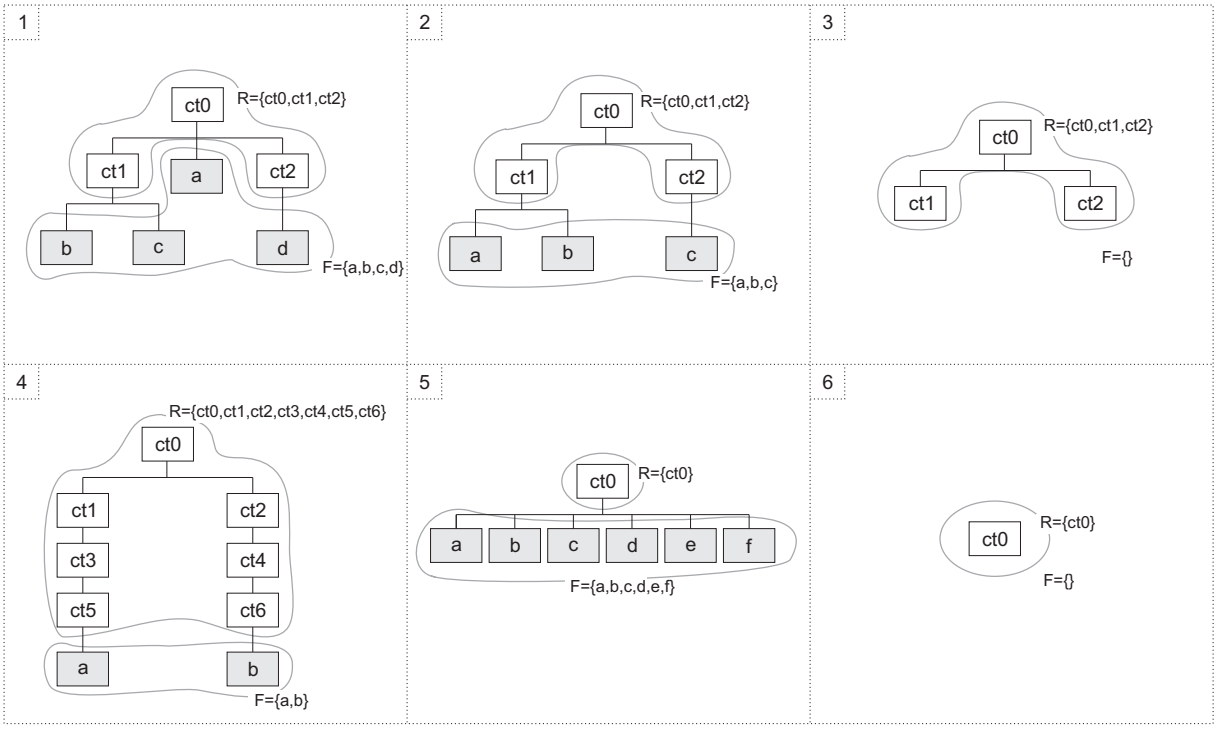



Figura 4.3: Exemplos de arquiteturas baseadas na CMS.

Como mencionado anteriormente, as funcionalidades do sistema são implementadas e disponibilizadas pelos seus componentes funcionais. Por isso, nos exemplos 3 e 6 ilustrados na Figura 4.3, apesar da arquitetura ser válida, o sistema não provê nenhuma funcionalidade pois não possui componentes funcionais ($F = \{\}$).

De acordo com a Definição 4.6, cada contêiner possui dois conjuntos de duplas, um relacionado aos serviços providos e outro relacionado aos eventos de interesse de cada uma de suas

entidades filhas, sejam componentes funcionais ou outros contêineres. O conjunto de duplas relacionado aos serviços, do tipo $\langle$*serviço, provedor*$\rangle$, mapeia cada serviço à sua respectiva entidade filha provedora. O conjunto de duplas relacionado aos eventos, do tipo $\langle$*evento, interessados*$\rangle$, mapeia cada evento às suas respectivas entidades filhas interessadas. Os conjuntos formam tabelas de serviços providos e eventos de interesse, as quais são descritas ao longo deste capítulo como ilustrado no exemplo da Figura 4.4. As tabelas são obtidas através das funções $FSP$ e $FEI$, descritas na Definição 4.6.

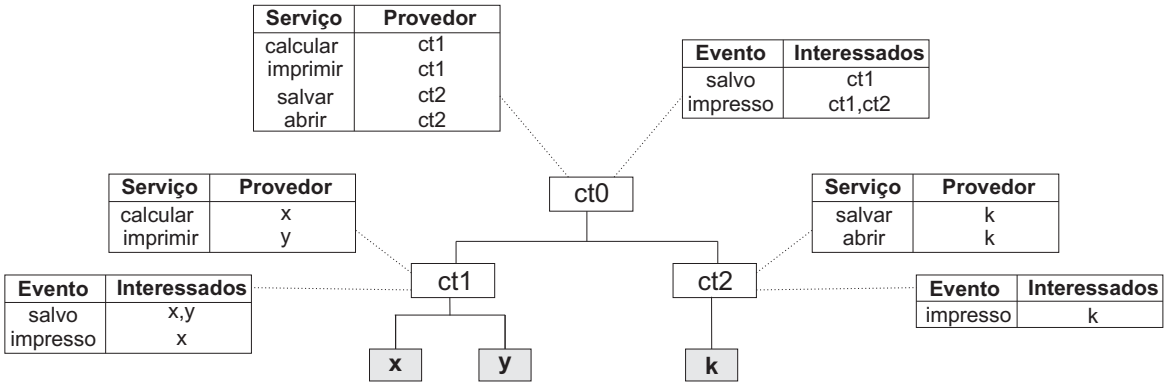



Figura 4.4: Tabelas de serviços providos e eventos de interesse das entidades filhas dos contêineres.

Considere que no exemplo da Figura 4.4 o componente *x* requer o serviço *salvar*. Com base nas definições 4.1, 4.2 e 4.6 pode-se descrever tal sistema da seguinte forma (aspas são utilizadas para denotar cadeias de caracteres):

- $cr = ct0, R = \{ct0, ct1, ct2\}, F = \{x, y, k\}$
- $x = \langle$"$x$", $\{sp_{calcular}\}, \{$"*salvar*"$\}, \{\}, \{ei_{salvo}, ei_{impresso}\}\rangle$,
  onde $sp_{calcular} = \langle$"*calcular*", $fns\rangle$, $ei_{salvo} = \langle$"*salvo*", $fne\rangle$ e $ei_{impresso} = \langle$"*impresso*", $fne\rangle$
- $y = \langle$"$y$", $\{sp_{imprimir}\}, \{\}, \{$"*impresso*"$\}, \{ei_{salvo}\}\rangle$,
  onde $sp_{imprimir} = \langle$"*imprimir*", $fns\rangle$
- $k = \langle$"$k$", $\{sp_{salvar}, sp_{abrir}\}, \{\}, \{$"*salvo*"$\}, \{ei_{impresso}\}\rangle$,
  onde $sp_{salvar} = \langle$"*salvar*", $fns\rangle$ e $sp_{abrir} = \langle$"*abrir*", $fns\rangle$
- $ct1 = \langle$"*ct1*", $\{x, y\}, \{\}, \{sp_{calcular}, sp_{imprimir}\}, \{ei_{salvo}, ei_{impresso}\}, \{\langle sp_{calcular}, x\rangle,$
  $\langle sp_{imprimir}, y\rangle\}, \{\langle ei_{salvo}, \{x, y\}\rangle, \langle ei_{impresso}, \{x\}\rangle\}\rangle$

- $ct2 = \langle$"$ct2$", $\{k\}, \{\}, \{sp_{salvar}, sp_{abrir}\}, \{ei_{impresso}\}, \{\langle sp_{salvar}, k\rangle, \langle sp_{abrir}, k\rangle\}$, $\{\langle ei_{impresso}, \{k\}\rangle\}\rangle$

- $ct0 = \langle$"$ct0$", $\{\}, \{ct1, ct2\}, \{sp_{calcular}, sp_{imprimir}, sp_{salvar}, sp_{abrir}\}, \{ei_{salvo}, ei_{impresso}\}$, $\{\langle sp_{calcular}, ct1\rangle, \langle sp_{imprimir}, ct1\rangle, \langle sp_{salvar}, ct2\rangle, \langle sp_{abrir}, ct2\rangle\}, \{\langle ei_{salvo}, \{ct1\}\rangle$, $\langle ei_{impresso}, \{ct1, ct2\}\rangle\}\rangle$

À primeira vista, pode-se pensar que uma arquitetura baseada em uma árvore com profundidade pequena poderia ser sempre a melhor escolha para um dado sistema. Isto seria aparentemente mais interessante em termos de desempenho e não iria requerer do desenvolvedor o estabelecimento de uma hierarquia de contêineres. Uma arquitetura formada por apenas um contêiner raiz e um conjunto de componentes funcionais filhos deste contêiner, por exemplo, teria uma estrutura similar a arquiteturas orientadas a serviços, como Jini [51], com um mediador de interação e um conjunto de implementadores.

Entretanto, a principal motivação para uma hierarquia de contêineres é que esta permite a manutenção da coesão das funcionalidades providas por um determinado conjunto de componentes funcionais. Uma vez mantida tal coesão funcional, tem-se uma redução do custo inerente à interação baseada nos contêineres. Esta mediação multi-nível é ilustrada na Figura 4.5. Quanto maior a coesão funcional entre os componentes funcionais, menos mediação multi-nível é necessária e, conseqüentemente, melhor será o desempenho.

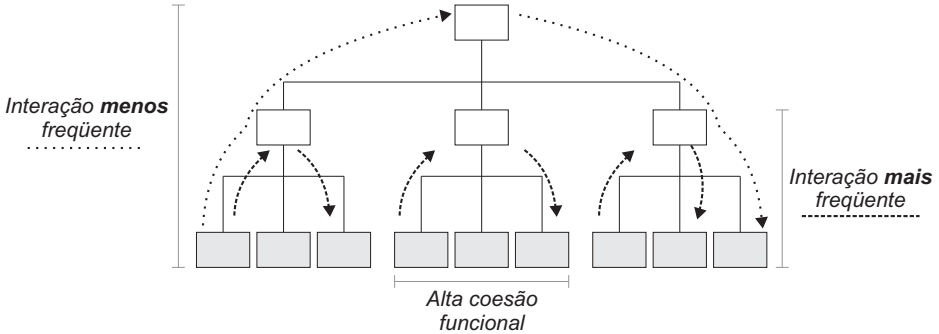



Figura 4.5: Mediação multi-nível: maior coesão funcional para obter melhor desempenho.

Além disso, torna-se possível reutilizar contêineres inteiros, em qualquer nível da hierarquia, sem a necessidade de entender seus componentes/contêineres filhos. Desta forma, permite-se uma composição recursiva de sistemas, uma vez que os contêineres raiz de um conjunto de sistemas

podem ser vistos como componentes de um novo sistema sendo construído. Portanto, pode-se compor sistemas através da integração de contêineres de sistemas existentes (Figure 4.6).

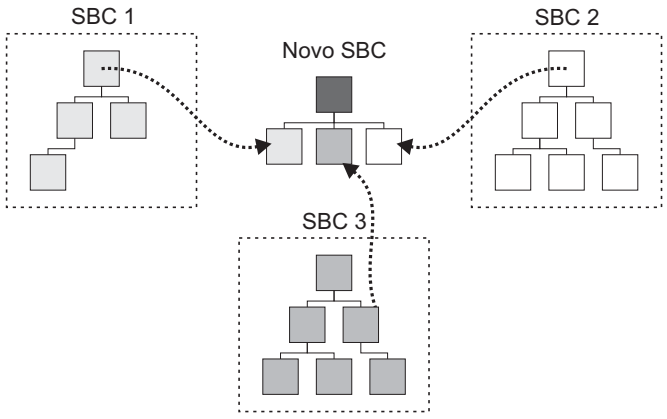



Figura 4.6: Composição recursiva: sistemas como componentes.

### 4.1.2 Adição e Remoção de Componentes Funcionais e Contêineres

Pela Definição 4.1, deve existir pelo menos um contêiner (contêiner raiz) para que se tenha uma arquitetura válida de sistema de acordo com a CMS. Sendo assim, a adição e a remoção de componentes funcionais e contêineres da árvore de componentes deve ocorrer sempre dentro de um determinado contêiner, seja este raiz ou não, o qual representa o contêiner pai da entidade sendo adicionada ou removida.

Não há restrição quanto ao nível em que um componente ou contêiner é adicionado na hierarquia. Da mesma forma, não há restrição quanto aos componentes sendo removidos. A única responsabilidade do desenvolvedor é manter a relação de dependência completa entre os componentes funcionais (ver Definição 4.5). Sendo assim, a cada processo de adição ou remoção, deve-se verificar se tal relação está sendo satisfeita pois, caso contrário, o sistema pode não funcionar corretamente.

Uma vez que cada contêiner possui uma tabela de serviços providos e eventos de interesse de suas entidades filhas, após a adição ou a remoção de um componente, estas tabelas devem ser atualizadas, desde o contêiner pai do componente até a raiz da hierarquia. O processo de adição de um componente (ver Definição 4.7), cujos passos são descritos a seguir, é ilustrado na Figura 4.7.

1. O componente *x*, que provê o serviço *calcular* e tem interesse no evento *salvo*, é adicionado

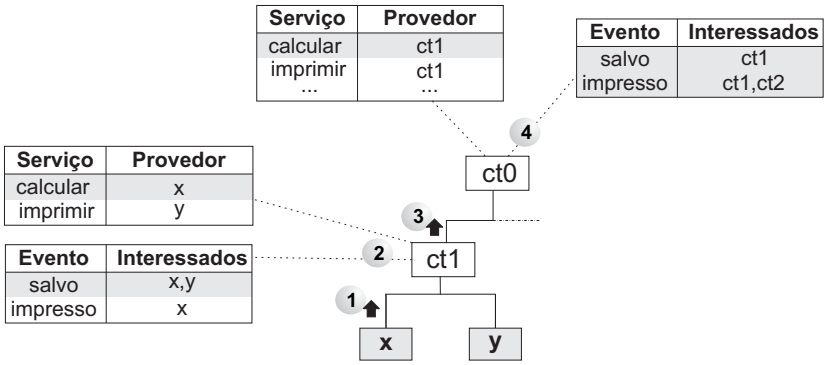



Figura 4.7: Adição de componentes: atualização das tabelas de serviços providos e eventos de interesse até a raiz da hierarquia.

ao contêiner *ct1*.

2. O contêiner *ct1* atualiza as suas tabelas de serviços providos e eventos de interesse de seus componentes filhos. As linhas atualizadas das tabelas estão destacadas em cinza na Figura 4.7.

3. O contêiner *ct1* solicita que seu contêiner pai (*ct0*) atualize as suas tabelas.

4. O contêiner *ct0* atualiza as suas tabelas de serviços providos e eventos de interesse de suas entidades filhas.

**Definição 4.7 (Adição de Componente)** Dado um SBC, define-se uma operação de adição de um componente funcional, $cf$, no sistema como filho do contêiner pai, $p$, tal que $p \in R$ como sendo

$$adicionarComponente(SBC, cf, p) \rightarrow [CF_p] \cup \{cf\},\ [SP_p] \cup [SP_{cf}],\ [EI_p] \cup [EI_{cf}],\ atualizarPai(p, SP_{cf}, EI_{cf})$$

onde:

$$atualizarPai(ct, SP, EI) \rightarrow \begin{cases} [SP_{ct}] \cup SP,\ [EI_{ct}] \cup EI & pai(ct)\ indefinido \\ [SP_{ct}] \cup SP,\ [EI_{ct}] \cup EI,\ atualizarPai(pai(ct), SP, EI) & pai(ct)\ definido \end{cases}$$

A operação de adição de um contêiner é semelhante à adição de um componente funcional. A principal diferença é que o contêiner já pode conter entidades filhas as quais devem também

fazer parte da hierarquia do sistema (ver Definição 4.8). Este tipo de operação é comum durante a composição de sistemas que, como já destacado anteriormente, é uma das principais motivações para a hierarquia de contêineres na CMS.

**Definição 4.8 (Adição de Contêiner)** Dado um SBC, define-se uma operação de adição de um contêiner, *ct*, no sistema como filho do contêiner pai, *p*, tal que $p \in R$ como sendo

$$adicionarConteiner(SBC, ct, p) \rightarrow [CT_p] \cup \{ct\},\ [SP_p] \cup [SP_{ct}],\ [EI_p] \cup [EI_{ct}]\ , atualizarPai(p, SP_{ct}, EI_{ct})$$

onde:

$$atualizarPai(ct, SP, EI) \rightarrow \begin{cases} [SP_{ct}] \cup SP,\ [EI_{ct}] \cup EI & pai(ct)\ indefinido \\ [SP_{ct}] \cup SP,\ [EI_{ct}] \cup EI,\ atualizarPai(pai(ct), SP, EI) & pai(ct)\ definido \end{cases}$$

O processo de remoção de um componente funcional é semelhante ao processo de adição. A principal diferença é que, durante a atualização dos serviços providos e eventos de interesse em cada contêiner, as tabelas têm suas linhas removidas, em vez de adicionadas. O processo de remoção de um componente funcional (ver Definição 4.9), cujos passos são descritos a seguir, é ilustrado na Figura 4.8.

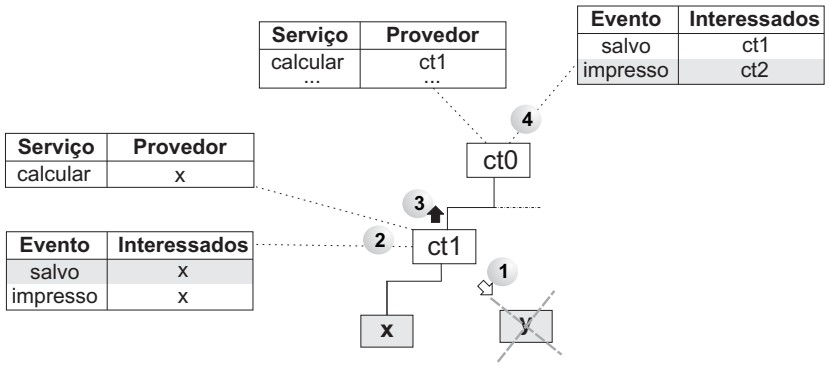



Figura 4.8: Remoção de componentes: atualização das tabelas de serviços providos e eventos de interesse até a raiz da hierarquia.

1. O componente *y*, que provê o serviço *imprimir* e tem interesse no evento *salvo*, é removido do contêiner *ct1*.

2. O contêiner *ct1* atualiza as suas tabelas de serviços providos e eventos de interesse de seus componentes filhos. Observe que as linhas referentes ao componente *y* ilustradas na Figura 4.7 não aparecem na Figura 4.8.

3. O contêiner *ct1* solicita que seu contêiner pai (*ct0*) atualize as suas tabelas.

4. O contêiner *ct0* atualiza as suas tabelas de serviços providos e eventos de interesse de suas entidades filhas.

Como ilustrado na Figura 4.8, os eventos que são de interesse de mais de um componente não podem ter a linha da tabela removida. Apenas a referência ao componente sendo removido deve ser retirada da tabela. Isto ocorre, por exemplo, com o evento *salvo* que, além de ser de interesse do componente *y* sendo removido, também é de interesse do componente *x*. Observe que a Definição 4.9 contempla este detalhe, removendo do conjunto de eventos de interesse apenas os elementos que não fazem interseção com os de outras entidades.

**Definição 4.9 (Remoção de Componente Funcional)** Dado um SBC, define-se uma operação de remoção de um componente funcional, $cf$, do sistema, filho do contêiner pai, $p$, tal que $p \in R$ e $EF_p$ é o conjunto de entidades filhas de $p$, como sendo

$$removerComponente(SBC, cf, p) \rightarrow [CF_p] - \{cf\},\ [SP_p] - [SP_{cf}],\ [EI_p] - ([EI_{cf}] - \bigcap_{i=0}^{|[EF_p]|} [EI_{ef_i}] \mid ef_i \in EF_p),$$
$$atualizarPai(p, SP_{cf}, EI_{cf})$$

onde:

$$atualizarPai(ct, SP, EI) \rightarrow \begin{cases} [SP_{ct}] - SP,\ [EI_p] - (EI - \bigcap_{i=0}^{|[EF_p]|} [EI_{ef_i}] \mid ef_i \in EF_p) & pai(ct)\ indefinido \\ [SP_{ct}] - SP,\ [EI_p] - (EI - \bigcap_{i=0}^{|[EF_p]|} [EI_{ef_i}] \mid ef_i \in EF_p), & \\ atualizarPai(pai(ct), SP, EI) & pai(ct)\ definido \end{cases}$$

Como mencionado anteriormente, é importante observar que uma operação de adição e, principalmente, de remoção, pode tornar incompleta a relação de dependência do sistema (ver Definição 4.5). No exemplo da Figura 4.9, após a remoção do componente *y*, o evento *impresso*

deixou de ser anunciado. Conseqüentemente, os componentes $x$ e $k$ interessados no evento terão suas funcionalidades comprometidas. Sendo assim, o desenvolvedor deve realizar as operações de adição e remoção em uma ordem em que a dependência entre os componentes funcionais se mantenha sempre completa. A definição da operação de remoção de contêiner é similar à definição da operação de remoção de componentes (ver Definição 4.10).

**Definição 4.10 (Remoção de Contêiner)** Dado um SBC, define-se uma operação de remoção de um contêiner, $ct$, do sistema, filho do contêiner pai, $p$, tal que $p \in R$ e $EF_p$ é o conjunto de entidades filhas de $p$, como sendo

$$removerConteiner(SBC, ct, p) \rightarrow [CT_p] - \{ct\},\ [SP_p] - [SP_{ct}],\ [EI_p] - ([EI_{ct}] - \bigcap_{i=0}^{||[EF_p]||} [EI_{ef_i}] \mid ef_i \in EF_p),$$
$$atualizarPai(p, SP_{ct}, EI_{ct})$$

onde:

$$atualizarPai(ct, SP, EI) \rightarrow \begin{cases} [SP_{ct}] - SP,\ [EI_p] - (EI - \bigcap_{i=0}^{||[EF_p]||} [EI_{ef_i}] \mid ef_i \in EF_p) & pai(ct)\ indefinido \\ [SP_{ct}] - SP,\ [EI_p] - (EI - \bigcap_{i=0}^{||[EF_p]||} [EI_{ef_i}] \mid ef_i \in EF_p), & \\ atualizarPai(pai(ct), SP, EI) & pai(ct)\ definido \end{cases}$$

### 4.1.3 Modelos de Interação Baseada em Serviços e Eventos

No modelo de componentes especificado na CMS são definidos dois tipos de interação entre componentes: baseada em serviços e baseada em eventos. Na interação baseada em eventos o foco está sobre o anúncio da mudança de estado de um determinado componente, localizado em um determinado contêiner, aos componentes interessados, mesmo que estes estejam localizados em contêineres diferentes. A interação baseada em serviços permite a invocação dos serviços de um determinado componente a partir de qualquer outro componente do sistema, ainda que pertençam a contêineres diferentes. Em ambos os casos não há referência explícita entre os componentes funcionais, o que permite a alteração dos provedores sem acarretar mudanças nos demais.

**Interação Baseada em Serviços**

Após a inserção de um componente em um determinado contêiner, seus serviços tornam-se disponíveis a qualquer outro componente do sistema (ver Definição 4.12). Sendo assim, considerando o serviço *salvar* implementado pelo componente *k*, pode-se solicitar a execução deste serviço a partir de um componente *x*, sem fazer referência direta a *k*. Este processo é ilustrado na Figura 4.9 e seus passos são descritos a seguir.

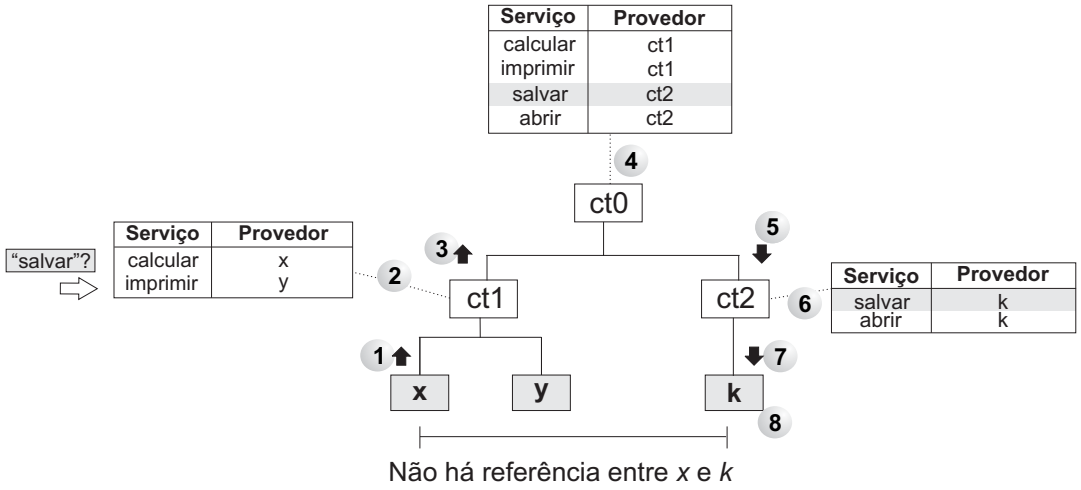



Figura 4.9: Interação baseada em serviços: localização e execução sem referência entre componentes funcionais.

1. O componente *x* solicita a execução do serviço *salvar* ao seu contêiner pai (ct1).

2. O contêiner *ct1* verifica, de acordo com sua tabela de serviços providos, que nenhum dos seus componentes filhos provê o serviço *salvar*.

3. O contêiner *ct1* então encaminha a solicitação ao seu contêiner pai (*ct0*).

4. O contêiner *ct0* verifica, de acordo com sua tabela de serviços providos, que um de seus componentes filhos provê o serviço *salvar* (*ct2*). Para o contêiner *ct0*, o contêiner *ct2* é visto como um componente que provê o serviço.

5. O contêiner *ct0* então encaminha a solicitação de serviço para o contêiner *ct2*.

6. O contêiner *ct2* não provê o serviço mas possui em sua tabela uma referência ao real provedor do serviço – o componente *k*.

7. O contêiner *ct2* então encaminha a solicitação de serviço para o componente funcional *k*.

8. O componente *k* executa o serviço *salvar* e retorna, caso exista, o resultado da execução.

Como ilustrado na Figura 4.9, não há referência alguma entre o componente solicitante do serviço (*x*) e o componente provedor do mesmo (*k*). Desta forma, é possível alterar o componente que provê o serviço *salvar* sem modificar o restante da estrutura.

**Definição 4.11 (Invocação de Serviço)** Considere um serviço com identificador *ids* provido por um dado componente, com parâmetros de execução *P*, sendo invocado a partir de um componente de origem *src*. A invocação desse serviço denotada por $invServ(ids, P, src)$ é definida como sendo

$$invServ(ids,P,src) = \begin{cases} invServ(ids,P,pai(src)) & src \in F \\ execServ(ids,P,prov(src,ids)) & ((src \in R) \wedge (prov_{def}(src,ids))) \\ invServ(ids,P,pai(src)) & ((src \in R) \wedge (\neg prov_{def}(src,ids)) \wedge (src \neq cr)) \\ ERRO & ((src \in R) \wedge (\neg prov_{def}(src,ids)) \wedge (src = cr)) \end{cases}$$

onde:

$$execServ(ids,P,src) = \begin{cases} execServ(ids,P,prov(src,ids)) & src \in R \\ fns(src,ids,P) & src \in F \end{cases}$$

Na Definição 4.11 são utilizadas execuções recursivas da operação *invServ*, de "baixo para cima" na hierarquia de componentes até que o contêiner que registra o serviço seja encontrado (ver Figura 4.10). Quando isto acontece, a operação *execServ* é executada "de cima para baixo" na hierarquia até que o componente funcional que provê o serviço seja encontrado e execute a operação que implementa o serviço (*fns*). Caso o contêiner raiz seja alcançado e o provedor do serviço não seja encontrado, tem-se um *ERRO*. Isto significa que o serviço não é provido por nenhum componente funcional da hierarquia.

Com base na Definição 4.11 e no exemplo ilustrado na Figura 4.9, tem-se a seguinte seqüência de operações relacionadas a uma invocação de serviço (ver Figura 4.10):

1. A operação se inicia com a invocação do serviço *salvar* a partir do componente *x*, ou seja, *invServ("salvar", P, x)*, onde *P* é o conjunto de parâmetros necessários à execução do serviço;

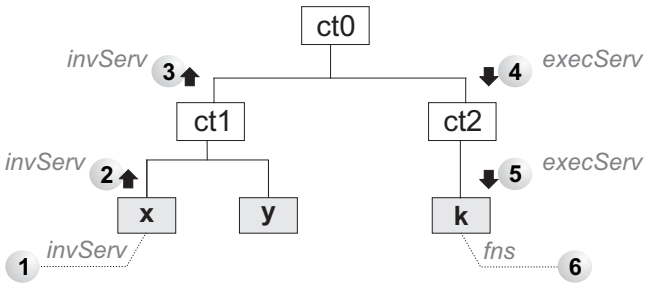



Figura 4.10: Invocação de serviço de acordo com a Definição 4.11 e o exemplo da Figura 4.9.

2. Uma vez que *x* é um componente funcional ($src \in F$), tem-se a execução recursiva da operação *invServ("salvar", P, ct1)* para encaminhar a requisição para o pai do componente *x*, ou seja, o contêiner *ct1* (*pai(src)*);

3. Considerando que *ct1* é um contêiner ($src \in R$), que não há uma entidade filha de *ct1* provedora do serviço ($\neg prov_{def}(src, ids)$) e que não se trata do contêiner raiz ($src \neq cr$), tem-se mais uma vez a chamada recursiva de *invServ("salvar", P, ct0)* para encaminhar a requisição para o pai do contêiner *ct1*, ou seja, o contêiner *ct0* (pai(src));

4. Uma vez que *ct0* é um contêiner ($src \in R$) e que há uma entidade filha de *ct0* provedora do serviço ($prov_{def}(src, ids)$), neste caso o contêiner *ct2*, tem-se a chamada de *execServ("salvar", P, ct2)*, para encaminhar a requisição para o provedor do serviço – o contêiner *ct2* (prov(src,ids));

5. De acordo com a definição de *execServ*, sendo *ct2* um contêiner ($src \in R$) tem-se novamente a chamada de *execServ("salvar", P, k)*, para encaminhar a requisição para o provedor do serviço – o componente *k* (prov(src,ids));

6. Por fim, sendo *k* um componente funcional ($src \in F$), tem-se a chamada da operação que implementa o serviço *salvar*, com a execução de *fns(k, "salvar", P))*.

**Interação Baseada em Eventos**

Quando um evento é anunciado por um determinado componente funcional, toda a hierarquia de componentes do sistema deve ser verificada para que todos os interessados no evento sejam notificados (ver Definição 4.12). A interação baseada em eventos também é realizada pelos contêineres,

não havendo referência direta entre os componentes funcionais. Na Figura 4.11 este processo é ilustrado, onde é possível verificar que não há referência alguma entre o componente funcional anunciante do evento (*y*) e os interessados no evento (*x* e *k*). Desta forma, o componente funcional que anuncia o evento poderia ser alterado sem modificar o restante da estrutura.

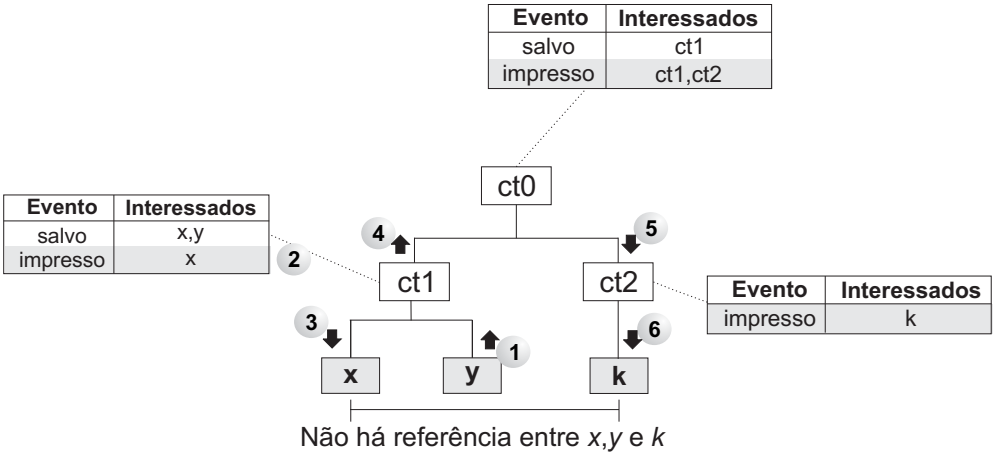



Figura 4.11: Interação baseada em eventos: notificação de eventos sem referência direta entre componentes funcionais.

1. O componente *y* anuncia um evento denominado *impresso*.

2. O anúncio é recebido, diretamente, apenas pelo seu contêiner pai (*ct1*) que verifica em sua tabela de eventos de interesse de seus componentes filhos se algum deles está interessado no evento.

3. O contêiner *ct1* encaminha o evento ao único interessado no evento de acordo com a sua tabela – o componente *x*.

4. Além disso, o contêiner *ct1* também encaminha o evento ao seu contêiner pai (*ct0*), pois pode haver mais algum interessado no evento no restante da hierarquia.

5. O contêiner *ct0*, de acordo com a sua tabela de eventos, repassa o evento aos interessados, exceto para o originador do anúncio do evento (*ct1*). Sendo assim, encaminha o evento ao contêiner *ct2*. Por representar a raiz da hierarquia, ele não possui um contêiner pai para o qual também repassaria o evento.

6. O contêiner *ct2* repassa o evento ao componente interessado de acordo com a sua tabela de eventos. No exemplo, o componente *k* é o único interessado.

**Definição 4.12 (Anúncio de Evento)** Considere um evento de interesse de um conjunto de componentes, com identificador **ide** e parâmetros de notificação **P**, sendo anunciado a partir de um componente de origem **src**, com origem anterior denotada por **ult**. Uma operação de anúncio desse evento, a qual é denotada por *anuEv(ide, P, src, ult)*, é definida como sendo

$$anuEv(ide,P,src,ult) = \begin{cases} anuEv(ide,P,pai(src),src) \\ \quad \textbf{\textit{se}}\ (src \in F) \vee ((src \in R) \wedge (\neg inter_{def}(src,ide)) \wedge (src \neq cr)) \\ recEv(ide,P,e_i)\ \forall e_i \in inter(src,ide) \mid e_i \neq ult, anuEv(ide,P,pai(src),src) \\ \quad \textbf{\textit{se}}\ ((src \in R) \wedge (inter_{def}(src,ide)) \wedge (src \neq cr)) \\ recEv(ide,P,e_i)\ \forall e_i \in inter(src,ide) \\ \quad \textbf{\textit{se}}\ ((src \in R) \wedge (inter_{def}((src,ide))) \wedge (src = cr)) \end{cases}$$

onde:

$$recEv(ide,P,src) = \begin{cases} recEv(ide,P,e_i)\ \forall e_i \in inter(src,ide) & src \in R \\ fne(src,ide,P) & src \in F \end{cases}$$

Da mesma forma que ocorre na definição de invocação de serviços, na Definição 4.12 são utilizadas execuções recursivas da operação *anuEv*, de "baixo para cima" na hierarquia de componentes, até que o componente raiz seja encontrado (ver Figura 4.12). A cada contêiner, verifica-se a tabela de interessados no evento e a operação *recEv* é executada "de cima para baixo" na hierarquia, até que todos os componentes funcionais interessados no evento sejam notificados e executem suas respectivas operações de recepção de evento (*fne*).

Com base na Definição 4.12 e no exemplo ilustrado na Figura 4.11, tem-se a seguinte seqüência de operações relacionadas a um anúncio de evento (ver Figura 4.12):

1. A operação se inicia com o anúncio do evento *impresso* a partir do componente *y*, ou seja, *anuEv("impresso", P, y, y)*, onde *P* é um conjunto de parâmetros relacionados ao evento;

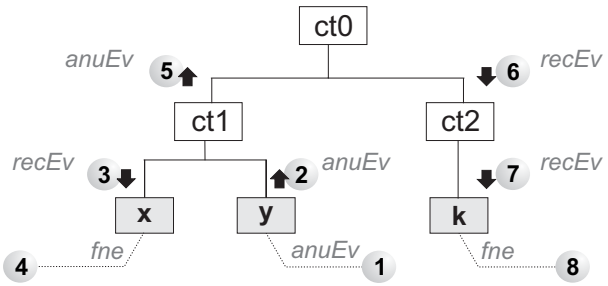



Figura 4.12: Exemplo de operação de anúncio de evento de acordo com a Definição 4.12 e o exemplo da Figura 4.11.

2. Uma vez que *y* é um componente funcional ($src \in F$), tem-se a execução recursiva da operação *anuEv("impresso", P, ct1, y)* para encaminhar a requisição para o pai do componente *y*, ou seja, o contêiner *ct1* (*pai(src)*);

3. Considerando que *ct1* é um contêiner ($src \in R$) e que há uma entidade filha de *ct1* interessada no evento ($inter_{def}(src, ids)$), tem-se a chamada de *recEv* para todos os interessados com exceção do originador do evento ($\forall e_i \in inter(src, ide) | e_i \neq ult$), neste caso, apenas para o componente *x*. Sendo assim, tem-se a chamada *recEv("impresso", P, x)* para encaminhar o anúncio para o componente *x*. Além disso, como descrito no passo 5, tem-se a chamada de *anuEv("impresso", P, ct0, ct1)* para também encaminhar o anúncio ao contêiner pai de *ct1*, ou seja, o contêiner *ct0* (ct0 = pai(ct1));

4. Na chamada de *recEv*, sendo *x* um componente funcional ($src \in F$), tem-se a chamada da operação que recebe o anúncio do evento *impresso*, com a execução de *fne(k, "impresso", P)*.

5. Na chamada de *anuEv*, realizada no passo 3, uma vez que *ct0* é um contêiner ($src \in R$) e há entidades filhas de *ct0* interessadas no evento ($inter_{def}(src, ids)$), neste caso os contêineres *ct1* e *ct2*, tem-se a chamada de *recEv* para todos os interessados com exceção do originador do evento (*ct1*). Sendo assim, tem-se *recEv("impresso", P, ct2)* para encaminhar a requisição para o interessado no evento – o contêiner *ct2*;

6. De acordo com a definição de *recEv*, sendo *ct2* um contêiner ($src \in R$) tem-se novamente a chamada de *recEv("impresso", P, k)* para encaminhar a requisição para o interessado no evento – o componente *k*;

7. Por fim, sendo $k$ um componente funcional ($src \in F$), tem-se a chamada da operação que implementa o recebimento do anúncio de evento *impresso*, com a execução de *fne(k, "impresso", P))*.

## 4.2 Aspectos Pragmáticos: Reutilização, Adaptação, Personalização, Versionamento e Execução

Nesta seção são abordados alguns aspectos que não foram considerados na definição do núcleo da CMS mas são indispensáveis à utilização prática da especificação no desenvolvimento de sistemas baseados em componentes. Alguns destes aspectos são conseqüência direta da definição apresentada anteriormente, outros são acessórios para aumentar o nível de reutilização e adaptação dos componentes e facilitar a construção e a execução de sistemas.

Os conceitos apresentados a seguir não interferem diretamente na definição do núcleo da CMS. Evidentemente que, no nível de projeto e implementação de um arcabouço para o desenvolvimento de sistemas de acordo com a CMS, estes conceitos devem ser contemplados, como descrito no Capítulo 5.

### 4.2.1 Reutilização de Componentes e Montagem de Sistemas: Definindo Apelidos

Uma das principais motivações para a utilização de uma abordagem baseada em componentes para a construção de sistemas de software é a reutilização. Através da reutilização de componentes pré-concebidos pertencentes a um dado domínio, reduz-se o tempo de desenvolvimento e, na medida em que o componente é reutilizado e melhorado, aumenta-se também a qualidade dos sistemas desenvolvidos.

Uma das principais características definidas na CMS que possui impacto direto sobre a reutilização de componentes e montagem de sistemas é o fato de que as dependências de serviços e eventos são resolvidas apenas com base no identificador dos serviços e eventos dos componentes funcionais. Sendo assim, dois serviços (ou eventos) com mesmo identificador, independentemente dos parâmetros e da funcionalidade relacionada, são considerados o mesmo na CMS.

Levando-se em conta uma reutilização caixa-preta, onde o código do componente não se encontra disponível para alteração, não é possível alterar o identificador definido para um dado serviço ou evento do componente. Então, como reutilizar dois componentes que possuam serviços de mesmo identificador, mas com funcionalidades diferentes? Como utilizar os dois serviços ao mesmo tempo no sistema?

Para resolver estes problemas, pode-se utilizar a noção de *apelidos* de serviços e eventos. Um apelido é um identificador de um serviço ou evento no contexto de um sistema específico. Ao reutilizar um componente, deve-se definir o apelido de cada serviço e evento daquele componente para que todas as dependências sejam resolvidas. Desta forma, mesmo reutilizando componentes com serviços de mesmo identificador, pode-se definir apelidos que possibilitem a utilização dos dois serviços ao mesmo tempo. Na Figura 4.13 ilustra-se este processo.

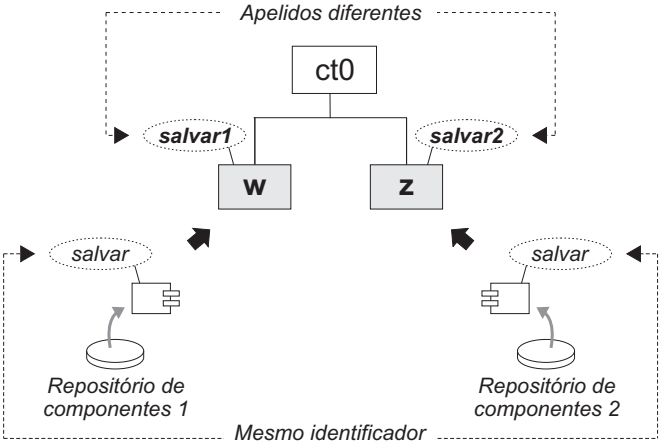



Figura 4.13: Apelidos definidos no momento da reutilização dos componentes funcionais e montagem do sistema.

Apelidos não interferem diretamente na definição do núcleo da CMS. Após a configuração dos mesmos, pode-se considerá-los os "novos identificadores" dos serviços e eventos e, sendo assim, aplicar a mesma definição do núcleo apresentada anteriormente.

Vale ressaltar que é possível definir um apelido tanto do lado do provedor do serviço quanto do lado do seu requisitante. O mesmo ocorre para o anunciante e o interessado em um evento. Por exemplo, considere que o componente *w* requer o serviço *salvar* e o componente *z* implementa o serviço *salvar2*. Tanto é possível redefinir o apelido do serviço provido *salvar2* para *salvar*, quanto redefinir o apelido do serviço requerido *salvar* para *salvar2*. É uma decisão do desen-

volvedor estabelecer de que lado o apelido deve ser definido para que se tenha uma dependência completa.

### 4.2.2 Sobrescrevendo Serviços: o Mecanismo de "Herança Caixa-preta"

Devido ao fato dos modelos de interação da CMS serem mediados por contêineres, é possível sobrescrever serviços mesmo com reutilização de componentes caixa-preta. Este mecanismo foi denominado *herança caixa-preta*. Em outras palavras, é possível reutilizar serviços de componentes sem estender a implementação do componente. Isto ocorre porque um componente pode requerer um serviço que ele mesmo provê. Uma vez que os serviços providos por componentes funcionais são acessados apenas via contêiner, é necessário apenas publicar funcionalidades internas do componente como serviços externos e acessá-los via contêiner.

Na Figura 4.14, um exemplo deste mecanismo é ilustrado. Considere o service *lerArquivo*, que é implementado pela seguinte seqüência de funcionalidades: *buffering*, *operaçãoES*. Se as funcionalidades internas são publicadas como serviços, o componente *w* pode acessá-las via contêiner. Sendo assim, o serviço *lerArquivo* é desacoplado das funcionalidades internas *buffering* e *operaçãoES*.

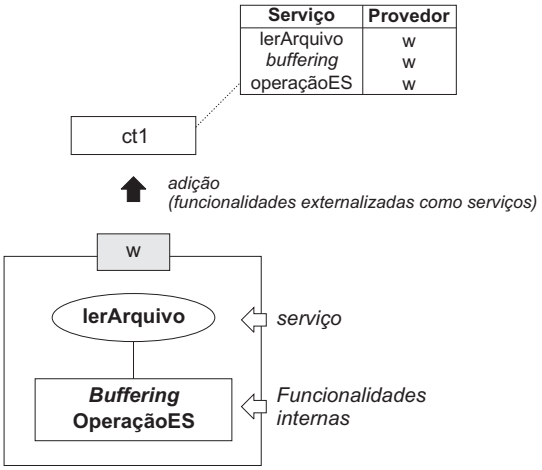



Figura 4.14: Funcionalidades internas disponibilizadas como serviços dos componentes.

Para sobrescrever a funcionalidade interna *buffering*, por exemplo, é necessário apenas adicionar um componente que provê um serviço com o mesmo identificador/apelido. Na Figura 4.15, o processo para sobrescrever o serviço é ilustrado. Nesta figura, o componente *z* sobrescreve o serviço *buffering* anteriormente registrado no contêiner *ct0* como sendo provido pelo componente

*w*. Uma vez que não há referências explícitas entre os componentes *w* e *z*, este processo pode ser realizado sem acarretar mudanças nos demais componentes.

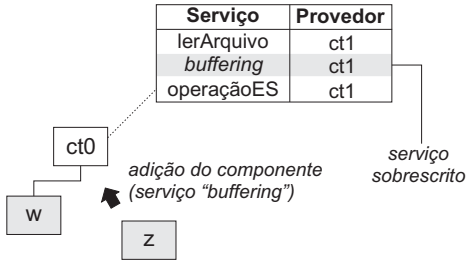



Figura 4.15: Sobrescrevendo serviços: herança caixa-preta.

Depois da adição do componente *z*, o serviço *lerArquivo* do componente *w* passa a ter como base o serviço *buffering* implementado pelo componente *z*. De um ponto de vista geral do sistema, as funcionalidades internas do componente *w* foram herdadas e sobrescreveu-se apenas a funcionalidade de *buffering*. É importante notar que, no nível de implementação, *z* não precisa estender *w*, é necessário apenas conhecer os serviços providos e requeridos pelo componente, ou seja, sua interface.

### 4.2.3 Modelo de Adaptadores: Adaptando Componentes a um Sistema

Quando um componente precisa ser utilizado em um contexto diferente daquele para o qual foi desenvolvido, ou seja, precisa ser reutilizado, sua interface pode precisar ser adaptada ao sistema. A utilização de mecanismos de reutilização de serviços por herança caixa-preta, como descrito anteriormente, demanda a construção ou reutilização de um outro componente. Para casos em que apenas parâmetros, retorno, exceções, restrições, dentre outras características da interface do componente precisam ser adaptados, pode-se definir um adaptador para o componente.

O conceito de adaptadores é bem estabelecido em paradigmas como orientação a objetos e componentes, com definições de padrões de projeto e implementação. A utilização de adaptadores no contexto do modelo de componentes especificado na CMS é similar à solução descrita no padrão de projeto *Adapter* [16]. Um *adaptador* é uma entidade que se coloca entre um provedor (componente) e um cliente (contêiner), adaptando a interface do provedor de acordo com a necessidade do cliente.

Da mesma forma que componentes funcionais, adaptadores são acessados apenas via contêi-

ner. Sendo assim, a interface de um componente pode ser adaptada em tempo de execução. O processo de inserção de um adaptador sobre um componente que já foi disponibilizado é apresentado na Figura 4.16. O componente *x* requisita o serviço *salvar* que é implementado pelo componente *k* originalmente, mas está sendo adaptado por *adapk*.

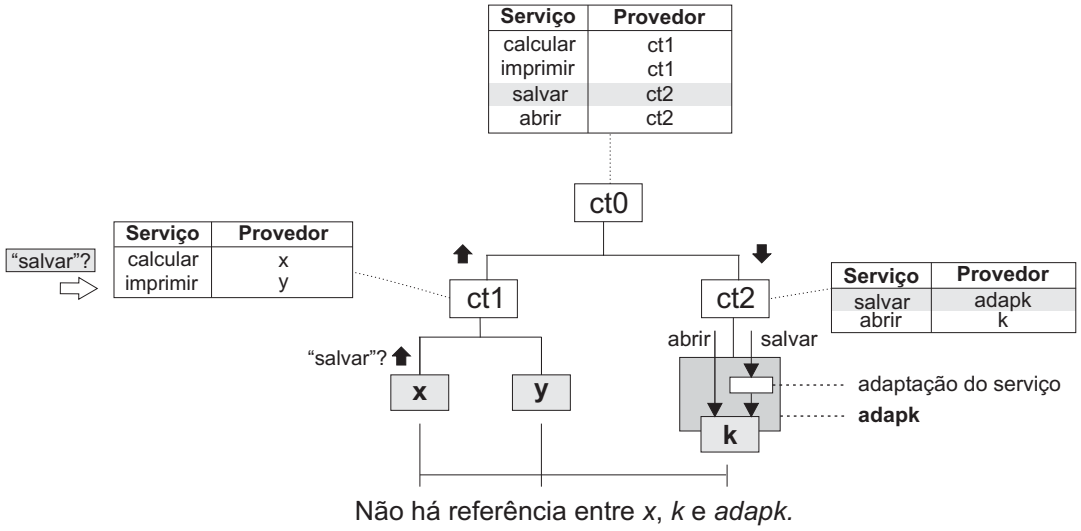



Figura 4.16: Adaptando serviços de componentes.

Um adaptador pode adaptar apenas alguns dos serviços ou eventos do componente. Sendo assim, é possível ter componentes utilizando o serviço original e outros utilizando o serviço adaptado. É o caso do serviço *abrir*, apresentado na Figura 4.16. Caso receba solicitação de execução deste serviço, *adapk* delegará para o componente *k* a execução do serviço utilizando a interface original.

Do ponto de vista do sistema baseado em componentes, um adaptador pode ser visto como um contêiner que possui apenas um componente funcional como entidade filha. Sendo assim, a definição do núcleo da CMS pode ser mantida simples e a entidade *adaptador* ser considerada apenas no nível de projeto e implementação, como descrito no Capítulo 5.

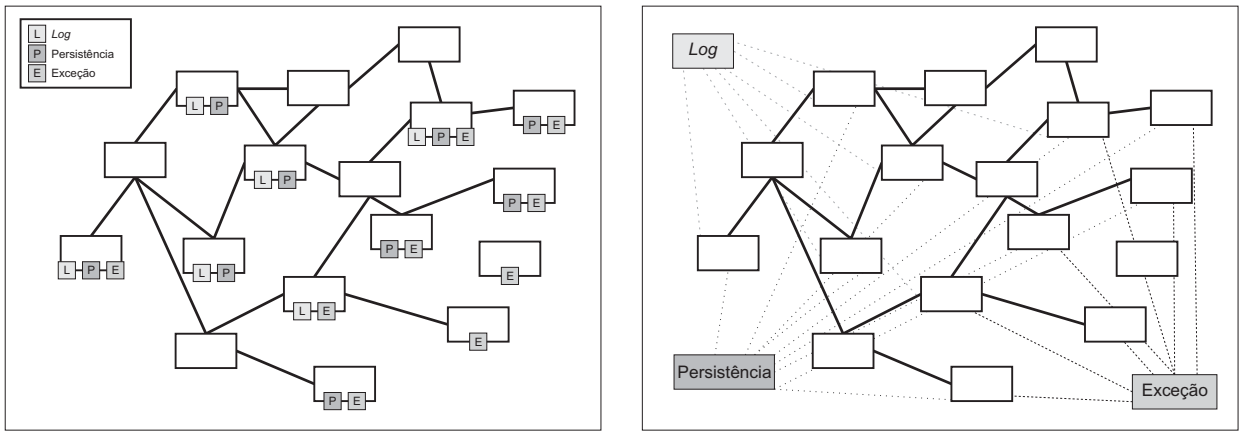
### 4.2.4 Modelo de Interesses: Personalizando a Execução de Componentes Caixa-Preta

O conceito de Separação de Interesses (*Separation of Concerns – SoC*) tem sido apontado por muitos pesquisadores como adequado para a especificação e implementação de requisitos trans-

versais, melhorando a modularização em sistemas de software [26]. Requisitos transversais, em geral não funcionais, são aqueles presentes em várias partes do projeto e da implementação do software, como tratamento de exceção, persistência, distribuição e registro (*logging*).

Linguagens de programação orientadas a aspectos por exemplo, como AspectJ [52], fornecem o suporte necessário à separação de interesses, permitindo que um requisito transversal seja implementado separadamente, como um *interesse* específico, ou seja, um *aspecto* [173]. Desta forma, evita-se espalhamento do código (*code scattering*), aumentando a coesão funcional e tornando o projeto mais elegante e extensível (Figura 4.17).

Na programação orientada a aspectos, a separação de interesses ocorre apenas em tempo de desenvolvimento. Deve haver uma ferramenta, como AspectJ [52] por exemplo, para entrelaçar (ou combinar) o projeto e o código relacionados aos requisitos transversais com o projeto dos requisitos funcionais para que se tenha um sistema executável.




(a) Sistema sem separação de interesses: requisitos transversais espalhados.

(b) Sistema com separação de interesses: requisitos transversais modularizados.

Figura 4.17: Separação de interesses: modularização e extensibilidade.

Uma outra vantagem importante da separação de interesses na programação orientada a aspectos é a possibilidade de personalizar facilmente a execução do software. Isto ocorre através da escolha de que aspectos devem ser entrelaçados em um dado momento. Sendo assim, é possível obter múltiplas visões de execução de um sistema, uma para cada combinação de aspectos entrelaçados. Estas visões podem ser muito úteis para se observar o comportamento do sistema em relação a funcionalidades específicas, desabilitando outras funcionalidades [174].

Contudo, este mecanismo de personalização utilizando aspectos, provido pelas abordagens atuais, não é possível no contexto do desenvolvimento de sistemas baseadas em componentes de prateleira. Uma vez que componentes de prateleira são inerentemente artefatos de software caixa-preta [24], a informação sobre aspectos presente no código fonte dos componentes não está acessível ao desenvolvedor do sistema. Portanto, não é possível definir que aspectos devem ser ativados ou desativados em um dado momento, impedindo a personalização da execução do componente e, conseqüentemente, do sistema sendo desenvolvido (ver Figura 4.18).

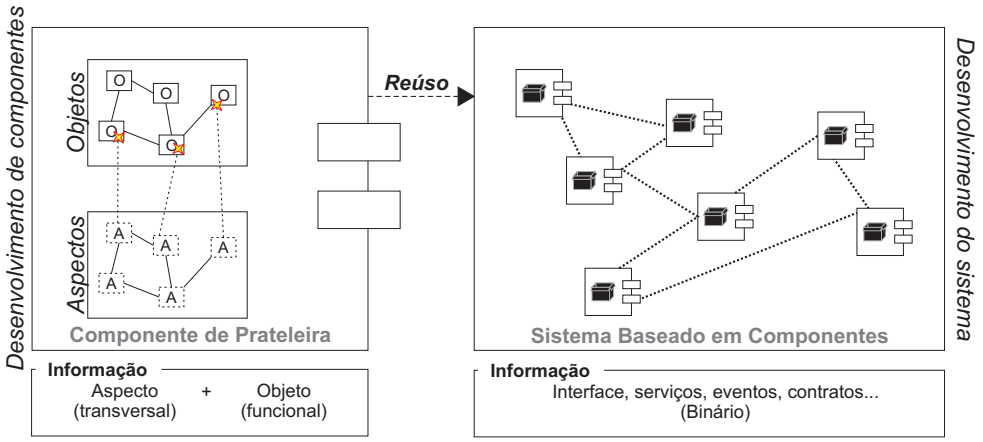



Figura 4.18: Componentes caixa-preta: perda de informação sobre os aspectos.

Para resolver este problema, a CMS provê suporte ao desenvolvimento de software baseado em componentes, permitindo a ativação e a desativação da implementação de interesses para cada componente do sistema. A metáfora utilizada é o conceito de interruptor. A idéia é disponibilizar ao desenvolvedor do sistema um mecanismo para “ligar” e ”desligar” os interesses implementados nos componentes, como interruptores de lâmpadas. Uma vez que a execução de cada componente pode ser personalizada separadamente, o desenvolvedor pode ter muitas visões de seu sistema, podendo executá-las de acordo com suas necessidades em um dado momento.

A arquitetura de personalização da execução de componentes, ilustrada na Figura 4.19, é composta por três entidades: interesses, interruptor e o componente de prateleira personalizável. Um componente de prateleira personalizável é um componente funcional que, no momento da reutilização e composição do sistema, pode ser configurado para personalizar sua execução. Vale ressaltar que a noção de componente personalizável é um conceito de projeto e implementação de software. Do ponto de vista de modelo de interação entre componentes e composição de sistema definidos no núcleo da CMS, um componente personalizável tem as mesmas características de

um componente funcional.

Um interesse é a implementação de um requisito transversal que pode ser ativado/desativado em tempo de reutilização/composição do componente e desenvolvimento do sistema. Cada componente possui um conjunto de interesses implementados e disponibilizados para ativação/desativação.

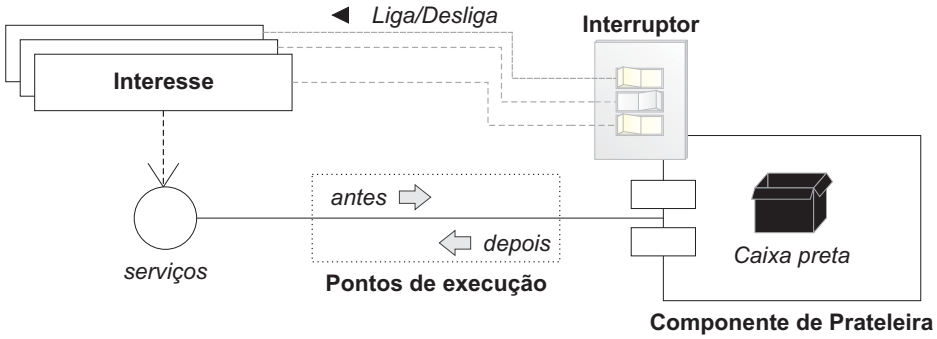



Figura 4.19: Arquitetura de personalização de execução de sistemas baseados na CMS.

A implementação dos interesses deve ser realizada durante a fase de desenvolvimento do componente e cada interesse pode estar relacionado a um conjunto de serviços providos pelos componentes personalizáveis. Para cada serviço relacionado, o desenvolvedor pode definir o código referente ao requisito transversal que deve ser executado *antes*, *depois* ou em caso de *exceção* na execução do serviço.

O interruptor controla quais interesses de um componente personalizável estão atualmente ativados ou desativados. O conjunto das possíveis combinações de ativação de interesses no interruptor representa o conjunto de possíveis visões de execução do componente. O estado do interruptor em um dado momento representa a visão de execução definida atualmente (ver Figura 4.20). Ao receber uma requisição de execução de serviço, o componente verifica junto ao seu interruptor quais interesses estão ativados. Então, executa o código relacionado aos mesmos.

Como será descrito no Capítulo 5, o projeto desta abordagem é simples, utilizando padrões de projeto de software, e não necessita de um processo de combinação utilizando uma ferramenta específica, como na programação orientada a aspectos.

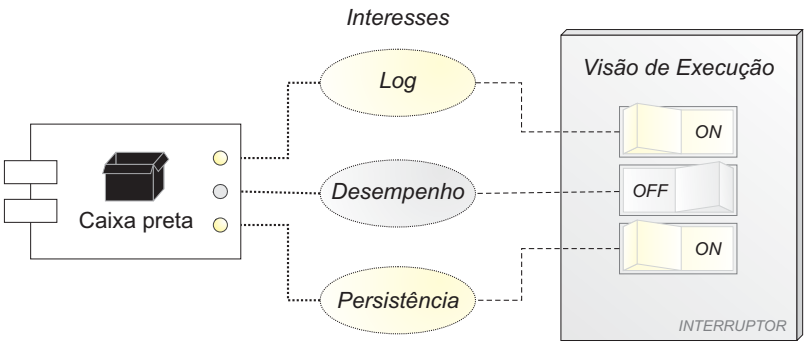



Figura 4.20: Interruptor de interesses.

### 4.2.5 Versionamento

Uma das características inerentes ao processo de desenvolvimento baseado em componentes é o versionamento. Durante o ciclo de vida de um componente, sua funcionalidade é melhorada e problemas de implementação encontrados durante sua reutilização em vários sistemas são resolvidos. Esta evolução da funcionalidade do componente é, em geral, acompanhada por um mecanismo de atribuição de versões.

O principal problema relacionado ao versionamento que deve ser considerado por tecnologias relacionadas ao desenvolvimento baseado em componentes é a execução paralela de várias versões de um dado componente. Por exemplo, considere que a versão 1.0 de um dado componente *x* é utilizada por outros componentes *y* e *w* do sistema. Considere também que uma nova versão (2.0) do componente *x* é disponibilizada, com melhorias que são indispensáveis a um melhor funcionamento do componente *y* mas que não é adequada às necessidades do componente *w*. Como manter as duas versões do componente funcionando no sistema, de modo que *w* utilize a versão 1.0 e *y* utilize a versão 2.0?

Na CMS, devido ao fato de o mecanismo de dependência de serviços e eventos funcionar com base na identificação dos mesmos, o versionamento tem que ser realizado no nível de serviços e eventos. Isto ocorre porque uma nova versão do componente contém, em geral, serviços com os mesmos identificadores da versão anterior. Sendo assim, ao adicionar as duas versões de um componente no sistema, poderão existir dois componentes com serviços de mesmo nome, o que não pode ocorrer de acordo com a CMS.

Para resolver este problema, utiliza-se o conceito de apelidos apresentado anteriormente. Ao adicionar uma nova versão (2.0) do componente *x*, deve-se alterar os apelidos de seus serviços

para que não entrem em conflito com os da versão 1.0. Pode-se inclusive definir o apelido do serviço de acordo com sua versão, como por exemplo, *calcular2.0*, *salvar2.0* e *abrir2.0*. Após a adição do componente, não haverá conflitos entre os nomes dos serviços e, assim, basta redefinir o apelido do serviço requerido pelo componente *y* para que se estabeleça uma dependência com a versão 2.0 do serviço. O componente *w* irá se manter utilizando a versão 1.0 do serviço. Na Figura 4.21, ilustra-se este processo.

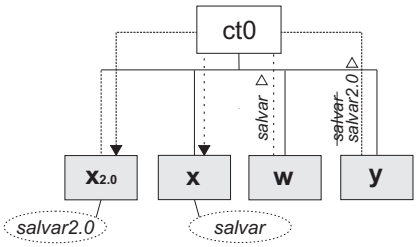



Figura 4.21: Versionamento na CMS.

O fato do versionamento ser realizado utilizando a estrutura de apelidos de serviços e eventos não significa que este mecanismo deve ser gerenciado pelo desenvolvedor. Pode-se prover ferramentas, acopladas ao ambiente de desenvolvimento (ver Capítulo 9), que implemente a gerência de versionamento, alterando automaticamente o apelido dos serviços de acordo com a versão do componente.

### 4.2.6 Execução dos Componentes e do Sistema

Todas as funcionalidades de um sistema baseado em componentes de acordo com a CMS podem ser implementadas utilizando componentes funcionais. Mesmo aquelas relacionadas à interação com o usuário, como componentes de interface gráfica e linha de comando. Porém, em alguns casos, a especificidade deste tipo de funcionalidade pode tornar difícil a reutilização do componente correspondente em outros contextos. Nestes casos, a construção de um componente com interface bem definida e preparado para reutilização traz um custo desnecessário dada a baixa probabilidade de que este seja reutilizado posteriormente.

Para evitar este problema, podem ser utilizados *scripts* de execução. Um *script* de execução implementa a rotina principal de execução do sistema. Ele pode ser comparado aos métodos *main* de linguagens como Java, C e C++. Através da implementação do *script* de execução o

desenvolvedor tem acesso ao contêiner raiz do sistema e, a partir dele, acesso a todos os serviços e eventos disponibilizados na hierarquia de componentes.

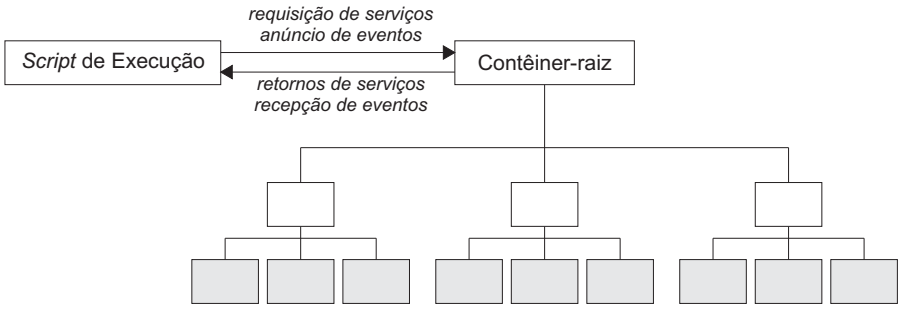



Figura 4.22: *Script* de execução: acesso a serviços e eventos através do contêiner raiz.

Um outro aspecto relacionado à execução do sistema diz respeito à inicialização e à interrupção da execução de seus componentes. Estas são rotinas opcionais a serem implementadas pelo desenvolvedor do componente. Entenda-se inicialização como o processo que inicializa o estado do componente e o torna pronto para prover sua funcionalidade. Interrupção, por sua vez, é o processo através do qual o componente finaliza sua execução, tornando suas funcionalidades indisponíveis.

Um exemplo simples da necessidade destes processos é um componente para comunicação com banco de dados. Durante a inicialização do componente, deve-se realizar as configurações necessárias para a utilização dos dados, assim como estabelecer comunicação com o servidor de banco de dados. Antes disso, não é possível acessar o banco e, sendo assim, os serviços/eventos do componente não devem estar disponíveis. A interrupção deste componente deve finalizar todas as conexões com o banco, aguardando o término das transações que ainda estão em processamento. Caso esta finalização não ocorra, podem ocorrer problemas nas transações em andamento ou em conexões futuras com o banco de dados.

Para efetuar a inicialização e a interrupção, o componente pode precisar de informações específicas do ambiente em que foi reutilizado. Estas informações são aqui denominadas *propriedades de inicialização*. *Propriedades de inicialização* são em geral utilizadas para configurar a inicialização e a interrupção da execução do componente em um dado sistema. Por exemplo, um componente *x* que provê comunicação de processos via rede precisa ter sua propriedade de inicialização configurada para se comunicar utilizando uma porta específica. Tem-se então uma propriedade *porta*, cujo valor pode ser configurado pelo desenvolvedor do sistema para *4444*, por

exemplo.

A inicialização/interrupção pode ser realizada de forma unitária (componente) ou hierárquica (contêiner). Ao inicializar/interromper um contêiner, todos as suas entidades filhas devem ser inicializadas/interrompidas, recursivamente, em uma ordem especificada pelo desenvolvedor do sistema. Sendo assim, ao inicializar/interromper o contêiner raiz, todos os contêineres e componentes funcionais do sistema serão inicializados/interrompidos. No caso da utilização de um *script* de execução, a inicilização/interrupção do *script* deve desencadear a inicialização/interrupção do contêiner raiz. Uma vez iniciada a hierarquia de componentes, todos os serviços providos e eventos anunciados pelos componentes funcionais estarão em funcionamento e, conseqüentemente, o sistema estará executando.

## 4.3 Cenários de Evolução

As definições inerentes à CMS apresentadas anteriormente estão relacionadas com o desenvolvimento, adaptação, personalização e execução de componentes funcionais e com a montagem de sistemas com base em uma hierarquia de contêineres e componentes funcionais. Nenhuma característica relacionada à evolução do software foi explicitamente mencionada anteriormente.

Isto ocorre porque, como descrito na introdução deste capítulo, o desenvolvedor que utiliza a CMS não deve estar preocupado com os mecanismos que permitem a evolução dinâmica não antecipada do software. Para ele, os possíveis cenários de evolução devem ser refletidos nas operações inerentes à construção do software, tais como adição, remoção de contêineres, componentes funcionais e adaptadores, herança caixa-preta, personalização, etc. Na Figura 4.23 são ilustrados os possíveis cenários de evolução dinâmica não antecipada de um sistema baseado na CMS. Cada um destes cenários é descrito a seguir, ordenados de acordo com o nível de complexidade inerente à atividade de evolução e relacionados com as operações definidas na CMS que contemplam tais cenários. Entenda-se complexidade por esforço necessário (tempo, custo...) para contemplar o cenário de evolução.

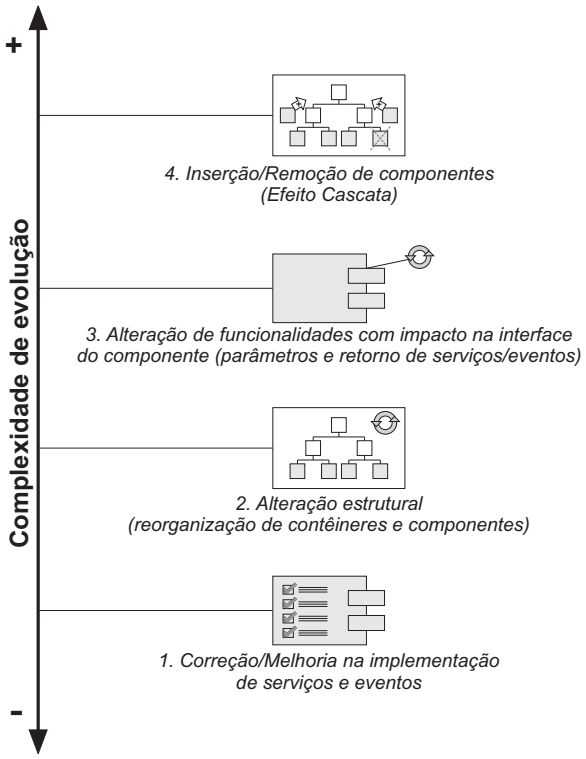



Figura 4.23: Cenários de evolução e níveis de complexidade.

## Nível 1 - Correção/Melhoria na implementação de serviços e eventos

O cenário de evolução com nível de complexidade mais baixo é a correção ou melhoria na implementação de serviços e eventos. Neste cenário, não há alteração na interface dos componentes funcionais, ou seja, seus serviços e eventos são alterados mas estes se mantêm com o mesmo identificador, parâmetros de execução e retorno.

Para contemplar este cenário em um sistema baseado na CMS, tem-se uma "troca" de componentes funcionais, ou seja, uma adição de um novo componente seguida de uma remoção do componente antigo, como definido na CMS. As tabelas de serviços e eventos do contêiner pai do componente removido são atualizadas e as próximas requisições de serviços e notificações de eventos serão automaticamente redirecionadas para o novo componente.

Na Figura 4.24, ilustra-se este processo. O componente *k* é trocado pelo componente *z* (ou uma nova versão do componente *k*) que provê, pelo menos, os mesmos serviços que o componente *k* provê e está interessado nos mesmos eventos de interesse do componente *k*. Evidentemente, se algum novo evento de interesse for inserido, ou serviço requerido, as dependências devem ser verificadas para que se mantenha uma dependência completa. Este é o caso tratado no nível 4.

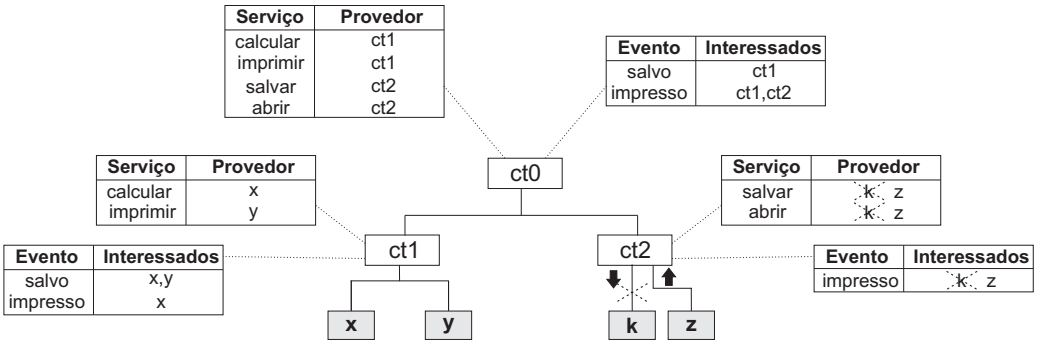



Figura 4.24: Cenário de evolução de nível 1 de complexidade.

Este é o cenário de evolução mais simples de ser contemplado pela CMS. De fato, este é um cenário contemplado por várias outras abordagens. Uma vez que a interface se mantém, várias linguagens orientadas a objetos permitem a troca do objeto que implementa tal interface, mesmo em tempo de execução. É com base neste mecanismo que arcabouços de aplicação e outros trabalhos relacionados ao desenvolvimento baseado em componentes oferecem suporte a este cenário, como descrito no Capítulo 3.

## Nível 2 - Alteração estrutural (reorganização de contêineres e componentes)

O cenário de evolução com nível de complexidade 2 é a alteração da estrutura do sistema, ou seja, a reorganização de contêineres e componentes funcionais na hierarquia. Neste nível não estão sendo considerados adição e remoção de novos componentes funcionais. São consideradas apenas a adição e remoção de novos contêineres que possam alterar a estrutura da hierarquia e a mudança de contêineres pais dos componentes funcionais existentes.

Para contemplar este cenário, são utilizados os processos de adição e remoção de contêineres e componentes funcionais definidos na CMS. Na Figura 4.25, ilustra-se a reorganização da hierarquia originalmente descrita no nível 1. Observe que novos contêineres foram inseridos e os componentes funcionais realocados em outros contêineres. As funcionalidades, no entanto, mantêm-se as mesmas, uma vez que novos componentes funcionais não foram adicionados. As alterações nas tabelas de serviços e eventos dos contêineres estão destacadas em cinza na Figura 4.25.

Apesar da simplicidade com que se contempla este cenário na CMS, ele é considerado um cenário bem complexo para um software existente e em funcionamento. Alterar a arquitetura de

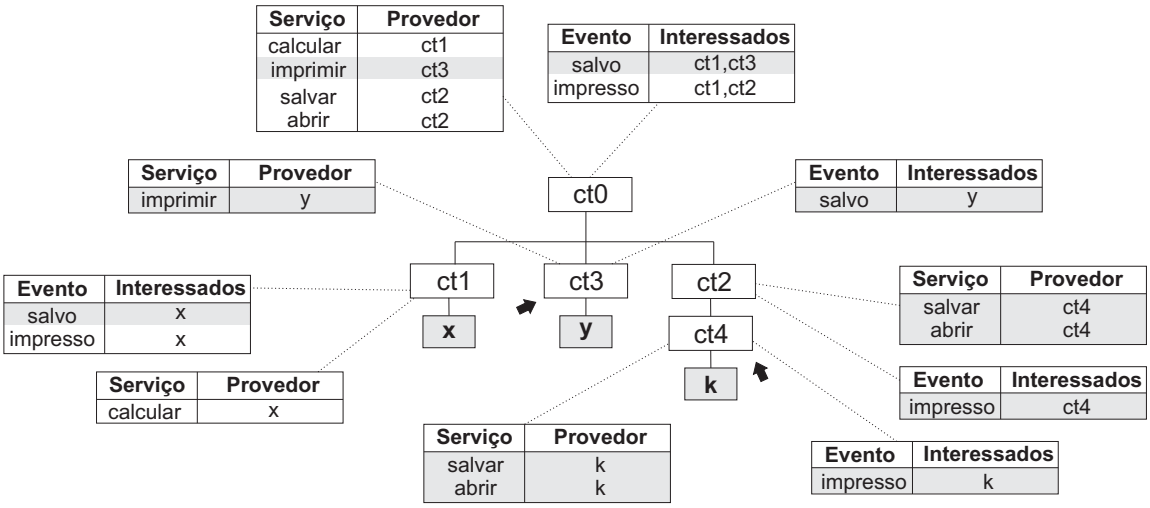



Figura 4.25: Cenário de evolução de nível 2 de complexidade.

um sistema em tempo de execução é algo não trivial nas diversas abordagens existentes para o desenvolvimento de software. A mudança arquitetural de um modo geral é apontada como de grande impacto sobre o projeto de baixo nível e implementação de um sistema [175].

## Nível 3 - Alteração de funcionalidades com impacto na interface do componente (parâmetros e retorno de serviços/eventos)

O cenário de evolução com nível de complexidade 3 é a alteração de funcionalidades com impacto na interface do componente. Uma vez que a mudança nos identificadores de serviços e eventos pode ser facilmente resolvida através de apelidos, o principal problema está relacionado à alteração de parâmetros e retorno de serviços e eventos. Quando isto ocorre, os componentes funcionais dependentes daquele serviço ou evento continuam referenciando o identificador correto, mas com parâmetros inválidos eles não funcionarão da forma correta.

Para contemplar este cenário, pode-se inserir um adaptador ao componente cujos serviços ou eventos tiveram seus parâmetros ou retorno alterados, como definido na CMS. Como ilustrado na Figura 4.26, pode-se adaptar os serviços ou eventos necessários para que um parâmetro a mais seja inserido ou removido para adequar a interface do provedor às requisições.

Vale ressaltar que no exemplo ilustrado na Figura 4.26 foi possível adaptar o novo componente funcional implementando o mesmo comportamento para todos os componentes requisitantes. Nos casos em que o novo parâmetro influencie no comportamento e dependa do requisitante do ser-

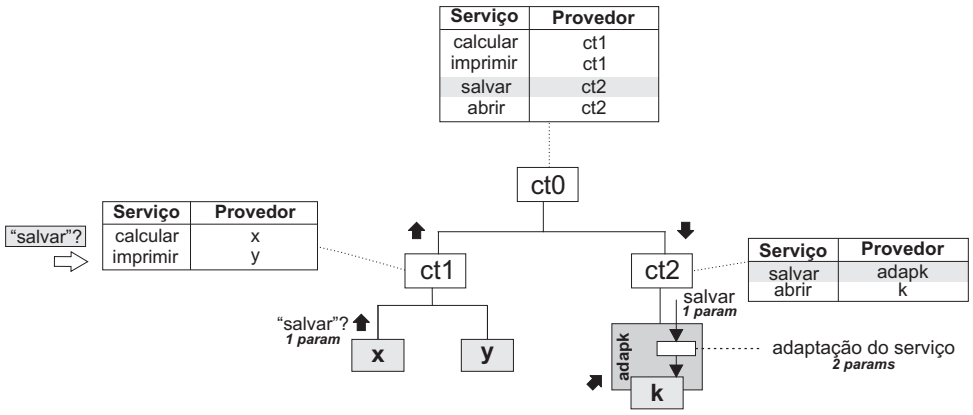



Figura 4.26: Cenário de evolução de nível 3 de complexidade.

viço, pode-se utilizar versionamento para o novo componente ou uma herança caixa-preta, como definido na CMS. Em casos mais complicados, onde os requisitantes tiverem que ser alterados, tem-se a mesma situação de efeito cascata que ocorre no nível 4.

## Nível 4 - Inserção/Remoção de componentes (Efeito Cascata)

O cenário de evolução com nível de complexidade mais alto é a inserção/remoção de componentes e o efeito cascata causado por estes processos. Diante do surgimento de novos requisitos para o sistema, novos componentes devem ser adicionados e componentes existentes podem ser removidos. Estas atividades acarretam um processo de gerência de dependências em efeito cascata.

A gerência de dependências é motivada por cenários como aqueles descritos no nível 1 e nível 3. Ela é necessária quando novos serviços ou eventos são inseridos e precisam ter suas dependências verificadas. O mesmo ocorre quando um componente existente é removido e os outros componentes que utilizavam seus serviços e eram interessados em seus eventos passam a ter problemas de dependência.

Ao alterar um componente *x*, por exemplo, pode-se ter uma alteração necessária no componente *y*, a qual pode afetar um componente *w* e, por sua vez, um outro componente *z*. Um exemplo deste efeito cascata é ilustrado na Figura 4.27, onde um novo componente relacionado ao modelo de negócio é inserido no sistema (*rellucros*) e precisa ter suas funcionalidades disponibilizadas também na interface do usuário, cujos componentes estão agrupados no contêiner *ctui*. Após a adição do novo componente, as funcionalidades não estão disponíveis para utilização pois não estão acessíveis via interface do sistema. Isto ocorre porque os componentes de interface de usuário

(GUI, de *Graphical User Interface*, e CLI, de *Command Line Interface*) não conhecem o novo componente e, portanto, precisam ser alterados.

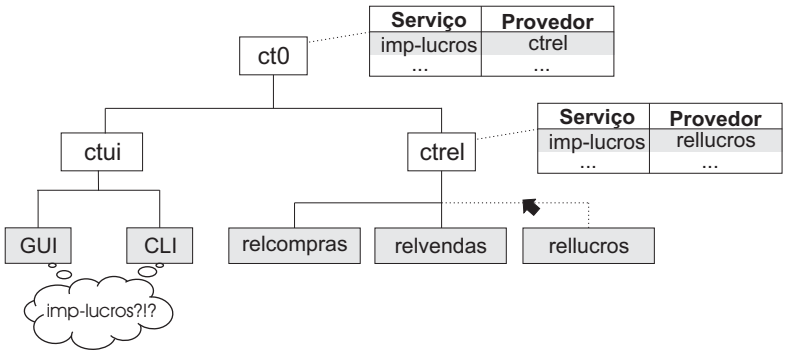



Figura 4.27: Cenário de evolução de nível 4 de complexidade.

De uma maneira geral, tem-se os seguintes passos para a gerência de dependências:

1. Identificar quais componentes foram afetados por uma alteração/inserção/remoção de um dado componente;

2. Identificar suas dependências de serviços e eventos e os problemas causados pela alteração/inserção/remoção do componente;

3. Aplicar uma das operações definidas na CMS para cada problema de dependência (adição e remoção de componente ou adaptador, versionamento, herança caixa-preta, etc), ou seja, para cada componente afetado;

4. Verificar se a dependência do sistema é completa, caso contrário, refazer os passos de 1 a 4 com os componentes alterados no processo.

A quantidade de iterações nesse processo depende do nível de coesão funcional do sistema e do tipo de evolução sendo realizada. Gerenciar as dependências após a inserção de uma nova funcionalidade pode ser uma tarefa custosa mas, utilizado a CMS, ela será uma tarefa possível mesmo em tempo de execução.

## 4.4 Considerações Finais

Neste capítulo foi apresentada a especificação do modelo de componentes (CMS) para o desenvolvimento de software com suporte à evolução dinâmica não antecipada. De acordo com a CMS,

um sistema baseado em componentes é formado por uma hierarquia de contêineres e componentes funcionais. Os componentes funcionais não contêm outros componentes e implementam as funcionalidades do sistema através de eventos e serviços. Já os contêineres não implementam funcionalidades e apenas gerenciam o acesso aos serviços e eventos de seus componentes filhos. Uma vez que não há acoplamento direto entre componentes funcionais, pode-se adicionar, alterar e remover componentes e seus serviços e eventos sem alterar os demais, provendo ao desenvolvedor o suporte à evolução dinâmica não antecipada.

Apresentou-se inicialmente a descrição formal da CMS utilizando teoria dos conjuntos. Tal descrição engloba características estruturais do modelo, como a composição de componentes e contêineres, assim como características funcionais, como a invocação de serviços e o anúncio e recepção de eventos.

Aspectos pragmáticos do modelo de componentes foram então apresentados, mais especificamente, detalhando o suporte a requisitos indispensáveis a sistemas baseados em componentes, tais como reutilização, adaptação, versionamento e execução. Além disso, foi apresentado um mecanismo para a herança de implementação de serviços, mesmo considerando componentes caixa-preta.

Por fim, foram apresentados os cenários de evolução dinâmica de software não antecipada no contexto de sistemas baseados em componentes, englobando desde alterações funcionais internas dos componentes até alterações estruturais e na interface dos componentes, causando o efeito cascata de evolução. Destaca-se então o suporte oferecido pela CMS ao desenvolvedor para gerenciar a implementação de tais cenários.

A descrição da CMS como técnica independente de linguagem e paradigma de programação facilita o entendimento do modelo por parte de desenvolvedores para a implementação de arcabouços de software que implementam a CMS, como apresentado no Capítulo 5. Da mesma forma, tem-se uma base conceitual para futuras extensões do modelo de componentes, ainda mantendo a simplicidade de seu núcleo.

# Capítulo 5

# Arcabouços de Software Baseados na CMS

Neste capítulo são apresentados os principais aspectos relacionados aos arcabouços de software que implementam a CMS. Inicialmente, descreve-se um arcabouço orientado a objetos genérico, independente de linguagem, que serve como base para a implementação de arcabouços em linguagens específicas. Por fim, apresentam-se os principais desafios relacionados à implementação de arcabouços em linguagens específicas, com base na experiência do desenvolvimento de arcabouços em Java, Python, CSharp e C++.

A descrição detalhada dos requisitos e do projeto orientado a objetos do GCF é apresentada no Apêndice A. No Apêndice B, tem-se uma descrição detalhada da implementação dos arcabouços nas linguagens Java, Python, C++ e CSharp, incluindo diretrizes para a utilização dos mesmos.

## 5.1 *Generic Component Framework* - GCF

Existem diversas linguagens de programação, sendo cada uma delas mais adequada para determinados requisitos e plataformas de desenvolvimento e implantação de software. Java [56] e C# [58], por exemplo, têm sido largamente utilizadas para o desenvolvimento de aplicações corporativas. Python [55], por sua vez, é uma linguagem voltada para a prototipação rápida de software e, assim como PHP [176], tem sido muito utilizada para a programação de aplicações voltadas para a Web. Para aplicações *desktop*, Object Pascal – a linguagem utilizada pelo ambiente Delphi [177] – e Visual Basic [178] são muito utilizadas.

Além disso, restrições da plataforma de hardware ou do sistema operacional de execução

da aplicação também podem determinar a escolha da linguagem mais adequada. Programação para o sistema operacional Windows, por exemplo, torna-se bem mais produtiva se realizada com Visual Basic ou C#. Programação para dispositivos móveis pode ser realizada utilizando-se Java/J2ME [179] ou Python mas, para algumas aplicações que necessitam de operações de mais baixo nível, linguagens como Symbian/C++ [180] são indispensáveis.

Dada a diversidade de linguagens e especificidades das aplicações, disponibilizar uma infra-estrutura de desenvolvimento de software em apenas uma linguagem pode acarretar dois problemas: diminuir a abrangência de utilização da infra-estrutura ou forçar a sua utilização mesmo em cenários em que a linguagem não seja adequada. A resolução destes problemas é a principal motivação para a definição de um arcabouço independente de linguagem para a CMS. Este arcabouço genérico é denominado GCF: *Generic Component Framework*.

GCF é um arcabouço "caixa-cinza" [17]. Isto porque o projeto do GCF contém tanto pontos de extensão com base em herança de classes, usados para o desenvolvimento de novos componentes, como pontos de extensão com base em composição, usados para a montagem da aplicação.

O projeto do GCF é baseado no padrão *Composite* [16], que é utilizado para permitir a composição recursiva de contêineres e componentes funcionais definida na CMS. As principais classes do projeto do GCF são apresentadas no diagrama da Figura 5.1. As classes `FunctionalComponent` e `Container` são instanciadas, respectivamente, em componentes funcionais e contêineres, como especificados na CMS. A classe abstrata `AbstractComponent` garante a composição recursiva, permitindo que os contêineres "desconheçam" a implementação dos seus componentes filhos, que podem ser tanto componentes funcionais como outros contêineres. Além disso, ela implementa os métodos comuns às suas subclasses e estabelece um conjunto de contratos como métodos abstratos que devem ser implementados pelas mesmas.

No diagrama simplificado ilustrado na Figura 5.1 são descritos os principais métodos que implementam os modelos de interação baseada em serviços e eventos da CMS. Uma descrição completa dos métodos e das classes que fazem parte do projeto orientado a objetos do GCF é apresentada no Apêndice A.

A interação baseada em serviços é implementada através de invocações sucessivas do método `doIt`, "de baixo para cima" na hierarquia de componentes até que o contêiner que registra o serviço seja encontrado. Quando isto acontece, o método `receiveRequest` é invocado "de cima para baixo" na hierarquia até que o componente funcional que implementa o serviço seja encontrado,

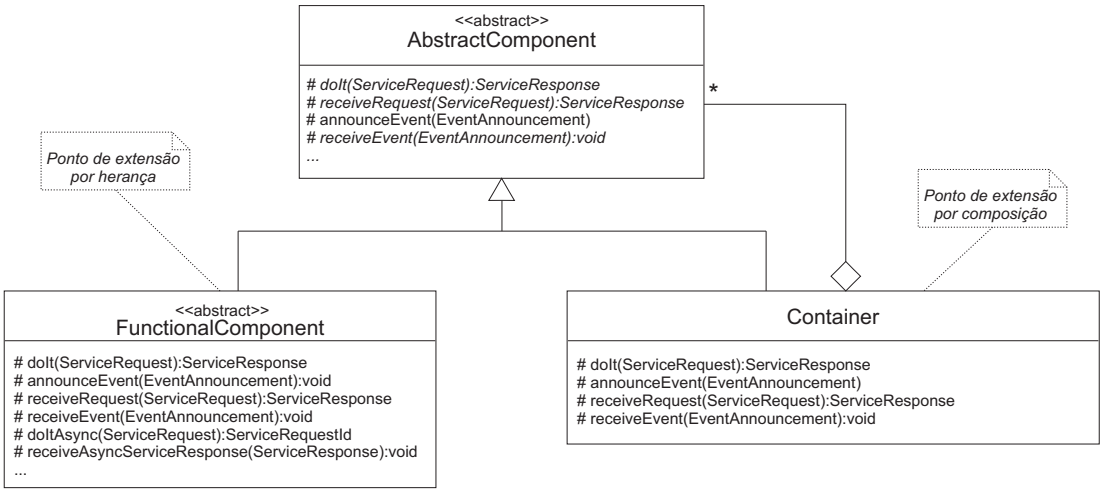



Figura 5.1: Principais classes do projeto do GCF.

como ilustrado na Figura 5.2. Caso o contêiner raiz seja alcançado e o provedor do serviço não seja encontrado, o arcabouço deve retornar um erro ao desenvolvedor. As exceções de execução do GCF são apresentadas posteriormente. Este processo é baseado na definição de invocação de serviço descrita no Capítulo 4.

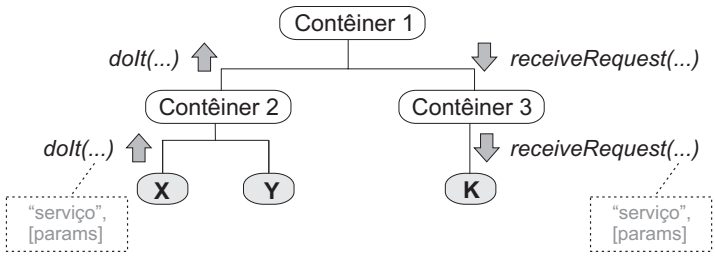



Figura 5.2: Interação entre objetos no modelo de interação baseada em serviços do GCF.

O mesmo processo ocorre na interação baseada em eventos, com invocações sucessivas do método `announceEvent`, de "baixo para cima" na hierarquia de componentes, até que o componente raiz seja encontrado (ver Figura 5.3). A cada contêiner, verifica-se o registro de interessados no evento e o método `receiveEvent` é invocado "de cima para baixo" na hierarquia, até que todos os componentes funcionais interessados no evento sejam notificados. Este processo é baseado na definição de anúncio de evento descrita no Capítulo 4.

O modelo de adaptadores definido na CMS foi projetado no GCF como uma extensão simples do seu núcleo. Na Figura 5.4, ilustra-se o diagrama simplificado de classes relacionadas ao projeto do modelo de adaptadores definido na CMS. A principal classe do diagrama é a classe

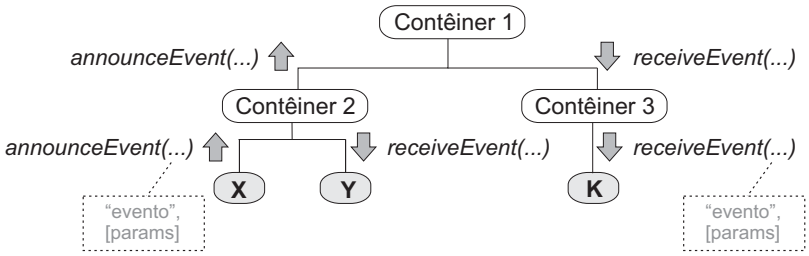



Figura 5.3: Interação entre objetos no modelo de interação baseada em eventos do GCF.

`Adapter`, que é uma subclasse de `Container` com uma referência direta a uma instância de `FunctionalComponent` – o componente funcional sendo adaptado. As classes `AdaptedEvent` e `AdaptedService` são estendidas para implementar serviços e eventos adaptados, registrados no próprio adaptador do componente. O adaptador então deve repassar requisições a serviços e eventos para os seus respectivos serviços e eventos adaptados sempre que estes estiverem registrados.

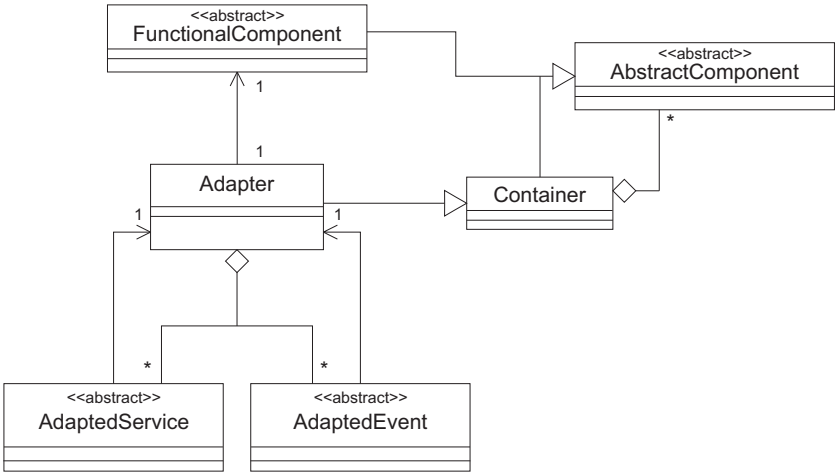



Figura 5.4: Diagrama simplificado de classes relacionadas ao modelo de adaptadores.

Por fim, tem-se o projeto do modelo de separação de interesses definido na CMS. Na Figura 5.5, ilustra-se o diagrama simplificado de classes relacionadas ao projeto do modelo de separação de interesses. A classe principal deste diagrama é `CustomizableComponent`, que representa um componente personalizável de acordo com a CMS. Isto significa que uma instância deste componente possui um *interruptor de interesses*, representado pela composição de uma instância da classe `ConcernSwitch`. Cada interesse é implementado como uma extensão da classe `Concern` e pode ser ativado ou desativado através dos métodos da classe `ConcernSwitch`. Através da extensão da classe `Concern`, pode-se sobrescrever métodos para interceptação da execução

de serviços antes (`interceptBefore(...)`), depois (`interceptAfter(...)`) e em caso de exceções (`interceptOnException(...)`).

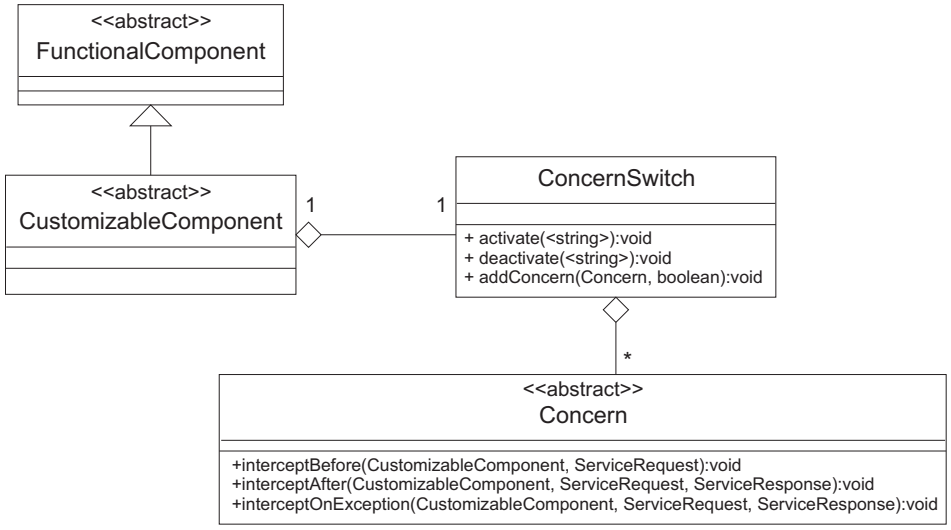



Figura 5.5: Diagrama simplificado de classes relacionadas ao modelo de separação de interesses.

Como mencionado anteriormente, os detalhes sobre cada classe do projeto do GCF e seus respectivos métodos são descritos no Apêndice A deste documento. Neste capítulo, tem-se apenas uma visão geral de como o GCF foi projetado e, a seguir, das implementações deste arcabouço que já foram realizadas.

## 5.2 Arcabouços Dependentes de Linguagem

Com base no projeto genérico do GCF, foram implementados diversos arcabouços nas linguagens Java (*Java Component Framework* - JCF), CSharp (*Sharp Component Framework* - SCF), Python (*Python Component Framework* - PyCF) e C++ (*C++ Component Framework* - CCF). Os três últimos foram implementados em cooperação com desenvolvedores de dois projetos de pesquisa do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) [61, 62] e em trabalhos de conclusão de curso de graduação [77, 78]. Esta participação de desenvolvedores que não estiveram envolvidos na concepção da CMS e do GCF contribuiu muito para o enriquecimento da especificação do projeto independente de linguagem apresentado anteriormente e detalhado no Apêndice A.

Os principais desafios inerentes à implementação do GCF estão relacionados à reflexão com-

putacional e à invocação assíncrona de métodos. Os demais aspectos de projeto do GCF são implementados por grande parte das linguagens orientadas a objetos, incluindo aquelas escolhidas neste trabalho.

A reflexão computacional é necessária para que um método que implementa um serviço ou trata a recepção de um evento tenha sua referência armazenada e seja invocado apenas através da cadeia de caracteres que o identifica. Essa característica é indispensável à utilização de identificadores e apelidos de serviços e eventos, assim como à manutenção da tabela de serviços e eventos de cada contêiner, tal como definido na CMS.

Considerando os arcabouços específicos de linguagem, o JCF, o PyCF e o SCF possuem um suporte semelhante à reflexão computacional, ao menos no que diz respeito à referência e invocação de métodos com base em uma cadeia de caracteres. Já a linguagem C++ não disponibiliza este suporte em sua API padrão. Para isso, utilizou-se a API de reflexão Seal Reflex [181] que possibilita a um programa C++ recuperar informações sobre uma classe e seus membros, assim como criar instâncias de uma classe através de seu nome, além de referenciar e invocar métodos através de informações de sua assinatura.

Já a funcionalidade de invocação assíncrona está relacionada à implementação do modelo de interação baseada em eventos e em serviços assíncronos. É necessário que um determinado método possa ser executado em uma nova linha de execução e que o resultado seja retornado à linha principal uma vez que a execução do serviço ou o tratamento do evento tenha sido finalizado.

Diferentemente do que ocorre com a funcionalidade de reflexão computacional, todas as implementações do GCF utilizaram o mesmo conceito para a implementação da funcionalidade de invocação assíncrona de métodos: *threads*. Mais especificamente, utilizou-se o padrão Future/Active Object [182], que provê uma solução elegante para esta funcionalidade.

Buscou-se na implementação dos arcabouços maximizar a semelhança da API de desenvolvimento disponibilizada ao usuário. Isto pode ser observado no exemplo de instanciação de componentes e contêineres, composição e execução de uma aplicação descrito a seguir. Na Listagem de Código 5.1, descreve-se a versão em Java da API, ou seja, do JCF. Nas linhas 2 a 6, são criadas instâncias de componentes e contêineres. Nas linhas 9 a 12, a hierarquia da aplicação é definida. Por fim, nas linhas 15 e 16, cria-se e inicia-se o *script* de execução e, dessa forma, a aplicação propriamente dita.

Listagem de Código 5.1: Exemplo de utilização da API do JCF.

```
1  //Instanciando os componentes, contêineres...
2  ComponentA compA = new ComponentA();
3  ComponentB compB = new ComponentB();
4  Container contA = new Container("ContainerA");
5  Container contB = new Container("ContainerB");
6  ScriptContainer root = new ScriptContainer("Root");
7
8  //Criando a hierarquia da aplicação...
9  contA.addComponent(compA);
10 contB.addComponent(compB);
11 root.addComponent(contA);
12 root.addComponent(contB);
13
14 //Criando e iniciando o script de execução
15 MyScript myScript = new MyScript(root);
16 myScript.init();
```

Na Listagem de Código 5.2, descreve-se a versão em Python do exemplo, ou seja, utilizando o PyCF. Excetuando-se as diferenças de sintaxe de linguagem, a estrutura do código é bem similar à versão em Java.

Listagem de Código 5.2: Exemplo de utilização da API do PyCF.

```
1  #Instanciando os componentes, contêineres...
2  compA = ComponentA();
3  compB = ComponentB();
4  contA = Container("ContainerA");
5  contB = Container("ContainerB");
6  root = ScriptContainer("Root");
7
8  #Criando a hierarquia da aplicação...
9  contA.addComponent(compA);
10 contB.addComponent(compB);
11 root.addComponent(contA);
12 root.addComponent(contB);
13
14 #Criando e iniciando o script de execução
15 myScript = MyScript(root);
16 myScript.init();
```

Na Listagem de Código 5.3, descreve-se a versão em C++ do exemplo. Tem-se uma grande semelhança desta implementação em relação às versões em Python e Java. Há poucas diferenças considerando-se toda a API do CCF em relação aos demais. A principal delas é que, ao contrário

das outras versões de arcabouços, o CCF exige, em alguns pontos, a implementação de métodos para desalocação de objetos da memória. Isto pode ser verificado no detalhamento do arcabouço apresentado no Apêndice B.

Listagem de Código 5.3: Exemplo de utilização da API do CCF.

```
1  //Instanciando os componentes...
2  ComponentA * compA = new ComponentA();
3  ComponentB * compB = new ComponentB();
4
5  //Criando a hierarquia da aplicação...
6  ScriptContainer * root = new ScriptContainer("Root");
7  Container * contA = new Container("ContainerA");
8  Container * contB = new Container("ContainerB");
9
10 //Criando a hierarquia da aplicação...
11 contA->addComponent(compA);
12 contB->addComponent(compB);
13 root->addComponent(contA);
14 root->addComponent(contB);
15
16 //Criando e iniciando o script de execução
17 MyScript * myScript = new MyScript(root);
18 myScript->init();
```

Finalmente, na Listagem de Código 5.4, descreve-se a versão em CSharp da implementação do exemplo descrito anteriormente, que é idêntica à versão em Java.

Listagem de Código 5.4: Exemplo de utilização da API do SCF.

```
1  //Instanciando os componentes, contêineres...
2  ComponentA compA = new ComponentA();
3  ComponentB compB = new ComponentB();
4  Container contA = new Container("ContainerA");
5  Container contB = new Container("ContainerB");
6  ScriptContainer root = new ScriptContainer("Root");
7
8  //Criando a hierarquia da aplicação...
9  contA.addComponent(compA);
10 contB.addComponent(compB);
11 root.addComponent(contA);
12 root.addComponent(contB);
13
14 //Criando e iniciando o script de execução
15 MyScript myScript = new MyScript(root);
16 myScript.init();
```

Assim como no exemplo descrito anteriormente, as demais funcionalidades da API das diferentes implementações do GCF são utilizadas de forma bem semelhante. Isto ocorre devido à especificação minuciosa do GCF, detalhada no Apêndice A, que serviu como base para todas os arcabouços específicos de linguagem. A grande motivação para a implementação de diversas implementações do arcabouço foi fortalecer o argumento de que a CMS é independente de linguagem e validar o projeto do GCF. Com base nesses arcabouços, é possível desenvolver software com suporte à evolução dinâmica não antecipada em linguagens diferentes, com um esforço mínimo de aprendizado ao migrar de uma linguagem para outra.

## 5.3 Considerações Finais

Neste capítulo foram apresentados os arcabouços de componentes que implementam a CMS. Mais especificamente, apresentou-se um projeto genérico de arcabouço baseado em componentes e foram discutidas as implementações existentes em linguagens específicas, como Java, Python, C++ e CSharp. Descrições mais detalhadas destes arcabouços são apresentadas nos apêndices A e B.

Utilizando os arcabouços apresentados neste capítulo é possível desenvolver componentes e compor e executar aplicações de acordo com os modelos de composição, interação e execução da CMS. Uma vez construída e iniciada a aplicação, deve-se considerar os diversos cenários de evolução não antecipada possíveis a serem realizados dinamicamente. Para implementar os cenários contemplados pela CMS descritos no Capítulo 4, não é necessário um conjunto específico de funcionalidades. As funcionalidades do GCF mencionadas anteriormente são suficientes para contemplar os cenários de evolução dinâmica não antecipada.

A principal questão relacionada à capacidade de uma implementação do GCF de prover evolução dinâmica não antecipada é como inserir dinamicamente na aplicação uma classe que não havia sido prevista anteriormente (carregamento dinâmico de classes). Por decisão de projeto, decidiu-se retirar esta responsabilidade da implementação dos arcabouços e colocá-las em servidores de aplicação, assim como ocorre em diversas outras tecnologias de componentes, tais como EJB [39] e .NET [183].

Estes servidores de aplicação são responsáveis por gerenciar o ciclo de vida dos componentes e das aplicações, assim como sua evolução. Cada linguagem possui seu próprio mecanismo

de carregamento dinâmico de classes e, portanto, ter-se-á um servidor de aplicação para cada implementação do GCF, como descrito no Capítulo 10.

Caso o desenvolvedor não deseje utilizar um servidor de aplicação, ele ainda poderá utilizar um arcabouço, mas terá que implementar e gerenciar os mecanismos de carregamento dinâmico de classes. Neste documento, considera-se que toda a infra-estrutura de desenvolvimento será utilizada, incluindo os servidores de aplicação. Por isso, não foi descrito um projeto genérico para carregamento dinâmico de classes, assim como não foram descritas as implementações deste mecanismo em diferentes linguagens.

# Capítulo 6

# Suporte para Aplicações Corporativas

A CMS e o GCF têm como principais características a simplicidade e a flexibilidade. Tais características permitem que sejam desenvolvidas aplicações com capacidade de evolução dinâmica mesmo quando não antecipada. Contudo, em aplicações desenvolvidas para domínios corporativos, outras características tornam-se indispensáveis.

Existem diversas plataformas para o desenvolvimento de aplicações corporativas, sendo J2EE e .NET as mais utilizadas atualmente. Estas plataformas oferecem um conjunto de características que, juntas, fornecem o suporte necessário para a construção de tais aplicações. Dentre estas características, destacam-se: distribuição, segurança, persistência, transações, suporte à Web e integração com sistemas legados [141].

Para manter a simplicidade da CMS, o suporte a aplicações corporativas foi projetado apenas sobre a arquitetura do GCF. Sendo assim, ele não está especificado na CMS. É o arcabouço que provê módulos extras para lidar com tais características.

Porém, se por um lado o suporte supracitado é imprescindível a aplicações corporativas, por outro lado, existem sistemas em que o mesmo não é requerido. Nestes casos, a alteração da arquitetura do GCF apenas introduz complexidade de entendimento e utilização aos desenvolvedores.

A forma encontrada para diminuir a complexidade do arcabouço, do ponto de vista do desenvolvedor, e ainda prover suporte corporativo, foi minimizar o acoplamento do arcabouço em relação à implementação das características corporativas. Desta forma, possibilita-se que o desenvolvedor utilize apenas as características que considere necessárias para contemplar seus requisitos, tornando o arcabouço configurável e definindo cada característica corporativa como “opcional”.

Para isso, a arquitetura de suporte a aplicações corporativas foi definida utilizando conceitos de arquitetura orientada a aspectos e padrões de projeto orientado a objetos. A abordagem de aspectos mostrou-se adequada para o desenvolvimento das características de persistência, transação e segurança, não acarretando mudanças no projeto do núcleo do GCF.

Distribuição, suporte à Web e à integração com sistemas legados foram projetados utilizando padrões de projeto orientados a objetos, tais como *Decorator* [16] e *Proxy* [16]. Estes padrões também poderiam ter sido utilizados na definição das outras características, projetadas com aspectos. Porém, tanto o projeto quanto sua implementação mostraram-se mais complexos e menos flexíveis do que a solução com aspectos.

Neste capítulo, são descritos a arquitetura e o projeto de integração de cada uma destas características ao GCF. Exemplos e ilustrações contendo trechos de código são descritos neste capítulo utilizando a sintaxe das linguagens Java e AspectJ. Os diagramas são descritos utilizando UML (*Unified Modeling Language*) [184] e para a notação de aspectos nestes diagramas utiliza-se a extensão de UML proposta em [185].

## 6.1 Distribuição

No contexto de desenvolvimento baseado em componentes, a distribuição diz respeito à possibilidade de que componentes estejam distribuídos fisicamente, em diferentes nós de uma rede. O mecanismo de comunicação via rede, assim como a interação baseada em serviços ou eventos com provedores distribuídos, deve ficar transparente para o desenvolvedor do componente. Na visão do desenvolvedor, um componente distribuído deve ser acessado da mesma forma que um componente local.

### 6.1.1 Arquitetura

A versão distribuída de uma arquitetura de componentes baseada na CMS permite estruturas como a ilustrada na Figura 6.1. Tanto componentes, como contêineres e adaptadores podem estar distribuídos em nós diferentes, dependendo dos requisitos da aplicação.

Em algumas arquiteturas de componentes distribuídos, tais como J2EE e CCM, entidades similares aos contêineres da CMS implementam a distribuição. Sendo assim, cada contêiner ge-

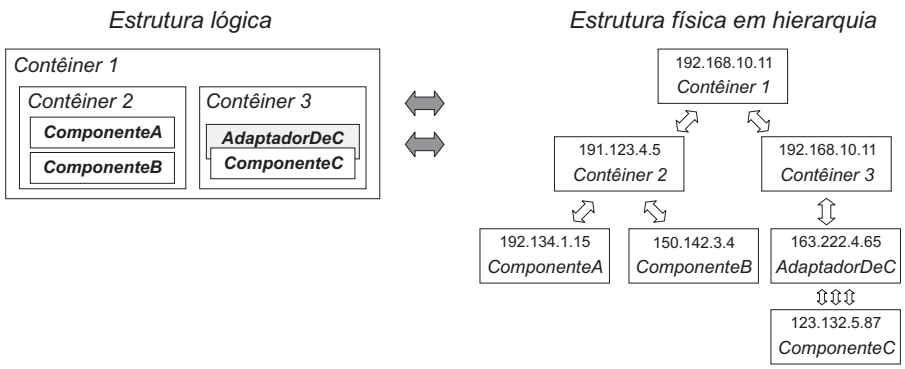



Figura 6.1: Exemplo de versão distribuída de uma arquitetura baseada na CMS.

rencia o acesso a seus componentes filhos distribuídos, ficando transparente ao desenvolvedor a localização dos provedores de serviços. Seguindo este mesmo modelo, o suporte à distribuição integrado ao GCF é gerenciado por extensões de contêineres, componentes funcionais e adaptadores, através da utilização do padrão *Proxy*.

Na extensão de contêineres para a arquitetura distribuída, as suas tabelas de serviços e eventos passam a ter referência a representantes das entidades filhas, os quais acessam o componente que provê o serviço ou está interessado em um evento, seja este local ou remoto. Da mesma forma, os componentes distribuídos possuem referência para um representante do seu contêiner pai. Este representante acessa o contêiner pai, seja ele local ou remoto.

Na Figura 6.2, ilustra-se uma arquitetura de componentes distribuídos. *Contêiner1* possui dois componentes filhos, sendo um deles, o componente *B*, localizado em outro endereço de rede. Para *Contêiner1*, o componente *B* é local. Porém, trata-se de um representante de *B*. Sendo assim, toda requisição encaminhada ao representante de *B*, é delegada via rede para o componente *B* real.

Da mesma forma, toda requisição que parte do componente *B* real é feita ao representante de *Contêiner1*. Para *B*, porém, trata-se de um contêiner local. O representante de *Contêiner1*, ao receber uma requisição, encaminha-a via rede para o *Contêiner1* real. A utilização de representantes torna distribuída uma arquitetura baseada na CMS, sem alterar os modelos de composição e interação.

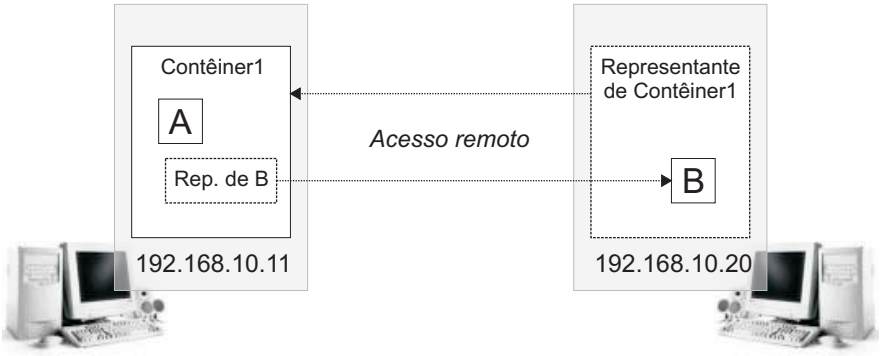



Figura 6.2: Representantes de componentes e contêineres.

### 6.1.2 Integração com o projeto do GCF

Na Figura 6.3, são ilustradas as principais classes referentes à integração do módulo de distribuição ao projeto do GCF. Estas classes são descritas a seguir.

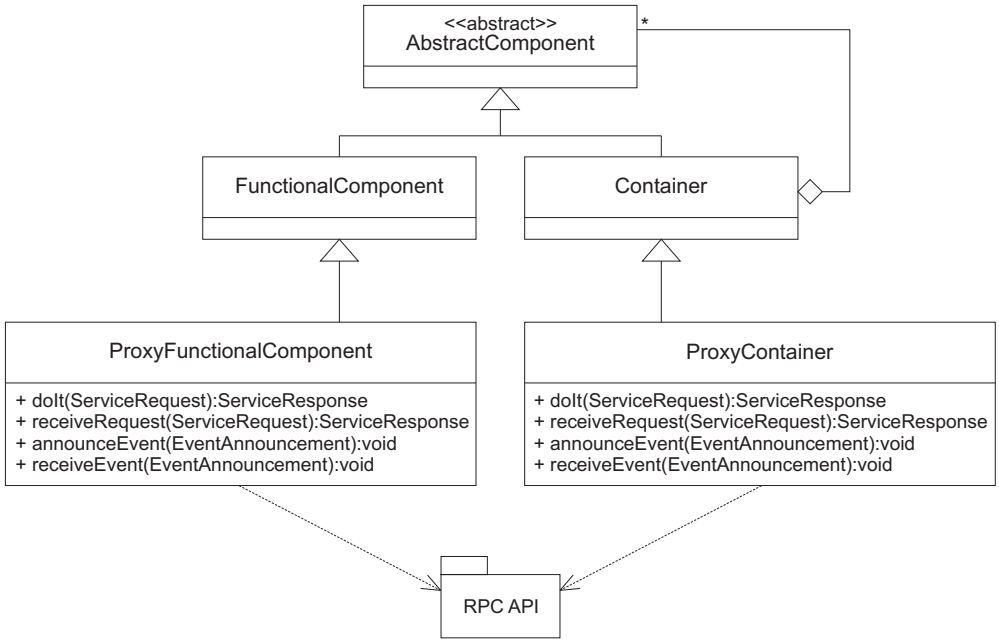



Figura 6.3: Integração do suporte de distribuição ao projeto do GCF.

- `ProxyFunctionalComponent` - Esta classe estende `FunctionalComponent`, sobrescrevendo os métodos de recepção e envio de serviços e eventos. Através de uma API de chamada de procedimento remoto, envia-se via rede a requisição para um representante remoto de um contêiner, ou seja, um `ProxyContainer`. Este contêiner encaminhará o serviço para o seu provedor seguindo o modelo de interação da CMS.

- `ProxyContainer` - Esta classe estende `Container`, sobrescrevendo os métodos de recepção e envio de serviços e eventos. Através da API de chamada de procedimento remoto, comunica-se com representantes remotos de contêineres (`ProxyContainer`) ou componentes funcionais (`ProxyFunctionalComponent`), dependendo da hierarquia de componentes da aplicação.

- RPC API - API para chamada de procedimento remoto (*Remote Procedure Call*) que permita aos representantes invocar métodos em objetos distribuídos.

### 6.1.3 Cenário de execução

A seguir são apresentados dois cenários de interação baseada em serviços entre componentes fisicamente distribuídos. No primeiro cenário, demonstra-se a visão interna, enfatizando o relacionamento entre as entidades do arcabouço. No segundo cenário, exemplifica-se o nível de distribuição possível com a arquitetura, apresentando uma visão externa e física da interação entre componentes. O mesmo raciocínio da interação baseada em serviços pode ser aplicado à interação baseada em eventos.

#### Visão interna

Para exemplificar a interação entre os componentes de forma distribuída, foram definidos três cenários principais: composição de componente distribuído; requisição de serviço implementado por componente distribuído; e requisição de serviço requerido por componente distribuído.

Como apresentado anteriormente, a composição de componentes locais é realizada em dois passos: criação de referência do componente via instanciação e inserção da referência em um contêiner através de um método da classe `Container`. Por exemplo, utilizando o arcabouço JCF: `parent.addComponent(new MyComponent())`, sendo `parent` o contêiner pai do componente.

Já na versão distribuída, a obtenção da referência e a adição no contêiner pai ocorre via registro. Por exemplo, no JCF: `ComponentRegistry.find("C", "193.165.5.12", parent)`, onde `parent` é a referência para o contêiner no qual o componente será inserido e para o qual será criado um representante na máquina onde se encontra o componente “C”, cujo endereço é “193.165.5.12”. O resultado deste processo é ilustrado na Figura 6.4.

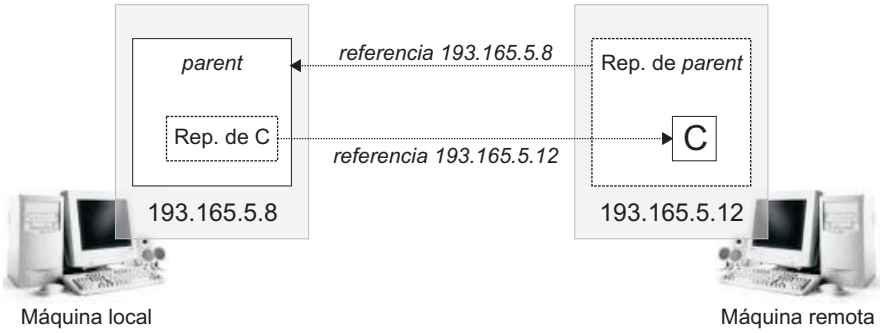



Figura 6.4: Composição de componentes distribuídos.

Uma vez estabelecida a arquitetura e definidas as referências remotas de contêineres e componentes funcionais, pode-se ilustrar os cenários de interação. Na Figura 6.5, ilustra-se um cenário em que um componente distribuído (*C*) implementa um serviço requisitado por outro componente.

De acordo com a CMS, a requisição do serviço por outro componente chegará ao componente *C* apenas através do seu contêiner pai, neste caso, *parent*. A seqüência de passos da requisição do serviço é ilustrada a seguir.

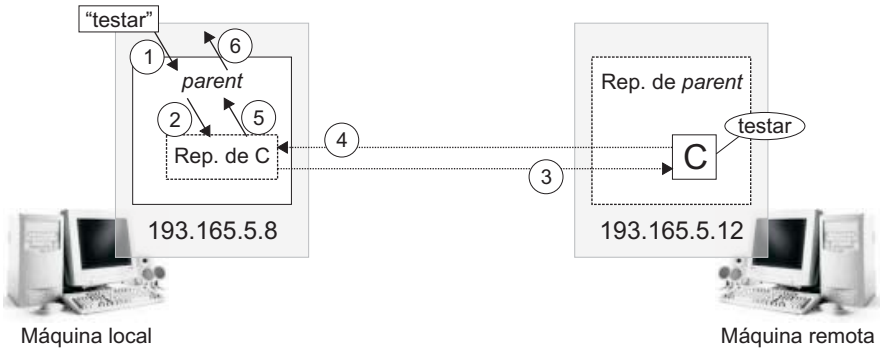



Figura 6.5: Cenário de interação: componente distribuído como provedor do serviço.

1. O contêiner *parent* recebe a requisição de execução do serviço "testar" que, de acordo com sua tabela de serviços, é provido por seu filho *C*.

2. A requisição é então encaminhada a *Rep. de C*, que funciona como representante de *C*, o que fica transparente ao contêiner *parent*.

3. *Rep. de C* requisita a execução do serviço ao componente distribuído *C*, que está localizado na máquina remota.

4. O componente *C* executa o serviço e retorna o resultado para o seu representante na máquina local - *Rep. de C*.

5. *Rep. de C* encaminha o resultado para o contêiner *parent*.

6. O contêiner *parent* então encaminha o resultado para o requisitante do serviço. Para *parent*, tratou-se de uma requisição local.

Na Figura 6.6, ilustra-se um cenário em que um componente distribuído (*C*) requisita um serviço implementado por outro componente. De acordo com a CMS, a requisição do serviço deve ser feita apenas ao contêiner pai de *C*, no caso, *parent*. A seqüência de passos da requisição do serviço é ilustrada a seguir.

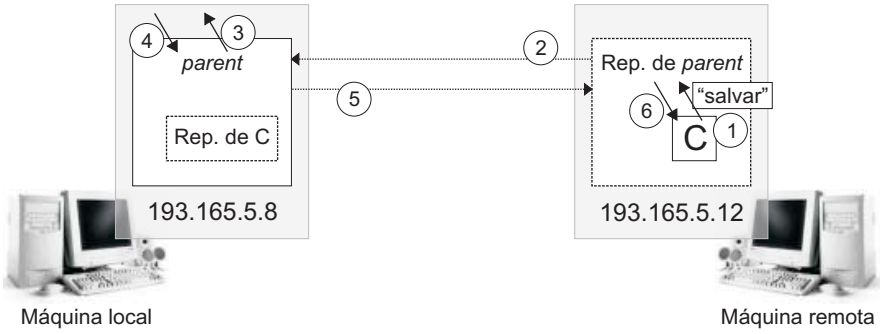



Figura 6.6: Cenário de interação: componente distribuído como requisitante do serviço.

1. O componente *C* requisita o serviço “salvar” ao seu contêiner pai, que na sua visão é *parent*. A requisição chega a *Rep. de parent*, que funciona como um representante de *parent*. Isto fica transparente ao componente *C*.

2. *Rep. de parent* requisita a execução do serviço ao contêiner real `parent`, que está localizado na máquina local (remota em relação a *C*).

3. O contêiner `parent` recebe a requisição e a encaminha ao seu pai - já que não possui outros componentes filhos que poderiam prover o serviço “salvar”. O provedor do serviço então será encontrado utilizando o mecanismo de interação definido na CMS .

4. O resultado da execução do serviço é retornado ao contêiner `parent`.

5. O contêiner `parent` então encaminha o resultado para o seu representante na máquina remota (local em relação a *C*).

6. Por fim, *Rep. de parent* encaminha o resultado para o componente *C*. Para *C*, tratou-se de uma requisição local.

**Visão externa**

Na Figura 6.7, ilustra-se um cenário de requisição do serviço “teste”, feita por *ComponenteA* de acordo com a arquitetura de distribuição, ou seja, com a utilização de representantes. Na Figura 6.7(a), ilustra-se a estrutura em árvore e na Figura 6.7(b) a estrutura física. A seqüência de passos durante a requisição do serviço é apresentada a seguir.

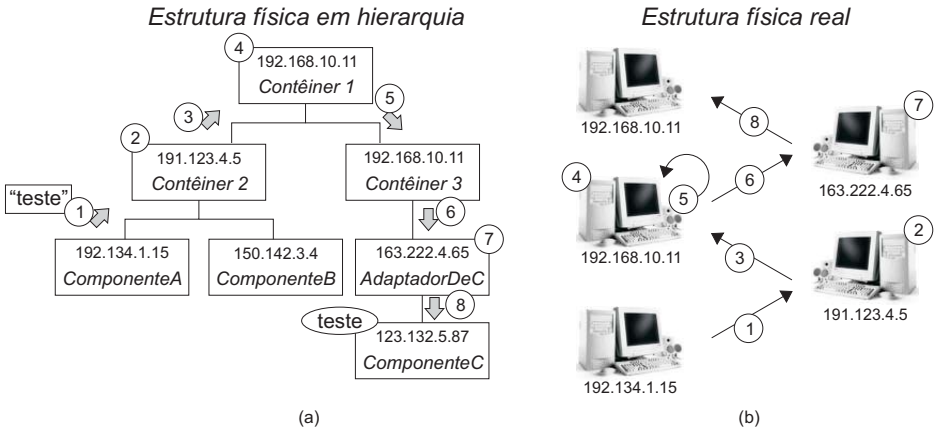



Figura 6.7: Invocação de serviço com representantes de componentes distribuídos.

1. *ComponenteA* requisita o serviço “teste” ao representante de seu contêiner pai (*Contêiner2*). O representante repassa a requisição para *Contêiner2* na máquina de endereço `191.123.4.5`.

2. *Contêiner2* recebe a requisição e verifica em sua tabela de serviços que nenhum dos seus filhos implementa o serviço “teste”.

3. *Contêiner2* então repassa a requisição para o representante de seu contêiner pai (*Contêiner1*). O representante encaminha a requisição para o *Contêiner1* na máquina de endereço `192.168.10.11`.

4. *Contêiner1* recebe a requisição e verifica em sua tabela de serviços que *Contêiner3* implementa o serviço “teste”.

5. *Contêiner1* então repassa a requisição para o representante do *Contêiner3*. O representante encaminha a requisição para o *Contêiner3* na mesma máquina.

6. *Contêiner3* recebe a requisição e verifica em sua tabela de serviços que *AdaptadorDeC* implementa o serviço “teste”, então repassa a requisição para o representante do adaptador. O representante encaminha a requisição para o *AdaptadorDeC* na máquina de endereço `163.222.4.5`.

7. *AdaptadorDeC* recebe a requisição e repassa ao representante do componente cujos serviços foram adaptados (*ComponenteC*).

8. O representante encaminha a requisição para *ComponenteC* na máquina referente ao endereço `163.222.4.5`. *ComponenteC* então executa o serviço.

## 6.2 Segurança

O suporte à segurança disponibilizado por um arcabouço baseado em componentes está diretamente relacionado à garantia de que apenas componentes autorizados podem ter acesso aos serviços e eventos disponibilizados por outros componentes. Ainda, este suporte deve ser transparente para o desenvolvedor do componente, que não conhece, *a priori*, em que ambientes seu componente será reutilizado. Um módulo de segurança inclui mecanismos de autenticação - garantia de que um componente é quem diz que é - e autorização - garantia de que um componente pode acessar um determinado serviço ou evento.

### 6.2.1 Arquitetura

Na CMS e no GCF, a identificação de serviços é feita através de nomes ou apelidos representados por cadeias de caracteres. Uma vez que esta identificação é única, ou seja, não podem existir dois serviços com o mesmo apelido, esta abordagem introduz um problema de segurança no arcabouço.

Por exemplo, considere o cenário ilustrado na Figura 6.8. Neste cenário, existe a possibilidade de interceptação de um serviço implementado pelo componente *K* e requisitado pelo componente *X*, fazendo com que o componente *Intruso* passe a receber as requisições que deveriam ir para *K*.

Para que isto ocorra, basta adicionar um componente que possua algum serviço com o mesmo apelido de algum outro serviço já disponibilizado. Isto pode representar uma maneira intrusiva de realizar operações não esperadas ou maliciosas no sistema, uma vez que o componente intruso será adicionado sem nenhum impacto na execução do sistema e dos outros componentes.

Se o serviço "debitar" tiver como parâmetros o nome do usuário e sua senha de *Internet Banking*, é possível descobrir o nome e senha dos clientes do banco apenas adicionando um componente com um serviço de mesmo apelido.

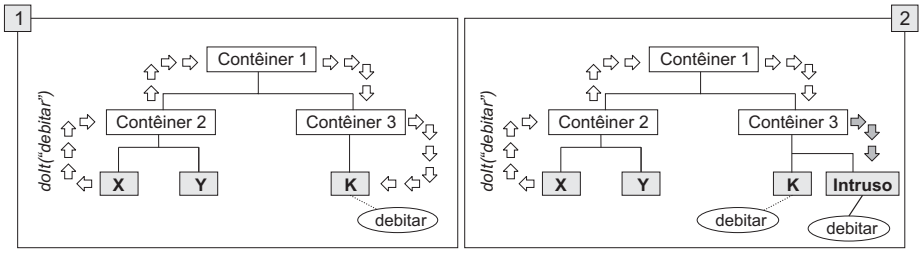



Figura 6.8: Problema de segurança em CMS/GCF.

A arquitetura de suporte à segurança foi projetada sem alterações na arquitetura da CMS e do GCF. Definiu-se um gerenciador de segurança que intercepta as requisições realizadas pelos componentes, adiciona o endereço do componente - para autorização - e uma senha - para autenticação. Quando a requisição chega ao componente provedor do serviço, o módulo de segurança retira o endereço e a senha da requisição, autentica e autoriza a execução do serviço e retorna o resultado. De acordo com esta arquitetura, ilustrada na Figura 6.9, os componentes não participam diretamente do processo de autenticação/autorização. Desta forma, impede-se que um componente não autorizado consiga interceptar requisições que deveriam ser encaminhadas a outros componentes.

### 6.2.2 Integração com o projeto do GCF

A interceptação das requisições dos componentes foi projetada utilizando conceitos de arquitetura orientada a aspectos [26]. Na Figura 6.10, são ilustradas as principais classes e o aspecto de

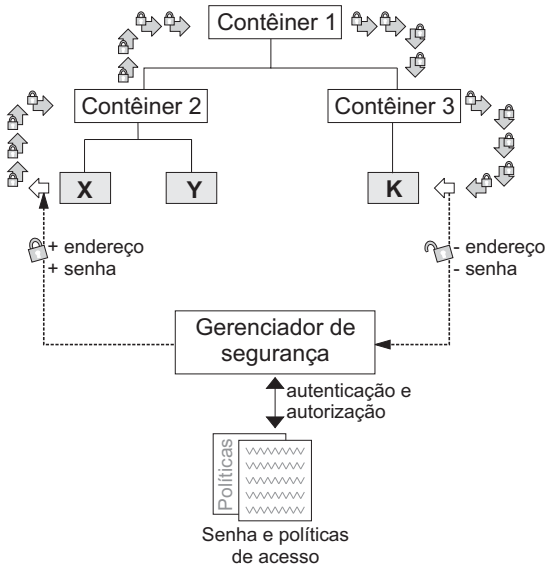



Figura 6.9: Arquitetura de suporte à segurança.

segurança implementados, as quais são descritas a seguir. A notação utilizada é uma extensão da linguagem UML proposta em [185].

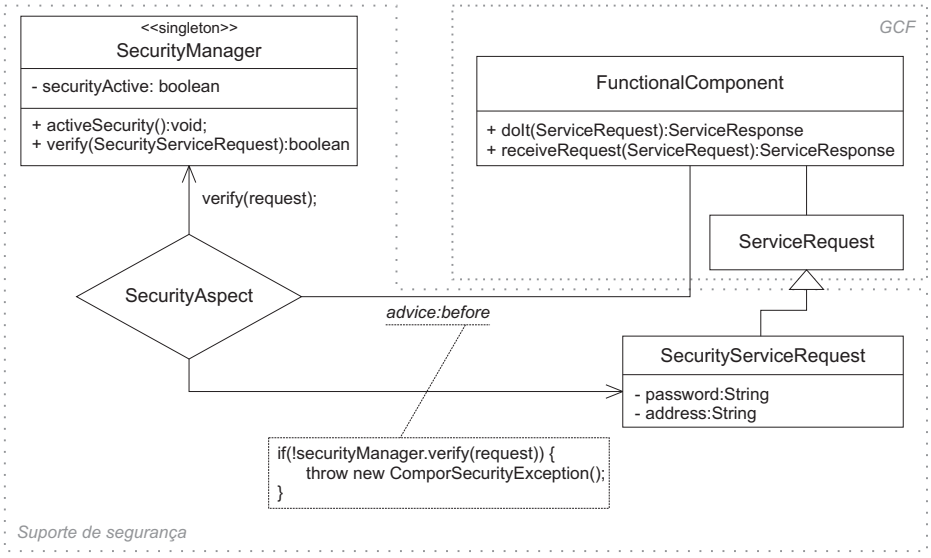



Figura 6.10: Integração do suporte de segurança ao projeto do GCF.

- `FunctionalComponent`: classe que implementa um componente funcional no GCF. Os métodos `doIt` e `receiveRequest` desta classe são interceptados por `SecurityAspect` antes de sua execução, utilizando o mecanismo `advice:before` [52]. Na interceptação de `doIt`, são adicionadas as informações para autorização e autenticação, alterando o formato

da requisição para SecurityServiceRequest. Na interceptação de receiveRequest, implementa-se o processo de autenticação e autorização, retornando a requisição para o seu formato original - ServiceRequest.

- SecurityManager: classe que implementa o gerenciador de segurança, responsável por gerenciar a senha e políticas de acesso e verificar se as requisições têm permissão de ser executadas. Além disso, é através do gerenciador que o mecanismo de segurança é habilitado e desabilitado, através do método activeSecurity().

- SecurityServiceRequest: requisição de serviço com suporte à segurança. Possui atributos password e address que são utilizados para a autenticação e a autorização pelo gerenciador de segurança.

- SecurityAspect: aspecto que implementa a interceptação da execução dos métodos doIt e receiveRequest da classe FunctionalComponent, utilizando advice:before. O aspecto requisita a autenticação e a autorização de execução do serviço através da verificação da requisição realizada pelo gerenciador de segurança (SecurityManager).

De acordo com o modelo apresentado na Figura 6.10, o projeto do arcabouço GCF, representado pelas classes FuncionalComponent e ServiceRequest, não foi alterado. Isto quer dizer que a utilização de segurança não é intrusiva em relação ao código do arcabouço, sendo opcional de acordo com as necessidades da aplicação. O projeto com aspectos torna simples a interceptação dos métodos de execução do GCF, permitindo o isolamento do código de segurança e mantendo a simplicidade da implementação do GCF.

### 6.2.3 Cenário de execução

Na Figura 6.11 ilustra-se a arquitetura referente ao módulo de segurança, cujos passos de funcionamento para a invocação de serviços são descritos a seguir. O mesmo funcionamento ocorre na interação baseada em eventos e disponibilização de novos componentes.

1. O desenvolvedor da aplicação cria um arquivo ".security" contendo a senha de acesso ao sistema, assim como um arquivo ".policies", contendo as políticas de acesso a serviços, e utiliza uma API de criptografia para criptografar o arquivo de senha.

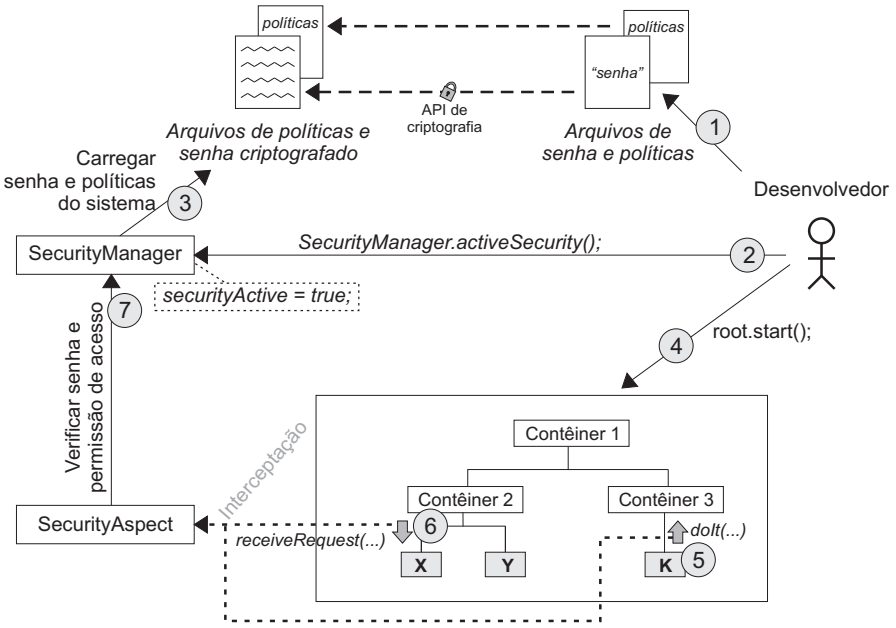



Figura 6.11: Cenário de execução do suporte à segurança.

2. No desenvolvimento da aplicação, a segurança deve ser ativada através da invocação do método `activeSecurity()` da classe singleton `SecurityManager`. Esta operação define que todas as invocações de serviços, disparos de eventos e adição de componentes serão verificadas.

3. Ao ativar a segurança, `SecurityManager` recupera a senha e o conjunto de políticas referentes ao sistema a partir dos arquivos definidos pelo desenvolvedor e os armazena em memória.

4. Ao iniciar o contêiner raiz da aplicação, todos os seus componentes serão iniciados e a aplicação inicia a sua execução através de uma seqüência de invocações de serviços e disparos de eventos.

5. Ao invocar um serviço, com a segurança habilitada, a requisição é interceptada no método `doIt` pelo aspecto `SecurityAspect` e uma requisição do tipo `SecurityServiceRequest`, contendo o endereço do componente e a senha de autenticação, é criada e encaminhada no lugar da requisição original.

6. Quando o componente recebe a requisição pelo método `receiveRequest`, esta é novamente interceptada pelo aspecto `SecurityAspect`, que a repassa para verificação por parte do

`SecurityManager`.

7. O `SecurityManager` verifica se a segurança está habilitada. Caso esteja, verifica a senha e as políticas de acesso definidas na requisição. Em caso de sucesso, o `SecurityAspect` libera a execução do método `receiveRequest` e, conseqüentemente, do serviço. Caso contrário, uma exceção `ComporSecurityException` é lançada.

## 6.3 Persistência

Aplicações corporativas em geral possuem a necessidade de persistir o estado no caso de interrupção da execução, principalmente o estado relacionado a suas entidades de negócio. O armazenamento da informação de estado de uma aplicação em algum tipo de repositório de dados é denominado persistência. Um suporte à persistência deve fornecer mecanismos para facilitar o trabalho do desenvolvedor para definir que informação será persistente, buscando ao máximo diminuir o esforço de configuração e codificação referente à esta característica.

### 6.3.1 Arquitetura

No contexto da CMS/GCF, que pode ser estentido ao contexto de aplicações baseadas em componentes de prateleira, os componentes da aplicação são caixa-preta e, portanto, gerenciam a persistência de seu estado internamente. Sendo assim, restam as entidades definidas do ponto de vista do desenvolvedor da aplicação.

Estas entidades são utilizadas como parâmetros de entrada para a execução de serviços e eventos dos componentes. Tais serviços e eventos, em alguns casos, podem alterar o estado das entidades de negócio que recebem como parâmetros e este estado precisa ser posteriormente persistido para que a alteração seja refletida em algum meio de armazenamento.

Partindo deste princípio, definiu-se uma arquitetura orientada a aspectos para persistir o estado de entidades de negócio após a execução de serviços e eventos. O aspecto de persistência - `PersistenceAspect` - intercepta os métodos de requisição de serviços e tratamento de eventos dos componentes, adicionando controle de transação, caso necessário, e implementando a persistência da entidade após a execução do serviço. Na Figura 6.12, ilustra-se a arquitetura proposta.

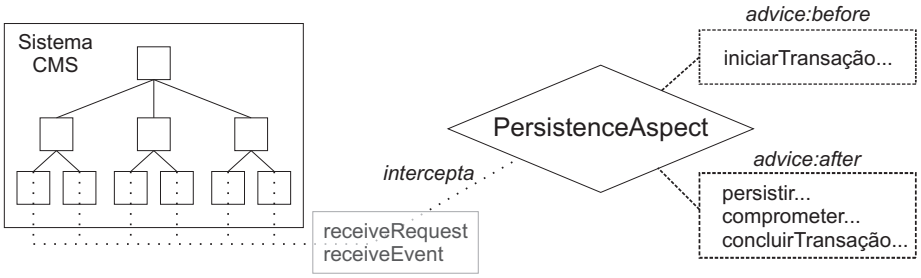



Figura 6.12: Arquitetura de suporte à persistência.

### 6.3.2 Integração com o projeto do GCF

A integração do suporte à persistência ao projeto do GCF foi realizada com base na utilização de aspectos e do padrão *AbstractFactory* [16]. Na Figura 6.13, ilustra-se o projeto de classes e aspectos de suporte à persistência integrado ao projeto do GCF. As entidades ilustradas na figura são descritas a seguir.

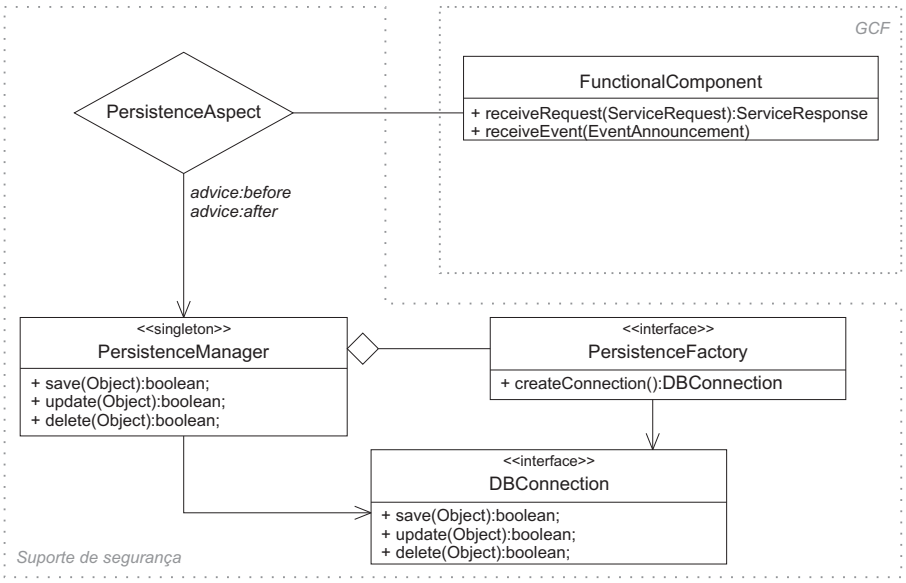



Figura 6.13: Integração do suporte à persistência ao projeto do GCF.

- `PersistenceAspect` - Este aspecto implementa a interceptação dos métodos de requisição de serviço (`receiveRequest`) e recebimento de notificação de eventos (`receiveEvent`) da classe `FunctionalComponent`. O aspecto utiliza mecanismos de `advice:before` e `advice:after`, respectivamente, para iniciar e concluir transações através do *singleton* `PersistenceManager`.

- `PersistenceManager` - Gerencia o mecanismo de persistência da aplicação. Ao ser iniciado, instancia uma implementação de `PersistenceFactory` especificada em um arquivo externo (XML), através da qual realiza conexões com o banco de dados. Além disso, carrega as regras de persistência, que definem quais entidades, em quais serviços, devem ser persistidas.

- `PersistenceFactory` - Representa uma fábrica abstrata tal qual definida no padrão *AbstractFactory*. Todas as classes que realizam conexão com diferentes gerenciadores de banco de dados devem implementar esta interface. De acordo com o padrão, isto permite um desacoplamento de `PersistenceManager` em relação ao tipo específico de banco utilizado.

- `DBConnection` - Representa um produto abstrato de acordo com o padrão *AbstractFactory*. Cada tipo de conexão com o banco de dados, de acordo com o gerenciador específico, implementará de forma diferente esta interface. Uma vez que `PersistenceManager` conhece apenas esta interface, ele não precisará ser alterado caso novos tipos de conexão sejam necessários.

### 6.3.3 Cenário de execução

Na Figura 6.14 ilustra-se um cenário de funcionamento do suporte à persistência. Os passos relacionados ao cenário são descritos a seguir.

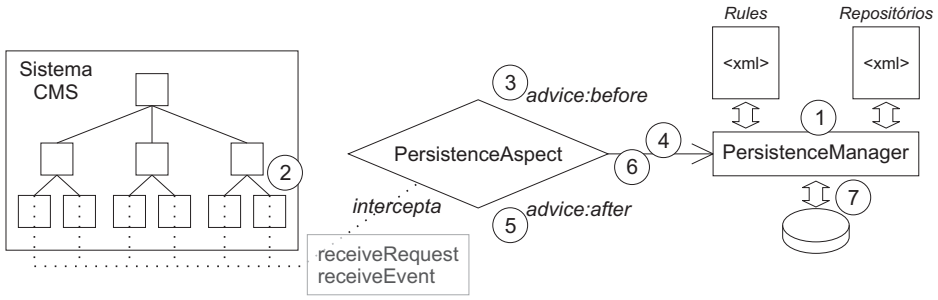



Figura 6.14: Cenário de execução do suporte à Web.

1. O gerenciador de persistência - `PersitenceManager` - ao ser iniciado carrega as informações de regras de persistência: serviços e eventos que requerem persistência; quais entidades

devem persistir; e se a transação é atômica ou não. Além disso, o gerenciador carrega a informação sobre qual fábrica utilizar para criar conexões com o banco. Isto permite que novas fábricas sejam criadas sem alteração no projeto e na implementação do suporte à Web.

2. Quando uma requisição de execução de serviço é recebida por um componente, o método `receiveRequest` é interceptado pelo aspecto `PersistenceAspect`.

3. Antes da execução do método, com o mecanismo de `advice:before`, o aspecto verifica junto ao gerenciador de persistência se o serviço exige persistência desta entidade e se a transação é atômica.

4. Em caso positivo, abre-se um contexto de transação para aquela entidade. Depois disto, o método é liberado para execução e o estado da entidade é alterado pela implementação do serviço.

5. Após a execução do método, o aspecto, via mecanismo de `advice:after`, intercepta o método antes do retorno.

6. O aspecto solicita então o comprometimento (*commit*) da operação ao gerenciador de persistência.

7. O gerenciador atualiza a informação no banco de dados, persistindo assim a alteração causada pelo serviço.

## 6.4 Transações

Além das características de transação implementadas pelo suporte à persistência de dados, outras características são necessárias em se tratando de aplicações baseadas em componentes de prateleira. Dentre elas, destaca-se o suporte à definição de *serviços transacionais*.

Uma vez que os componentes podem ser desenvolvidos por terceiros e utilizados em diferentes aplicações, algo que acontece com freqüência é a composição de serviços. Por exemplo, um serviço de transferência bancária - "transferir"- pode ser provido por um componente *K*, sendo a composição de dois outros serviços, "debitar" e "creditar", implementados pelos componentes

*X* e *Y*, respectivamente. O problema está em como garantir que uma transferência de dinheiro executada por *K* será feita corretamente, considerando que tanto *X* quanto *Y* podem falhar. Em termos de transação, o problema é como executar um ou mais serviços de forma atômica - ou todos são executados ou nenhum é.

Um suporte à transação deve implementar o controle das transações de modo transparente aos desenvolvedores dos componentes. Além disso, deve disponibilizar mecanismos para que desenvolvedores definam e implementem serviços transacionais.

### 6.4.1 Arquitetura

O mecanismo de gerência de transações proposto se baseia no protocolo de comprometimento em duas fases (*two-phased commit*). De acordo com este protocolo, tem-se um *coordenador*, que é responsável pela gerência das transações. Na primeira fase do protocolo, todos os componentes com serviços transacionais devem invocar um serviço de inicialização (*init*). De acordo com as respostas de todos os componentes, o coordenador decide na segunda fase se deverá executar *commit* ou *rollback*, enviando uma mensagem de decisão a todos os componentes.

Na Figura 6.15, ilustra-se a arquitetura baseada em aspectos de suporte a transações. O aspecto `TransactionAspect` implementa o coordenador, responsável por interceptar as chamadas ao método `receiveRequest` da classe `FunctionalComponent` e definir, de acordo com o estado da transação, que método deve ser invocado. Para isso o serviço deve ter sido definido como um serviço transacional através da implementação dos métodos `*Init`, `*Commit` e `*Rollback`, onde `*` representa o nome do serviço.

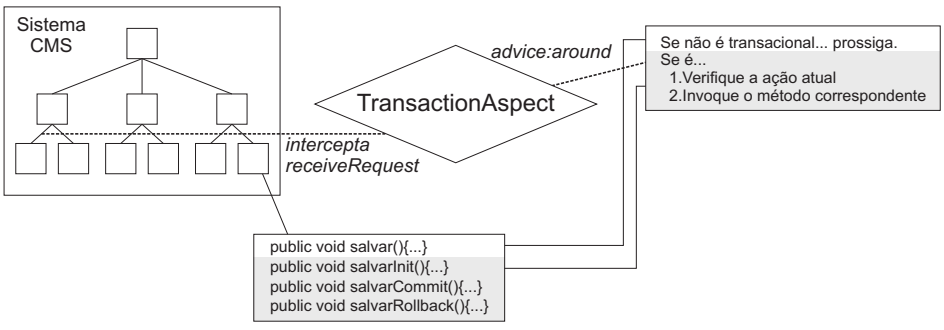



Figura 6.15: Arquitetura de suporte a transações.

### 6.4.2 Integração com o projeto do GCF

Na Figura 6.16, ilustra-se o projeto de classes de suporte a transações integrado ao projeto do GCF. As entidades ilustradas na figura são descritas a seguir.

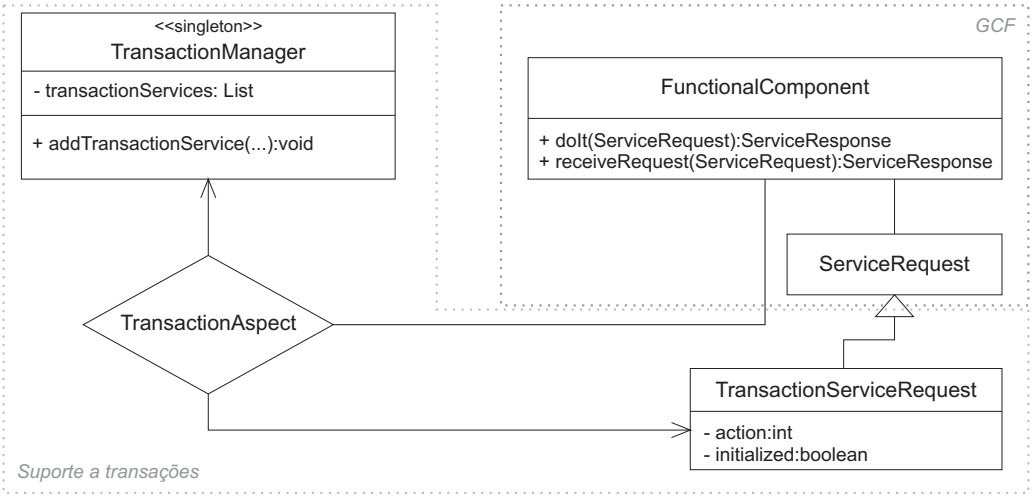



Figura 6.16: Integração do suporte a transações ao projeto do GCF.

- `TransactionManager` - Esta classe implementa o registro e gerência de execução de serviços transacionais. Toda a implementação do protocolo em suas duas fases é gerenciada por esta classe, incluindo a parte de persistência das informações sobre o estado da execução de cada serviço transacional para que o sistema se recupere em caso de parada abrupta do funcionamento.

- `TransactionAspect` - Este aspecto intercepta a execução do método `receiveRequest` da classe `FunctionalComponent` e verifica se tal requisição é uma instância da classe `TransactionServiceRequest`. Se o for, repassa a instância para que seja gerenciada pelo `TransactionManager` e, de acordo com o estado da execução, invocará o método correspondente de `FunctionalComponent` (`*init`, `*rollback`, `*commit`).

- `TransactionServiceRequest` - Estende a classe `ServiceRequest` para representar um serviço transacional. Possui atributos referentes ao estado atual da inicialização e última ação ocorrida.

### 6.4.3 Cenário de execução

Para o suporte a transação, tem-se três cenários principais a serem considerados: sucesso, erro e interrupção abrupta da aplicação. Para os três cenários considere o exemplo de transferência bancária descrito anteriormente.

No primeiro cenário, a transferência do dinheiro entre as contas é realizada com sucesso através de um serviço transacional que executa de forma atômica as operações de saque e depósito. Ao ser interceptado pelo aspecto `TransactionAspect`, tem-se a execução dos métodos `saqueInit` e `depositoInit` seguida com sucesso da execução de `saqueCommit` e `depositoCommit`.

No segundo cenário, a transferência foi iniciada para saque da conta de origem (`saqueInit`) mas houve um erro no depósito da conta de destino. Assim sendo, faz-se necessário que a transação seja desfeita. Isto é realizado através da invocação do serviço `saqueRollback`.

No terceiro cenário, tem-se um problema de interrupção abrupta da aplicação, por exemplo, por falha de hardware. No momento em que a operação de depósito ainda estava sendo realizada, ocorre uma interrupção inesperada no software por falha de hardware. Portanto, o saque na conta de origem foi realizado, porém o mesmo não ocorre para o depósito na conta de destino, gerando uma inconsistência de informações entre as contas bancárias. Com base na estrutura de persistência de informações de transações implementada pelo `TransactionManager` tem-se suporte a este cenário, possibilitando que transações não finalizadas possam ser desfeitas ao reiniciar o sistema utilizando o mesmo mecanismo de *rollback* descrito anteriormente.

## 6.5 Web

O número de aplicações corporativas baseadas na Web vem crescendo cada vez mais, desde o impacto causado pelo advento da tecnologia Java, que trouxe uma opção barata para o alto custo de atualização de módulos cliente do tipo *desktop*: o uso do navegador como cliente da aplicação. Além do advento de Java e de outras tecnologias utilizadas para programação para Web, o crescente aumento de pessoas que têm acesso irrestrito à Internet atualmente torna imprescindível a uma empresa ter suas aplicações acessíveis via rede mundial de computadores. No contexto do arcabouço GCF, dar suporte à Web significa disponibilizar mecanismos para o desenvolvimento de aplicações voltadas para a Web que utilizem como base a CMS, permitindo evolução dinâmica

não antecipada.

### 6.5.1 Arquitetura

A arquitetura de suporte à Web é ilustrada na Figura 6.17. Seguindo o padrão arquitetural MVC (Modelo-Visão-Controlador) [186], a aplicação GCF representa o *modelo* e as páginas Web representam a *visão*. Sendo assim, o foco do projeto de suporte à Web está no *controlador*, implementado no módulo Web. O *controlador* é responsável por processar requisições da visão e encaminhar ao modelo, assim como enviar às visões as notificações de mudanças de estado no modelo.

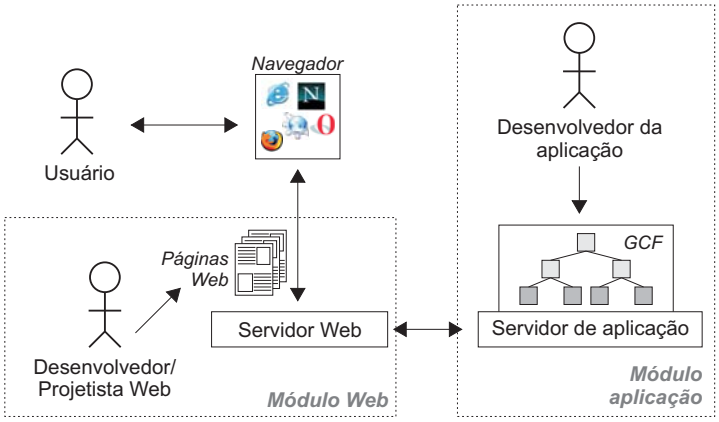



Figura 6.17: Arquitetura de suporte à Web.

Para minimizar o acoplamento entre a *visão* e o *controlador*, utilizou-se o conceito de *biblioteca de tags*. Trata-se de um conjunto de especificações baseadas em XML para a construção de aplicações Web com forte separação entre modelo, visão e controlador. Algumas das tecnologias que implementam o conceito de biblioteca de *tags* em várias linguagens são *Java Server Pages* (JSP) [187], *Java Server Faces* (JSF) [188] e *Python Server Pages* (PSP) [189].

Utilizando *tags* pré-definidas, o desenvolvedor Web pode requisitar a invocação de serviços sem precisar ter acesso ao código da aplicação GCF. Isto é possível através da implementação de processadores de *tags* que implementam a interpretação das *tags*, repassam as requisições para a aplicação GCF e exibem o resultado em local definido pelo desenvolvedor na página Web.

A sintaxe de invocação de serviço é ilustrada na Figura 6.18. A *tag* `<compor:invokeService>` especifica uma invocação de serviço, com nome definido no atributo `name` e cujo resultado será

armazenado na variável de nome “inseridoOk”, como definido no atributo `var`. Os parâmetros dos serviços são especificados através das *tags* `<compor:param>`.

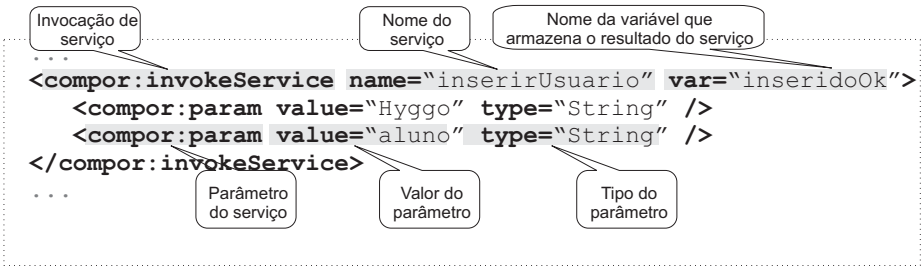



Figura 6.18: Invocação de serviço baseada em *tags*.

Uma vez que as *tags* seguem uma sintaxe XML, esta mesma sintaxe de invocação de serviço pode ser utilizada para qualquer implementação em qualquer linguagem. Porém, algumas alterações mais específicas podem ser necessárias dependendo da tecnologia de interpretação das *tags*.

Com base no conceito de biblioteca de *tags*, como ilustrado na Figura 6.19, a arquitetura interna do módulo Web possui duas entidades: um processador de *tags* e um formatador de resultados, que realizam a comunicação com a aplicação baseada no GCF. Estas entidades são descritas a seguir.

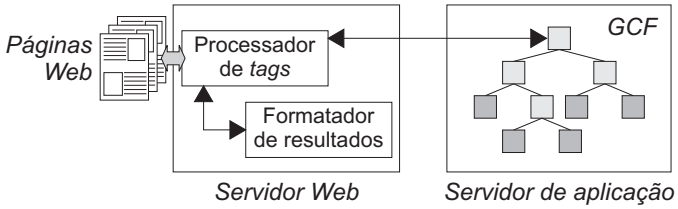



Figura 6.19: Arquitetura interna: módulo do servidor Web.

- *Processador de tags:* principal elemento do arcabouço, o processador de *tags* representa o ponto de integração entre a arquitetura Web e a aplicação GCF. O processador de *tags* é responsável por verificar as *tags* descritas pelo *Desenvolvedor Web* na página requisitada ao navegador, validá-las e, posteriormente, fazer as invocações necessárias à aplicação GCF, verificando resultados e exceções.

- *Aplicação GCF/CMS:* aplicação baseada em componentes de acordo com a CMS, desenvolvida utilizando alguma implementação do GCF.

- *Formatador de resultados:* formata os resultados obtidos da invocação de serviços à aplicação GCF. Este módulo é indispensável para a apresentação de resultados com tipos de dados mais complexos, tais como entidades do modelo de negócio da aplicação.

### 6.5.2 Integração com o projeto do GCF

Na Figura 6.20, ilustra-se o projeto de classes de suporte à Web integrado ao projeto do GCF. As entidades ilustradas na figura são descritas a seguir.

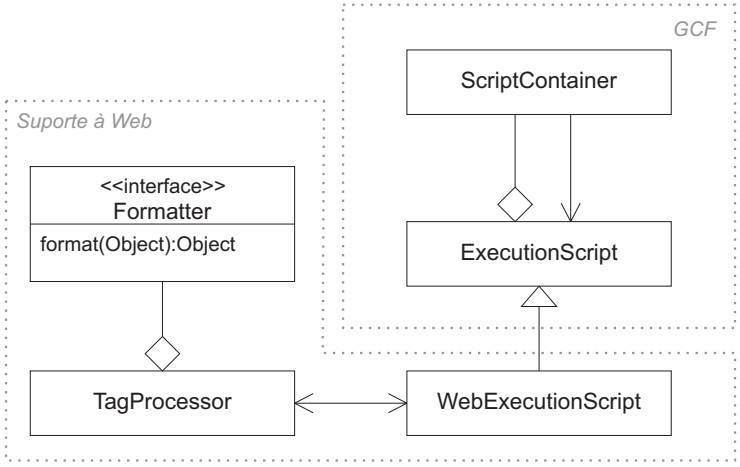



Figura 6.20: Integração do suporte à Web ao projeto do GCF.

- `WebExecutionScript` - Recebe requisições do processador de *tags* - `TagProcessor` - e as encaminha ao contêiner raiz da aplicação GCF. Da mesma forma, recebe eventos do contêiner raiz e os repassa ao `TagProcessor` na próxima requisição realizada. A independência entre os módulos Web e de aplicação implementada pelo `WebExecutionScript` garante que o suporte à evolução não antecipada continue sendo possível do lado da aplicação.

- `TagProcessor` - Interpreta as *tags* descritas nas páginas Web e repassa requisições para o `WebExecutionScript`. De acordo com a tecnologia utilizada, o processador pode ter um tipo específico, que já implemente a interpretação e o redirecionamento, como é o caso de JSF.

- `Formatter` - Formata o resultado das requisições de serviços e anúncios de eventos oriundos da aplicação GCF para serem exibidos nas páginas Web. A implementação do formatador depende da aplicação e o tipo de formatador utilizado é especificado nas *tags*. A

formatação padrão é de cadeia de caracteres através do método `toString()`, por exemplo, na versão em Java.

### 6.5.3 Cenário de execução

Na Figura 6.21 ilustra-se um cenário de funcionamento do suporte à Web a partir da exibição de uma página Web contendo *tags* de requisição de serviços. Os passos relacionados ao cenário são descritos a seguir.

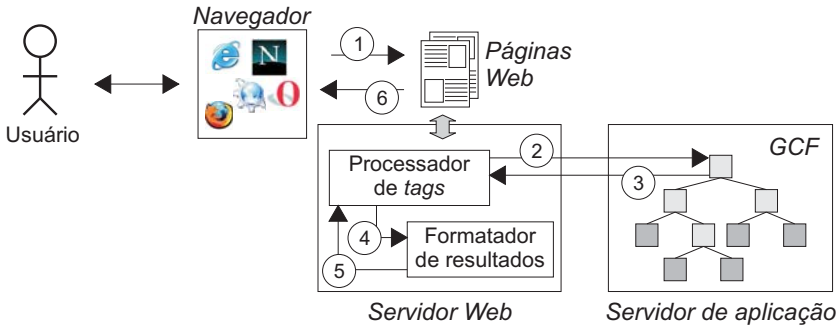



Figura 6.21: Cenário de execução do suporte à Web.

1. Cada *tag* de invocação de serviço da página Web é processada pelo *Processador de tags* que verifica o nome da aplicação baseada no GCF contendo os serviços a serem invocados. O *Processador de tags* prepara os parâmetros do serviço de acordo com o que foi definido em cada *tag* e encapsula todas as informações dentro de objetos do tipo `ServiceRequest`.

2. O *Processador de tags* solicita a execução do(s) serviço(s) ao contêiner raiz da aplicação GCF através de uma instância de `WebExecutionScript`.

3. Após a execução de cada serviço requisitado, a aplicação baseada em GCF retorna o resultado, encapsulado em um objeto do tipo `ServiceResponse`, ao *Processador de tags*.

4. O *Processador de tags* repassa o resultado para o *Formatador de resultados* que formata o resultado para exibição na página Web. Caso não seja indicado na *tag* nenhum formatador específico, a exibição do objeto será formatada como uma cadeia de caracteres.

5. Após a formatação, os resultados são encaminhados novamente para o *Processador de tags*.

6. Por fim, o navegador exibe ao usuário a página requisitada com os resultados da invocação de serviços. Para ele, trata-se de uma página Web qualquer.

## 6.6 Integração com sistemas legados

Em um ambiente corporativo, existe uma grande diversidade de sistemas desenvolvidos por diferentes empresas e que não foram projetados para serem integrados. Estes sistemas em geral possuem um código fechado e documentação pouco esclarecedora quanto ao acesso por parte de novos sistemas. Estes sistemas, denominados sistemas legados, e a necessidade de integração com os mesmos, representam complexidade para o desenvolvedor de novas aplicações corporativas. Um suporte à integração com sistemas legados deve oferecer mecanismos para integrá-los com novos sistemas, desenvolvidos com base na CMS, ainda garantindo a flexibilidade e a simplicidade provida pela especificação de composição.

### 6.6.1 Arquitetura

A arquitetura de integração com sistemas legados utiliza a própria infra-estrutura de componentes definida na CMS para criar um *Componente Legado* (CL), que implementa uma *Estratégia de Integração* (EI). O EI é implementado de acordo com a forma como é possível interagir com o sistema legado. Cada CL possui um EI específico, através do qual repassará as invocações de serviços para o sistema legado, gerenciando a execução e retorno do serviço.

Para o restante do sistema baseado na CMS, trata-se de um componente funcional comum, provendo serviços como qualquer outro. Isto torna a integração com o sistema legado extremamente flexível e independente do restante do sistema sendo desenvolvido. Na Figura 6.22, ilustra-se a arquitetura de integração.

### 6.6.2 Integração com o projeto do GCF

O projeto de classes para a integração com sistemas legados, ilustrado na Figura 6.23, baseia-se na extensão da classe `FunctionalComponent` definida no GCF e no padrão *Strategy* [16]. A classe `LegacyComponent` estende a funcionalidade da classe `FunctionalComponent`, sobrescrevendo o

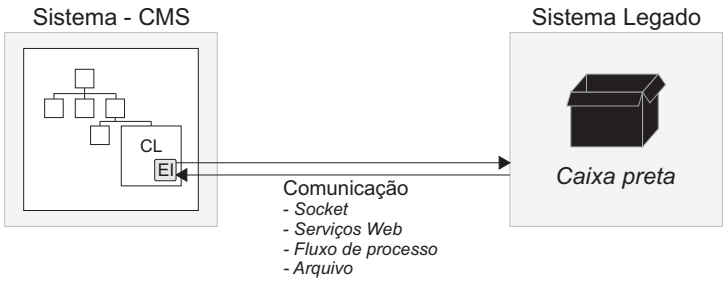



Figura 6.22: Arquitetura de suporte à integração com sistemas legados.

método `receiveRequest` de maneira que a requisição seja delegada para a chamada do método `request`, definido na interface `IntegrationStrategy`.

Classes com a funcionalidade de integração com sistemas legados implementam a interface `IntegrationStrategy`. Isto torna possível que, de acordo com a forma de integração, o desenvolvedor possa criar sua própria estratégia de integração e relacioná-la ao `LegacyComponent` sendo desenvolvido. Exemplos de implementações da interface `LegacyComponent` são descritas abaixo.

- `WebServicesStrategy`: implementa a comunicação através de serviços Web (*Web Services*) [49]. Isto requer que seja fornecido um conjunto de serviços descritos em WSDL (*Web Services Description Language*), implementados em linguagem compatível com a do sistema legado. A estratégia implementa o estabelecimento da conexão e as requisições de acesso aos serviços.

- `SocketStrategy`: implementa a comunicação via *sockets*. Para isso é necessário que o sistema legado receba requisições via rede através de *sockets*. Como o conteúdo da mensagem é enviado e recebido em *bytes*, torna-se possível estabelecer a comunicação com o sistema legado mesmo utilizando uma linguagem de programação diferente.

- `FileStrategy`: implementa a comunicação através de sistema de arquivos. Os arquivos são utilizados como meio compartilhado de informação entre o sistema sendo desenvolvido e o sistema legado. O processo é semelhante ao acesso compartilhado de banco de dados, que também é uma forma de integração entre os sistemas.

- `ProcessStreamStrategy`: implementa a comunicação através da escrita e leitura de *bytes* nos fluxos de entrada e saída do processo de execução do sistema legado. Alguns siste-

mas legados possuem mecanismos para a execução via linha de comando. Estes sistemas, em geral, possibilitam que serviços internos do sistema possam ser executados de forma personalizada através da parametrização da execução do processo via linha de comando. Nestes casos, é possível criar um fluxo de comunicação entre os processos do sistema sendo desenvolvido e do sistema legado e, por este canal, realizar as requisições.

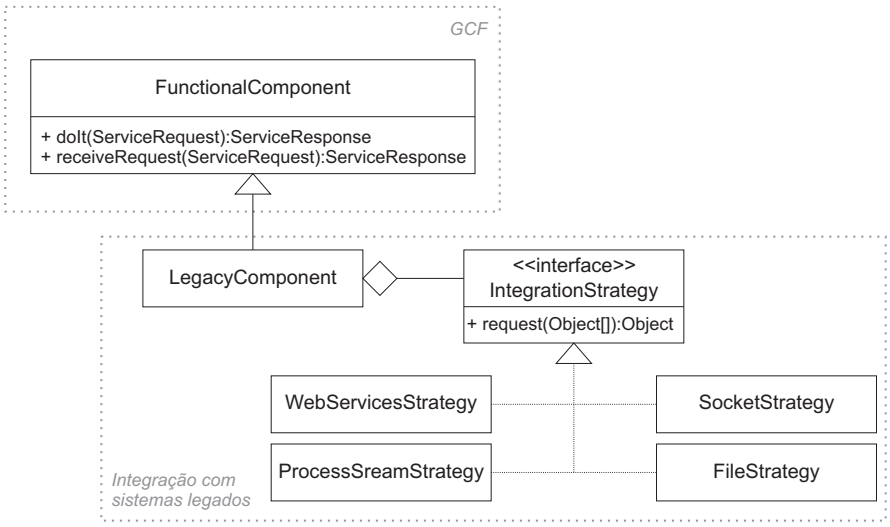



Figura 6.23: Integração do suporte a legados ao projeto do GCF.

### 6.6.3 Cenário de execução

Na Figura 6.24, ilustra-se um cenário de invocação de serviço provido por um componente legado que delega a execução a um sistema legado. Os passos relacionados ao cenário são descritos a seguir.

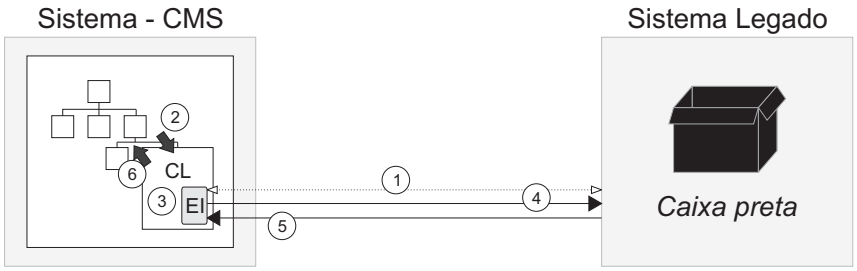



Figura 6.24: Cenário de execução do suporte à integração com sistemas legados.

1. Ao ser iniciado, o componente legado estabelece conexão com o sistema legado, de acordo com o tipo de estratégia adotada para a comunicação com o mesmo. Após este passo, o sistema legado estará pronto para receber requisições.

2. O contêiner pai do componente legado repassa para ele uma requisição de serviço.

3. O componente legado requisita à estratégia de integração que realize a requisição referente ao serviço junto ao sistema legado.

4. A estratégia de integração envia os dados da requisição para o sistema legado.

5. O sistema legado processa a requisição e retorna o resultado.

6. A estratégia repassa o resultado para o componente legado que encapsula este resultado em uma `ServiceResponse` e a retorna para o contêiner pai.

## 6.7 Outras características

Outras características que também são importantes no contexto de sistemas corporativos, mas que ainda não foram incorporadas ao GCF, são discutidas a seguir, enfatizando como poderiam ser contempladas no arcabouço de maneira flexível e se tornarem opcionais.

### 6.7.1 Balanceamento de carga

O balanceamento de carga se refere ao suporte para redirecionamento de requisições de clientes para os servidores com menos carga de processamento. Se um servidor estiver sobrecarregado, as requisições deveriam ser encaminhadas a outro servidor.

No caso de um sistema baseado em componentes distribuídos, o balanceamento de carga pode ser realizado no nível de serviços de componentes. Para isso, é necessário ter várias cópias de um mesmo componente, em máquinas diferentes. Desta forma, pode-se utilizar algoritmos de balanceamento de carga e escalonamento de requisições para definir para qual componente uma dada requisição deveria ser encaminhada.

Para integrar um suporte de balanceamento de carga ao GCF sem alterar sua estrutura base, em primeiro lugar, é necessário inserir vários componentes que provêem os mesmos serviços em um

contêiner da aplicação. Evidentemente, os apelidos dos serviços devem ser diferentes pois, caso sejam iguais, de acordo com a CMS apenas um único componente receberá todas as requisições.

Após construir a infra-estrutura de componentes distribuídos, é necessário implementar algoritmos de balanceamento. Isto pode ser realizado utilizando um aspecto - `LoadBalancingAspect` - específico para balanceamento de carga, que intercepte o método de redirecionamento de serviços dos contêineres para os componentes - `receiveRequest`. Neste momento, define-se o provedor mais adequado para a execução do serviço. Para este caso, o mecanismo `advice:around` é mais adequado pois fornece um controle maior sobre a execução do método `receiveRequest`.

Na Figura 6.25, ilustra-se uma possível arquitetura para balanceamento de carga integrada ao GCF.

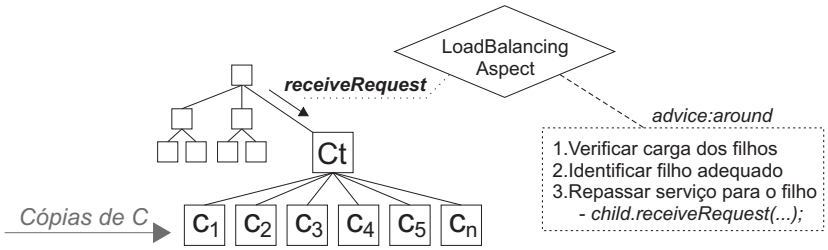



Figura 6.25: Integração do suporte a balanceamento de carga ao GCF.

### 6.7.2 Registro e auditoria

A funcionalidade de registro de eventos ocorridos no sistema (*Logging*) é indispensável para identificar a causa de potenciais problemas na execução do sistema. Caso este registro seja realizado de uma forma que permita, a partir dele, realizar uma auditoria na execução do software e facilitar a busca por um erro no sistema, a funcionalidade de registro é ainda mais útil.

Registro de eventos é uma das funcionalidades mais utilizadas para exemplificar a utilidade de arquiteturas orientadas a aspectos. Portanto, torna-se conveniente projetar a integração do suporte a registro de eventos ao arcabouço GCF usando aspectos.

Dependendo do nível de registro a ser realizado, mais ou menos métodos podem ser interceptados pelo aspecto de registro - `LogAspect`. As principais classes a serem relacionadas com o aspecto são `AbstractComponent`, `Container`, `FunctionalComponent` e `Adapter`. Os mecanismos `advice:before` e `advice:around` são utilizados para registrar, respectivamente, o início

e o término de operações, tais como: inicialização e interrupção de componentes, adaptadores e contêineres; invocação e execução de serviços; anúncio e recebimento de eventos; exceções de execução; dentre muitos outros tipos de eventos.

As informações podem ser armazenadas em arquivo texto, seguindo algum formato estruturado, tal como XML (*eXtensible Markup Language*). Isto permite a construção de uma ferramenta de leitura que, por exemplo, a partir de um arquivo XML, constrói diagramas de seqüência de mensagens para facilitar a descoberta da causa raiz de um erro. Na Figura 6.26, ilustra-se uma possível arquitetura de registro de eventos e auditoria integrada ao GCF. A utilização de aspectos garante flexibilidade ao projeto do arcabouço e mantém a sua estrutura original.

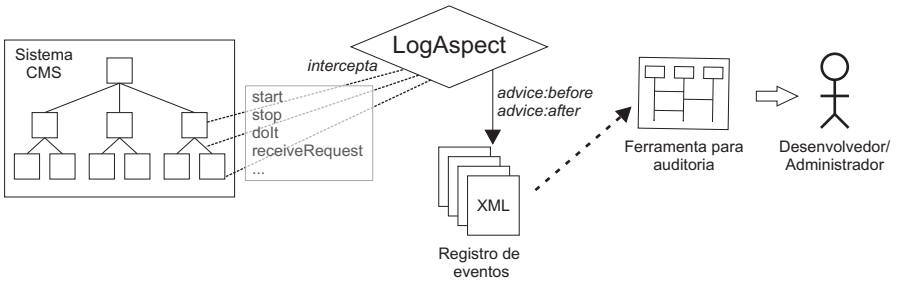



Figura 6.26: Integração do suporte a registro de eventos ao GCF.

### 6.7.3 Gerenciamento e monitoramento

O gerenciamento e o monitoramento do sistema estão relacionados à identificação e visualização do estado do sistema em um dado momento, aliados à possibilidade de tomada de decisão do próprio sistema em caso de falhas no seu funcionamento.

Dado o contexto de desenvolvimento baseado em componentes de prateleira, o comportamento interno do componente desenvolvido por terceiros pode causar problemas na aplicação como um todo. Uma vez que não se tem *a priori* um mecanismo de monitoramento interno do componente - apenas da interface - o módulo de gerenciamento do sistema não pode indicar que houve uma falha interna a menos que seja explicitamente estipulado pelo desenvolvedor da aplicação. Por exemplo, pode-se definir que sempre que um serviço da aplicação não for encontrado - o que pode representar uma falha no acesso a um componente distribuído - o contêiner raiz do sistema seja interrompido.

Uma arquitetura de monitoramento pode ser projetada seguindo o mesmo modelo de registro

de eventos apresentado anteriormente. Porém, além de apenas registrar algo ocorrido, algumas funcionalidades pró-ativas podem ser providas: notificação do administrador; reinicialização de componentes, contêineres e adaptadores; recuperação de estado do sistema utilizando o suporte a transações; dentre outras funcionalidades.

Na Figura 6.27, ilustra-se uma possível arquitetura de monitoramento integrada ao GCF. Assim como na arquitetura de registro de eventos, a utilização de aspectos garante flexibilidade e mantém a estrutura do arcabouço. Nesta arquitetura, o aspecto `MonitoringAspect` intercepta os métodos das principais classes do arcabouço e, com base em regras do tipo (evento - ação), determina qual ação a ser tomada dado um evento ocorrido.

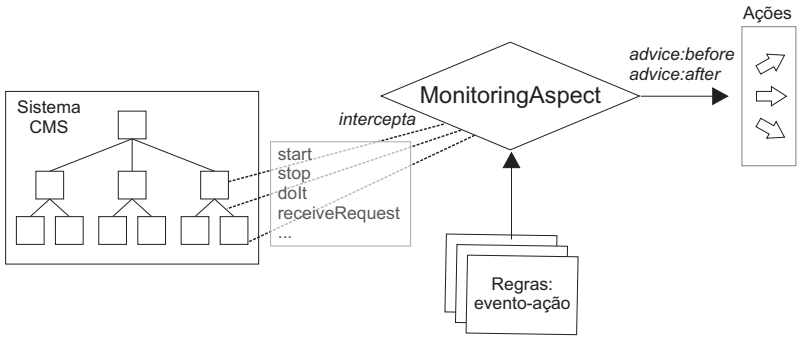



Figura 6.27: Integração do suporte a monitoramento ao GCF.

## 6.8 Considerações Finais

Neste capítulo foi apresentado o suporte para aplicações corporativas acoplado ao projeto do GCF. A grande motivação para a proposição deste suporte é viabilizar a utilização da CMS no desenvolvimento de aplicações com características de distribuição, persistência, transação, dentre outras. Dada a carência de soluções de desenvolvimento de aplicações corporativas com suporte à evolução dinâmica não antecipada, a CMS se apresenta como uma abordagem promissora também nessa área.

O suporte a aplicações corporativas foi acoplado ao projeto do GCF de forma que cada requisito possa ser utilizado sob demanda. Sendo assim, é possível combinar características diferentes de acordo com a necessidade da aplicação, tornando mais simples o entendimento do projeto. Isto foi obtido através da utilização de padrões de projeto e programação orientada a aspectos para

evitar alterações no projeto original do GCF.

Dentre as características abordadas, estão incluídas distribuição, transação, persistência, integração a sistemas legados e suporte à Web. Foram apresentados a arquitetura, projeto de classes e cenário de funcionamento de cada uma delas, servindo como base para implementações em linguagens específicas. Os projetos de outras características, tais como registro, auditoria e balanceamento de carga, também são resumidamente discutidas, apenas como forma de demonstrar que é possível contemplá-las de maneira não intrusiva ao projeto do GCF.

# Capítulo 7

# Técnica para Especificação e Verificação Formal Utilizando Alloy

Neste capítulo apresenta-se uma técnica baseada no método formal Alloy para a especificação e análise de software baseado na CMS. Com esta técnica é possível verificar se um dado cenário de evolução pode ir de encontro à corretude da especificação da aplicação. Descreve-se um arcabouço Alloy para a especificação das propriedades da aplicação e dos componentes e um conjunto de diretrizes para a análise e a verificação do software utilizando tal arcabouço.

Como já mencionado no Capítulo 2, há duas justificativas para a utilização de Alloy. A primeira é o fato da linguagem ter características similares à abordagem orientada a objetos, a qual já é familiar aos desenvolvedores de software. A segunda é o suporte da ferramenta *Alloy Analyzer*, que é leve, de fácil extensão e de código aberto, o que permite a integração ao ambiente de desenvolvimento apresentado no Capítulo 9. Além disso, a descrição da especificação de um sistema é textual, facilitando a composição da especificação e das propriedades a serem verificadas, que é a base da técnica aqui proposta.

## 7.1 Arcabouço Baseado em Alloy

Para que sejam especificados e analisados os componentes e as aplicações baseadas na CMS, é necessário um arcabouço para padronizar tais especificações. Diferentemente dos demais arcabouços apresentados nesta tese, o arcabouço baseado em Alloy não implementa a CMS. O intuito

aqui não é verificar se os modelos de interação e disponibilização da CMS estão corretos e sim, verificar se um dado cenário de evolução, ou seja, uma mudança na especificação inicial vai de encontro à corretude do sistema. Em [115], pode-se obter detalhes sobre um trabalho anterior sobre a verificação formal da CMS.

O mais importante para verificar o impacto da evolução sobre a especificação existente é a relação de dependência entre os componentes, representada pelas dependências de serviços e eventos. Uma vez que, de acordo com a CMS, a arquitetura da aplicação não influencia na funcionalidade da mesma, considera-se para a especificação e a verificação formal apenas os seus componentes funcionais. Assim como na CMS, componentes possuem serviços providos e requeridos e eventos de interesse e anunciados. A descrição desta composição em Alloy é apresentada na Listagem de Código 7.1.

Listagem de Código 7.1: Definição de servico/evento e componente funcional.

```
1  module cms
2
3  abstract sig Service{}
4  abstract sig Event{}
5
6  abstract sig FunctionalComponent{
7          providedServices: set Service,
8          requiredServices: set Service,
9          announcedEvents: set Event,
10         eventsOfInterest: set Event
11 }
```

Na linha 1 da Listagem de Código 7.1, cria-se um módulo denominado `cms`, o qual servirá como base para a descrição de modelos de aplicação e componentes. Nas linhas 3 e 4, são especificadas as entidades `Service` e `Event`, como abstratas, as quais serão estendidas na modelagem de componentes específicos. Por fim, define-se `FunctionalComponent` na linha 6, com campos para serviços providos e requeridos e eventos de interesse.

As dependências de serviços e eventos são definidas como descrito na Listagem de Código 7.2. Tem-se a definição de dependência de serviços – `ServiceDependence` – e os campos para a relação de dependência entre os componentes (`components`) e o serviço que causa a dependência (`service`). Além disso, algumas restrições são adicionadas nas linhas 5 a 8, referentes à unicidade da relação de dependência, restrição a relações cíclicas e exigência de dependência completa, respectivamente.

Listagem de Código 7.2: Definição de dependência de serviços.

```
1 abstract sig ServiceDependence{
2     components: FunctionalComponent -> FunctionalComponent,
3     service: one Service
4 }{
5     one components
6     no (components&iden)
7     service in (FunctionalComponent.~components).requiredServices
8     service in (FunctionalComponent.components).providedServices
9 }
```

Seguindo o mesmo raciocínio da definição de dependência de serviços, tem-se na Listagem de Código 7.3 a definição da dependência de eventos – `EventDependence`. Observe que nas linhas 5 a 8 são definidas as mesmas restrições da dependência de serviços.

Listagem de Código 7.3: Definição de dependência de eventos.

```
1 abstract sig EventDependence{
2     components: FunctionalComponent -> FunctionalComponent,
3     event: one Event
4 }{
5     one components
6     no (components&iden)
7     event in (FunctionalComponent.~components).eventsOfInterest
8     event in (FunctionalComponent.components).announcedEvents
9 }
```

Por fim, na Listagem de Código 7.4, tem-se a definição de uma aplicação – `Application`. Uma aplicação contém campos para um conjunto de componentes funcionais, dependências de serviços e de eventos.

Listagem de Código 7.4: Definição de aplicação.

```
1 abstract sig Application{
2     components: set FunctionalComponent,
3     serviceDependences: set ServiceDependence,
4     eventDependences: set EventDependence
5 }
```

Uma vez definido o esqueleto de código Alloy para a especificação de aplicações, componentes e dependências de serviços e eventos, deve-se agora definir as propriedades específicas dos componentes de uma aplicação. Para isso, as assinaturas abstratas definidas anteriormente serão estendidas e novos campos serão adicionados de acordo com o componente ou aplicação sendo

especificado. A composição destas especificações é então verificada usando o analisador Alloy. Cada um destes passos é descrito a seguir.

## 7.2 Técnica para Especificação e Verificação Formal

Na Figura 7.1, são ilustrados os passos referentes à técnica de especificação e verificação formal de sistemas baseados na CMS utilizando o arcabouço descrito anteriormente. Cada um dos passos é então descrito a seguir. Esta técnica se insere como parte do processo de desenvolvimento apresentado no Capítulo 11. Para facilitar o entendimento dos passos descritos a seguir, cada passo será exemplificado no contexto de uma aplicação de transferência bancária.

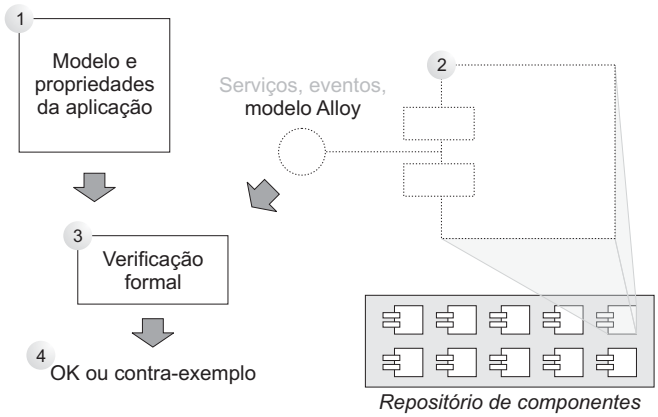



Figura 7.1: Técnica para análise e verificação formal.

### 1. Especificando a aplicação em seu ciclo de desenvolvimento

Ao desenvolver a aplicação, o desenvolvedor define um modelo Alloy para o sistema com base nos serviços e eventos a serem providos pelos seus componentes e um conjunto de assertivas que devem ser verificadas. Não é necessário especificar o sistema como um todo, pode-se apenas modelar o que se considerar necessário e mais crítico. Na Listagem de Código 7.5 descreve-se o início da definição do módulo `banking_app`, com a definição de conta, número de transferência, serviços de saque, depósito e transferência e dependências de serviços entre componentes. Por fim, define-se a aplicação propriamente dita. Tem-se também uma assertiva simples para verificação da igualdade no número de saques e depósitos ocorridos em transferências.

Listagem de Código 7.5: Definição da aplicação de transferência bancária.

```
1  //Definição do módulo
2  module banking_app
3
4  //Importando definições do modelo da CMS
5  open cms
6
7  //Criando assinaturas para contas e números de transferências
8  sig Account{}
9  sig TransferNumber{}
10
11 //Criando assinaturas para serviços como extensões de Service
12 one sig sacar extends Service{}
13 one sig depositar extends Service{}
14 one sig transferir extends Service{}
15
16 //Criando assinatura para dependência entre Transferidor e Sacador
17 one sig TransferenciaSaque extends ServiceDependence{
18         number:TransferNumber,
19         account:Account
20 }{
21         components = Transferidor->Sacador
22         service=sacar
23 }
24
25 //Criando assinatura para dependência entre Transferidor e Depositante
26 one sig TransferenciaDeposito extends ServiceDependence{
27         number:TransferNumber,
28         account:Account
29 }{
30         components = Transferidor->Depositante
31         service=depositar
32 }
33
34 //Criando assinatura para a aplicação
35 one sig BankingApp extends Application{}{
36         components = Depositante + Sacador + Transferidor
37         serviceDependences = TransferenciaSaque + TransferenciaDeposito
38 }
39
40 //Assertiva para transferência indiciando a igualdade de saques e depósitos em transferências
41 assert transferOk{
42     #TransferenciaSaque=#TransferenciaDeposito
43 }
```

## 2. Adicionando componentes

A cada componente adicionado, pode-se verificar se o modelo do componente, mesclado ao modelo da aplicação, ainda mantém a corretude das propriedades iniciais. Observe que este cenário pode ocorrer tanto no momento da evolução, como para o auxílio ao desenvolvimento. Na Listagem de Código 7.6 descreve-se a definição dos componentes da aplicação de transferência bancária. Observe que as especificações dos componentes já trazem assertivas que podem ir de encontro à especificação da aplicação e vice-versa.

Listagem de Código 7.6: Definição dos componentes de transferência bancária.

```
1  //Criando assinatura para o componente sacador
2  one sig Sacador extends FunctionalComponent{
3  }{
4          providedServices = sacar
5          no requiredServices
6          no eventsOfInterest
7          no announcedEvents
8  }
9
10 //Criando assinatura para o componente depositante
11 one sig Depositante extends FunctionalComponent{}{
12         providedServices = depositar
13         no requiredServices
14         no eventsOfInterest
15         no announcedEvents
16 }
17
18 //Criando assinatura para o componente transferidor
19 one sig Transferidor extends FunctionalComponent{}{
20         providedServices = transferir
21         requiredServices = sacar+depositar
22         no eventsOfInterest
23         no announcedEvents
24 }
25
26 //Assertiva para transferência indiciando a desigualdade de saques e depósitos em transferências
27 assert transferOk2{
28         #TransferenciaSaque!=#TransferenciaDeposito
29 }
```

## 3. Reunindo as especificações e executando a verificação

O *Alloy Analyzer* verifica a corretude das propriedades de acordo com todos os modelos, mesclados em um modelo único. Além disso, deve-se definir quais assertivas devem ser verificadas. Isto é realizado através da execução do comando `check` para cada uma das assertivas, como descrito na Listagem de Código 7.7. Considerando componentes caixa-preta, nos quais a especificação foi definida sem o contexto de uma aplicação específica em mente, será provavelmente necessário realizar adaptações na especificação para compatibilizar as definições das assinaturas e campos.

Listagem de Código 7.7: Código para execução da verificação das assertivas.

```
1 check transferOk
2 check transferOk2
```

## 4. Analisando resultados da verificação

O resultado da análise é retornado ao desenvolvedor, indicando que as propriedades foram verificadas ou os cenários em que as propriedades não foram verificadas (contra-exemplos). Abaixo, lista-se o resultado da execução da verificação do exemplo anterior usando o *Alloy Analyzer* para a versão 4.0 de Alloy.

```
Executing "Check transferOk"
   Solver=sat4j Bitwidth=4 MaxSeq=4 Symmetry=20
   No counterexample found. Assertion may be valid. 31ms.

Executing "Check transferOk2"
   Solver=sat4j Bitwidth=4 MaxSeq=4 Symmetry=20
   658 vars. 128 primary vars. 1080 clauses. 109ms.
   Counterexample found.Assertion is invalid. 31ms.

2 commands were executed. The results are:
   #1: No counterexample found. transferOk may be valid.
   #2: Counterexample found.transferOk2 is invalid.
```

Observe que a assertiva `trasnferOk` é indicada como *possivelmente* correta. Isto ocorre porque não se pode garantir que ela realmente esteja correta pois não são verificadas todas as possibi-

lidades. Para aumentar tal garantia, deve-se aumentar o escopo de execução através do comando `check <assertiva> for <num-instancias>`, aumentando assim o número de instâncias geradas e verificadas. Para maiores informações sobre este procedimento, pode-se consultar o livro referência de Alloy [60].

Para a segunda assertiva, foi encontrado um contra-exemplo. Na Figura 7.2, ilustra-se a visão diagramática do resultado gerada pelo *Alloy Analyzer*. Observe que o número de instâncias de `TransferenciaSaque` é igual ao número de `TransferenciaDeposito`, como afirmado pela assertiva `transferOk` mas negado pela assertiva `transferOk2`, por isso, representa um contra-exemplo desta última.

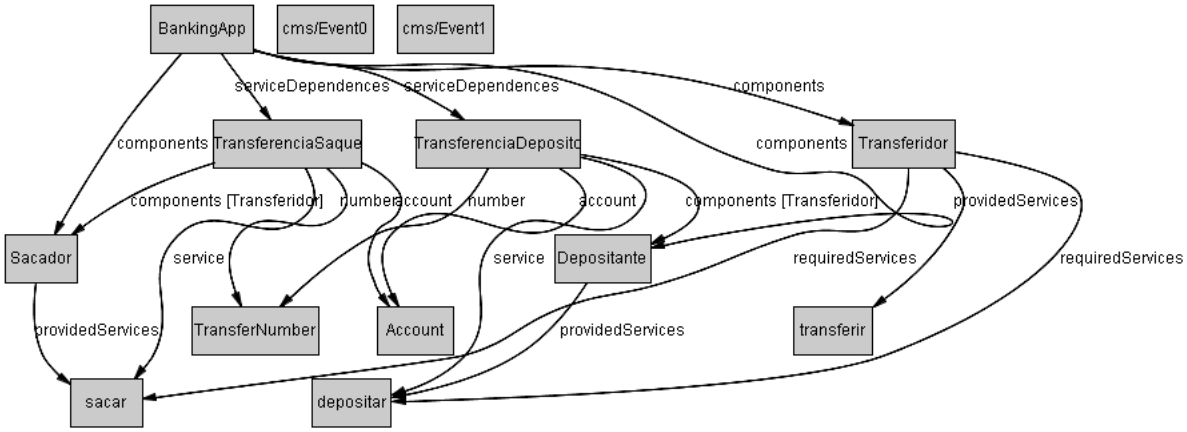



Figura 7.2: Resultado do *Alloy Analyzer*: contra-exemplo.

Através desta verificação, observa-se que o componente de transferência adicionado, seja incrementalmente no desenvolvimento ou em um cenário de evolução, possui uma especificação que vai de encontro àquela definida pelo sistema. Com isso, pode-se evitar a utilização de tal componente ou, ainda, analisar a possibilidade de flexibilização da especificação da aplicação. Independente da escolha do desenvolvedor, ele pode contar com um suporte em tempo de projeto que, ao menos, irá alertá-lo de possíveis inconsistências na especificação do sistema como um todo.

## 7.3 Considerações Finais

Neste capítulo foi apresentada uma técnica baseada no método formal Alloy para a especificação e análise de software baseado na CMS. Mais especificamente, apresentou-se um arcabouço em Alloy para a especificação das propriedades dos componentes e da aplicação e um conjunto de diretrizes para a análise e a verificação do software utilizando tal arcabouço.

A técnica se baseia na composição de modelos Alloy da aplicação sendo desenvolvida e dos componentes que a compõem. Desta forma, tem-se uma nova especificação composta, que é analisada para descobrir inconsistências e apontar possíveis cenários em que as propriedades desejadas da aplicação não são verificadas.

Há também um suporte do ambiente de composição apresentado no Capítulo 9 para a aplicação da técnica. Este ambiente se utiliza da ferramenta *Alloy Analyzer* para disponibilizar um *plug-in* que esconde do desenvolvedor o processo de composição dos modelos e execução da análise, evitando que o mesmo necessite sair do seu ambiente de desenvolvimento.

Com a técnica proposta, pode-se verificar se um dado cenário de evolução pode impactar negativamente a corretude da especificação da aplicação, provendo ao desenvolvedor a opção de, por exemplo, não inserir um dado componente caso alguma propriedade não seja verificada. A aplicação desta técnica, assim como da ferramenta apresentada no Capítulo 9, faz parte do processo de desenvolvimento apresentado no Capítulo 11.

# Capítulo 8

# Modelo de análise de desempenho

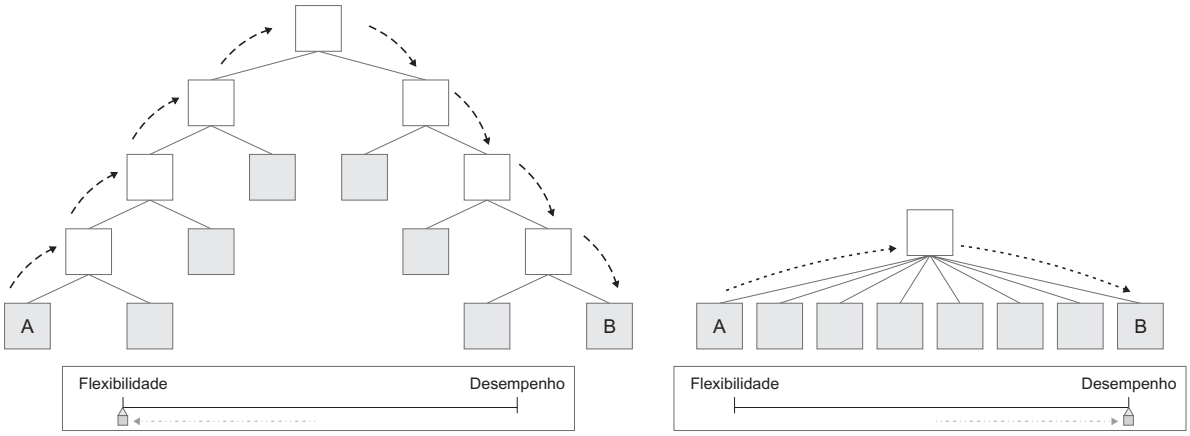
Existe uma relação de compromisso entre flexibilidade e desempenho na CMS. Apesar da flexibilidade que permite a evolução dinâmica não antecipada de componentes, seus modelos de interação e de disponibilização introduzem uma perda de desempenho durante as requisições de serviços e disparos de eventos. Isto ocorre devido à mediação da interação entre componentes via contêineres, criando uma indireção na interação entre os componentes, o que se repete a cada nível de profundidade da hierarquia.

De fato, como mencionado no Capítulo 4, a arquitetura hierárquica foi definida na CMS para beneficiar a composição de aplicações. Porém, desempenho é um requisito não funcional crítico para determinados sistemas e, dada a suspeita intuitiva de que tal hierarquia impacta negativamente o desempenho da aplicação, esta parece ser a principal desvantagem das aplicações baseadas na CMS.

Entretanto, dois argumentos podem ser apresentados em favor da CMS. O primeiro deles são os baixos valores relacionados à perda de desempenho obtidos no perfilamento do código da implementação em Java da CMS (JCF), os quais serão apresentados posteriormente. Este argumento é pouco consistente, uma vez que tais valores foram obtidos sob condições específicas, em uma plataforma de hardware e software específica. Desta forma, não se pode dizer que estes mesmos valores serão obtidos em outras plataformas, como dispositivos móveis, por exemplo.

O segundo e principal argumento diz respeito ao fato de que a profundidade da hierarquia pode ser controlada de acordo com as necessidades de desempenho da aplicação (Figura 8.1). Este é um argumento consistente pois tal característica permite definir uma arquitetura que privilegie a

flexibilidade ou o desempenho, ou ainda um meio termo entre os dois.




(a) Arquitetura para flexibilidade.

(b) Arquitetura para desempenho.

Figura 8.1: Controle da profundidade da hierarquia: flexibilidade x desempenho.

Considerando os argumentos acima, mais importante do que realizar uma medição em um cenário específico é disponibilizar um mecanismo para que o desenvolvedor possa identificar qual a melhor arquitetura dados os requisitos de desempenho da aplicação. Desta forma, pode-se inclusive identificar se uma dada implementação da CMS é adequada ou não para o desenvolvimento de uma aplicação. Afinal, não se espera que a CMS seja a "bala de prata" da Engenharia de Software [190].

Neste capítulo apresenta-se um modelo analítico para a avaliação do desempenho de aplicações com arquiteturas específicas, baseadas na CMS. Através desta avaliação, torna-se possível diminuir possíveis gargalos causados pela disposição dos componentes na arquitetura, buscando assim o melhor balanceamento entre flexibilidade e desempenho, de acordo com as necessidades da aplicação.

Como forma de validar o modelo analítico, realizou-se um perfilamento do código do JCF e os resultados foram comparados aos obtidos através do modelo. O modelo e os resultados do perfilamento, uma discussão sobre os resultados e diretrizes para utilização do modelo são apresentados a seguir.

## 8.1 Modelo Analítico

As principais operações definidas na CMS contempladas no modelo proposto são: adição, remoção e substituição de componentes; mudança de apelidos, tanto de serviços quanto de eventos; requisição de serviços; e anúncio de eventos. O desempenho é medido com base no tempo gasto nas operações necessárias para implementar a CMS, como o acesso a estruturas de dados onde estão armazenados os componentes, serviços e eventos. As fórmulas apresentadas no modelo analítico possuem parâmetros referentes a tais operações, os quais devem ser obtidos via perfilamento de código na linguagem e plataforma alvo específica. Estes valores são definidos aqui como valores a serem "calibrados".

### 8.1.1 Adição, Remoção e Substituição de Componentes

#### Adição de Componentes

De acordo com a CMS, a adição de componentes na hierarquia ocorre através da inserção de componentes dentro de contêineres, tornando seus serviços disponíveis para outros componentes e eventos de interesse. Três sub-operações estão relacionadas à inserção de um componente em um contêiner: registro do componente no contêiner; atualização da tabela de serviços e eventos para cada contêiner até a raiz da hierarquia; remoção de referências a antigos componentes provedores de serviços com apelido igual a algum serviço do novo componente. Desta forma, o tempo da operação de adição de um componente pode ser estimado como descrito na Definição 8.1.

**Definição 8.1 (Tempo de adição de componente)** Considere um componente cf a ser inserido em um contêiner ct em qualquer nível da hierarquia. O tempo $T_{ad}$ gasto para adicionar cf em ct é calculado por:

$$T_{ad} = t_{rg} + T_{ata} + T_{rm}$$

onde:

- $t_{rg}$ é o tempo médio para registrar um componente em um contêiner (valor calibrado);
- $T_{ata}$ é o tempo gasto para atualizar a tabela de serviços providos e eventos de interesse de cada contêiner a partir de ct até a raiz da hierarquia, sendo dado por

$$T_{ata} = p_c \times (n_s \times t_{sa} + n_e \times t_{ea})$$

onde:

- $p_c$ é a profundidade de cf [1];
- $n_s$ é o número de serviços providos por cf;
- $t_{sa}$ é o tempo médio para o registro de um serviço provido por um componente na tabela de serviços de um contêiner (valor calibrado);
- $n_e$ é o número de eventos de interesse de cf;
- $t_{ea}$ é o tempo médio para o registro de um evento de interesse de um componente na tabela de eventos de um contêiner (valor calibrado);

- $T_{rm}$ é o tempo gasto para remover as referências a antigos provedores de serviços com apelidos iguais aos de cf, sendo dado por

$$T_{rm} = \sum_{i=1}^{m} n_{a_i} \times t_{rr}$$

onde:

- $m$ é o número de antigos componentes provedores de serviço;
- $n_{a_i}$ é o número de arestas do provedor antigo $i$ até o menor ancestral comum [2] entre o provedor antigo e cf;
- $t_{rr}$ é o tempo médio para a remoção de uma referência a um provedor de serviço em um contêiner (valor calibrado).

Por exemplo, na arquitetura ilustrada na Figura 8.2, a adição do componente $X$ resulta em seu registro junto a seu contêiner pai e o registro de seus serviços providos e eventos de interesse em dois contêineres. Supondo que não havia antigos provedores de serviço e que $t_{rg} = 20\mu s$, $t_{sa} = 22\mu s$, $t_{ea} = 22\mu s$, $n_s = 2$ e $n_e = 3$, tem-se $T_{rm} = 0$ e $T_{ata} = 2 \times (22\mu \times 2 + 22\mu \times 3) = 220\mu s$. Logo, o tempo total seria $T_{ad} = 20\mu + 220\mu + 0 = 240\mu s$.

[1] A *profundidade* de um nó de árvore é o número de arestas da raiz ao nó.

[2] O *menor ancestral comum* entre dois nós de árvore é o ancestral comum mais profundo [172].

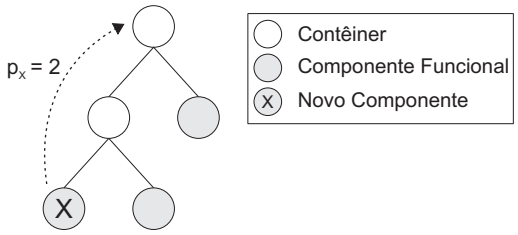



Figura 8.2: Exemplo de avaliação de desempenho da adição de componentes – melhor caso.

O cenário apresentado na Figura 8.2 representa o melhor caso da avaliação de desempenho para a adição de componentes. Isto ocorre porque não foram considerados provedores antigos dos serviços disponibilizados pelo novo componente *X*. No pior caso, considera-se que todos os novos serviços providos pelo novo componente já eram disponibilizados por vários outros componentes. Quanto mais antigos provedores existirem, maior o impacto negativo sobre o desempenho. Na Figura 8.3 ilustra-se o pior caso, no qual todos os componentes já eram provedores de algum serviço do componente sendo adicionado. Supondo os valores do cenário anterior e $t_{rr} = 18\mu s$, tem-se $T_{rm} = \sum_{i=1}^{2} n_{a_i} \times 18\mu = (1+1) \times 18\mu = 36\mu s$ e, portanto, $T_{ad} = 240\mu + 36\mu = 276\mu s$.

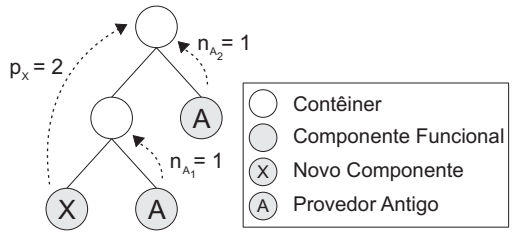



Figura 8.3: Exemplo de avaliação de desempenho da adição de componentes – pior caso.

**Remoção de Componentes**

De acordo com a CMS, a remoção de um componente da hierarquia ocorre através de duas sub-operações: remoção do registro do componente do seu contêiner pai; remoção dos registros da tabela de serviços providos e eventos de interesse para cada contêiner até a raiz da hierarquia. Desta forma, o tempo da operação de remoção de um componente pode ser estimado como descrito na Definição 8.2.

**Definição 8.2 (Tempo de remoção de componente)** Considere um componente cf a ser removido de um contêiner ct em qualquer nível da hierarquia. O tempo $T_{re}$ gasto para remover cf de ct

é calculado por:

$$T_{re} = t_{dg} + T_{atr}$$

onde:

- $t_{dg}$ é o tempo médio para remover o registro de um componente de um contêiner (valor calibrado);
- $T_{atr}$ é o tempo gasto para remover os registros da tabela de serviços providos e de eventos de interesse de cada contêiner a partir de ct até a raiz da hierarquia, sendo dado por:

$$T_{atr} = p_c \times (n_s \times t_{sr} + n_e \times t_{er})$$

onde:

  - $p_c$ é a profundidade de cf;
  - $n_s$ é o número de serviços providos por cf;
  - $t_{sr}$ é o tempo médio para remover um registro de serviço provido por um componente da tabela de serviços providos de um contêiner (valor calibrado);
  - $n_e$ é o número de eventos de interesse de cf;
  - $t_{er}$ é o tempo médio para remover um registro de um evento de interesse de um componente da tabela de eventos de interesse de um contêiner (valor calibrado);

Por exemplo, na arquitetura ilustrada na Figura 8.4, a remoção do componente $X$ resulta na remoção do seu registro junto a seu contêiner pai e na remoção do registro de seus serviços providos e eventos de interesse em dois contêineres. Supondo que $t_{dg} = 20\mu s$, $n_s = 2$, $t_{sr} = 20\mu s$, $n_e = 3$ e $t_{er} = 20\mu s$, tem-se $T_{atr} = 2 \times (2 \times 20\mu + 3 \times 20\mu) = 200\mu s$. Logo, o tempo total seria $T_{re} = 20\mu + 200\mu = 220\mu s$. Os resultados obtidos com a operação de remoção são similares ao melhor caso da operação de adição.

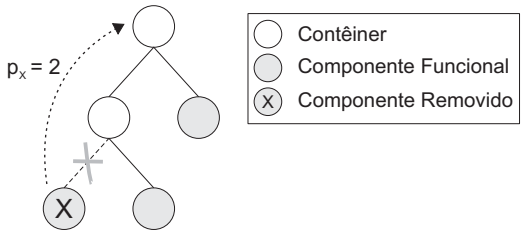



Figura 8.4: Exemplo de avaliação de desempenho da remoção de componentes.

**Substituição de Componentes**

De acordo com a CMS, a operação de substituição de componentes é composta de duas sub-operações executadas em seqüência: adição do novo componente e remoção do componente antigo. Sendo assim, pode-se aplicar o somatório das fórmulas definidas anteriormente para obter o tempo de substituição de um componente por outro, como descrito na Definição 8.3.

**Definição 8.3 (Tempo de substituição de componente)** Considere um componente $cf_1$ a ser substituído por outro componente $cf_2$, dentro de um contêiner ct em qualquer nível da hierarquia. O tempo $T_{sb}$ gasto para substituir $cf_1$ por $cf_2$ é calculado por:

$$T_{sb} = T_{ad_{c2}} + T_{re_{c1}}$$

onde:

- $T_{ad_{c2}}$ é o tempo gasto para adicionar o componente $cf_2$ em ct;
- $T_{re_{c1}}$ é o tempo gasto para remover o componente $cf_1$ de ct.

É importante ressaltar que após a adição do novo componente todas as novas requisições aos serviços do componente antigo que foram publicados com mesmo nome já serão recebidas pelo novo componente. Sendo assim, apesar da operação como um todo ter o tempo avaliado como a soma das operações de adição e remoção, do ponto de vista da execução do sistema, este tempo pode ser equivalente apenas à operação de adição, caso o novo componente tenha no mínimo os mesmos serviços providos anteriormente.

### 8.1.2 Mudança de Apelidos

De acordo com a CMS, a operação de mudança de um apelido de um serviço provido ou evento de interesse é realizada através da atualização da tabela de serviços e eventos para cada contêiner

até a raiz da hierarquia. Desta forma, o tempo da operação de mudança de apelido de um serviço ou evento pode ser estimado como descrito na Definição 8.4.

**Definição 8.4 (Tempo de mudança de apelido)** Considere um serviço ou evento "x" de um componente cf a ser alterado, estando cf em qualquer nível da hierarquia. O tempo $T_{ma}$ gasto para mudar o apelido de "x" é calculado por:

$$T_{ma} = p_c \times t_l$$

onde:

- $p_c$ é a profundidade de cf;
- $t_l$ é o tempo médio para atualização do registro de um serviço em um contêiner (valor calibrado).

### 8.1.3 Requisição de Serviços

De acordo com a CMS, quando ocorre uma requisição de serviço, esta é propagada através da hierarquia, alcançando o provedor do serviço, caso este exista. Desta forma, o tempo da operação de requisição de serviço pode ser estimado como descrito na Definição 8.5.

**Definição 8.5 (Tempo de requisição de serviço)** Considere um serviço "x" de um componente $cf_2$ a ser requisitado por um componente $cf_1$, estando $cf_1$ e $cf_2$ em qualquer nível da hierarquia. O tempo $T_{rs}$ gasto na requisição do serviço "x" é calculado por:

$$T_{rs} = (p_r + p_p) \times t_{ts}$$

onde:

- $p_r$ é a profundidade do componente requisitante $cf_1$;
- $p_p$ é a profundidade do componente provedor $cf_2$;
- $t_{ts}$ é o tempo médio para acesso à tabela de serviços de um contêiner (valor calibrado).

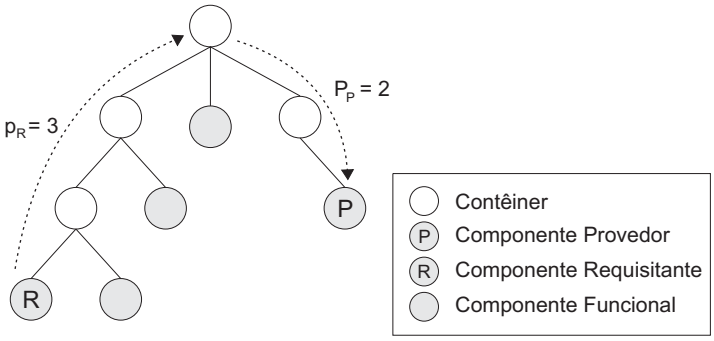



Figura 8.5: Exemplo de avaliação de desempenho da requisição de serviço.

Na Figura 8.5, ilustra-se a avaliação de desempenho de uma requisição de serviço. Neste caso, o componente *R* requisita um serviço provido pelo componente *P*. Assumindo-se o tempo de consulta à tabela de serviços como sendo $t_{ts} = 20\mu s$, tem-se que o tempo para esta operação é dado por $T_{rs} = (3+2) \times 20\mu = 100\mu s$.

É importante ressaltar que o modelo analítico não considera a execução interna dos serviços providos. O foco do modelo é identificar qual o impacto da arquitetura da CMS sobre o desempenho da aplicação. Para saber o valor total de requisição e execução do serviço, deve-se realizar perfilamento da implementação do serviço do componente.

### 8.1.4 Anúncio de Eventos

Ao contrário da operação de requisição de serviço, quando um evento é anunciado, ele é propagado através da hierarquia para todos os componentes interessados. Uma vez que vários componentes podem estar interessados em um evento específico, podem existir vários alvos para o evento. Se a mesma fórmula usada para serviços for utilizada, uma aresta poderá ser contada duas ou mais vezes. Por isso, os conceitos de *Contêiner Base* e *Participante* foram definidos (definições 8.6 e 8.7).

**Definição 8.6 (Contêiner Base)** Um Contêiner Base é: um contêiner composto por no mínimo dois participantes, sendo cada um pertencente a diferentes sub-árvores; ou um contêiner-raiz.

**Definição 8.7 (Participante)** Um Participante é: um componente interessado em algum evento; um componente anunciante de algum evento; ou um contêiner base.

**Definição 8.8 (Tempo de anúncio de evento)** Considere um evento "y" de um componente cf a ser anunciado, estando cf em qualquer nível da hierarquia. O tempo $T_{ae}$ gasto no anúncio do evento "y" é calculado por:

$$T_{ae} = t_{nl} + \sum_{i=1}^{n} g_{k_i} \times t_{te}$$

onde:

- $t_{nl}$ é o tempo médio para a criação de uma nova linha de execução para disparo do evento (valor calibrado);
- $n$ é o número de participantes $k$ na árvore;
- $g_{k_i}$ é o número de arestas a partir de cada participante ($k$) ao seu menor ancestral comum que também seja contêiner base;
- $t_{te}$ é o tempo médio para acesso à tabela de eventos de um contêiner (valor calibrado).

Por exemplo, considere a árvore representando uma arquitetura de uma aplicação de acordo com a CMS ilustrada na Figura 8.6. Nesta figura, existe um componente que anuncia um evento (A) e seis componentes que estão interessados nele (I). Supondo que $t_{nl} = 80\mu s$ e $t_{te} = 30\mu s$, tem-se que o tempo da operação é $T_{ae} = 80\mu + \sum_{i=1}^{11} g_{k_i} \times 30\mu = 80\mu + 15 \times 30\mu = 530\mu s$.

## 8.2 Análise Baseada em Perfilamento de Código

Para validar o modelo de avaliação de desempenho proposto, utilizou-se a versão 0.9 do arcabouço JCF. Foi realizado o perfilamento da execução do JCF para obter resultados reais de desempenho e compará-los ao esperado pela aplicação do modelo de avaliação.

Para a coleta dos dados, utilizou-se o perfilador de código Java JProfiler [191] a fim de verificar o desempenho do processo de disponibilização de componentes e interação baseada em serviços e eventos, refletidos em chamadas de métodos e acesso a estruturas de dados. Para tanto, todos os testes foram executados numa máquina dedicada Pentium IV 2.8 GHz e 512 MB de memória, *Java Virtual Machine* Sun versão 1.4.2, com sistema operacional Windows XP.

A seguir, apresenta-se uma comparação das estimativas obtidas pela aplicação do modelo em relação às médias reais obtidas a partir do perfilamento do JCF. Para a coleta dos resultados

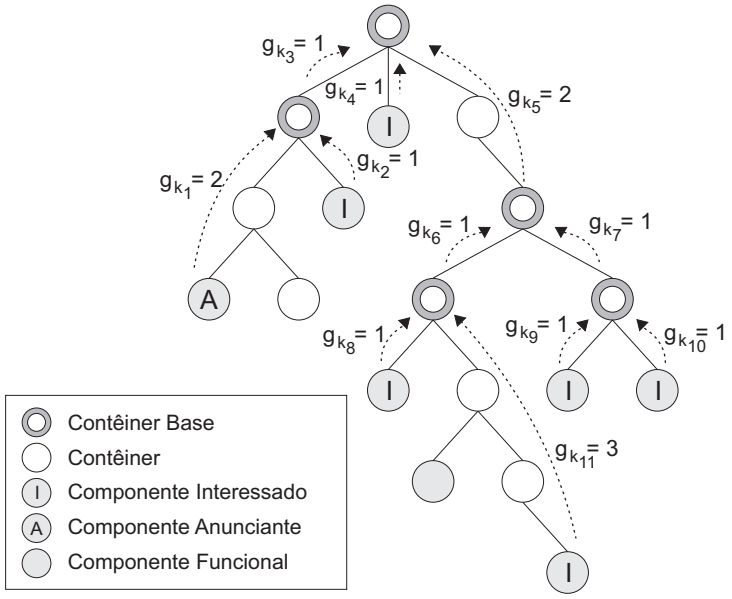



Figura 8.6: Exemplo de avaliação de desempenho do anúncio de evento.

do perfilamento, alguns "picos" de consumo de CPU foram omitidos para o cálculo dos tempos médios obtidos no perfilamento. Nos experimentos realizados, utilizou-se como ponto de partida as mesmas hierarquias de componentes apresentadas nos exemplos anteriores. A avaliação foi realizada para adição de componentes, requisição de serviços e anúncio de eventos.

Para avaliar o desempenho da adição de um novo nó componente, a profundidade da árvore foi variada. Dessa forma, a abscissa do gráfico da Figura 8.7 representa a *profundidade do novo nó*.

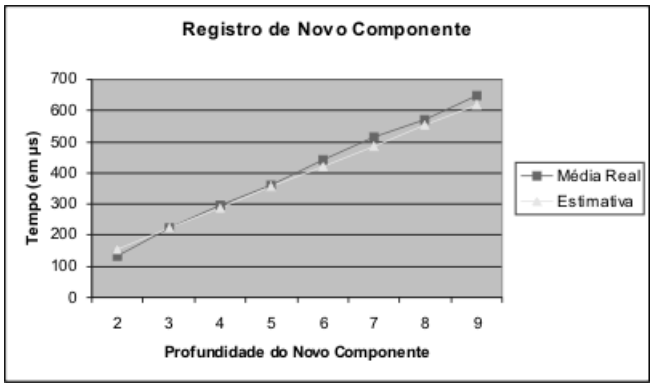



Figura 8.7: Estimativas do modelo x Resultados do perfilamento do JCF: adição de componente.

Para a requisição de serviços, as profundidades dos *nós provedor* e *nós requisitante* foram variadas. Além disso, foram exploradas diferentes configurações da árvore para uma mesma

combinação de profundidades. Dessa forma, a abscissa do gráfico da Figura 8.8 representa o *tamanho total do caminho* percorrido pela requisição de serviço.

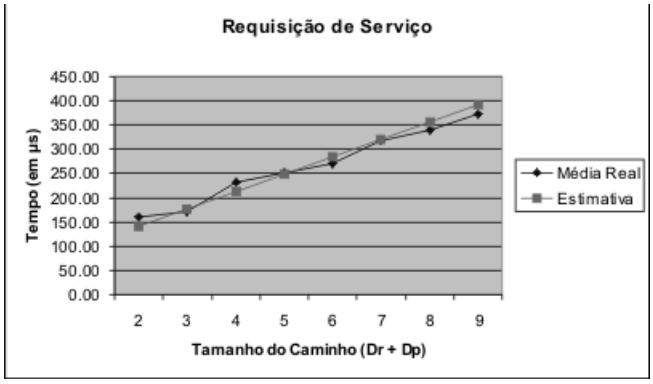



Figura 8.8: Estimativas do modelo x Resultados do perfilamento do JCF: invocação de serviço.

Por fim, para a operação de anúncio de evento, diferentes quantidades de *nós interessados* foram consideradas, também explorando diferentes configurações para cada quantidade de interessados. Dessa forma, a abscissa do gráfico da Figura 8.9 representa o *número total de arestas*.

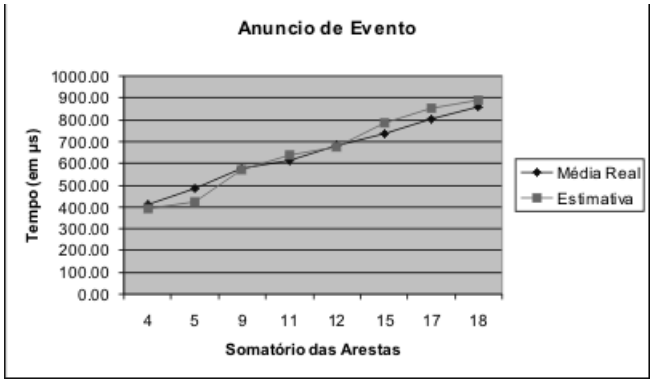



Figura 8.9: Estimativas do modelo x Resultados do perfilamento do JCF: anúncio de evento.

Os resultados do perfilamento apontam que o modelo sugere valores bem próximos das médias reais de execução, com erros médios em torno de 5% para todas as operações. Provavelmente o erro deve estar relacionado a eventuais otimizações realizadas pela máquina virtual Java e/ou sistema operacional. Ainda assim, o modelo de avaliação proposto se apresenta como uma ferramenta eficaz para identificar problemas de desempenho na utilização da CMS, ainda em tempo de definição da arquitetura.

## 8.3 Diretrizes para Utilização do Modelo

Os resultados obtidos pela aplicação do modelo não serão comparados a resultados obtidos em outras abordagens ou trabalhos relacionados. Este aspecto comparativo de desempenho está fora do escopo deste trabalho, onde outros aspectos são considerados e apresentados no Capítulo 3. A motivação para o modelo analítico proposto é ajudar a responder às seguintes perguntas: i) A implementação na linguagem *X* da CMS é adequada à minha aplicação? ii) Dado que tal linguagem é adequada, a arquitetura *Y* é adequada à minha aplicação? Para responder a estas perguntas, as etapas a seguir devem ser realizadas.

**1. Elencar requisitos de desempenho da aplicação**

Identificar quais os requisitos de desempenho da aplicação com base no tempo máximo esperado para executar cada uma das operações inerentes à CMS. Pode-se ter restrições de desempenho para cada uma das operações em relação a componentes, serviços e eventos específicos, dependendo da necessidade de avaliação, como ilustrado na Tabela 8.1.

| **Operação** | **Tempo Máximo** |
|---|---|
| Adição do componente *X* no contêiner $Ct_1$ | 100$\mu$ s |
| Remoção do componente *Y* | 60$\mu$ s |
| Substituição do componente *K* por *Z* | 130$\mu$ s |
| Mudança do apelido do serviço “salvar” do componente *L* | 200$\mu$ s |
| Mudança do apelido do evento “salvo” do componente *P* | 350$\mu$ s |
| Requisição do serviço “registrar” pelo component *Z* | 250$\mu$ s |
| Anúncio do evento “registrado” pelo componente *W* | 300$\mu$ s |
| ... | ... |

Tabela 8.1: Exemplo de tabela de requisitos de desempenho.

**2. Calibrar os valores dos parâmetros**

Realizar medições de desempenho para obter os valores dos parâmetros utilizados nas fórmulas do modelo analítico de acordo com a plataforma alvo de software e hardware, na linguagem de

implementação da CMS escolhida. Na Tabela 8.2 ilustra-se uma tabela de calibração com valores hipotéticos.

Para obter os valores para outras plataformas e linguagens, deve-se utilizar uma ferramenta para perfilamento de código, como por exemplo: *JProfiler* [191], para Java; *The Python Profiler* [192], para Python; e *gprof* [193], para C++.

| Sigla | Descrição | Valor |
|---|---|---|
| $t_{rg}$ | Tempo médio para registrar um componente em um contêiner. | $10\mu s$ |
| $t_{sa}$ | Tempo médio para o registro de um serviço provido por um componente na tabela de serviços de um contêiner. | $10\mu s$ |
| $t_{ea}$ | Tempo médio para o registro de um evento de interesse de um componente na tabela de eventos de um contêiner. | $10\mu s$ |
| $t_{rr}$ | Tempo médio para a remoção de uma referência a um provedor de serviço em um contêiner. | $10\mu s$ |
| $t_{dg}$ | Tempo médio para remover o registro de um componente de um contêiner. | $10\mu s$ |
| $t_{sr}$ | Tempo médio para remover um registro de serviço provido por um componente da tabela de serviços providos de um contêiner. | $10\mu s$ |
| $t_{er}$ | Tempo médio para remover um registro de um evento de interesse de um componente da tabela de eventos de interesse de um contêiner. | $10\mu s$ |
| $t_{l}$ | Tempo médio para atualização do registro de um serviço em um contêiner. | $10\mu s$ |
| $t_{ts}$ | Tempo médio para acesso à tabela de serviços de um contêiner. | $10\mu s$ |
| $t_{te}$ | Tempo médio para acesso à tabela de eventos de um contêiner. | $10\mu s$ |
| $t_{nl}$ | Tempo médio para a criação de uma nova linha de execução para disparo do evento. | $10\mu s$ |

Tabela 8.2: Parâmetros de calibração com valores hipotéticos.

**3. Aplicar os valores calibrados ao modelo analítico**

Utilizar os parâmetros de calibração obtidos na Etapa 2 para calcular cada uma das operações da tabela de requisitos definida na Etapa 1. Na Tabela 8.3 ilustra-se um exemplo de tabela de resultados obtidos através da aplicação das fórmulas.

| **Operação** | **Resultado** |
|---|---|
| Adição do componente $X$ no contêiner $Ct_1$ | 90$\mu$ s |
| Remoção do componente $Y$ | 60$\mu$ s |
| Substituição do componente $K$ por $Z$ | 180$\mu$ s |
| Mudança do apelido do serviço “salvar” do componente $L$ | 150$\mu$ s |
| Mudança do apelido do evento “salvo” do componente $P$ | 250$\mu$ s |
| Requisição do serviço “registrar” pelo component $Z$ | 300$\mu$ s |
| Anúncio do evento “registrado” pelo componente $W$ | 168$\mu$ s |
| ... | ... |

Tabela 8.3: Exemplo de tabela de valores obtidos da aplicação das fórmulas.

**4. Comparar os resultados obtidos com os valores dos requisitos elencados**

Na Tabela 8.4, ilustra-se a comparação entre os valores obtidos na aplicação do modelo analítico e os valores de tempo máximo requeridos para a aplicação. Neste exemplo, duas das operações extrapolaram o tempo máximo requerido. Estas operações estão em destaque na Tabela 8.4.

| **Operação** | **Resultado** | **Requerido(máx)** |
|---|---|---|
| Adição do componente $X$ no contêiner $Ct_1$ | 90$\mu$ s | 100$\mu s$ |
| Remoção do componente $Y$ | 55$\mu$ s | 60$\mu s$ |
| Substituição do componente $K$ por $Z$ | 180$\mu$ s | 130$\mu s$ |
| Mudança do apelido do serviço “salvar” do componente $L$ | 150$\mu$ s | 200$\mu s$ |
| Mudança do apelido do evento “salvo” do componente $P$ | 250$\mu$ s | 350$\mu s$ |
| Requisição do serviço “registrar” pelo component $Z$ | 300$\mu$ s | 250$\mu s$ |
| Anúncio do evento “registrado” pelo componente $W$ | 168$\mu$ s | 300$\mu s$ |
| ... | ... | ... |

Tabela 8.4: Tabela comparativa dos valores obtidos da aplicação das fórmulas e os requisitos de desempenho.

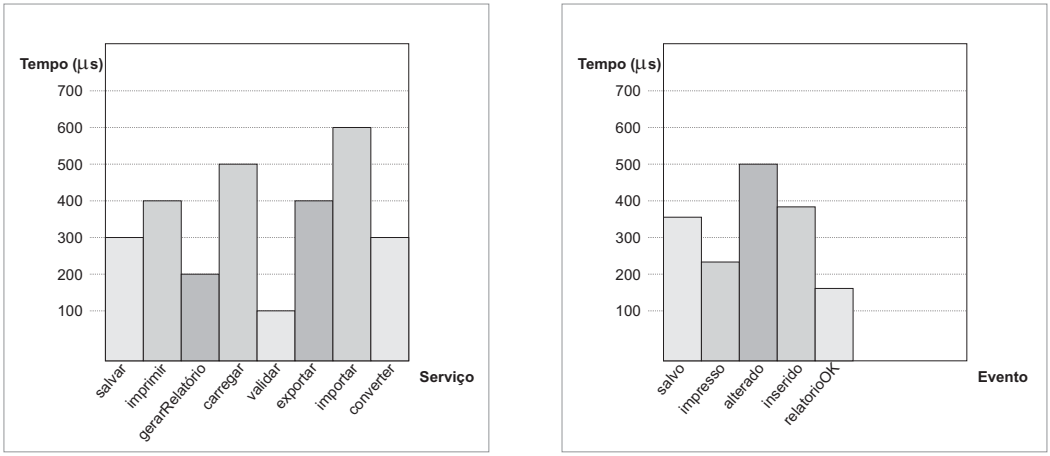
**5. Redefinir a arquitetura ou descartar o uso da CMS**

De acordo com o resultado da comparação realizada na Etapa 4, deve-se redefinir a arquitetura para resolver possíveis problemas de desempenho. Esta redefinição pode ocorrer através da reorganização dos componentes na hierarquia, reduzindo a profundidade da árvore de contêineres e componentes, por exemplo. Depois, retorna-se à Etapa 3 para aplicar o modelo de acordo com a nova arquitetura. Se mesmo com uma arquitetura de profundidade igual a 1 os requisitos de desempenho não forem contemplados, pode ser o caso de descartar a utilização da implementação da CMS escolhida ou até mesmo da própria CMS.

Independentemente dos requisitos serem contemplados ou não, pode-se buscar uma arquitetura mais balanceada em termos de desempenho através de uma visualização mais geral dos tempos gastos para invocação de serviços e anúncios de eventos entre os componentes. Isto pode ser realizado aplicando o modelo analítico a todos os componentes, focando em seus serviços requeridos e nos interessados em seus eventos.

Na Figura 8.10, ilustra-se o gráfico referente aos resultados da aplicação dos dados do exemplo anterior considerando a requisição de todos os serviços requeridos e anúncio de todos os eventos.

Além disso, pode-se calcular a média do tempo gasto na requisição de serviços e anúncio de eventos em um componente e ter uma visão geral dos componentes do sistema em relação a esta média, como ilustrado na Figura 8.11. Desta forma, pode-se identificar quais componentes poderiam ser reorganizados levando em conta todos os seus serviços e eventos.




(a) Tempos de requisição de serviços.

(b) Tempos de anúncio de eventos.

Figura 8.10: Gráficos descrevendo tempo gasto por cada serviço e evento.

Vale ressaltar que os gráficos exibidos na Figura 8.10 e 8.11 não contemplam os dados referentes à múltiplas execuções de serviço, ou seja, considera-se que cada serviço requerido, por exemplo, é requisitado apenas uma vez. Ainda assim, a informação pode servir como base para reorganizar a arquitetura e melhorar o desempenho.

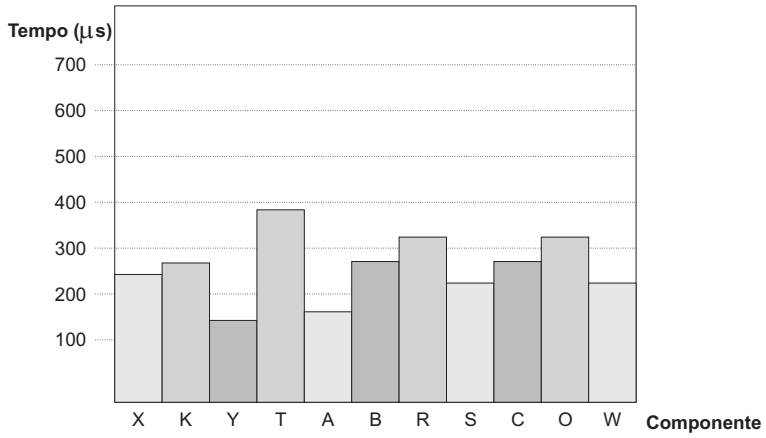



Figura 8.11: Gráfico descrevendo tempo médio gasto por cada serviço e evento para cada componente.

É evidente que aplicar estas etapas manualmente é impraticável, considerando uma grande quantidade de componentes em uma aplicação real. Até mesmo porque a cada alteração na arquitetura, uma nova análise deve ser feita e novos gráficos gerados. Esta é a motivação para uma das ferramentas desenvolvidas e pertencentes ao ambiente de composição descrito no Capítulo 9.

## 8.4 Considerações Finais

Neste capítulo foi apresentado um modelo analítico para avaliação de desempenho de aplicações baseadas na CMS, de acordo com a arquitetura das mesmas. O modelo apresenta fórmulas para avaliação de desempenho das operações de adição, remoção e substituição de componentes, invocação de serviços, anúncio de eventos e mudança de apelidos de serviços e eventos. Estas fórmulas devem ser calibradas com valores específicos de aplicação para obter a perda de desempenho para uma dada arquitetura. Por fim, apresentou-se um conjunto de diretrizes a serem seguidas para facilitar a aplicação do modelo proposto.

O resultado da avaliação indica que o impacto sobre o desempenho pode ser minimizado através do gerenciamento da profundidade da hierarquia de contêineres. Por exemplo, considere

uma arquitetura onde existe apenas um contêiner (a raiz da hierarquia) e todos os seus filhos são componentes funcionais, ou seja, sua profundidade é igual a 1. Neste caso: uma operação de disponibilização de componente é executada com apenas um cadastro de serviços e eventos; uma requisição de serviço é feita através de uma consulta à tabela de serviços providos; e um anúncio de evento é feito através de uma consulta à tabela de eventos de interesse. Considerando o tempo médio de consulta às tabelas extremamente pequeno (poucos microsegundos), a perda de desempenho é pouco significativa.

Embora melhore o desempenho das operações, uma hierarquia de pouca profundidade reduz a modularidade, coesão e flexibilidade da arquitetura. Isto ocorre porque o uso de contêineres permite a modularização de componentes relacionados, possibilitando a definição de uma fachada para os serviços e eventos dos componentes e permitindo a mudança de contêineres inteiros por outros contêineres ou componentes, como discutido no Capítulo 4.

Por outro lado, quanto mais profunda for a hierarquia, maior a quantidade de acessos às tabelas de serviços e eventos, reduzindo assim o desempenho da aplicação. Um exemplo de "hierarquia profunda" é ilustrado na Figura 8.6, onde ocorreram quinze acessos a estruturas de dados para o anúncio de um evento para apenas seis componentes interessados.

Portanto, dependendo dos requisitos de desempenho e flexibilidade de uma aplicação, uma hierarquia mais ou menos profunda será mais adequada. O modelo de avaliação de desempenho apresentado é útil para avaliar o desempenho de arquiteturas específicas, permitindo analisar e escolher a arquitetura adequada de acordo com os requisitos da aplicação.

Os resultados do perfilamento revelaram que o modelo sugere valores bem próximos das médias reais de execução, com erros médios em torno de 5% para todas as operações. Ainda assim, o modelo de avaliação proposto se apresenta como uma ferramenta eficaz para identificar problemas de desempenho na utilização da CMS, ainda em tempo de definição da arquitetura. Além disso, uma vez que os parâmetros de calibração foram definidos genericamente e não estão vinculados às características específicas de Java, ele pode ser utilizado para avaliar arquiteturas CMS mesmo com implementações em outras linguagens. Basta apenas calibrar os parâmetros com os valores da linguagem escolhida na plataforma alvo.

# Capítulo 9

# Ambientes de Desenvolvimento e Composição de Software Baseado na CMS

Os arcabouços de software que implementam a CMS, como o JCF por exemplo, provêem uma API para o desenvolvimento de aplicações com suporte à evolução dinâmica não antecipada. Porém, como visto no Capítulo 5, dependendo do porte da aplicação sendo desenvolvida, a implementação de componentes, serviços e eventos, assim como a composição da aplicação, apenas utilizando a API podem ser atividades custosas. A gerência das dependências entre os componentes também se torna complexa sem o auxílio de uma ferramenta.

Neste capítulo, apresentam-se os ambientes de desenvolvimento e composição de software baseado na CMS. O primeiro deles, denominado *Component Development Environment* (CDE), provê suporte ao desenvolvimento de componentes. O segundo, denominado *Component Composition Tools* (CCT), provê suporte à composição de aplicações, as quais utilizam os componente desenvolvidos no CDE.

Os ambientes foram implementados em Java sobre a plataforma Eclipse [59] como conjuntos de *plug-ins*. A escolha de Eclipse como plataforma-base para o ambiente deve-se ao suporte desta plataforma ao desenvolvimento de ambientes de desenvolvimento, incluindo interface gráfica, *wizards*, internacionalização, gerenciamento de janelas, visões e perspectivas, dentre outras características que facilitam a construção de tais ambientes.

Apesar de serem implementados em Java, os ambientes são extensíveis ao desenvolvimento e composição de software utilizando outras linguagens, desde que se tenha um arcabouço corres-

pondente e que algumas adaptações específicas de linguagem sejam realizadas na interface gráfica de algumas ferramentas dos ambientes, como por exemplo, no editor de componentes do CDE. Atualmente, apenas componentes e aplicações Java podem ser desenvolvidos utilizando o CDE e o CCT.

## 9.1 *Component Development Environment*

O objetivo do CDE é facilitar as atividades de desenvolvimento de componentes ao utilizar um arcabouço que implementa a CMS. Dentre estas atividades destaca-se a especificação de serviços providos, serviços requeridos, eventos de interesse, eventos anunciados, propriedades de inicialização e interesses.

Além disso, o CDE oferece suporte ao empacotamento de componentes, com geração automática de um descritor XML (*eXtensible Markup Language*) [194] para o componente, o qual será interpretado posteriormente pela paleta de componentes do CCT. Todas estas atividades são de responsabilidade do desenvolvedor do componente e, sem o auxílio de uma ferramenta, tornam-se muito dispendiosas.

### 9.1.1 Arquitetura do CDE

Na Figura 9.1, ilustra-se a arquitetura do CDE. As ferramentas que fazem parte desta arquitetura são descritas a seguir.

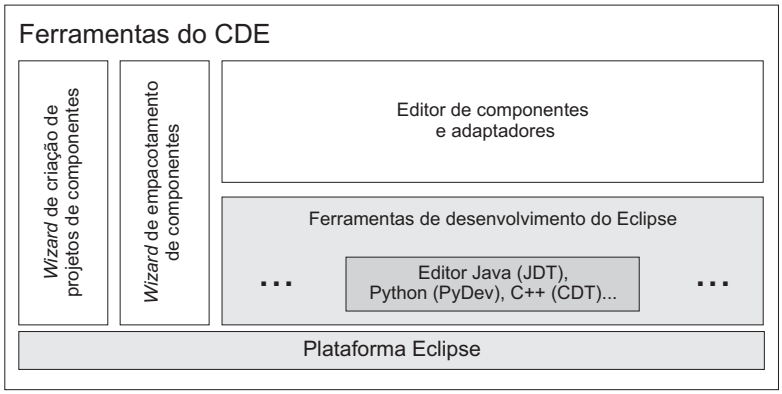



Figura 9.1: Arquitetura do CDE.

- **Plataforma Eclipse -** Infra-estrutura disponibilizada pela plataforma Eclipse para a construção de ferramentas. Possui uma API rica para o desenvolvimento de *plug-ins*, com suporte à construção de elementos de interface, gerência de projeto e internacionalização, propriedades, *wizards* [1], dentre outros [195, 59].

- **Ferramentas de Desenvolvimento do Eclipse -** Conjunto de ferramentas disponibilizado para a plataforma Eclipse para programação usando uma linguagem específica. Dentre as ferramentas disponibilizadas, incluem-se editores para classes, visualizadores para estruturas de pacotes/módulos, dentre outras. Exemplos de tais ferramentas são: JDT (*Java Development Tools*) [196], para Java; CDT (*C++ Development Tools*) [197], para C++; e PyDev (*Python Development Environment*) [198], para Python.

- **Editor de componentes e adaptadores -** Editor de classes estendido para a construção de componentes de acordo com a CMS. Implementa mecanismos relacionados à listagem de serviços e eventos disponíveis na paleta de componentes. Este é o único módulo do CDE que é dependente de linguagem, pois suas funcionalidades dependem da sintaxe da mesma. No caso de Java, este editor estende o JDT. Para implementação em outras linguagens devem ser estendidos os editores correspondentes. Por exemplo, para C++ e Python, podem ser estendidos os editores do CDT e do PyDev, respectivamente.

- ***Wizard* de criação de projeto de componentes -** Esta ferramenta é utilizada para auxiliar o desenvolvedor na criação de um projeto de componente de acordo com a CMS. Após a inserção dos dados referentes ao componente sendo criado e da finalização do *wizard*, cria-se automaticamente uma estrutura de projeto no Eclipse para o desenvolvimento de componentes, com inserção automática das bibliotecas necessárias e geração de esqueleto de código da classe principal do componente.

- ***Wizard* de empacotamento de componentes -** Esta ferramenta é utilizada para "empacotar" o conjunto de classes que implementa o componente e gerar um arquivo de componente, com extensão *.cmc*, que é utilizado em projetos de composição através do CCT. O arquivo *.cmc* contém um descritor XML gerado automaticamente por esta ferramenta.

[1] *Wizards* são seqüências de formulários cujo preenchimento é realizado passo a passo.

### 9.1.2 Utilizando a interface gráfica das ferramentas do CDE

O primeiro passo para a criação de um projeto de desenvolvimento de componente é a utilização do *wizard* ilustrado na Figura 9.2. Na primeira tela são requisitadas informações sobre o nome do projeto e a organização do projeto, como por exemplo, a separação dos arquivos fontes dos binários.

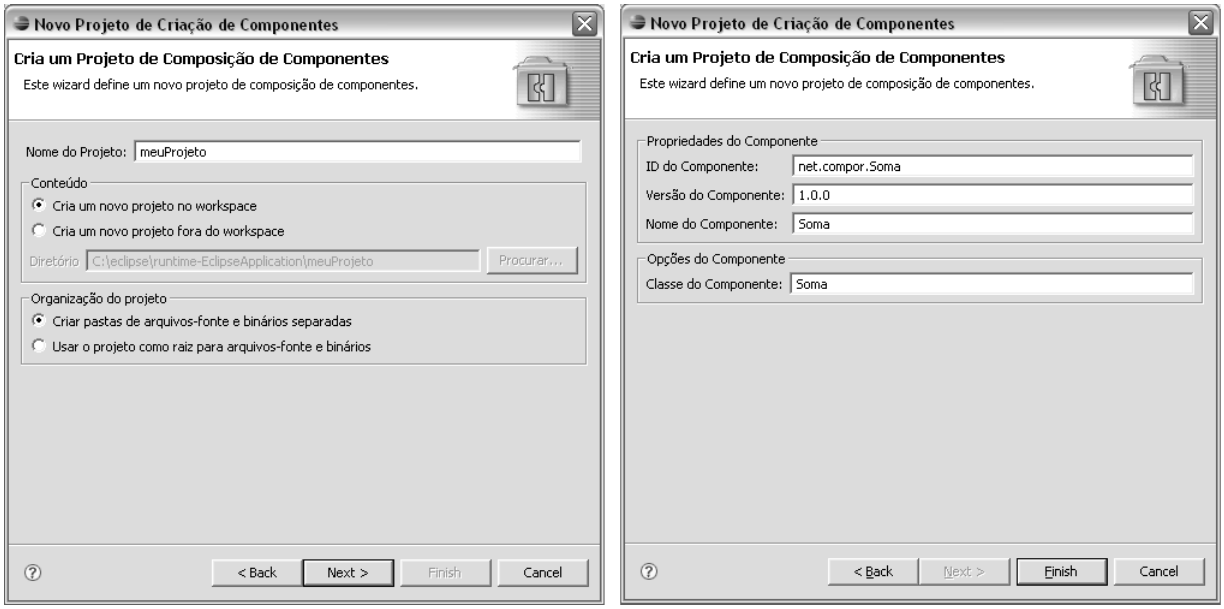



Figura 9.2: *Wizard* de criação de componentes.

Na tela seguinte, deve-se definir o identificador único do componente, a versão do mesmo e o nome do componente. Além disso, define-se o nome da classe principal do componente, cujo esqueleto será gerado automaticamente. Ao finalizar o *wizard*, a perspectiva de desenvolvimento na linguagem específica, neste caso, Java, é aberta. As bibliotecas necessárias ao desenvolvimento do componente são carregadas automaticamente, como ilustrado na Figura 9.3. Um descritor XML contendo informações sobre o componente também é automaticamente crido e colocado na pasta do projeto (*component.xml*). Todas as informações sobre o componente são armazenadas neste arquivo.

Um esqueleto da classe principal do componente é automaticamente gerado, como ilustrado na Figura 9.4. A partir de então, trata-se de uma implementação na linguagem específica, neste exemplo, em Java. O editor ainda provê mecanismos de assistência de código para facilitar a identificação de serviços providos e eventos anunciados pelos componentes da paleta do CCT.

Esta mesma funcionalidade pode ser utilizada para a criação de adaptadores e *scripts* de execução.

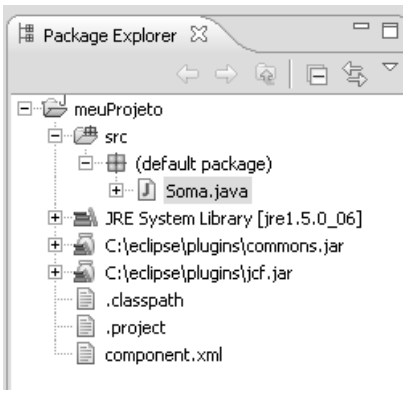



Figura 9.3: Pacotes, classes e bibliotecas do componente.

Após finalizar a implementação do componente, deve-se utilizar o *wizard* de empacotamento de componente para poder disponibilizá-lo para fazer parte de uma aplicação. Na Figura 9.5, ilustra-se tal *wizard*, que é baseado no mecanismo de exportação de classes Java para arquivos JAR (Java ARchive). O resultado da aplicação desta ferramenta é um arquivo *.cmc*, referente ao componente, contendo todas as informações necessárias para a utilização em uma aplicação.

## 9.2 *Component Composition Tools*

O objetivo do CCT é prover ao desenvolvedor da aplicação o suporte necessário para a construção da hierarquia de contêineres e componentes funcionais, de acordo com a CMS . Além disso, deve fornecer mecanismos para a verificação de possíveis erros de dependências não contempladas, sejam de serviços ou de eventos. Por fim, deve estar integrado aos servidores de aplicação para realizar a preparação e a implantação de componentes e aplicações.

### 9.2.1 Arquitetura do CCT

Na Figura 9.6, ilustra-se a arquitetura das ferramentas do CCT. As ferramentas que compõem esta arquitetura são descritas a seguir, com exceção da plataforma Eclipse, já descrita anteriormente.

- ***Wizard* de criação de projeto de aplicação -** Esta ferramenta é utilizada para auxiliar o desenvolvedor na criação de um projeto de aplicação de acordo com a CMS. Após a inserção

```
import net.compor.frameworks.jcf.EventAnnouncement;
import net.compor.frameworks.jcf.FunctionalComponent;
import net.compor.frameworks.jcf.service.ServiceResponse;

public class Soma extends FunctionalComponent {

    public Soma() {
        super("Soma");
    }

    @Override
    protected void startImpl() {
        // TODO Auto-generated method stub
    }

    @Override
    protected void stopImpl() {
        // TODO Auto-generated method stub
    }

    @Override
    protected void cannotReceiveEvent(EventAnnouncement arg0, Throwable arg1) {
        // TODO Auto-generated method stub
    }

    public void receiveServiceResponse(ServiceResponse arg0) {
        // TODO Auto-generated method stub
    }
}
```

Figura 9.4: Classe principal do componente.

dos dados referentes à aplicação sendo criada e da finalização do *wizard*, cria-se automaticamente uma estrutura de projeto no Eclipse para a composição da aplicação, com contêiner raiz e geração do descritor XML da aplicação (*application.xml*).

- **Paleta de componentes -** Ferramenta para disponibilização de componentes a serem usados na composição do sistema. Possui uma paleta de componentes carregada a partir de um conjunto de arquivos *.cmc* armazenados em um dado diretório. A partir desta paleta os componentes podem ser "arrastados" para a árvore de componentes.

- **Árvore/Inspetor de componentes -** Ferramenta para construção da hierarquia de componentes e contêineres do sistema. Além da árvore, um inspetor para exibição das propriedades dos nós da árvore é também disponibilizado.

- **Análise de desempenho -** Ferramenta para a análise de desempenho das aplicações baseadas na CMS. Esta ferramenta utiliza o modelo analítico descrito no Capítulo 8 para exibir gráficos que auxiliem o usuário a identificar possíveis problemas na estrutura hierárquica que estão causando impacto negativo sobre o desempenho.

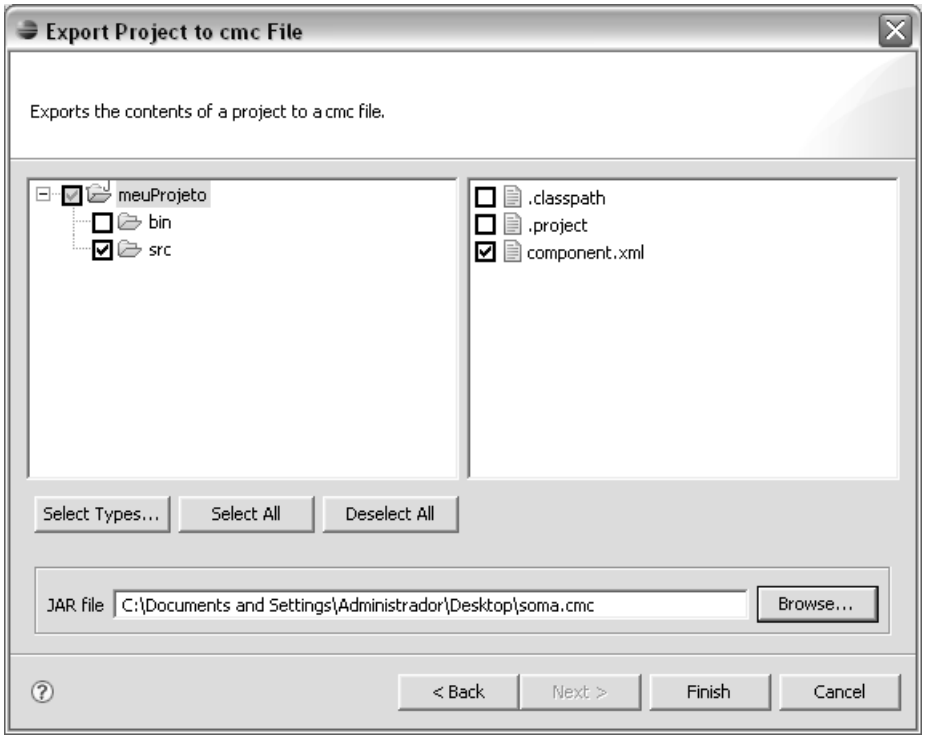



Figura 9.5: *Wizard* de exportação de componentes.

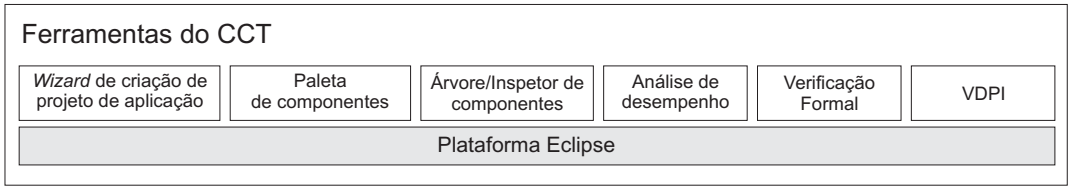



Figura 9.6: Arquitetura das ferramentas do CCT.

- **Verificação formal -** Ferramenta para verificação formal das propriedades do sistema diante de um determinado cenário de evolução. Esta ferramenta utiliza a técnica descrita no Capítulo 7 para dar um retorno ao desenvolvedor quanto ao impacto na corretude do sistema durante as atividades de evolução.

- **VDPI -** *Validação de Dependências, Preparação e Implantação*. Ferramenta para verificação de tipos de parâmetros, retornos e exceções relacionadas às dependências de serviços e eventos entre os componentes da aplicação. Esta ferramenta é responsável pela *preparação* e *implantação* dos componentes e da aplicação nos servidores de aplicação, os quais serão descritos posteriormente.

### 9.2.2 Utilizando a interface gráfica das ferramentas do CCT

A interface gráfica do CCT é formada por uma perspectiva específica. Uma perspectiva no Eclipse é composta por um conjunto de painéis, denominados *visões*. Quando um novo projeto de composição é criado, a perspectiva do CCT é exibida automaticamente.

O primeiro passo para a criação de um projeto é realizado através do *wizard* de criação de projeto do próprio Eclipse, como ilustrado na Figura 9.7. O próximo passo é fornecer os dados básicos do projeto através do *wizard* de criação de projeto (Figura 9.8). As informações são as mesmas fornecidas para a criação de um projeto Java. Porém, a estrutura física do projeto criado é específica para o CCT.

Cada projeto no CCT é criado com um contêiner raiz, um diretório de saída (caso a primeira opção de *Project Layout* seja escolhida), além dos arquivos *.project*, *.cctproject* e *RootContainer.def*. O arquivo *.project* contém informações sobre a natureza do projeto (no caso, de composição de software) e do mecanismo de construção (*build*), que será detalhado posteriormente. O arquivo *.cctproject* contém informações específicas do CCT, enquanto o arquivo *RootContainer.def* descreve informações sobre o contêiner raiz.

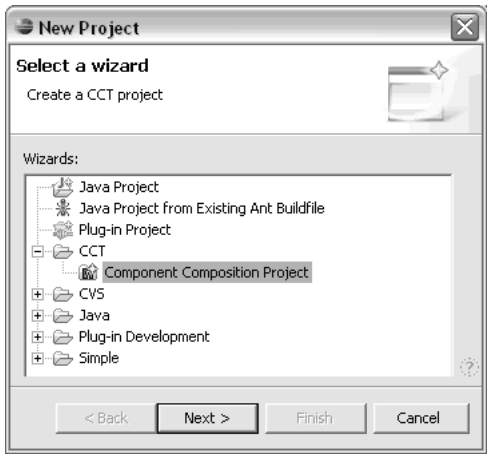



Figura 9.7: Primeiro passo para a criação de um projeto de composição.

Com base no diretório de projeto e com a informação sobre o contêiner raiz, a visão da árvore de componentes pode ser construída. A partir do diretório do contêiner raiz, cada pasta é considerada um contêiner da aplicação. Sendo assim, tem-se a estrutura hierárquica da aplicação representada pela estrutura hierárquica de diretórios. Na Figura 9.9, ilustra-se a visão da árvore de componentes. Observe que o arquivo *RootContainer.def* não é exibido nessa visão. Isto ocorre

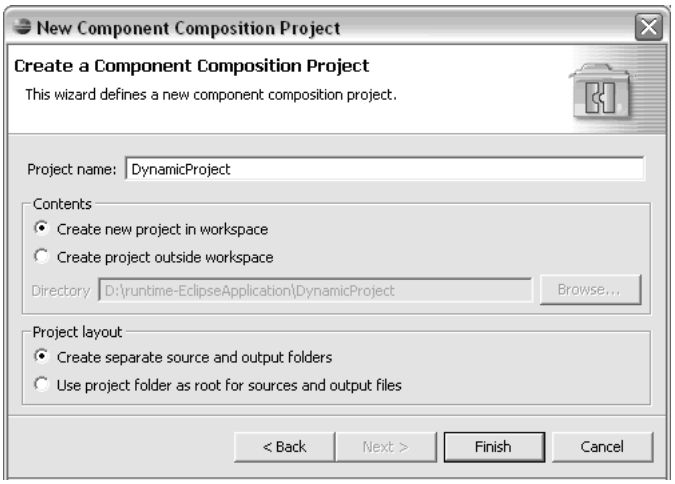



Figura 9.8: *Wizard* para a criação de um novo projeto de composição.

porque a própria visão filtra os arquivos que são exibidos de acordo com a extensão, que é *.def* neste caso. Além disso, identifica-se o contêiner da aplicação, definindo um ícone específico para contêineres.

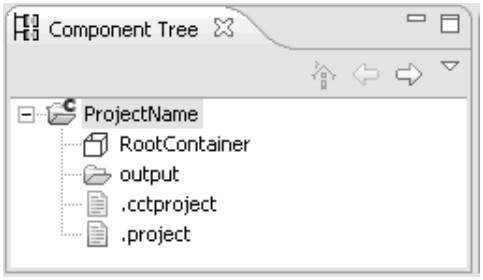



Figura 9.9: Visão da árvore de componentes.

A criação de novos contêineres também é realizada através da árvore de componentes. Basta clicar com o botão direito do mouse sobre algum contêiner e adicionar um novo contêiner como seu "filho". As informações de cada nó da árvore são exibidas na *visão de propriedades*. De acordo com o tipo de nó selecionado, tem-se um conjunto de propriedades específicas. As propriedades são as mesmas definidas na CMS, como serviços providos e requeridos, eventos anunciados e de interesse, dados de inicialização etc. É através desta visão que algumas propriedades de configuração podem ser definidas, tais como os apelidos de serviços e eventos. Na Figura 9.10, ilustra-se a visão de propriedades.

A inserção de componentes na aplicação ocorre com base na visão da paleta de componentes. A informação sobre os componentes é carregada na inicialização do CCT, a partir do diretório

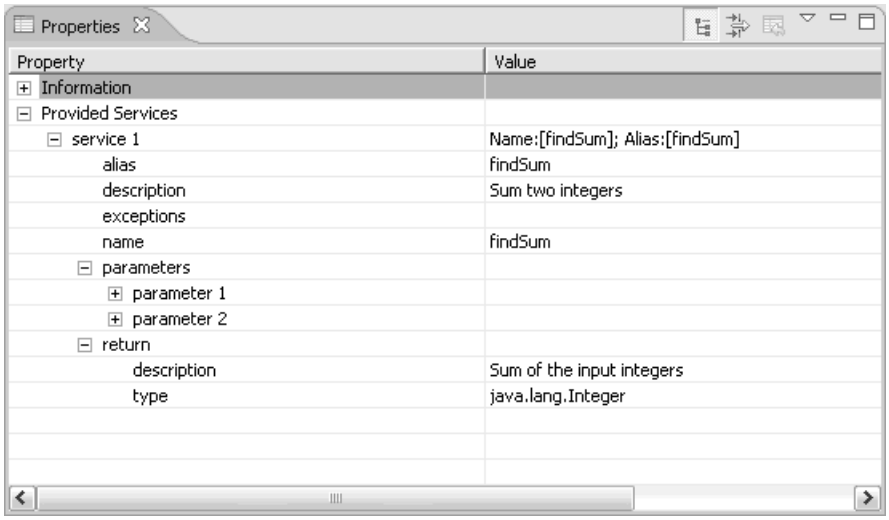



Figura 9.10: Inspetor da árvore de componentes.

onde se encontra o repositório de componentes. Por exemplo, na Figura 9.11, ilustra-se a paleta de componentes contendo quatro componentes e um domínio (“exemplo”), representado na interface gráfica por uma aba. Cada aba corresponde a um domínio diferente que é armazenado em uma estrutura de pasta diferente.

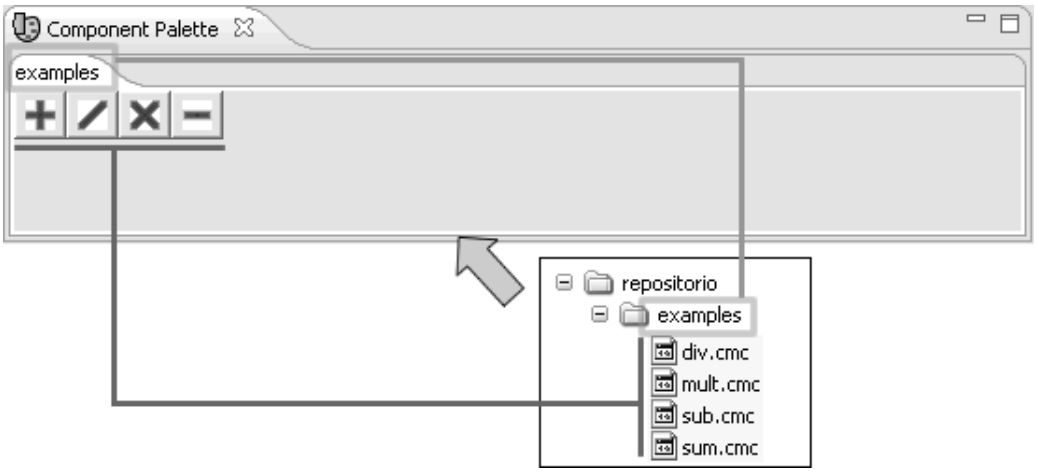



Figura 9.11: Paleta de componentes.

Os arquivos de componentes possuem extensão *.cmc*, um tipo especial de arquivo *.zip*. A ferramenta busca por arquivos com esta extensão e extrai o arquivo de ícone e um arquivo *component.xml* contendo as informações de serviços, eventos e propriedades de inicialização do componente. Uma vez que tais serviços são descritos em XML, o mecanismo de construção da paleta é independente da linguagem de implementação do componente.

Neste momento da construção da aplicação, pode-se utilizar a ferramenta de análise de desempenho para descobrir potenciais problemas de desempenho oriundos da disposição dos componentes na arquitetura. Na Figura 9.12 são ilustrados os gráficos gerados e exibidos na visão de análise do CCT. A partir destas informações, pode-se reorganizar os componentes da hierarquia visando obter uma melhor coesão funcional e melhor desempenho.

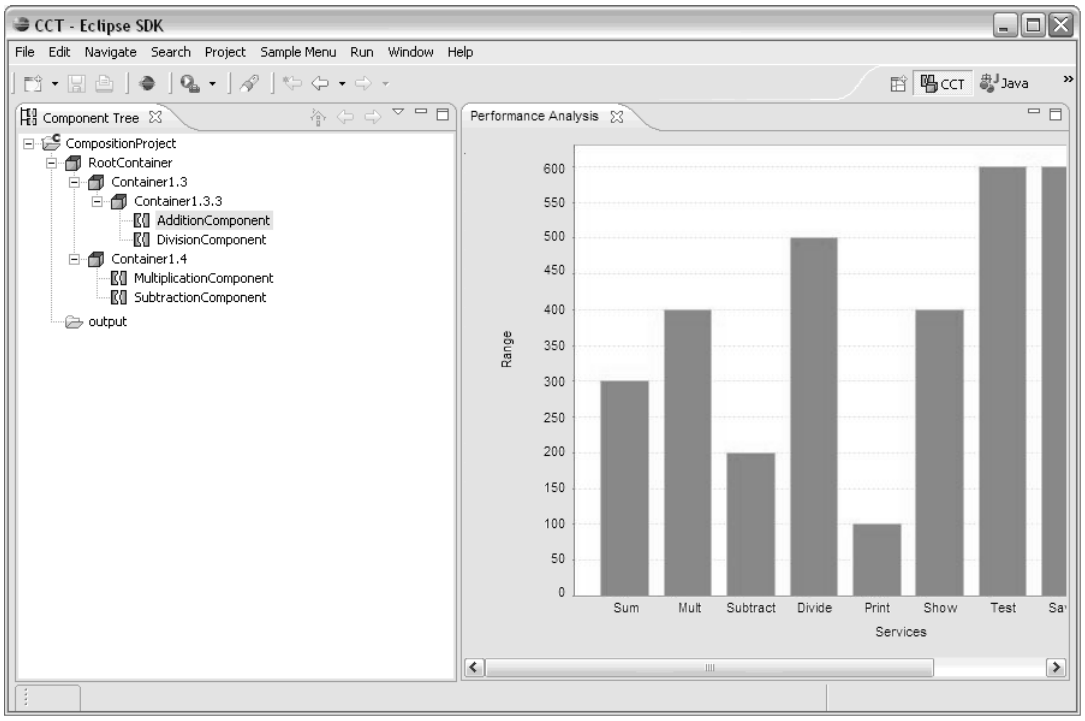



Figura 9.12: Visão de análise de desempenho.

Após a definição da hierarquia e a composição de componentes, pode-se utilizar a ferramenta VDPI para a verificação de problemas de dependência, preparar a aplicação para implantação e realizar a implantação dos componentes e da própria aplicação. A VDPI estende os mecanismos do Eclipse para construção (*build*) e execução (*run*) de aplicações. Fazendo uma comparação com uma ferramenta de desenvolvimento orientado a objetos, a *preparação* do CCT equivale à compilação e a *implantação* equivale à execução.

Ao solicitar a *preparação* (*build*) da aplicação, o CCT verifica se existem erros de dependências de serviços e eventos. Na Figura 9.13 ilustra-se a visão de exibição de problemas do Eclipse descrevendo um erro e uma advertência relacionada à dependência de serviços. O CCT considera um erro de dependência qualquer serviço requerido que não é provido por outro componente e qualquer evento de interesse não anunciado por outro componente. Advertências ocorrem quando

um evento anunciado não é de interesse de nenhum outro componente ou quando um serviço provido por um componente não é requerido por nenhum outro.

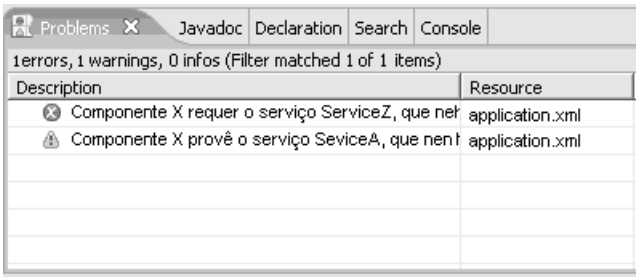



Figura 9.13: Visão de problemas de dependência.

As funcionalidades de *preparação* e *implantação* da ferramenta VDPI estão diretamente relacionada à integração do CCT com os servidores de aplicação. Estas funcionalidades são detalhadas no Capítulo 10, dentro do contexto do ciclo de desenvolvimento de aplicações utilizando o CCT.

Uma vez que a aplicação esteja em execução, diante da necessidade de evoluir a aplicação, com a inserção ou alteração de componentes, por exemplo, pode-se utilizar a ferramenta de verificação formal para verificar se a evolução irá impactar a corretude da especificação do sistema. Como ilustrado na Figura 9.14, na versão atual desta ferramenta, o resultado da verificação através da chamada externa da ferramenta *Alloy Analyzer* é apresentado diretamente no console do Eclipse. Em versões futuras, as informações podem ser interpretadas para dar ao usuário um retorno mais amigável, inclusive, apontando os componentes e o cenário em que as propriedades do sistema não são satisfeitas.

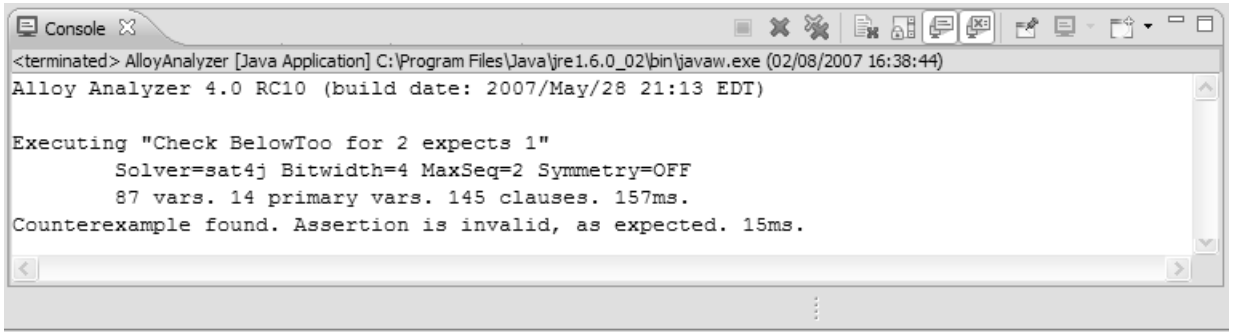



Figura 9.14: Visão de verificação formal.

Como forma de ilustrar o conjunto de ferramentas disponibilizadas no CCT, na Figura 9.15

ilustra-se a perspectiva do CCT. Nesta perspectiva, as visões relacionadas às ferramentas descritas anteriormente estão reunidas.

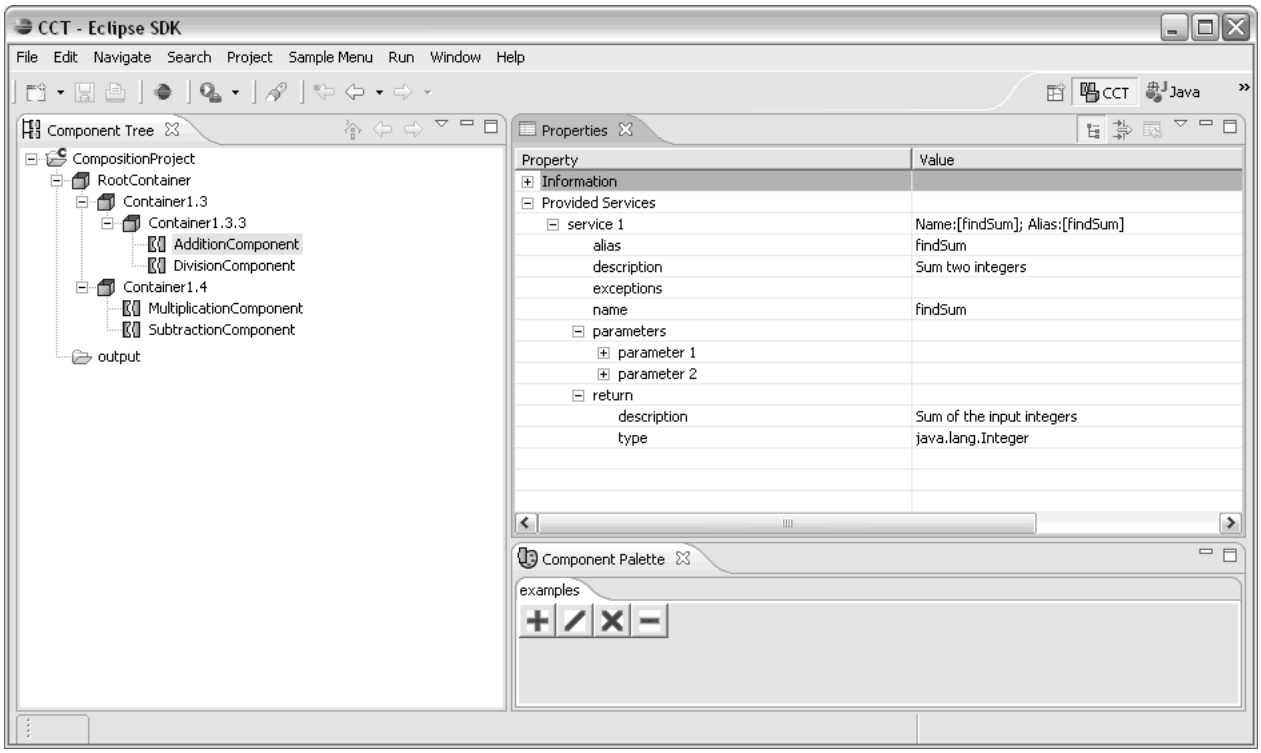



Figura 9.15: Perspectiva de composição de aplicações.

## 9.3 Considerações Finais

Neste capítulo foram apresentados os ambientes de desenvolvimento de componentes (CDE) e composição de aplicações (CCT) baseadas na CMS. Inicialmente, discute-se a arquitetura do CDE, detalhando seus módulos e como utilizar a sua interface. Vale ressaltar que algumas das ferramentas do CDE, tais como o editor de componentes, são dependentes de linguagem e, atualmente, tem-se apenas a implementação das mesmas para aplicações desenvolvidas em Java, ou seja, utilizando o JCF.

Após a descrição do CDE, apresenta-se a arquitetura do CCT, detalhando seus módulos e como utilizar as suas diversas ferramentas, incluindo composição de aplicações, análise de desempenho, verificação formal e implantação de componentes, sendo esta última integrada com os ambientes de execução discutidos no Capítulo 10. Ao contrário do que ocorre com o CDE,

todas as ferramentas do CCT são independentes da linguagem na qual foram implementados os componentes da aplicação.

A grande motivação para o desenvolvimento do CDE e do CCT é melhorar o suporte provido ao desenvolvedor de aplicações baseadas na CMS. Apenas a utilização dos arcabouços de componentes discutidos no Capítulo 5 torna difícil a gerência de dependências entre os componentes, empacotamento dos mesmos para reutilização e composição de aplicações no formato esperado pelos ambientes de execução. O CDE e o CCT disponibilizam uma interface gráfica amigável para automatizar estas tarefas, permitindo ao desenvolvedor se concentrar nas atividades específicas do desenvolvimento de sua aplicação.

Os ambientes de desenvolvimento e composição estão sendo implementados em cooperação com vários desenvolvedores no contexto de dois projetos de pesquisa [61, 62] do CNPq e em trabalhos de conclusão de curso de graduação [70, 71, 72]. O código fonte e maiores informações sobre o andamento do desenvolvimento podem ser obtidos nos sites dos ambientes nos endereços `http://gforge.embedded.ufcg.edu.br/projects/cde` e `http://gforge.embedded.ufcg.edu.br/projects/ccf`.

# Capítulo 10

# Ambientes de Execução

Como apresentado no Capítulo 9, a funcionalidade do CCT está restrita à composição de aplicações e não trata da execução das mesmas. Para a execução das aplicações baseadas na CMS, utiliza-se o conceito de servidores de aplicação. Um *Component Application Server* (CAS) é um software dependente de linguagem que gerencia a execução de aplicações baseadas em componentes. Os componentes e aplicações criados utilizando o CCT são implantados em máquinas distribuídas que estão executando um CAS. A seguir, apresenta-se a arquitetura genérica do CAS, a integração do CAS com o CCT e, por fim, a implementação da arquitetura em Java, denominada JCAS.

## 10.1 Arquitetura dos CASs

A arquitetura geral de um CAS é ilustrada na Figura 10.1. Um CAS é composto de dois módulos principais: cliente e servidor. O cliente é o módulo de desenvolvimento que envia componentes para os vários módulos servidores, os quais executam a aplicação através da inicialização da execução dos componentes. Cada um dos módulos é descrito a seguir.

### 10.1.1 CAS Cliente

O CAS cliente é formado por três aplicações simples: uma para *preparação para implantação* (*build*); uma para *implantação* dos componentes da aplicação (*deploy*); e outra para a *evolução* da aplicação (*evolve*). Os parâmetros de entrada para estas aplicações são o descritor XML da

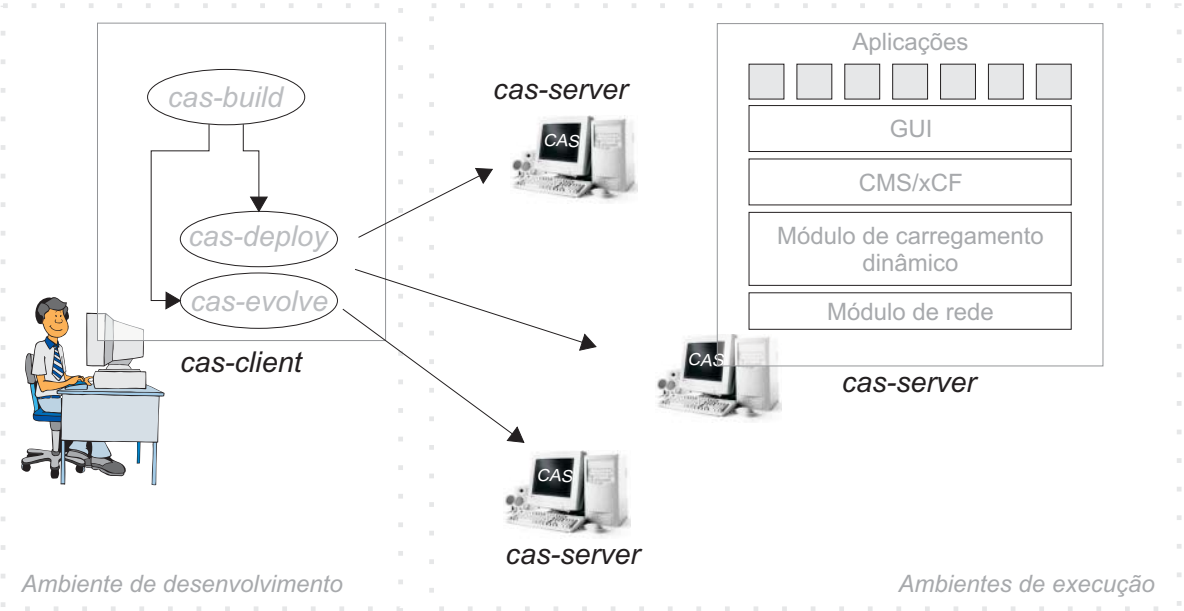



Figura 10.1: Arquitetura geral de um CAS.

aplicação (*application.xml*) e os arquivos dos componentes. A aplicação para *evolução* ainda recebe como parâmetro o conjunto de alterações do projeto que levaram à evolução. O arquivo XML representa o elo de ligação entre o CCT e o CAS, sendo assim, as aplicações se tornam independentes possibilitando a utilização do CAS mesmo quando não se utiliza o CCT.

A aplicação *build* realiza um pré-processamento nos componentes da aplicação para adequá-los ao ambiente de execução alvo. Dependendo da linguagem, este passo pode não ser necessário. No caso de Java, ele foi imprescindível, como descrito posteriormente.

Já a aplicação *deploy* é responsável por estabelecer comunicação com os nós de rede que estão executando o CAS servidor e enviar os componentes e informações da aplicação para os mesmos. O endereço de cada componente é definido no próprio XML da aplicação. Os componentes para um mesmo endereço são agrupados e enviados conjuntamente para diminuir a quantidade de requisições aos servidores. Após esta etapa, os servidores de aplicação estão prontos para iniciar os componentes e, assim, as aplicações.

Por fim, a aplicação *evolve* funciona da mesma forma que a aplicação *deploy*, mas só pode ser utilizada para evoluir aplicações já implantadas. Além dos novos componentes e do novo XML da aplicação, esta aplicação envia aos servidores uma lista de alterações que levaram à evolução, seguindo a linguagem de *script* definida a seguir. Os comandos presentes na lista serão

interpretados pelos servidores de aplicação que realizarão as mudanças, na ordem indicada no arquivo de alterações, dinamicamente.

**ADD <parent-path> <component-file>,** onde <parent-path> é o caminho completo do contêiner pai do componente a ser adicionado e <component-file> é o arquivo referente ao componente.

**REMOVE <component-path>,** onde <component-path> é o caminho completo do componente a ser removido.

**SET_ALIAS_EVENT_OF_INTEREST <component-path> <event-name> <new-alias>,** . onde <component-path> é o caminho completo do componente; <event-name> é o nome do evento de interesse cujo apelido será alterado; e <new-alias> é o novo apelido do evento de interesse.

**SET_ALIAS_ANNOUNCED_EVENT <component-path> <event-name> <new-alias>,** . onde <component-path> é o caminho completo do componente; <event-name> é o nome do evento anunciado cujo apelido será alterado; e <new-alias> é o novo apelido do evento anunciado.

**SET_ALIAS_PROVIDED_SERVICE <component-path> <service-name> <new-alias>,** . onde <component-path> é o caminho completo do componente; <service-name> é o nome do serviço provido cujo apelido será alterado; e <new-alias> é o novo apelido do serviço provido.

**SET_ALIAS_REQUIRED_SERVICE <component-path> <service-name> <new-alias>,** . onde <component-path> é o caminho completo do componente; <service-name> é o nome do serviço requerido cujo apelido será alterado; e <new-alias> é o novo apelido do serviço requerido.

**ADD_ADAPTER <component-path> <adapter-path>,** onde <component-path> é o caminho completo do componente a ser adaptado e <adapter-path> é o caminho do arquivo do adaptador.

**REMOVE_ADAPTER <component-path>,** onde <component-path> é o caminho completo do componente cujo adaptador será removido.

**ACTIVATE_CONCERN <component-path> <concern-name>,** onde <component-path> é o caminho completo do componente e <concern-name> é o interesse a ser ativado.

**DEACTIVATE_CONCERN <component-path> <concern-name>,** onde <component-path> é o caminho completo do componente e <concern-name> é o interesse a ser desativado.

### 10.1.2 CAS Servidor

Como ilustrado na Figura 10.1, o CAS servidor é composto por quatro módulos, os quais são descritos a seguir.

- **Módulo de rede –** Este módulo implementa a recepção de componentes e informações sobre as aplicações enviadas através do CAS cliente, mais especificamente, das aplicações *deploy* e *evolve*. Os componentes são armazenados em pastas específicas de acordo com a aplicação a que pertencem. Caso já exista uma aplicação com mesmo nome da que está sendo implantada, trata-se de um cenário de evolução. Em ambos os casos, o módulo de carregamento dinâmico é notificado para que as atualizações necessárias na hierarquia de componentes sejam realizadas.

- **Módulo de carregamento dinâmico –** Este é o principal módulo do CAS servidor. Ele gerencia o carregamento e evolução dinâmica dos componentes implantados no CAS servidor. Para a implementação deste módulo é necessária a utilização de um mecanismo de carregamento dinâmico de classes.

- **CMS/xCF –** Arcabouço que implementa a CMS. O arcabouço é dependente de linguagem e funciona como a infra-estrutura de execução das aplicações baseadas na CMS.

- **GUI –** Interface gráfica do usuário (*Graphical User Interface*) do CAS servidor. Através desta interface é possível iniciar e interromper a execução de aplicações implantadas em um determinado CAS. Além disso, pode-se visualizar as informações referentes aos componentes implantados para cada aplicação, seus serviços e eventos.

## 10.2 Integração CCT-CAS

Na Figura 10.2, ilustra-se a arquitetura de integração entre o CCT e o CAS. Os passos inerentes ao cenário de integração são descritos a seguir.

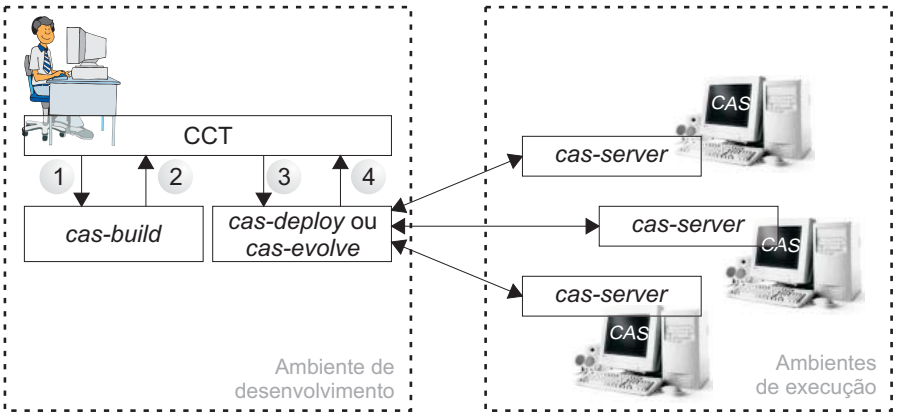



Figura 10.2: Integração CCT-CAS.

1. Quando o desenvolvedor requisita a preparação para a implantação através da interface gráfica do CCT, o CCT faz uma chamada de processo à aplicação cliente *cas-build*. No contexto do CCT, esta atividade representa a *preparação para implantação*.

2. O resultado deste processo é armazenado no diretório de saída definido no próprio CCT. Para o usuário do CCT, este processo é transparente. O local onde se encontra a aplicação *cas-build* é configurável via interface do CCT. Sendo assim, para direcionar este processo para um *cas-build* em uma linguagem diferente basta alterar o local da aplicação *cas-build* correspondente.

3. Caso o processo de preparação tenha ocorrido sem erros, o desenvolvedor pode solicitar a implantação dos componentes através da interface do CCT. O CCT faz uma chamada de processo à aplicação cliente *cas-deploy*, que estabelece comunicação via rede com os CAS servidores, de acordo com o endereço de cada componente. No caso de evolução de uma aplicação existente, a aplicação cliente *cas-evolve* é utilizada.

4. Os componentes são então enviados entre os CAS cliente e servidores e o resultado do processo é retornado ao desenvolvedor na interface do CCT. Este processo é transparente

ao desenvolvedor. Assim como o *cas-build*, os locais das aplicações *cas-deploy* e *cas-evolve* são configuráveis via interface do CCT. Logo, torna-se possível alterar a aplicação de implantação via interface gráfica e, conseqüentemente, manter a independência entre CAS e CCT.

## 10.3 Ciclo de Desenvolvimento

Na Figura 10.3, apresenta-se o ciclo de desenvolvimento de software utilizando o CDE, o CCT e o CAS.

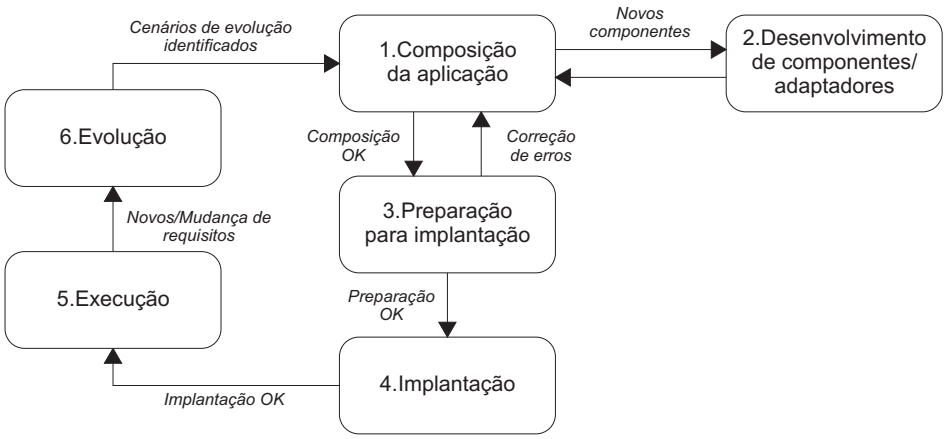



Figura 10.3: Ciclo de desenvolvimento de software utilizando o CDE, o CCT e o CAS.

1. **Composição da aplicação:** a composição é realizada utilizando a interface gráfica do CCT descrita no Capítulo 9. Caso os componentes existentes atualmente na paleta de componentes não sejam suficientes, novos deverão ser desenvolvidos utilizando o CDE, adicionados à paleta de componentes do CCT e então inseridos na aplicação. O resultado da composição da aplicação é a hierarquia de contêineres e componentes da aplicação, com apelidos de serviços e eventos definidos, assim como as dependências entre eles.

2. **Desenvolvimento de componentes/adaptadores:** cada componente deve ser desenvolvido utilizando a ferramenta de edição de componentes/adaptadores do CDE, descrita no Capítulo 9. Após o desenvolvimento, a ferramenta disponibiliza o arquivo *.cmc* referente ao novo componente na pasta relativa à paleta de componentes.

3. **Preparação para implantação:** nesta fase, o CCT é utilizado para a verificação de erros de dependência entre os componentes. Caso existam erros, retorna-se à fase de composição da aplicação. Caso não haja problemas de dependências, o CCT integrado ao CAS realiza a preparação dos componentes para a implantação (*build*). Dependendo da linguagem dos componentes, esta preparação pode não ser necessária. Caso não haja problemas na preparação, passa-se à fase de implantação. O resultado desta fase é um arquivo XML (*application.xml*) contendo todas as informações necessárias para a implantação do software, assim como um conjunto de arquivos de componentes para implantação gerados especificamente para o ambiente de execução alvo.

4. **Implantação:** nesta fase, os componentes são transferidos via rede, através do CAS(*deploy* ou *evolve*), para os ambientes de execução alvo, de acordo com o endereço definido no *application.xml* e configurável via interface gráfica do inspetor do CCT (Figura 10.4). O arquivo da aplicação também é enviado para um CAS e no caso de evolução, a lista de alterações também é enviada. Caso os componentes da aplicação não sejam enviados para o mesmo local do *application.xml*, ou seja, caso a aplicação seja composta por componentes distribuídos, o arcabouço específico de linguagem terá que dar suporte à distribuição.

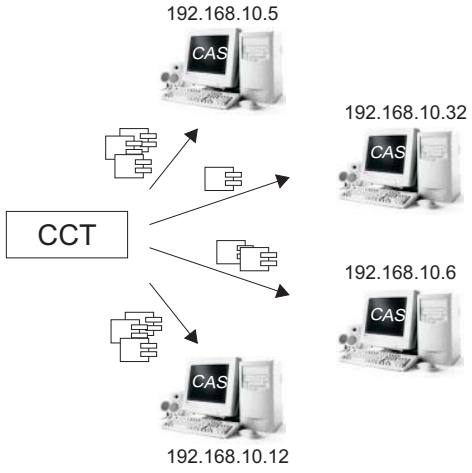



Figura 10.4: Implantação dos componentes para a execução da aplicação.

5. **Execução:** A execução das aplicações é gerenciada pelo CAS. Através dele é possível visualizar informações sobre os componente da aplicação, assim como interromper as aplicações caso necessário. Caso não haja nenhum problema com o computador que esteja exe-

cutando o CAS, a aplicação permanecerá executando por tempo indeterminado. Quando novos requisitos surgirem ou os atuais forem alterados, passa-se à fase de evolução.

6. **Evolução:** Na fase de evolução, tem-se como entrada um conjunto de novos requisitos ou de mudanças de requisitos existentes e como saída um conjunto de cenários de evolução identificados. Cada um destes cenários será mapeado para um ou mais comandos da linguagem de *script* apresentada anteriormente. O ciclo então recomeça, cabendo ao ambiente de composição (CCT) registrar cada uma das alterações sendo feitas na aplicação. Este registro de alterações é repassado ao CAS na fase de implantação. Cada CAS é então responsável pela gerência de evolução dinâmica dos componentes e da aplicação.

## 10.4 *Java Component Application Server* (JCAS)

A versão em Java do CAS, o JCAS, foi desenvolvida sobre a plataforma Oscar [43]. Oscar é uma implementação da especificação OSGi (*Open Service Gateway initiative*), que em linhas gerais é uma tecnologia para carregamento dinâmico de módulos (componentes ou serviços) desenvolvidos em Java [42]. A arquitetura do JCAS é ilustrada na Figura 10.5. O módulo de carregamento de classes é implementado pelo Oscar/OSGi. Apesar do Oscar possuir uma interface gráfica, foi utilizada apenas a sua API para que fosse construída uma interface gráfica específica para o JCAS.

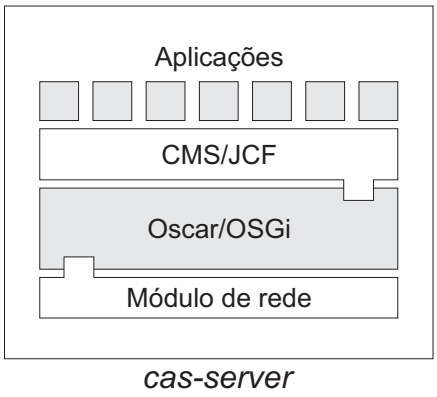



Figura 10.5: Arquitetura do JCAS.

Cada módulo de acordo com OSGi é denominado *bundle*. *Bundles* podem ser carregados dinamicamente e possuem dependências com outros *bundles*. Como forma de manter o encapsulamento entre os módulos da arquitetura, tanto a API do JCF como o *módulo de rede* são também

disponibilizados como *bundles* do Oscar. Os módulos cliente e servidor do JCAS são descritos a seguir.

### 10.4.1 JCAS - Cliente

**Preparação para implantação (*jcas-build*)**

A aplicação *jcas-build* adapta os arquivos de componentes para a implantação e execução pelo JCAS servidor. Como ilustrado na Figura 10.6, a aplicação recebe como parâmetros de entrada o arquivo *application.xml* descrevendo a estrutura da aplicação e a pasta contendo os arquivos dos componentes. O resultado é um conjunto de arquivos contendo *bundles* OSGi.

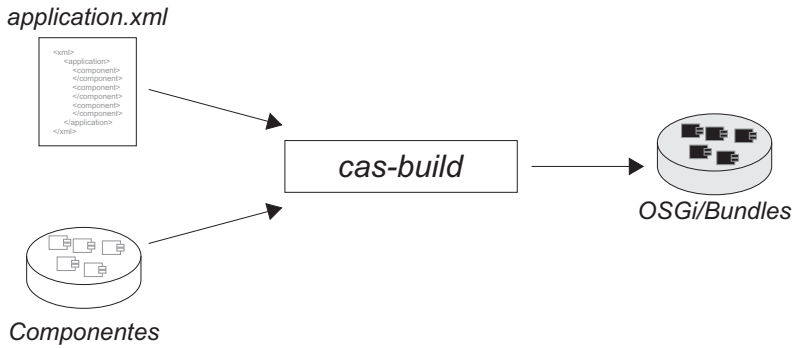



Figura 10.6: Execução da aplicação *jcas-build*.

Este pré-processamento simplifica o carregamento dinâmico dos componentes pelo JCAS servidor, assim como o processo de inicialização e evolução dos componentes e da aplicação. Após a execução do *jcas-build*, os componentes já estão prontos para a implantação. A aplicação é executada via linha de comando, o que permite a integração com o CCT através de chamada de processo. A sintaxe para a execução do *jcas-build* é descrita a seguir.

```
jcas-build <descriptor-path> <component-lib-path> <output-path>,
```

onde: `<descriptor-path>` é o caminho absoluto do descritor da aplicação (*application.xml*); `<component-lib-path>` é o caminho absoluto do diretório contendo os componentes; e, por fim, `<output-path>` é o diretório de saída onde serão armazenados os *bundles* dos componentes gerados pela aplicação.

**Implantação (*jcas-deploy*)**

A aplicação *jcas-deploy* realiza a implantação dos componentes e da aplicação recebendo como parâmetro de entrada o arquivo *application.xml* e pasta contendo os *bundles* gerados pela aplicação *jcas-build*. Como ilustrado na Figura 10.5, cada componente e o *application.xml* são enviados ao JCAS servidor via rede, de acordo com o endereço definido no arquivo *application.xml*. Este mesmo arquivo é enviado para cada um dos servidores nos quais foram implantados componentes. A aplicação também é executada via linha de comando, o que permite integração com o CCT através de chamada de processo. A sintaxe para a execução do *jcas-deploy* é descrita a seguir. Após a execução do *jcas-deploy*, os componentes e as aplicações estão prontos para a inicialização através da interface do JCAS servidor.

```
jcas-deploy <descriptor-path> <bundles-path>,
```

onde: `<descriptor-path>` é o caminho absoluto do descritor da aplicação (*application.xml*) e `<bundles-path>` é o caminho absoluto do diretório contendo os *bundles* gerados pelo *jcas-build*.

**Evolução (*jcas-evolve*)**

A aplicação *jcas-evolve* também realiza a implantação dos componentes e da aplicação recebendo como parâmetro de entrada o arquivo *application.xml* e a pasta contendo os *bundles* gerados pela aplicação *jcas-build*. Porém, neste caso, a implantação só ocorre para aplicações já existentes, ou seja, trata-se de uma evolução. Por isso, mais um parâmetro é recebido como entrada para a aplicação: uma lista de alterações que levaram à evolução, em ordem cronológica, a serem realizadas dinamicamente. A aplicação também é executada via linha de comando, o que permite a integração com o CCT através de chamada de processo. A sintaxe para a execução do *jcas-evolve* é descrita a seguir.

```
jcas-evolve <descriptor-path> <bundles-path> <operations-file>
```

onde: `<descriptor-path>` é o caminho absoluto do descritor da aplicação (*application.xml*), `<bundles-path>` é o caminho absoluto do diretório contendo os *bundles* gerados pelo *jcas-build* e <operations-file> é o caminho absoluto para o arquivo com as alterações a serem contempladas dinamicamente.

### 10.4.2 JCAS - Servidor

A aplicação servidor do JCAS executa como um processo do sistema operacional em *background*. Um ícone é exibido na bandeja da interface do sistema indicando que o JCAS servidor está no ar e que, portanto, esta máquina pode receber aplicações e componentes do JCAS cliente (Figura 10.7).

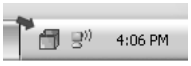



Figura 10.7: JCAS executando em *background*.

Ao clicar sobre o ícone da bandeja, exibe-se a tela principal do JCAS servidor, que é ilustrada na Figura 10.8. Nesta tela, cada aba representa uma aplicação implantada no JCAS, com descrição da aplicação, visualização da sua árvore de componentes e as propriedades dos mesmos. Esta informação é obtida através do arquivo *application.xml* enviado no momento da implantação da aplicação. Através desta interface é possível apenas iniciar ou interromper as aplicações já implantadas. Qualquer operação de implantação de uma nova aplicação ou evolução de uma aplicação existente deve ser realizada através do módulo cliente do JCAS.

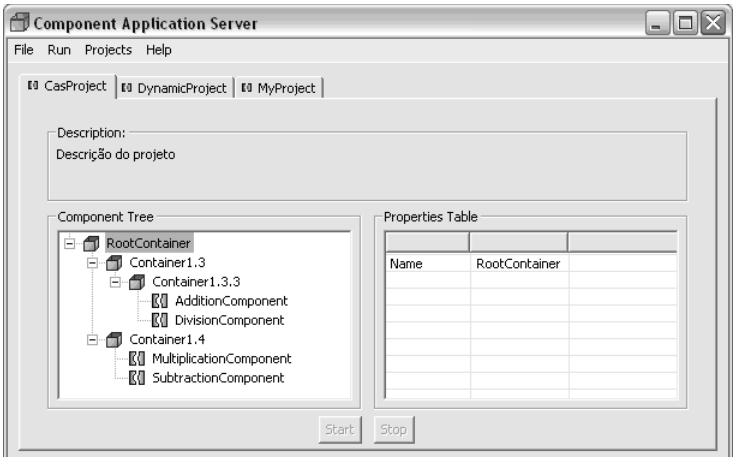



Figura 10.8: Interface gráfica do JCAS.

## 10.5 Considerações Finais

Neste capítulo foi discutido como ocorre a implantação, execução e evolução de aplicações baseadas na CMS e desenvolvidas utilizando os arcabouços de componentes baseados no GCF através de servidores de aplicação de componentes (CAS). Um CAS é um software dependente de linguagem que gerencia a execução de aplicações baseadas na CMS, dando suporte à implantação de componentes e aplicações e gerenciando o ciclo de execução e evolução dos mesmos.

Da maneira como foi concebida sua arquitetura, um CAS é independente da utilização do ambiente de composição descrito no Capítulo 9, ou seja, é possível utilizar qualquer servidor de aplicação sem utilizar o CCT. Evidentemente que, ao utilizar o CCT, o desenvolvedor terá uma maior facilidade para empacotar os componentes e especificar a aplicação no formato esperado pelo CAS. Além disso, existe uma ponte entre os dois ambientes para que o desenvolvedor não precise sair do CCT para realizar atividades de implantação, execução e evolução.

Neste capítulo, inicialmente, a arquitetura geral de um CAS foi apresentada, destacando os principais módulos de software que a compõem, mais especificamente, os módulos cliente e servidor. Descreve-se então como ocorre a integração entre o CCT e o CAS, independente da linguagem utilizada para o desenvolvimento da aplicação. O ciclo de desenvolvimento de aplicações utilizando CCT e CAS é também descrito, incluindo atividades de desenvolvimento, composição, implantação, execução e evolução. Este ciclo de desenvolvimento focado em ferramentas é a base para o processo de desenvolvimento descrito no Capítulo 11. Finalmente, apresenta-se a implementação do CAS para aplicações em Java, denominado JCAS. São descritos os módulos cliente e servidor da aplicação, incluindo implantação de componentes e aplicações Java, assim como execução e evolução destas aplicações.

Com o suporte provido pelo CAS, conclui-se a descrição das ferramentas de software disponibilizadas na infraestrutura proposta nesta tese. O desenvolvedor utiliza o CCT para a composição de aplicações, desenvolvendo componentes utilizando os arcabouços específicos baseados no GCF. Utiliza-se então o CAS para implantar os componentes e as aplicações compostas utilizando o CCT. Estas aplicações são executadas de acordo com o *middleware* implementado pelos arcabouços baseados no GCF. Por fim, a evolução das aplicações pode ser realizada via CCT ou via linha de comando através do módulo CAS cliente refletindo as alterações nas aplicações e componentes implantados no CAS servidor.

Assim como os ambientes de desenvolvimento e composição, o JCAS está sendo implementado em cooperação com vários desenvolvedores no contexto de dois projetos de pesquisa [61, 62] do CNPq. O código fonte e maiores informações sobre o andamento do desenvolvimento podem ser obtidos no site do ambiente no endereço `http://gforge.embedded.ufcg.edu.br/projects/jcas`.

# Capítulo 11

# Processo de Desenvolvimento

Neste capítulo apresenta-se um processo de desenvolvimento de software com suporte à evolução dinâmica não antecipada centrado na infra-estrutura proposta nesta tese. Segundo Sommerville [199], um processo é definido como "o conjunto de atividades que levam à produção de um software".

De uma maneira informal, um processo é um conjunto de regras que definem *quem* faz *o que*, *quando* e *como* para se desenvolver um software. Mais especificamente, tem-se respectivamente: a definição dos papéis ou atores envolvidos (programador, projetista, analista, etc); as atividades exercidas e artefatos gerados por cada um dos atores (análise, codificação, testes, etc); a seqüência em que estas atividades ocorrem; e um detalhamento de como cada uma das atividades deve ser desempenhada.

O processo aqui proposto é fundamentado nas características gerais do processo de desenvolvimento baseado em componentes e não está diretamente acoplado a nenhum processo mais específico, tal como *Rational Unified Process* (RUP) [200], *eXtreme Programming* (XP) [201], *Test-Driven Development* (TDD) [202], etc. Além disso, também não está acoplado a métodos de desenvolvimento baseado em componentes, tais como Catalysis [203] e UML Components [204].

A seguir, são apresentados os papéis envolvidos no processo de desenvolvimento, as atividades desempenhadas por eles, o ciclo de desenvolvimento e um detalhamento da execução de tais atividades. Não se trata de um processo rígido e sim de um conjunto de diretrizes a serem seguidas e adaptadas de acordo com a necessidade da equipe e requisitos do sistema. Para cada diretriz, faz-se uma referência ao suporte ferramental provido pela infra-estrutura proposta nesta tese, caso

este exista. Espera-se com tais diretrizes guiar o desenvolvedor na aplicação de tal infra-estrutura.

## 11.1 Papéis

Os seguintes papéis são definidos no processo de desenvolvimento de software com suporte à evolução dinâmica não antecipada. Evidentemente que a lista não é exaustiva e vai depender do tamanho da equipe, estrutura da organização e características do software. Uma lista mais completa de papéis relacionados aos diversos aspectos de um processo de desenvolvimento de software pode ser encontrada em [200]. Vale ressaltar que vários papéis podem ser desempenhados por um único indivíduo assim como vários indivíduos podem desempenhar um mesmo papel.

### Ciclo de Desenvolvimento de Componentes

Os papéis descritos abaixo estão relacionados ao ciclo de desenvolvimento de componentes.

- **Analista de Domínio -** Responsável pela análise do domínio de aplicação independente de uma aplicação específica. O analista de domínio deve identificar as funcionalidades relevantes em um dado domínio para que sejam posteriormente implementadas e disponibilizadas por componentes, os quais serão reutilizados para aumentar a produtividade no ciclo de desenvolvimento de aplicações. Não há suporte ferramental da infra-estrutura para o indivíduo que desempenha este papel.

- **Analista Formal de Componentes -** Responsável por especificar, utilizando a linguagem formal Alloy, as propriedades inerentes a um dado componente que devem ser verificadas durante a composição de aplicações, seguindo a técnica apresentada no Capítulo 7.

- **Projetista de Componentes -** Responsável pelo projeto arquitetural e de baixo nível do componente sendo desenvolvido e também pela definição dos serviços, eventos e interesses a serem disponibilizados no mesmo. Não há suporte ferramental da infra-estrutura para o indivíduo que desempenha este papel.

- **Programador de Componentes -** Responsável pela implementação do componente. Para isso, deve-se utilizar a especificação do GCF e uma implementação em linguagem específica (JCF, CCF, PyCF ou SCF), tal como descrito no Capítulo 5. Além disso, tem-se o

suporte do ambiente de desenvolvimento de componentes, CDE, que também dá suporte à disponibilização de componentes no repositório, como apresentado no Capítulo 9.

- **Testador de Componentes -** Responsável por realizar os testes do componente. Após testado, o componente é disponibilizado no repositório de componentes. Não há suporte ferramental da infra-estrutura específico para testes de componentes. Porém, pode-se utilizar o CDE juntamente com um arcabouço de testes xUnit, como JUnit [205] para Java, por exemplo.

- **Gerente de Repositório -** Responsável por reunir e disponibilizar a documentação do sistema, de acordo com os requisitos da organização, cliente ou do próprio componente. Além disso, é responsável por disponibilizar o componente no repositório, juntamente com as informações relevantes para facilitar a sua posterior reutilização. Para a disponibilização, pode-se utilizar o CDE. Não há suporte ferramental da infra-estrutura para a documentação.

## Ciclo de Desenvolvimento de Aplicações

Os papéis descritos abaixo estão relacionados ao ciclo de desenvolvimento com componentes, ou seja, de aplicações.

- **Analista de Requisitos -** Responsável por identificar os requisitos do sistema junto ao cliente, os quais serão posteriormente implementados como serviços providos e eventos anunciados por componentes. Isto é válido também para cenários de evolução, onde novos requisitos não previstos inicialmente são identificados. Não há suporte ferramental da infra-estrutura para o indivíduo que desempenha este papel.

- **Analista Formal de Aplicações -** Responsável por especificar, utilizando a linguagem formal Alloy, as propriedades inerentes à aplicação que devem ser verificadas durante a evolução da aplicação através da inserção, remoção ou alteração de um componente, como descrito no Capítulo 7. Também é responsável por analisar o impacto de um cenário de evolução sobre a corretude da especificação, para isso, pode-se utilizar a ferramenta de análise descrita no Capítulo 9.

- **Projetista de Aplicações -** Responsável pelo projeto arquitetural da aplicação, definindo a hierarquia de contêineres e identificando os componentes funcionais para implementar os requisitos definidos pelo analista, de acordo com a CMS e considerando desempenho e flexibilidade, seguindo o modelo analítico apresentado no Capítulo 8. É também responsável pelo modelo conceitual da aplicação, identificando possíveis adaptações de componentes a serem realizadas para que a aplicação funcione corretamente.

- **Programador de Aplicações -** Responsável pela implementação de *scripts* de execução, adaptadores e componentes personalizados, de acordo com a CMS. Para isso, utiliza-se o CDE e uma implementação de arcabouço em linguagem específica, como o JCF, por exemplo. Caso haja algum componente a ser implementado, esta tarefa deve ser realizada pelo *Programador de Componentes*.

- **Testador de Aplicações -** Responsável por realizar os testes de integração entre os componentes da aplicação. Para isso, pode-se utilizar o suporte à verificação de dependências disponibilizado na ferramenta de composição, o CCT, apresentada no Capítulo 9.

- **Gerente de Implantação -** Responsável por gerenciar a implantação de componentes e aplicações, utilizando os servidores de aplicação apresentados no Capítulo 10.

- **Gerente de Evolução -** Responsável por, com base nos requisitos de evolução, identificar o impacto da evolução de acordo com os cenários apresentados no Capítulo 4 e gerenciar a nova iteração do ciclo de desenvolvimento.

## 11.2 Ciclo de Desenvolvimento e suas Atividades

As atividades descritas a seguir são definidas no processo de desenvolvimento de software com suporte à evolução dinâmica não antecipada. Estas atividades são executadas de acordo com o ciclo de desenvolvimento ilustrado na Figura 11.1. Vale ressaltar que, de acordo com o processo específico a ser utilizado, atividades de teste podem preceder atividades de programação (ex.: processos ágeis), assim como atividades de documentação podem preceder novas iterações no ciclo (ex.: processos preditivos). De qualquer forma, tem-se na Figura 11.1 a seqüência clássica

de desenvolvimento baseado em componentes com adaptações para o processo proposto nesta tese.

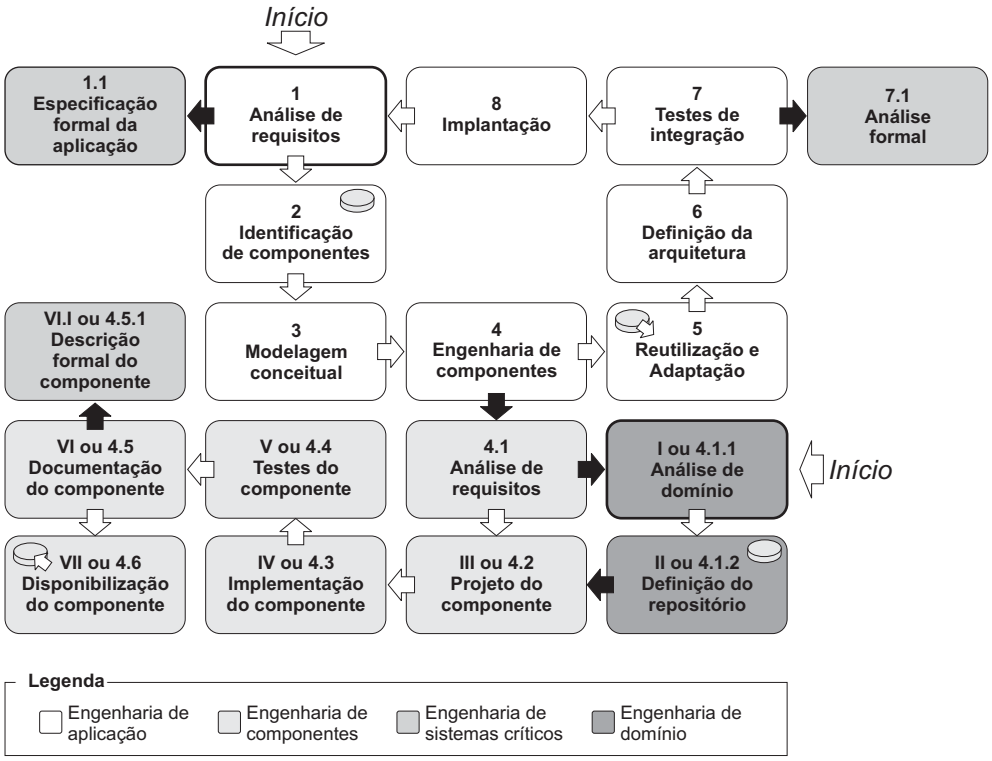



Figura 11.1: Ciclo de desenvolvimento de software.

## Ciclo de Desenvolvimento de Componentes

### Engenharia de Domínio

Considerando o ciclo clássico de desenvolvimento baseado em componentes, tem-se uma fase de Engenharia de Domínio que objetiva a concepção de um repositório de componentes cujas funcionalidades são consideradas relevantes para tal domínio. Por exemplo, uma engenharia do domínio de aplicações corporativas tem como objetivo a identificação de componentes relevantes, tais como persistência, geração de relatórios, transações, dentre outros. Após o desenvolvimento destes componentes, os quais são disponibilizados em um repositório, tem-se um conjunto pré-concebido de partes de aplicações naquele domínio, reduzindo o custo e o tempo de produção de uma nova aplicação, uma vez que várias partes podem ser reutilizadas.

A fase de engenharia de domínio deve ser a fase inicial do desenvolvimento baseado em com-

ponentes mas, uma vez que funciona de forma independente da engenharia de aplicações, pode ser executada, por demanda, sempre que uma dada funcionalidade necessária à aplicação não esteja disponível no repositório. Neste caso, volta-se novamente à engenharia de domínio para que a funcionalidade seja analisada e um novo componente seja definido, enriquecendo cada vez mais o repositório. É por este motivo que as fases a seguir possuem duas numerações: uma indicando o ciclo inicial independente e a outra indicando o ciclo por demanda.

- **I ou 4.1.1: Análise de Domínio -** Nesta fase são identificadas as funcionalidades inerentes a um dado domínio que são relevantes para uma aplicação naquele domínio. Para isso, podem ser utilizados diversos métodos, tais como os clássicos FODA [206], ODM [207], DSSA [208], KBSE [209] e DODE [210]. Como resultado desta fase, tem-se uma definição de que funcionalidades serão potencialmente reutilizadas em novas aplicações no domínio sendo analisado. Esta atividade é desempenhada pelo *Analista de Domínio* e não há suporte ferramental da infra-estrutura para a mesma.

- **II ou 4.2.2: Definição do Repositório -** Uma vez identificadas as funcionalidades inerentes ao domínio de aplicação, pode-se definir os componentes que irão compor o repositório. Como resultado desta fase, tem-se uma definição de que componentes estarão disponíveis no repositório no fim do ciclo corrente de desenvolvimento. Esta atividade é desempenhada pelo *Analista de Domínio* e não há suporte ferramental da infra-estrutura para a mesma.

**Engenharia de Componentes**

Da mesma forma que a Engenharia de Domínio, a Engenharia de Componentes possui um ciclo independente daquele da aplicação propriamente dita. Sendo assim, também possui duas numerações: uma para o ciclo inicial independente e outra para a engenharia de componentes por demanda das aplicações. Em resumo, a Engenharia de Componentes, cujas atividades são descritas a seguir, tem como objetivo o desenvolvimento de componentes de acordo com a definição do repositório realizada nas atividades de Engenharia de Domínio ou com os requisitos específicos demandados pela aplicação.

- **4.1: Análise de Requisitos do Componente -** Nesta atividade, tem-se como objetivo a análise dos requisitos de um componente com base na necessidade da aplicação. Neste

momento, deve-se definir se o componente é realmente específico da aplicação ou se pode ser utilizado por outras do mesmo domínio, o que neste caso levaria o processo à atividade de *Análise de Domínio*. No caso de ser específico da aplicação, tem-se como resultado desta atividade o conjunto de requisitos a serem implementados e disponibilizados pelo novo componente. Esta atividade é realizada pelo *Analista de Requisitos* e não há suporte ferramental da infra-estrutura para a mesma.

- **III ou 4.2: Projeto do Componente -** Nesta atividade, realiza-se o projeto arquitetural e de baixo nível do componente. Com base nos requisitos elencados via análise de requisitos por demanda da aplicação ou via análise de domínio, devem ser definidos os serviços, eventos e interesses do componente, de acordo com a CMS. Esta tarefa é desempenhada pelo *Projetista de Componentes* e não há suporte ferramental da infra-estrutura para a mesma.

- **IV ou 4.3: Implementação do Componente -** Nesta atividade, realiza-se a implementação do componente projetado anteriormente. Para isso, deve-se utilizar a especificação do GCF e uma implementação em linguagem específica (JCF, CCF, PyCF ou SCF). Além disso, tem-se o suporte de desenvolvimento do CDE. Esta atividade é desempenhada pelo *Programador de Componentes*.

- **V ou 4.4: Testes do Componente -** Nesta atividade, são realizados os testes do componente implementado anteriormente. Para isso, pode-se utilizar um arcabouço de testes do tipo xUnit, integrado ao CDE. Porém, não há suporte específico para testes de componentes no CDE. Esta tarefa é realizada pelo *Testador de Componentes*.

- **VI ou 4.5: Documentação do Componente -** Nesta atividade, que envolve vários papéis, descreve-se a documentação do componente. A documentação de projeto é de responsabilidade do *Projetista de Componentes*, a documentação de código é realizada pelo *Programador de Componentes* e os requisitos do componente podem ser descritos pelo *Analista de Requisitos*. Contudo, o responsável principal pela revisão e disponibilização da documentação é o *Gerente de Repositório*. Não há suporte ferramental da infra-estrutura para esta atividade.

- **VII ou 4.6: Disponibilização do Componente -** Nesta atividade, realiza-se a disponibilização do componente no repositório. Para isso, pode-se utilizar a ferramenta de disponibi-

lização de componentes do CDE. Esta tarefa é desempenhada pelo *Gerente de Repositório*. Após a disponibilização, o componente estará pronto para ser reutilizado em diversas aplicações.

#### Engenharia de Sistemas Críticos

A confiança no funcionamento de um sistema de acordo com sua especificação é um requisito desejável em qualquer aplicação. Contudo, este tipo de requisito é mandatório no contexto de sistemas críticos, onde o mau funcionamento pode acarretar perdas financeiras ou risco de vida. Por isso, esta atividade está sendo descrita separadamente, pois, de acordo com a demanda de confiança no funcionamento de uma dada aplicação, pode-se ou não considerar tal atividade no processo de desenvolvimento.

- **VI.I ou 4.5.1: Descrição Formal do Componente -** Nesta atividade, deve-se descrever formalmente as propriedades referentes ao componente que devem ser verificadas durante a composição de aplicações. Para isso, deve-se utilizar a técnica para a especificação e verificação formal descrita no Capítulo 7. Esta atividade é desempenhada pelo *Analista Formal de Componentes*.

## Ciclo de Desenvolvimento de Aplicações

#### Engenharia de Aplicações

A Engenharia de Aplicações é mais voltada para atividades de composição de aplicações. Atividades de desenvolvimento neste contexto são apenas para adaptações e personalizações necessárias aos componentes reutilizados, assim como para a definição do *script* de execução da aplicação.

- **1: Análise de Requisitos da Aplicação -** Nesta atividade, realiza-se a análise dos requisitos a serem contemplados pela aplicação. O resultado é o conjunto de requisitos funcionais e não funcionais que devem ser implementados pelos componentes da aplicação. Esta tarefa é desempenhada pelo *Analista de Requisitos* e não há suporte ferramental da infra-estrutura para a mesma.
- **2: Identificação de Componentes -** Nesta atividade deve-se realizar a identificação dos componentes do repositório de componentes que contemplam os requisitos elencados na

atividade anterior. Esta tarefa é realizada pelo *Projetista da Aplicação*, que pode utilizar a paleta de componentes e o inspetor de propriedades do CCT para definir que componentes reutilizar.

- **3: Modelagem Conceitual -** Nesta atividade, que também é de responsabilidade do *Projetista da Aplicação*, realiza-se a modelagem conceitual da aplicação. Mais especificamente, deve-se definir as entidades de negócio da aplicação de acordo com os componentes reutilizados. Não há suporte ferramental da infra-estrutura para esta atividade.

- **4: Engenharia de Componentes -** Esta atividade representa a interseção entre os ciclos de desenvolvimento de aplicações e o ciclo de desenvolvimento de componentes. Caso haja alguma funcionalidade requerida pela aplicação que não seja contemplada por nenhum componente do repositório, deve-se desenvolver um novo componente. Esta tarefa então engloba todas as fases do ciclo de desenvolvimento de componentes mencionado anteriormente, tendo como resultado a disponibilização do novo componente no repositório.

- **5: Reutilização e Adaptação -** Uma vez que os componentes necessários para a composição da aplicação já estão todos no repositório, esta tarefa diz respeito à reutilização e adaptação dos componentes. Para isso, faz-se uso do mecanismos de adaptação e personalização da CMS/GCF e dos ambientes CDE e CCT. Esta tarefa é realizada pelo *Programador de Aplicações* em conjunto com o *Projetista de Aplicações*, sendo o último responsável por adequar as adaptações ao projeto e modelo conceitual previamente definidos.

- **6: Definição da Arquitetura -** Uma vez desenvolvidos os componentes da aplicação, deve-se definir a melhor arquitetura para a mesma. A hierarquia de contêineres e componentes é então definida levando-se em conta coesão funcional, flexibilidade e desempenho. Para isso, pode-se utilizar o modelo analítico proposto no Capítulo 8 e a ferramenta de análise apresentada em Capítulo 9. Esta tarefa também é desempenhada pelo *Projetista de Aplicações*.

- **7: Testes de Integração -** Nesta tarefa, são realizados os testes de integração entre os componentes da aplicação. Para isso, pode-se utilizar a ferramenta de testes do CCT. Esta tarefa é desempenhada pelo *Testador de Aplicações*.

- **8: Implantação -** Uma vez composta e testada a aplicação, deve-se implantar a mesma e seus componentes nos servidores de aplicação (CAS). O mesmo ocorre para o ciclo de evolução, seguindo as diretrizes de utilização dos CAS apresentadas no Capítulo 10.

**Engenharia de Sistemas Críticos**

Como descrito anteriormente, as atividades abaixo devem ser realizadas de acordo com a demanda de confiança no funcionamento de uma dada aplicação. Em especial, as tarefas relativas à especificação e análise formal de aplicações dependem da atividade de descrição formal de propriedades dos componentes. Sem as descrições dos componentes não é possível verificar se um dado cenário de evolução vai de encontro à corretude da especificação da aplicação.

- **1.1: Especificação Formal da Aplicação -** Nesta atividade, deve-se descrever formalmente as propriedades referentes à aplicação que devem ser verificadas diante de cenários de evolução, em especial, inserção, remoção e alteração de componentes. Para isso, deve-se utilizar a técnica para a especificação formal descrita no Capítulo 7. Esta atividade é desempenhada pelo *Analista Formal de Aplicações*.

- **7.1: Análise Formal -** Nesta atividade, verifica-se se um dado cenário de evolução vai de encontro à corretude da especificação do sistema. Para isso, utiliza-se a técnica para verificação formal descrita no Capítulo 7 e a ferramenta de verificação formal do CCT. Esta atividade também é desempenhada pelo *Analista Formal de Aplicações* e é complementar aos testes de integração de componentes.

## 11.3 Considerações Finais

Neste capítulo foi apresentado um processo para o desenvolvimento de software com suporte à evolução dinâmica não antecipada. O processo é centrado nos arcabouços e ferramentas disponibilizados pela infra-estrutura proposta nesta tese. São descritos os papéis envolvidos no processo, assim como as atividades relativas ao ciclo de desenvolvimento de componente e aplicações.

É importante ressaltar que o processo proposto não é rígido. Seu ciclo de desenvolvimento determina apenas uma possível seqüência a ser seguida para a concepção do software. Considerando abordagens preditivas, por exemplo, pode-se ter um foco maior em análise e projeto, do que

aquele mencionado neste capítulo. Já no caso de métodos ágeis, deve-se inserir uma atividade de refatoramento [211] e as atividades de testes poderiam ser realizadas antes da programação dos componentes e da montagem da aplicação.

Artefatos de documentação resultantes de cada atividade também não foram definidos. Isto foi feito porque, de acordo com o processo utilizado, pode-se ter uma carga diferente de documentação. Sendo assim, pode-se utilizar a notação que o desenvolvedor considerar mais adequada. Em [76], utiliza-se uma adaptação deste processo para utilização com *Catalysis* [203] e *UML Components* [204].

Com este capítulo, encerra-se a descrição da infra-estrutura proposta nesta tese. O processo serve como um guia para a utilização da infra-estrutura, fundamentado na utilização do modelo de componentes, arcabouços e ferramentas apresentadas nos capítulos anteriores.

# Capítulo 12

# Aplicações Desenvolvidas

Neste capítulo são apresentadas as aplicações desenvolvidas utilizando a infra-estrutura proposta neste trabalho. As aplicações foram desenvolvidas em cooperação com desenvolvedores no contexto de dissertações de mestrado [67, 68]. A idéia foi interferir o mínimo possível sobre as decisões de projeto de cada uma delas, como forma de avaliar a qualidade das APIs dos arcabouços de componentes.

Uma vez que o arcabouço Java foi o primeiro a ser desenvolvido, as aplicações mais complexas foram desenvolvidas utilizando o JCF. Nas outras linguagens, apenas aplicações simples foram desenvolvidas (*toy examples*), as quais estão disponíveis juntamente com as APIs dos arcabouços descritos no Apêndice B. A seguir, são descritas aplicações nos domínios de computação pervasiva, comunidades virtuais móveis e sistemas multi-agentes abertos. Estes domínios foram escolhidos devido às características de dinamicidade e imprevisibilidade inerentes aos mesmos, o que demanda uma infra-estrutura para evolução dinâmica não antecipada.

## 12.1 *Wings*

Em 1991, Mark Weiser descreveu as bases do que hoje é denominado Computação Pervasiva [212]. De um modo geral, pode-se dizer que Weiser vislumbrou um mundo onde a computação está presente nos objetos do dia-a-dia, os quais interagem entre si para realizar tarefas sem a intervenção direta dos usuários [213].

Contudo, tanto as tecnologias de hardware quanto as de software necessárias para a implanta-

ção de ambientes pervasivos ainda não estavam disponíveis naquela época. Hoje em dia, porém, com os avanços dos dispositivos eletrônicos e tecnologias sem fio, tornou-se possível o desenvolvimento das aplicações imaginadas por Weiser. Estes avanços despertaram o interesse da indústria e da comunidade científica para o campo da Computação Pervasiva, com diversas soluções propostas para tornar a visão de Weiser uma realidade [214, 215, 216, 217, 218].

Um dos principais objetivos da computação pervasiva é tornar-se "invisível" aos usuários [214]. Para isto, aplicações embutidas em dispositivos eletrônicos devem ser pró-ativas, identificando as necessidades dos usuários e provendo aos mesmos informação e recursos relevantes. Embora algumas informações possam ser obtidas a partir do próprio dispositivo do usuário, tais como seus contatos da agenda, outras só podem ser adquiridas através da interação com outros dispositivos.

Para que esta interação ocorra, um dispositivo deve primeiro descobrir os outros dispositivos que estão ao seu redor. Este mecanismo de descoberta é contemplado em protocolos de rede tais como *UPnP* (*Universal Plug and Play*) [219] e *JXTA* [220], dentre outros. Estes protocolos podem ser implementados utilizando diferentes tecnologias de comunicação em rede sem fio. Sendo assim, ambientes pervasivos podem ser implantados utilizando diferentes configurações de tecnologia de comunicação e protocolo de rede. Por exemplo, pode-se utilizar *UPnP* sobre *Wi-Fi* [221] e *JXTA* sobre *Bluetooth* [222].

Contudo, dispositivos móveis como *smart phones* e *handhelds* introduzem uma característica chave para a implantação de ambientes pervasivos: mobilidade. Considerando que os dispositivos estão em movimento, eles podem acessar diferentes ambientes, com diferentes características, protocolos e tecnologias de comunicação. Contudo, é muito difícil prever quais protocolos serão necessários para uma aplicação, considerando todos os ambientes acessados pelos dispositivos móveis.

Além disso, dispositivos móveis possuem uma capacidade de memória, processamento e armazenamento muito limitada. Sendo assim, torna-se inviável embutir em uma aplicação todos os protocolos e tecnologias de rede sem fio existentes. Sendo assim, é necessário um mecanismo que permita a inserção e remoção de tais protocolos e tecnologias de comunicação sob demanda, sempre que necessário.

O mesmo raciocínio é aplicado aos mecanismos de aquisição de informação sobre o ambiente pervasivo, o *contexto* [223]. Dependendo das características do ambiente, a informação de contexto pode ser recuperada e tratada de formas diferentes, o que demanda uma aquisição por

demanda de um módulo específico para tratar a informação de contexto de cada ambiente [224].

É neste contexto que se insere a primeira aplicação desenvolvida com a abordagem proposta nesta tese. Trata-se de uma infra-estrutura para provisão de serviços para computação pervasiva, denominada *Wings* [213, 224, 68]. *Wings* provê mecanismos para o desenvolvimento de aplicações pervasivas, com suporte a redes heterogêneas e à inserção/remoção de componentes de aquisição de contexto sob demanda.

A implementação do *Wings* serviu como pontapé inicial para o refinamento da CMS e da primeira versão do JCF. Ele foi implementado em J2ME [225], perfil CDC (*Connected Device Configuration*). Quando o *Wings* foi desenvolvido, apenas o JCF estava implementado e não havia suporte para o desenvolvimento dos componentes assim como para a composição e a execução da aplicação. Desta forma, as dificuldades no desenvolvimento do *Wings* foram as principais motivações para o investimento no CDE, CCT e CAS.

A seguir são descritos a arquitetura e um cenário de funcionamento do *Wings*, destacando a especificação desta arquitetura com base na CMS. Detalhes de projeto de baixo nível, implementação e código fonte podem ser encontrados em [68] e no site do *Wings* (`http://stage.compor.net/applications/wings`).

### 12.1.1 Arquitetura do *Wings*

A arquitetura do *Wings* se baseia na CMS, o que garante que seus componentes possam ser adicionados, removidos e alterados em tempo de execução [224]. Tal arquitetura está relacionada a três conceitos básicos:

- **Contexto:** encapsula informação usada por aplicações pervasivas para melhorar a interação com usuários [223].
- **Serviço:** representa alguma funcionalidade provida por um dispositivo, por exemplo, transmissão de audio e impressão. Cada serviço possui um nome, uma descrição, uma lista de parâmetros e um tipo de retorno. Estas informações são úteis, por exemplo, para tentar encontrar serviços alternativos que não estão mais disponíveis no ambiente.
- **Ponto (*peer*)**: pontos de redes remotos (*hosts*). Cada *ponto* possui um nome e uma lista de serviços que provê.

A arquitetura do *Wings* é ilustrada na Figura 12.1 [224]. Os conceitos de *ponto* e *serviço* são implementados pelo módulo de *Rede Pervasiva*. O conceito de *contexto*, por sua vez, é implementado no módulo de *Ciência de Contexto*. Cada um destes módulos é descrito a seguir.

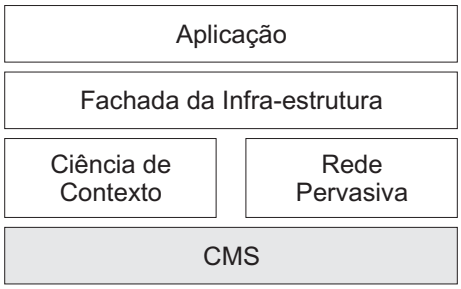



Figura 12.1: Arquitetura do *Wings*.

**Módulo de Rede Pervasiva**

Este módulo implementa os mecanismos que permitem a descoberta de pontos por um dispositivo, assim como a provisão de serviços por parte do mesmo (anúncio e descoberta). Estas características são implementadas por dois tipos de componentes, no caso do *Wings*, denominados *plug-ins*: *Plug-ins de Provisão de Serviços* (PPSs) e *Plug-ins de Descoberta de Nós* (PDNs). Cada um destes *plug-ins* é uma extensão de componente funcional, como definido na CMS. Desta forma, é possível inserir, remover e alterar PPSs e PDNs em tempo de execução.

Além disso, é possível implantar diferentes implementações de tais *plug-ins*, ao mesmo tempo. Sendo assim, a descoberta de nós e a provisão de serviços pode ser realizada com diferentes tecnologias e protocolos de rede. Por exemplo, as operações podem ser executadas sobre as redes *UPnP*, *JXTA* e *Bluetooth*, aumentando o número de nós e serviços aos quais um dispositivo tem acesso.

Para que isto seja possível, cada PPS e PDN provê um conjunto específico de serviços. O PPS provê quatro serviços: *discoverServices*, *stopServiceDiscovery*, *advertiseService* e *unadvertiseService*. O serviço *discoverServices* busca por serviços remotos disponíveis na rede. O serviço recebe como argumento um conjunto de palavras-chave representando a funcionalidade desejada, assim como uma referência a um objeto ouvinte (*listener*). Este ouvinte será notificado quando um serviço compatível com as palavras-chave for encontrado. O serviço retorna um identificador único para a busca. Este identificador deve ser provido ao serviço *stopServiceDiscovery* para que

a busca por um serviço seja interrompida. Por fim, os serviços *advertiseService* e *unadvertiseService* são usados, respectivamente, para registrar e desregistrar um serviço.

Cada PDN, por sua vez, provê apenas dois serviços: *discoverPeers* e *stopPeerDiscovery*. O primeiro inicia a descoberta de nós na rede associados com o PPS. Ele recebe um ouvinte como argumento, que é notificado quando um nó é descoberto. Da mesma forma que o serviço *discoverServices*, o serviço *discoverPeers* também retorna um identificador para a busca. Tal identificador deve ser provido ao serviço *stopPeerDiscovery* para que a busca por um nó seja interrompida.

O *Módulo de Rede Pervasiva* é implementado como um contêiner, como definido na CMS. Portanto, PPSs e PDNs podem ser inseridos e removidos da infra-estrutura sem reiniciar a aplicação. É através deste contêiner que todos os serviços disponibilizados pelos PPSs e PDNs são acessados, "escondendo" do módulo superior todos os detalhes relacionados ao processo de descoberta de nós e serviços.

Na Figura 12.2, ilustra-se o processo de descoberta de serviços através do contêiner referente ao *Módulo de Rede Pervasiva*. Cada passo ilustrado na figura é descrito a seguir.

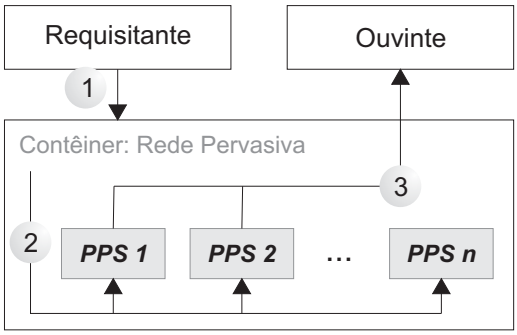



Figura 12.2: Processo de Descoberta de Nós.

1. Um requisitante inicia a descoberta de serviço acessando o contêiner do *Módulo de Rede Pervasiva*.

2. O contêiner encaminha a requisição de descoberta de serviço para cada PPS instalado.

3. Quando um PPS descobre um serviço, ele notifica ao ouvinte associado à busca. Neste exemplo, os PPSs 1 e 2 descobriram serviços e notificaram o ouvinte.

### Módulo de Ciência de Contexto

O *Módulo de Ciência de Contexto* provê um mecanismo para a disponibilização de informação de contexto para aplicações que executam no módulo de aplicação do *Wings*. Este processo pode ser realizado de duas maneiras: usando pares chave-valor ou eventos de contexto. No primeiro caso, cada informação é associada a uma chave, que é usada para recuperar o valor atual da informação. O nível atual de bateria do dispositivo e o número de nós próximos ao dispositivo são exemplos deste tipo de informação de contexto.

Eventos de contexto associam informação de contexto com uma condição. Cada condição é associada a uma chave, que serve para registrar um ouvinte à mesma. Quando a condição é satisfeita, um evento é disparado para notificar todos os ouvintes cadastrados. Por exemplo, uma aplicação pode estar interessada no evento de "bateria baixa" para aplicar alguma técnica de redução de consumo de energia ou armazenar dados críticos antes da bateria ter sido completamente consumida.

A informação de contexto é disponibilizada através de um tipo específico de *plug-in*, denominado *Plug-in de Ciência de Contexto* (PCC). Para cada domínio específico da informação sendo disponibilizada (casas inteligentes, informação turística, bibliotecas etc), tem-se um PCC associado. Cada PCC provê dois serviços: *retrieveInformation* e *registerContextCondition*, associados respectivamente às abordagens de pares chave-valor e baseada em evento de contexto.

O encapsulamento de mecanismos de acesso à informação de contexto em *plug-ins* torna a infra-estrutura extensível e adaptável a diferentes ambientes pervasivos. Além disso, considerando que cada PCC é um componente tal qual definido na CMS, ele pode ser carregado dinamicamente, de acordo com a demanda da aplicação e do usuário. Assim como o *Módulo de Rede Pervasiva*, os PCCs estão encapsulados em um contêiner, como definido na CMS.

### Módulo de Fachada da Infra-estrutura

O *Módulo de Fachada da Infra-estrutura* apenas provê uma interface única de acesso aos serviços por parte das aplicações. Para isto, a fachada possui um método de acesso para cada serviço provido pelos *plug-ins* da camada abaixo, ou seja, PPSs, PDNs e PCCs. Além disso, mantém referência para o contêiner raiz que contém os módulos de rede pervasiva e ciência de contexto (Figura 12.3).

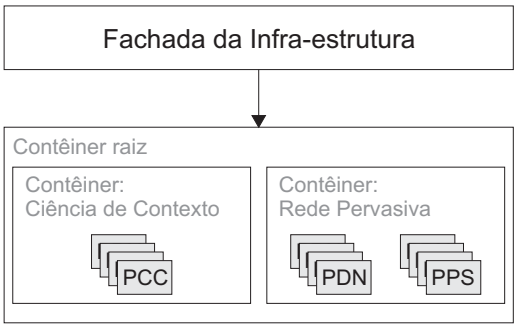



Figura 12.3: Acesso da fachada ao contêiner raiz.

A fachada permite que aplicações não necessitem conhecer detalhes de arquitetura e implementação dos mecanismos de descoberta de nós, acesso a serviços e informação de contexto. Considerando o vocabulário de padrões de projeto orientado a objetos, o *Módulo de Fachada da Infra-estrutura* implementa o padrão *Facade* [16].

**Módulo de Aplicação**

Este módulo representa todo tipo de software que faz uso da informação de contexto e serviços disponibilizados pelo *Wings*. Aplicações construídas no nível deste módulo são denominadas *Winglets*. As *Winglets* utilizam o *Módulo de Fachada da Infra-estrutura* para acessar os serviços disponibilizados pelos *plug-ins*.

### 12.1.2 Cenário de Execução: Biblioteca Pervasiva

Para exemplificar a utilização do *Wings*, foi desenvolvida uma aplicação para a biblioteca do Laboratório de Sistemas Embarcados e Computação Pervasiva (http://embedded.ufcg.edu.br). A *Biblioteca Pervasiva* é um software com arquitetura cliente-servidor, onde os clientes são implementados em dispositivos móveis e o servidor em um computador localizado no laboratório.

A aplicação é simples, porém, serve como piloto para exercitar as principais características do *Wings*: contexto, nós e serviços. A arquitetura da aplicação é ilustrada na Figura 12.4.

O servidor da biblioteca disponibiliza dois serviços a serem acessados pelas *Winglets*: *busca por palavra-chave* (Figura 12.5), que permite que se busque os livros da biblioteca relacionados a um conjunto de palavras-chave (Figuras 12.6 e 12.7); e *consulta de livro específico*, que permite verificar a disponibilidade de um livro específico.

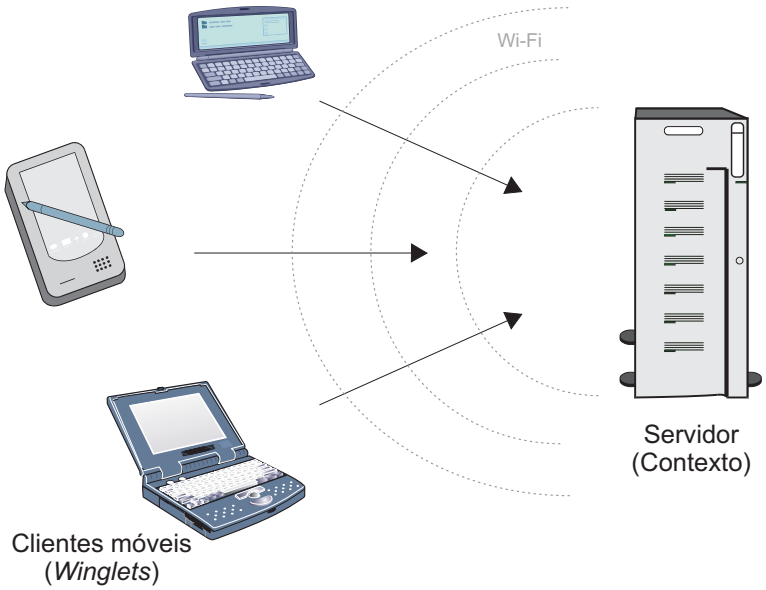



Figura 12.4: Cenário de Execução: Biblioteca Pervasiva.

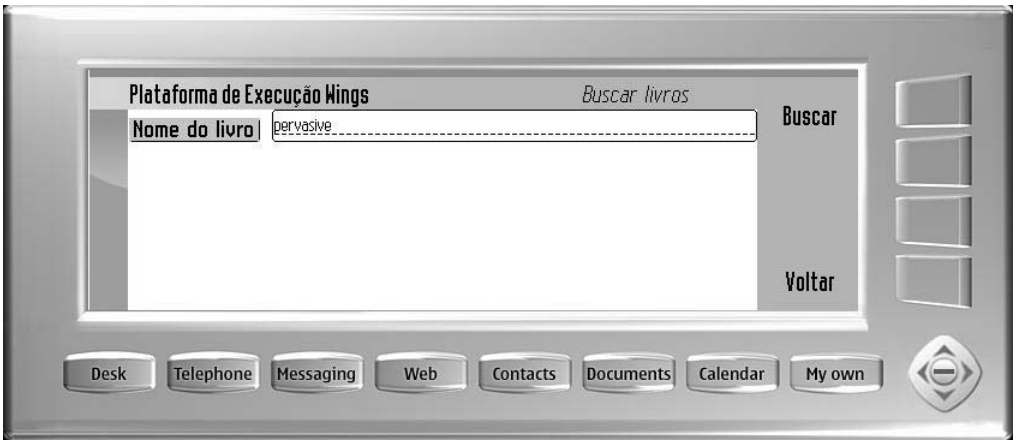



Figura 12.5: Tela da aplicação da Biblioteca Pervasiva para coletar as palavras-chave a serem usadas na busca por livros.


Figura 12.6: Tela da aplicação da Biblioteca Pervasiva para exibição dos livros encontrados durante a busca.

Para os clientes móveis, foram implementados três *plug-ins*: um *PCC*, que disponibiliza a informação de contexto sobre os livros da biblioteca; um *PDS*, que utiliza *web services* para a descoberta do serviço da biblioteca; e um *PDN*, que é utilizado para descoberta de nós na proximidade. O dispositivo móvel utilizado para executar a aplicação cliente foi um *smart phone Nokia Communicator 9500* equipado com interface de rede *Wi-Fi*.

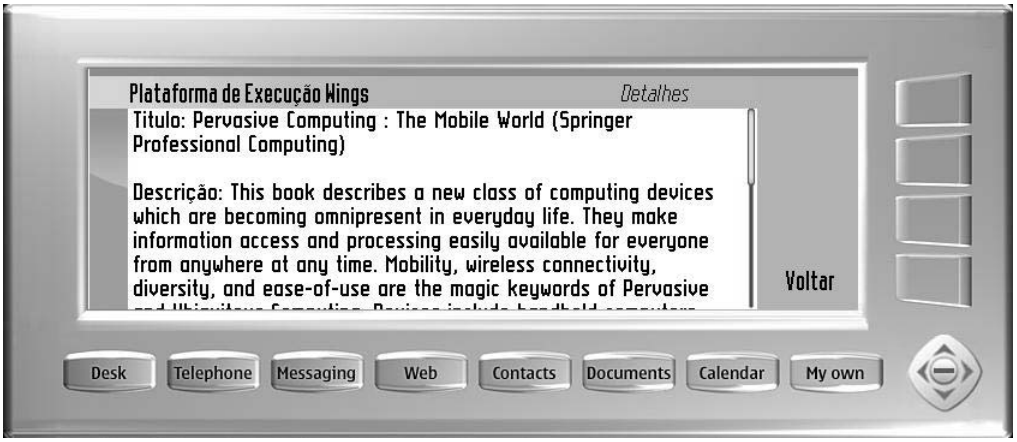



Figura 12.7: Tela da aplicação da Biblioteca Pervasiva para exibição dos detalhes de um livro selecionado.

Além dos serviços de busca, a biblioteca possui também um serviço de notificação de livros de interesse. Este serviço foi implementado usando o mecanismo de *evento de contexto*, mencionado anteriormente. A funcionalidade torna possível cadastrar um livro de interesse e, quando este estiver disponível na biblioteca, notificar o usuário da disponibilidade do mesmo (Figura 12.8).

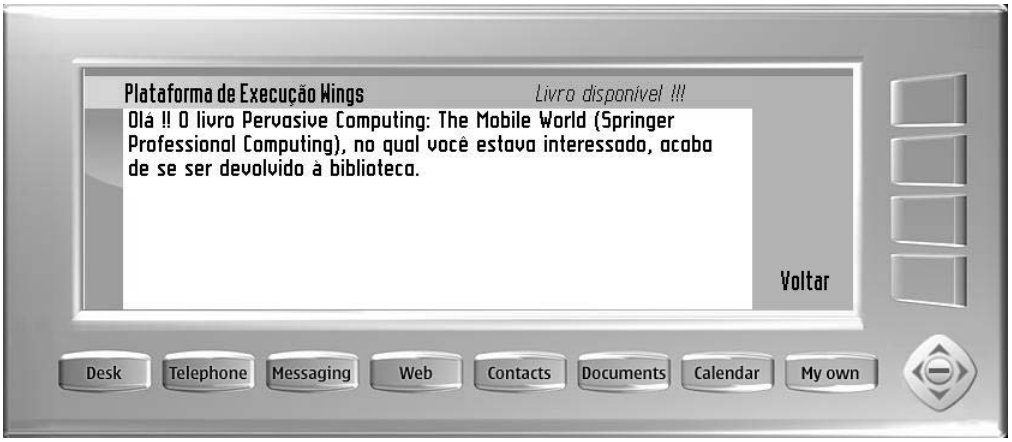



Figura 12.8: Tela da Biblioteca Pervasiva para notificação de que um livro de interesse encontra-se disponível.

### 12.1.3 Cenário de Evolução

Como mencionado anteriormente, Computação Pervasiva é um dos domínios chave para a aplicação da infra-estrutura proposta nesta tese. A característica aberta e dinâmica das aplicações requer que novos módulos não previstos em tempo de projeto sejam carregados sob demanda, dinamicamente. No exemplo da biblioteca pervasiva, novos *plug-ins* podem ser carregados, com novos mecanismos de comunicação e acesso a contexto, descoberta de nós e acesso a serviços.

Por exemplo, considere o seguinte cenário de evolução ilustrado na Figura 12.9. Um usuário portando dispositivo móvel está acessando uma rede *Bluetooth* e, aos poucos, desloca-se para fora do alcance daquela rede ao mesmo tempo em que entra na área de abrangência de uma rede *Wi-Fi*.

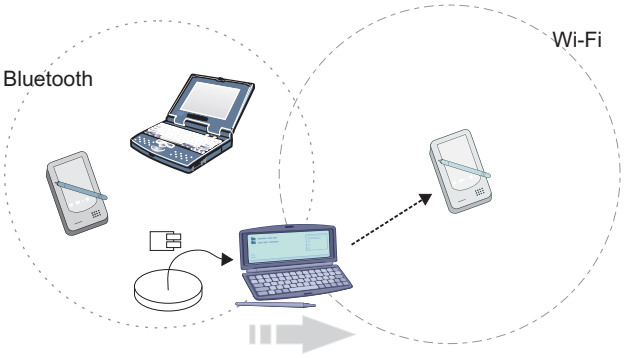



Figura 12.9: Cenário de evolução dinâmica não antecipada de software.

Com informações de contexto obtidas via rede *Bluetooth*, o dispositivo descobre que a rede na qual está entrando é uma rede *Wi-Fi* e que não existe um *plug-in* implementado no dispositivo

para estabelecer tal comunicação. Neste momento, pode-se obter via *Bluetooth* o *plug-in* necessário e, utilizando a infra-estrutura da CMS/JCF sobre a qual foi construída o *Wings*, adicioná-lo dinamicamente à aplicação.

Vale ressaltar que qualquer componente do *Wings* poderia ter sido utilizado no exemplo de cenário de evolução. Sendo mais específico, os *plug-ins* de ciência de contexto e descoberta de serviços são potenciais pontos de evolução considerando aplicações pervasivas, mas com a utilização da CMS/JCF qualquer outro componente da aplicação poderia ser alterado.

É importante notar também que, em nenhum momento, foi especificado que componentes poderiam ser evoluídos na aplicação. O único requisito para o desenvolvedor é a concepção da aplicação à luz da CMS e utilizando o JCF. A infra-estrutura esconde os mecanismos que permitem a evolução de qualquer parte da aplicação, mesmo dinamicamente.

## 12.2 Comunidades Virtuais Móveis

Ainda no contexto de computação pervasiva e dispositivos móveis, desenvolveu-se uma infra-estrutura para a construção de aplicações de Comunidades Virtuais Móveis (CVM). Neste tipo de aplicação, usuários utilizam dispositivos móveis interconectados que constituem comunidades de acordo com interesses em comum. Com base na comunicação entre os dispositivos, usuários são notificados da proximidade de outros usuários e, de acordo com a similaridade de seus interesses, estabelecem novas comunidades. Nestas comunidades são compartilhados conteúdos que são acessados de acordo com os níveis de autorização de cada indivíduo.

O objetivo desta infra-estrutura é prover suporte à construção de aplicações para comunidades móveis, provendo os serviços essenciais e comuns a este tipo de aplicação. Mais especificamente, os seguintes serviços são providos [67]:

- **Notificação da proximidade de indivíduos -** Esta funcionalidade tem dois objetivos: promover o aumento na rede de relacionamentos e aumentar o nível de interação entre os indivíduos. No caso da notificação de proximidade entre indivíduos desconhecidos, tem-se a oportunidade de aumentar a rede de relacionamentos ao criar novas comunidades baseadas em interesses em comum. Já no caso da notificação de proximidade de indivíduos conhecidos, aumenta-se a possibilidade de interação entre os mesmos.

- **Notificação da proximidade de dispositivos -** A notificação da proximidade de indivíduos depende da notificação da proximidade de dispositivos.

- **Identificação da similaridade entre indivíduos -** A partir da notificação de proximidade de indivíduos, tem-se a possibilidade de identificar a similaridade de interesses entre os mesmos. A medida de similaridade funciona como um filtro, evitando o excesso de notificações em locais públicos – apenas indivíduos com interesses comuns são notificados.

- **Formação de comunidades -** A partir da identificação de similaridade, deve ser possível constituir comunidades virtuais para compartilhamento de conteúdo entre indivíduos com interesses em comum. Uma vez que estas comunidades são formadas de acordo com a proximidade física dos indivíduos elas fazem sentido somente na área de cobertura da conexão existente entre os dispositivos.

- **Disponibilização de conteúdo em comunidades -** A partir da formação de comunidades, deve ser possível disponibilizar conteúdo entre os indivíduos. Por conteúdo, entenda-se documentos, planilhas eletrônicas, artigos, apresentações, etc.

- **Restrição de acesso ao conteúdo das comunidades -** Uma vez que há disponibilização de conteúdo, deve haver mecanismos de autenticação e autorização para evitar que o mesmo seja visualizado por indivíduos que não pertençam a uma dada comunidade.

### 12.2.1 Arquitetura

Na Figura 12.10, ilustra-se a arquitetura da infra-estrutura para a construção de comunidades virtuais móveis proposta baseada na CMS. Os componentes da arquitetura são descritos a seguir.

O componente `UserSearcher` implementa a notificação de proximidade de indivíduos, buscando usuários próximos a um determinado usuário. O componente *UserManager* é responsável pelo gerenciamento do acesso a informações dos usuários e implementa as funcionalidades de formação de comunidades e representação dos interesses de indivíduos. A notificação da proximidade de dispositivos é implementada pelo componente `DeviceSearcher` que busca por dispositivos próximos ao dispositivo do usuário. A identificação da similaridade entre indivíduos é contemplada pelo componente `Similarity`. O componente `ContentSharing` implementa os

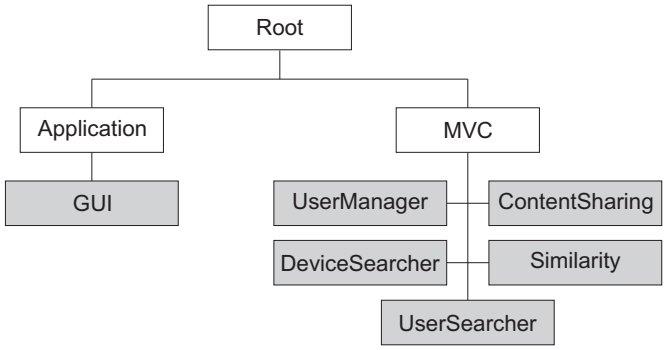



Figura 12.10: Arquitetura baseada na CMS para a infra-estrutura de CVM.

mecanismos de disponibilização de conteúdo em comunidades, assim como a restrição de acesso a este conteúdo disponibilizado. Por fim, o componente `GUI` implementa as funcionalidades de interface com o usuário.

### 12.2.2 Aplicação Desenvolvida para CVM

Para avaliar e validar a especificação da infra-estrutura para o desenvolvimento de comunidades virtuais móveis, foi implementada uma versão funcional de cada um dos componentes, para a plataforma J2ME, perfil CDC, assim como no *Wings*. A seguir são apresentadas as principais funcionalidades da aplicação com suas respectivas telas de interface gráfica. A plataforma utilizada para implantação da aplicação foi o *Nokia Communicator 9500*.

O primeiro passo para utilização de uma aplicação de comunidade virtual móvel é a definição dos interesses. O usuário pode editar seus interesses utilizando a tela ilustrada na Figura 12.11. Novos interesses podem ser definidos na caixa de texto *Interesse* e adicionados através do botão de adição. Após a adição de um novo interesse, a lista de interesses é atualizada. Interesses também podem ser excluídos através da seleção na lista e utilização do botão de exclusão. Após a exclusão de um interesse, a lista de interesses também é atualizada. Esta funcionalidade é implementada pelo componente *UserManager*.

Uma vez definidos os interesses do usuário, sempre que um novo usuário encontrado pela infra-estrutura é encontrado, exibe-se uma mensagem similar à ilustrada na Figura 12.12. Com base nesta notificação, o usuário pode ou não adicionar este novo usuário à sua lista de contatos. A busca por usuários é realizada pelo componente *UserSearcher*. O componente *Similariry* é responsável pela verificação de similaridade. Atualmente, a similaridade é verificada através de


Figura 12.11: Tela para a edição dos interesses do usuário.

pares chave-valor.

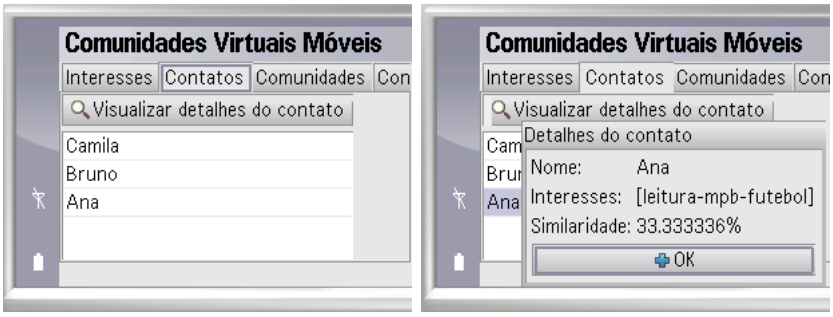

Figura 12.12: Tela de notificação da proximidade de usuários similares.

Para evitar receber notificações indesejadas, o usuário pode ajustar o limiar de similaridade que deseja utilizar. Este ajuste é realizado através da tela ilustrada na Figura 12.13. O botão *Salvar* é utilizado para armazenar o valor do limiar de similaridade. Este tipo de informação é gerenciada pelo componente *UserManager*.


Figura 12.13: Tela para o ajuste do limiar de similaridade.

Uma vez identificados novos usuários com interesses em comum, pode-se utilizar as telas ilustradas na Figura 12.14 para a visualização da lista de contatos do usuário. O usuário pode

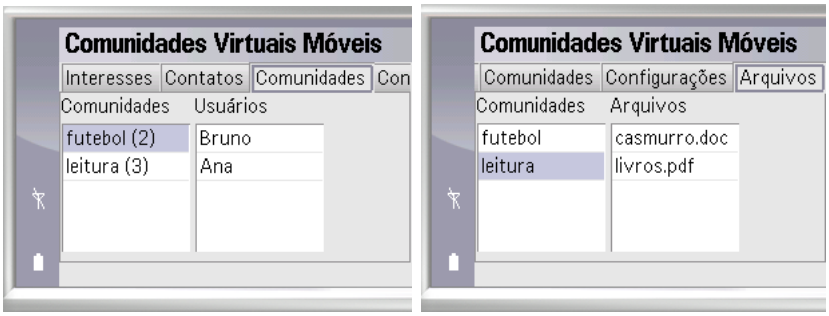
visualizar a sua lista de contatos de acordo com a tela apresentada na Figura 12.14(a). Além disso, pode visualizar os detalhes relativos a cada contato, tais como nome, lista de interesses e a similaridade existente (ver Figura 12.14(b)).




(a) Lista de contatos (b) Detalhes de um contato

Figura 12.14: Telas para a visualização da lista de contatos do usuário.

Por fim, através da tela ilustrada na Figura 12.15, pode-se visualizar todas as comunidades das quais o usuário faz parte, assim como o número de contatos que participam de cada comunidade (ver Figura 12.15(a)). Além disso, pode-se também visualizar e acessar os arquivos compartilhados disponíveis em cada uma destas comunidades (Figura 12.15(b)).




(a) Lista de comunidades (b) Arquivos disponíveis

Figura 12.15: Telas para a visualização da lista de comunidades.

Os cenários de evolução testados nesta aplicação foram a mudança do componente de similaridade com a adição de um novo componente dependente. Assim como ocorre na construção do *Wings*, em nenhum momento foi especificado que componentes poderiam ser evoluídos na aplicação. A infra-estrutura esconde os mecanismos que permitem a evolução dinâmica de qualquer parte da aplicação.

## 12.3 Sistemas Multi-Agentes Abertos

A abordagem multi-agentes tem sido apontada como um novo e promissor paradigma para a engenharia de sistemas de software complexos [46]. Um Sistema Multi-Agentes (SMA) é caracterizado por conjuntos de agentes com objetivos e papéis específicos que interagem de acordo com protocolos estabelecidos dentro do contexto de uma organização [45].

Em geral, agentes cooperam e co-existem dentro de um *ambiente*. Na literatura de agentes, existem diversas definições para ambiente, que está às vezes relacionado com a infra-estrutura de software ou hardware que gerencia a implantação e a execução de sistemas multi-agentes. De acordo com este tipo de definição, um ambiente deve prover as condições para que agentes existam e sejam executados [226].

No contexto deste trabalho, um ambiente é definido como uma entidade contendo recursos, situados ou não, que pode ser acessada por agentes de acordo com restrições do próprio ambiente. Do ponto de vista de engenharia de software, a entidade ambiente se torna ainda mais complexa no contexto de sistemas multi-agentes abertos. Neste tipo de sistema, há uma forte característica de dinamicidade e heterogeneidade de agentes, fazendo com que recursos, restrições e os próprios agentes entrem e saiam do ambiente dinamicamente e de maneira imprevisível [29]. Estas características de sistemas multi-agentes abertos demandam uma infra-estrutura de software que permita realizar alterações, inserções e remoções, em tempo de execução dos diversos componentes de um SMA: agentes, recursos, restrições e o próprio ambiente.

Com base nessa motivação, definiu-se uma especificação de agentes baseada na CMS, denominada *Agent Model Specification* (AMS). A AMS tem como objetivo a concepção de uma especificação de agentes com suporte à evolução não antecipada de software. Em sua primeira fase, cujo resultado é descrito neste documento, o foco foi a especificação dos conceitos relacionados à entidade *ambiente*.

### 12.3.1 Requisitos para a Especificação de Agentes

Como mencionado anteriormente, um ambiente pode ser visto como uma entidade que possui um conjunto de recursos, posicionados ou não, que são acessados pelos agentes que compõem o sistema. O acesso aos recursos é controlado por restrições impostas pelo próprio ambiente e pode ser redefinido pelos agentes ou pelo desenvolvedor. Os recursos pertencentes ao ambiente

também podem ser redefinidos, assim como novos podem ser adicionados. Levando em conta estes possíveis cenários, os seguintes requisitos são definidos para flexibilização do projeto de software do ambiente.

- *Desacoplamento entre agentes e ambiente:* o agente deve ter acesso aos recursos disponíveis no ambiente, de acordo com as restrições de acesso. Porém, o acoplamento Agente-Ambiente e Agente-Recurso deve ser definido de forma a permitir a mudança dos recursos e do ambiente, sem alteração nos agentes. Da mesma forma, mudanças em agentes não devem acarretar mudanças de software nos recursos aos quais tais agentes têm acesso.

- *Desacoplamento entre recursos:* os recursos disponibilizados no ambiente são independentes do tipo de arquitetura multi-agentes, ou pelo menos, grande parte delas. Desta forma, a nível de software, um recurso pode ser tão simples quanto um objeto mantendo um estado, ou tão complexo quanto um gerenciador de banco de dados distribuído. Sendo assim, não há como prever se a aplicação em desenvolvimento irá necessitar de acessos entre recursos. Considerando esta possibilidade, a inserção, remoção ou alteração de um recurso não deve acarretar mudança de software em outros recursos do ambiente.

- *Desacoplamento entre ambiente e recursos:* em domínios em que o ambiente possua "responsabilidades extras" em relação aos recursos nele contidos, a unidade de software referente ao ambiente deve poder ser alterada sem acarretar mudanças de software nos recursos.

Com base nestes requisitos, cada uma das entidades relacionadas ao ambiente em sistemas multi-agentes foi mapeada para entidades da especificação da CMS, buscando obter o mesmo suporte de evolução não antecipada de software. Atualmente, tem-se realizado o mesmo processo para mapear as entidades referentes à organização de agentes e o projeto interno de agentes.

### 12.3.2 *Agent Model Specification* (AMS)

No nível de ambiente, as entidades de composição são *Ambiente*, *Recursos* e *Restrições*. Cada uma destas entidades é discutida isoladamente a seguir.

**Definição do ambiente**

A estrutura de composição do ambiente é apresentada na Figura 12.16. O ambiente é representado por um contêiner, nos termos da especificação CMS, gerenciando o acesso aos seus recursos e a interação entre eles. Além disso, o ambiente agrega algumas outras características em relação ao contêiner da CMS, tais como: gerência de posicionamento de recursos via mapa/grade e imposição de restrições sobre o acesso aos recursos.

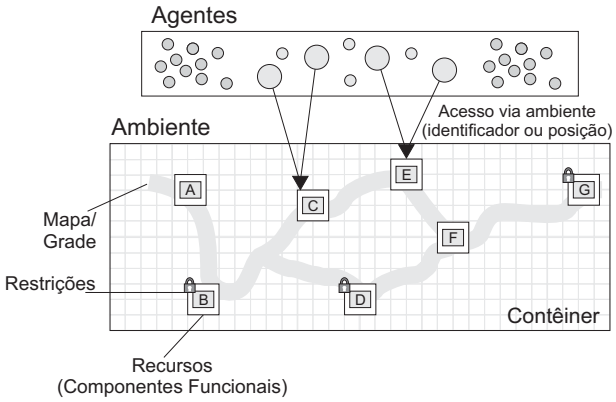



Figura 12.16: Composição do ambiente.

A estrutura em árvore do ambiente ilustrado na Figura 12.16 é apresentada na Figura 12.17. Os modelos de disponibilização e interação entre recursos são os mesmos descritos para os componentes da CMS. O ponto de acesso aos recursos pelos agentes do sistema é o contêiner do ambiente.

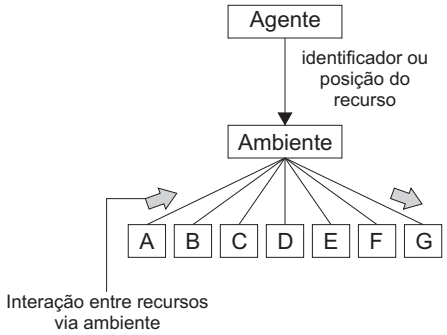



Figura 12.17: Estrutura em árvore da composição do ambiente.

Os recursos são representados por componentes funcionais da especificação CMS. Ao contrário da disponibilização de serviços de componentes funcionais, o acesso a recursos pode ser feito

via identificador ou posicionamento do recurso. Para permitir o posicionamento dos recursos no ambiente, definem-se os conceitos de grade e mapa. Uma grade é uma matriz de posições para a construção do mapa do ambiente. Um mapa é um subconjunto das posições definidas na matriz, que representam as posições nas quais podem existir recursos. Na Figura 12.18 é ilustrada a definição do mapa da Figura 12.16 como um subconjunto das posições da grade. Quanto mais fiel à topologia for a descrição do ambiente, maior deverá ser a quantidade de posições na grade.

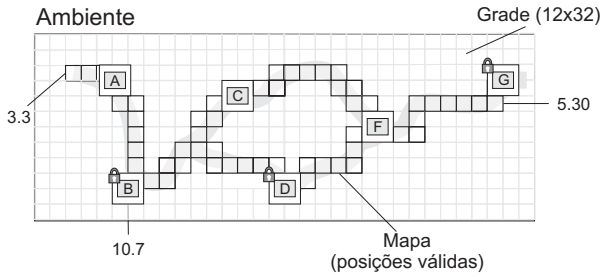



Figura 12.18: Definição do mapa do ambiente.

### Definição de recursos

Como mencionado anteriormente, um ambiente é um contêiner que agrega funcionalidades de gerência de acesso aos recursos. Os recursos são então representados por componentes funcionais, cujos serviços são: *leitura*, *escrita*, *remoção* e *criação*. Através da leitura e da escrita se obtém acesso ao recurso e altera-se o seu estado, respectivamente. A remoção e a criação são gerenciadas pelo contêiner, mas pode-se definir no recurso algum pré- ou pós-processamento de criação e remoção.

Outros serviços podem ser definidos para os recursos, mas os quatro acima descritos são serviços obrigatórios. Além disso, eventos podem ser disparados e recebidos por recursos, assim como a invocação de serviços entre recursos pode ser feita através do ambiente, seguindo a especificação CMS.

### Definição de restrições

Em termos de projeto de software, no contexto deste trabalho, uma restrição é considerada uma adaptação ao provimento de um serviço ou evento por parte de um recurso. Sendo assim, impor

restrições de operações sobre um recurso é equivalente a definir um *Adaptador* dos serviços/eventos providos por aquele recurso, de acordo com a especificação CMS.

As restrições de acesso básicas, tais como aquelas relacionadas à permissão dos serviços de leitura, escrita, criação e remoção, podem ser gerenciadas pelo ambiente, de uma forma padrão. Além disso, uma restrição pode ser redefinida como um novo adaptador personalizado, caso a restrição às operações básicas não seja suficiente. Uma vez que o adaptador pode ser composto dinamicamente, de acordo com a especificação CMS, as restrições na AMS também podem ser redefinidas dinamicamente.

### 12.3.3 *Java Agent Framework* (JAF)

*Java Agent Framework* (JAF) é uma implementação em Java da AMS. Assim como a AMS foi definida com base nos conceitos da CMS, o projeto do JAF foi concebida como uma extensão do projeto do JCF/GCF. As principais classes do projeto do JAF são: `Environment` e `MappedEnvironment`, que estendem a classe `Container`; a classe `Resource`, que estende a classe `FunctionalComponent`; e `Constraint`, que estende a classe `Adapter`. Na Figura 12.19 ilustra-se um diagrama de classes simplificado do arcabouço [227].

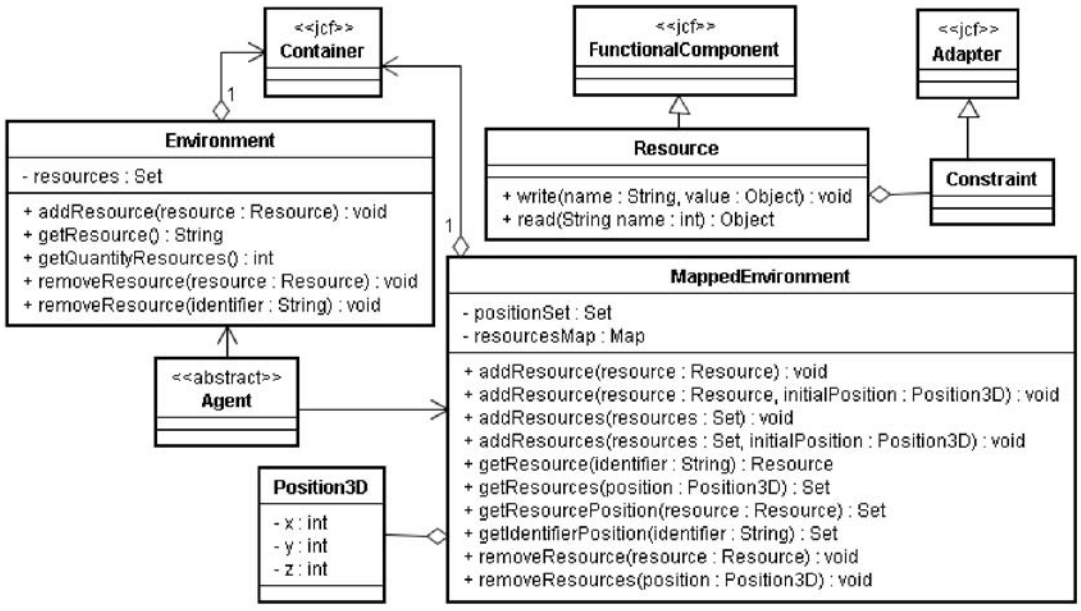



Figura 12.19: Diagrama de classes simplificado do JAF.

O arcabouço provê suporte a dois tipos de ambientes. A classe `Environment` implementa um

ambiente não situado, ou seja, onde posições dos recursos não são relevantes. Esta classe pode ser usada para implementar um ambiente representado por um banco de dados, em uma aplicação de comércio eletrônico, por exemplo. A classe `MappedEnvironment` implementa um ambiente situado, onde o posicionamento dos recursos é essencial, como em uma aplicação de redes de sensores por exemplo. A diferença entres estas classes é que `MappedEnvironment` contém uma referência para `Map`, que possui um conjunto de pontos de três dimensões (`Position3D`).

Uma vez que não está sendo considerado o projeto interno dos agentes, a classe `Agent` é apenas uma classe de acesso ao ambiente. Nesse caso, considera-se um agente uma entidade caixa-preta e não há restrição quanto à sua implementação.

Algumas aplicações de exemplo já foram desenvolvidas utilizando a AMS e o JAF. Dentre elas uma aplicação em comércio eletrônico (ambiente não situado) e aplicações de simulação e jogos (ambientes situados) [227]. Com o suporte à evolução dinâmica não antecipada provido pela CMS, os recursos e restrições do ambiente podem ser alterados/adicionados/removidos dinamicamente, mesmo que os mecanismos que permitam tal evolução não sejam de conhecimento do desenvolvedor.

## 12.4 Considerações Finais

Neste capítulo foram apresentadas as principais aplicações desenvolvidas com a infra-estrutura proposta nesta tese. Mais especificamente, foram descritas infra-estruturas para o desenvolvimento de aplicações pervasivas, comunidades virtuais móveis e sistemas multi-agentes abertos. Para cada uma delas, foram discutidos a arquitetura baseada na CMS assim como os cenários testados para a evolução não antecipada de software. Dadas as características de dinamicidade e imprevisibilidade de tais aplicações, elas podem ser consideradas *killer applications* no contexto de evolução não antecipada de software.

A infra-estrutura para computação pervasiva, denominada *Wings*, foi a primeira aplicação desenvolvida para validação da CMS e JCF. *Wings* utiliza uma arquitetura baseada em *plug-ins* construída sobre o modelo de componentes da CMS. Os *plug-ins* podem ser adicionados, removidos e alterados dinamicamente, como discutido no cenário de evolução.

Ainda no mesmo domínio de aplicação, a infra-estrutura para o desenvolvimento de comunidades virtuais móveis foi definida como uma extensão do *Wings*, seguindo uma arquitetura

baseada na CMS. Os principais serviços relacionados a esta aplicação são descoberta de pares com interesses similares, notificação de proximidade, formação de comunidades e compartilhamento de recursos. Todos eles disponíveis em uma rede *ad hoc*, com heterogeneidade de serviços e aplicações, demandando uma estrutura para evolução dinâmica não antecipada.

Por fim, são apresentados os primeiros resultados de um projeto de infra-estrutura para o desenvolvimento de sistemas multi-agentes abertos baseada na CMS. Este projeto está sendo desenvolvido em uma cooperação entre a Universidade Federal de Campina Grande e a Pontifícia Universidade Católica do Rio de Janeiro [61], com apoio do CNPq. Os diversos conceitos da CMS foram mapeados para conceitos relacionados ao domínio de sistemas multi-agentes, criando uma especificação de agentes (*Agent Model Specification* - AMS). Atualmente, o foco do projeto está sobre a definição de ambientes, como descrito neste capítulo.

# Capítulo 13

# Conclusão

Nesta tese abordou-se o problema de suporte de engenharia à evolução dinâmica não antecipada de software. A motivação para a abordagem deste tema é o grande impacto das diversas atividades inerentes à evolução sobre o tempo e custo total de produção do software. Como discutido neste documento, o impacto da evolução é ainda maior quando as inevitáveis alterações no software ao longo do seu ciclo de vida não foram previstas no seu projeto inicial, acarretando mudanças na sua arquitetura, projeto e código.

Este tipo de evolução não antecipada é ainda mais complexo de ser gerenciado em sistemas que necessitam funcionar de forma ininterrupta, por razões financeiras e de segurança, tais como sistemas bancários, telecomunicações, militares, dentre outros. Outros tipos de aplicações no domínio de computação pervasiva e autônoma também possuem o requisito de evolução dinâmica não antecipada.

Diversas abordagens existentes foram analisadas para identificar qual seria a mais adequada ao suporte à evolução dinâmica não antecipada. O maior problema de tais abordagens é que elas não deixam este suporte transparente para o desenvolvedor. Ou seja, o desenvolvedor deve apontar os possíveis pontos de extensão do sistema ou deve gerenciar, além dos próprios requisitos de sua aplicação, os mecanismos que permitem a evolução dinâmica não antecipada.

Motivados por este problema, no contexto desta tese, foi concebida uma infra-estrutura para o desenvolvimento de software com suporte à evolução não antecipada. Mais especificamente, definiu-se um modelo de componentes que é a base conceitual para a composição de aplicações, um conjunto de ferramentas para desenvolvimento e execução do software e um processo de

desenvolvimento para guiar a utilização do modelo de componentes e das ferramentas. Além disso, foram desenvolvidas diversas aplicações para validar a infra-estrutura proposta.

Com o auxílio desta infra-estrutura o desenvolvedor constrói aplicações baseadas em componentes com suporte à evolução não antecipada mas não precisa estar ciente dos mecanismos que promovem tal evolução. Uma vez utilizando as diretrizes e ferramentas propostas, o software desenvolvido já estará pronto para evoluir dinamicamente, mesmo considerando alterações não previstas *a priori*.

A seguir, apresenta-se um resumo das principais contribuições, limitações e perspectivas futuras do trabalho. Espera-se com estas contribuições não apenas auxiliar futuras pesquisas na área mas, também, auxiliar desenvolvedores no processo de desenvolvimento de software. Pois, por um lado, as diversas publicações obtidas durante a realização do trabalho já representam um indicativo de que tal contribuição existe [174, 114, 169, 228, 170, 229, 224, 213, 116, 115, 227, 230, 231]. Contudo, por outro lado, em termos práticos, apenas com a implantação destas contribuições na indústria de software será possível argumentar com convicção que a infra-estrutura proposta realmente impacta positivamente, como esperado, o processo de engenharia de software.

## 13.1 Contribuições

A seguir apresenta-se um resumo das principais contribuições do trabalho no contexto de desenvolvimento de software com suporte à evolução dinâmica não antecipada:

- **Definição de um modelo de composição de componentes –** Foi definida e formalizada uma especificação de modelo de componentes, denominada CMS, para a construção de software com suporte à evolução dinâmica não antecipada. A especificação de componentes inclui modelos de interação entre componentes, composição e execução de aplicações, além de modelos para adaptação e personalização de componentes visando maximizar a reutilização dos mesmos.

- **Desenvolvimento de arcabouços de componentes –** Foi desenvolvido um projeto genérico de arcabouço de componentes que implementa a CMS, denominado GCF. Este arcabouço, que serve como modelo para o desenvolvimento de arcabouços dependentes de linguagem, foi implementado nas linguagens Java (JCF), Python (PyCF), C++ (CCF) e CSharp (SCF)

como forma de validar o projeto do arcabouço genérico e a viabilidade da implementação da CMS em diversas linguagens. Além disso, apresentou-se uma extensão do GCF para a construção de aplicações corporativas com suporte à evolução dinâmica não antecipada.

- **Técnica para verificação formal –** Uma técnica para a verificação de propriedades do sistema diante de cenários de evolução foi definida. A técnica se baseia na composição de modelos de componentes e da aplicação escritos na linguagem Alloy. Utilizando a ferramenta de análise de Alloy (*Alloy Analyzer*) torna-se possível verificar se, ao inserir, alterar ou remover um dado componente, há algum impacto negativo sobre a corretude do sistema.

- **Modelo para análise de desempenho –** Para auxiliar o desenvolvedor na identificação da arquitetura mais adequada para a sua aplicação, considerando os requisitos de coesão funcional, flexibilidade e desempenho, definiu-se um modelo de análise de desempenho. Com este modelo, torna-se possível identificar o impacto de uma determinada arquitetura baseada na CMS sobre o desempenho de uma aplicação. É possível também gerar gráficos de desempenho da aplicação como um todo, considerando cada componente e, com base nestas informações, reorganizar os componentes na arquitetura buscando melhor desempenho.

- **Desenvolvimento de ambientes para composição e execução de aplicações –** Foram desenvolvidos ambientes para o auxílio ao desenvolvimento de componentes e composição de aplicações de acordo com a CMS sobre a plataforma Eclipse, denominados CDE e CCT, respectivamente. Dentre as ferramentas disponíveis, incluem-se: analisador de dependências entre componentes; verificador formal de corretude baseada na técnica mencionada anteriormente; e um analisador de desempenho baseado no modelo também mencionado anteriormente. Estes ambientes estão sendo disponibilizados atualmente apenas para aplicações e componentes Java. Além disso, foi implementado um ambiente para o auxílio à execução de aplicações desenvolvidas utilizando o JCF, denominado JCAS. O JCAS foi construído sobre a plataforma Oscar/OSGi e funciona como servidor de aplicações desenvolvidas utilizando o JCF.

- **Processo de desenvolvimento –** Como forma de guiar o desenvolvedor na aplicação da infra-estrutura, definiu-se um processo de desenvolvimento de software com suporte à evolução dinâmica não antecipada. O processo é fundamentado no ciclo de desenvolvimento

baseado em componentes e também inclui diretrizes para a utilização das ferramentas em cada fase do ciclo de desenvolvimento.

- **Desenvolvimento de aplicações-piloto –** Foram desenvolvidas diversas aplicações utilizando a infra-estrutura como forma de validar seus diversos aspectos dentro do processo de engenharia de software. A principal delas foi uma infra-estrutura para o desenvolvimento de aplicações pervasivas denominada *Wings*. Outras aplicações no domínio de comunidades virtuais móveis e sistemas multi-agentes também foram desenvolvidas.

## 13.2 Limitações do Trabalho

A seguir, apresenta-se uma visão crítica das limitações deste trabalho, as quais devem ser abordadas para dar continuidade à pesquisa desenvolvida nesta tese.

- **Transferência de estado -** Não há suporte a transferência de estado entre os componentes da aplicação. Desta forma, o desenvolvedor deve tratar a passagem de estado utilizando técnicas de persistência de dados. O foco do trabalho é sobre a estrutura e funcionalidades da aplicação. Componentes podem armazenar estado internamente mas, caso este componente seja trocado por outro, não há um mecanismo provido pela infra-estrutura para transferir o estado do componente antigo para o novo componente.

- **Escalabilidade -** Não foi realizado um estudo sobre a escalabilidade da abordagem proposta, principalmente no que diz respeito à quantidade de componentes, serviços e eventos considerando diversos níveis de profundidade da hierarquia de contêineres. Este é um dos fatores determinantes para a utilização em larga escala da abordagem na indústria.

- **Evolução de dados -** A evolução do modelo de dados da aplicação não é tratada na abordagem descrita. Caso haja um cenário de evolução que cause alterações no modelo de dados da aplicação, o desenvolvedor precisará gerenciar a atualização dinâmica. Há alguns problemas complexos a serem resolvidos para prover evolução de dados. O principal deles é como refletir alterações em uma entidade do modelo nos diversos componentes que o utilizam como base de dados, considerando que tal entidade pode, por exemplo, passar a possuir novos relacionamentos e/ou novas restrições de integridade.

- **Mudança de plataforma -** Uma das formas de evolução de software descrita na literatura é aquela que envolve mudança de plataforma. Evidentemente que, ao se considerar evolução dinâmica, a mudança de plataforma se torna bem mais complexa e, pragmaticamente, não é simples visualizar um cenário onde ela seria realmente necessária. Porém, como forma de delimitar o escopo e limitações deste trabalho, vale ressaltar que a infra-estrutura não prevê a migração de uma aplicação de uma plataforma para outra em tempo de execução.

- **Verificação em tempo de execução -** Uma das características relevantes no contexto de evolução não antecipada que não foi abordada neste trabalho é a verificação em tempo de execução. Para isso, seria necessário um monitoramento da execução da aplicação para verificar se a mudança de estado ou evolução causou algum impacto na corretude da especificação do sistema. Caso isso ocorra, deve-se desfazer a ação ou evolução ocorrida no sistema para retorná-lo a um estado consistente com as propriedades esperadas (*rollback*). A abordagem utilizada na infra-estrutura introduzida nesta tese é preventiva e não corretiva, ou seja, busca-se manter a corretude da especificação apenas *antes* de evoluir a aplicação.

- **Ambientes para aplicações em C++, Python e CSharp -** Apesar das arquiteturas do CCT, CDE e CAS serem genéricas, foram realizadas adaptações apenas para a linguagem Java. Sendo assim, apenas componentes e aplicações Java podem ser desenvolvidos. No caso do CCT e do CDE, a adaptação necessária para que sejam utilizados para o desenvolvimento de aplicações em outras linguagens é mínima. Já no caso do CAS, além de suporte a carregamento dinâmico e reflexão computacional, tem-se um conjunto de funcionalidades que podem ter uma implementação complexa de acordo com a linguagem de implementação, dentre elas, carregamento de classes a partir de caminhos do disco, gerência de diversas classes de componentes em um mesmo espaço de nomes (*sand-box*), dentre outras. Na versão em Java, estas características foram herdadas de OSGi.

- **Suporte para aplicações corporativas em C++, Python e CSharp -** Da mesma forma que os ambientes mencionados anteriormente, o suporte a aplicações corporativas foi apenas implementado na linguagem Java. Características como integração com a Web podem se tornar complexas de implementar de acordo com a linguagem utilizada. Evidentemente que deve-se levar em conta as características específicas das linguagens para avaliar se é

realmente viável prover todo o suporte proposto em cada uma delas.

## 13.3 Perspectivas

Além das limitações mencionadas anteriormente, vários rumos de pesquisa podem ser tomados com base nas contribuições deste trabalho. Alguns destes potenciais rumos de continuidade são descritos a seguir.

- **Gerência de repositório de componentes -** Um dos principais passos inerentes ao desenvolvimento baseado em componentes é a montagem de um repositório de componentes. É com base nesse repositório que os componentes podem ser reutilizados, adaptados, melhorados, levando aos poucos o desenvolvimento a um alto nível de qualidade, reutilização e produtividade. O estabelecimento de um repositório inclui mecanismos para certificação de componentes e controle de qualidade dos mesmos. Este é um dos potenciais trabalhos futuros desta tese. Nesse contexto, um projeto inicial já vem sendo desenvolvido [63].

- **Recuperação automática ou auxiliada -** Uma vez estabelecido um repositório de componentes, um outro desafio diz respeito à recuperação do componente mais adequado para um dado requisito da aplicação. No caso específico da recuperação de componentes baseados na infra-estrutura proposta, pode-se utilizar inclusive os modelos de análise de desempenho e verificação formal para dar suporte à recuperação automática ou, ao menos auxiliada, de potenciais componentes dado um conjunto de requisitos.

- **Métricas de software -** Um outro trabalho considerado indispensável para a utilização em larga escala da infra-estrutura é a proposição de métricas para avaliar experimentalmente se o aspecto de evolutibilidade da aplicação é realmente beneficiado com a utilização da CMS. Um estudo nesse sentido iria fortalecer o argumento de que a CMS provê em grandes projetos, na prática, aquilo que se observa em projetos de estudo de casos.

- **Suporte à computação autônoma -** Por fim, um trabalho de grande interesse de pesquisa pela comunidade científica atualmente é o provimento de soluções para computação autônoma [104]. De fato, alguns trabalhos já vêm sendo desenvolvidos no sentido de prover

suporte à composição autônoma acoplado à infra-estrutura introduzida nesta tese. Mais especificamente, há um trabalho de mestrado em andamento cujo tema é a implementação de mecanismos que permitam a auto-configuração de aplicações baseadas na CMS [69]. Outras funcionalidades relacionadas à computação autônoma, tais como auto-otimização, auto-recuperação e auto-proteção, podem ser também abordadas aproveitando o arcabouço conceitual da CMS e motivadas pela inerente característica de dinamicidade e imprevisibilidade deste tipo de aplicação.

# Bibliografia


[1] Research Institute in Software Evolution. Software Evolution Group. http://www.dur.ac.uk/RISE/ - Acessado em 10/09/2007.

[2] Peter Ebraert, Yves Vandewoude, Theo D'Hondt, Yolande Berbers. Pitfalls in Unanticipated Dynamic Software Evolution. *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution(RAM-SE'05)*, p. 41–50, Glasgow, Escócia, 2005.

[3] Bennet Lientz, E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.

[4] Len Erlikh. Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2(3):17–23, 2000.

[5] M. Zelkowitz, A. Shaw, J. Gannon. *Principles of Software Engineering and Design*. Prentice-Hall, 1979.

[6] B. Lientz, E. Swanson. Problems in Application Software Maintenance. *Communications of the ACM*, 24(11):763–769, 1981.

[7] J. R. McKee. Maintenance as a Function of Design. *Proceedings of AFIPS National Computer Conference*, p. 187–193, Las Vegas, EUA, 1984.

[8] O. Port. The Software Trap - Automate or Else. *Business Week*, 9(3051):142–154, 1988.

[9] S. Huff. Information Systems Maintenance. *The Business Quarterly*, 55(1):30–32, 1990.

[10] J. Moad. Maintaining the Competitive Edge. *Datamation*, 36(4):61–66, 1990.

[11] A. Eastwood. Firm Fires Shots at Legacy Systems. *Computing Canada*, 19(2):17, 1993.

[12] Bashar Nuseibeh, Steve Easterbrook. Requirements Engineering: a Roadmap. *Proceedings of the Conference on The Future of Software Engineering*, p. 35–46, New York, NY, EUA, 2000. ACM Press.

[13] Luke Hohmann. *Beyond Software Architecture: Creating and Sustaining Winning Solutions*. Addison-Wesley, 2003.

[14] Shari Lawrence Pfleeger. The Nature of System Change. *IEEE Software*, 15(3):87–90, 1998.

[15] David L. Parnas. Designing Software for Ease of Extension and Contraction. *Proceedings of the 3rd International Conference on Software Engineering*, p. 264–277, Piscataway, NJ, EUA, 1978. IEEE Press.

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

[17] Mohamed Fayad, Ralph Johnson, Douglas Schmidt. *Building Application Frameworks*. Wiley, 2000.

[18] Kevin Loney. *Oracle Database 10g: The Complete Reference*. McGraw-Hill Osborne Media, 2004.

[19] Michael Kofler. *The Definitive Guide to MySQL 5*. Apress, 2005.

[20] Korry Douglas. *PostgreSQL*. Sams, 2005.

[21] Ivica Crnkovic. Component-based Software Engineering - New Challenges in Software Development. *Software Focus*, v. 4, p. 127–133. Wiley, 2001.

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

[23] Marco Torchiano, Letizia Jaccheri, Carl-Fredrik Sorensen, Alf Inge Wang. COTS Products Characterization. *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, p. 335–338. ACM Press, 2002.

[24] M. Torchiano, M. Morisio. Overlooked Aspects of COTS-based Development. *IEEE Computer*, 21(2):88–93, 2004.

[25] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, Kurt Wallnau. Technical Concepts of Component-based Software Engineering. Relatório Técnico, Carnegie Mellon - Software Engeneering Institute, Maio 2000.

[26] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. Mehmet Akşit, Satoshi Matsuoka, editores, *Proceedings European Conference on Object-Oriented Programming*, v. 1241, p. 220–242. Springer-Verlag, Berlin, Heidelberg, New York, 1997.

[27] M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. *Proceedings of Fourth International Conference on Web Information Systems Engineering*, p. 3–12, Rome, Itália, 2003. IEEE.

[28] Johannes Mayer, Ingo Melzer, Franz Schweiggert. Lightweight Plug-In-Based Application Development. *Revised Papers from the International Conference NetObjectDays o Objects, Components, Architectures, Services, and Applications for a Networked World*, p. 87–102. Springer-Verlag, 2003.

[29] Michael Wooldridge. *An Introduction to Multiagent Systems*. Wiley, 2002.

[30] Joshua Kerievsky. Stop Over-Engineering! Software Development Magazine, Março 2002.

[31] FUSE Workshop. First International Workshop on Foundations of Unanticipated Software Evolution. http://www.informatik.uni-bonn.de/ gk/use/fuse2004/ - Acessado em 10/09/2007.

[32] Gunter Kniesel, Joost Noppen, Tom Mens, Jim Buckley. 1st International Workshop on Unanticipated Software Evolution. *ECOOP Workshop Reader*, v. 2548 - *LNCS*. Springer Verlag, 2002.

[33] Cezar Taurion. 24 Horas no Ar. Revista TI, Março 2002.

[34] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[35] P. Maes. Concepts and Experiments in Computational Reflection. *ACM SIGPLAN Notices*, 22(12):147–155, 1987.

[36] S. Liang, G. Bracha. Dynamic Class Loading in the JavaTM Virtual Machine. *Proceedings of OOPSLA'98*, p. 36–44, Vancouver, Canadá, 1998.

[37] Robert Chatley, Susan Eisenbach, Jeff Magee. Modelling a Framework for Plugins. *Proceedings of the Specification and Verification of Component-Based Systems workshop at ESEC/FSE'03*, p. 49–57, Helsinki, Finlândia, 2003.

[38] Eclipse.org. Eclipse Project. http://eclipse.org/ - Acessado em 10/09/2007.

[39] Ed Roman, Scott W. Ambler, Tyler Jewell. *Mastering Enterprise JavaBeans*. John Wiley and Sons, 2001.

[40] Object Management Group. CORBA FAQ and Resources. http://www.omg.org - Acessado em 10/09/2007.

[41] I. Ben-Shaul, O. Holder, B. Lavva. Dynamic Adaptation and Deployment of Distributed Components In Hadas. *IEEE Transactions on Software Engineering*, 27(9):769–787, 2001.

[42] OSGi Alliance. Open Services Gateway initiative. http://www.osgi.org - Acessado em 10/09/2007.

[43] R. S. Hall, H. Cervantes. An OSGi Implementation and Experience Report. *Proceedings of IEEE First Consumer Communications and Networking Conference*, p. 394–399, Las Vegas, EUA, 2004.

[44] Knopflerfish.org. Knopflerfish Framework. http://www.knopflerfish.org/ - Acessado em 10/09/2007.

[45] Nicholas R. Jennings. Agent-Oriented Software Engineering. Francisco J. Garijo, Magnus Boman, editores, *Proceedings of the 9th European Workshop on Modelling Autonomous*

*Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99)*, v. 1647, p. 1–7. Springer-Verlag: Heidelberg, Alemanha, 1999.

[46] Nicholas R. Jennings. An Agent-based Approach for Building Complex Software Systems. *Communications of the ACM*, 44(4):35–41, 2001.

[47] Kok-Leong Ong, Wee-Keong Ng. A Survey of Multi-Agent Interaction Techniques and Protocols. Relatório Técnico CAIS-TR04-98, School of Applied Science, Technological University, Nanyang, Singapura, 2000.

[48] F. Bellifemine, G. Rimassa, A. Poggi. JADE - A FIPA-Compliant Agent Framework. *Proceedings of the 4th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, p. 97–108, Inglaterra, 1999.

[49] Ethan Cerami. *Web Services Essentials*. O'Reilly, 2002.

[50] A. K. Lui, D. J. Miron. Issues Concerning the Development of a CORBA Trading Service. Relatório Técnico DSTO-TR-0841, Defense Science and Technology Organization, Salisbury, Austrália, 1999.

[51] Jim Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.

[52] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold. An Overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming*, p. 327–353, Londres, Reino Unido, 2001. Springer-Verlag.

[53] Ruzanna Chitchyan, Ian Sommerville. Comparing Dynamic AO Systems. *Dynamic Aspects Workshop in International Conference on Aspect Oriented Software Development*, p. 1–14, Lancaster, Reino Unido, 2004.

[54] Dave Thomas, Chad Fowler, Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2004.

[55] Mark Pilgrim. *Dive Into Python*. Apress, 2004.

[56] Bruce Eckel. *Thinking in Java*. Prentice Hall, 2006.

[57] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2000.

[58] Jesse Liberty. *Programming C# : Building .NET Applications with C#*. O'Reilly Media, 2005.

[59] Eric Clayberg, Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison-Wesley, 2004.

[60] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[61] Composição Dinâmica de Software para Computação Móvel e Ubíqua. Projeto CT-INFO/MCT/CNPq nº 011/2005, 2005.

[62] Mobile Services - Arcabouço Baseado em Componentes para a Disponibilização de Serviços para Dispositivos Móveis. Projeto Edital Universal CNPq - 019/2004, 2004.

[63] Desenvolvimento de uma Biblioteca de Componentes COMPOR. Projeto UFAL/FAPEAL, 2005.

[64] Arcabouço para Composição Dinâmica de Software Baseada em Agentes. Projeto PIBIC 2005/2006 - PROPEP/UFAL/CNPq, 2005.

[65] Concepção e Desenvolvimento de uma Biblioteca de Componentes de Software. Projeto PIBIC 2006/2007 - PROPEP/UFAL/CNPq, 2006.

[66] Desenvolvimento de Aplicações Embarcadas e Pervasivas. Projeto de Cooperação com a Nokia do Brasil Ltda., 2005.

[67] Glauber Vinícius Ventura de Melo Ferreira. Infra-estrutura de Software Baseada em Componentes para a Construção de Aplicações para Comunidades Virtuais Móveis. Dissertação de Mestrado, Universidade Federal de Campina Grande, Campina Grande, Brasil, 2006.

[68] Emerson Cavalcante Loureiro Filho. Um Middleware Extensível para Disponibilização de Serviços em Ambientes Pervasivos. Dissertação de Mestrado, Universidade Federal de Campina Grande, Campina Grande, Brasil, 2006.

[69] Mario Hozano de Souza. Uma Infra-Estrutura para o Desenvolvimento de Aplicações Pervasivas Auto-Configuráveis. Proposta de Dissertação de Mestrado. Departamento de Sistemas e Computação, Universidade Federal de Campina Grande, 2007.

[70] Waldemar Pires Ferreira Neto. Implementação do Módulo de Verificação de Dependências e Análise de Desempenho para o Ambiente de Composição de Componentes COMPOR. Monografia de Conclusão de Curso de Graduação, Universidade Federal de Campina Grande, Campina Grande, Paraíba, 2006.

[71] Stefânia Marques. Um Plug-in para o Desenvolvimento de Componentes do Modelo COMPOR. Monografia de Conclusão de Curso de Graduação, Universidade Federal de Campina Grande, Campina Grande, Paraíba, 2006.

[72] Kleinner Farias. COMPOR-Test: uma Ferramenta de Testes para o Ambiente de Composição de Componentes COMPOR. Monografia de Conclusão de Curso de Graduação, Universidade Federal de Alagoas, Maceió, Alagoas, 2006.

[73] Mário Hozano Lucas de Souza. Um Arcabouço de Software para Composição Dinâmica de Aplicações Web. Monografia de Conclusão de Curso de Graduação, Universidade Federal de Alagoas, Maceió, Alagoas, 2006.

[74] Willy Carvalho Tiengo. Desenvolvimento de uma Infra-estrutura de Suporte à Evolução Dinâmica de Sistemas Corporativos baseada no Arcabouço OSGi. Monografia de Conclusão de Curso de Graduação, Universidade Federal de Alagoas, Maceió, Alagoas, 2006.

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

[76] Marcílio Ferreira de Souza. Adaptando os métodos Catalysis e UML Components ao Modelo de Componentes COMPOR-CM. Monografia de Conclusão de Curso de Especialização, Centro de Estudos Superiores de Maceió, Maceió, Alagoas, 2004.

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

[78] Yguaratã Cavalcanti. Uma Implementação em Python do Modelo de Componentes COMPOR. Monografia de Conclusão de Curso de Graduação, Universidade Federal de Alagoas, Maceió, Alagoas, 2007.

[79] Grady Booch. *Software Component with ADA*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, EUA, 1987.

[80] Ivar Jacobson. *Object-oriented Software Engineering*. ACM Press, New York, NY, EUA, 1992.

[81] Oscar Nierstrasz, Dennis Tsichritzis, editores. *Object-oriented Software Composition*. Prentice Hall International (UK) Ltd., Hertfordshire, Reino Unido, 1995.

[82] Clemens Szypersky. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[83] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer Verlag, 2006.

[84] Ivica Crnkovic, Magnus Larsson, editores. *Building Reliable Component-Based Software Systems*. Artech House Publishers, 2002.

[85] OMG. CORBA Component Model, v3.0. http://www.omg.org/technology/documents/formal/components.htm - Acessado em 10/09/2007.

[86] Jeffrey M. Voas. The Challenges of Using COTS Software in Component-Based Development. *Computer*, 31(6):44–45, 1998.

[87] Nancy Talbert. The Cost of COTS. *Computer*, 31(6):46–52, 1998.

[88] B. Deifel. Supporting Reuse and Flexibility in COTS Variation Development. *Proceedings of Fifth International Workshop on Requirements Engineering: Foundation for Software Quality*, Heidelberg, Alemanha, 1999.

[89] Tricia Oberndorf. COTS and Open Systems - An Overview. Relatório Técnico, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, EUA, 2007.

[90] I. Crnkovic, S. Larsson, M. Chaudron. Component-based Development Process and Component Lifecycle. *27th International Conference on Information Technology Interfaces*, p. 591–596, Cavtat, Croacia, 2005.

[91] Chris Lüer, David S. Rosenblum, André van der Hoek. The Evolution of Software Evolvability. *Proceedings of the 4th International Workshop on Principles of Software Evolution*, p. 134–137, New York, NY, EUA, 2001. ACM Press.

[92] Ned Chapin. Do We Know What Preventive Maintenance Is? *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, p. 15, Washington, DC, EUA, 2000. IEEE Computer Society.

[93] Ned Chapin, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil, Wui-Gee Tan. Types of Software Evolution and Software Maintenance. *Journal of Software Maintenance*, 13(1):3–30, 2001.

[94] Tom Mens, Jim Buckley, Matthias Zenger, Awais Rashid. Towards a Taxonomy of Software Evolution. *International Workshop on Unanticipated Software Evolution*, Warsaw, Polônia, 2003.

[95] Peyman Oreizy, Nenad Medvidovic, Richard N. Taylor. Architecture-based Runtime Software Evolution. *Proceedings of the 20th International Conference on Software Engineering*, p. 177–186, Washington, DC, EUA, 1998. IEEE Computer Society.

[96] Qianxiang Wang, Junrong Shen, Xiaopeng Wang, Hong Mei. A Component-based Approach to Online Software Evolution: Research Articles. *Journal Software Maintenance and Evolution*, 18(3):181–205, 2006.

[97] Michael Hicks, Scott Nettles. Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, 2005.

[98] K. Whisnant, Z. T. Kalbarczyk, R. K. Iyer. A System Model for Dynamically Reconfigurable Software. *IBM Systems Journal*, 42(1):45–59, 2003.

[99] Mark E. Segal, Ophir Frieder. On-the-Fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, 10(2):53–65, 1993.

[100] Renato Cerqueira, Carlos Cassino, Roberto Ierusalimschy. Dynamic Component Gluing Across Different Componentware Systems. *Proceedings of the International Symposium on Distributed Objects and Applications*, p. 362, Washington, DC, EUA, 1999. IEEE Computer Society.

[101] How Much Is an Hour of Downtime Worth to You? Must-Know Business Continuity Strategies, Yankee Group, Boston, EUA, Julho 2002.

[102] Fran Berman, Geoffrey Fox, Anthony J.G. Hey, editores. *Grid Computing: Making The Global Infrastructure a Reality*. John Wiley & Sons, 2003.

[103] Loreno Oliveira, Leandro Sales, Emerson Loureiro, Hyggo Almeida, Angelo Perkusich. Filling the Gap Between Mobile and Service Oriented Computing - Issues for Evolving Mobile Computing Towards Wired Infrastructures and Vice Versa. *International Journal of Web and Grid Services*, 2:355–378, 2006.

[104] Mazeiar Salehie, Ladan Tahvildari. Autonomic Computing: Emerging Trends and Open Problems. *Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software*, p. 1–7, New York, NY, EUA, 2005. ACM Press.

[105] Roger Pressman. *Engenharia de Software*. MacronBooks, 1995.

[106] P. Oreizy, R. Taylor. On the Role of Software Architectures in Runtime System Reconfiguration. *Proceedings of the International Conference on Configurable Distributed Systems*, p. 61, Washington, DC, EUA, 1998. IEEE Computer Society.

[107] Vaclav Rajlich. Software Evolution: A Road Map. *International Conference on Software Maintenance*, p. 6, Florence, Itália, 2001.

[108] Daniel Jackson. Lightweight Formal Methods. *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, p. 1, Londres, Reino Unido, 2001. Springer-Verlag.

[109] Daniel Jackson, Kevin Sullivan. COM Revisited: Tool-assisted Modelling of an Architectural Framework. *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT Interna-*

*tional Symposium on Foundations of Software Engineering*, p. 149–158, New York, NY, USA, 2000. ACM Press.

[110] Daniel Jackson, Yu-Chung Ng, Jeannette M. Wing. A Nitpick Analysis of Mobile IPv6. *Formal Aspects of Computing*, 11(6):591–615, 1999.

[111] Daniel Jackson. Alloy: a Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology.*, 11(2):256–290, 2002.

[112] J. M. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall, 1999.

[113] MIT Software Design Group. The Alloy Analyzer. http://alloy.mit.edu/ - Acessado em 10/09/2007.

[114] Hyggo Almeida, Elthon Oliveira, Nádia Barbosa, Frederico Bublitz, Leandro Silva, Angelo Perkusich. Modelagem e Verificação Formal de Sistemas de Informação Baseados em Componentes. *Anais do II Simpósio Brasileiro de Sistemas de Informação*, Florianópolis, SC, Brasil, 2005.

[115] Elthon Oliveira, Hyggo Almeida, Leandro Silva, Angelo Perkusich. Formal Modelling and Verification of a Software Component Model using Coloured Petri Nets and Model Checking. *Proceedings of 22nd Annual ACM Symposium on Applied Computing*, p. 1427–1431, Seoul, Korea, 2007.

[116] Elthon Oliveira, Hyggo Almeida, Leandro Silva, Frederico Bublitz, Nádia Barbosa, Angelo Perkusich. Formal Modeling and Verification of Virtual Community Systems. Goran Putnik, Maria Cunha, editores, *Encyclopedia of Networked and Virtual Organizations*, v. 1. Idea Group Publishing, Hershey, PA, EUA, 2007, no prelo.

[117] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis, Methods and Practical Use*. EACTS – Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[118] Glauber Ferreira, Emerson Loureiro, Elthon Oliveira, Hyggo Almeida, Leandro Silva, Angelo Perkusich. A Java Code Annotation Approach for Model Checking Software Systems. *Proceedings of 22nd Annual ACM Symposium on Applied Computing*, p. 1536–1537, Seoul, Korea, 2007.

[119] Bell Labs. Promela Reference. http://spinroot.com/spin/Man/Intro.html - Acessado em 10/09/2007.

[120] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

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

[122] F. Doucet, S. Shukla, M. Otsuka, R. Gupta. BALBOA: a Component-based Design Environment for System Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(12):1597–1612, 2003.

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

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

[125] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, Jr. E. James Whitehead, Jason E. Robbins. A Component- and Message-based Architectural Style for GUI Software. *Proceedings of the 17th International Conference on Software Engineering*, p. 295–304, New York, NY, EUA, 1995. ACM Press.

[126] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, Jr. E. James Whitehead, Jason E. Robbins, Kari A. Nies, Peyman Oreizy, Deborah L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6):390–406, 1996.

[127] Nenad Medvidovic, Peyman Oreizy, Richard N. Taylor. Reuse of Off-The-Shelf Components in C2-style Architectures. *Proceedings of the 19th International Conference on Software Engineering*, p. 692–700, New York, NY, EUA, 1997. ACM Press.

[128] John Keeney, Vinny Cahill. Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework. *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, p. 3, Washington, DC, EUA, 2003. IEEE Computer Society.

[129] Barry Redmond, Vinny Cahill. Supporting Unanticipated Dynamic Adaptation of Application Behaviour. *Proceedings of the 16th European Conference on Object-Oriented Programming*, p. 205–230, Londres, Reino Unido, 2002. Springer-Verlag.

[130] Noriki Amano, Takuo Watanabe. A Procedural Model of Dynamic Adaptability and its Description Language. *Proceedings of Internation Workshop on Foundations of Software Evolution*, p. 103–107, Kyoto, Japan, 1998.

[131] Noriki Amano, Takuo Watanabe. LEAD: Linguistc Approach to Dynamic Adaptability for Practical Applications. *Proceedings of IFIP Conference on System Implementation*, p. 277–290, 1998.

[132] Noriki Amano, Takuo Watanabe. LEAD++: An Object-Oriented Language Based on a Reflective Model for Dynamic Software Adaptation. *Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems*, p. 41–50, Washington, DC, EUA, 1999. IEEE Computer Society.

[133] P. Maes, D. Nardi, editores. *Meta-level Architecture and Reflection*. Elsevier Science, 1988.

[134] Israel Ben-Shaul, Avron Cohen, Ophir Holder, Boris Lavva. HADAS: A Network-Centric Framework for Interoperability Programming. *International Journal of Cooperative Information Systems*, 6(3-4):293–314, 1997.

[135] I. Kim, D. Bae. A Dynamic Composition Model for Addressing Constrained Environments. *OOPSLA Workshop Reuse in Constrained Environments (RICE)*, Anaheim, California, EUA, 2003.

[136] Brendan Gowing, Vinny Cahill. Meta-Object Protocols for C++: The Iguana Approach. Relatório Técnico, University of Bologna, Bologna, Itália, 1996.

[137] Barry Redmond, Vinny Cahill. Iguana/J: Towards a Dynamic and Efficient Reflective Architecture for Java. *Workshop on Reflection and Meta-Level Architectures at 14th European Conference on Object-Oriented Programming*, Cannes, França, 2000.

[138] Davy Suvée, Wim Vanderperren, Viviane Jonckers. JAsCo: an Aspect-oriented Approach Tailored for Component based Software Development. *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, p. 21–29, Boston, Massachusetts, EUA, 2003. ACM Press.

[139] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, Viviane Jonckers. Adaptive programming in JAsCo. *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, p. 75–86, Chicago, Illinois, EUA, 2005. ACM Press.

[140] Qianxiang Wang, Feng Chen, Hong Mei, Fuqing Yang. An Application Server to Support Online Evolution. *ICSM'02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, p. 131–140, Washington, DC, EUA, 2002. IEEE Computer Society.

[141] Inderjeet Singh, Beth Stearns, Mark Johnson, editores. *Designing Enterprise Applications with the Java(TM) 2 Platform (Enterprise Edition)*. Addison-Wesley Professional, 2 ed., 2002.

[142] NetBeans.org. NetBeans IDE. http://www.netbeans.org/ - Acessado em 10/09/2007.

[143] David Urting, Yolande Berbers, Stefan Van Baelen, Tom Holvoet, Yves Vandewoude, Peter Rigole. A Tool for Component Based Design of Embedded Software. *Proceedings of the Fortieth International Conference on Tools Pacific*, p. 159–168, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[144] Kaffe. Kaffe Virtual Machine. http://www.kaffe.org - Acessado em 10/09/2007.

[145] Ana M. Roldán, Ernesto Pimentel, Antonio Brogi. Safe Composition of Linda-based Components. *Electronic Notes Theoretical Computer Science*, 82(6), 2003.

[146] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, Iulian Neamtiu. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.*, 29(4):1–66, 2007.

[147] Feng Chen, Hongji Yang, Bing Qiao, William Cheng-Chung Chu. A Formal Model Driven Approach to Dependable Software Evolution. *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, p. 205–214, Washington, DC, USA, 2006. IEEE Computer Society.

[148] U. Waqar, F. Khendek, D. Vincent. A Formal Approach for Software Maintenance. *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, p. 608–617, Washington, DC, USA, 2002. IEEE Computer Society.

[149] Ian Warren, Jing Sun, Sanjev Krishnamohan, Thiranjith Weerasinghe. An Automated Formal Approach to Managing Dynamic Reconfiguration. *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, p. 37–46, Washington, DC, USA, 2006. IEEE Computer Society.

[150] J. Dowling, V. Cahill. Dynamic Software Evolution and the K-Component Model. *Workshop on Software Evolution, OOPSLA*, Tampa Bay, Florida, EUA, 2001.

[151] Alessandro Orso, Anup Rao, Mary Jean Harrold. A Technique for Dynamic Updating of Java Software. *18th International Conference on Software Maintenance*, p. 649–658, Montreal, Quebec, Canada, 2002.

[152] Giacomo Piccinelli, Christian Zirpins, Winfried Lamersdorf. The FRESCO Framework: An Overview. *Symposium on Applications and the Internet Workshops (SAINT 2003 Workshops)*, p. 120–123, Orlando, FL, EUA, 2003. IEEE Computer Society.

[153] Y. Vandewoude, P. Rigole, D. Urting, Y. Berbers. Draco : An Adaptive Runtime Environment for Components. Relatório Técnico Report CW 372, Department of Computer Science, K.U.Leuven, Leuven, Belgica, 2003.

[154] Humberto Cervantes, Richard S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. *Proceedings of the International Conference on Software Engineering (ICSE)*, p. 614–623. IEEE Computer Society, 2004.

[155] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, Jean-Bernard Stefani. The FRACTAL Component Model and its Support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Software Practice and Experience.*, 36(11-12):1257–1284, 2006.

[156] Christian Becker, Marcus Handte, Gregor Schiele, Kurt Rothermel. PCOM - A Component System for Pervasive Computing. *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, p. 67–76, Orlando, Florida, USA, 2004. IEEE Computer Society.

[157] Christian Becker, Gregor Schiele, Holger Gubbels, Kurt Rothermel. BASE - A Micro-Broker-Based Middleware for Pervasive Computing. *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, p. 443–451, Fort Worth, USA, 2003. IEEE Computer Society.

[158] A. Mukhija, M. Glinz. Runtime Adaptation of Applications through Dynamic Recomposition of Components. *Proceedings of 18th International Conference on Architecture of Computing Systems*, p. 124–138, Innsbruck, Austria, 2005.

[159] Sun Microsystems. Java Beans Specification. http://java.sun.com/products/javabeans/docs/spec.html - Acessado em 10/09/2007.

[160] OMG. The Object Management Group (OMG). http://www.omg.org/ - Acessado em 10/09/2007.

[161] Dale Rogerson. *Inside COM*. Microsoft Press, 1997.

[162] Frank Redmond. *DCOM: Microsoft Distributed Component Object Model*. John Wiley & Sons, 1997.

[163] Juval Lowy. *Programming .NET Components*. O'Reilly, 2005.

[164] Z. Jarir, P. C. David, T. Ledoux. Dynamic Adaptability of Services in Enterprise JavaBeans Architecture. *Proceedings of Seventh International Workshop on Component-Oriented Programming at ECOOP 2002*, Malaga, Spain, 2002.

[165] Eric Newcomer. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley, 2002.

[166] Jonas Bonér. AspectWerkz - Dynamic AOP for Java. http://www.jonasboner.com/public/publications/aosd2004_aspectwerkz.pdf - Acessado em 10/09/2007., 2004.

[167] Angela Nicoara, Gustavo Alonso. Dynamic AOP with PROSE. *Proceedings of International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005) in conjunction with the 17th Conference on Advanced Information Systems Engineering (CAISE 2005)*, Porto, Portugal, 2005.

[168] Nanning Aspects. http://nanning.codehaus.org/ - Acessado em 10/09/2007.

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

[170] Hyggo Almeida, Angelo Perkusich, Rodrigo Paes, Evandro Costa. Composição Dinâmica de Componentes para Aplicações com Mudanças Freqüentes de Requisitos. *Anais do 4o Workshop de Desenvolvimento Baseado em Componentes*, p. 9–14, João Pessoa, Paraíba, Brasil, 2004.

[171] Paulo Blauth Menezes. *Matemática Discreta para Computação e Informática*. Sagra Luzzatto, 2004.

[172] Donald E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1997.

[173] S.K. Miller. Aspect-oriented Programming Takes Aim at Software Complexity. *IEEE Computer*, 34(4):18–21, 2001.

[174] Hyggo Almeida, Emerson Loureiro, Angelo Perkusich, Evandro Costa. Usando Aspectos para Personalizar a Execução de Aplicações Baseadas em Componentes de Prateleira.

*II Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP/SBES'05)*, p. 34–41, Uberlândia, Minas Gerais, Brasil, 2005.

[175] Roland T. Mittermeir. Facets of Software Evolution. Nazim H. Madhavji, Juan Fernandez-Ramil, Dewayne E. Perry, editores, *Software Evolution and Feedback - Theory and Practice*, p. 71–94. Wiley, 2006.

[176] Rasmus Lerdorf, Kevin Tatroe, Peter MacIntyre. *Programming PHP*. O'Reilly Media, 2006.

[177] Marco Cantù. *Mastering Delphi 7*. Sybex, 2003.

[178] Rod Stephens. *Visual Basic 2005 Programmer's Reference*. Wrox, 2005.

[179] Jonathan Knudsen. *Wireless Java: Developing with J2ME*. Apress, 2003.

[180] Jo Stichbury. *Symbian OS Explained : Effective C++ Programming for Smartphones*. John Wiley & Sons, 2005.

[181] Seal Reflex API. http://seal-reflex.web.cern.ch/seal-reflex/ - Acessado em 10/09/2007.

[182] R. Greg Lavender, Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. *Proceedings of Second Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, EUA, 1995.

[183] Joe Duffy. *Professional .NET Framework 2.0*. Wrox, 2006.

[184] Grady Booch, James Rumbaugh, Ivar Jacobson. *UML - Guia do usuário*. Campus, 2000.

[185] Cristina Chavez, Carlos Lucena. Support for Aspect-oriented Software Development. *Doctoral Symposium at OOPSLA'2001*, p. 14–18, Tamba Bay, USA, 2001.

[186] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996.

[187] Hans Bergsten. *Java Server Pages*. O'Reilly Media, 2003.

[188] Chris Schalk, Ed Burns, James Holmes. *JavaServer Faces: The Complete Reference*. McGraw-Hill Osborne Media, 2006.

[189] Angell Enterprises. Python Server Pages. http://www.ciobriefings.com/psp/ - Acessado em 10/09/2007.

[190] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[191] EJ-Technologies. JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html - Acessado em 10/09/2007.

[192] James Roskind. The Python Profiler. http://docs.python.org/lib/profile.html - Acessado em 10/09/2007.

[193] Jay Fenlason, Richard Stallman. GNU gprof - The GNU Profiler. http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html - Acessado em 10/09/2007.

[194] W3C. Extensible Markup Language (XML). http://www.w3.org/XML/ - Acessado em 10/09/2007.

[195] Serry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, Pat McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2003.

[196] Eclipse.org. Java Development Tools Subproject. http://www.eclipse.org/jdt/index.html - Acessado em 10/09/2007.

[197] Eclipse.org. Eclipse C/C++ Development Tooling - CDT. http://www.eclipse.org/cdt/ - Acessado em 10/09/2007.

[198] PyDev. PyDev - Python Development Environment. http://pydev.sourceforge.net/ - Acessado em 10/09/2007.

[199] Ian Sommerville. *Software Engineering*. Addison Wesley, 2000.

[200] Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[201] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison - Wesley, 1999.

[202] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.

[203] Desmond Francis D'Souza, Alan Cameron Wills. *Objects, Components, and Frameworks with UML : The Catalysis Approach*. Addison-Wesley, 1998.

[204] John Cheesman, John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[205] Kent Beck, Erich Gamma. JUnit Test Infected: Programmers Love Writing Tests - Java Report. http://junit.sourceforge.net/doc/testinfected/testing.htm - Acessado em 10/09/2007., 1998.

[206] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Relatório Técnico CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[207] M. Simos, D. Creps, C. Klinger, L. Levine, D. Allemang. Organization Domain Modelling (ODM) Guidebook Version 2.0. Relatório Técnico STARS-VC-A025/001/00, Synquiry Technologies, Inc, 1996.

[208] R. N. Taylor, W. Tracz, L. Coglianese. Software Development Using Domain-specific Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 20(5):27–37, 1995.

[209] H. Gomma, Kerschberg, V. Sugmaran, C. Bosh, I. Tavakoli, OHara. A Knowledge-Based Software Engineering Environment for Reusable Software Requirements and Architectures. *Automated Software Engineering*, 3(3-4), 1996.

[210] G. Fischer. Domain-Oriented Design Environments. *Automated Software Engineering - The International Journal of Automated Reasoning and Artificial Intelligence in Software Engineering*, 1(2):177–203, 1994.

[211] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison - Wesley, 1999.

[212] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–100, 1991.

[213] Emerson Loureiro, Loreno Oliveira, Hyggo Almeida, Glauber Ferreira, Angelo Perkusich. Improving Flexibility on Host Discovery for Pervasive Computing Middlewares. *Proceedings of the 3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*, p. 1–8, New York, NY, EUA, 2005. ACM Press.

[214] D. Garlan, D. Siewiorek, A. Smailagic, P. Steenkiste. Project Aura: Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.

[215] Tao Gu, Hung Keng Pung, Da Qing Zhang. Toward an OSGi-Based Infrastructure for Context-Aware Applications. *IEEE Pervasive Computing*, 3(4):66–74, 2004.

[216] Stephen S. Yau, Fariaz Karim, Yu Wang, Bin Wang, Sandeep K. S. Gupta. Reconfigurable Context-Sensitive Middleware for Pervasive Computing. *IEEE Pervasive Computing*, 1(3):33–40, 2002.

[217] Robert Grimm. One.World: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004.

[218] Tim Kindberg, John Barton. A Web-Based Nomadic Computing System. *Computer Networks*, 35(4):443–456, 2001.

[219] UPnP Forum. http://www.upnp.org - Acessado em 10/09/2007.

[220] L. Gong. JXTA: A Network Programming Environment. *IEEE Internet Computing*, 5(3):88–95, 2001.

[221] Wi-Fi Alliance. http://www.wi-fi.org - Acessado em 10/09/2007.

[222] Bluetooth Special Interest Group. http://www.bluetooth.org - Acessado em 10/09/2007.

[223] A. K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 1(5):4–7, 2001.

[224] Emerson Loureiro, Frederico Bublitz, Nadia Barbosa, Angelo Perkusich, Hyggo Almeida, Glauber Ferreira. A Flexible Middleware for Service Provision Over Heterogeneous Pervasive Networks. *Proceedings of the 2006 International Symposium on on World of Wireless,*

*Mobile and Multimedia Networks*, p. 609–614, Washington, DC, EUA, 2006. IEEE Computer Society.

[225] John W. Muchow. *Core J2ME Technology*. Prentice Hall, 2001.

[226] James Odell, H. Van Dyke Parunak, Mitchell Fleischer. Modeling Agents and their Environment: The Communication Environment. *Journal of Object Technology*, 2(3):39–52, 2003.

[227] Camila Nunes, Marcilio Souza, Hyggo Almeida, Glauber Ferreira, Angelo Perkusich, Evandro Costa. Applying a Component-Based Framework to Develop Multi-Agent Environments: Case Study. *Proceedings of 22nd Annual ACM Symposium on Applied Computing*, p. 37–41, Seoul, Korea, 2007.

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

[229] Gustavo Carvalho, Hyggo Almeida, Maíra Gatti, Glauber Ferreira, Rodrigo Paes, Angelo Perkusich, Carlos Lucena. Dynamic Law Evolution in Governance Mechanisms for Open Multi-Agent Systems. *Proceedings of Workshop on Sofware Engineering for Agent-oriented Systems*, p. 83–94, Florianópolis, SC, Brasil, 2006.

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

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

[232] Kurt Bittner, Ian Spence. *Use Case Modeling*. Addison-Wesley Professional, 2002.

[233] Sun Microsystems. The Reflection API. http://java.sun.com/docs/books/tutorial/reflect/ - Acessado em 10/09/2007.

[234] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley Professional, 2001.

[235] H. M. Deitel, P. J. Deitel. *Java How to Program*. Prentice Hall, 2004.

[236] Kathy Sierra, Bert Bates. *Head First Java*. O'Reilly Media, 2005.

[237] Sun Microsystems. Java.sun.com - The Source for Java Developers. http://java.sun.com/ - Acessado em 10/09/2007.

[238] Java World. JavaWorld.com - Fueling Innovation. http://www.javaworld.com/ - Acessado em 10/09/2007.

[239] J. Vlissides, J. Coplien, N. Kerth. *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

# Apêndice A

# Projeto de Arcabouço de Componentes Independente de Linguagem

Neste apêndice são detalhados os requisitos e o projeto orientado a objetos do arcabouço de componentes independente de linguagem baseado na CMS, denominado *Generic Component Framework* (GCF). Este projeto foi utilizado como base para a implementação dos arcabouços dependentes de linguagem detalhados no Apêndice B.

## A.1 Casos de Uso do Arcabouço

Diagramas de casos de uso são definidos na linguagem unificada de modelagem (UML) [184] e são utilizados para a descrição das funcionalidades de um sistema do ponto de vista do usuário. No caso do GCF, o usuário pode ser tanto o desenvolvedor da aplicação como o desenvolvedor de componentes. Estes são denominados *atores* e são representados como ilustrado na Figura A.1, que também ilustra a notação UML para *casos de uso*.

Seguindo a notação ilustrada na Figura A.1, a seguir são apresentados os casos de uso do GCF. Buscou-se utilizar uma versão simplificada do diagrama para facilitar a legibilidade, características de notação mais complexas para este tipo de diagrama podem ser encontradas em [232].

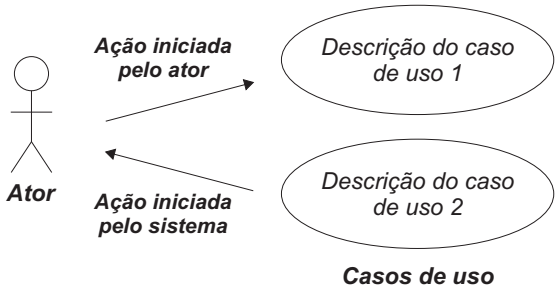



Figura A.1: Notação UML para Diagramas de Casos de Uso.

### A.1.1 Casos de Uso: Desenvolvimento de Componentes (DC)

Estes casos de uso estão relacionados ao desenvolvimento de componentes a serem disponibilizados em uma aplicação de acordo com a CMS. Na Figura A.2, ilustra-se o diagrama de casos de uso relacionados à construção de componentes. A descrição dos casos de uso é apresentada a seguir.

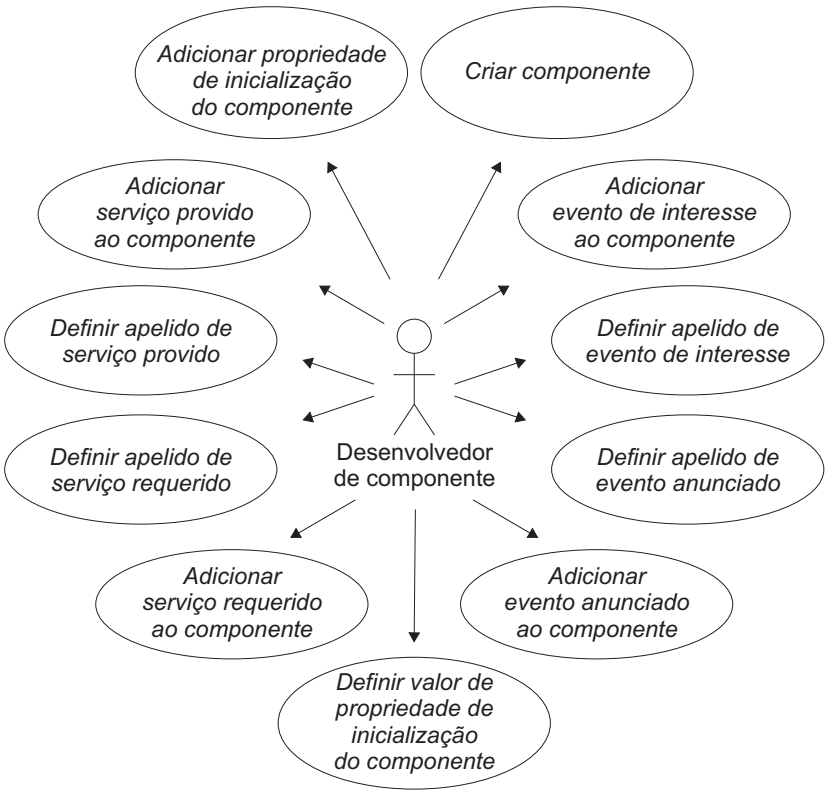



Figura A.2: Diagrama de casos de uso relacionados à construção de componentes.

**(DC:01) Criar componente** - Criar um componente funcional com funcionalidade específica. O arcabouço deve prover uma classe padrão para componente funcional a ser estendida pelos componentes específicos.

**(DC:02) Adicionar serviço provido ao componente** - Adicionar um serviço provido a um componente, informando seu *nome*, *descrição*, *tipos de parâmetros*, *tipo de retorno*, *tipos de exceção* que possam ocorrer durante a execução, uma referência para o *método* que implementa o serviço e um conjunto de apelidos de serviços requeridos para a execução deste serviço.

**(DC:03) Definir apelido de um serviço provido** - Definir o apelido de um serviço provido por um componente. A partir de então, o serviço deverá ser acessado pelo contêiner através do novo apelido.

**(DC:04) Adicionar serviço requerido ao componente** - Adicionar um serviço requerido a um componente, informando seu *nome*, *descrição*, *tipos de parâmetros*, *tipo de retorno* e *tipos de exceção* que possam ocorrer durante a execução.

**(DC:05) Definir apelido de um serviço requerido** - Definir o apelido de um serviço requerido por um componente. A partir de então, o serviço deverá ser requisitado ao contêiner através do novo apelido.

**(DC:06) Adicionar evento de interesse ao componente** - Adicionar um evento de interesse a um componente, informando seu *nome*, *descrição*, *tipos de parâmetros* e uma referência para o *método* que implementa a recepção do serviço.

**(DC:07) Definir apelido de um evento de interesse** - Definir o apelido de um evento de interesse de um componente. A partir de então, o evento deverá ser recebido do contêiner através do novo apelido.

**(DC:08) Adicionar evento anunciado ao componente** - Adicionar um evento anunciado a um componente, informando seu *nome*, *descrição* e *tipos de parâmetros*.

**(DC:09) Definir apelido de um evento anunciado** - Definir o apelido de um evento anunciado de um componente. A partir de então, o evento deverá ser anunciado ao contêiner através do novo apelido.

**(DC:10) Adicionar propriedade de inicialização do componente** - Adicionar uma propriedade de inicialização informando o *nome* da propriedade e seu *valor*.

**(DC:11) Definir valor de propriedade de inicialização do componente** - Definir o *valor* de uma propriedade de inicialização existente.

Na Figura A.3, ilustra-se o diagrama de casos de uso relacionados ao desenvolvimento de componentes, com foco na execução do componente. A descrição dos casos de uso é apresentada a seguir.

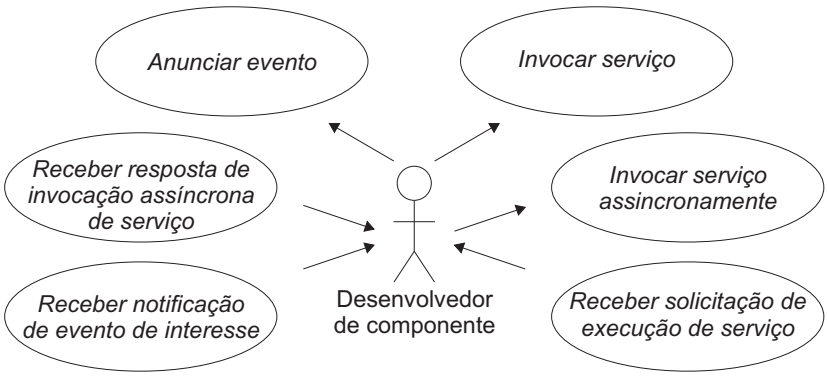



Figura A.3: Diagrama de casos de uso relacionados ao desenvolvimento de componentes, com foco em execução.

**(DC:12) Invocar serviço** - Invocar um serviço de outro componente através do contêiner, informando apelido e parâmetros de execução. Apenas serviços declarados como requeridos podem ser invocados (ver caso de uso DC:04).

**(DC:13) Invocar serviço assincronamente** - Invocar um serviço de outro componente através do contêiner, de forma assíncrona, informando *apelido* e *parâmetros* de execução. Um identificador da requisição deve ser gerado para posterior identificação do resultado. Apenas serviços declarados como requeridos podem ser invocados (ver caso de uso DC:04).

**(DC:14) Anunciar evento** - Anunciar um evento para o contêiner do componente, informando *apelido* e *parâmetros* de execução. Apenas eventos declarados como anunciados podem ser anunciados (ver caso de uso DC:08).

**(DC:15) Receber solicitação de execução de serviço** - Receber solicitação de execução de um serviço e executar o mesmo. O arcabouço deverá direcionar, sem intervenção do desenvolvedor, a requisição para o método declarado como implementador do serviço (ver caso de uso DC:02).

**(DC:16) Receber resposta de invocação assíncrona de serviço** - Receber resposta de invocação assíncrona e tratá-la de acordo com o identificador de requisição. Fica a cargo do desenvolvedor criar uma fila de requisições, caso várias invocações assíncronas sejam realizadas.

**(DC:17) Receber notificação de evento de interesse** - Receber notificação de um evento e tratar o mesmo. O arcabouço deverá direcionar, sem intervenção do desenvolvedor, a notificação para o método declarado como implementador da recepção do evento (ver caso de uso DC:06).

### A.1.2 Casos de Uso: Desenvolvimento de Aplicações (DA)

Estes casos de uso estão relacionados ao desenvolvimento de aplicações de acordo com a CMS. Na Figura A.4, ilustra-se o diagrama de casos de uso relacionados ao desenvolvimento de aplicações, com foco na parametrização dos componentes das mesmas. A descrição dos casos de uso é apresentada a seguir.

**(DA:01) Verificar existência de um serviço provido por um componente** - Verificar se um determinado *apelido* é igual ao de algum serviço provido pelo componente.

**(DA:02) Definir apelido de serviço provido por um componente** - Ver caso de uso DC:03.

**(DA:03) Obter apelido de serviço provido por um componente** - Recuperar o apelido atualmente definido para um serviço provido.

**(DA:04) Obter os serviços providos por um componente** - Recuperar o grupo de serviços providos por um componente.

**(DA:05) Definir apelido de serviço requerido por um componente** - Ver caso de uso DC:05.

**(DA:06) Obter apelido de serviço requerido por um componente** - Recuperar o apelido atualmente definido para um serviço requerido.


Figura A.4: Diagrama de casos de uso relacionados ao desenvolvimento de aplicações, com foco em parametrização de componentes.

**(DA:07) Obter os serviços requeridos por um componente -** Recuperar o grupo de serviços requeridos por um componente.

**(DA:08) Verificar existência de um evento de interesse de um componente -** Verificar se um determinado *apelido* é igual ao de algum evento de interesse do componente.

**(DA:09) Definir apelido de evento de interesse de um componente** - Ver caso de uso DC:07.

**(DA:10) Obter apelido de evento de interesse de um componente -** Recuperar o apelido atualmente definido para um evento de interesse.

**(DA:11) Obter os eventos de interesse de um componente** - Recuperar o grupo de eventos de interesse de um componente.

**(DA:12) Definir apelido de evento anunciado por um componente** - Ver caso de uso DC:09.

**(DA:13) Obter apelido de evento anunciado por um componente** - Recuperar o apelido atualmente definido para um evento anunciado.

**(DA:14) Obter os eventos anunciados por um componente** - Recuperar o grupo de eventos anunciados por um componente.

**(DA:15) Definir valor de propriedade de inicialização do componente** - Ver caso de uso DC:11.

**(DA:16) Obter propriedades de inicialização** - Recuperar o grupo de propriedades de inicialização do componente.

**(DA:17) Obter valor de propriedade de inicialização** - Recuperar o valor de uma propriedade de inicialização dado o seu *nome*.

Na Figura A.5, ilustra-se o diagrama de casos de uso relacionados ao desenvolvimento de aplicações, com foco na composição das mesmas. A descrição dos casos de uso é apresentada a seguir.

**(DA:18) Instanciar componente** - Criar uma instância de um componente, informando seu *nome*.

**(DA:19) Instanciar contêiner** - Criar uma instância de um contêiner, informando seu *nome*.

**(DA:20) Adicionar componente a um contêiner** - Adicionar uma instância de componente a um contêiner. O arcabouço atualizará no contêiner as informações das tabelas de serviços e eventos referentes ao seu novo componente. A atualização será propagada até a raiz da hierarquia.

**(DA:21) Adicionar contêiner a outro contêiner** - Adicionar uma instância de um contêiner a outro contêiner. O arcabouço atualizará no contêiner pai as informações das tabelas de

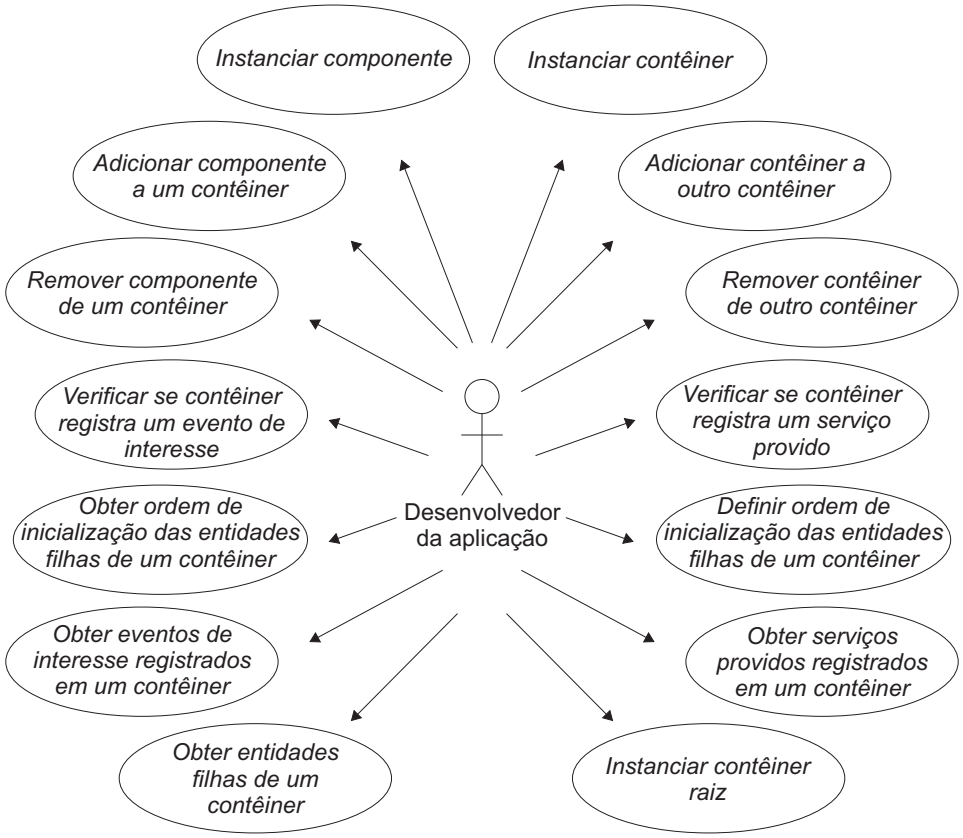



Figura A.5: Diagrama de casos de uso relacionados ao desenvolvimento de aplicações, com foco em composição.

serviços e eventos referentes ao seu novo contêiner-filho. A atualização será propagada até a raiz da hierarquia.

**(DA:22) Remover componente de um contêiner** - Remover um componente previamente adicionado de um contêiner. O arcabouço atualizará no contêiner as informações das tabelas de serviços e eventos referentes ao seu antigo componente. A atualização será propagada até a raiz da hierarquia.

**(DA:23) Remover contêiner de outro contêiner** - Remover um contêiner previamente adicionado de um outro contêiner. O arcabouço atualizará no contêiner pai as informações das tabelas de serviços e eventos referentes ao seu antigo contêiner-filho. A atualização será propagada até a raiz da hierarquia.

**(DA:24) Verificar se contêiner registra um evento de interesse** - Verificar se um determinado *apelido* é igual ao de algum evento de interesse registrado no contêiner.

**(DA:25) Verificar se contêiner registra um serviço provido** - Verificar se um determinado *apelido* é igual ao de algum serviço provido registrado no contêiner.

**(DA:26) Obter ordem de inicialização das entidades filhas de um contêiner** - Obter a ordem em que as entidades filhas de um contêiner são iniciadas, ao solicitar inicialização do próprio contêiner.

**(DA:27) Definir ordem de inicialização das entidades filhas de um contêiner** - Definir a ordem em que as entidades filhas de um contêiner são iniciadas, ao solicitar inicialização do próprio contêiner.

**(DA:28) Obter eventos de interesse registrados em um contêiner** - Obter o grupo de eventos de interesse registrados em um dado contêiner.

**(DA:29) Obter serviços providos registrados em um contêiner** - Obter o grupo de serviços providos registrados em um dado contêiner.

**(DA:30) Obter entidades filhas de um contêiner** - Obter o grupo de entidades filhas de um dado contêiner.

**(DA:31) Instanciar contêiner raiz** - Instanciar contêiner que será a raiz da hierarquia. A única diferença entre este contêiner e um contêiner comum é que a este será associado um *script* de execução.

Na Figura A.6, ilustra-se o diagrama de casos de uso relacionados ao desenvolvimento de aplicações, com foco na execução das mesmas. A descrição dos casos de uso é apresentada a seguir.

**(DA:32) Criar *script* de execução** - Criar um *script* de execução específico da aplicação. O arcabouço deve prover uma classe padrão para *script* de execução a ser estendida pelos *scripts* específicos.

**(DA:33) Instanciar *script* de execução** - Criar uma instância de *script* de execução informando a instância do contêiner raiz referente à hierarquia da aplicação.

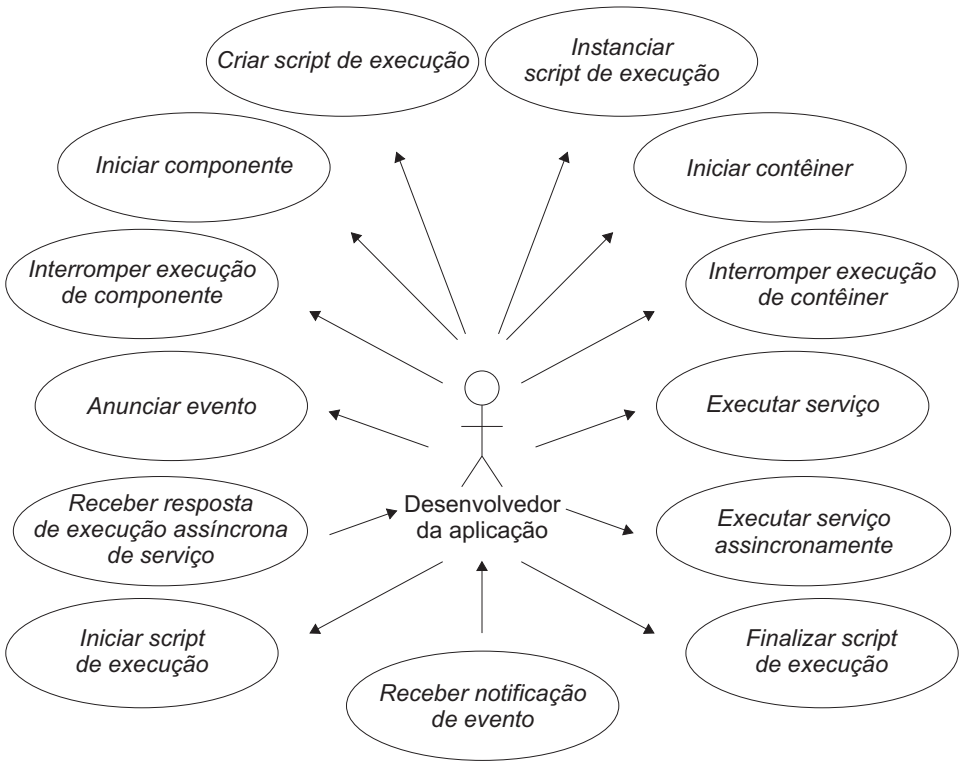



Figura A.6: Diagrama de casos de uso relacionados ao desenvolvimento de aplicações, com foco em execução.

**(DA:34) Iniciar componente** - Iniciar um componente da aplicação. A partir de então, os serviços providos pelo componente estarão disponíveis.

**(DA:35) Iniciar contêiner** - Iniciar um contêiner da aplicação. A inicialização de um contêiner equivale à inicialização de todos os seus componentes filhos, de acordo com a ordem previamente definida (ver caso de uso DA:27).

**(DA:36) Interromper execução de componente** - Interromper a execução de um componente da aplicação previamente iniciado. A partir de então, os serviços providos pelo componente não estarão mais disponíveis.

**(DA:37) Interromper execução de contêiner** - Interromper a execução de um contêiner da aplicação previamente iniciado. A interrupção da execução de um contêiner equivale à interrupção da execução de todos os seus componentes filhos, na ordem inversa da inicialização previamente definida (ver caso de uso DA:27).

**(DA:38) Anunciar evento** - Anunciar um dado evento aos interessados da hierarquia. O anúncio é realizado via contêiner raiz.

**(DA:39) Executar serviço** - Executar um dado serviço da hierarquia. A requisição é realizada via contêiner raiz.

**(DA:40) Executar serviço assincronamente** - Executar um dado serviço da hierarquia de forma assíncrona. A requisição é realizada via contêiner raiz.

**(DA:41) Receber resposta de execução assíncrona de serviço** - Ver caso de uso DC:16. Nesse caso, a resposta é recebida via contêiner raiz.

**(DA:42) Receber notificação de evento** - Receber notificação de evento via contêiner raiz da hierarquia e tratar o mesmo.

**(DA:43) Iniciar *script* de execução** - Iniciar o *script* de execução da aplicação. A inicialização do *script* equivale à inicialização do contêiner raiz da aplicação e, conseqüentemente, de todos os seus componentes.

**(DA:44) Interromper execução de *script* de execução** - Interromper a execução do *script* de execução da aplicação. A interrupção da execução do *script* equivale à interrupção da execução do contêiner raiz da aplicação e, conseqüentemente, de todos os seus componentes.

### A.1.3 Casos de Uso: Desenvolvimento de Adaptadores (DD)

Estes casos de uso estão relacionados ao desenvolvimento de adaptadores de acordo com a CMS. Considerando que componentes devem ser adaptados para suprir necessidades de uma determinada aplicação, desenvolver adaptadores é papel do desenvolvedor da aplicação. Na Figura A.7, ilustra-se o diagrama de casos de uso relacionados ao desenvolvimento de adaptadores. A descrição dos casos de uso é apresentada a seguir.

**(DD:01) Instanciar adaptador** - Criar uma instância de adaptador informando o *nome* do mesmo.

**(DD:02) Definir componente adaptado** - Associar um componente a um adaptador previamente criado. A partir de então, todos os serviços providos e eventos declarados como adaptados estarão disponíveis.

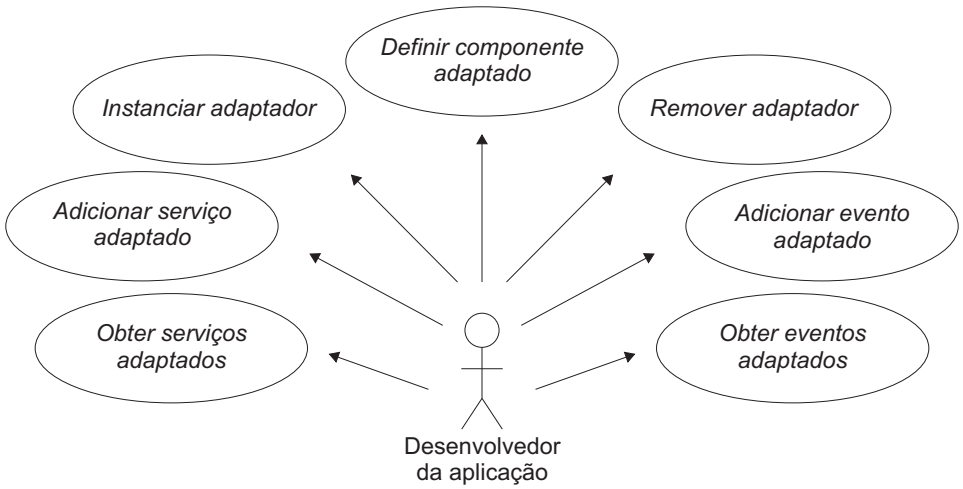



Figura A.7: Casos de uso relacionados à implementação do modelo de adaptadores da CMS.

**(DD:03) Remover adaptador** - Remover um adaptador previamente associado a um componente. Após a remoção, os serviços providos e eventos de interesse declarados como adaptados não estarão mais disponíveis e os originais voltam a funcionar.

**(DD:04) Adicionar serviço adaptado** - Adicionar um serviço provido adaptado ao adaptador. A partir de então, requisições ao serviço do componente associado serão tratadas pelo adaptador.

**(DD:05) Adicionar evento adaptado** - Adicionar um evento de interesse adaptado ao adaptador. A partir de então, notificações de evento do componente associado serão tratadas pelo adaptador.

**(DD:06) Obter serviços adaptados** - Obter o grupo de serviços providos adaptados pelo adaptador.

**(DD:07) Obter eventos adaptados** - Obter o grupo de eventos de interesse adaptados pelo adaptador.

### A.1.4 Casos de Uso: Modelo de Separação de Interesses (DS)

Estes casos de uso estão relacionados ao modelo de separação de interesses de acordo com a CMS. Na Figura A.8, ilustra-se o diagrama de casos de uso relacionados ao desenvolvimento de interes-

ses (*concerns*) para personalização da execução dos componente, na visão do desenvolvedor do componente. A descrição dos casos de uso é apresentada a seguir.

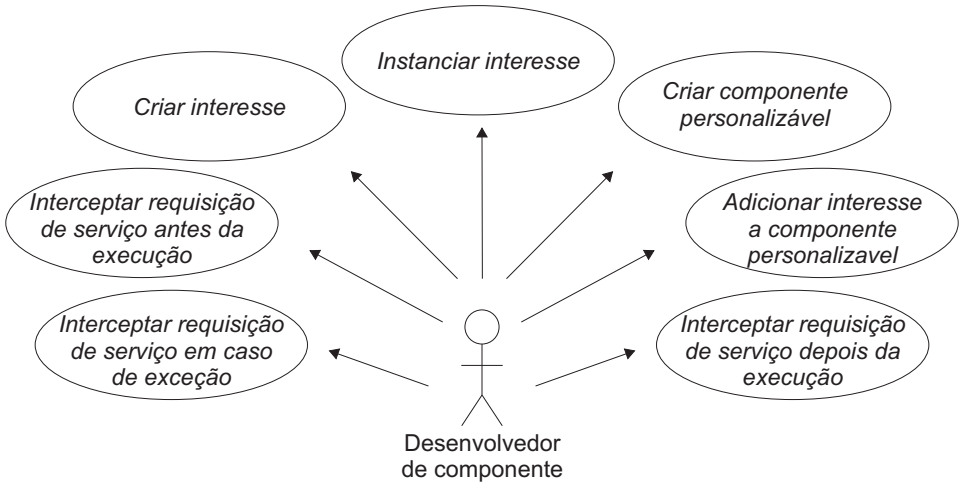



Figura A.8: Casos de uso relacionados à implementação do modelo de separação de interesses da CMS, do ponto de vista do desenvolvedor do componente.

**(DS:01) Criar interesse** - Criar um interesse específico do componente. O arcabouço deve prover uma classe padrão para interesse, a ser estendida pelos interesses específicos.

**(DS:02) Instanciar interesse** - Instanciar um interesse específico.

**(DS:03) Criar componente personalizável** - Criar um componente personalizável específico. O arcabouço deve prover uma classe padrão para componente personalizável, a ser estendida pelos componentes personalizáveis específicos.

**(DS:04) Interceptar requisição de serviço antes da execução** - Implementar o código a ser executado na interceptação do serviço antes da execução do mesmo.

**(DS:05) Interceptar requisição de serviço depois da execução** - Implementar o código a ser executado na interceptação do serviço depois da execução do mesmo.

**(DS:06) Interceptar requisição de serviço em caso de exceção** - Implementar o código a ser executado na interceptação do serviço em caso de exceção.

**(DS:07) Adicionar interesse a componente personalizável** - Adicionar um interesse previamente instanciado a um componente personalizável.

Na Figura A.9, ilustra-se o diagrama de casos de uso relacionados à configuração dos interesses ativados e desativados em um componente, na visão do desenvolvedor da aplicação. A descrição dos casos de uso é apresentada a seguir.

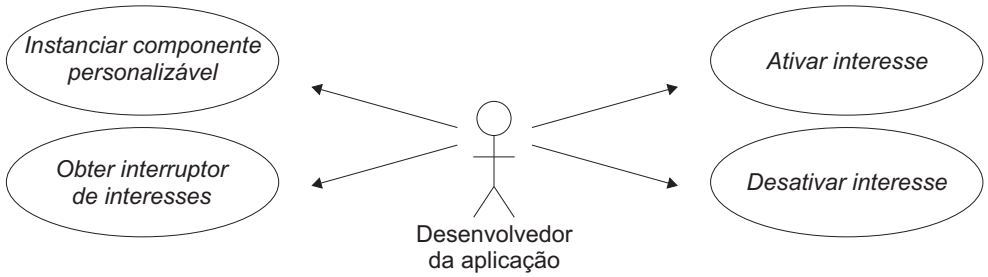



Figura A.9: Casos de uso relacionados à implementação do modelo de separação de interesses da CMS, do ponto de vista do desenvolvedor da aplicação.

**(DS:08) Instanciar componente personalizável** - Criar uma instância de componente personalizável específico.

**(DS:09) Obter interruptor de interesses** - Obter grupo de interesses personalizáveis de um componente.

**(DS:10) Ativar interesse** - Ativar um interesse personalizável do componente.

**(DS:11) Desativar interesse** - Desativar um interesse personalizável do componente.

## A.2 Projeto do Arcabouço

Com base nos casos de uso descritos anteriormente, definiu-se o projeto orientado a objetos do GCF. Na Figura A.10, apresenta-se a visão geral do projeto, com um diagrama simplificado utilizando notação UML [184], apenas para destacar os relacionamentos entre as entidades. Cada parte do projeto será detalhada a seguir, de acordo com a ordem dos casos de uso e níveis de complexidade.

### A.2.1 Núcleo do arcabouço

O núcleo do projeto está relacionado aos mecanismos de composição e interação, referentes aos casos de uso de desenvolvimento de componentes (DC) e desenvolvimento de aplicação (DA),

descritos anteriormente. A base do projeto de classes é o padrão de projeto *Composite* [16], que promove a composição recursiva de objetos, formando uma hierarquia parte-todo, como no caso da arquitetura da CMS.

O diagrama de classes referente ao núcleo do projeto do GCF é ilustrado na Figura A.11. As classes `FunctionalComponent` e `Container` são instanciadas, respectivamente, em componentes funcionais e contêineres, como especificados na CMS. A classe abstrata `AbstractComponent` garante a composição recursiva, permitindo que os contêineres "desconheçam" a implementação dos seus componentes filhos, que podem ser tanto componentes funcionais como outros contêineres. Além disso, ela implementa os métodos comuns às suas subclasses e estabelece um conjunto de contratos como métodos abstratos que devem ser implementados pelas mesmas.

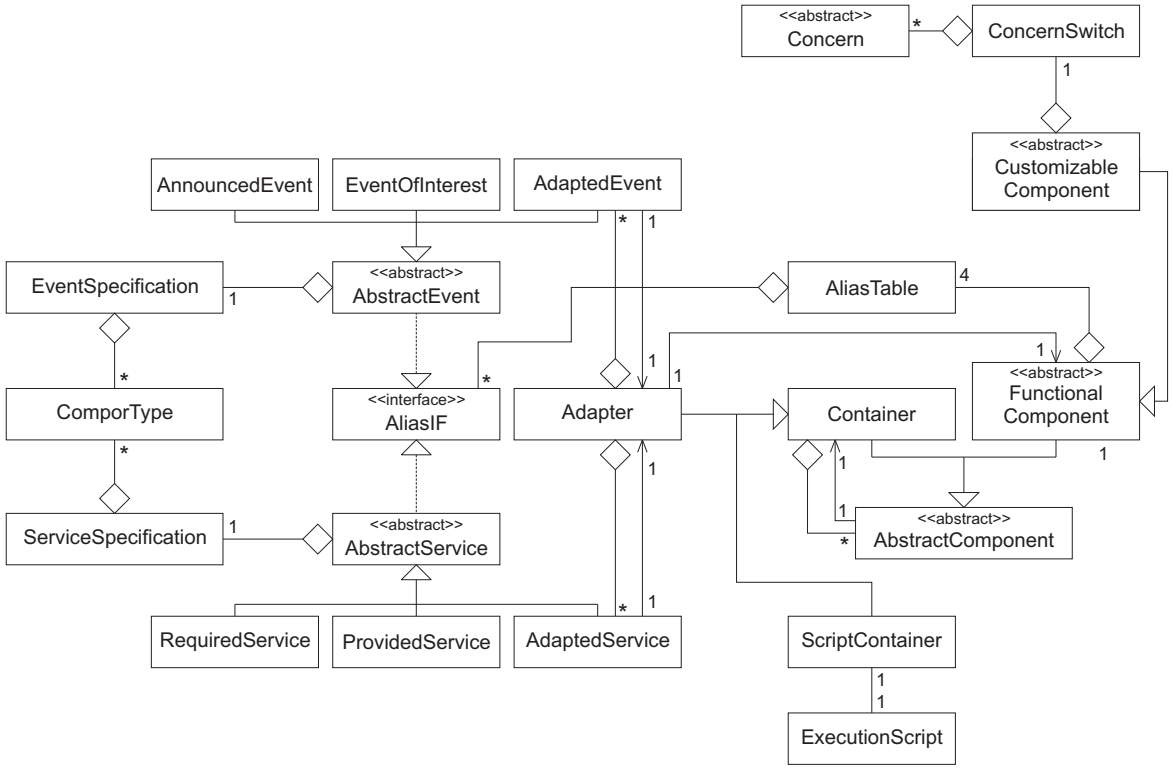



Figura A.10: Diagrama de classes simplificado – Visão geral.

Os métodos abstratos definidos na classe `AbstractComponent` possuem diferentes implementações para `FunctionalComponent` e `Container`. Para fins de explanação, estes métodos estão divididos nos seguintes grupos, na notação de cada classe: *composição*, onde se apresentam os métodos relacionados à composição da hierarquia; *interação*, onde estão relacionados os métodos para interação baseada em serviços e eventos; e *execução*, onde se apresentam os métodos

referentes à execução.

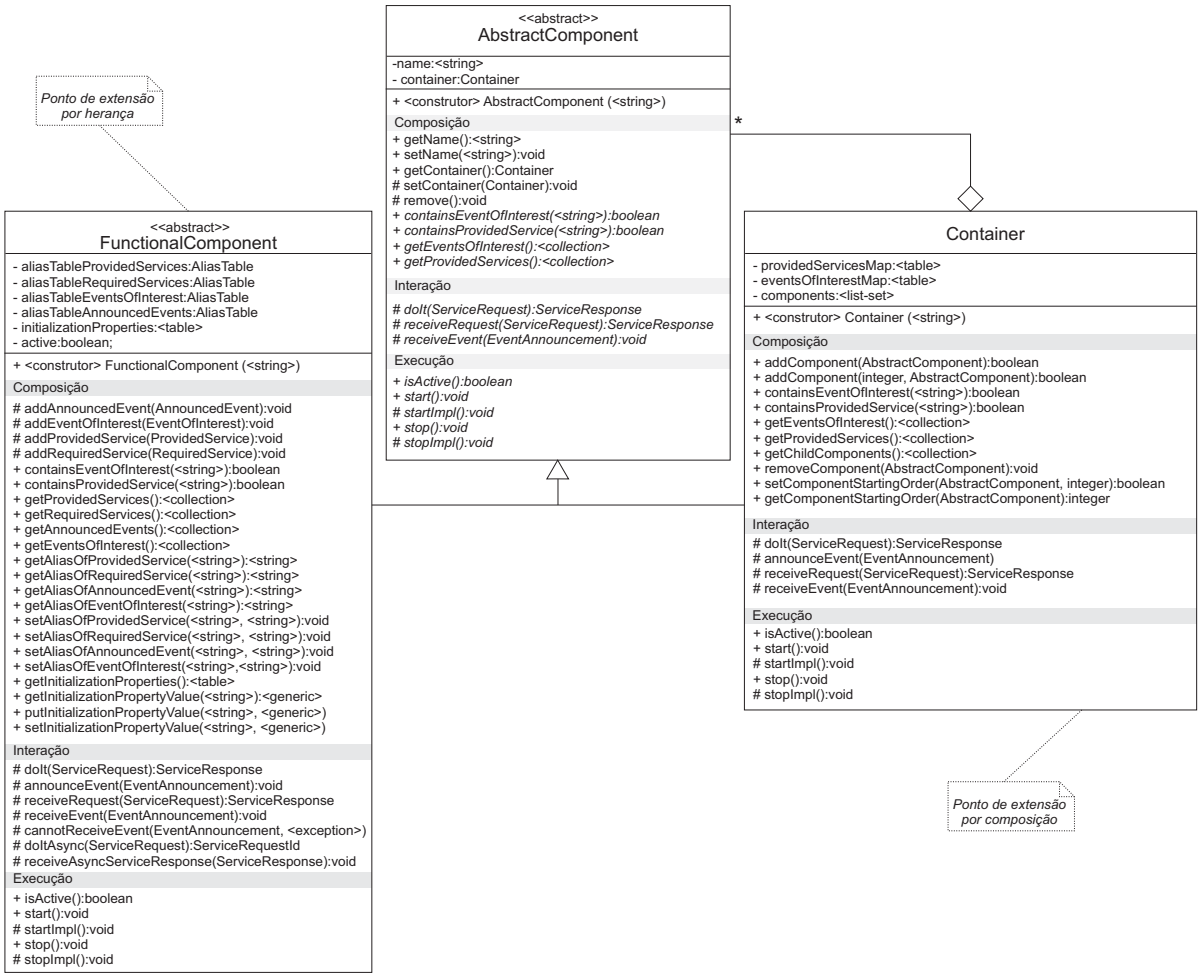



Figura A.11: Diagrama de classes do núcleo do GCF.

Uma vez que cada linguagem possui seu próprio arcabouço de tipos primitivos e estruturas de dados, serão utilizados os estereótipos descritos na Tabela A.1 para representar tipos e estruturas que não fazem parte do projeto do GCF. Estes devem ser mapeados para os tipos e as estruturas específicas, de acordo com a linguagem de programação. As classes que fazem parte do arcabouço referentes aos tipos de atributos, parâmetros e retornos de métodos serão explicadas posteriormente.

Dada a importância das classes `AbstractComponent`, `FunctionalComponent` e `Container` para o projeto do GCF, todos os seus métodos, ilustrados na Figura A.11 serão detalhados a seguir. Nas demais classes do projeto, serão apenas destacados os aspectos principais, considerando que os diagramas de classes são suficientemente explicativos.

| Estereótipo | Descrição |
|---|---|
| <string> | Cadeia de caracteres. |
| <collection> | Grupo de elementos. |
| <table> | Tabela chave-valor. |
| <generic> | Tipo genérico para referência polimórfica. Por exemplo, `Object`, em Java. |
| <class> | Referência a uma classe em tempo de execução. Por exemplo, `Class`, em Java. |
| <method> | Referência a um método em tempo de execução. Por exemplo, `Method`, em Java. |
| <list> | Grupo de elementos ordenados (lista). |
| <set> | Grupo de elementos sem repetição (conjunto). |
| <list-set> | Lista de elementos ordenados e sem repetição (conjunto ordenado). |
| <exception> | Estrutura que encapsula dados sobre exceções de compilação e execução. Por exemplo, `Throwable`, em Java. |

Tabela A.1: Estereótipos referentes a tipos e estruturas de dados.

### A classe `AbstractComponent`

A classe `AbstractComponent` possui apenas dois atributos: *name*, que representa o nome do componente, seja ele contêiner ou componente; e *container*, que representa o seu contêiner pai. Além dos métodos de acesso (*gets* e *sets*), que dispensam maiores explicações, a seguir são descritas as funcionalidades dos outros métodos, com referência aos casos de uso apresentados anteriormente.

***remove():void*** Método concreto. Remove a entidade do seu contêiner pai (casos de uso DA:22 e DA:23).

***containsEventOfInterest(<string>):boolean*** Método abstrato. Verifica se um dado evento de interesse, cujo *apelido* é passado como parâmetro, está registrado nesta entidade. Retorna o resultado da verificação como um valor booleano (caso de uso DA:08).

***containsProvidedService(<string>):boolean*** Método abstrato. Verifica se um dado serviço provido, cujo *apelido* é passado como parâmetro, está registrado nesta entidade. Retorna o resultado da verificação como um valor booleano (caso de uso DA:01).

***getEventsOfInterest():<collection>*** Método abstrato. Retorna o grupo de eventos de interesse da entidade (caso de uso DA:11).

***getProvidedServices():<collection>*** Método abstrato. Retorna o grupo de serviços providos pela entidade (caso de uso DA:04).

***doIt(ServiceRequest):ServiceResponse*** Método abstrato. Invoca um serviço, cujas informações estão encapsuladas em uma instância de `ServiceRequest`. A resposta da invocação é encapsulada em uma instância de `ServiceResponse` (caso de uso DC:12).

***receiveRequest(ServiceRequest):ServiceResponse*** Método abstrato. Recebe uma requisição de serviço, cujas informações estão encapsuladas em uma instância de `ServiceRequest`, executa o serviço e retorna a resposta da invocação encapsulada em uma instância da classe `ServiceResponse` (caso de uso DC:15).

***receiveEvent(EventAnnouncement):void*** Método abstrato. Recebe uma notificação de evento de interesse cujas informações estão encapsuladas em uma instância de `EventAnnouncement` (caso de uso DC:17).

***start():void*** Método abstrato. Inicia a execução da entidade (casos de uso DA:34 e DA:35).

***startImpl():void*** Método abstrato. Implementa o código de inicialização específico definido pelo desenvolvedor da entidade (casos de uso DA:34 e DA:35).

***stop():void*** Método abstrato. Interrompe a execução da entidade (casos de uso DC:36 e DA:37).

***stopImpl():void*** Método abstrato. Implementa o código de interrupção da execução específico definido pelo desenvolvedor da entidade (casos de uso DA:36 e DA:37).

***isActive():boolean*** Método abstrato. Indica se a entidade foi iniciada ou não (casos de uso DA:34, DA:35, DA:36 e DA:37).

### A classe `FunctionalComponent`

A classe abstrata `FunctionalComponent` é o principal ponto de extensão por herança do arcabouço. Todos os componentes desenvolvidos no GCF devem estender a implementação desta

classe (caso de uso DC:01), dando origem a componentes específicos de domínio que serão instanciados pelo desenvolvedor da aplicação (caso de uso DA:18). A classe possui cinco tabelas referentes aos *serviços providos*, *serviços requeridos*, *eventos de interesse*, *eventos anunciados* e *propriedades de inicialização*. Além disso, possui um atributo booleano cujo valor indica se o componente foi iniciado ou não.

A maioria dos métodos está relacionada ao acesso e à alteração dos valores contidos nestas tabelas. Por exemplo, os métodos `addProvidedService(...)`, `addRequiredService(...)`, `addEventOfInterest(...)` e `addAnnouncedEvent(...)` realizam inserções nas respectivas tabelas (casos de uso DC:02, DC:04, DC:06 e DC:08). Os métodos `getProvidedServices()`, `getRequiredServices()`, `getEventsOfInterest()` e `getAnnouncedEvents()` retornam os valores das respectivas tabelas (casos de uso DA:04, DA:07, DA:11 e DA:14). Os métodos `getAlias*` e `setAlias*`, respectivamente, recuperam e redefinem o valor do *apelido* de um dado serviço, cujo *nome* é passado como parâmetro (casos de uso DA:02, DA:03, DA:05, DA:06, DA:09, DA:10, DA:12 e DA:13). Por fim, o método `getInitializationProperties()` retorna o conjunto de propriedades de inicialização do componente (caso de uso DA:16), as quais podem ser criadas, acessadas individualmente e redefinidas através dos métodos `putInitialization Property(name, value)`, `getInitializationProperty(name)` e `setInitializationProperty(name, value)`, respectivamente (casos de uso DC:10, DA:17 e DA:15).

Além dos métodos de acesso supracitados, outros métodos de `FunctionalComponent` são descritos a seguir de forma mais detalhada, dada a importância no funcionamento do arcabouço.

***containsProvidedService(<string>):boolean*** Método concreto. Sobrescreve o método da classe `AbstractComponent`. Verifica a existência do serviço, cujo apelido é passado por parâmetro, na tabela de serviços providos (DA:01).

***containsEventOfInterest(<string>):boolean*** Método concreto. Sobrescreve o método da classe `AbstractComponent`. Verifica a existência do evento, cujo apelido é passado por parâmetro, na tabela de eventos de interesses (DA:08).

***doIt(ServiceRequest):ServiceResponse*** Método concreto. Sobrescreve o método definido na classe `AbstractComponent`. No componente funcional, a implementação deste método simplesmente repassa a requisição do serviço para o contêiner pai através do método `doIt(...)` da classe `Container` (caso de uso DC:12).

***receiveRequest(ServiceRequest):ServiceResponse*** Método concreto. Sobrescreve o método de `AbstractComponent`. No componente funcional, recupera a referência ao método definido pelo desenvolvedor do componente como sendo o implementador do serviço e invoca o método utilizando os parâmetros encapsulados na instância de `ServiceRequest`. Após receber o resultado da execução do método, encapsula este resultado ou a exceção da execução do método em uma instância de `ServiceResponse` (caso de uso DC:15). A invocação do método deve ser realizada utilizando mecanismos de reflexão computacional [35], uma vez que o método não é conhecido *a priori*.

***announceEvent(EventAnnouncement):void*** Método concreto. Anuncia um evento aos interessados da hierarquia de componentes. A implementação desse método cria uma nova linha de execução e repassa o anúncio para o método `announceEvent(...)` da classe `Container` (caso de uso DC:14). A invocação assíncrona pode ser realizada através da implementação do padrão *Future/Active Object* [182].

***receiveEvent(EventAnnouncement):void*** Método concreto. Sobrescreve o método definido na classe `AbstractComponent`. No componente funcional, recupera a referência ao método definido pelo desenvolvedor do componente como sendo o implementador do tratamento do evento e invoca o método utilizando os parâmetros encapsulados na instância da classe `EventAnnouncement` (caso de uso DC:17). A invocação do método deve ser realizada utilizando mecanismos de reflexão computacional uma vez que o método não é conhecido *a priori*.

***cannotReceiveEvent(EventAnnouncement, <exception>):void*** Método concreto que possui implementação vazia. Este método deve ser sobrescrito pelo componente sendo desenvolvido, caso este queira tratar eventos que, por algum motivo, não puderam ser recebidos. As informações do evento são encapsuladas em uma instância de `EventAnnouncement` assim como informações do erro ocorrido, encapsulada em uma instância de `<exception>` (caso de uso DC:17).

***doItAsync(ServiceRequest):ServiceRequestId*** Método concreto. Cria uma nova linha de execução na qual é executada uma chamada ao método `doIt(...)` implementado na própria classe `FunctionalComponent`. Retorna um identificador da requisição encapsulado em

uma instância de `ServiceRequestId` que é utilizado para identificar a resposta assíncrona na implementação do método `receiveAsyncServiceResponse(...)`. A invocação assíncrona pode ser realizada através da implementação do padrão *Future/Active Object* [182] (caso de uso DC:13).

***receiveAsyncServiceResponse(ServiceResponse):void*** Método concreto. Possui implementação vazia na classe `FunctionalComponent`. As classes dos componentes que estendem esta classe devem sobrescrever este método caso queiram tratar respostas de invocações assíncronas realizadas pelo método `doItAsync(...)`. A resposta vem encapsulada em uma instância de `ServiceResponse`, a qual também armazena o identificador da requisição (caso de uso DC:16).

***start():void*** Método concreto. Sobrescreve o método de `AbstractComponent`. No componente funcional, verifica se o componente já está ativo. Caso não esteja, executa o método `startImpl()` e depois define o valor do atributo *active* para *verdadeiro* (caso de uso DA:34).

***startImpl():void*** Método concreto. Sobrescreve o método de `AbstractComponent`. No componente funcional, este método tem implementação vazia. As classes dos componentes que estendem `FunctionalComponent` devem sobrescrever este método caso possuam *script* de inicialização (caso de uso DA:34).

***stop():void*** Método concreto. Sobrescreve o método de `AbstractComponent`. No componente funcional, verifica se o componente já está inativo. Caso não esteja, executa o método `stopImpl()` e depois define o valor do atributo *active* para *falso* (caso de uso DC:36).

***stopImpl():void*** Método concreto. Sobrescreve o método de `AbstractComponent`. No componente funcional, este método tem implementação vazia. As classes dos componentes que estendem `FunctionalComponent` devem sobrescrever este método caso possuam *script* de interrupção de execução (caso de uso DA:36).

***isActive():boolean*** Método concreto. No componente funcional, este método simplesmente retorna o valor do atributo booleano *active* (casos de uso DA:34 e DA:36).

### A classe `Container`

Por fim, a classe `Container` é o principal ponto de extensão por composição do GCF. Nesta classe é implementada a gerência da composição das instâncias de `FunctionalComponent` e das próprias instâncias de `Container`. A classe possui três atributos: `providedServicesMap`, `eventsOfInterestMap` e `components`. Os dois primeiros são tabelas chave-valor contendo os serviços providos e eventos de interesse das suas entidades filhas, sejam elas componentes funcionais ou outros contêineres, o que fica abstraído pela interface de `AbstractComponent`. O último atributo é uma lista das entidades filhas, sem repetição, que define a ordem em que as mesmas devem ser inicializadas.

Componentes funcionais ou outros contêineres são adicionados a uma instância de `Container` através do método sobrecarregado `addComponent(...)`, sendo o parâmetro inteiro referente à ordem de inicialização da entidade sendo inserida (casos de uso DA:20 e DA:21). A implementação deste método deve atualizar as tabelas de serviços providos e eventos de interesse da instância do contêiner e repassar a atualização para o seu contêiner pai para que a atualização ocorra até o contêiner raiz. O mesmo processo de atualização deve ocorrer na remoção de componentes, implementada no método `removeComponent(...)` (casos de uso DA:22 e DA:23).

Além de alguns métodos de acesso (`getEventsOfInterest()`, `getProvidedServices()` e `getChildComponents()`, casos de uso DA:28, DA:29 e DA:30) e dos métodos de obtenção e definição da ordem de inicialização (`getComponentStartingOrder()` e `setComponentStartingOrder(...)`, casos de uso DA:26 e DA:27), os quais dispensam maiores explicações, os seguintes métodos são descritos mais detalhadamente devido à importância no funcionamento do arcabouço.

***containsProvidedService(<string>):boolean*** Método concreto. Sobrescreve o método da classe `AbstractComponent`. Verifica a existência do serviço, cujo apelido é passado por parâmetro, na tabela de serviços providos (DA:25).

***containsEventOfInterest(<string>):boolean*** Método concreto. Sobrescreve o método da classe `AbstractComponent`. Verifica a existência do evento, cujo apelido é passado por parâmetro, na tabela de eventos de interesses (DA:24).

***doIt(ServiceRequest):ServiceResponse*** Método concreto. Sobrescreve o método definido na classe `AbstractComponent`. No contêiner, a implementação deste método verifica se o serviço

requisitado está registrado na tabela de serviços providos. Se sim, repassa o serviço para o seu provedor, através da chamada do método `receiveRequest(...)`. Caso contrário, repassa a requisição ao contêiner pai (caso de uso DC:12).

***receiveRequest(ServiceRequest):ServiceResponse*** Método concreto. Sobrescreve o método de `AbstractComponent`. No contêiner, verifica quem é o implementador do serviço requisitado realizando busca na tabela de serviços providos e repassa a requisição ao mesmo através do método `receiveRequest(...)` (caso de uso DC:15).

***announceEvent(AbstractComponent, EventAnnouncement):void*** Método concreto. Anuncia um evento aos interessados da hierarquia de componentes. A implementação desse método na classe `Container` verifica a existência de interessados na tabela de eventos de interesse e repassa o anúncio para os mesmos, através do método `receiveEvent(...)`. Além disso, encaminha o evento para o contêiner pai através da chamada recursiva do método `announceEvent(...)` para notificar outros interessados no evento. O primeiro parâmetro é o originador do evento (caso de uso DC:14).

***receiveEvent(EventAnnouncement):void*** Método concreto. Sobrescreve o método definido na classe `AbstractComponent`. No contêiner, verifica quem é o implementador do tratamento do evento realizando busca na tabela de eventos de interesse e repassa a requisição ao mesmo através do método `receiveEvent(...)` (caso de uso DC:17).

***start():void*** Método concreto. Sobrescreve o método de `AbstractComponent`. No contêiner, verifica se o componente já está ativo. Caso não esteja, executa o método `startImpl()` e depois define o valor do atributo *active* para *verdadeiro* (caso de uso DA:35).

***startImpl():void*** Método concreto. Sobrescreve o método de `AbstractComponent`. No contêiner, este método inicia todas as entidades filhas na ordem de inicialização definida pela lista de componentes através da chamada do método `start()`(caso de uso DA:35).

***stop():void*** Método concreto. Sobrescreve o método de `AbstractComponent`. No contêiner, verifica se o componente está inativo. Caso não esteja, executa o método `stopImpl()` e depois define o valor do atributo *active* para *falso* (caso de uso DC:37).

***stopImpl():void*** Método concreto. Sobrescreve o método de `AbstractComponent`. No contêiner, este método interrompe a execução de todas as entidades filhas na ordem inversa da inicialização definida pela lista de componentes através da chamada do método `stop()` (caso de uso DA:37).

***isActive():boolean*** Método concreto. No contêiner, este método retorna um valor referente à conjunção (*AND*) dos valores obtidos da chamada do método `isActive()` em todos as suas entidades filhas. Ou seja, um contêiner estará ativo apenas quando todas as suas entidades filhas estiverem ativas (casos de uso DA:35 e DA:37).

**Detalhando a implementação dos mecanismos de interação**

Através da implementação das classes e métodos descritos anteriormente são desenvolvidos os modelos de inserção, remoção e alteração de componentes, assim como os modelos de interação baseada em serviços e eventos. Em resumo, a implementação dos mecanismos de interação funciona como descrito a seguir.

A interação baseada em serviços é implementada através de invocações sucessivas do método `doIt`, "de baixo para cima" na hierarquia de componentes até que o contêiner que registra o serviço seja encontrado. Quando isto acontece, o método `receiveRequest` é invocado "de cima para baixo" na hierarquia até que o componente funcional que implementa o serviço seja encontrado, como ilustrado na Figura A.12. Caso o contêiner raiz seja alcançado e o provedor do serviço não seja encontrado, o arcabouço deve retornar um erro ao desenvolvedor. As exceções de execução do GCF são apresentadas posteriormente. Este processo é baseado na definição de invocação de serviço descrita no Capítulo 4.

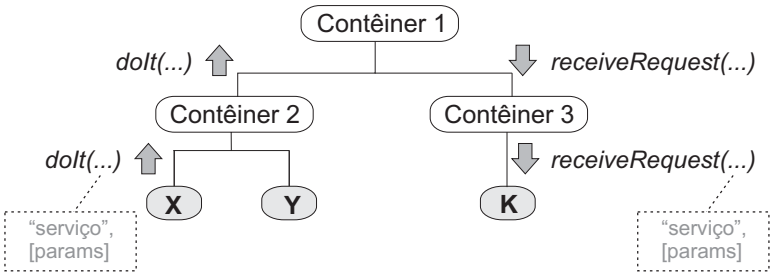



Figura A.12: Interação entre objetos no modelo de interação baseada em serviços do GCF.

O mesmo processo ocorre na interação baseada em eventos, com invocações sucessivas do mé-

todo `announceEvent`, de “baixo para cima” na hierarquia de componentes, até que o componente raiz seja encontrado (ver Figura A.13). A cada contêiner, verifica-se o registro de interessados no evento e o método `receiveEvent` é invocado “de cima para baixo” na hierarquia, até que todos os componentes funcionais interessados no evento sejam notificados. Este processo é baseado na definição de anúncio de evento descrita no Capítulo 4.

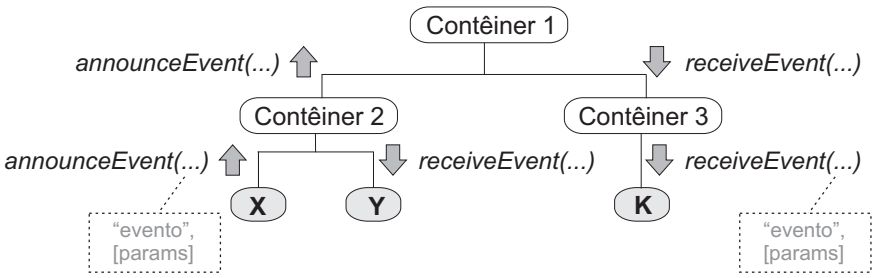



Figura A.13: Interação entre objetos no modelo de interação baseada em eventos do GCF.

### Classes relacionadas à execução

Na Figura A.14, são ilustradas as classes relacionadas à execução de aplicações utilizando o GCF. A classe `ScriptContainer` representa a raiz da hierarquia de uma aplicação baseada na CMS. Esta classe estende a classe `Container` apenas para acrescentar um atributo do tipo `ExecutionScript`. Além dos métodos de acesso a este atributo, `ScriptContainer` sobrescreve o método `announceEvent(...)`, para que este repasse eventos anunciados na hierarquia também para o `script` de execução.

A outra classe ilustrada na Figura A.14 é `ExecutionScript`. Esta pode ser considerada a *classe principal* de toda aplicação construída com o GCF. Ela possui um método `main()` que deve ser sobrescrito pelo desenvolvedor caso haja algum código que ele considere não “componentizável”, como por exemplo, uma interface gráfica específica da aplicação (casos de uso DA:32 e DA:33). Os métodos `init()` e `finish()` inicializam e interrompem a execução do `script` (casos de uso DA:43 e DA:44), o que é refletido diretamente na hierarquia de componentes através do contêiner raiz. A inicialização e finalização são registradas no atributo `active`, cujo valor pode ser recuperado através do método `isActive()`. Além disso, alguns métodos considerados importantes para o funcionamento do arcabouço são detalhados a seguir.

***exec(ServiceRequest):ServiceResponse*** Executa a requisição de um serviço junto à hierarquia de

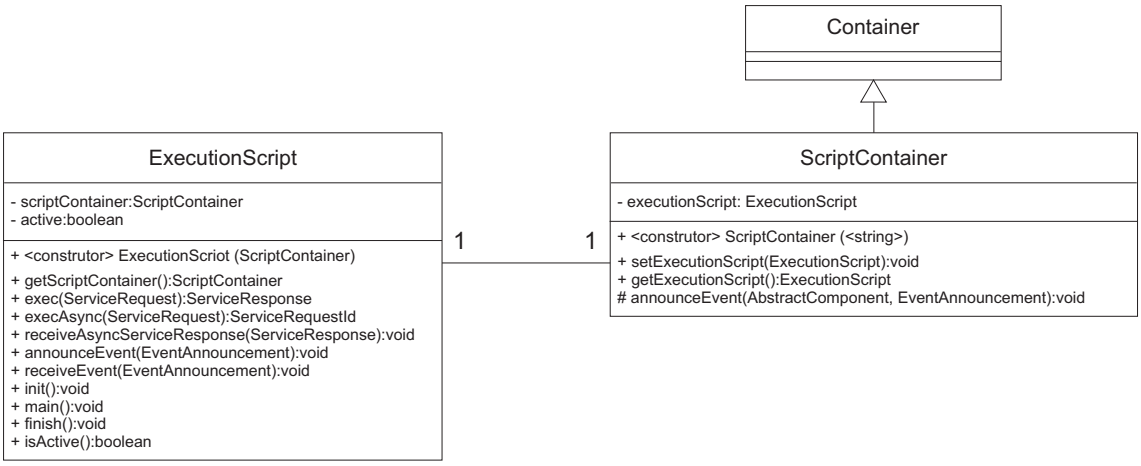



Figura A.14: Diagrama de classes do GCF, foco na execução.

componentes através da referência ao contêiner raiz (caso de uso DA:39).

***execAsync(ServiceRequest):ServiceRequestId*** Executa a requisição de um serviço à hierarquia de componentes através da referência ao contêiner raiz em uma nova linha de execução, assim como o método `doItAsync(...)` da classe `FunctionalComponent` (caso de uso DA:40).

***receiveAsyncServiceResponse(ServiceResponse):void*** Recebe a resposta de uma execução assíncrona de serviço. Este método tem implementação padrão vazia e deve ser sobrescrito caso o desenvolvedor utilize chamadas assíncronas e queira tratar as respostas das mesmas. A resposta vem encapsulada em uma instância de `ServiceResponse`, a qual também armazena o identificador da requisição (caso de uso DC:41).

***announceEvent(EventAnnouncement):void*** Anuncia um evento aos interessados da hierarquia de componentes através da referência ao contêiner raiz (caso de uso DA:38).

***receiveEvent(EventAnnouncement):void*** Recebe os eventos anunciados na hierarquia. Este método tem implementação padrão vazia e deve ser sobrescrito caso o desenvolvedor necessite receber e tratar os eventos recebidos através do contêiner raiz (caso de uso DA:42).

**Classes auxiliares**

Na descrição das classes principais do núcleo, fez-se referência a algumas classes do GCF utilizadas como tipos de atributos de classes, parâmetros e retornos de métodos. Estas classes são aqui denominadas auxiliares pois servem apenas para encapsular dados em objetos para facilitar o entendimento e diminuir o acoplamento do arcabouço a tipos específicos de linguagem.

Na Figura A.15, ilustra-se o diagrama de classes auxiliares relacionadas ao modelo de interação baseada em serviços. A maioria dos métodos destas classes são métodos de acesso, dispensando maiores explicações. A especificação de serviços é encapsulada em instâncias de `ServiceSpecification`. Serviços providos e requeridos são implementados por instâncias da classe `ProvidedService` e `RequiredService`, cujos métodos em comum são implementados na classe `AbstractService`. A classe `ProvidedService` armazena uma referência ao método definido pelo usuário como implementador do serviço, o que pode ser realizado utilizando mecanismos de reflexão computacional [35].

Além das classes relacionadas a serviços, a classe utilitária `AliasTable` implementa uma tabela de três colunas (nome, apelido e valor), armazenando instâncias do tipo `AliasIF`. Esta tabela é utilizada para armazenar serviços providos, serviços requeridos, eventos de interesse e eventos anunciados na classe `FunctionalComponent`. `AliasTable` provê métodos para busca de apelidos, nomes e valores, facilitando a recuperação destas informações. Por fim, a classe `ComporType` encapsula informações sobre um tipo, seja de parâmetro, de retorno ou de exceção.

Na Figura A.16, ilustra-se o diagrama de classes auxiliares relacionadas ao modelo de interação baseada em eventos. Assim como as classes relacionadas a serviços, a maioria dos métodos destas classes são métodos de acesso, dispensando maiores explicações. A especificação de eventos é encapsulada em instâncias de `EventSpecification`. Eventos de interesse e anunciados são implementados por instâncias da classe `EventOfInterest` e `AnnouncedEvent`, cujos métodos em comum são implementados na classe `AbstractEvent`. A classe `EventOfInterest` armazena uma referência ao método definido pelo usuário como implementador do tratamento do evento, o que também pode ser realizado utilizando mecanismos de reflexão computacional.

Por fim, algumas classes auxiliares são utilizadas para encapsular os dados de requisição de serviços e anúncio de eventos. Estas classes são descritas a seguir.

- `ServiceRequestId`, que encapsula um número inteiro como identificador único de requi-

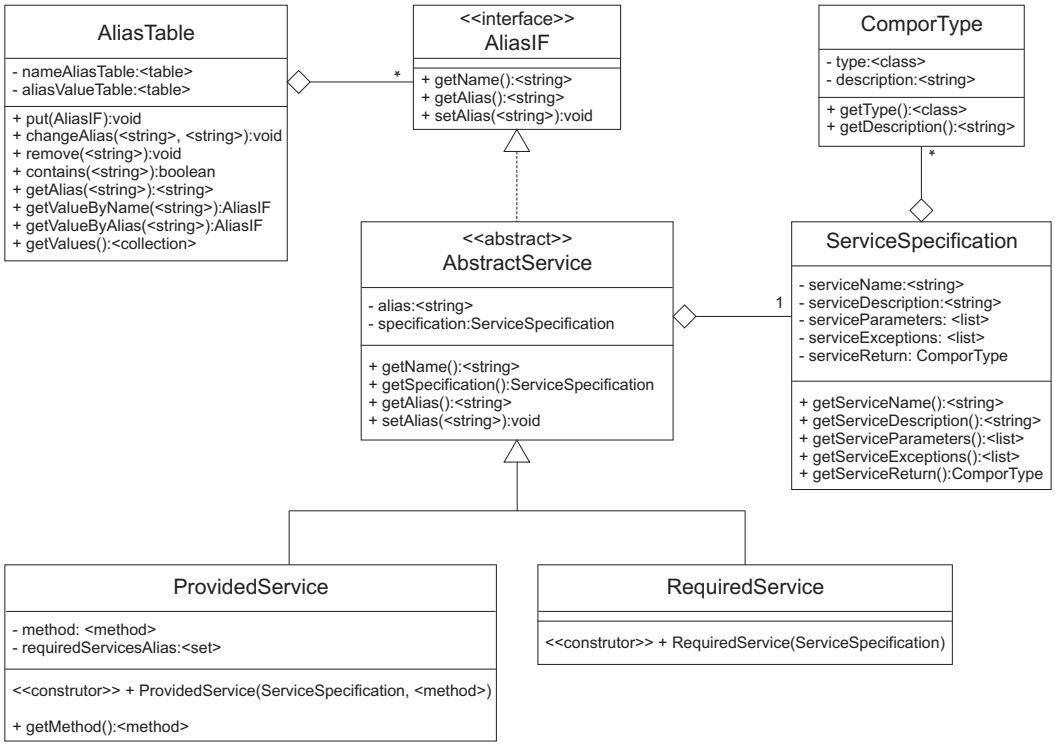



Figura A.15: Diagrama de classes auxiliares – Serviços.

sição de serviço;

- `ServiceRequest`, que encapsula um nome e um apelido de serviço do tipo `<string>`, uma lista de parâmetros (`<list>`) e um identificador de requisição (`ServiceRequestId`);
- `ServiceResponse`, que encapsula o retorno de uma requisição do tipo `<generic>`, um identificador de requisição `ServiceRequest` e um objeto referente a uma possível exceção ocorrida na execução do serviço (`<exception>`);
- `EventAnnouncement`, que encapsula um nome e um apelido de evento do tipo `<string>` e uma lista de parâmetros (`<list>`).

**Gerenciamento de exceções**

A utilização de um arcabouço de software se torna mais complicada quando as mensagens de erro são pouco informativas ou muito genéricas, como por exemplo, exceção de *ponteiro nulo* ou *divisão por zero*. Nestes casos, fica muito difícil identificar a causa do erro e, conseqüentemente, corrigi-lo.

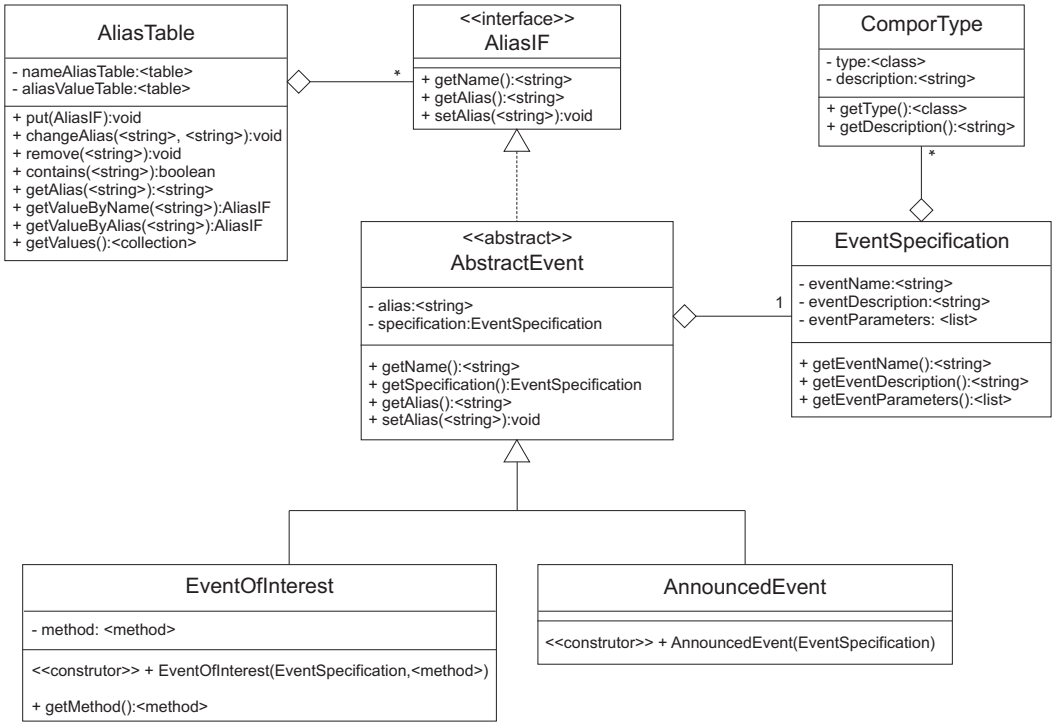



Figura A.16: Diagrama de classes auxiliares – Eventos.

A maioria das linguagens orientadas a objetos fornecem mecanismos para criação e manipulação de exceções específicas da aplicação. Isto possibilita a definição de mensagens mais informativas, facilitando assim o desenvolvimento.

No GCF, define-se um conjunto de exceções para facilitar a utilização das classes e métodos do arcabouço. Estas exceções são descritas na Tabela A.2.

De acordo com o mecanismo de gerenciamento de exceção da linguagem de implementação do arcabouço, as exceções descritas na tabela podem ser de compilação ou de execução. O desenvolvimento com exceções de tempo de execução demanda menos esforço de codificação, porém tornam o código mais suscetível a erros de programação pois não exigem o tratamento prévio por parte do programador.

A nomenclatura utilizada para as exceções segue o padrão da *Sun Microsystems* para a linguagem Java. Esta padronização tem como objetivo facilitar a transição do desenvolvedor de uma implementação do GCF para outra. A forma de implementação de cada uma destas exceções depende das características de cada linguagem. No GCF, apenas seus nomes e a situação em que elas ocorrem são definidas, como descrito na Tabela A.2.

| **Exceção** | **Ocorre quando...** |
|---|---|
| *AbstractComponentNullNameException* | Nome de componente funcional ou contêiner sendo criado é *nulo*. |
| *AliasAlreadyInUseException* | Ao redefinir um apelido, este já se encontra em uso por outro serviço ou evento. |
| *AliasNullException* | Define-se apelido *nulo*. |
| *AnnouncedEventNotDeclaredException* | Evento anunciado não foi previamente declarado. |
| *ComponentNotStartedException* | Serviço provido é requisitado antes do componente ser iniciado. |
| *EventNullNameException* | Nome de evento sendo criado é *nulo*. |
| *IllegalAccessMethodException* | Método implementador de um serviço provido não está acessível (privado, por exemplo). |
| *InvalidEventMethodException* | Método implementador de recepção de evento não é válido (privado, por exemplo). |
| *InvalidInitializationPropertyException* | Definição de valor de propriedade inexistente. |
| *InvalidParametersException* | Parâmetros passados na requisição de um serviço são inválidos. |
| *RequiredServiceNotDeclaredException* | Invocação de um serviço requerido que não foi previamente declarado. |
| *ServiceNotImplementedException* | Serviço requisitado não é implementado por nenhum componente da hierarquia. |
| *ServiceNullNameException* | Serviço tem nome *nulo*. |

Tabela A.2: Exceções do GCF.

### A.2.2 Adaptadores

Na Figura A.17, ilustra-se o diagrama de classes relacionadas ao projeto do modelo de adaptadores definido na CMS. A principal classe do diagrama é a classe `Adapter`, que é uma subclasse de `Container` com uma referência direta a uma instância de `FunctionalComponent` – o componente funcional sendo adaptado.

Além disso, a classe `Adapter` possui tabelas para serviços e eventos adaptados, representadas pelos atributos *adaptedServices* e *adaptedEvents*. Nestas tabelas são armazenadas instâncias das classes `AdaptedService` e `AdaptedEvent`. Estas classes possuem estruturas semelhantes, com referência ao adaptador e um método a ser implementado pelo desenvolvedor para adaptar serviços e adaptar eventos: `invokeService(...)` e `invokeEvent(...)`, respectivamente.

Além dos métodos de acesso definidos na classe `Adapter` (`setFunctionalComponent(...)`, `getAdaptedServices()` e `getAdaptedEvents()`, casos de uso DD:02, DD:06 e DD:07) os seguintes métodos desta classe são descritos com mais detalhes devido à importância no funcionamento do arcabouço.

***invokeOriginalService(ServiceRequest):ServiceResponse*** Invoca o serviço sendo adaptado. Este método pode ser utilizado na implementação do método `invokeService(...)`, como forma de acessar a implementação original do serviço.

***invokeOriginalEvent(EventAnnouncement):void*** Invoca o evento sendo adaptado. Este método pode ser utilizado na implementação do método `invokeEvent(...)`, como forma de acessar a implementação original do evento.

***addAdaptedService(AdaptedService):void*** Adiciona um serviço adaptado a um adaptador. A partir de então, requisições ao serviço serão repassadas à implementação definida na instância de `AdaptedService` e não mais no componente funcional (caso de uso DD:04).

***addAdaptedEvent(AdaptedEvent):void*** Adiciona um evento adaptado a um adaptador. A partir de então, requisições ao evento serão repassadas à implementação definida na instância de `AdaptedEvent` e não mais no componente funcional (caso de uso DD:05).

***receiveRequest(ServiceRequest):ServiceResponse*** Sobrescreve o método da classe `Container` para redirecionar requisições de serviços adaptados às suas respectivas implementações.

***receiveEvent(EventAnnouncement):void*** Sobrescreve o método da classe `Container` para redirecionar notificações de eventos adaptados às suas respectivas implementações.

***remove():void*** Remove o adaptador do seu componente funcional adaptado. A alteração tem que ser refletida no contêiner pai, que passará a encaminhar as requisições novamente para o componente funcional (caso de uso DD:03).

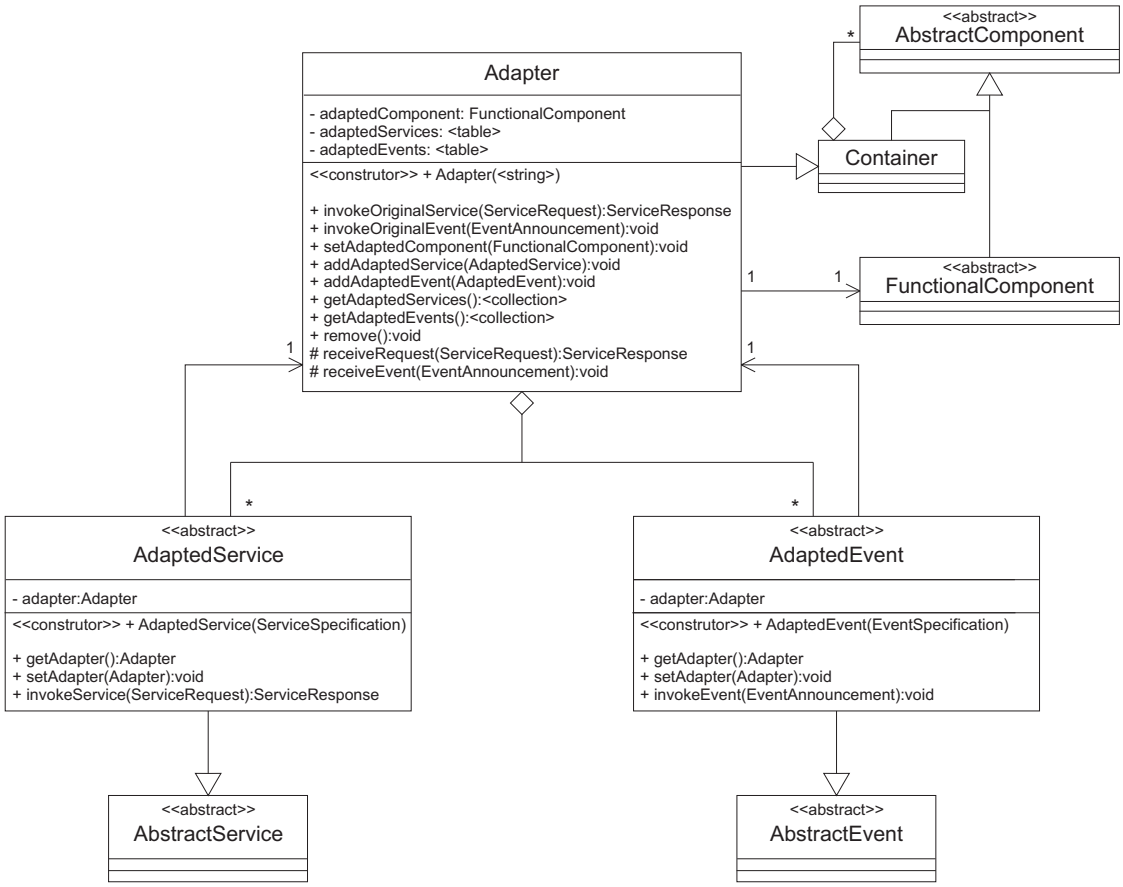



Figura A.17: Diagrama de classes relacionadas ao modelo de adaptadores.

### A.2.3 Separação de interesses

Na Figura A.18, ilustra-se o diagrama de classes relacionadas ao projeto do modelo de separação de interesses da CMS. A classe principal deste diagrama é a classe `CustomizableComponent`, que representa um componente personalizável de acordo com a CMS. Isto significa que uma instância deste componente possui um *interruptor de interesses*, representado pela composição de

uma instância da classe `ConcernSwitch`. Cada interesse é implementado como uma extensão da classe `Concern` e pode ser ativado ou desativado através dos métodos da classe `ConcernSwitch`.

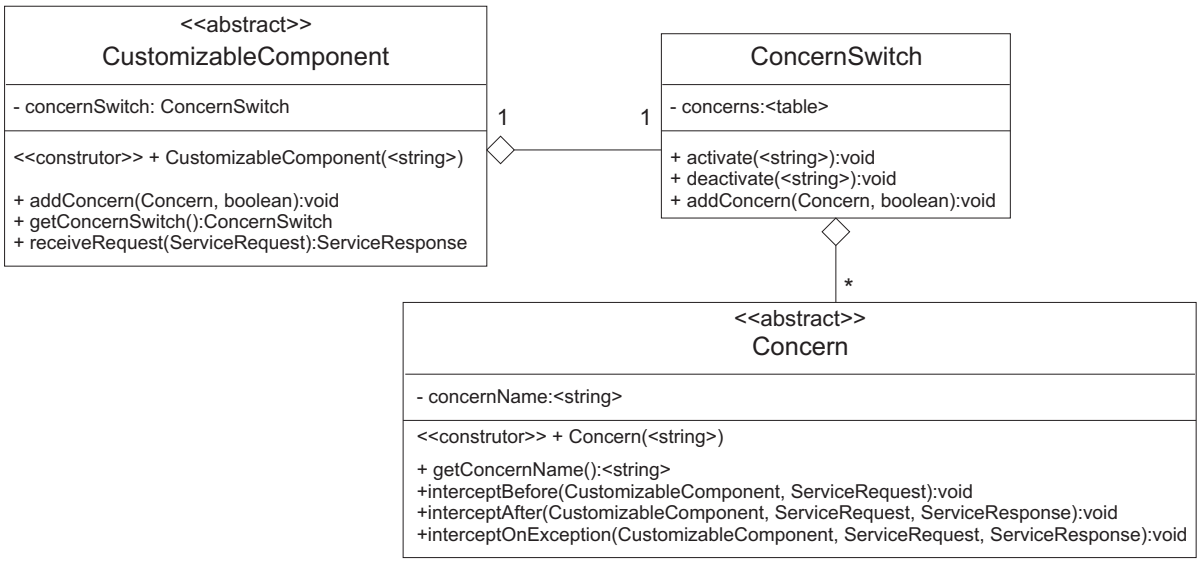



Figura A.18: Diagrama de classes relacionadas ao modelo de separação de interesses.

Com exceção dos métodos de acesso, a seguir são descritas as três classes e seus respectivos métodos como forma de detalhar o projeto do modelo de separação de interesses do GCF.

#### `CustomizableComponent`

Assim como ocorre com `FunctionalComponent`, a classe `CustomizableComponent` deve ser estendida para a criação de um componente específico da aplicação. A única diferença entre essas duas classes é a presença do interruptor de aspectos, que permite a inserção de interesses para personalização da execução dos componentes. Ao ser instanciado pelo desenvolvedor da aplicação (caso de uso DS:08), pode-se obter os interesses disponíveis para a ativação/desativação através do método `getConcernSwitch()` (caso de uso DS:09), personalizando assim a execução do componente. Os principais métodos desta classe são descritos a seguir.

***addConcern(Concern):void*** Adiciona um novo interesse a este componente personalizável. Este método simplesmente repassa a requisição de adição ao método `addConcern(...)` da classe `ConcernSwitch`, através do atributo *concernSwitch* (caso de uso DS:07).

***receiveRequest(ServiceRequest):ServiceResponse*** Sobrescreve o método `receiveRequest(..)` de `FunctionalComponent` para adicionar o comportamento de verificação de interesses

ativos. Para cada interesse ativo recuperado a partir do `ConcernSwitch`, a requisição é repassada à instância do *interesse* através dos métodos `interceptBefore(...)`, `interceptAfter(...)` e `interceptOnException(...)`. Este método realiza uma "decoração" no comportamento de `FunctionalComponent`, tal qual descrito no padrão *Decorator* [16]. Independente da sintaxe da linguagem, o que deve ocorrer é a seguinte seqüência de passos:

1. Recuperar interesses ativos a partir do interruptor;
2. Para cada interesse ativo, invocar `interceptBefore(...)`;
3. Executar `receiveRequest(...)` da super classe (`FunctionalComponent`);
4. Caso haja erro na execução, invocar `interceptOnException(...)` para cada interesse ativo;
5. Caso não haja erro na execução, invocar `interceptAfter(...)` para cada interesse ativo.

**`ConcernSwitch`**

Como descrito anteriormente, o `ConcernSwitch` é o interruptor de interesses. O desenvolvedor não precisa ter acesso à instância do interruptor, uma vez que esta fica encapsulada dentro da classe `CustomizableComponent`. Os principais métodos de `ConcernSwitch` são descritos a seguir.

***addConcern(Concern, boolean):void*** Adiciona um novo interesse à tabela de interesses do interruptor. O atributo booleano se refere ao estado inicial de ativação do interesse: *true*, ativado; *false*, desativado (caso de uso DS:07).

***activate(<string>):void*** Ativa um determinado interesse registrado no interruptor. A partir de então as requisições a serviços interceptadas pela classe `CustomizableComponent` serão redirecionadas para tal interesse (caso de uso DS:10);

***deactivate(<string>):void*** Desativa um determinado interesse registrado no interruptor. A partir de então as requisições a serviços interceptadas pela classe `CustomizableComponent` não serão mais redirecionadas para tal interesse (caso de uso DS:11);

**`Concern`**

A classe `Concern` é utilizada para a implementação dos interesses que poderão ser ativados e desativados em um dado componente personalizável. Ao interceptar as requisições de serviços, a classe `CustomizableComponent` repassa o fluxo de execução para os interesses ativados. O desenvolvedor do componente personalizável deve estender a classe `Concern` para criar interesses específicos, sobrescrevendo os métodos que considerar necessários (casos de uso DS:01 e DS:02). Os principais métodos da classe `Concern` são descritos a seguir.

***interceptBefore(CustomizableComponent, ServiceRequest):void*** Método concreto. Deve ser sobrescrito pelo desenvolvedor caso o interesse possua algum código a ser executado antes da execução de um dado serviço. Os parâmetros do método permitem ao desenvolvedor acessar informações sobre o serviço sendo requisitado e sobre o componente que o provê (caso de uso DS:04).

***interceptAfter(CustomizableComponent, ServiceRequest, ServiceResponse):void*** .

Método concreto. Deve ser sobrescrito pelo desenvolvedor caso o interesse possua algum código a ser executado após a execução de um dado serviço. Os parâmetros do método permitem ao desenvolvedor acessar informações sobre o serviço sendo requisitado, o resultado da execução do mesmo e sobre o componente que o provê (caso de uso DS:05).

***interceptOnException(CustomizableComponent, ServiceRequest, ServiceResponse):void*** .

Método concreto. Deve ser sobrescrito pelo desenvolvedor caso o interesse possua algum código a ser executado ao ocorrer um erro na execução de um dado serviço. Os parâmetros do método permitem ao desenvolvedor acessar informações sobre o serviço sendo requisitado, sobre o erro ocorrido (encapsulado na instância de `ServiceResponse`) e sobre o componente que o provê (caso de uso DS:06).

# Apêndice B

# Implementações de Arcabouços de Componentes

Neste apêndice são detalhadas as implementações da CMS, ou seja, as versões Java, Python, C++ e CSharp do arcabouço GCF descrito no Apêndice A. Uma vez que o projeto destes arcabouços é o mesmo do GCF, neste capítulo apresentam-se apenas os aspectos de implementação específicos de cada linguagem.

Mais especificamente, discute-se como são implementados os mecanismos de reflexão computacional necessários para o funcionamento básico dos arcabouços e os mecanismos para execução assíncrona, utilizados para a interação baseada em eventos e a invocação assíncrona de serviços.

Além disso, apresenta-se um roteiro de como utilizar as principais funcionalidades de cada arcabouço. Para isso, será utilizada como exemplo a arquitetura de aplicação ilustrada na Figura B.1. Nesta arquitetura, tem-se o contêiner raiz, denominado “Root”, contêineres relacionados ao domínio de impressão (“Printing”) e formatação (“Formatting”), nos quais estão presentes os componentes “My Component” e “Name Formatter”.

Apesar deste roteiro ser específico para cada linguagem, com sua respectiva sintaxe, a implementação do arcabouço em outras linguagens deve tentar ao máximo seguir o mesmo roteiro, facilitando o trabalho do desenvolvedor quando necessitar mudar a linguagem de implementação.

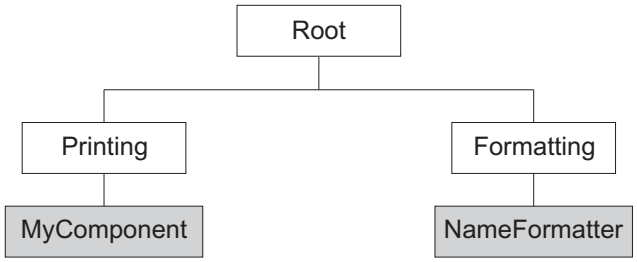



Figura B.1: Exemplo de arquitetura de aplicação.

## B.1 *Java Component Framework* (JCF)

### B.1.1 Implementação do JCF

A riqueza de funcionalidades e documentação da API padrão da linguagem Java facilitaram a implementação do JCF. Como descrito a seguir, a própria API da linguagem disponibiliza classes e métodos para a implementação da execução assíncrona e dos mecanismos de reflexão. Da mesma forma, os estereótipos de tipos apresentados no Apêndice A possuem implementação direta na API padrão de Java, como descrito na Tabela B.1.

| **Estereótipo** | **Descrição** |
|---|---|
| <string> | Classe `java.lang.String`. |
| <collection> | Interface `java.util.Collection` implementada pela classe `java.util.ArrayList`. |
| <table> | Interface `java.util.Map` implementada por `java.util.Hashtable`. |
| <generic> | Classe `java.lang.Object`. |
| <class> | Classe `java.lang.Class`. |
| <method> | Classe `java.lang.Method`. |
| <list> | Interface `java.util.List` implementada por `java.util.ArrayList`. |
| <set> | Interface `java.util.Set` implementada por `java.util.HashSet`. |
| <list-set> | Classe `net.compor.commons.ListSet`. |
| <exception> | Classe `java.lang.Throwable`. |

Tabela B.1: Estereótipos referentes a tipos e estruturas de dados em Java.

**Reflexão Computacional**

A API de reflexão de Java possibilita a um programa Java inspecionar e manipular a si mesmo [233]. Através dela, pode-se obter informações sobre uma classe e seus membros, assim como criar instâncias de uma classe através do seu nome, além de referenciar e invocar métodos através de informações de sua assinatura (nome e parâmetros).

Esta API contém a classe `java.lang.Class` e demais classes do pacote `java.lang.reflect`, sendo parte da linguagem desde a versão 1.1. Dentre as classes do pacote `java.lang.reflect`, destaca-se a classe `Method`, a qual será usada juntamente com a classe `Class` para implementar os mecanismos de interação baseada em serviços e eventos especificados na CMS e projetados no GCF.

De uma forma geral, para implementar tais mecanismos, apenas duas funcionalidades são requeridas de uma API de reflexão: a possibilidade de armazenar em um atributo uma referência para um método de uma classe; e a possibilidade de recuperar esta referência e invocar o método posteriormente.

A primeira funcionalidade é exemplificada na Listagem de Código B.1. A sintaxe da linguagem Java utilizada para descrever o exemplo não será explicada. Mais informações sobre a sintaxe da linguagem podem ser encontradas em vários livros [234, 235, 236, 56] ou na Internet [237, 238].

A classe de exemplo `ReflectionInJava` possui um atributo privado declarado na linha 5 denominado *method* do tipo `java.lang.reflect.Method`, cujo valor pode ser recuperado por outras classes através do método `getMethod()` (linhas 12 a 14).

No construtor desta classe, na linha 8, a classe `Class` é utilizada para recuperar a referência à classe do objeto atual, que é armazenada na variável *thisClass*. Esta variável é utilizada para acessar as informações da classe, como a referência ao método `test(String)`, através do *nome* e do *tipo* de parâmetro. Na linha 9, tem-se a atribuição da referência do método `test(String)` ao atributo *method* da classe.

A segunda funcionalidade se refere ao acesso à referência do método armazenado no atributo da classe e posterior invocação do mesmo. Isto pode ser feito através do método `invoke()` da classe `Method`, como descrito na Listagem de Código B.2.

Listagem de Código B.1: Exemplo de instância de `Method` armazenada como atributo de uma classe.

```
1  import java.lang.reflect.Method;
2
3  public class ReflectionInJava {
4
5    private Method method; //Atributo de referência a um método
6
7    public ReflectionInJava() throws SecurityException, NoSuchMethodException{
8      Class thisClass = super.getClass(); //Referência à classe
9      this.method = thisClass.getDeclaredMethod("test", new Class[]{String.class}); //Atribuição
10   }
11
12   public Method getMethod(){
13     return this.method;
14   }
15
16   public void test(String param){
17     System.out.println("Reflexão computacional em Java. " + param);
18   }
19 }
```

A referência ao método é recuperada na linha 6 e a invocação do método é realizada na linha 7. O resultado da invocação realizada na linha 7 é o mesmo de uma chamada explícita ao método `test(String)` da classe `ReflectionInJava` (linha 8): “Reflexão computacional em Java. Funciona!”.

Listagem de Código B.2: Exemplo de invocação de método via reflexão.

```
1  import java.lang.reflect.Method;
2
3  public class MainClass {
4      public static void main(String[] args) throws Throwable{
5          ReflectionInJava ref = new ReflectionInJava();
6          Method method = ref.getMethod();
7          method.invoke(ref, new Object[]{"Funciona!"});
8          ref.test("Funciona!");
9      }
10 }
```

Através das funcionalidades exemplificadas nas listagens de código B.1 e B.2, torna-se possível implementar o suporte necessário à definição de serviços providos e eventos de interesse,

os quais necessitam armazenar uma referência a um método para posterior invocação pelo JCF, como ilustrado na Figura B.2. Nesta figura, ilustra-se a operação de invocação de serviços implementada pela classe `FunctionalComponent`, que é a superclasse de todos os componentes funcionais. A implementação do método `receiveRequest(...)` da superclasse de `K` recupera o método definido pelo usuário como implementador do serviço e invoca o método utilizando o mecanismo de reflexão de Java. O mesmo funcionamento ocorre para a notificação de eventos de interesse.

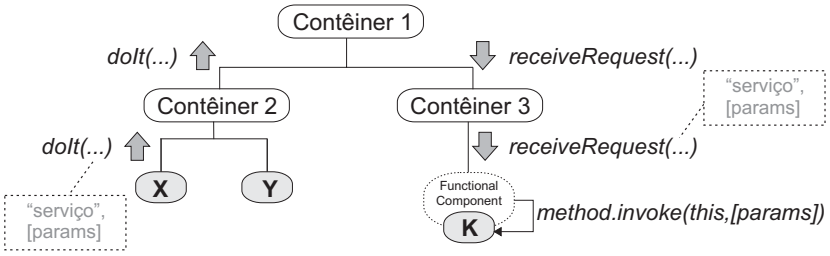



Figura B.2: Funcionamento do mecanismo de reflexão no JCF.

### Execução Assíncrona

A implementação de execução assíncrona em Java também foi realizada utilizando a API padrão da linguagem, mais especificamente a API de *Threads*, cuja principal classe é `java.lang.Thread`. Esta classe permite que um processamento seja encapsulado em um método de um objeto e executado em outra linha de execução, notificando o objeto criador da nova *thread* quando o processamento estiver finalizado (Figura B.3). Este tipo de invocação assíncrona é a motivação para o padrão de projeto Future/Active Object [182], que provê uma solução elegante para esta funcionalidade e foi utilizado no JCF.

Na Figura B.4 ilustra-se o diagrama de classes do JCF relacionadas aos mecanismos de invocação assíncrona de serviços e eventos. A classe abstrata `AsyncMethodInvocation` estende a classe `Thread` da API de Java e implementa uma invocação assíncrona de método. O método `run()` sobrescreve o método da superclasse como descrito na Listagem de Código B.3. O método `invoke()` inicia a execução da `thread` que executará o método `run()`.

Na linha 2, a variável *result*, referente à resposta da invocação assíncrona, é criada como uma instância de `AsyncResult`, que apenas encapsula a resposta ou possível exceção ocorrida

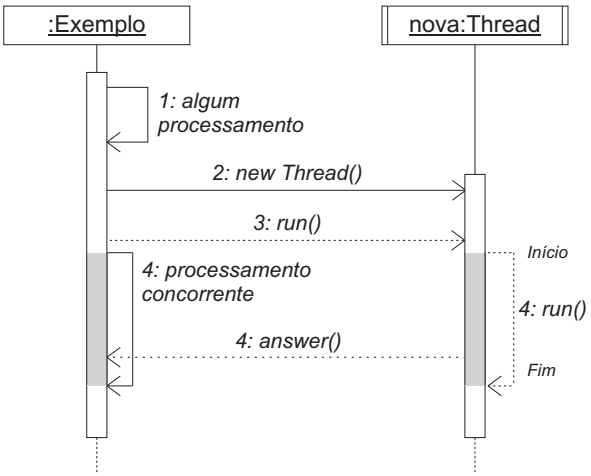



Figura B.3: Diagrama de seqüência de eventos com criação de nova *thread*.

na execução. Na linha 4, o método abstrato `invokeAsyncMethod()` é invocado e seu resultado é armazenado na variável *result*. Este método deve ser sobrescrito pelas subclasses de `AsyncMethodInvocation` contendo o código a ser executado assincronamente. Caso haja alguma exceção na execução do método, a exceção é armazenada na variável *result* (linha 6). Por fim, o método abstrato `receiveAsyncMethodResult(...)` é invocado para retornar a resposta assíncrona (linha 8). Este método deve ser sobrescrito pelas subclasses de `AsyncMethodInvocation` para tratar o recebimento da resposta assíncrona. Depois deste processo, a linha de execução é finalizada.

Listagem de Código B.3: Implementação dos métodos `run()` e `invoke()` de `AsyncMethodInvocation`.

```
1  public void run() {
2    AsyncResult result = new AsyncResult();
3    try {
4      result.setResult(invokeAsyncMethod());
5    } catch (Exception e) {
6      result.setException(e);
7    }
8    receiveAsyncMethodResult(result);
9  }
10
11 public void invoke(){
12   super.start();
13 }
```

As classes concretas `AsyncService` e `AsyncEvent` estendem `AsyncMethodInvocation` com

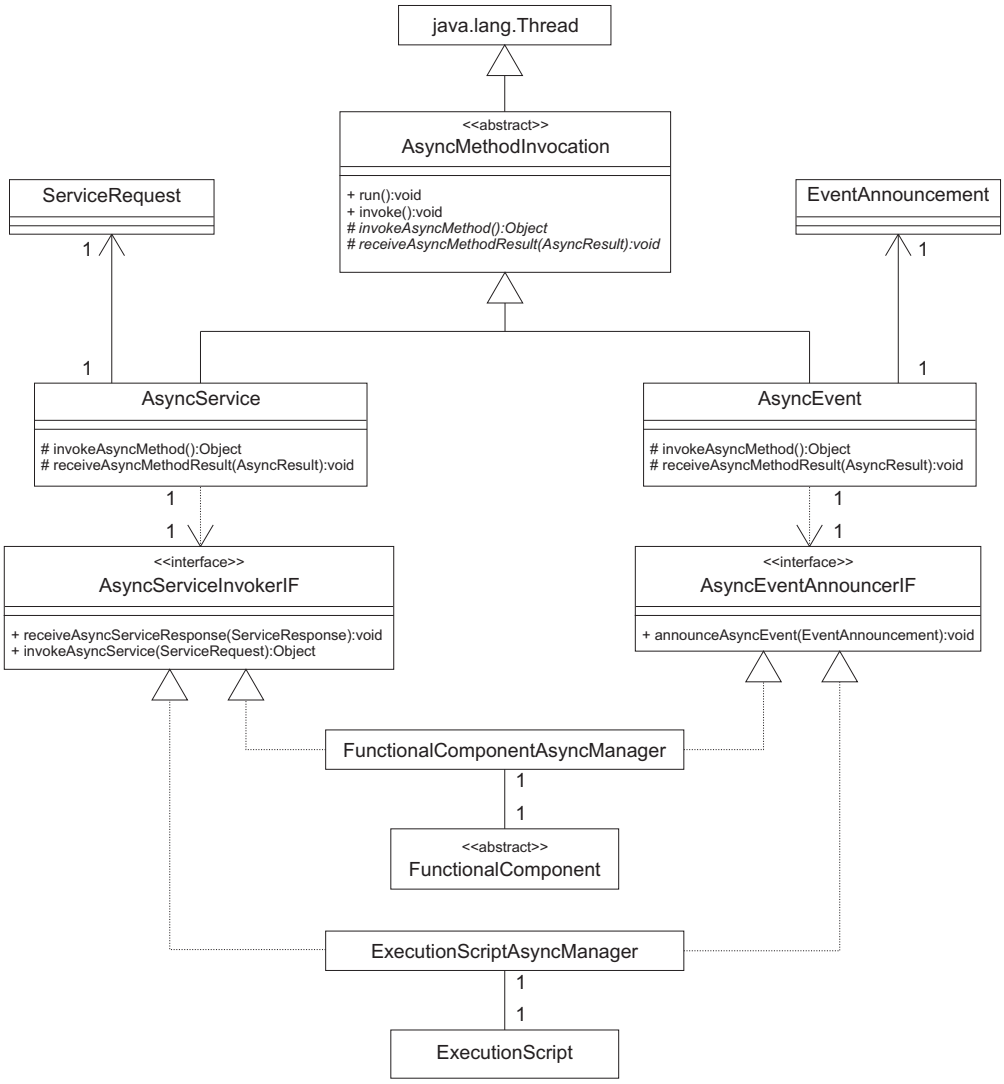



Figura B.4: Diagrama de classes relacionadas à invocação assíncrona no JCF.

implementações dos métodos especificamente voltadas para o tratamento de serviços e eventos. Estas classes lidam com instâncias de `ServiceRequest` e `EventAnnouncement`, respectivamente. Além disso, elas tratam o resultado de forma personalizada: para serviços, o resultado já é retornado como instância de `ServiceRequest`; e no caso de eventos, o resultado não é retornado.

Classes que necessitem implementar invocações assíncronas de serviços e eventos devem implementar as interfaces `AsyncServiceInvokerIF` e `AsyncEventInvokerIF`, as quais são referenciadas por `AsyncService` e `AsyncEvent`. No caso do JCF, foram criadas classes de gerenciamento de invocações assíncronas para componentes funcionais (`FunctionalComponentAsyncManager`) e *scripts* de execução (`ExecutionScriptAsyncManager`). Desta forma, as classes

FunctionalComponent e ExecutionScript fazem uso destas classes para invocar serviços e anunciar eventos assincronamente.

### B.1.2 Como Utilizar o Arcabouço JCF?

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

#### Criando componentes

*1. Estendendo a classe `FunctionalComponent`*

O primeiro passo para a criação de componentes é criar uma subclasse concreta da classe abstrata FunctionalComponent, como descrito na Listagem de Código B.4.

Listagem de Código B.4: Exemplo de extensão da classe FunctionalComponent.

```
1 public class MyComponent extends FunctionalComponent{
2   public MyComponent(){
3     super("My Component");
4   }
5 }
```

O construtor da superclasse possui um parâmetro do tipo *String* referente ao nome do componente. No exemplo em questão, cria-se um construtor simples para a classe MyComponent, sem parâmetros, mas não há restrições no arcabouço quanto à parametrização do construtor.

*2. Implementando o componente, acessando serviços e anunciando eventos*

Uma vez criada a classe referente ao componente funcional (MyComponent), pode-se implementar os métodos que serão declarados posteriormente como implementadores de serviços providos e eventos de interesse. Estes métodos devem ter visibilidade pública para que possam ser acessados via reflexão, como descrito na Listagem de Código B.5 (linhas 7 e 18).

Considere, por exemplo, que o método print(String) será disponibilizado como implementador do serviço “print”. Este serviço recebe um texto como parâmetro, formata-o e o exibe na tela. Porém, esta formatação deve ser realizada por outro componente, cujo serviço é aqui denominado “format”. Nas linhas 9, 10 e 11, descreve-se a sintaxe para a criação da requisição

de serviço, invocação do mesmo através do método `doIt(...)` e exibição da resposta. Caso haja alguma exceção na execução do serviço implementado no componente provedor, esta informação é recuperada através do método `response.getException()`. Se houver um erro na execução do arcabouço, tal como `ServiceNotImplementedException`, a execução será redirecionada para a cláusula `catch` (linha 13). No caso de chamadas assíncronas, utiliza-se o método `doItAsync(...)` que retorna um identificador da requisição e executa a requisição em uma nova linha de execução. O desenvolvedor deve sobrescrever o método `receiveAsyncReponse(...)` para receber a resposta assíncrona.

Listagem de Código B.5: Exemplo de invocação de serviço.

```
1  public class MyComponent extends FunctionalComponent{
2    public MyComponent(){
3      super("My Component");
4    }
5
6    /* Método implementador do serviço "print" */
7    public void print(String text) {
8      try {
9        ServiceRequest request = new ServiceRequest("format", new Object[] {text});
10       ServiceResponse response = super.doIt(request);
11       System.out.println(response.getData());
12     } catch (Exception e){
13       e.printStackTrace();
14     }
15   }
16
17   /* Método implementador do tratamento do evento "saved" */
18   public void receiveSaved(String fileName){
19     System.out.println("Arquivo salvo: " + fileName);
20   }
21 }
```

O anúncio de eventos ocorre de forma semelhante. Considere, por exemplo, que o mesmo método anuncia um evento denominado “printed” após imprimir com sucesso o resultado na tela. Esta funcionalidade é descrita nas linhas 14 e 15 da Listagem de Código B.6.

Neste exemplo o evento não possui parâmetros. Para eventos com parâmetros, basta passá-los como segundo argumento do método `announceEvent(...)`, de forma semelhante ao método `doIt()`. Como mencionado anteriormente, a invocação de eventos é assíncrona. Logo, o fluxo de execução irá continuar independente da linha de execução de notificação do evento.

É importante ressaltar que a execução dos métodos `doIt(...)`, `announceEvent(...)` e `doItAsync(...)` só fazem sentido quando o componente faz parte de uma hierarquia de aplicação baseada na CMS. Caso contrário, a instanciação da classe `MyComponent` seguida da invocação do método `print(String)` causará diversas exceções na execução.

Listagem de Código B.6: Exemplo de anúncio de evento.

```
1  public class MyComponent extends FunctionalComponent{
2    public MyComponent(){
3      super("My Component");
4    }
5
6    /* Método implementador do serviço "print" */
7    public void print(String text) {
8      try {
9        ServiceRequest request = new ServiceRequest("format", new Object[] {text});
10       ServiceResponse response = super.doIt(request);
11       if (!response.hasException()){
12           System.out.println(response.getData());
13           //Anúncio do evento
14           EventAnnouncement announcement = new EventAnnouncement("printed");
15           super.announceEvent(announcement);
16       }else{
17          System.out.println("Problemas na formatação: " + response.getException());
18       }
19     } catch (Exception e){
20       e.printStackTrace();
21     }
22   }
23
24   /* Método implementador do tratamento do evento "saved" */
25   public void receiveSaved(){
26     System.out.println("Salvou!!!");
27   }
28 }
```

*3. Declarando e adicionando serviços e eventos aos componentes*

Uma vez implementada a classe principal do componente, deve-se agora declarar quais dos seus métodos são implementadores de serviços providos e de tratamento de eventos de interesse. Da mesma forma, deve-se declarar quais são os serviços requeridos pelo componente “My Component” e também quais são os eventos anunciados por ele (Listagem de Código B.7).

As declarações de serviços e eventos devem ser feitas no próprio construtor do componente.

Isto é necessário para garantir que, uma vez que o componente esteja instanciado, todos os seus serviços e eventos já estejam devidamente declarados.

Listagem de Código B.7: Declaração de serviços e eventos.

```
1  //Declarando e adicionando o serviço provido "print"
2  List params1 = new ArrayList(); params1.add(new ComporType("name", String.class));
3  List reqs = new ArrayList(); reqs.add("format");
4  ServiceSpecification spec1 = new ServiceSpecification("print", "Imprime nome formatado na tela.",
5                                                        params1, null, null);
6  Method method1 = this.getClass().getDeclaredMethod("print",new Class[]{String.class});
7  super.addProvidedService(new ProvidedService(spec1, method1, reqs));
8
9  //Declarando e adicionando o evento de interesse "saved"
10 List params2 = new ArrayList(); params2.add(new ComporType("fileName", String.class));
11 EventSpecification spec2 = new EventSpecification("saved","Arquivo salvo.",params2);
12 Method method2 = this.getClass().getDeclaredMethod("receiveSaved", new Class[]{String.class});
13 super.addEventOfInterest(new EventOfInterest(spec2, method2));
14
15 //Declarando e adicionando o serviço requerido "format"
16 List params3 = new ArrayList(); params3.add(new ComporType("name", String.class));
17 ServiceSpecification spec3 = new ServiceSpecification("format","Formata o nome.",
18                                      params3, null, new ComporType("formatado", String.class));
19 super.addRequiredService(new RequiredService(spec3));
20
21 //Declarando e adicionando o evento anunciado "printed"
22 EventSpecification spec4 = new EventSpecification("printed", "Nome impresso na tela.", null);
23 super.addAnnouncedEvent(new AnnouncedEvent(spec4));
```

O primeiro bloco da Listagem de Código B.7 implementa a declaração e a adição do serviço provido “print”, cujo método implementador é definido como sendo o método `print(String)` da classe `MyComponent`. Na linha 2, cria-se uma lista contendo o tipo do parâmetro do serviço, neste caso, `String`. A lista de serviços requeridos, no caso apenas “format”, é criada na linha 3. Na linha 4, cria-se uma especificação do serviço, definindo o *nome*, *descrição*, *parâmetros*, *exceções* e *retorno*, tendo os dois últimos valores nulos, pois o método não tem exceções na assinatura e seu retorno é `void`. Na linha 6, cria-se uma instância de referência ao método `print(String)` usando reflexão para, por fim, na linha 7, criar uma instância de `ProvidedService` e adicioná-la ao componente. A partir de então, o componente “My Component” passa a prover o serviço denominado “print”.

A definição do método `receiveSaved()` como implementador do tratamento do evento “sa-

ved" ocorre de forma similar à definição de serviço provido. A única diferença é o formato da especificação de eventos que não possui os parâmetros referentes às exceções e o retorno, como ilustrado nas linhas 10 a 13. Uma vez declarado o evento de interesse, as próximas notificações do evento "saved" serão recebidas pelo componente através do método `receiveSaved(...)`. É importante observar que, como neste exemplo, o nome do evento ou do serviço pode ser diferente do nome do método.

Nas linhas 16 a 19 descreve-se o código que implementa a declaração do serviço requerido "format". Os serviços requeridos a serem declarados são aqueles invocados pelo método `doIt(...)` ou `doItAsync(...)`. O mesmo ocorre para a declaração de eventos anunciados pelos componentes. Neste exemplo, apenas o evento "printed" foi anunciado, através da invocação do método `announceEvent(...)`, e por isso deve ser declarado como descrito nas linhas 22 e 23.

*4. Implementando a inicialização e a interrupção dos componentes*

Códigos de inicialização e interrupção dos componentes são implementados sobrescrevendo os métodos `startImpl()` e `stopImpl()` (Listagem de Código B.8, linhas 7 e 8). As propriedades necessárias à inicialização do componente devem ser declaradas no construtor da classe do componente. Na linha 5, ilustra-se a criação de uma propriedade de inicialização e sua posterior utilização é descrita na linha 12. De acordo com a aplicação, o valor desta propriedade poderá ser redefinido. Caso contrário, será considerado o seu valor padrão, também descrito na linha 5.

Listagem de Código B.8: Inicialização do componente.

```
1  public class MyComponent extends FunctionalComponent{
2    public MyComponent(){
3      super("My Component");
4      ...
5      super.putInitializationPropertyValue("saved_message", "Salvou!");
6    }
7    public void startImpl() {System.out.println("Iniciando componente...");}
8    public void stopImpl() {System.out.println("Finalizando componente...");}
9
10   /* Método implementador do tratamento do evento "saved" */
11   public void receiveSaved(){
12     System.out.println(super.getInitializationPropertyValue("saved_message"));
13   }
14 }
```

**Criando componentes personalizáveis e interesses**

*1. Criando interesses*

O primeiro passo para a criação de interesses é criar uma subclasse concreta da classe `Concern`. Na Listagem de Código B.9, descreve-se a criação da classe `LogConcern`.

Listagem de Código B.9: Exemplo de criação de interesse.

```
1  public class LogConcern extends Concern {
2    public LogConcern() {
3      super("log");
4    }
5    //Interceptando requisições de serviço antes da execução
6    public void interceptBefore(CustomizableComponent component, ServiceRequest request) {
7      System.out.println("Log – Antes:" + request.getServiceName());
8    }
9    //Interceptando requisições de serviço depois da execução
10   public void interceptAfter(CustomizableComponent component,
11                              ServiceRequest request, ServiceResponse response) {
12     System.out.println("Log – Depois:" + request.getServiceName());
13   }
14 }
```

A classe `LogConcern` estende `Concern` para adicionar o comportamento de registro de informação nas requisições de serviço, antes (linha 6) e depois (linha 10) do serviço a ser executado. Ainda é possível interceptar a execução quando ocorre exceção sobrescrevendo o método `interceptOnException(...)`, o que não foi realizado neste exemplo para ressaltar que só é necessário sobrescrever os comportamentos que se deseja interceptar.

*2. Criando componente personalizáveis*

Após a criação dos interesses, pode-se criar componentes personalizáveis. A criação de um componente personalizável ocorre através da extensão da classe `CustomizableComponent`. A implementação deste tipo de componente ocorre da mesma forma que um componente funcional comum, com publicação de serviços, eventos e propriedades de inicialização. A única diferença é que um componente personalizável torna possível a inserção de interesses que podem ser ativados e desativados posteriormente pelo desenvolvedor da aplicação.

Na Listagem de Código B.10, descreve-se a criação de um componente personalizável. Na linha 6, adiciona-se o interesse de registro criado anteriormente (`LogConcern`), inicializando-o como um interesse desativado. A partir de então, este interesse pode ser ativado pelo desenvolve-

dor da aplicação quando este desejar executar a funcionalidade de registro de informações.

Listagem de Código B.10: Exemplo de criação de componente personalizável.

```
1  public class NameFormatter extends CustomizableComponent {
2    public NameFormatter(){
3      super("Name Formatter");
4      ...
5      //Adicionando interesse ao componente...
6      super.addConcern(new LogConcern(), false);
7      ...
8    }
9    ...
10 }
```

### Criando adaptadores

*1. Criando eventos e serviços adaptados*

A criação de eventos e serviços adaptados ocorre através da extensão das classes `AdaptedEvent` e `AdaptedService`. Nas listagens de código B.11 e B.12, descreve-se a implementação das classes `MyAdaptedService` e `MyAdaptedEvent`, que implementa a adaptação do serviço "print" e do tratamento do evento "saved" do componente "My Component".

Listagem de Código B.11: Exemplo de criação de serviço adaptado.

```
1  public class MyAdaptedService extends AdaptedService{
2    public MyAdaptedService(ServiceSpecification specification) {
3      super(specification);
4    }
5
6    //Implementação da adaptação do serviço...
7    public ServiceResponse invokeService(ServiceRequest serviceRequest)
8                    throws InvalidParametersException, IllegalAccessMethodException,
9                    ComponentNotStartedException {
10     System.out.println("Adaptação");
11     return getAdapter().invokeOriginalService(serviceRequest);//Acesso ao serviço original
12   }
13 }
```

*2. Definindo adaptadores para componentes*

Uma vez definidos os serviços e eventos adaptados, basta criar uma instância da classe `Adapter`, adicionar os serviços e eventos adaptados a esta instância e definir o componente funcional sendo

adaptado. Na Listagem de Código B.13, descreve-se a implementação do componente “My Component”. A especificação do serviço e evento adaptado deve ser a mesma daqueles originais. Por isso, nas linhas 2 e 3, faz-se referência às especificações definidas na Listagem de Código B.7. A partir de então, as requisições ao serviço “print” e as notificações do evento “saved” serão redirecionadas aos respectivos adaptadores.

Listagem de Código B.12: Exemplo de criação de evento adaptado.

```
1  public class MyAdaptedEvent extends AdaptedEvent {
2    public MyAdaptedEvent(EventSpecification specification) {
3      super(specification);
4    }
5    //Implementação da adaptação do evento...
6    public void invokeEvent(EventAnnouncement event) {
7      System.out.println("Adaptação");
8      getAdapter().invokeOriginalEvent(event);//Acesso ao evento original
9    }
10 }
```

Listagem de Código B.13: Exemplo de criação de adaptador.

```
1 Adapter adapter = new Adapter("Adaptador de My Component");
2 adapter.addAdaptedService(new MyAdaptedService(spec1));
3 adapter.addAdaptedEvent(new MyAdaptedEvent(spec2));
4 adapter.setFunctionalComponent(myComponent);
```

**Criando aplicações**

*1. Instanciando e configurando os componentes*

O primeiro passo para a construção da aplicação é a instanciação e a configuração dos seus componentes funcionais (Listagem de Código B.14). Na aplicação de exemplo, as classes dos componentes a serem instanciadas são `MyComponent` e `NameFormatter` (linhas 2 e 3). Após a instanciação, deve-se definir, quando necessário, os valores das propriedades de inicialização, apelidos de serviços e eventos e personalizar a execução definindo os interesses ativos.

Para efeito de explicação do exemplo, considere que o componente “Name Formatter” provê o serviço requerido “format” e anuncia o evento “logSaved”, que possui a mesma especificação do evento de interesse do componente “My Component” (“saved”) mas tem o nome diferente. Sendo assim, deve-se redefinir o apelido do evento ou do lado do interessado ou do lado do anunciante.

Na linha 6, altera-se o apelido do evento do lado do anunciante, no caso, o componente “Name Formatter”. A partir de então, o serviço que era anunciado como “logSaved”, passa a ser anunciado como “saved” e assim a notificação chegará ao componente “My Component”. O mesmo processo ocorre na redefinição de apelidos de serviços. De uma forma padrão, todos os apelidos de serviços e eventos são iguais aos seus nomes. A redefinição de apelidos só é utilizada quando os nomes do lado do requisitante e do lado do provedor são diferentes.

Por fim, na linha 9, a execução do componente “Name Formatter” é personalizada através da ativação do interesse de registro (*log*). A partir de então, a funcionalidade de registro implementada neste interesse será executada quando o componente for iniciado.

Listagem de Código B.14: Exemplo de instanciação e configuração de componentes.

```
1 //Instanciando os componentes...
2 NameFormatter nameFormatter = new NameFormatter();
3 MyComponent myComponent = new MyComponent();
4
5 //Redefinindo o apelido do evento no componente anunciante...
6 nameFormatter.setAliasOfAnnouncedEvent("logSaved", "saved")
7
8 //Personalizando a execução do componente...
9 nameFormatter.getConcernSwitch().activate("log");
```

*2. Criando a hierarquia da aplicação*

Na Listagem de Código B.15, descreve-se a construção da hierarquia da aplicação ilustrada na Figura B.1. Nas linhas 1, 2 e 3 são instanciados os contêineres. Observe que o contêiner raiz deve ser instanciado utilizando a classe `ScriptContainer` para que seja possível associá-lo a um *script* de execução posteriormente.

Listagem de Código B.15: Exemplo de construção de hierarquia da aplicação.

```
1 ScriptContainer root = new ScriptContainer("Root");
2 Container printing = new Container("Printing");
3 Container formatting = new Container("Formatting");
4
5 printing.addComponent(myComponent);
6 formatting.addComponent(nameFormatter);
7 root.addComponent(printing);
8 root.addComponent(formatting);
```

Nas linhas 5, 6, 7 e 8, os componentes são adicionados aos contêineres “Printing” e “Formatting” e estes são adicionados ao contêiner “Root”. Depois deste processo, a hierarquia da

aplicação está pronta para ser executada, restando apenas a definição de um *script* de execução para que a aplicação baseada na CMS esteja completa.

**Executando aplicações**

*1. Criando o* script *de execução*

Uma vez criada a hierarquia da aplicação, deve-se então definir o *script* de execução responsável por iniciar a execução da mesma. Quando os componentes funcionais da aplicação implementam todas as funcionalidades, inclusive a interface de interação com o usuário, pode-se criar um *script* de execução apenas instanciando a classe `ExecutionScript`, cujo construtor recebe como parâmetro uma instância de `ScriptContainer`. Para exemplificar a extensão de `ExecutionScript`, na Listagem de Código B.16 descreve-se a implementação da classe `MyScript`.

Listagem de Código B.16: Exemplo de criação de *script* de execução.

```
1  public class MyScript extends ExecutionScript {
2    public MyScript(ScriptContainer rootContainer) {
3      super(rootContainer);
4    }
5    //Código principal de inicialização da aplicação...
6    protected void main() {
7      System.out.println("Nome formatado abaixo...");
8      ServiceRequest req = new ServiceRequest("print", new Object[]{"nome"});
9      try {
10       super.exec(req);
11     } catch (Exception e) {
12       e.printStackTrace();
13     }
14     System.out.println("Fim!!!");
15   }
16 }
```

O método `main` é executado na inicialização do *script* de execução e através dele é possível acessar a hierarquia de componentes utilizando os métodos `exec(...)` e `execAsync(...)`, para invocar serviços, e `announceEvent(...)`, para anunciar eventos. Neste exemplo, o serviço "print" é executado na linha 10. Notificações de eventos também podem ser recebidas sobrescrevendo o método `receiveEvent(...)`.

*2. Iniciando a aplicação*

A inicialização da aplicação é realizada através da instanciação e inicialização do *script* de execução (Listagem de Código B.17). Para instanciar o *script*, deve-se definir o contêiner raiz da aplicação. Depois, basta iniciar a aplicação utilizando o método `init()` que primeiro iniciará o contêiner raiz (e conseqüentemente toda a hierarquia) e depois invocará o método `main`. A finalização do *script* pode ser realizada através do método `finish()`, que requisita a interrupção da execução da hierarquia de componentes.

Listagem de Código B.17: Exemplo de instanciação de *script* de execução.

```
1 MyScript myScript = new MyScript(root);
2 myScript.init();
```

## B.2 *Python Component Framework* (PYCF)

### B.2.1 Implementação do PYCF

Como toda linguagem de programação, Python possui diversas particularidades que tornam a implementação do arcabouço GCF diferente de outras linguagens. Sendo assim, a seguir apresenta-se como funcionam os mecanismos de reflexão computacional e de execução assíncrona, os quais são providos pela biblioteca padrão de Python. Da mesma forma, os estereótipos de tipos apresentados no Apêndice A possuem implementação direta na API padrão de Python, como descrito na Tabela B.2.

#### Reflexão Computacional

Python possui mecanismos de reflexão computacional embutidos na própria linguagem, não sendo necessária nenhuma biblioteca específica para isso. Esses mecanismos permitem que sejam extraídas informações a respeito de classes e seus membros, importar novos módulos dinamicamente, criar instâncias de classes dinamicamente através de seus nomes, ou guardar referências de métodos de classe para uma execução posterior.

Como mencionado anteriormente, para implementar os mecanismos de interação baseada em serviços e eventos projetados no GCF, são necessárias apenas duas funcionalidades de reflexão computacional: a possibilidade de armazenar em um atributo uma referência para um método de

| Estereótipo | Descrição |
|---|---|
| <string> | Classe `str` |
| <collection> | Classe `list` |
| <table> | Classe `dict` |
| <generic> | Classe `object` |
| <class> | Classe `instance` |
| <method> | Classe `instancemethod` |
| <list> | Classe `list` |
| <set> | Classe `list` |
| <list-set> | Classe `set` |
| <exception> | Classe `Exception` |

Tabela B.2: Estereótipos referentes a tipos e estruturas de dados em Python.

uma classe e a possibilidade de recuperar esta referência e invocar o método posteriormente. A primeira funcionalidade implementada em Python é descrita na Listagem de Código B.18.

Na classe `ReflectionInPython`, o método `getMethod()` retorna uma referência para o método `test()`. Essa referência é obtida através da função embutida `getattr(...)`, utilizada no construtor da classe (`__init__()`). O primeiro parâmetro dessa função se refere à classe que contém o método, que nesse caso é a própria classe (`self`), e o segundo parâmetro é uma `<string>` contendo o nome do método que será referenciado (`"test"`).

Listagem de Código B.18: Obtendo uma referência para um método de uma classe.

```
1
2  class ReflectionInPython (object):
3
4      def __init__(self):
5          pass
6
7      def getMethod(self):
8          return getattr(self, "test")
9
10     def test(self, paramStr):
11         print "Executando " + paramStr
```

Na Listagem de Código B.19, descreve-se a implementação em Python da segunda funciona-

lidade – o acesso a uma referência de método e posterior execução do mesmo. Para a execução do método não é necessário utilizar nenhuma função ou módulo especial, basta apenas chamar a referência ao método como se fosse um método comum.

No construtor da classe `TestReflectionInPython` descrita na Listagem de Código B.19, cria-se uma instância da classe `ReflectionInPython` na linha 4 e, logo após, obtém-se uma referência para o método `test()` na linha 6. Na linha 7 o método `test()` é executado através da referência obtida por reflexão, retornando o resultado "Executando via reflexão!". Já na linha 8, o método `test()` é executado explicitamente, retornando "Executando normalmente!".

Listagem de Código B.19: Recuperando referência para um método e executando-o via reflexão.

```
1 class TestReflectionInPython (object):
2
3     def __init__(self):
4         self.reflectionInPython = ReflectionInPython()
5
6         method = self.reflectionInPython.getMethod("test")
7         method("Executando via reflexão!")
8         self.reflectionInPython.test("Executando normalmente!")
```

As funcionalidades demonstradas nas listagens B.18 e B.19 fornecem o suporte necessário à implementação da definição de serviços providos e eventos de interesse, pois as referências para os mesmos precisam ser armazenadas para posterior execução no PyCF.

### Execução Assíncrona

A execução assíncrona no PyCF é implementada de maneira similar à execução assíncrona do JCF, pois a linguagem Python fornece um módulo padrão para manipulação de *threads* assim como na linguagem Java. Na Listagem de Código B.20 descreve-se a implementação em Python do padrão *ActiveObject* [239], onde a execução do método é feita assincronamente em relação ao objeto que fez a chamada original. Para maiores informações sobre o projeto de classes, veja a Figura B.4.

Na linha 9 da Listagem de Código B.20 descreve-se o método `run()` da superclasse `Thread` sendo sobrescrito. Na linha 10 é criada a instância `result` da classe `AsyncResult`, a qual encapsula o resultado da operação assíncrona ou uma possível exceção. Na linha 12 o método abstrato `invokeAsyncMethod()` é executado e o resultado é armazenado no objeto `result`. O método

`invokeAsyncMethod()` deve ser sobrescrito nas subclasses da classe `AsyncMethodInvocation`. Finalmente, na linha 16, o método `receiveAsyncMethodResult(...)` é executado para retornar o resultado da execução. O método `receiveAsyncMethodResult(...)` também deve ser sobrescrito nas subclasses de `ComporAsyncMethodInvocation`.

Listagem de Código B.20: Implementação dos métodos `run()` e `invoke()` de `AsyncMethodInvocation`.

```
1  from threading import Thread
2  from AsyncResult import AsyncResult
3
4  class AsyncMethodInvocation(Thread):
5      def __init__(self):
6          Thread.__init__(self)
7          self.result = None
8
9      def run(self):
10         self.result = AsyncResult()
11         try:
12             self.result.setResult(self.invokeAsynchronousMethod())
13         except Exception, e:
14             self.result.exception = e
15             raise e
16         self.receiveAsynchronousMethodResult(self.result)
17
18     def invokeAsyncMethod(self):
19         raise NotImplementedError("Some implementation is needed!")
20
21     def receiveAsyncMethodResult(self, result):
22         raise NotImplementedError("Some implementation is needed!")
23
24     def invoke(self):
25         self.start()
```

## B.2.2 Como Utilizar o Arcabouço PyCF?

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

**Criando componentes**

*1. Estendendo a classe `FunctionalComponent`*

Para criar um componente no PYCF é necessário que o programador construa uma classe em Python e a mesma herde da classe `FunctionalComponent` fornecida pelo arcabouço, passando como parâmetro para o construtor da classe pai uma `<string>` com o nome do componente, como descrito na Listagem de Código B.21.

Listagem de Código B.21: Para criar um componente estende-se a classe *FunctionalComponent*.

```
1  class MyComponent(FunctionalComponent):
2
3      def __init__(self, componentName="MyComponent"):
4          FunctionalComponent.__init__(self, componentName)
```

*2. Implementando o componente, acessando serviços e anunciando eventos*

Para qualquer componente executar serviços fornecidos por outros componentes, primeiro deve-se declarar explicitamente no construtor do componente que o mesmo necessita de tais serviços. Na Listagem de Código B.22, descreve-se a declaração dos serviços que um componente chamado `MyComponent` requer, indicando o método que faz uso dos mesmos.

Listagem de Código B.22: Exemplo de invocação de serviço.

```
1  class MyComponent(FunctionalComponent):
2
3      def __init__(self, componentName="MyComponent"):
4          FunctionalComponent.__init__(self, componentName)
5          serviceSpecification = ServiceSpecification("format")
6          requiredService = RequiredService(serviceSpecification)
7          self.addRequiredService(requiredService)
8
9      # Método implementador do serviço "print"
10     def print(self,text):
11         serviceRequest = ServiceRequest("format", (text))
12         resultado = self.doIt(serviceRequest)
13         if not resultado.hasException():
14             print resultado.data
15         else:
16             raise resultado.exception
17
18     # Método implementador do tratamento do evento "saved"
19     def receiveSaved(self, fileName):
20         print 'Arquivo salvo:' + fileName
```

Nas linhas 5, 6 e 7 da Listagem de Código B.22, cria-se uma especificação de serviço e faz-se uso da mesma para criar um objeto do tipo `RequiredService`, o qual será adicionado na tabela de serviços requeridos do componente. Uma vez que os serviços requeridos já foram informados no construtor da classe, o método `requesitaServico()` pode fazer uma solicitação de serviço requerido, como é descrito nas linhas 11 e 12. O método `doIt(...)` na linha 12 envia a solicitação para a hierarquia de componentes do PyCF e o mesmo retorna uma resposta armazenada na variável `resultado`. Para invocações assíncronas, deve-se utilizar o método `doItAsync(...)` e o desenvolvedor deverá implementar o método `receiveAsyncResponse(...)` para receber a resposta da execução assíncrona.

O anúncio de eventos ocorre de maneira similar à solicitação de serviços. Na Listagem de Código B.6, tem-se uma implementação alternativa do método implementador do serviço "print" do componente `MyComponent`. Quando esse serviço é executado, caso haja sucesso na formatação e o resultado seja impresso, anuncia-se o evento "printed" (ver linhas 10 e 11).

Listagem de Código B.23: Exemplo de anúncio de evento.

```
1  class MyComponent(FunctionalComponent):
2
3      # Método implementador do serviço "print"
4      def print(self,text):
5          serviceRequest = ServiceRequest("format", (text))
6          resultado = self.doIt(serviceRequest)
7          if not resultado.hasException():
8              print resultado.data
9              # Anúncio do evento
10             eventAnnoucement = EventAnnouncement("printed")
11             self.announceEvent(eventAnnoucement)
12         else:
13             raise resultado.exception
```

Neste exemplo o evento não possui parâmetros. Para eventos com parâmetros, basta passá-los como segundo argumento do método `announceEvent(...)`, de forma semelhante ao método `doIt()`. Como mencionado anteriormente, a invocação de eventos é assíncrona. Logo, o fluxo de execução irá continuar independente da linha de execução de notificação do evento.

É importante notar que os métodos `doIt(...)`, `doItAsync(...)` e `announceEvent(...)` só devem ser chamados em componentes que fazem parte de uma hierarquia de aplicação baseada na CMS, caso contrário o PyCF lançará várias exceções durante a execução.

*3. Declarando e adicionando serviços e eventos aos componentes*

Uma vez implementada a classe principal do componente, deve-se agora declarar quais dos seus métodos são implementadores de serviços providos e de tratamento de eventos de interesse. Da mesma forma, deve-se declarar quais são os serviços requeridos pelo componente “My Component” e também quais são os eventos anunciados por ele (Listagem de Código B.24).

As declarações de serviços e eventos devem ser feitas no próprio construtor do componente. Isto é necessário para garantir que, uma vez que o componente esteja instanciado, todos os seus serviços e eventos já estejam devidamente declarados.

Listagem de Código B.24: Declaração de serviços e eventos.

```
1  # Declarando e adicionando o serviço provido "print"
2  parameters1 = [ComporType(str, "texto")]
3  serviceSpecification1 = ServiceSpecification("print", "Imprime texto formatado.",
4                                               parameters1, None, None)
5  method1 = getattr(self, "print")
6  providedService = ProvidedService(serviceSpecification1,method1,requiredServicesAlias=('format'))
7  self.addProvidedService(providedService)
8
9  # Declarando e adicionando o evento de interesse "saved"
10 parameters2 = [ComporType(str, "nome do arquivo")]
11 eventSpecification1 = EventSpecification("saved", "Arquivo salvo.", parameters2)
12 method2 = getattr(self, "receiveSaved")
13 eventOfInterest = EventOfInterest(eventSpecification1, method2)
14 self.addEventOfInterest(eventOfInterest)
15
16 # Declarando e adicionando o serviço requerido "format"
17 serviceSpecification2 = ServiceSpecification("format")
18 self.addRequiredService(RequiredService(serviceSpecification2))
19
20 # Declarando e adicionando o evento anunciado "printed"
21 eventSpecification2 = EventAnnouncement("printed")
22 self.addAnnouncedEvent(AnnouncedEvent(eventSpecification2))
```

O primeiro bloco da Listagem de Código B.24 implementa a declaração e a adição do serviço provido “print”, cujo método implementador é definido como sendo o método `print(...)` da classe `MyComponent`. Na linha 2, cria-se uma lista contendo o tipo do parâmetro do serviço, neste caso, `str`. Na linha 3, cria-se uma especificação do serviço, definindo o *nome*, *descrição*, *parâmetros*, *exceções* e *retorno*, tendo os dois últimos valores nulos, pois o método não tem exceções na assinatura nem retorno. Na linha 5, cria-se uma instância de referência ao método

`print()` usando reflexão para, por fim, nas linhas 6 e 7, criar uma instância de `ProvidedService` e adicioná-la ao componente, indicando o serviço "format" como requerido. A partir de então, o componente "My Component" passa a prover o serviço denominado "print".

A definição do método `receiveSaved()` como implementador do tratamento do evento "saved" ocorre de forma similar à definição de serviço provido. A única diferença é o formato da especificação de eventos que não possui os parâmetros referentes às exceções e o retorno, como ilustrado na linha 11. Uma vez declarado o evento de interesse, as próximas notificações do evento "saved" serão recebidas pelo componente através do método `receiveSaved(...)`. É importante observar que, como neste exemplo, o nome do evento ou do serviço pode ser diferente do nome do método.

Nas linhas 17 e 18 descreve-se o código que implementa a declaração do serviço requerido "format". Os serviços requeridos a serem declarados são aqueles invocados pelo método `doIt(...)` ou `doItAsync(...)`. O mesmo ocorre para a declaração de eventos anunciados pelos componentes. Neste exemplo, apenas o evento "printed" foi anunciado, através da invocação do método `announceEvent(...)`, e por isso deve ser declarado como descrito nas linhas 21 e 22.

*4. Implementando a inicialização e a interrupção dos componentes*

Códigos de inicialização e interrupção dos componentes são implementados sobrescrevendo os métodos `startImpl()` e `stopImpl()`. Exemplos de implementação destes métodos de inicialização são descritos nas linhas 8 a 11 da Listagem de Código B.25.

Listagem de Código B.25: Inicialização do componente.

```
1 class MyComponent(FunctionalComponent):
2     def __init__(self, componentName="MyComponent"):
3         FunctionalComponent.__init__(self, componentName)
4         ...
5         self.putInitializationPropertyValue("saved_message", "Salvou!")
6     def startImpl(self):
7         print "Inicializando..."
8     def stopImpl(self):
9         print "Finalizando..."+ self.getInitializationPropertyValue("saved_message")
```

As propriedades necessárias à inicialização do componente devem ser declaradas no construtor da classe do componente. Na linha 5 da Listagem de Código B.25, ilustra-se a criação de uma propriedade de inicialização e sua posterior utilização é descrita na linha 9. De acordo com a

aplicação, o valor desta propriedade poderá ser redefinido. Caso contrário, será considerado o seu valor padrão, também descrito na linha 5.

**Criando componentes personalizáveis e interesses**

*1. Criando interesses*

A criação de interesses torna possível interceptar os serviços de um componente em diferentes estágios da execução. No PyCF os serviços podem ser interceptados antes da execução, depois da execução ou quando alguma exceção for gerada.

Na Listagem de Código B.26 descreve-se um exemplo de criação de interesses. Para criar um interesse é necessário criar uma classe que estenda a superclasse `Concern`, como descrito na linha 1. Depois disso, basta implementar os métodos `interceptBefore(...)` para interceptações antes da execução, `interceptAfter(...)` para interceptações após a execução e `interceptOnException(...)` para interceptações durante a ocorrência de exceções, como descrito na Listagem de Código B.26 nas linhas 5, 8 e 11 respectivamente. É importante ressaltar que o desenvolvedor só precisa implementar os métodos relacionados à parte do código que deseja interceptar.

Listagem de Código B.26: Exemplo de criação de interesse.

```
1  class FilePermission(Concern):
2      def __init__(self, concernName="FilePermission"):
3          Concern.__init__(self, concernName)
4
5      def interceptBefore(self, _component, _request):
6          print "Verificando permissões..."
7
8      def interceptAfter(self, _component, _request, _response):
9          print "Permissão concedida e operação efetuada."
10
11     def interceptOnException(self, _component, _request, _response):
12         print "Usuário não possui permissões suficientes."
```

*2. Criando componentes personalizáveis*

Para fazer uso dos interesses é necessário que os componentes que devem ser interceptados sejam criados estendendo a superclasse `CustomizableComponent`. A implementação desses tipos de componentes se faz da mesma forma dos componentes funcionais comuns.

Na Listagem de Código B.27 descreve-se a criação de um componente personalizado. Na linha 6 é adicionado o interesse `FilePermission` criado na Listagem de Código B.27.

Listagem de Código B.27: Exemplo de criação de componente personalizável.

```
1  class WriteFile(CustomizableComponent):
2      def __init__(self):
3          CustomizableComponent.__init__(self, "WriteFile")
4
5          #Adicionando interesse ao componente...
6          self.addConcern(FilePermission(), False)
```

### Criando adaptadores

*1. Criando eventos e serviços adaptados*

Serviços e eventos podem ser adaptados através das classes `AdaptedService` e `AdaptedEvent` respectivamente. As listagens de código B.28 e B.29 demonstram o uso de adaptadores.

Listagem de Código B.28: Exemplo de criação de serviço adaptado.

```
1  class NewAdaptedService(AdaptedService):
2
3      def __init__(self, serviceSpecification):
4          AdaptedService.__init__(self, serviceSpecification)
5
6      def invokeService(self, serviceRequest):
7          print "Adaptando serviço..."
8
9          #Acesso ao servico original...
10         return self.getAdapter().invokeOriginalService(serviceRequest)
```

Listagem de Código B.29: Exemplo de criação de evento adaptado.

```
1  class NewAdaptedEvent(AdaptedEvent):
2
3      def __init__(self, eventSpecification):
4          AdaptedEvent.__init__(self, eventSpecification)
5
6      def invokeEvent(self, eventAnnouncement):
7          print "Adaptando evento..."
8
9          #Acesso ao evento original...
10         return self.getAdapter().invokeOriginalEvent(eventAnnouncement)
```

*2. Definindo adaptadores para componentes*

Depois de criar os serviços e eventos adaptados é necessário definir os adaptadores para os componentes que implementam esses serviços e eventos inicialmente. Isso é feito criando uma instância da classe `Adapter` e adicionando os componentes a serem adaptados pela mesma. Na Listagem de Código B.30 ilustra-se a criação de um adaptador. As especificações dos serviços e eventos devem ser iguais às originais. Nas linhas 2 e 3 são adicionados o serviço e o evento adaptado criados nas listagens de código B.28 e B.29, respectivamente. Na linha 4 adiciona-se o componente, previamente criado, que faz a implementação original dos mesmos.

Listagem de Código B.30: Exemplo de criação de adaptador.

```
1 adapter = Adapter("Adaptador de componente")
2 adapter.addAdaptedService(NewAdaptedService(serviceSpecification))
3 adapter.addAdaptedSEvent(NewAdaptedEvent(eventSpecification))
4 adapter.setFunciontalComponent(component)
```

**Criando aplicações**

O exemplo de arquitetura ilustrado no início do capítulo, mais especificamente na Figura B.1, é utilizado para descrever a criação de uma aplicação no PyCF.

*1. Instanciando e configurando os componentes*

O primeiro passo para a construção da aplicação é a instanciação e a configuração dos seus componentes funcionais (Listagem de Código B.31). Na aplicação de exemplo, as classes dos componentes a serem instanciadas são `MyComponent` e `NameFormatter` (linhas 2 e 3). Após a instanciação, deve-se definir, quando necessário, os valores das propriedades de inicialização, apelidos de serviços e eventos e personalizar a execução definindo os interesses ativos.

Para efeito de explicação do exemplo, considere que o componente “Name Formatter” provê o serviço requerido “format” e anuncia o evento “logSaved”, que possui a mesma especificação do evento de interesse do componente “My Component” (“saved”) mas tem o nome diferente. Sendo assim, deve-se redefinir o apelido do evento ou do lado do interessado ou do lado do anunciante. Na linha 6, altera-se o apelido do evento do lado do anunciante, no caso, o componente “Name Formatter”. A partir de então, o serviço que era anunciado como “logSaved”, passa a ser anunciado como “saved” e assim a notificação chegará ao componente “My Component”. O mesmo processo ocorre na redefinição de apelidos de serviços. De uma forma padrão, todos os apelidos

de serviços e eventos são iguais aos seus nomes. A redefinição de apelidos só é utilizada quando os nomes do lado do requisitante e do lado do provedor são diferentes.

Por fim, na linha 9, a execução do componente “Name Formatter” é personalizada através da ativação do interesse de registro (*log*). A partir de então, a funcionalidade de registro implementada neste interesse será executada quando o componente for iniciado.

Listagem de Código B.31: Exemplo de instanciação e configuração de componentes.

```
1 # Instanciando os componentes...
2 nameFormatter = NameFormatter()
3 myComponent = MyComponent()
4
5 # Redefinindo o apelido do evento no componente anunciante...
6 nameFormatter.setAliasOfAnnouncedEvent("logSaved", "saved")
7
8 # Personalizando a execução do componente...
9 nameFormatter.getConcernSwitch().activate("log");
```

*2. Criando a hierarquia da aplicação*

Na Listagem de Código B.32, descreve-se a construção da hierarquia da aplicação ilustrada na Figura B.1. Nas linhas 1, 2 e 3 são instanciados os contêineres. Observe que o contêiner raiz deve ser instanciado utilizando a classe `ScriptContainer` para que seja possível associá-lo a um *script* de execução posteriormente.

Nas linhas 5, 6, 7 e 8, os componentes são adicionados aos contêineres “Printing” e “Formatting” e estes são adicionados ao contêiner “Root”. Depois deste processo, a hierarquia da aplicação está pronta para ser executada, restando apenas a definição de um *script* de execução para que a aplicação baseada na CMS esteja completa.

Listagem de Código B.32: Exemplo de construção de hierarquia da aplicação.

```
1 root = ScriptContainer("Root");
2 printing = Container("Printing");
3 formatting = Container("Formatting");
4
5 printing.addComponent(myComponent);
6 formatting.addComponent(nameFormatter);
7 root.addComponent(printing);
8 root.addComponent(formatting);
```

**Executando aplicações**

*1. Criando o* script *de execução*

Uma vez criada a hierarquia da aplicação, deve-se então definir o *script* de execução responsável por iniciar a execução da mesma. Quando os componentes funcionais da aplicação implementam todas as funcionalidades, inclusive a interface de interação com o usuário, pode-se criar um *script* de execução apenas instanciando a classe `ExecutionScript`, cujo construtor recebe como parâmetro uma instância de `ScriptContainer`. Para exemplificar a extensão de `ExecutionScript`, na Listagem de Código B.33 descreve-se a implementação da classe `MyScript`.

Listagem de Código B.33: Exemplo de criação de *script* de execução.

```
1  class MyScript(ExecutionScript):
2      def __init__(self, scriptContainer):
3          ExecutionScript.__init__(self, scriptContainer)
4
5      def main(self):
6          print "Nome formatado abaixo..."
7          request = ServiceRequest("print", ("nome"))
8          try:
9              self._exec(request)
10         except Exception, e:
11             print e
12         print "Fim!!!"
```

O método `main` é executado na inicialização do *script* de execução e através dele é possível acessar a hierarquia de componentes utilizando os métodos `exec(...)` e `execAsync(...)`, para invocar serviços, e `announceEvent(...)`, para anunciar eventos. Neste exemplo, o serviço "print" é executado na linha 9. Notificações de eventos também podem ser recebidas sobrescrevendo o método `receiveEvent(...)`.

*2. Iniciando a aplicação*

A inicialização da aplicação é realizada através da instanciação e inicialização do *script* de execução (Listagem de Código B.34). Para instanciar o *script*, deve-se definir o contêiner raiz da aplicação. Depois, basta iniciar a aplicação utilizando o método `init()` que primeiro iniciará o contêiner raiz (e conseqüentemente toda a hierarquia) e depois invocará o método `main`. A

finalização do *script* pode ser realizada através do método `finish()`, que requisita a interrupção da execução da hierarquia de componentes.

Listagem de Código B.34: Exemplo de instanciação de *script* de execução.

```
1 myScript = MyScript(root)
2 myScript.init()
```

## B.3 *C++ Component Framework* (CCF)

### B.3.1 Implementação do CCF

Algumas funcionalidades da API de C++ como, por exemplo, a vasta quantidade de coleções disponíveis na *Standard Template Library* (STL) e os mecanismos de execução assíncrona facilitaram a implementação do CCF. Porém, para viabilizar a reflexão foi necessária a biblioteca Seal Reflex [181] que não é um padrão da linguagem. Os estereótipos de tipos apresentados no Apêndice A possuem implementação direta na API padrão de C++, como descrito na Tabela B.3.

| **Estereótipo** | **Descrição** |
|---|---|
| <string> | Classe std::string |
| <collection> | Template std::vector da biblioteca STL |
| <table> | Template std::map da biblioteca STL |
| <generic> | Ponteiro para void |
| <class> | — |
| <method> | Classe Member da biblioteca Seal Reflex |
| <list> | Template std::list da biblioteca STL |
| <set> | Template std::set da biblioteca STL |
| <list-set> | — |
| <exception> | Classe std::exception |

Tabela B.3: Estereótipos referentes a tipos e estruturas de dados em C++.

**Reflexão computacional**

A API de reflexão Seal Reflex possibilita a um programa C++ recuperar informações sobre uma classe e seus membros, assim como criar instâncias de uma classe através de seu nome, além de referenciar e invocar métodos através de informações de sua assinatura (nome). Esta API contém oito classes, sendo a classe `Member` a de maior utilidade para os mecanismos de interação baseada em serviços e eventos especificados na CMS e projetados no GCF.

Seal reflex satisfaz as duas funcionalidades requeridas de uma API de reflexão na especificação CMS: a possibilidade de armazenar uma referência para um método de uma classe em um atributo e a possibilidade de recuperar esta referência para invocar o método posteriormente.

A primeira funcionalidade é exemplificada na Listagem de Código B.35. A classe de exemplo `ReflectionInCpp` possui um atributo privado declarado na linha 17 denominado `m_method` do tipo `Member`, cujo valor pode ser recuperado por outras classes através do método `getMethod()` (linha 14).

Listagem de Código B.35: Exemplo de instância de `Member` armazenada como atributo de uma classe.

```
1  #include "Reflex/Reflex.h"
2  #include <iostream.h>
3  #include <string>
4  using namespace ROOT::Reflex; using namespace std;
5  class ReflectionInCpp {
6  public:
7      inline ReflectionInCpp(){
8          Type type = Type::ByName("ReflectionInCpp"); //Referência à classe
9          m_method = type.MemberByName("test");
10     }
11     inline virtual ~ReflectionInCpp() {}
12
13     inline void test(const string & param){cout << "Reflexão em C++. " << param << endl;}
14     inline Member & getMethod(){return m_method;}
15
16 private:
17     Member m_method; //Atributo de referência a um método
18 };
```

No construtor desta classe, descrito nas linhas 7 a 10 da Listagem de Código B.35, a variável `type` é utilizada para armazenar a referência da classe `ReflectionInCpp`. Esta variável é utilizada para acessar as informações da classe através do nome, neste exemplo, do método `test`. Na

linha 9, a referência do método `test` é atribuída ao membro da classe `m_method`.

A segunda funcionalidade se refere à recuperação da referência do método armazenado no atributo da classe e posterior invocação do mesmo. Isto pode ser feito através do método `invoke()` da classe `Member`, como descrito na Listagem de Código B.36. A referência do método é recuperada na linha 8, a invocação é realizada na linha 13. O resultado da invocação realizada na linha 13 é o mesmo de uma chamada explícita do método test da classe Reflection (linha 14): "Reflexão em C++. Funciona!".

Listagem de Código B.36: Exemplo de invocação de método via reflexão.

```
1  #include "Reflex/Reflex.h"
2  #include <string>
3
4  using namespace ROOT::Reflex; using namespace std;
5
6  int main() {
7      ReflectionInCpp * ref = new ReflectionInCpp();
8      Member method = ref->getMethod();
9      vector<void *> parameters;
10     string param = "Funciona!"
11
12     parameters.push_back(&param);
13     method.invoke( Object(Type(), ref), parameters );
14     ref->test("Funciona!");
15
16     delete ref;
17     return 0;
18 }
```

Através das funcionalidades exemplificadas nas listagens de código B.35 e B.36, torna-se possível implementar o suporte necessário à definição de serviços providos e eventos de interesse, os quais necessitam armazenar uma referência de um método para posterior invocação pelo CCF. O funcionamento deste mecanismo no CCF é similar ao implementado no JCF.

### Execução assíncrona

A implementação da execução assíncrona no CCF foi realizada utilizando a biblioteca *Pthreads* que faz parte da linguagem C. Esta biblioteca permite que um processamento seja encapsulado em uma função e executado em outra *thread*. Após a finalização do processamento o objeto criador da nova thread é notificado. Este funcionamento é similar ao que foi implementado no JCF e

PyCF, com a implementação do padrão *ActiveObject* [239]. Para maiores informações sobre o projeto de classes, veja a Figura B.4.

As classes `AsyncService` e `AsyncEvent` implementam os métodos para requisição de serviços (`ServiceRequest`) e o anúncio de eventos (`EventAnnouncement`), respectivamente. Além disso, elas tratam o resultado de forma personalizada, ou seja, na classe `AsyncService` o resultado é retornado como instância de `ServiceResponse` e na classe `AsyncEvent` o resultado não é retornado. Classes que necessitem implementar invocações assíncronas de serviços e de eventos devem estender as classes abstratas `AsyncServiceInvokerIF` e `AsyncEventInvokerIF`, as quais são referenciadas por `ComponentAsyncService` e `ComponentAsyncEvent`. No caso do CCF, foram criadas classes para o gerenciamento de invocações assíncronas para componentes funcionais (classe `FunctionalComponentAsyncManager`) assim como para *scripts* de execução (classe `ExecutionScriptAsyncManager`). As classes `FunctionalComponent` e `ExecutionScript` fazem uso destas classes para invocar serviços e anunciar eventos assincronamente.

Na invocação assíncrona de serviço, descrita na Listagem de Código B.37, a variável `result`, referente à resposta da invocação assíncrona, é criada como uma instância de `AsyncResult`, que apenas encapsula a resposta ou possível exceção ocorrida na execução.

Na linha 9, o método `invokeAsyncMethod` é invocado e seu resultado é armazenado na variável `result`. Caso haja alguma exceção na execução do método, a exceção é também armazenada na variável `result` (linha 11). Antes de finalizar o seu processamento, o método `receiveAsyncMethodResult` é invocado para retornar a resposta assíncrona (linha 13). Por fim, a quantidade de tarefas assíncronas é decrementada (linha 22) e a `thread` é finalizada. Este controle de tarefas assíncronas em execução é a única diferença significativa de implementação entre o CCF e as demais implementações da CMS.

A invocação assíncrona de evento tem uma implementação similar à invocação assíncrona de serviço. De fato, todo anúncio de evento é realizado de forma assíncrona, seguindo a implementação descrita na Listagem de Código B.38. Da mesma forma que ocorre com serviços, inicia-se uma chamada assíncrona, através do método `invokeAsyncMethod` (linha 7). Como o anúncio de um evento não retorna resultado, não é necessária uma instância de `AsyncResult`. Conseqüentemente, não é necessária a chamada de um método para a recepção do resultado do evento. Esta é a única diferença de implementação entre as invocações assíncronas de serviço e evento. Por fim, decrementa-se a quantidade de tarefas assíncronas (linha 16) e finaliza-se a *thread* (linha 17).

Listagem de Código B.37: Implementação dos métodos para invocação assíncrona (serviço).

```
void * AsyncMethodInvocation::asychronousService(void * _param) {
    service_t * service = (service_t *) _param;
    ComponentAsyncService * component = new ComponentAsyncService(*service->asyncManager,
                                                service->serviceRequest);
    AsyncResult result;
    if (component != NULL){
        try{
          result.setResult(component->invokeAsyncMethod());
        }catch (exception &e){
          result.setException(e);
        }
        component->receiveAsynchronousMethodResult(result);
        delete component;
        component = NULL;
    }else{
        delete service->serviceRequest;
        service->serviceRequest = NULL;
    }
    delete service;
    service = NULL;
    Jobs::instance()->decreaseJobs();
    pthread_exit(0);
    return 0;
}
```

Listagem de Código B.38: Implementação dos métodos para invocação assíncrona (evento).

```
void * AsyncMethodInvocation::asychronousEvent(void * _param) {
    announcement_t * announce = (announcement_t *) _param;
    ComponentAsyncEvent * component = new ComponentAsyncEvent(*announce->asyncManager,
                                                announce->eventAnnouncement);
    if (component != NULL){
        component->invokeAsynchronousMethod();
        delete component;
        component = NULL;
    }else{
        delete announce->eventAnnouncement;
        announce->eventAnnouncement = NULL;
    }
    delete announce;
    announce = NULL;
    Jobs::instance()->decreaseJobs();
    pthread_exit(0);
    return 0;
}
```

### B.3.2 Como utilizar o arcabouço CCF?

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

#### Criando componentes

*1. Estendendo a classe `FunctionalComponent`*

O primeiro passo para a criação de componentes é a implementação de uma subclasse da classe `FunctionalComponent`. Na Listagem de Código B.39 descreve-se um exemplo de implementação de uma classe referente a um componente funcional. O construtor da superclasse possui um parâmetro do tipo *string* referente ao nome do componente. No exemplo em questão, cria-se um construtor para a classe `MyComponent`, sem parâmetros, mas não há restrições no arcabouço quanto à parametrização do construtor.

Listagem de Código B.39: Exemplo de extensão da classe `FunctionalComponent`.

```
1 class MyComponent : public FunctionalComponent { public:
2     inline MyComponent() : FunctionalComponent("My Component") { }
3
4     inline virtual ~MyComponent() {}
5 };
```

*2. Implementando o componente, acessando serviços e anunciando eventos*

Uma vez criada a classe referente ao componente funcional (`MyComponent`), pode-se construir os métodos que serão declarados posteriormente como implementadores de serviços providos e eventos de interesse. Estes métodos devem ter visibilidade pública para que possam ser acessados via reflexão, como descrito na Listagem de Código B.40 (linhas 8 e 22).

Considere, por exemplo, que o método `print` será disponibilizado como implementador do serviço “print”. Este serviço recebe um texto como parâmetro, formata-o e o exibe na tela. Porém, esta formatação deve ser realizada por outro componente, cujo serviço é aqui denominado “format”. Nas linhas 12, 13 e 14, descreve-se a sintaxe para a criação da requisição de serviço, invocação do mesmo através do método `doIt` e exibição da resposta. Caso haja alguma exceção na execução do serviço implementado no componente provedor, esta informação é recuperada atra-

vés do método `getException` da classe `ServiceResponse`. Se houver um erro na execução do arcabouço, tal como `ServiceNotImplementedException`, a execução será redirecionada para a cláusula `catch` (linha 16). No caso de chamadas assíncronas, utiliza-se o método `doItAsync` que retorna um identificador da requisição e executa a requisição em uma nova linha de execução. O desenvolvedor deve sobrescrever o método `receiveAsyncReponse` para receber a resposta assíncrona.

Listagem de Código B.40: Exemplo de invocação de serviço.

```
1  class MyComponent : public FunctionalComponent {
2  public:
3      inline MyComponent() : FunctionalComponent("My Component") {}
4
5      inline virtual ~MyComponent() {}
6
7      /* Método implementador do serviço "print" */
8      inline void print(string _text){
9          vector <void *> param;
10         param.push_back(&_text);
11         try{
12             ServiceRequest request("format" , param);
13             ServiceResponse * response = doIt(request);
14             cout << *(string *)response->getData() << endl;
15             delete response;
16         }catch (exception & e){
17             cout << e.what() << endl;
18         }
19     }
20
21     /* Método implementador do tratamento do evento "saved" */
22     inline void receiveSaved(string _fileName){
23         cout << "Arquivo salvo: " << _fileName << endl;
24     }
25 };
```

O anúncio de eventos ocorre de forma semelhante. Considere, por exemplo, que o mesmo método anuncia um evento denominado “printed” após imprimir com sucesso o resultado na tela. Esta funcionalidade é descrita nas linhas 15 e 16 da Listagem de Código B.41. Neste exemplo, o evento não possui parâmetros. Para eventos com parâmetros, basta passá-los como segundo argumento do construtor de `EventAnnouncement`. A invocação de eventos é assíncrona, ou seja, o fluxo de execução irá continuar independente da linha de execução que realizará a notificação

do evento.

Listagem de Código B.41: Exemplo de anúncio de evento.

```
1  class MyComponent : public FunctionalComponent {
2  public:
3      inline MyComponent() : FunctionalComponent("My Component") {}
4      inline virtual ~MyComponent() {}
5      /* Método implementador do serviço "print" */
6      inline void print(string _text){
7          vector <void *> param;
8          param.push_back(&_text);
9          try{
10             ServiceRequest request("format" , param);
11             ServiceResponse * response = doIt(request);
12             if(!response->hasException()){
13                 cout << *(string *)response->getData() << endl;
14                 //Anúncio do evento
15                 EventAnnouncement * announcement = new EventAnnouncement("printed");
16                 announceEvent(announcement);
17             }else{
18                 cout << "Problemas na formatação: " << response->getException()->what() << endl;
19             }
20             delete response;
21         }catch (exception & e){cout << e.what() << endl;}
22     }
23     /* Método implementador do tratamento do evento "saved" */
24     inline void receiveSaved(){cout << "Salvou!!!" << endl;}
25 };
```

É importante ressaltar que a execução dos métodos `doIt`, `announceEvent` e `doItAsync` só fazem sentido quando o componente faz parte de uma hierarquia de aplicação baseada na CMS. Caso contrário, a instanciação da classe `MyComponent` seguida da invocação do método `print` causará diversas exceções na execução.

*3. Declarando e adicionando serviços e eventos aos componentes*

Uma vez implementada a classe principal do componente, deve-se agora declarar quais dos seus métodos são implementadores de serviços providos e de tratamento de eventos de interesse. Da mesma forma, deve-se declarar quais são os serviços requeridos pelo componente “My Component” e também quais são os eventos anunciados por ele (Listagem de Código B.42).

As declarações de serviços e eventos devem ser feitas no próprio construtor do componente. Isto é necessário para garantir que, uma vez que o componente esteja instanciado, todos os seus

serviços e eventos já estejam devidamente declarados.

Listagem de Código B.42: Declaração de serviços e eventos.

```
1  Type type = Type::ByName("MyComponent");
2  ServiceSpecification *spec1, *spec3; ProvidedService *service1;
3  RequiredService * service3; EventSpecification *spec2, *spec4;
4  EventOfInterest *event2; AnnouncedEvent * event4;
5  Member method1, method2; list<ComporType *> parameters;
6  list<exception *> exceptions; set<string> reqs;
7
8  //Declarando e adicionando serviço provido "print"
9  parameters.push_back(new ComporType("string", "name"));
10 reqs.insert("format");
11 spec1 = new ServiceSpecification("print", "Imprime nome formatado na tela.",
12                                  parameters, exceptions, NULL, destroyParameters);
13 method1 = type.MemberByName("print");
14 service1 = new ProvidedService(spec1, method1, reqs);
15 addProvidedService(service1);
16
17 //Declarando e adicionando evento de interesse "saved"
18 parameters.clear();
19 parameters.push_back(new ComporType("string", "fileName"));
20 spec2 = new EventSpecification("saved", "Arquivo salvo.",
21                                parameters);
22 method2 = type.MemberByName("receiveSaved");
23 event2 = new EventOfInterest(spec2, method2);
24 addEventOfInterest(event2);
25
26 //Declarando e adicionando serviço requerido "format"
27 parameters.clear();
28 parameters.push_back(new ComporType("string", "name"));
29 spec3 = new ServiceSpecification("format", "Formata o nome.", parameters,
30                      exceptions, new ComporType("string", "formatado"));
31 service3 = new RequiredService(spec);
32 addRequiredService(service3);
33
34 //Declarando e adicionando evento anunciado "printed"
35 parameters.clear();
36 spec4 = new EventSpecification("printed", "Nome impresso na tela.",
37                                parameters);
38 event4 = new AnnouncedEvent(spec4);
39 addAnnouncedEvent(event4);
```

Nas linhas 8 a 15 da Listagem de Código B.42 implementa-se a declaração e a adição do serviço provido “print”, cujo método implementador é definido como sendo o método `print` da

classe `MyComponent`. Na linha 5, cria-se uma lista contendo o tipo do parâmetro do serviço, neste caso, *string*. A lista de serviços requeridos, no caso apenas "format", é criada na linha 6. Na linha 11, cria-se uma especificação do serviço, definindo o nome, descrição, parâmetros, exceções, retorno e função para destruir os parâmetros, sendo o antepenúltimo argumento uma lista de exceções vazia e o penúltimo um valor nulo, pois o serviço não lança exceções e também não retorna resultado. Na linha 13, recupera-se uma referência do método `print` usando reflexão. Por fim, nas linhas 14 e 15, cria-se uma instância de `ProvidedService`, a qual é adicionada ao componente. A partir de então, o componente "My Component" passa a prover o serviço denominado "print".

A definição do método `receiveSaved` como implementador do tratamento do evento "saved" ocorre de forma similar à definição de serviço provido. A única diferença é o formato da especificação de eventos que não possui exceções e nem retorno, como ilustrado nas linhas 18 a 24. Uma vez declarado o evento de interesse, as próximas notificações do evento "saved" serão recebidas pelo componente através do método `receiveSaved`. É importante observar que, como neste exemplo, o nome do evento ou do serviço pode ser diferente do nome do método.

Por fim, nas linhas 27 a 32 descreve-se o código que implementa a declaração do serviço requerido "format". Os serviços requeridos a serem declarados são aqueles invocados pelo método `doIt` ou `doItAsync`. O mesmo ocorre para a declaração de eventos que serão anunciados pelos componentes. Neste exemplo, apenas o evento "printed" foi anunciado, através da invocação do método `announceEvent`, e por isso deve ser declarado como descrito nas linhas 35 a 39.

*4. Implementando a inicialização e a interrupção dos componentes*

Códigos de inicialização e interrupção dos componentes são implementados sobrescrevendo os métodos `startImpl` e `stopImpl`, como descrito nas linhas 6 e 7 da Listagem de Código B.43.

As propriedades necessárias à inicialização do componente devem ser declaradas no construtor da classe do componente. Na linha 3, ilustra-se a criação de uma propriedade de inicialização e sua posterior utilização é descrita na linha 27. De acordo com a aplicação, o valor desta propriedade poderá ser redefinido. Caso contrário, será considerado o seu valor padrão, também descrito na linha 3. Para desalocar as propriedades de inicialização, o componente deve sobrescrever os métodos `destroyInitializationProperties`, como descrito nas linhas 9 e 17. Este processo de desalocação é uma característica específica do CCF, em relação às demais implementações da CMS.

Listagem de Código B.43: Inicialização do componente.

```
1  class MyComponent : public FunctionalComponent { public:
2      inline MyComponent() : FunctionalComponent("My Component"){
3          putInitializationPropertyValue("saved_message", new string("Salvou!"));
4      }
5      inline virtual ~MyComponent() {}
6      inline void startImpl() { cout << "Iniciando o componente..." << endl; }
7      inline void stopImpl() { cout << "Finalizando o componente..." << endl; }
8
9      inline void destroyInitializationProperties(){
10         map<string, void *>::iterator itr = m_initializationProperties.begin();
11         for (; itr != m_initializationProperties.end(); itr++){
12             delete (string *)itr->second;
13         }
14         m_initializationProperties.clear();
15     }
16
17     inline void destroyInitializationProperties(const string & _propertyName){
18         map<string, void *>::iterator itr = m_initializationProperties.find(_propertyName);
19         if (itr != m_initializationProperties.end()){
20             delete (string *)itr->second;
21             m_initializationProperties.erase(_propertyName);
22         }
23     }
24
25     /* Método implementador do tratamento do evento "saved" */
26     inline void receiveSaved(){
27         cout << *(string *)getInitializationPropertyValue("saved_message") << endl;
28     }
29 };
```

### Criando componentes personalizáveis e interesses

*1. Criando interesses*

O primeiro passo é a criação de interesses. Na Listagem de Código B.44, descreve-se a criação da classe `LogConcern`. Esta classe estende `Concern` para adicionar o comportamento de registro de informação nas requisições de serviço, antes (linha 6) e depois (linha 12) do serviço a ser executado. Ainda é possível interceptar a execução quando ocorre exceção sobrescrevendo o método `interceptOnException(...)`, o que não foi realizado neste exemplo para ressaltar que só é necessário sobrescrever os comportamentos que se deseja interceptar.

Listagem de Código B.44: Exemplo de criação de interesse.

```
1  class LogConcern : public Concern {
2  public:
3      inline LogConcern() : Concern("log") {}
4
5      //Interceptando requisições de serviços antes da execução
6      inline void interceptBefore(CustomizableComponent * _component,
7                                  ServiceRequest & _request){
8          cout << "Log - Antes : " << _request.getServiceName();
9      }
10
11     //Interceptando requisições de serviço depois da execução
12     inline void interceptAfter(CustomizableComponent * _component,
13                           ServiceRequest & _request, ServiceResponse * _response){
14         cout << "Log - Depois : " << _request.getServiceName();
15     }
```

*2. Criando componentes personalizáveis*

Após a criação dos interesses, pode-se criar componentes personalizáveis. A criação de um componente personalizável ocorre via extensão da classe `CustomizableComponent`. A implementação deste tipo de componente ocorre da mesma forma que um componente funcional comum, com publicação de serviços, eventos e propriedades de inicialização. A única diferença é que um componente personalizável torna possível a inserção de interesses que podem ser ativados e desativados posteriormente pelo desenvolvedor da aplicação.

Na Listagem de Código B.45, descreve-se a criação de um componente personalizável. Na linha 6, adiciona-se o interesse de registro criado anteriormente (`LogConcern`), inicializando-o como um interesse desativado. A partir de então, este interesse pode ser ativado pelo desenvolvedor da aplicação quando este desejar executar a funcionalidade de registro de informações.

Listagem de Código B.45: Exemplo de criação de componente personalizável.

```
1  class NameFormatter : public CustomizableComponent {
2  public:
3      inline NameFormatter() : CustomizableComponent("Name Formatter"){
4              ...
5              //Adicionando interesse ao componente...
6              addConcern(new LogConcern(), false);
7              ...
8      }
```

**Criando adaptadores**

*1. Criando eventos e serviços adaptados*

A criação de eventos e serviços adaptados ocorre através da extensão das classes `AdaptedEvent` e `AdaptedService`. Nas listagens de código B.46 e B.47, descreve-se a implementação das classes `MyAdaptedService` e `MyAdaptedEvent`, que implementam a adaptação do serviço "print" e do tratamento do evento "saved" do componente "My Component".

Listagem de Código B.46: Exemplo de criação de serviço adaptado.

```
1  class MyAdaptedService : public AdaptedService {
2  public:
3      inline MyAdaptedService(ServiceSpecification * _specification) :
4          AdaptedService(_specification) {}
5      //Implementação da adaptação do serviço ...
6      inline ServiceResponse * invokeService(ServiceRequest & _serviceRequest)
7          throw(ComporInvalidParametersException,
8                ComporIllegalAccessMethodException,
9                ComporComponentNotStartedException){
10         cout << "Adaptação";
11         //Acesso ao serviço original
12         return getAdapter()->invokeOriginalService(_serviceRequest);
13     }
```

Listagem de Código B.47: Exemplo de criação de evento adaptado.

```
1  class MyAdaptedEvent : public AdaptedEvent {
2  public:
3      inline MyAdaptedEvent(EventSpecification * _specification) :
4          AdaptedEvent(_specification) {}
5      //Implementação da adaptação do evento ...
6      inline void invokeEvent(EventAnnouncement & _event){
7          cout << "Adaptação";
8          //Acesso ao evento original
9          getAdapter()->invokeOriginalEvent(_event);
10     }
```

*2. Definindo adaptadores para componentes*

Uma vez definidos os serviços e eventos adaptados, basta criar uma instância da classe Adapter, adicionar os serviços e eventos adaptados a esta instância e definir o componente funcional sendo adaptado. Na Listagem de Código B.48, descreve-se a implementação do componente "My Component". A especificação do serviço e evento adaptado deve ser a mesma daqueles originais. Por isso, nas linhas 3 e 4, faz-se referência às especificações definidas na Listagem de

Código B.42. A partir de então, as requisições ao serviço “print” e as notificações do evento “saved” serão redirecionadas aos respectivos adaptadores.

Listagem de Código B.48: Exemplo de criação de adaptador.

```
1 Adapter * adapter = new Adapter("Adaptador de My Component");
2
3 adapter->addAdaptedService(new MyAdaptedService(spec1));
4 adapter->addAdaptedEvent(new MyAdaptedEvent(spec2));
5 adapter->setFunctionalComponent(myComponent);
```

**Criando aplicações**

O exemplo de arquitetura ilustrado no início do capítulo, mais especificamente na Figura B.1, é utilizado para descrever a criação de uma aplicação no CCF.

*1. Instanciando e configurando os componentes*

O primeiro passo para a construção da aplicação é a instanciação e a configuração dos seus componentes funcionais (Listagem de Código B.49). Na aplicação de exemplo, as classes dos componentes a serem instanciadas são `MyComponent` e `NameFormatter` (linhas 2 e 3). Após a instanciação, deve-se definir, quando necessário, os valores das propriedades de inicialização, apelidos de serviços e eventos e personalizar a execução definindo os interesses ativos.

Para efeito de explicação do exemplo, considere que o componente “Name Formatter” provê o serviço “format” e anuncia o evento “logSaved”, que possui a mesma especificação do evento de interesse do componente “My Component” (“saved”) mas tem o nome diferente. Sendo assim, deve-se redefinir o apelido do evento ou do lado do interessado ou do lado do anunciante. Na linha 6, altera-se o apelido do evento do lado do anunciante, no caso, o componente “Name Formatter”. A partir de então, o serviço que era anunciado como “logSaved”, passa a ser anunciado como “saved” e assim a notificação chegará ao componente “My Component”. O mesmo processo ocorre na redefinição de apelidos de serviços. De uma forma padrão, todos os apelidos de serviços e eventos são iguais aos seus nomes. A redefinição de apelidos só é utilizada quando os nomes do lado do requisitante e do lado do provedor são diferentes.

Por fim, na linha 9, a execução do componente “Name Formatter” é personalizada através da ativação do interesse de registro (log). A partir de então, a funcionalidade de registro implementada neste interesse será executada quando o componente for iniciado.

Listagem de Código B.49: Exemplo de instanciação e configuração de componentes.

```
1 //Instanciando os componentes...
2 NameFormatter * nameFormatter = new NameFormatter();
3 MyComponent * myComponent = new MyComponent();
4
5 //Redefinindo o apelido do evento no componente anunciante...
6 nameFormatter->setAliasOfAnnouncedEvent("logSaved" , "saved");
7
8 //Personalizando a execução do componente...
9 nameFormatter->getConcernSwitch().activate("log");
```

*2. Criando a hierarquia da aplicação*

Na Listagem de Código B.50, descreve-se a construção da hierarquia da aplicação de exemplo. Nas linhas 1, 2 e 3 são instanciados os contêineres. Observe que o contêiner raiz deve ser instanciado utilizando a classe `ScriptContainer` para que seja possível associá-lo a um *script* de execução posteriormente. Nas linhas 5, 6, 7 e 8 os componentes são adicionados aos contêineres "Printing" e "Formatting" e estes são adicionados ao contêiner "Root". Depois deste processo, a hierarquia da aplicação está pronta para ser executada, faltando apenas a definição de um *script* de execução para que a aplicação baseada na CMS esteja completa.

Listagem de Código B.50: Exemplo de construção de hierarquia da aplicação.

```
1 ScriptContainer * root = new ScriptContainer("Root");
2 Container * printing = new Container("Printing");
3 Container * formatting = new Container("Formatting");
4
5 printing->addComponent(myComponent);
6 formatting->addComponent(nameFormatter);
7 root->addComponent(printing);
8 root->addComponent(formatting);
```

**Executando aplicações**

*1. Criando o* script *de execução*

Uma vez criada a hierarquia da aplicação, deve-se então definir o *script* de execução responsável por iniciar a execução da mesma. Quando os componentes funcionais da aplicação implementam todas as funcionalidades, inclusive a interface de interação com o usuário, pode-se criar um *script* de execução apenas instanciando a classe `ExecutionScript`, cujo construtor recebe como parâmetro uma instância de `ScriptContainer`. Para exemplificar a exten-

são de ExecutionScript, na Listagem de Código B.51 descreve-se a implementação da classe MyScript.

Listagem de Código B.51: Exemplo de criação de *script* de execução.

```
1  class MyScript: public ExecutionScript {
2  public:
3      inline MyScript(ScriptContainer *_rootContainer):
4          ExecutionScript(_rootContainer){}
5      inline virtual ~MyScript(){}
6  protected:
7      inline void main(){
8          vector <void *> param;
9          cout << "Nome formatado abaixo..." << endl;
10         param.push_back(new string("nome"));
11         ServiceRequest req("print", param);
12         try{
13             exec(req);
14         }catch(exception & e){
15             cout << e.what() << endl;
16         }
17         cout << "Fim!!!" << endl;
18     }
19 };
```

O método main é executado na inicialização do *script* de execução e através dele é possível acessar a hierarquia de componentes utilizando os métodos exec e execAsync, para invocar serviços, e announceEvent, para anunciar eventos. Neste exemplo, o serviço "print" é executado na linha 13. Notificações de eventos também podem ser recebidas sobrescrevendo o método receiveEvent.

*2. Iniciando a aplicação*

A inicialização da aplicação é realizada através da instanciação e inicialização do *script* de execução (Listagem de Código B.52).

Listagem de Código B.52: Exemplo de criação de *script* de execução.

```
1  MyScript * myScript = new MyScript(root);
2  myScript->init();
```

Para instanciar o *script*, deve-se definir o contêiner raiz da aplicação. Depois, basta iniciar a aplicação utilizando o método init que primeiro iniciará o contêiner raiz (e conseqüentemente toda a hierarquia) e depois invocará o método main. A finalização do *script* pode ser realizada

através do método `finish`, que requisita a interrupção da execução da hierarquia de componentes.

## B.4 *Sharp Component Framework* (SCF)

### B.4.1 Implementação do SCF

A API da linguagem CSharp, assim como a de Java, já possui funcionalidades nativas para a implementação dos mecanismos de execução assíncrona e reflexão computacional do SCF. Da mesma forma, os estereótipos de tipos apresentados no Apêndice A possuem implementação direta na API padrão de CSharp, como descrito na Tabela B.4.

| **Estereótipo** | **Descrição** |
|---|---|
| <string> | Classe `System.String` |
| <collection> | Classe `System.Collections.ArrayList` |
| <table> | Classe `System.Collections.HashTable` |
| <generic> | Classe `System.Type` |
| <class> | Classe `System.Object` |
| <method> | Classe `System.Reflection.MethodInfo` |
| <list> | Classe `System.Collections.ArrayList` |
| <set> | — |
| <list-set> | — |
| <exception> | Classe `System.Exception` |

Tabela B.4: Estereótipos referentes a tipos e estruturas de dados em CSharp.

#### Reflexão Computacional

As principais classes para reflexão em CSharp estão localizadas no espaço de nomes (*namespace*) `System.Reflection`. `System.Type` é a principal classe para fazer a descoberta dos metadados do objeto. Sendo assim, antes de qualquer coisa, é necessário obter o objeto `Type` do tipo que se deseja fazer reflexão. Para isso, pode-se utilizar o operador `typeof` ou o método `getType`.

Dentre as classes do *namespace* `System.Reflection`, a classe `MethodInfo` é utilizada juntamente com a classe `Type` para implementar os mecanismos de interação baseada em serviços e

eventos especificados na CMS.

Como já discutido anteriormente, de uma forma geral, apenas duas funcionalidades são requeridas para implementar os mecanismos de reflexão: a possibilidade de armazenar em um atributo uma referência para um método de uma classe; e a possibilidade de recuperar esta referência e invocar o método posteriormente.

A primeira funcionalidade é exemplificada na Listagem de Código B.53. A classe de exemplo `ReflectionInCSharp` possui um atributo privado declarado na linha 4 denominado `method` do tipo `MethodInfo`, cujo valor pode ser recuperado através do método `GetMethod()` (linha 8).

Listagem de Código B.53: Exemplo de instância de `MethodInfo` armazenada como atributo de uma classe.

```
1  using System;
2  using System.Reflection;
3  public class ReflectionInCSharp{
4      private MethodInfo method; // Atributo de referência a um método.
5      public ReflectionInCSharp() {
6          try{
7              Type thisClass = this.GetType(); // Referência à classe.
8              this.method = thisClass.GetMethod("print", new Type[]{typeof(String)});
9          }catch(System.ArgumentException){
10             Console.WriteLine("Argumentos inválidos.");
11         }catch(System.Reflection.AmbiguousMatchException){
12             Console.WriteLine("Mais de um método com mesmo nome.");
13         }
14     }
15     public MethodInfo getMethod() {return this.method;}
16     public void print(String param){
17         Console.WriteLine("Reflexão em CSharp. "+param);
18     }
19 }
```

No construtor da classe `ReflectionInCSharp`, na linha 7, a classe `Type` é utilizada para recuperar a referência à classe do objeto atual, que é armazenada na variável `thisClass`. Esta variável é utilizada para acessar as informações da classe, como a referência ao método `print(String)`, através do nome e do tipo de parâmetro. Na linha 8, tem-se a atribuição da referência do método `print(String)` ao atributo `method` da classe.

A segunda funcionalidade relacionada à implementação do mecanismo de reflexão diz respeito à recuperação da referência do método armazenado no atributo da classe e posterior invocação do

mesmo. Isto pode ser feito através do método `Invoke()` da classe `MethodInfo`, como descrito na Listagem de Código B.54. A referência do método é recuperada na linha 8, a invocação é realizada na linha 9. O resultado da invocação realizada na linha 9 é o mesmo de uma chamada explícita do método test da classe `Reflection` (linha 10): "Reflexão em CSharp. Funciona!".

Listagem de Código B.54: Exemplo de invocação de método via reflexão.

```
1  using System;
2  using System.Reflection;
3
4  public class Test{
5      public static void Main(String[] args) {
6          try {
7              ReflectionInCSharp _myClass = new ReflectionInCSharp();
8              MethodInfo method = _myClass.getMethod();
9              method.Invoke(_myClass, new Object[]{"Funciona!"});
10             _myClass.escrever("Funciona!");
11         }catch(Exception){
12             throw new System.Exception();
13         }
14     }
15 }
```

Através das funcionalidades exemplificadas nas listagens de código B.53 e B.54, torna-se possível implementar o suporte necessário à definição de serviços providos e eventos de interesse, os quais necessitam armazenar uma referência de um método para posterior invocação pelo SCF. O funcionamento deste mecanismo no SCF é similar ao implementado no SCF.

### Execução Assíncrona

A execução assíncrona foi implementada utilizando o conceito de `Threads`. Esta funcionalidade permite escrever programas com múltiplas linhas de execução. Em CSharp, o *namespace* `System.Threading` provê um conjunto de classes e interfaces para o controle de programação concorrente.

A forma mais simples de criar uma *thread* em CSharp é criando uma instância da classe `Thread`. O construtor dessa classe recebe apenas um argumento: uma instância delegada. A linguagem provê a classe delegada `ThreadStart` para este propósito especificamente, que aponta para o método que for designado.

Na Listagem de Código B.55, tem-se um exemplo da utilização de `Threads`. Neste exemplo,

foram criados dois métodos: um método chamado `Inc` que conta de zero até mil (linhas 17 e 21); e outro método chamado `Dec` que conta regressivamente de mil até zero (linhas 22 e 26). Para executar tais métodos em linhas de execução diferentes, deve-se criar duas novas *threads*, cada uma inicializada com uma `ThreadStart` (linhas 11 a 14).

Listagem de Código B.55: Implementação dos métodos para invocação assíncrona

```
1  using System;
2  using System.Threading;
3
4  class TestThreads  {
5      static void Main(){
6          TestThreads t = new TestThreads();
7          t.DoTest();
8      }
9
10     public void DoTest(){
11         Thread t1 = new Thread(new ThreadStart(Inc));
12         Thread t2 = new Thread(new ThreadStart(Dec));
13         t1.Start();
14         t2.Start();
15     }
16
17     public void Inc() {
18         for(int i = 0; i <= 1000; i++){
19             System.Console.WriteLine("Inc: {0}", i );
20         }
21     }
22     public void Dec(){
23         for (int i = 1000; i >= 0; i--){
24             System.Console.WriteLine("Dec: {0}", i );
25         }
26     }
27 }
```

Seguindo o projeto de classes do GCF, a classe abstrata `AsyncMethodInvocation` utiliza o mecanismo de *threads* de `System.Threading` e implementa uma invocação assíncrona de método.

Classes que necessitem implementar invocações assíncronas de serviços e eventos devem implementar as interfaces `AsyncServiceInvokerIF` e `AsyncEventInvokerIF`, as quais são referenciadas por classes que estendem a classe `AsyncMethodInvocation` e implementam métodos voltados especificamente para o tratamento de serviços e eventos.

No caso do SCF, foram criadas classes de gerenciamento de invocações assíncronas para componentes funcionais (classe `FunctionalComponentAsyncManager`) e para *scripts* de execução (classe `ExecutionScriptAsyncManager`). Desta forma, as classes `FunctionalComponent` e `ExecutionScript` fazem uso destas classes para invocar serviços e anunciar eventos assincronamente. A implementação no SCF é similar à implementação realizada no JCF.

### B.4.2 Como utilizar o arcabouço SCF?

A seguir, descreve-se como utilizar a API do arcabouço SCF para construir componentes e aplicações. Apenas as principais funcionalidades são descritas. Vale ressaltar que na versão atual do SCF descrita a seguir ainda não se encontram implementados os modelos de adaptadores e separação de interesses definidos na CMS. Mais detalhes sobre a API e o código fonte podem ser encontrados no site do arcabouço (`http://gforge.embedded.ufcg.edu.br/projects/scf`).

#### Criando componentes

*1. Estendendo a classe `FunctionalComponent`*

O primeiro passo para a criação de componentes é a implementação de uma subclasse da classe `FunctionalComponent`. Na Listagem de Código B.56 descreve-se um exemplo de implementação de uma classe referente a um componente funcional. O construtor da superclasse possui um parâmetro do tipo *String* referente ao nome do componente. No exemplo em questão, cria-se um construtor para a classe `MyComponent`, sem parâmetros, mas não há restrições no arcabouço quanto à parametrização do construtor.

Listagem de Código B.56: Exemplo de extensão da classe `FunctionalComponent`.

```
1 public class MyComponente : FunctionalComponent{
2         public MyComponente() : base("MyComponente"){}
3 }
```

*2. Implementando o componente, acessando serviços e anunciando eventos*

Uma vez criada a classe referente ao componente funcional (`MyComponent`), pode-se construir os métodos que serão declarados posteriormente como implementadores de serviços providos e eventos de interesse. Estes métodos devem ter visibilidade pública para que possam ser acessados via reflexão, como descrito na Listagem de Código B.57 (linhas 7 e 18).

Listagem de Código B.57: Exemplo de invocação de serviço.

```
1  using System;
2
3  public class MyComponent: FunctionalComponent{
4      public MyComponent(): base("MyComponent"){}
5
6      //Método implementador do serviço "print"
7      public void print(String text) {
8          try {
9              ServiceRequest request = new ServiceRequest("format",new Object[]{text});
10             ServiceResponse response = base.doIt(request);
11             Console.WriteLine(response.getData());
12         }catch(System.Exception e){
13                         Console.WriteLine(e.StackTrace());
14         }
15     }
16
17     //Método implementador do tratamento de evento "saved"
18     public void receiveSaved(String fileName) {
19         Console.WriteLine("Arquivo salvo: " + fileName);
20     }
21 }
```

No exemplo da Listagem de Código B.57, o método `print(String)` implementa o serviço "print", que recebe um texto como argumento e exibe-o formatado na tela. Este componente não possui o método para realizar tal formatação, sendo necessário requisitar este serviço a outro componente. Nas linhas 9, 10 e 11, descreve-se a sintaxe para a criação da requisição deste serviço, através do método `doIt(...)`. Se houver algum erro na execução ou se tal serviço não estiver implementado por nenhum outro componente, retorna-se como resultado uma exceção, que será capturada pela cláusula `catch` (linha 12). No caso de chamadas assíncronas, utiliza-se o método `doItAsync(...)`.

O anúncio de eventos ocorre de forma semelhante. Considere, por exemplo, que o mesmo método anuncia um evento denominado "printed" após imprimir com sucesso o resultado na tela. Esta funcionalidade é descrita nas linhas 14 e 15 da Listagem de Código B.58.

Neste exemplo, o evento não possui parâmetros. Para eventos com parâmetros, basta passá-los como um segundo argumento do construtor da classe `EventAnnouncement`. A invocação de eventos é assíncrona, ou seja, o fluxo de execução irá continuar independente da linha de execução que realizará a notificação do evento.

É importante ressaltar que a execução dos métodos `doIt`, `announceEvent` e `doItAsync` só fazem sentido quando o componente faz parte de uma hierarquia de aplicação baseada na CMS. Caso contrário, a instanciação da classe `MyComponent` seguida da invocação do método `print` causará diversas exceções na execução.

Listagem de Código B.58: Exemplo de anúncio de evento.

```
1  using System;
2
3  public class MyComponent: FunctionalComponent{
4      public MyComponent():base("'MyComponent"){}
5
6      //Método implementador do serviço "print"
7      public void print(String text) {
8          try {
9              ServiceRequest request = new ServiceRequest("format",new Object[]{text});
10             ServiceResponse response = base.doIt(request);
11             if (!response.hasException()){
12                 Console.WriteLine(response.getData());
13                 //Anúncio de evento
14                 EventAnnoucement announcement = new EventAnnouncement("printed");
15                 base.annouceEvent(announcement);
16             }else{
17                 Console.WriteLine("Problemas na formatação: " + response.getException( ));
18             }
19         }catch(System.Exception e){
20             Console.WriteLine(e.StackTrace());
21         }
22     }
23
24     //Método implementador do tratamento de evento "saved"
25     public void receiveSaved() {
26         Console.WriteLine("Arquivo Salvo!!!");
27     }
28 }
```

*3. Declarando e adicionando serviços e eventos aos componentes*

Uma vez implementada a classe principal do componente, deve-se agora declarar quais dos seus métodos são implementadores de serviços providos e de tratamento de eventos de interesse. Da mesma forma, deve-se declarar quais são os serviços requeridos pelo componente “My Component” e também quais são os eventos anunciados por ele (Listagem de Código B.59).

As declarações de serviços e eventos devem ser feitas no próprio construtor do componente.

Isto é necessário para garantir que, uma vez que o componente esteja instanciado, todos os seus serviços e eventos já estejam devidamente declarados.

Listagem de Código B.59: Declaração de serviços e eventos.

```
1  //Declarando e adicionando serviço provido "print"
2  ArrayList params1 = new ArrayList(); params1.Add(new ComporType("name",typeof(String)));
3  ArrayList reqs = new ArrayList(); reqs.Add("format");
4  ServiceSpecification spec1 = new ServiceSpecification("print","Imprime texto formatado.",
5                                  params1,null,null);
6  MethodInfo method1 = this.GetType().GetMethod("print",new Type[]{typeof(String)});
7  base.addProvidedService(new ProvidedService(spec1,method1,reqs));
8
9  //Declarando e adicionando evento de interesse "saved"
10 ArrayList params2 = new ArrayList(); params2.Add(new ComporType("fileName",typeof(String)));
11 EventSpecification spec2 = new EventSpecification("saved","Arquivo salvo.",params2);
12 MethodInfo method2 = this.GetType().GetMethod("receiveSaved",new Type[]{typeof(String)});
13 base.addEventOfInterest(new EventOfInterest(spec2,method2));
14
15 //Declarando e adicionando serviço requerido "format"
16 ArrayList params3 = new ArrayList(); params3.Add(new ComporType("name",typeof(String)));
17 ServiceSpecification spec3 = new ServiceSpecification("format","Formata texto.",
18                                      params3,null,new ComporType("formatado",typeof(String)));
19 base.addRequiredService(new RequiredService(spec3));
20
21 //Declarando e adicionando evento anunciado "printed"
22 EventSpecification spec4 = new EventSpecification("printed","Texto impresso.",null);
23 base.addAnnouncedEvent(new AnnouncedEvent(spec4));
```

Nas linhas 2 a 7 da Listagem de Código B.59, implementa-se a declaração e a adição do serviço provido “print”, cujo método implementador é definido como sendo o método `print` da classe `MyComponent`. Na linha 2, cria-se uma lista contendo o tipo do parâmetro do serviço, neste caso, *String*. A lista de serviços requeridos, no caso apenas “format”, é criada na linha 3. Na linha 4, cria-se uma especificação do serviço, definindo o nome, descrição, parâmetros, exceções, retorno e função para destruir os parâmetros, sendo os dois últimos argumentos nulos, pois o serviço não lança exceções e também não retorna resultado. Na linha 6, recupera-se uma referência do método `print` usando reflexão. Por fim, na linha 7, cria-se uma instância de `ProvidedService`, a qual é adicionada ao componente. A partir de então, o componente “MyComponent” passa a prover o serviço denominado “print”.

A definição do método `receiveSaved` como implementador do tratamento do evento “saved” ocorre de forma similar à definição de serviço provido. A única diferença é o formato da espe-

cificação de eventos que não possui exceções e nem retorno, como ilustrado nas linhas 10 a 13. Uma vez declarado o evento de interesse, as próximas notificações do evento “saved” serão recebidas pelo componente através do método `receiveSaved`. É importante observar que, como neste exemplo, o nome do evento ou do serviço pode ser diferente do nome do método.

Por fim, nas linhas 16 a 19 descreve-se o código que implementa a declaração do serviço requerido “format”. Os serviços requeridos a serem declarados são aqueles invocados pelo método `doIt` ou `doItAsync`. O mesmo ocorre para a declaração de eventos que serão anunciados pelos componentes. Neste exemplo, apenas o evento “printed” foi anunciado, através da invocação do método `announceEvent`, e por isso deve ser declarado como descrito nas linhas 22 e 23.

*4. Implementando a inicialização e a interrupção dos componentes*

Códigos de inicialização e interrupção dos componentes são implementados sobrescrevendo os métodos `startImpl` e `stopImpl`, como descrito nas linhas 6 e 7 da Listagem de Código B.60.

Listagem de Código B.60: Inicialização do componente.

```
1  public class MyComponent: FunctionalComponent{
2      public MyComponent() : base("MyComponent"){
3          base.putInitializationPropertyValue ("message","Está salvo!") ;
4      }
5
6      public override void startImpl(){Console.WriteLine ("Iniciando componente ... ");}
7      public override void stopImpl(){Console.WriteLine("Finalizando componente ... ");}
8
9      public void receiveSaved(){
10         Console.WriteLine(base.getInitializationPropertyValue( "message" ));
11     }
12 }
```

As propriedades necessárias à inicialização do componente devem ser declaradas no construtor da classe do componente. Na linha 3, da Listagem de Código B.60, ilustra-se a criação de uma propriedade de inicialização e sua posterior utilização é descrita na linha 10. De acordo com a aplicação, o valor desta propriedade poderá ser redefinido. Caso contrário, será considerado o seu valor padrão, também descrito na linha 3.

### Criando aplicações

O exemplo de arquitetura ilustrado no início do capítulo, mais especificamente na Figura B.1, é utilizado para descrever a criação de uma aplicação no SCF.

*1. Instanciando e configurando os componentes*

O primeiro passo para a construção da aplicação é a instanciação e a configuração dos seus componentes funcionais (Listagem de Código B.61). Na aplicação de exemplo, as classes dos componentes a serem instanciadas são `MyComponent` e `NameFormatter` (linhas 2 e 3). Após a instanciação, deve-se definir, quando necessário, os valores das propriedades de inicialização, apelidos de serviços e eventos e personalizar a execução definindo os interesses ativos, sendo estes últimos ainda não disponíveis na API do SCF.

Para efeito de explicação do exemplo, considere que o componente "Name Formatter" provê o serviço "format" e anuncia o evento "logSaved", que possui a mesma especificação do evento de interesse do componente "My Component" ("saved") mas tem o nome diferente. Sendo assim, deve-se redefinir o apelido do evento ou do lado do interessado ou do lado do anunciante. Na linha 6, altera-se o apelido do evento do lado do anunciante, no caso, o componente "Name Formatter". A partir de então, o serviço que era anunciado como "logSaved", passa a ser anunciado como "saved" e assim a notificação chegará ao componente "My Component". O mesmo processo ocorre na redefinição de apelidos de serviços. De uma forma padrão, todos os apelidos de serviços e eventos são iguais aos seus nomes. A redefinição de apelidos só é utilizada quando os nomes do lado do requisitante e do lado do provedor são diferentes.

Listagem de Código B.61: Exemplo de instanciação e configuração de componentes.

```
1 //Instanciando os componentes...
2 MyComponent myComponent = new MyComponent();
3 NameFormatter nameFormatter = new NameFormatter();
4
5 //Redefinindo o apelido do evento no componente anunciante...
6 nameFormatter.SetAliasOfAnnouncedEvent("logSaved" , "saved");
```

*2. Criando a hierarquia da aplicação*

Na Listagem de Código B.62, descreve-se a construção da hierarquia da aplicação de exemplo. Nas linhas 1, 2 e 3 são instanciados os contêineres. Observe que o contêiner raiz deve ser instanciado utilizando a classe `ScriptContainer` para que seja possível associá-lo a um *script* de execução posteriormente. Nas linhas 5, 6, 7 e 8 os componentes são adicionados aos contêineres "Printing" e "Formatting" e estes são adicionados ao contêiner "Root". Depois deste processo, a hierarquia da aplicação está pronta para ser executada, faltando apenas a definição de um *script* de execução para que a aplicação baseada na CMS esteja completa.

Listagem de Código B.62: Exemplo de construção de hierarquia da aplicação.

```
1 ScriptContainer root = new ScriptContainer("Root");
2 Container printing = new Container("Printing");
3 Container formatting = new Container("Formatting");
4
5 printing.addComponent(myComponent);
6 formatting.addComponent(nameFormatter);
7 root.addComponent(printing);
8 root.addComponent(formatting);
```

**Executando aplicações**

*1. Criando o* script *de execução*

Uma vez criada a hierarquia da aplicação, deve-se então definir o *script* de execução responsável por iniciar a execução da mesma. Quando os componentes funcionais da aplicação implementam todas as funcionalidades, inclusive a interface de interação com o usuário, pode-se criar um *script* de execução apenas instanciando a classe `ExecutionScript`, cujo construtor recebe como parâmetro uma instância de `ScriptContainer`. Para exemplificar a extensão de `ExecutionScript`, na Listagem de Código B.63 descreve-se a implementação da classe `MyScript`.

Listagem de Código B.63: Exemplo de criação de *script* de execução.

```
1  public class MyScript:ExecutionScript{
2      public MyScript(ScriptContainer _rootContainer):base(_rootContainer){}
3
4      //Código de inicialização da aplicação...
5      protected void main(){
6          Console.WriteLine("Nome formatado abaixo...");
7          ServiceRequest req = new ServiceRequest("print",new Object[]{"nome"});
8          try{
9              base.exec(req);
10         }catch(Exception e){
11             Console.WriteLine(e.StackTrace);
12         }
13         Console.WriteLine("Finalizado.");
14     }
15 }
```

O método `main` é executado na inicialização do *script* de execução e através dele é possível acessar a hierarquia de componentes utilizando os métodos `exec` e `execAsync`, para invocar

serviços, e `announceEvent`, para anunciar eventos. Neste exemplo, o serviço "print" é executado na linha 9. Notificações de eventos também podem ser recebidas sobrescrevendo o método `receiveEvent`.

*2. Iniciando a aplicação*

A inicialização da aplicação é realizada através da instanciação e inicialização do *script* de execução (Listagem de Código B.64). Para instanciar o *script*, deve-se definir o contêiner raiz da aplicação.

Depois, basta iniciar a aplicação utilizando o método `init` que primeiro iniciará o contêiner raiz (e conseqüentemente toda a hierarquia) e depois invocará o método main. A finalização do *script* pode ser realizada através do método `finish`, que requisita a interrupção da execução da hierarquia de componentes.

Listagem de Código B.64: Exemplo de criação de *script* de execução.

```
1 MyScript myScript = new MyScript(root);
2 myScript.init();
```

# Livros Grátis

( http://www.livrosgratis.com.br )

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo