INFRAESTRUTURA DE EXECUÇÃO DE PLANOS DE RECONFIGURAÇÃO DINÂMICA PARA O *FRAMEWORK* DE COMPONENTES IPOJO/OSGI

Fabio Lattario Fonseca

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Cláudia Maria Lima Werner

Rio de Janeiro
Março de 2013

INFRAESTRUTURA DE EXECUÇÃO DE PLANOS DE RECONFIGURAÇÃO DINÂMICA PARA O *FRAMEWORK* DE COMPONENTES IPOJO/OSGI

Fabio Lattario Fonseca

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof.ª Cláudia Maria Lima Werner, D.Sc.

Prof. Toacy Cavalcanti de Oliveira, D.Sc.

Prof. Renato Fontoura de Gusmão Cerqueira, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2013

Fonseca, Fabio Lattario

Infraestrutura de Execução de Planos de Reconfiguração Dinâmica para o *Framework* de Componentes iPOJO/OSGi/Fabio Lattario Fonseca. – Rio de Janeiro: UFRJ/COPPE, 2013.

XIV, 131 p.: il.; 29, 7cm.

Orientador: Cláudia Maria Lima Werner

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2013.

Referências Bibliográficas: p. 110 – 116.

1. *Framework* de Componentes. 2. Reconfiguração Dinâmica. 3. Sistemas Auto-Adaptáveis. I. Werner, Cláudia Maria Lima. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Aos meus pais, por seu infinito amor e suporte em todos os momentos da minha vida.

Agradecimentos

A seção de agradecimentos é algo realmente oportuno em uma dissertação de mestrado, pois muitos me ajudaram direta e indiretamente a chegar até aqui. Deixar estas palavras registradas eternamente junto com o meu trabalho é uma obrigação, uma honra, e uma singela homenagem, pois sem esta ajuda, este trabalho não existiria.

Gostaria de agradecer, primeiramente, à Cláudia, minha orientadora, que acreditou em mim enquanto aluno de tempo parcial e me deu a oportunidade de fazer este mestrado. Sua participação foi muito importante ao longo do curso, cobrando resultados no momento certo e me direcionando até a conclusão deste trabalho.

Este trabalho também não seria possível sem a valorosa ajuda do Marco, meu co-orientador não oficial. Obrigado pelas longas conversas, em almoços e reuniões, e pela paciência e persistência, mesmo quando eu teimava em não querer entender as coisas. E obrigado, também, pelas dicas eno-gastronômicas! Eu gostava muito desta parte das nossas conversas e espero que este assunto continue ativo.

Agradeço, também, ao Toacy e ao Renato, por terem aceitado participar da minha banca e pelos valorosos comentários que, certamente, elevarão o nível do trabalho.

Agradeço à minha família, sempre me apoiando e estando presente, principalmente nos momentos mais difíceis da minha vida. Sem vocês eu não teria chegado onde estou hoje.

Não posso deixar de agradecer, também, à minha família espiritual, aos meus amigos invisíveis, que estão sempre comigo, me apoiando e me guiando através dos desafios desta vida. Certamente tiveram um papel fundamental nesta jornada, me dando a paz e a tranquilidade para seguir em frente sempre.

Agradeço aos meus amigos de montanha, companheiros da UNICERJ, que me viram sumir temporariamente para conseguir finalizar o mestrado. Estou voltando às montanhas para novas e memoráveis excursões!

Gostaria de agradecer também aos meus colegas da GE, pelo apoio e pelas discussões conceituais sobre o tema deste trabalho, pelas palavras de incentivo de quem já trilhou este caminho, pelas cobranças que me ajudavam a me manter focado, pelas injeções de ânimo e por "nunca faltar tão pouco".

Por fim, gostaria de agradecer, especialmente, à minha querida esposa, companheira, amante e amiga Brenda. Sem você, eu não teria, sequer, começado o mestrado. Obrigado pela força, pelo apoio e por ter cuidado sozinha da nossa vida, nos vários momentos em que eu precisei me isolar e me concentrar para evoluir com os estudos. Sem dúvida, este mestrado também é seu!

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

INFRAESTRUTURA DE EXECUÇÃO DE PLANOS DE RECONFIGURAÇÃO DINÂMICA PARA O *FRAMEWORK* DE COMPONENTES IPOJO/OSGI

Fabio Lattario Fonseca

Março/2013

Orientador: Cláudia Maria Lima Werner

Programa: Engenharia de Sistemas e Computação

O aumento da complexidade de sistemas de software, sua crescente utilização como solução para os mais diversos problemas e o avanço da computação móvel têm criado novos desafios para a comunidade de Engenharia de Software. Em resposta a estes desafios, novas maneiras de se projetar, desenvolver e gerenciar sistemas de software tem sido investigadas. Neste contexto, sistemas Auto-Adaptáveis, que são sistemas capazes de alterar o seu comportamento em resposta a alterações no seu contexto de execução e em si próprios, têm se mostrado promissores como uma solução possível para estes novos problemas.

Dentre os vários problemas de pesquisa desta área, a execução da adaptação do sistema através de técnicas de reconfiguração dinâmica, que, neste trabalho, significa a alteração da sua estrutura arquitetural em tempo de execução, ainda permanece um desafio. Esta dissertação aborda a execução da reconfiguração dinâmica através de um plano de reconfiguração arquitetural, investigando este tema, e propondo e desenvolvendo uma infraestrutura para a execução deste plano, utilizando a plataforma OSGi, estendida pelo *framework* de componentes iPOJO. O trabalho é avaliado através de uma prova de conceito, onde alguns cenários de utilização ilustram as funcionalidades desenvolvidas.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

INFRASTRUCTURE FOR THE EXECUTION OF A DYNAMIC RECONFIGURATION PLAN TO THE IPOJO/OSGI COMPONENT FRAMEWORK

Fabio Lattario Fonseca

March/2013

Advisor: Cláudia Maria Lima Werner

Department: Systems Engineering and Computer Science

The increase in the complexity of software systems, its use as a solution for the most diverse and critical problems and the advances accomplished in the area of mobile computing, bring new challenges for the Software Engineering community. As an answer to these challenges, new tecniques for design, development and management of software systems are being investigated. In this context, Self-Adaptive systems, which are systems capable of altering their own behavior in response to changes in their execution context or in themselves, are being considered as a promising possible solution for these new problems.

Among the several research problems of this area, the adaptation execution using dynamic reconfiguration techniques, which, in this work, mean changing the architecture of the system in runtime, still remains an open problem. This dissertation addresses the dynamic reconfiguration as an execution of an architectural reconfiguration plan, investigating its issues, proposing and developing an infrastructure for its execution over the OSGi platform extended by the iPOJO component framework. This work is evaluated by a proof of concept, where the developed capabilities are ilustrated by utilization scenarios.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Com o passar do tempo e a crescente utilização da tecnologia como habilitador para a solução dos mais diversos problemas da humanidade, o software, devido à sua flexibilidade e possibilidade de aplicação em diferentes domínios, emerge como um importante elemento das soluções propostas.

Por este motivo sua complexidade vem aumentando consideravelmente. Este fato tem levado a comunidade de Engenharia de Software a investigar novas maneiras de projetar, desenvolver e gerenciar sistemas de software e seus serviços (DE LEMOS *et al.*, 2011).

Como consequência desta evolução, sistemas de software devem ser cada vez mais versáteis, flexíveis, robustos, eficientes, customizáveis, configuráveis e auto-otimizáveis, adaptando-se a mudanças no seu contexto operacional, ambiente ou requisitos. Com isso, Sistemas Auto-Adaptáveis, que são sistemas capazes de modificar seu comportamento em resposta a alterações no seu ambiente e no próprio sistema (CHENG *et al.*, 2009), tornaram-se um importante campo de pesquisa em franca evolução.

Uma linha de pesquisa que se beneficia destes avanços e que tem se mostrado bastante relevante atualmente é a Computação Ubíqua (WEISER, 1993), que expandiu a presença do software para virtualmente qualquer lugar sem que seja sequer notado. Este paradigma traz a ideia de que não existem barreiras na interação humano-computador, devendo o sistema ubíquo possuir uma maneira de monitorar o ambiente em que se encontra e responder a mudanças neste ambiente, interagindo com humanos ou outros sistemas de maneiras, muitas vezes, não previstas em tempo de projeto.

Software ubíquos não podem partir do princípio que o ambiente de execução é conhecido *à priori*. Este tipo de software deve ser concebido para lidar com diversos níveis de variabilidade. A heterogeneidade da sua infraestrutura de computação e comunicação, mobilidade, mudanças na disponibilidade de recursos e alterações na demanda de utilização requerem que estes sistemas de software sejam adaptáveis ao

seu contexto de execução (INVERARDI e TIVOLI, 2009).

No entanto, diversos tipos de sistemas de software possuem características semelhantes e também vêm sendo beneficiados pelos avanços na área de Sistemas Auto-Adaptáveis. Segundo CHENG *et al.* (2009),

> É importante enfatizar que em muitas iniciativas que exploram o comportamento auto adaptável, o elemento comum que habilita a auto-adaptação é o software. Isso se aplica à pesquisa em várias áreas de aplicação e tecnologia como interfaces com o usuário adaptáveis, computação autonômica, *dependable computing*, sistemas embarcados, redes móveis e *ad-hoc*, sistemas multi-agentes, redes de sensores, arquiteturas orientadas à serviço e computação ubíqua.

Uma aplicação do software em particular que traz alguns dos mais complexos desafios da área são os sistemas utilizados em aplicações críticas que precisam estar em execução 100% do tempo e não podem apresentar falhas. A evolução destes sistemas em termos da adição de novas funcionalidades ou até mesmo de correção de falhas deve ser realizada com o sistema em execução, enquanto está atendendo requisições. Para estes sistemas, determinadas técnicas devem garantir a sua manutenção e evolução (OREIZY, 1996) sem que seja necessária uma janela de *downtime*.

A área de sistemas auto-adaptáveis vem, portanto, recebendo atenção da comunidade de engenharia de software ao longo dos últimos anos. No entanto, a maioria dos trabalhos encontrados na área focam na decisão do "que" adaptar e "quando" adaptar, propondo soluções para o processo de adaptação, deixando de olhar para o "como" adaptar. A realização apropriada da auto-adaptação ainda é um desafio nos dias de hoje (CHENG *et al.*, 2009).

O foco deste trabalho está justamente no desafio de obter uma maneira apropriada para a execução da adaptação em si, ou seja, no "como" adaptar. Nas próximas seções, será apresentado um resumo do estudo preliminar realizado com o objetivo de estabelecer o escopo da pesquisa e, em seguida, este escopo será definido.

1.1 Sistemas Auto-Adaptáveis

Sistemas de software auto-adaptáveis possuem a habilidade de modificar seu comportamento em resposta a alterações no seu ambiente e no próprio sistema (CHENG *et al.*, 2009), sendo capazes de lidar com um ambiente em constante evolução e com requisitos que podem não ser conhecidos em tempo de projeto. Em (LADDAGA *et al.*, 2003), o autor argumenta que a principal razão a favor da utilização de sistemas auto-adaptáveis é o aumento dos custos de se lidar com a complexidade inerente aos sistemas de software no atendimento aos seus requisitos.

Para que um sistema auto-adaptável possa adequar o seu comportamento à mudanças ocorridas em si mesmo e no seu ambiente de execução, ele precisa possuir a capacidade de monitorar algumas variáveis de interesse. Esta capacidade chama-se sensibilidade ao contexto, onde contexto é qualquer informação que pode ser usada para caracterizar a situação de uma entidade. Uma entidade pode ser uma pessoa, um lugar ou um objeto considerado relevante para a interação entre um usuário e uma aplicação, incluindo o próprio usuário e a aplicação (DEY *et al.*, 2001).

Ao longo do tempo, muitos estudos foram publicados na área de sistemas auto-adaptáveis com o objetivo de classificar, segmentar e caracterizar esta classe de sistemas de software buscando, em especial, entender os seus requisitos.

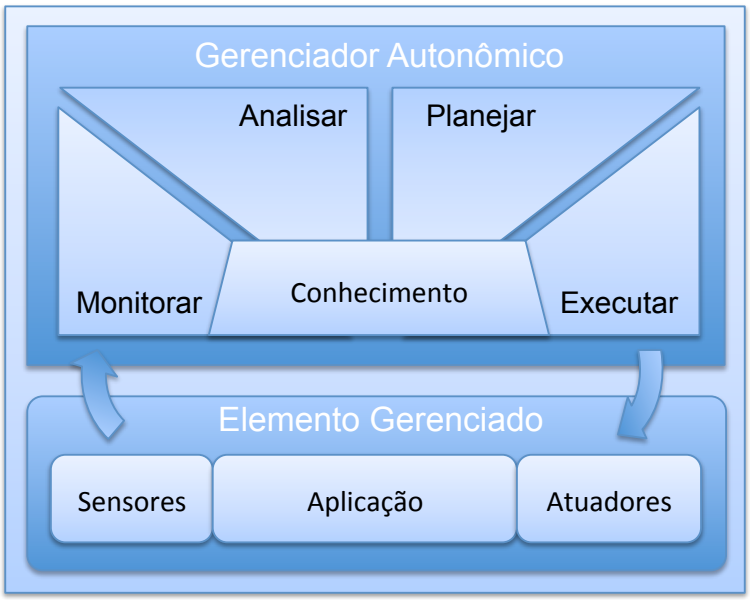

Figura 1.1: Ciclo MAPE-K (KEPHART e CHESS, 2003)

Em termos de modelagem da adaptação, na área de Computação Autonômica[1], existe o conceito da adaptação ser produzida por um processo cíclico de realimentação em uma malha fechada, inspirado da Teoria de Controle (OGATA, 1997). Este ciclo é chamado de MAPE-K e a sua representação conceitual pode ser vista na Figura 1.1. Como pode ser visto, o ciclo MAPE-K divide o processo de adaptação em 4 fases:

- Monitorar: a fase de monitoramento tem a responsabilidade de adquirir através dos sensores da aplicação e de mecanismos de monitoramento do ambiente de execução, as informações de contexto necessárias para a decisão da adaptação. Estas informações podem ser quantitativas, geralmente associadas à

[1]A área de Computação Autonômica, termo cunhado pela IBM (KEPHART e CHESS, 2003) possui vários pontos em comum com a área de Software Auto-Adaptável (SALEHIE e TAHVILDARI, 2009) e, neste trabalho, os dois termos serão utilizados indistintamente.

parâmetros de QoS ou qualitativas, associadas a quão bem os requisitos da aplicação estão sendo atendidos com a configuração corrente da aplicação;

- Analisar: a análise consiste em verificar se no contexto atual de execução, a configuração corrente é a que melhor atende os requisitos da aplicação. É a fase de decisão da adaptação. Neste ponto, são tomadas duas decisões: se a aplicação será adaptada e qual é a configuração ideal da aplicação dado o seu contexto de execução.

- Planejar: uma vez que seja decidido que uma adaptação será realizada, a fase de planejamento tem o objetivo de criar um plano de adaptação. Um plano de adaptação é uma sequência de ações de alteração arquitetural, com o objetivo de levar o sistema de uma determinada configuração arquitetural até a configuração final, definida na fase de Análise.

- Executar: por fim, uma vez com o plano de adaptação definido, é necessário realizar de fato a reconfiguração da aplicação, por meio dos atuadores. Esta fase lida com os aspectos técnicos da reconfiguração arquitetural.

Além disso, o elemento central que viabiliza estas fases é uma base de conhecimento através da qual a adaptação será decidida.

Vários trabalhos foram publicados propondo abordagens para tratar todo o ciclo de adaptação ou parte dele (ALIA *et al.*, 2007; FLOCH *et al.*, 2006; OREIZY *et al.*, 1999). No entanto, na área de sistemas auto-adaptáveis, pouco foco se dá à fase de execução da adaptação em si. As soluções apresentadas são restritas a domínios específicos ou não detalham os aspectos da execução do plano de reconfiguração arquitetural. Além disso, muitas vezes, utilizam abordagens dificilmente reutilizáveis. Por este motivo, este trabalho foca na execução da adaptação em tempo de execução (atividade também conhecida como Reconfiguração Dinâmica), buscando chegar a uma plataforma sobre a qual sistemas auto-adaptáveis possam ser desenvolvidos.

São muitos os desafios da área de Reconfiguração Dinâmica: como reconfigurar a aplicação em tempo de execução sem ser necessário reiniciá-la; tratamento do erro de reconfiguração; manutenção do estado de execução da aplicação antes e depois da reconfiguração, ou seja, não se pode perder requisições sendo executadas; entre outros.

De maneira geral, pode-se dizer que existem duas abordagens para a obtenção do comportamento adaptável (MCKINLEY *et al.*, 2004). Uma adaptação por parâmetro modifica variáveis internas da aplicação de forma a mudar o seu comportamento. Um exemplo muito citado é o protocolo TCP, que altera parâmetros relacionados à transmissão e quantidade de re-tentativas de envio em resposta ao aparente congestionamento da rede. Porém adaptações por parâmetro possuem a limitação de não

possibilitarem a inserção de novos algoritmos e componentes à aplicação após o seu projeto e construção iniciais.

Por sua vez, a outra abordagem conhecida por adaptação composicional (MCKINLEY *et al.*, 2004) altera a estrutura da aplicação, trocando algoritmos e componentes por outros que sejam mais adequados ao ambiente de execução corrente. Deste modo, uma aplicação pode ser alterada para atender à requisitos não previstos em tempo de projeto. Esta abordagem suporta a reconfiguração dinâmica da aplicação em tempo de execução para, por exemplo, adicionar e remover componentes em um dispositivo com memória limitada ou adicionar novos comportamentos em sistemas já implantados. Por outro lado, esta abordagem requer um controle maior do mecanismo de adaptação, além de aumentar a complexidade do software adaptável, oferecendo um campo fértil para pesquisas.

Outro aspecto que deve estar claro no projeto de um sistema auto-adaptável é em que nível de abstração o comportamento adaptável será obtido. Em (OREIZY *et al.*, 1999), os autores propõem um espectro da adaptação que busca dar uma noção comparativa entre diversas abordagens para obtenção da adaptação. A Figura 1.2 mostra este espectro.

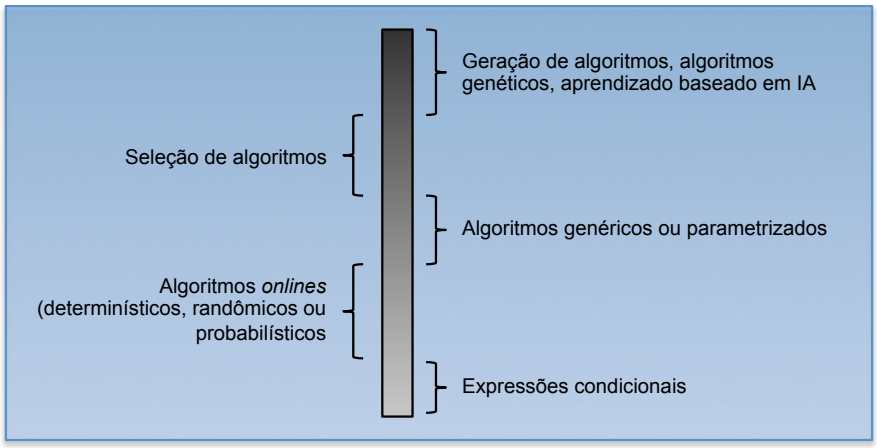

Figura 1.2: Espectro da adaptação (OREIZY *et al.*, 1999)

Na base do espectro, conforme o autor, expressões condicionais podem ser consideradas uma forma de adaptação, pois de acordo com o valor de determinada variável a aplicação determina que comportamento adotar.

Na outra extremidade, abordagens baseadas em Inteligência Artificial também são formas de se obter o comportamento auto-adaptável, já que utilizam propriedades do ambiente de execução e conhecimento obtido em tentativas anteriores de geração de novos algoritmos para alterar o comportamento da aplicação.

De maneira geral, abordagens próximas ao topo do espectro têm a capacidade de separar melhor a lógica responsável por produzir a adaptação da lógica das funcionalidades da adaptação em si. Esta característica traz uma vantagem muito im-

portante a medida que facilita a manutenção e evolução da aplicação. Além disso, como a pesquisa descrita nesta dissertação foca na adaptação composicional, o interesse maior é nas técnicas próximas ao topo do espectro, que viabilizem a alteração estrutural da aplicação.

Um conceito muito útil relacionado à técnicas no topo do espectro é a Arquitetura de Software. A arquitetura de um sistema é a ponte de ligação entre os seus Requisitos e o Código que o implementa. É nela que estão concentradas as decisões centrais que determinam a estrutura de um sistema, incluindo como o sistema é composto de partes que interagem entre si (GARLAN, 2000).

Neste sentido, KRAMER e MAGEE (2007) argumentam que a utilização do nível arquitetural, onde a aplicação é vista como um conjunto de componentes e conectores, traz diversas vantagens para um sistema auto-adaptável, entre elas:

- Abstração: a arquitetura de um sistema fornece um nível de abstração mais favorável para descrever as alterações dinâmicas de um sistema através da recombinação de seus componentes, conectores e ligação entre eles do que o nível do algoritmo, mais próximo do código fonte;

- Generalização: os conceitos arquiteturais e a solução para a obtenção da adaptação enquanto uma reorganização dos componentes e conectores são generalizáveis a diferentes domínios;

- Potencial para escalabilidade: arquiteturas de software geralmente suportam a composição hierárquica de componentes, o que facilita a integração entre sistemas e a criação de sistemas maiores que englobem outros sistemas menores, viabilizando a sua utilização em aplicações complexas que demandam grandes sistemas.

Além disso, através da composição de componentes já existentes ou customizados especificamente para a ocasião, um sistema pode ser construído tão rápido e de maneira tão eficiente quanto um carro pode ser montado a partir de suas partes (SZYPERSKI, 2002).

A separação da lógica de adaptação da lógica de negócio da aplicação é uma característica desejável. Neste sentido, a lógica da adaptação de um sistema auto-adaptável pode ser classificada como interna ou externa (SALEHIE e TAHVILDARI, 2009), como mostra a Figura 1.3. Uma abordagem externa, neste caso, não pressupõe necessariamente, uma aplicação separada. Significa, apenas, que a lógica da adaptação (motor da adaptação – *adaptation engine*) está completamente separada da lógica da aplicação (software adaptável – *adaptable software*). Já uma abordagem interna mistura estas duas partes, dificultando a manutenção e tornando o sistema menos escalável.

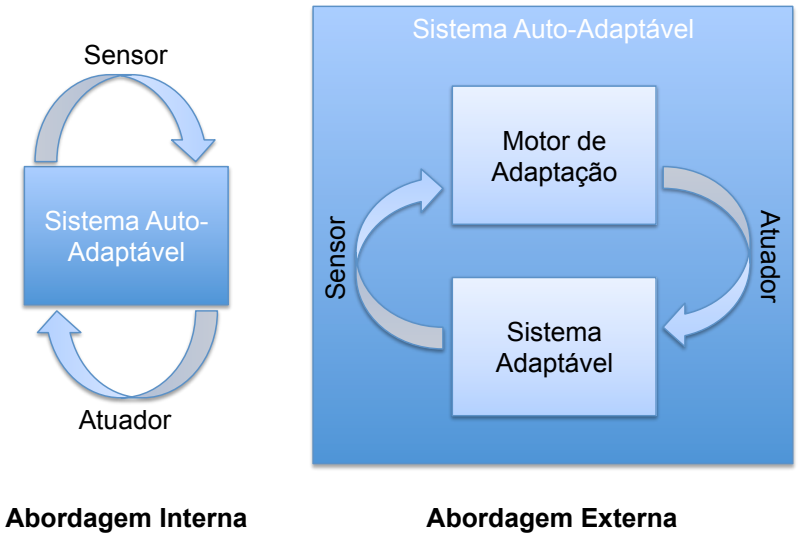

Figura 1.3: Adaptação interna/externa

Observando a figura é possível traçar um paralelo direto da adaptação externa com o ciclo MAPE-K, onde o Gerenciador Autonômico faz o papel do motor de adaptação e o Elemento Gerenciado faz o papel da aplicação adaptável.

A adaptação externa pode ser conseguida de diversas maneiras. MCKINLEY *et al.* (2004) argumentam que um local apropriado para implementação da lógica da adaptação seria um middleware. Um middleware é uma camada intermediária que separa a lógica da aplicação das camadas de mais baixo nível como, por exemplo, a rede e o sistema operacional. Em (HALLSTEINSEN *et al.*, 2005), esta abordagem é utilizada para a criação de aplicações a serem utilizadas em dispositivos móveis.

Outro conceito muito importante para a adaptação de um sistema em tempo de execução e principal habilitador da Reconfiguração Dinâmica, é a Reflexão Computacional. Segundo MAES (1987), um sistema reflexivo é aquele que incorpora estruturas que representam aspectos dele mesmo. Esta auto representação faz com que o sistema seja capaz de responder perguntas e suportar ações sobre ele mesmo, característica imprescindível a um sistema auto adaptável (MCKINLEY *et al.*, 2004).

Na Figura 1.4, é possível ver as principais tecnologias e técnicas habilitadoras da adaptação composicional sob a ótica de MCKINLEY *et al.* (2004).

Este trabalho visa prover uma plataforma sobre a qual a Reconfiguração Dinâmica possa ser executada com sucesso, considerando os aspectos aqui descritos de um sistema auto-adaptável.

1.2 OSGi como plataforma base para a adaptação

Conforme já colocado, uma vez que o nível adequado para a promoção do comportamento adaptável em uma aplicação é o nível arquitetural, é importante que a

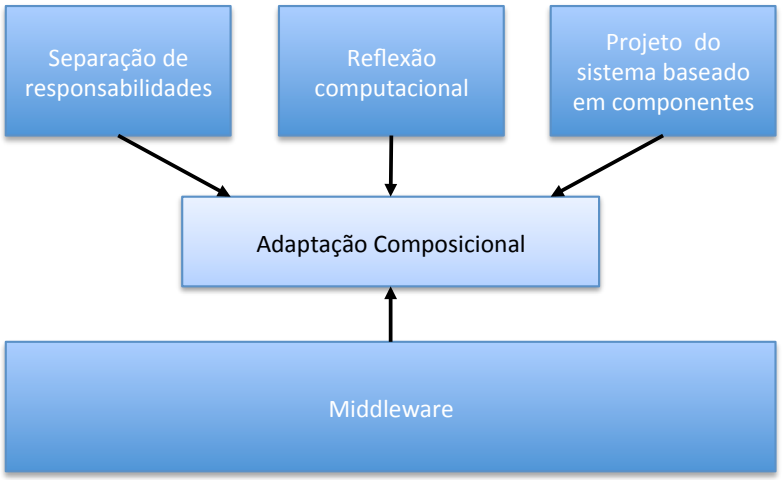

Figura 1.4: Tecnologias e técnicas habilitadoras da adaptação composicional (MCKINLEY *et al.*, 2004)

plataforma utilizada suporte este nível de abstração. No nível arquitetural, onde a aplicação é vista como um conjunto de componentes e conectores, a reconfiguração dinâmica consiste em alterar a topologia da aplicação, ou seja, alterar as ligações entre os componentes através dos conectores em tempo de execução.

De acordo com as definições provenientes da área de Desenvolvimento Baseado em Componentes (SZYPERSKI, 2002), uma configuração arquitetural é uma determinada topologia de um sistema, descrita por meio de componentes e conectores (TAYLOR *et al.*, 2009). Esta configuração arquitetural obedece a um determinado modelo de componentes, que define as suas regras de formação. Logo, é preciso que a plataforma utilizada seja compatível com estes conceitos.

De acordo com a divisão proposta por MCKINLEY *et al.* (2004), uma das tecnologias habilitadoras da adaptação composicional é o *middleware*. Sendo assim, para esta pesquisa, é importante a escolha de uma plataforma/*middleware* robusta e capaz de suprir as necessidades de reflexão de um sistema adaptável.

A plataforma escolhida foi o OSGi[2], devido ao seu alto nível de maturidade e sua larga utilização em diversas iniciativas tanto na academia quanto na indústria. O OSGi nasceu como um *framework* de modularização para a linguagem Java (HALL *et al.*, 2011). Ele estende a linguagem Java além dos limites de modularização definidos pela Orientação à Objetos e cria o conceito de *Bundle*[3], que é o módulo OSGi. Como uma continuação dos esforços para definir um mecanismo de modularização, o OSGi define um modelo de programação orientado a serviços, oferecendo um ambiente dinâmico propício para a obtenção do comportamento adaptável.

Inicialmente, o foco principal do OSGi estava em automação residencial. Suas primeiras especificações (que começaram a ser escritas em 1998) tinham o objetivo

[2]http://www.osgi.org
[3]ver Seção 2.3.1

de resolver o problema de como construir aplicações a partir de dispositivos independentes[4]. Com o tempo, o foco foi mudando, já que os principais problemas em sistemas de software estavam em como integrar artefatos que não foram projetados para trabalharem em conjunto. Ao longo do tempo, então, as especificações do OSGi foram mudando para proverem a solução para estes problemas de maneira genérica e não mais focando no campo de automação residencial. Atualmente, a especificação do OSGi define um sistema de módulos dinâmicos, utilizado desde sistemas complexos como IDEs e servidores de aplicações até aplicações executadas em dispositivos móveis.

A especificação OSGi, hoje em dia, está na sua versão R5[5] (lançada em Junho de 2012). A sua primeira versão (R1) data de 2000 e começou a ser escrita em 1998. Durante estes mais de 12 anos, o OSGi foi amplamente utilizado, testado e estendido em diversos ambientes. O tempo de existência também garante uma comunidade grande e ativa para cada uma das suas implementações.

Em termos de utilização o OSGi pode ser encontrado em uma ampla gama de aplicações. Desde aplicações complexas como a IDE Eclipse[6] e servidores de aplicação como IBM Websphere[7], Oracle Weblogic[8] e JBoss[9], até pequenas diversas iniciativas. Na página da OSGi Alliance[10], é possível navegar pelos projetos dos diferentes mercados onde o OSGi é utilizado.

Com o alto nível de popularidade do OSGi, uma solução que possibilite a execução da adaptação de sistemas adaptáveis construídos sobre esta plataforma poderá ter um grande alcance e utilização.

No entanto, apesar da sua maturidade e abrangência mercadológica, falta ao OSGi um modelo de componentes que seja hierárquico e ao mesmo tempo suporte Reconfiguração Dinâmica.

Alguns trabalhos tentam suprir esta lacuna do OSGi. O que mais se destaca é o iPOJO (ESCOFFIER e HALL, 2007), que pode ser definido como um *framework* de injeção de dependências criado sobre o OSGi. O iPOJO define um modelo de componentes hierárquico, porém com uma capacidade de reconfiguração dinâmica muito limitada, permitindo apenas a troca de componentes equivalentes em cenários específicos. Apesar de não existir um consenso a respeito de quais seriam os cenários a serem atendidos por um *framework* que suporte reconfiguração dinâmica na sua totalidade, para esta pesquisa, as ações de reconfiguração a serem consideradas estão

[4]http://www.osgi.org/Technology/WhyOSGi

[5]Este trabalho, no entanto, utiliza a versão R4.2, disponível no início do desenvolvimento.

[6]http://eclipse.org/osgi/

[7]http://www.ibm.com/developerworks/websphere/techjournal/1007_robinson/1007_robinson.html

[8]http://www.oracle.com/technetwork/articles/oraclesoa-evs-082081.html

[9]http://www.jboss.org/jbossas/osgi

[10]http://www.osgi.org/Markets/HomePage

detalhadas no Capítulo 4 (Seção 4.2.2). Como exemplo, podemos citar a adição e remoção de componentes e a alteração da ligação entre componentes. Atualmente, o OSGi não as suporta na sua totalidade.

Outras iniciativas visam a criação de um *framework* de reconfiguração dinâmica para diferentes modelos de componentes. Uma delas é o FraSCAti (SEINTURIER *et al.*, 2012). Esta iniciativa visa implementar a especificação SCA no framework Fractal. O resultado foi um interessante trabalho que alia as características SOA do SCA com a capacidade de reconfiguração dinâmica do Fractal. Além disso, existe uma série de ferramentas que foram construídas sobre a API do Fractal para facilitar a reconfiguração dinâmica, que podem ser utilizadas no FraSCAti. Porém, se por um lado o FraSCAti se mostra uma solução interessante, por outro não tem a base instalada e o nível de utilização do OSGi, além de ser um trabalho ainda em nível de encubação.

1.3 Definição do escopo e objetivos do trabalho

Neste ponto, já é possível estabelecer o escopo deste trabalho. A Figura 1.5 mostra um resumo do que foi descrito nas seções anteriores.

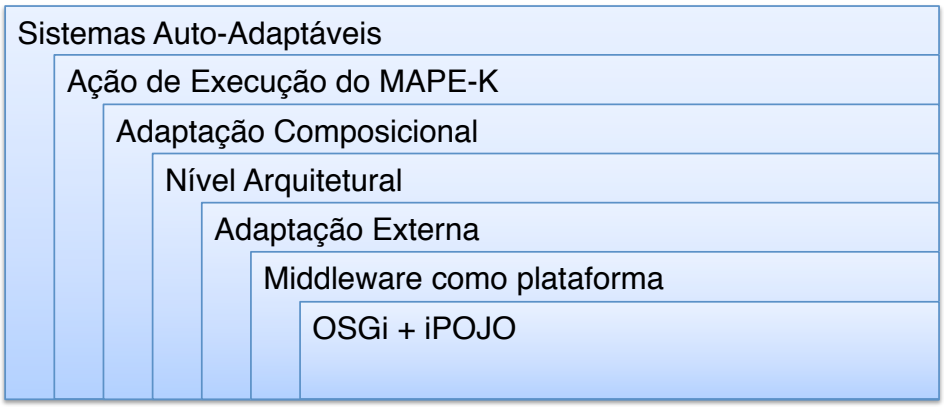

Figura 1.5: Foco de atuação desta pesquisa

Como colocado anteriormente, não foi encontrado nas pesquisas exploratórias iniciais um *framework* que ofereça capacidades de Reconfiguração Dinâmica para o OSGi. Sendo assim, para que a execução de uma adaptação composicional no nível arquitetural possa acontecer de fato em um sistema adaptável, é necessário criar esta extensão. Logo, podemos estabelecer o primeiro objetivo:

O1: Pesquisa e desenvolvimento de uma extensão que habilite a Reconfiguração Dinâmica no nível arquitetural no OSGi.

Este *framework* aproveitará as capacidades atuais do iPOJO e o estenderá para

que este suporte ações de reconfiguração dinâmica provenientes de um plano de reconfiguração arquitetural determinado na ação "Planejar" do ciclo MAPE-K.

Além disso, a execução de um plano de reconfiguração arquitetural em sistemas dinâmicos como os criados com o OSGi/iPOJO oferece diversos desafios que devem ser pesquisados e considerados no mecanismo de execução do plano como, por exemplo, a possibilidade de erro de execução. Logo, o segundo objetivo pode ser estabelecido como:

O2: Pesquisa das questões relacionadas à execução de um plano de reconfiguração arquitetural em um sistema dinâmico e desenvolvimento de um mecanismo para tal objetivo.

O presente trabalho visa atingir estes dois objetivos, criando uma infraestrutura de execução de planos de reconfiguração dinâmica para o *framework* de componentes iPOJO, executado sobre a plataforma OSGi.

1.4 Organização da dissertação

Esta dissertação está dividida da seguinte maneira:

Capitulo 1: Introdução: Este capítulo introduziu o tema de pesquisa, além de mostrar também o resultado da pesquisa exploratória inicial que mirou em aspectos da adaptação e problemas em aberto. Introduziu, também, o OSGi e justificou a sua utilização, além de formular os objetivos que devem ser atingidos ao final do trabalho.

Capítulo 2: Fundamentação Teórica: Neste capítulo, são discutidos os aspectos teóricos por trás deste trabalho. Começa descrevendo os conceitos de Arquitetura de Software utilizados neste trabalho, citando os seus aspectos relacionados à Reconfiguração Dinâmica. Em seguida, detalha o funcionamento do OSGi e, finalmente, do iPOJO, mostrando suas estruturas internas em cima das quais a extensão proposta neste trabalho se apoia.

Capítulo 3: Trabalhos Relacionados: Neste capítulo, são reunidos alguns trabalhos relacionados ao tema desta dissertação, ressaltando seus pontos fortes e fracos.

Capítulo 4: Um Mecanismo para a Execução de um Plano de Reconfiguração Arquitetural para iPOJO/OSGi: Neste capítulo, é detalhada a pesquisa realizada neste trabalho. É realizada uma discussão conceitual explicitando os principais desafios encontrados na definição da extensão para o iPOJO e do mecanismo

de execução do plano de reconfiguração proposto, estabelecendo os seus requisitos. Por fim, são providos os detalhes de implementação da extensão e do mecanismo citados.

Capítulo 5: Avaliação: Neste capítulo, é realizada uma discussão a respeito da validade da solução proposta. A abordagem utilizada para a avaliação das contribuições deste trabalho é a definição de uma Prova de Conceito contendo cenários que demonstrem os resultados obtidos.

Capítulo 6: Conclusão: Neste capítulo, é feita a conclusão do trabalho com um resumo dos resultados que foram atingidos, das limitações deste trabalho, além de propor uma lista de trabalhos futuros.

Apêndice A: Detalhamento COMANCHE - Código Java e Configuração dos Componentes iPOJO: Este apêndice inclui o código do Comanche adaptado para o iPOJO.

Apêndice B: Plano de Iniciação do Comanche: Neste apêndice, o plano de reconfiguração utilizado para iniciar o Comanche é incluído.

Capítulo 2

Fundamentação Teórica

Este capítulo visa definir alguns conceitos que foram utilizados neste trabalho, necessários para que o leitor tenha um pleno entendimento das contribuições e suas implicações.

O primeiro assunto a ser discutido é Arquitetura de Software, na Seção 2.1. São abordados conceitos de componentes, conectores, configuração arquitetural, modelo de componentes, entre outros. Também são abordados alguns aspectos de reconfiguração dinâmica.

Na Seção 2.2, alguns conceitos de Planejamento Automatizado são introduzidos. Esta disciplina da área de Inteligência Artificial é utilizada para a obtenção de um plano de reconfiguração arquitetural a ser executado no sistema adaptável, dentro de uma estratégia de adaptação de mais alto nível, na qual o trabalho descrito nesta dissertação está inserido.

Como este trabalho lida, fundamentalmente, com desafios de uma implementação, são abordados, na Seção 2.3, conceitos específicos do *middleware* utilizado, que é o OSGi. É discutida a maneira como os *bundles*, entidades que implementam o conceito de modularização do OSGi, funcionam. Em seguida, como eles se comportam do ponto de vista de ciclo de vida, dando enfoque ao seu comportamento dinâmico. Por fim, é abordado o conceito de serviço estabelecido pelo OSGi.

Finalmente, na Seção 2.4, é abordado o *framework* de componentes iPOJO, que estende o OSGi, definindo um modelo de componentes hierárquico sobre a sua infraestrutura de serviços. É discutido também o conceito de *handler* e a sua capacidade de extensibilidade, sobre a qual uma das contribuições deste trabalho se apoia.

2.1 Arquitetura de Software

A presente pesquisa apresenta um mecanismo para a execução de um plano de reconfiguração no nível arquitetural. É importante ressaltar que o nível arquitetural abrange um amplo espectro de abstração. Portanto, esta seção tem o objetivo de

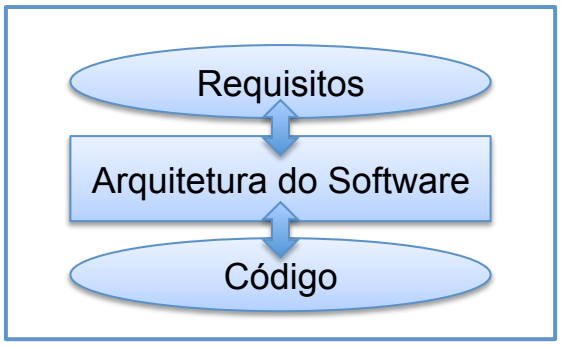

Figura 2.1: Nível arquitetural como elemento de ligação entre os requisitos de um sistema e sua implementação

definir os principais conceitos utilizados neste trabalho a respeito de arquitetura de software, com os quais o plano de reconfiguração será executado.

Com o passar do tempo e com o aumento da complexidade dos sistemas de software, o desenho e a especificação da estrutura de sistemas compelxos foi ganhando uma importância maior do que a escolha de seus algoritmos e estruturas de dados (SHAW e GARLAN, 1996). Pensar em termos de estrutura do sistema significa pensar em componentes, composição de componentes, comunicação entre componentes, acesso à dados, protocolos, escalabilidade e desempenho. Estas questões estão ligadas à um nível mais abstrato do que o código da aplicação. Este nível é chamado de Arquitetura do Software, ou nível arquitetural.

O nível de abstração de uma arquitetura de software pode variar bastante. O nível arquitetural pode descrever um sistema como um conjunto de componentes hierárquicos, onde um sistema inteiro pode se tornar um componente (composto) em um sistema maior. Além disso, a arquitetura de um sistema exerce um papel fundamental de ligação entre o seu requisito e o seu código (GARLAN, 2000) (Figura 2.1).

O principal conceito relacionado à arquitetura de um sistema é o componente. O conceito de componente possui muitas definições e uma das mais aceitas é proveniente da área de Engenharia de Software Baseada em Componentes (ou CBSE – *Component Based Software Engineering*):

> Um componente de software é uma unidade de composição com interfaces especificadas por contratos e dependências explícitas. Um componente de software pode ser implantado de maneira independente e é passível de composição por outras entidades. (SZYPERSKI, 2002)

Segundo VANDEWOUDE (2007), esta definição engloba algumas importantes características dos componentes, tais como:

- Princípio "caixa-preta": um componente é uma estrutura auto contida que esconde a sua implementação interna da sua camada pública de acesso. Com

isso podem ser utilizados sem que seja necessário saber detalhes da sua implementação.

- Passível de composição por outras entidades: a intenção da CBSE é utilizar componentes como blocos de construção de sistemas, diminuindo o tempo necessário para construir uma aplicação. A composição de componentes deve necessitar do mínimo possível de código intermediário, mas sem quebrar o princípio da caixa-preta.

- Interfaces baseadas em contrato: Seguindo o princípio da caixa-preta e da possibilidade de composição, por consequência, é necessário que um componente especifique claramente o que é capaz de fazer e as suas dependências em relação ao ambiente externo. Isto é conseguido através da utilização de um contrato bem definido.

O trabalho de pesquisa tema desta dissertação tem uma forte ligação com aspectos técnicos e de implementação. Por este motivo, é necessário separar dois conceitos que muitas vezes são confundidos: componente e instância de componente. Uma instância de componente está para um componente assim como um objeto está para uma classe, em Orientação a Objetos. Isto nos leva à definição de componente e instância de componente que serão utilizados neste trabalho:

Definição 1 (Componente): Um componente é uma entidade lógica abstrata que encapsula a implementação de uma funcionalidade da aplicação, escondendo o seu código interno. Sua utilização se dá por meio de uma interface pública definida na forma de um contrato que explicita suas dependências externas. Além disso, um componente deve suportar a composição hierárquica por entidades externas.

Definição 2 (Instância de componente): Uma instância de componente é uma materialização de um componente em um *framework* de componentes, contendo um estado de execução.

Um componente pode dar origem à várias instâncias, tantas quantas forem necessárias dependendo do cenário de execução. Estas instâncias devem ser executadas sobre uma estrutura de software que a suporte, com regras específicas para instanciação de componentes e a troca de mensagens entre as instâncias, através de suas interfaces. Estas regras são definidas por um modelo de componentes e implementadas por um *framework* de componentes:

Definição 3 (Modelo de Componentes): Um modelo de componentes define regras para a criação de componentes que devem ser obedecidas pelos desen-

volvedores. Provê, ainda, a semântica da execução e padrões de comunicação entre componentes (VANDEWOUDE, 2007).

Definição 4 (*Framework* de Componentes): É um conjunto de interfaces e regras de interação que governam como os componentes que são "plugados" no *framework* devem interagir. Tipicamente, *frameworks* de componentes também proveem uma implementação destas regras (**?**).

Segundo VANDEWOUDE (2007), um *framework* de componentes é muitas vezes visto como um mini sistema operacional: componentes se relacionam com o *framework* da mesma maneira que processos se relacionam com o sistema operacional em uma abordagem centrada em *middleware*.

Na pesquisa descrita nesta dissertação, um conceito muito importante e bastante utilizado é o de configuração arquitetural:

Definição 5 (Configuração Arquitetural): A configuração arquitetural de um sistema é a topologia de interconexão das suas instâncias de componentes. Ela define o sistema em termos de um conjunto de componentes ligados entre si de maneira hierárquica (ou não), com o objetivo de atender aos seus requisitos.

A partir de uma configuração arquitetural é fácil definir um diagrama visual do sistema como, por exemplo, o ilustrado na Figura 2.2, que mostra algumas características adicionais dos componentes como as interfaces requeridas e providas.

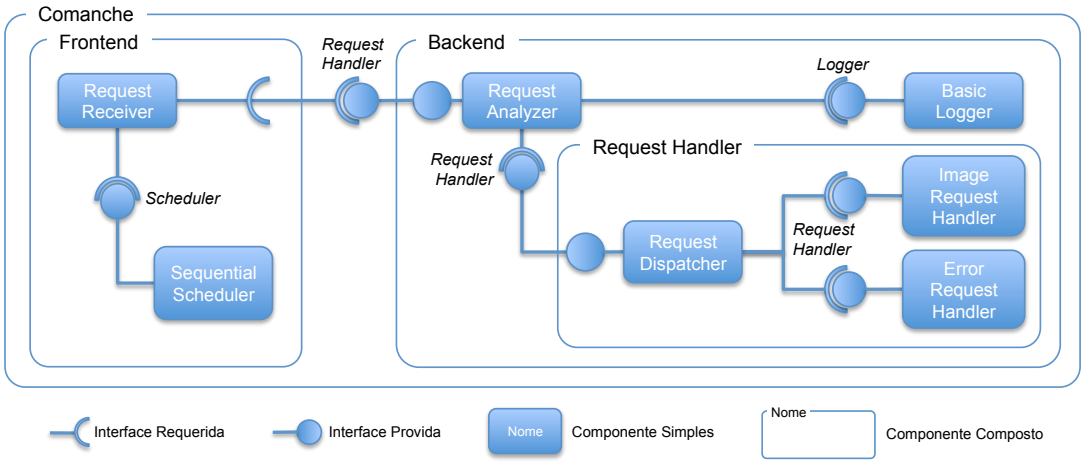

Figura 2.2: Modelo da arquitetura de um servidor *web* simples chamado de Comanche

2.1.1 Aspectos da Reconfiguração Dinâmica

Esta dissertação utiliza a Reconfiguração Dinâmica como habilitador para a execução da adaptação. Por este motivo, alguns aspectos devem ser abordados para o correto entendimento das contribuições deste trabalho.

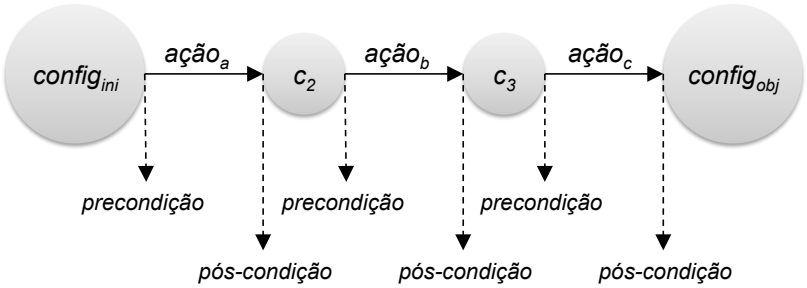

Figura 2.3: Execução de uma reconfiguração, ressaltando a verificação das precondições e pós-condições

Segundo HOFMEISTER (1993), a reconfiguração dinâmica de um sistema é definida como a sua alteração em tempo de execução. Esta ação é necessária, por exemplo, para realizar alterações em um sistema que deve permanecer continuamente disponível. Embora uma alteração em tempo de execução em um sistema seja algo muito amplo, para a pesquisa desta dissertação, a reconfiguração dinâmica é considerada no nível arquitetural, onde alterações são feitas em cima das entidades descritas na Seção 2.1.

Em (LÉGER *et al.*, 2010), os autores definem uma reconfiguração dinâmica como uma transação composta por ações de reconfigurações arquiteturais. Este modelo de transações suporta tolerância à falhas, para manter a consistência arquitetural, e execução distribuída da reconfiguração.

Para suportar tolerância à falhas, uma ação de reconfiguração deve contar com um conjunto de precondições e pós-condições. A execução de uma reconfiguração dinâmica envolve a validação das precondições e das pós-condições de cada ação conforme sua execução, como mostra a Figura 2.3. Estas condições determinam se aquela ação é aplicável ao sistema, dado o seu estado (configuração arquitetural) corrente, além de definir critérios de sucesso para a execução.

Uma transação de reconfiguração deve possuir as propriedades ACID - *Atomicity, Consistency, Isolation, Durability* (Atomicidade, Consistência, Isolamento e Durabilidade). O conceito geral destas propriedades é:

Atomicidade Em uma transação, ou a reconfiguração é executada com sucesso por completo, ou ela aborta e o sistema volta ao estado inicial através de um mecanismo de compensação;

Consistência Uma reconfiguração deve levar o sistema de um estado consistente para outro estado consistente, de acordo com o critério de definição de consistência do estado do sistema;

Isolamento Reconfigurações são executadas independentemente. O resultado de uma reconfiguração que ainda não sofreu *commit* não é visível até que isto

aconteça. Os autores propõem um mecanismo para impedir o acesso à aplicação enquanto a reconfiguração não é finalizada; e

Durabilidade Uma vez que a reconfiguração é executada com sucesso, seus efeitos são permanentes. O novo estado do sistema é persistido para o caso de necessidade de recuperação em caso de falhas graves como, por exemplo, falhas de *hardware*.

2.2 Planejamento Automatizado

Segundo GHALLAB *et al.* (2004), "planejamento é o lado intelectual da ação". É um processo de deliberação abstrato e explícito que escolhe e organiza ações através da antecipação dos resultados esperados. Esta deliberação tem em vista atingir da melhor forma possível os objetivos pré-especificados. Planejamento Automatizado é uma área de Inteligência Artificial que estuda este processo de deliberação computacionalmente.

Planejar está relacionado a escolher e organizar ações para alterar o estado de um Sistema de Transição de Estados. Formalmente, um *problema de planejamento* é definido pela seguinte expressão:

$$P = (\Sigma, s_0, g)$$

Onde:

s_0	é o estado inicial, e
g	é o estado objetivo (ou *goal*)

Um problema de planejamento tem o propósito de produzir uma sequência de ações que levam o sistema do estado inicial s_0 ao estado final ou objetivo g. Esta sequência de ações é denominada *plano*. Um plano é gerado por um *planejador*, que é um programa que realiza a busca pela sequência de ações adequada para a solução do problema de planejamento para o sistema Σ em questão. O sistema Σ pode ser definido pela expressão:

$$\Sigma = (S, A, E, \gamma)$$

Onde:

$S = s_1, s_2, \ldots$	são estados,
$A = a_1, a_2, \ldots$	são ações,
$E = e_1, e_2, \ldots$	são eventos, e
$\gamma = S \times A \times E \rightarrow 2^S$	é uma função de transição de estados.

O estado de um sistema pode ser alterado tanto através de *eventos*, quanto de *ações*. A diferença entre eles é que o executor do plano gerado tem controle sobre as

ações, enquanto que eventos são produzidos externamente e também podem alterar o estado do sistema, mas que ocorrem sem que o executor do plano tenha controle ou conhecimento prévio.

As ações e eventos são operações que, quando executadas no sistema Σ, provocam como *efeito* uma transição de estado. Para que possam acontecer, possuem *precondições* que precisam ser satisfeitas no momento da sua execução. Se as precondições de uma ação são verdadeiras, dizemos que a ação é *aplicável* ao sistema Σ. A função de transição de estados γ determina os estados de origem partir dos quais chega-se aos estados de destino, através de ações e eventos.

Por fim, o conjunto de ações possíveis, que podem acontecer no sistema Σ em questão, faz parte da descrição do seu *domínio*. O domínio descreve as regras do mundo no qual o sistema está inserido e é aplicável para uma grande quantidade de problemas a serem resolvidos para aquele mundo em particular.

2.3 OSGi

O *framework* OSGi faz parte de uma iniciativa maior chamada de *OSGi Services Platform*. Esta plataforma foi criada e é mantida pela OSGi Alliance, fundada em 1999. É uma plataforma aberta e com baixos requisitos de recursos computacionais, cujo foco inicial foi dispositivos embarcados e automação residencial.

Porém, suas características de dinamicidade chamaram a atenção de outras áreas e, atualmente, a plataforma vem sendo utilizada também em aplicações diversas e sistemas corporativos. A lista de empresas e aplicações é bem extensa[1] e este é um dos motivos pelos quais foi escolhido para ser utilizado neste trabalho em detrimento à outros *frameworks* e *middlewares*.

Na *OSGi Services Platform*, o *framework* OSGi ocupa um lugar central, como mostra a Figura 2.4. Ele é um *framework* Java de uso geral, seguro e gerenciável que se define como um "sistema dinâmico de módulos para Java"[2]. Também pode ser visto como um *middleware*, já que exerce o papel de uma camada existente entre a aplicação e as camadas mais baixas da máquina virtual Java e o sistema operacional.

O *framework* é dividido em camadas, onde as camadas de mais alto nível dependem das camadas de mais baixo nível, mas estas podem ser utilizadas separadamente, sem depender das camadas superiores.

As próximas seções detalham os conceitos mais importantes de cada camada, até chegar na camada de serviços, sobre a qual o iPOJO (Seção 2.4) se baseia.

[1] http://www.osgi.org/Technology/WhyOSGi
[2] http://www.osgi.org

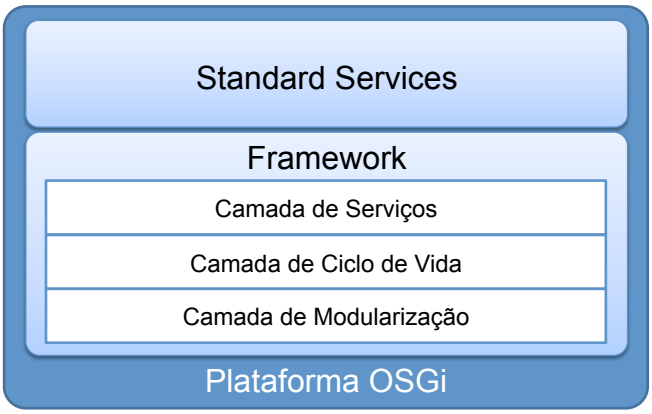

Figura 2.4: Plataforma OSGi e suas camadas internas

2.3.1 Camada de Modularização

A dinamicidade do OSGi se baseia no conceito de *bundle*:

Definição 6 (*Bundle*): Um *bundle* é um módulo no universo do OSGi. Pode ser entendido como "um conjunto de classes logicamente encapsuladas, uma API pública opcional baseada em um subconjunto destas classes e um conjunto de dependências externas" (HALL *et al.*, 2011).

Através de um *bundle* é possível realizar o conceito de "separação de responsabilidades", onde se encapsula o código tornando visível ao ambiente externo somente o que precisa ser exposto para que o módulo seja utilizado. Um *bundle* é, assim, funcionalmente independente dos demais.

A camada de modularização define regras claras para o compartilhamento ou a ocultação de pacotes Java entre *bundles*. É importante ressaltar que a plataforma Java provê uma certa capacidade inata de modularização, porém mais voltada para a implementação de conceitos da orientação a objetos, no nível da classe e seus métodos. Esta capacidade oferecida não é adequada para um nível mais alto de granularidade, desejável para um ambiente mais dinâmico. Em um *bundle*, deve-se determinar explicitamente quais pacotes Java podem ser visíveis externamente. Desta forma, estende-se o conceito de classes públicas, privadas e protegidas da linguagem Java em um nível de granularidade mais elevado. A Figura 2.5 ilustra este conceito visualmente.

Um *bundle* também pode ser visto como uma unidade física de modularização na forma de um arquivo .JAR contendo código, outros recursos necessários (como, por exemplo, figuras) e metadado, onde a fronteira de um JAR também serve como fronteira de encapsulamento para a modularização lógica em tempo de execução. Através deste metadado[3], as informações de dependência do *bundle*, além da sua

[3]Mais informações sobre a sintaxe deste metadado pode ser encontrada em (HALL *et al.*, 2011).

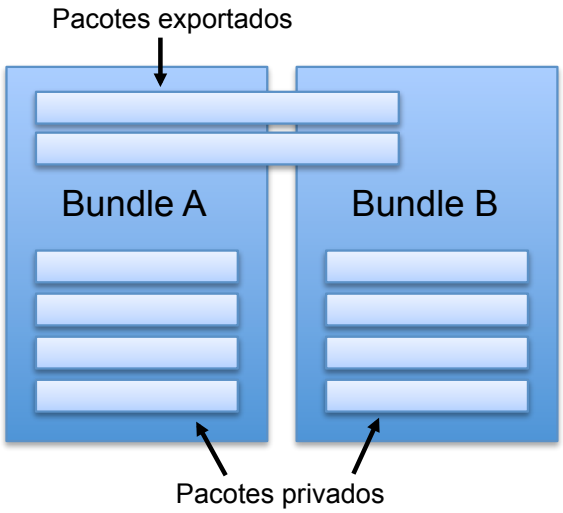

Figura 2.5: Ilustração do conceito de pacotes privados e exportados

identificação única e outros parâmetros, podem ser lidas pelo *framework*. Estas informações serão utilizadas no gerenciamento do *bundle*.

Para assegurar a separação lógica de um *bundle*, a especificação do OSGi determina que cada *bundle* tenha o seu próprio *classloader*. O *classloader* é um objeto responsável por carregar classes e produzir objetos. Ele busca o código da classe em questão e o instancia na memória, gerando um objeto daquela classe. Cada objeto gerado está associado ao classloader que a criou.

Em uma aplicação Java normal, cada *classloader* está associado, por sua vez, a um *classloader* pai. Isto é feito para que uma classe não seja carregada mais de uma vez por diferentes *classloaders*. A máquina virtual Java possui a instância do *classloader* que é pai de todos os outros da hierarquia (*bootstrap classloader*). Ao produzir um objeto, o *classloader* em questão sempre delega a criação do objeto para o *classloader* pai antes de tentar ele mesmo instanciar a classe.

No OSGi, este processo funciona de maneira diferente. Não existe mais a hierarquia de *classloaders* completa. Cada *bundle* possui o seu próprio *classloader* e todos são filhos do *bootstrap classloader*. Logo, o código de cada *bundle* só consegue carregar e instanciar classes do próprio *bundle*. A Figura 2.6 mostra um exemplo de dois *bundles* que possuem uma relação de dependência.

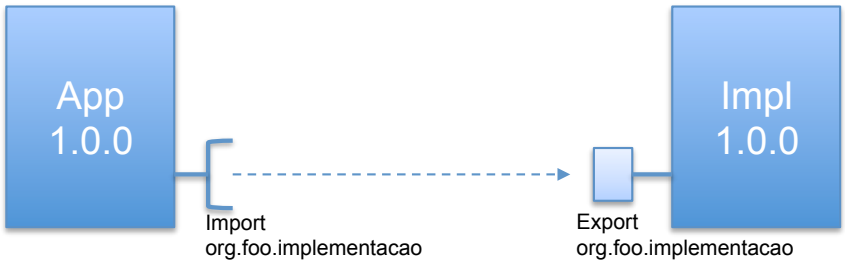

Figura 2.6: Dependência entre *bundles*

Na figura, é possível ver que o *bundle* "App" depende do *bundle* "Impl". Isto significa que o código interno do *bundle* "App" faz referência à classes que só existem no *bundle* "Impl". Ao estabelecer a relação de dependência entre os *bundles*, o *classloader* do *bundle* "App" delega a criação dos seus objetos para o *classloader* do *bundle* "Impl", conseguindo, então, visibilidade das classes que estão fisicamente encapsuladas no JAR deste último. Caso a classe seja local, ele, então, realiza a busca da classe e criação do objeto. Desta forma, o OSGi cria um isolamento entre os *bundles* através do mecanismo de carga das classes.

Uma vez que o mecanismo de *classloaders* do OSGi pressupõe a definição explícita das dependências entre *bundles*, é necessário verificar estas dependências antes do início da sua execução. Por isso, antes da utilização de um *bundle*, o *framework* realiza automaticamente esta tarefa. Isto elimina a necessidade da definição manual do *classpath* de uma aplicação Java, atividade que pode ser bem trabalhosa em um sistema de grande porte. É importante ressaltar que a resolução das dependências leva em consideração a versão do *bundle*, o que faz com que seja possível carregar ao mesmo tempo diferentes versões de um *bundle*, capacidade bastante interessante para promover a evolução do software em tempo de execução. Uma vez com as dependências resolvidas, os *bundles* ganham o acesso necessário para que possam ser executados sem exceções do tipo "ClassNotFoundException" em tempo de execução.

2.3.2 Camada de Ciclo de Vida

Esta camada implementa as operações de gerenciamento do ciclo de vida de um *bundle*. O ciclo de vida de um *bundle* é definido pelo diagrama de estados exibido na Figura 2.7. Todas as trocas de estado neste diagrama podem ser realizadas de maneira programática, em tempo de execução, através da API provida pela camada em questão.

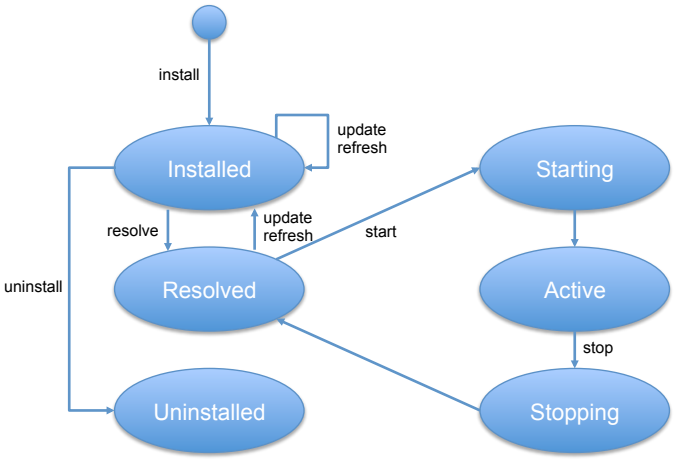

Figura 2.7: Ciclo de vida de um *bundle* no OSGi

A instalação de um *bundle* é a primeira etapa do seu ciclo de vida. Esta ação deve ser realizada por outro *bundle* ou de maneira programática, quando o *framework* estiver sendo inicializado. Neste ponto, é importante esclarecer que o próprio *framework*, após a sua inicialização, assume o papel do "*bundle* zero", ou seja, ele, por si só, é um *bundle*, que pode realizar a carga de todos os outros.

Uma vez instalado, o *bundle* tem suas dependências resolvidas, ou seja, é verificado se os demais *bundles* dos quais ele depende estão disponíveis para serem utilizados. Isto é necessário para que os *bundles* possam ter acesso às suas APIs públicas, conforme colocado na Seção 2.3.1. Somente quando todas as dependências são satisfeitas, o *bundle* segue para o estado "*Resolved*".

Quando as dependências do *bundle* estão resolvidas, ele pode ser iniciado. Opcionalmente, um *bundle* pode registrar seu interesse em receber uma notificação quando é iniciado. Para isto, ele precisa implementar um *Bundle Activator*, que é uma classe que implementa a *Interface* Java `BundleActivator`. Esta *Interface* possui os métodos `start` e `stop`, que são chamados quando um *bundle* inicia e para, respectivamente, a sua execução. Nem todo *bundle*, porém, possui um *Bundle Activator*. Esta classe só é necessária quando existirem passos a serem realizados na inicialização e/ou na parada do *bundle*.

Quando o *bundle* é inicializado, ele recebe um objeto chamado `BundleContext`, por meio do qual possui acesso aos métodos da camada de ciclo de vida, que o possibilita instalar e iniciar outros *bundles*. Além disso, através do `BundleContext`, tem acesso à informações gerais sobre o estado do *framework*, pode se registrar para receber determinados eventos e tem acesso à camada de serviços, detalhada na Seção 2.3.3.

Para que um *bundle* possa ser parado, o método `stop` do seu *Bundle Activator* deve ser chamado. Isto é feito por outro *bundle* ou pelo próprio *bundle*, em cenários específicos, através do `BundleContext`. Uma vez parado o *bundle* pode ser desinstalado, quando passa a não mais estar disponível no ambiente de execução do OSGi.

O *framework* OSGi possui um mecanismo de eventos implementado em cima do ciclo de vida de um *bundle*. Isto é importante pois, como o ambiente do OSGi é muito dinâmico, alterações de estado de *bundles* já existentes ou o surgimento de novos *bundles* podem acontecer a qualquer momento.

Cada alteração de estado de um *bundle* dispara um evento que pode ser escutado por outros *bundles* para que alguma ação seja tomada, dependendo da necessidade. Existem dois tipos de eventos: `FrameworkEvent` e `BundleEvent`. O primeiro está relacionado à determinadas ações que podem acontecer no *framework*, como, por exemplo, o término da sua inicialização ou um erro. Já o segundo está relacionado às alterações de estado dos *bundles*.

2.3.3 Camada de Serviços

Um serviço pode ser entendido como um "trabalho feito para outros", regido por um contrato. Quando se utiliza serviços, o foco não é em "como" o trabalho será feito, mas sim "qual" trabalho será feito. Qualquer serviço que atenda ao contrato definido, teoricamente, é capaz de realizar o trabalho que se deseja.

A utilização de serviços pressupõe duas partes, a primeira que expõe o serviço, ou seja, que disponibiliza a funcionalidade, e a segunda que consome o serviço, ou seja, que utiliza a funcionalidade disponibilizada.

A camada de Serviços define um modelo de serviços dinâmico e altamente integrado com a camada de ciclo de vida. É a parte SOA do OSGi. Utiliza um modelo de publicação, busca e ligacão (*publish*, *find*, *bind*) de serviços, como mostra a Figura 2.8. Um serviço, neste contexto, é uma classe Java registrada no catálogo de serviços do OSGi sob uma ou mais *Interfaces*, que fazem o papel do contrato.

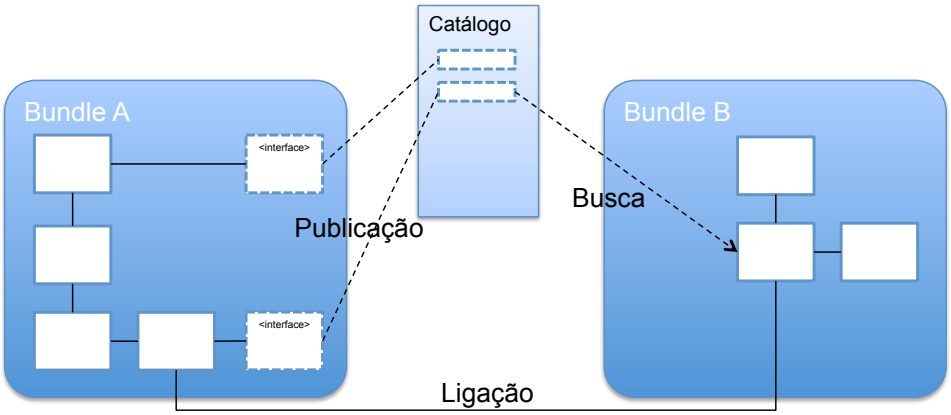

Figura 2.8: Ilustração do mecanismo de publicação, busca e ligação da camada de serviços do OSGi

Bundles podem registrar um serviço, procurar por um serviço específico e/ou receber notificações sobre a sua disponibilidade. Como o contrato de um serviço no OSGi é uma *Interface* Java, é possível realizar a troca da sua implementação em tempo de execução, o que é uma vantagem para sistemas auto-adaptáveis.

Um serviço existe no contexto de um *bundle*. O serviço deve ser registrado por um *bundle* para que outros *bundles* possam ter acesso a ele e só é acessível enquanto o seu *bundle* estiver ativo. Caso o *bundle* seja parado, o *framework* remove o registro daquele serviço. Por este motivo, se por um lado a utilização de serviços aumenta o desacoplamento geral do sistema, por outro, é necessário prever o cenário do serviço não estar mais disponível quando tiver que ser utilizado.

O *framework* OSGi possui um catálogo de serviços (*Interfaces* Java registradas), exibido na Figura 2.8, que mantém uma lista de suas implementações. Através deste catálogo, provê uma maneira simples, mas poderosa, do *bundle* acessar um serviço somente quando ele for necessário. Além disso, a camada de serviços estende o

mecanismo de eventos para que os *bundles* possam ser notificados do surgimento, modificação ou exclusão de um serviço.

No entanto, a utilização de serviços tem um custo associado, que é o da busca e acesso ao serviço, necessário sempre que se desejar utilizá-lo da primeira vez. Encontrar o serviço adequado é uma operação que pode acontecer apenas na sua primeira utilização ou sempre que for utilizado, dependendo do nível de dinamicidade do sistema.

A camada de Serviços do OSGi provê as bases para a criação de aplicações altamente dinâmicas utilizando serviços. No entanto, a especificação "core" do OSGi não define maneiras de se lidar com esta dinamicidade automaticamente. Isto é deixado à cargo das aplicações. Um serviço pode ficar indisponível a qualquer momento e quem o tiver utilizando precisa lidar com o cenário de indisponibilidade por si mesmo.

Além disso, o mecanismo de publicação, busca e ligação, etapas necessárias para a utilização de serviços requer a utilização de uma certa quantidade de código para cada ligação entre serviços. Estas etapas não estão automatizadas na camada de serviços do OSGi.

Estas deficiências abrem caminho para a criação de extensões do OSGi que visam preencher estas lacunas. A próxima seção descreve algumas delas, cuja solução para os problemas expostos passa pela definição de um modelo de componentes, elevando o nível de abstração para o patamar desejado para este trabalho.

2.3.4 Modelos de Componentes para o OSGi

De maneira geral, os modelos de componentes definidos como uma extensão da camada de serviços do OSGi se apoiam no dinamismo oferecido por esta última. Utilizam mais algumas chaves adicionadas ao metadado do OSGi, para obter as informações sobre os componentes que ficam encapsulados em um *bundle*. Além disso, também disponibilizam uma maneira automática para lidar com as dependências entre componentes, criando a noção de serviço provido e consumido.

A especificação do OSGi traz a definição de um modelo de componentes chamado *Declarative Services*, que oferece uma solução para alguns dos problemas colocados na seção anterior.

O *Declarative Services* é um modelo de componentes leve que tem o objetivo de possuir um consumo de memória baixo e simplificar a utilização de serviços, provendo um código padrão para lidar com a dependência entre serviços. Sua implementação foca em tempos de inicialização rápidos.

Outro modelo de componentes que também faz parte da especificação do OSGi é o *Blueprint*. Sua implementação é fortemente baseada na integração do OSGi com o

framework de componentes Spring, através da iniciativa *Spring Dynamic Modules*[4].

Como o foco do *Blueprint* está voltado para aplicações mais complexas, ele é mais pesado do que o *Declarative Services*, mas também possui mais funcionalidades e utiliza técnicas mais poderosas para atingir os seus objetivos como, por exemplo, injeção de dependências. Injeção de dependências é um padrão de projeto que utiliza *Interfaces* Java e a característica de Inversão de Controle provida pelos *frameworks* de componentes para desacoplar a ligação entre classes[5]. Isto facilita, ainda, o adiamento para o tempo de execução da decisão a respeito de qual objeto será utilizado para atender às necessidades de determinada parte do código de outro objeto.

Por fim, outro *framework* de componentes criado para tratar as lacunas expostas na seção anterior é o iPOJO. Sendo o mais completo dos três, o iPOJO também utiliza injeção de dependências para realizar a ligação entre os seus componentes. Além disso, possui um alto nível de extensibilidade, característica fundamental para este trabalho, conseguida através da criação de um nível de indireção que envolve o código do componente, provendo uma maneira de interceptar suas chamadas. Isto é conseguido através da manipulação direta do *bytecode* do componente, em uma etapa posterior à compilação do código Java do componente. Finalmente, é o único que oferece um modelo de componentes verdadeiramente hierárquico. Por estes motivos, foi o escolhido para ser utilizado neste trabalho. Para uma comparação mais detalhada entre estas três abordagens, referir-se a (HALL *et al.*, 2011). A próxima seção detalha o iPOJO.

2.4 iPOJO

Esta seção descreve em detalhes algumas características do iPOJO necessárias para o entendimento do mecanismo de execução do plano de reconfiguração arquitetural proposto no Capítulo 4.

2.4.1 Conceitos Básicos

O iPOJO pode ser definido como um *framework* de componentes hierárquico que utiliza injeção de dependências para realizar a ligação entre seus componentes. Ele estende a camada de serviços do OSGi de forma a aproveitar toda dinamicidade e baixo nível de acoplamento providos pelos serviços.

Em termos conceituais, o componente descrito pelo modelo do iPOJO atende à Definição 1. Também provê uma separação clara entre componente e a ideia de

[4]http://www.springsource.org/osgi
[5]http://martinfowler.com/articles/injection.html

instância de componente, de acordo com a Definição 2. No modelo do iPOJO, as duas entidades existem no mesmo nível. Porém, para deixar clara a separação entre componente e instância de componente, o iPOJO chama o componente de **Tipo de Componente**.

Um tipo de componente é a definição do componente que será materializado, posteriormente, através das suas instâncias. No entanto, neste trabalho, no contexto do iPOJO, o Tipo de Componente será chamado simplesmente de Componente. Um tipo de componente define em termos técnicos uma *Factory* (ou Fábrica) (GAMMA *et al.*, 1994). Uma fábrica é um padrão de projeto que, no contexto do iPOJO, é uma entidade responsável pela criação de instâncias de componentes. Um tipo de componente pode estar ativo ou inativo, quando não está disponível para criar novas instâncias. Uma instância de componente pode, por sua vez, estar válida ou inválida e seu estado depende, na maioria dos casos, da resolução das suas dependências.

Uma instância de componente pode depender funcionalmente de outras instâncias para conseguir executar as tarefas para as quais foi projetada. Estas dependências são projetadas na forma de interfaces entre componentes, que, para o iPOJO, são serviços. Um componente pode expor (ou prover) e/ou consumir (ou requerer) serviços.

Definição 7 (Serviço Provido): Um serviço provido representa uma capacidade do componente. Em termos arquiteturais, é um ponto de ligação com outros componentes através do recebimento de requisições. É um serviço OSGi cadastrado no seu catálogo interno.

Na prática, um serviço provido é uma *Interface* Java implementada pela classe que representa o componente. Ao identificar um serviço provido, o iPOJO registra a classe em questão, identificada pela *Interface* implementada no catálogo de serviços interno do OSGi. Quando necessário, o iPOJO busca neste catálogo de serviços pela *Interface* necessária e instancia a classe que a implementa, injetando-a no componente que consome o serviço. Este componente, então, deve declarar um serviço requerido:

Definição 8 (Serviço Requerido): Um serviço requerido representa uma necessidade do componente que o requer, uma lacuna não atendida. Significa que, para funcionar corretamente, o componente que o declara deve ser ligado a um serviço provido correspondente, para o qual envia requisições.

Um serviço requerido é um atributo (cuja cardinalidade pode ser uma ou muitas instâncias) de uma classe que representa um componente, cujo tipo é uma *Interface* Java, e que é injetado em tempo de execução pelo *framework* iPOJO. O iPOJO identifica a cardinalidade do atributo automaticamente e injeta uma ou mais instâncias de componente no referido atributo.

Ao definir os conceitos de serviço provido e requerido e implementar a realização das suas ligações no *framework*, o iPOJO automatiza as etapas de publicação, busca e ligação dos serviços do OSGi.

Um componente iPOJO pode ter propriedades que estão associadas ao conceito de adaptação por parâmetro (MCKINLEY *et al.*, 2004). Estas propriedades podem ter seus valores alterados em tempo de execução pelo *framework*.

Definição 9 (Propriedade): Uma propriedade de um componente representa um atributo que pode ser alterado em tempo de execução pelo *framework* iPOJO.

Uma propriedade de um componente é um atributo da classe Java que o representa, da mesma forma que um serviço requerido. A diferença de uma propriedade para um serviço requerido é que a propriedade não tem a função de realizar uma ligação com outro componente. É apenas um valor que pode ser alterado em tempo de execução pelo *framework* ou pela própria classe.

Por fim, como todo *framework*, o iPOJO possui a característica de inverter o controle de execução da aplicação. Ou seja, os componentes são chamados e controlados pelo *framework* e não detêm o controle do fluxo principal da aplicação. Porém, o iPOJO dá a oportunidade dos componentes se registrarem para serem chamadas em momentos específicos da sua execução através de *callbacks*:

Definição 10 (*Callback*): Um *callback* de um componente é a porta de entrada de requisições geradas pelo *framework* em resposta a determinados eventos ocorridos na sua infraestrutura como, por exemplo, o consumo de um serviço ou a alteração do estado do *framework*.

A declaração das características de um componente pode ser feita de três maneiras:

1. **Metadados em um XML:** o iPOJO lê de um XML a declaração do componente, das suas características e das suas instâncias. Desta forma, é possível separar o código do componente da sua declaração;

2. ***Annotations*:** a declaração dos componentes e todas as suas características pode ser feita também através de *annotations* Java, diretamente no código. Isto evita a necessidade de um arquivo XML externo; e

3. **API:** o *framework* do iPOJO também provê uma API para a criação e instanciação de componentes.

As três maneiras são equivalentes e geram componentes e instâncias idênticas. Sua utilização é mais uma questão de preferência do desenvolvedor, sendo que uma

diferença da API para as outras duas maneiras é que estas definem o componente em tempo de compilação, através de uma etapa extra de manipulação de *bytecode* do *bundle* que abriga os componentes. Já via API, a definição do componente é realizada em tempo de execução.

Um componente iPOJO é, basicamente, um POJO[6], ou seja, uma classe Java, adicionada de controles para que possa ser gerenciada pelo *framework*. A Figura 2.9 mostra uma representação de um componente iPOJO.

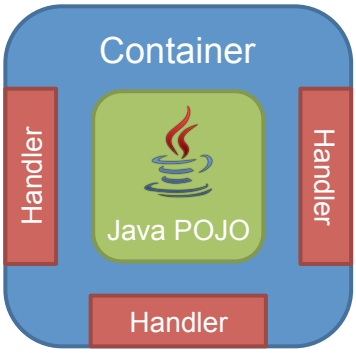

Figura 2.9: Representação de um componente iPOJO

O processo de criação do componente iPOJO cria uma camada de encapsulamento em torno do POJO. Esta camada intercepta as chamadas aos atributos e métodos do POJO e dá ao *framework* controle total sobre a sua execução, alimentando o seu mecanismo interno de geração de eventos.

Estes pontos de interceptação presentes na camada que encapsula o POJO não alteram o comportamento do código original em si, mas delegam para os *handlers* as funções de gerenciamento do componente. Um componente pode ter diversos *handlers* plugados, responsáveis por gerenciar suas características não funcionais.

Definição 11 (*Handler*): Um *handler* é uma classe Java que é "plugada" no contêiner que encapsula o POJO e é responsável por toda comunicação do POJO com o mundo externo. Cada *handler* é responsável por uma característica não funcional do componente como, por exemplo, publicação de serviços, consumo de serviços, persistência, etc.

Em tempo de execução, um *handler* possui dois estados possíveis: válido ou inválido, assim como as instâncias de componente. O estado em que ele se encontra depende do quão bem conseguiu realizar o seu objetivo. Por exemplo, um *handler* responsável por consumir um serviço pode ficar inválido caso não exista nenhum outro componente provendo aquele serviço. O estado dos *handlers* determina diretamente o estado da instância de componente à qual estão associados. Se todos

[6] *Plain Old Java Object*

os *handlers* de uma instância estão válidos, o estado da instância também é válido. Caso contrário, a instância fica inválida.

Os *handlers* são a chave do mecanismo de extensibilidade do *framework*. Através da criação de novos *handlers*, o componente pode ser estendido e adicionado de novas funcionalidades como, por exemplo, a capacidade de persistência das suas características ou a capacidade de troca de mensagens SOAP.

A especificação do iPOJO define seis *handlers* que são responsáveis pelas funcionalidades principais descritas até aqui. Existem, entretanto, outros *handlers* disponíveis que visam prover outras capacidades para o POJO.

O ***Handler* de Publicação de Serviços** é o responsável por definir os serviços que um componente expõe para ser utilizado por outros componentes. A partir da informação declarada no metadado do componente, realiza o cadastro do serviço no catálogo de serviços do OSGi.

O *handler* análogo ao de Publicação de Serviços é o ***Handler* de Consumo de Serviços**, que define uma dependência. Além disso, este *handler* automatiza o processo de busca de serviços providos e ligação com os componentes provedores. Esta automatização é capaz de gerenciar automaticamente o dinamismo inerente aos serviços. Caso a instância de componente que esteja provendo o serviço fique inválida, ou inacessível, este *handler* recebe um evento, automaticamente seleciona o próximo componente ativo que disponibilize aquele serviço e o instancia realizando a ligação necessária. Somente no caso da não existência de outro componente que atenda o serviço requerido, o *handler* fica inválido, invalidando, por consequência, a instância de componente.

Uma variação do *handler* de Consumo de Serviços é o ***Handler* de Dependência Temporal**. Ele funciona da mesma maneira, mas quando, por algum motivo, o serviço requerido não está mais disponível em nenhum componente ativo, ao invés de invalidar a instância de componente à qual está relacionado, ele coloca a sua *thread* em espera por tempo configurável (três segundos por padrão). Caso, durante este tempo de espera, algum componente provedor do serviço requerido fique ativo, este é instanciado, a dependência volta a ficar satisfeita e a *thread* tem a sua execução liberada.

Caso, durante o tempo de espera, um componente provedor de serviço não fique ativo, o *handler* pode ser configurado para atuar de quatro maneiras diferentes: injeta *null* no serviço requerido; injeta um objeto *nullable*[7]; injeta um *array* vazio, no caso do serviço requerido ser multivalorado; ou injeta uma instância de uma implementação padrão da *Interface* Java em questão.

Caso um componente precise realizar ações quando seu estado for alterado (de

[7]Um objeto "nullable" implementa a *Inteface* que identifica o serviço requerido de maneira a não fazer nada. Chamadas aos métodos deste objeto não produzirão nenhum efeito.

válido para inválido ou vice-versa), basta que defina *callbacks* na sua declaração. Quando isto acontece, o ***Handler* de Gerenciamento de *Callbacks*** é o responsável pelo cadastramento de métodos do componente que o declara para serem chamados no caso de eventos de validação e invalidação.

Outro *handler* principal definido pelo iPOJO é o ***Handler* de Arquitetura**. A função deste *handler* é prover acesso ao componente para a capacidade de Reflexão Computacional do iPOJO. Ele não afeta o comportamento do componente, mas provê acesso ao estado da instância de componente e de todos os seus *handlers*. É muito útil para a investigação de problemas já que fornece uma descrição interna completa da instância de componente. Ele é associado à todos os componentes automaticamente, mas na declaração do componente é possível, através de um parâmetro, impedir que esta associação seja feita.

2.4.2 Modelo de composições hierárquicas do iPOJO

Uma das características do iPOJO que o diferencia dos outros *frameworks* de componente para OSGi é prover meios para a criação de aplicações através da composição hierárquica de componentes. Uma composição é um componente composto e estes dois termos serão utilizados indistintamente nesta dissertação.

Esta característica alavanca a propriedade dos componentes de serem entidades "caixa-preta". Isso significa que, conceitualmente, um componente formado por outros componentes de maneira hierárquica atua como se fosse um componente simples, não necessitando de nenhum tratamento especial por parte da aplicação para sua utilização. Na prática, uma composição é implementada no *framework* como uma camada de encapsulamento que agrupa instâncias de componentes (simples ou compostos), definindo as suas fronteiras, tornando-o, com isso, uma "caixa-preta".

A camada de encapsulamento cria uma fronteira lógica que define dois mundos: o interno à composição e o externo à composição. Logo, uma instância de componente interna que provê um serviço só pode ser utilizada por outras instâncias internas à composição. O mesmo acontece com serviços requeridos, que só podem ser satisfeitos por serviços internos à composição. Como podem haver diversos níveis hierárquicos, vários escopos de visibilidade são definidos.

A maneira do iPOJO de implementar esta fronteira lógica que delimita uma composição é através do gerenciamento de vários catálogos de serviço. Cada composição possui o seu próprio catálogo de serviços, acessível somente pelos componentes internos à composição. Em uma composição com vários níveis hierárquicos, cada nível possui o seu próprio catálogo de serviços. Além disso, esta fronteira lógica também controla a visibilidade das instâncias de componente, só permitindo o seu acesso internamente à composição.

Porém, para que um componente composto realmente faça o papel de um componente simples, é preciso que ele tenha a capacidade de prover e consumir serviços. Este canal de comunicação entre os mundos internos e externos à composição também é responsabilidade da camada de encapsulamento.

Para atingir tal objetivo, o *framework* define as ações de importar e exportar serviços para o nível hierarquicamente superior. Na prática, isso significa que um serviço provido por um componente interno à composição é registrado também no catálogo de serviços do nível superior e um serviço requerido por um componente interno pode ser atendido por um serviço provido no nível superior, que é importado e registrado no catálogo de serviços interno à composição. Podemos definir os conceitos de serviço importado e exportado:

Definição 12 (Serviço Importado): Um serviço importado é um serviço provido por um componente de mesmo nível hierárquico que a composição e que é registrado no seu catálogo interno, de forma que pode ser utilizado por componentes internos para atender às suas dependências. Está para um componente composto assim como um serviço requerido está para um componente simples.

Definição 13 (Serviço Exportado): De forma complementar ao serviço importado, um serviço exportado é provido por um componente interno à composição e registrado no catálogo de serviços do nível hierárquico no qual a composição se encontra. Está para o componente composto assim como um serviço provido está para um componente simples.

A Figura 2.10 ilustra os conceitos de composição expostos até aqui.

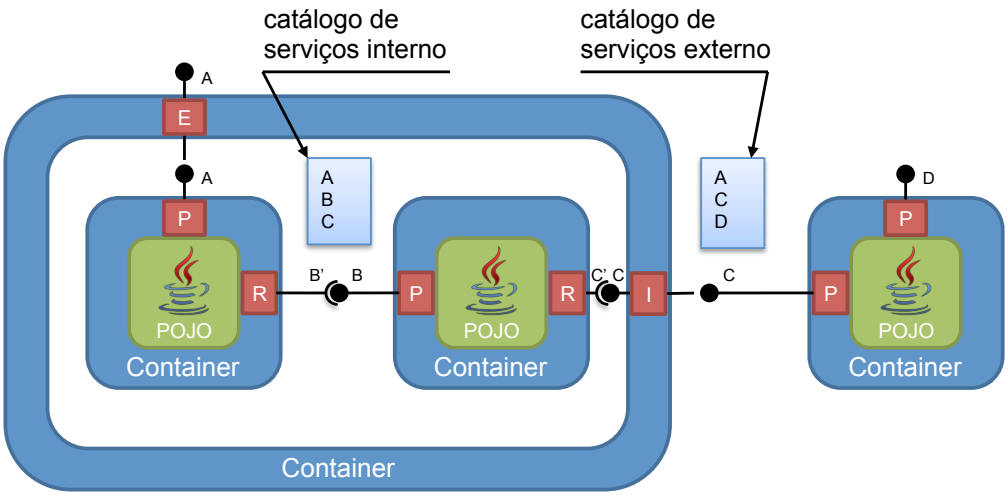

Figura 2.10: Representação de um componente composto e o relacionamento entre o mundo interno e externo

Assim como no contêiner que envolve o POJO transformando-o em um componente iPOJO simples, a importação e a exportação de serviços entre os contextos

são funcionalidades delegadas para *handlers* que são plugados no contêiner externo, que representa o componente composto.

Um dos *handlers* disponíveis para uma composição é o ***Handler* de Exportação de Serviço**. Este *handler* é o responsável por prover visibilidade de um serviço interno no escopo imediatamente superior. Para que funcione corretamente, deve conhecer a especificação da *Interface* que deve ser exportada e um parâmetro de filtro, que indica qual instância interna à composição provê aquele serviço.

Outro *handler* importante é o ***Handler* de Importação e Instanciação de Serviço**. A primeira função deste *handler* é a de realizar a operação inversa ao anterior, importanto um serviço do nível hierárquico imediatamente superior. Para isso, realiza o seu cadastro no registro interno do componente composto. Isto faz com que ele se torne visível dentro da composição.

A segunda função deste *handler* é a ação de instanciação. Enquanto a ação de importação faz com que um determinado serviço externo seja visível internamente, a ação de instanciação cria, de fato, uma instância a partir de sua especificação. Este comportamento é diferente de uma simples criação de instância de componente dentro de uma composição. Nesta última operação, o tipo de componente é explicitamente definido, enquanto que na ação de instanciação do *handler* em questão, qualquer tipo de componente que implemente a *Interface* Java passada como parâmetro pode ser instanciado. Esta funcionalidade é muito útil quando se deseja que o *framework* gerencie o cenário específico de reconfiguração, onde uma simples troca de componentes, sem a alteração radical da topologia da composição, é suficiente.

2.5 Considerações Finais

Esta seção apresentou os principais conceitos que estão por trás deste trabalho. Em particular os conceitos relacionados à arquitetura de software e desenvolvimento baseado em componentes, apresentados na Seção 2.1.

Além disso, são introduzidos conceitos referentes à Planejamento Automatizado, uma disciplina da área de Inteligência Artificial muito importante para o entendimento da base na qual este trabalho está apoiado. Este conceito é utilizado para a determinação da noção de Plano de Reconfiguração, que é entidade fundamental para a execução da adaptação.

Por fim, o OSGi e o iPOJO, que formam a plataforma utilizada neste trabalho, ocupam lugar de destaque. No contexto do iPOJO, o conceito de *handler* é de suma importância, pois é o mecanismo através do qual é possível estender o iPOJO.

Capítulo 3

Trabalhos Relacionados

Este capítulo visa descrever os trabalhos relacionados encontrados em uma pesquisa realizada nas áreas de Sistemas Auto-Adaptáveis e Reconfiguração Dinâmica.

O trabalho descrito nesta dissertação nasceu na área de Sistemas Auto-Adaptáveis. Por isso, foi realizado um estudo desta área. Posteriormente, ao concentrar os estudos na execução da adaptação, a área de Reconfiguração Dinâmica foi explorada.

Este capítulo conta um pouco desta história. Começa analisando alguns trabalhos na linha de Sistemas Auto-Adaptáveis, que podem ser classificados quanto ao tipo de abordagem: trabalhos que remetem à Sistemas Autonômicos, que visam definir mecanismos para a implementação do ciclo MAPE-K, e trabalhos inspirados em Linhas de Produtos de Software Dinâmicas (HALLSTEINSEN *et al.*, 2008), onde pontos de variabilidade definidos em tempo de projeto destacam as partes da aplicação que podem ser alteradas em tempo de execução. Estes dois tipos de abordagens, apesar de apresentarem capacidades de reconfiguração dinâmica, esta são, geralmente, implícitas e limitadas, não provendo a flexibilidade necessária para determinadas reconfigurações não previstas em tempo de projeto.

Em seguida, são descritos alguns trabalhos na linha de Reconfiguração Dinâmica. Considerando a Reconfiguração Dinâmica uma maneira possível de se executar a adaptação, podemos dizer que estes trabalhos estão relacionados apenas à ação de Execução do ciclo MAPE-K.

A reconfiguração dinâmica é conseguida através da execução de ações de reconfiguração, geralmente de forma direta, alterando a arquitetura do sistema. Por este motivo, estas ações estão fortemente ligadas aos aspectos arquiteturais do *framework* que as implementa. Uma característica destas abordagens é que elas propõem um conjunto de ações genéricas, que podem ser aplicadas a qualquer sistema que seja construído utilizando o *framework* de componentes que as define. Ao elevar as ações de reconfiguração à entidades de primeiro nível, estes trabalhos oferecem uma plataforma para a criação de abordagens mais flexíveis, que suportem um conjunto maior

de adaptações não previstas em tempo de projeto.

As próximas seções detalham alguns trabalhos nestas linhas, incluindo alguns que utilizam o OSGi e o iPOJO como plataforma para adaptação/reconfiguração dinâmica.

3.1 Abordagens inspiradas em Sistemas Autonômicos

Dentre os trabalhos relacionados aos conceitos de computação autômica, um que se destaca por utilizar o OSGi como plataforma para a adaptação é o A-OSGi (FERREIRA *et al.*, 2009). Neste trabalho, os autores apresentam uma extensão do OSGi que tem o objetivo de suportar os aspectos autonômicos definidos no ciclo MAPE-K.

O A-OSGi conta com um conjunto de sensores para o monitoramento de parâmetros de QoS, dentre os quais podemos citar a utilização de CPU e memória de cada *bundle*. Inclui, ainda, um mecanismo para controlar mais facilmente o consumo e disponibilização dos serviços de cada *bundle* em tempo de execução. Por fim, adiciona suporte para a interpretação de regras através das quais são definidos os comportamentos autonômicos de aplicações que utilizarem o A-OSGi.

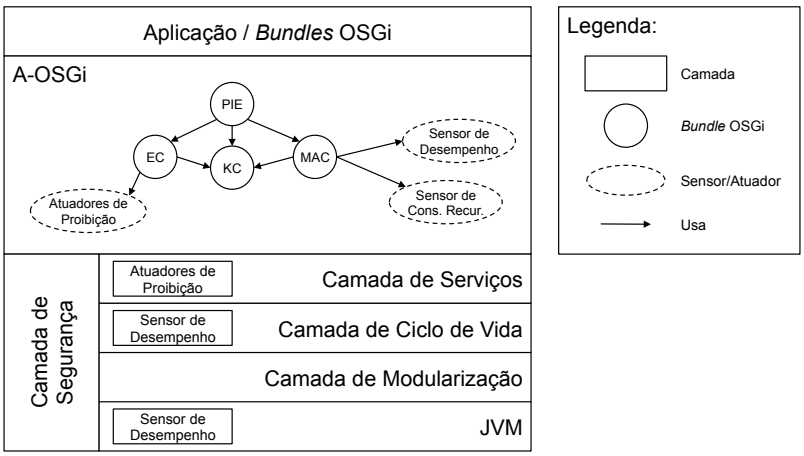

Figura 3.1: Arquitetura do A-OSGi

Para implementar o ciclo MAPE-K como um todo, o A-OSGi conta com um conjunto de componentes implementados na forma de *bundles* OSGi. A Figura 3.1 mostra a arquitetura do A-OSGi, ilustrando a interação entre estes componentes, descritos a seguir:

- Componente de Monitoração e Análise (*Monitoring and Analysis Component – MAC*): Este componente é o responsável por extrair as informações dos sensores. Ele interage com os *bundles*, com o núcleo do OSGi e com a JVM para

monitorar dados como: consumo de CPU e memória, métricas de performance, disponibilidade de *bundles* e serviços, e ligações realizadas entre *bundles* e serviços. Ele utiliza um mecanismo de *publish-subscribe* para gerar informações para os outros componentes interessados.

- Componente de Execução (*Execution Component – EC*): Este componente é responsável por executar as ações em cima da aplicação. Ele pode iniciar e parar *bundles*, alterar propriedades dos serviços e alterar ligações em tempo de execução, definindo "proibições" e "obrigações" de ligação para cada *bundle* da aplicação.

- Componente de Conhecimento (*Knowledge Component – KC*): A responsabilidade deste componente é centralizar informações sobre o estado do ambiente de execução do A-OSGi. Ele mantém uma lista de *bundles* instalados, serviços registrados e dependências (ligações) entre *bundles* e serviços.

- Interpretador e Aplicador de Políticas (*Policy Interpreter and Enforcer – PIE*): Este componente interpreta as políticas do sistema, definidas por um conjunto de regras ECA (*Event-Condition-Action*). O PIE é direcionado pelos eventos gerados pelo MAC, que o notifica em termos da necessidade de adaptação. Utiliza o KC como fonte de informação e pode usar o EC para solicitar a execução de um conjunto de ações, dependendo do estado do sistema e do conjunto de regras ECA pré-estabelecidas.

Para implementar estes componentes, os autores modificaram a implementação de OSGi utilizada (Apache Felix) de forma a:

1. Adicionar na inicialização dos *bundles* um mecanismo para controlar as *threads* instanciadas por cada *bundle*, que passam a ser identificados por um parâmetro chamado *ThreadGroup*. Desta forma, é possível saber o consumo de recursos por *bundle*. Esta modificação teve que ser feita tanto na camada de ciclo de vida do OSGi quanto no nível da própria JVM.

2. A camada de serviços do OSGi também teve que ser modificada para adicionar suporte às "obrigações" e "proibições" de ligação entre *bundles* e serviços.

No artigo, o A-OSGi é avaliado com uma aplicação simples para controle de informações de uma loja virtual que vende CDs e DVDs. Baseado no consumo de recursos corrente do servidor onde a loja é hospedada, o sistema automaticamente troca o nível do serviço que pode ser básico (onde a venda é realizada sem nenhuma funcionalidade adicional) ou premium (onde a venda é realizada e um sistema de sugestões oferece outros CDs/DVDs relacionados). Os dados de execução mostram

que há ganhos na utilização do A-OSGi gerenciando as adaptações e a alteração entre os níveis de serviço básico e premium em relação a um cenário que utiliza OSGi, mantendo o nível de serviço fixo em premium, sem adaptações em relação à utilização da máquina.

É importante ressaltar que o A-OSGi é um exemplo de uma plataforma para a obtenção do comportamento adaptável, que se baseia em um conjunto pré-definido de configurações arquiteturais possíveis para um sistema autonômico a ser contruído sobre ele. Para a inclusão de novos comportamentos, é necessário alterar as regras que definem a adaptação, adicionando suporte para novos *bundles*.

Além disso, apesar da abordagem proposta ter o mérito de tratar todo o ciclo MAPE-K, apresenta alguns pontos negativos. O primeiro está em contar com um conjunto de customizações na implementação de OSGi utilizada (Apache Felix), inclusive para controlar as ligações entre serviços. Esta característica requer que as customizações sejam também consideradas em outras implementações do OSGi, ou até mesmo em novas versões do Apache Felix, para que o A-OSGi seja portável.

Por fim, o A-OSGi utiliza o *bundle* como um componente. Embora existam muitas definições de componente e não haja um consenso a respeito do que pode ou não ser considerado um componente, um *bundle* não foi projetado como um. Por exemplo, considerando que deve ser possível instanciar um componente mais de uma vez, isto não é possível com um *bundle*. É necessário criar uma cópia, renomeá-la e torná-la um novo *bundle*, para que possa ser instanciado novamente. Isto dificulta, por exemplo, a existência de um *pool* de *bundles* com o objetivo de atender à requisições simultâneas em um sistema *multithread*. Além disso, não existe um mecanismo de composição hierárquica entre *bundles*, que seguem um modelo "flat" de interconexão, não atendendo aos requisitos da definição de modelo de componentes adotada nesta dissertação (Seção 2.1).

Ainda na direção do suporte à sistemas autonômicos, podemos citar o Autonomic iPOJO (DIACONESCU *et al.*, 2008), que utiliza o modelo de componentes definido pelo iPOJO. Este modelo de componentes foi projetado para aproveitar as capacidades SOA do OSGi.

O foco do Autonomic iPOJO está em prover capacidades autonômicas para o iPOJO no domínio de sistemas ubíquos e pervasivos. A principal motivação é a proliferação de dispositivos eletrônicos cada vez menores e mais "inteligentes". Os autores argumentam que a utilização de uma plataforma orientada à serviços facilita a reconfiguração da aplicação, dado o alto nível de desacoplamento entre os serviços. O iPOJO foi escolhido como *framework* de componentes utilizado por já ter sido usado em outros projetos de pesquisa com sucesso (por exemplo em (ESCOFFIER *et al.*, 2008)) e por oferecer uma interessante mescla dos conceitos de orientação à serviços e computação baseada em componentes (ESCOFFIER *et al.*, 2007).

O trabalho se baseia fortemente nos conceitos de computação autonômica, em particular com o ciclo MAPE-K. O artigo argumenta que as capacidades autonômicas ou de adaptação devem ser providas de maneira reutilizável pelo *middleware* no qual se apoia a aplicação adaptável, utilizando o conceito de Adaptação Externa (Figura 1.3).

A contribuição principal do trabalho é um conjunto de *handlers* que estendem o iPOJO para suportar as operações autonômicas definidas pelo ciclo MAPE-K. Estas operações são suportadas por um conjunto de *touchpoints* (*handlers*), como mostra a Figura 3.2.

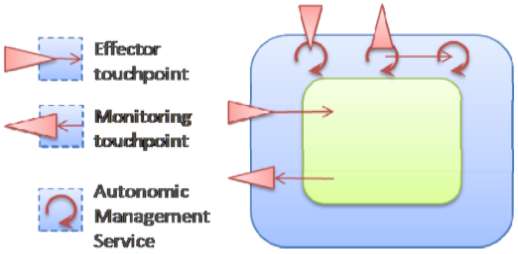

Figura 3.2: *Handlers* fazendo o papel dos *touchpoints*

Foi neste trabalho que alguns dos *handlers* que hoje encontram-se incorporados ao iPOJO foram criados. O primeiro deles é o *handler* de gerenciamento de propriedades, que permite a alteração de algumas propriedades dos componentes, mapeadas em atributos da classe Java que implementa aquele componente. Outro *handler* que foi incorporado ao iPOJO é o *handler* de introspecção (*Architecture Handler*), com o qual é possível obter informação atualizada em tempo de execução a respeito das ligações entre os componentes.

O coração da solução encontra-se, no entanto, no Gerenciador de Dependências, também implementado na forma de um *handler*, que torna possível alterar as propriedades de uma dependência. Dentre as propriedades, encontra-se um filtro que pode ser utilizado para definir qual instância de componente deve ser utilizada para satisfazer a necessidade de um determinado serviço requerido. Através deste filtro, o iPOJO realiza a ligação com a instância desejada.

Por fim, outro *handler* bastante útil definido neste trabalho é o de Gerenciamento de Ciclo de Vida. Este *handler* dá ao usuário a capacidade de instalar, desinstalar, instanciar, iniciar e parar um componente, capacidades antes só possíveis no nível do *bundle*, no contexto do OSGi.

Além destes *handlers*, este trabalho define o mecanismo através do qual o iPOJO realiza a ligação automática entre componentes com serviços requeridos e providos. Este mecanismo faz parte do núcleo do *framework* de componentes do iPOJO, não estando ligado a características não funcionais dos componentes, que é função dos *handlers*.

Através destas contribuições, então, o Autonomic iPOJO provê ao iPOJO funcionalidades úteis no contexto da computação autonômica. Algumas destas funcionalidades implementam capacidades de Reconfiguração Dinâmica como, por exemplo, a alteração das ligações entre componentes e a capacidade de instanciá-los e pará-los em tempo de execução.

No entanto, a decisão a respeito da adaptação fica a cargo do próprio *framework* que tem a autonomia de alterar a ligação entre componentes em tempo de execução, baseando-se na disponibilidade dos componentes e em um mecanismo de ordenação que define a prioridade de cada componente, quando escolhido para uma determinada ligação. Com isso, o usuário não tem o controle da reconfiguração e não pode executar comandos de reconfiguração de forma direta em cima da aplicação. Neste caso, se o usuário quisesse, por exemplo, realizar uma troca de componentes (uma operação simples no contexto de uma reconfiguração), ele teria que tornar o componente indisponível e se assegurar que o novo componente desejado fosse o escolhido dentre os disponíveis. Prover uma forma direta para realizar esta ação não é o objetivo do Autonomic iPOJO.

Em comparação com o A-OSGi (FERREIRA *et al.*, 2009), no entanto, o Autonomic iPOJO se posiciona um pouco mais na direção do suporte à adaptação não antecipada, já que é capaz de refazer a ligação de componentes considerando componentes não existentes em tempo de projeto da aplicação. Além disso, o Autonomic iPOJO não considera um modelo de componentes hierárquico, limitando-se, como o A-OSGi, em uma configuração arquitetural "flat".

3.2 Abordagens inspiradas em Linha de Produtos de Software Dinâmica

Um grupo que publicou muitos trabalhos na definição de *frameworks* para suporte à adaptação inspirados no conceito de Linha de Produtos de Software Dinâmica (HALLSTEINSEN *et al.*, 2008) é o SINTEF[1], uma organização de pesquisa norueguesa.

Em (ALIA *et al.*, 2007), é definida uma abordagem para a adaptação de software executado em dispositivos móveis. A abordagem é baseada em um *middleware* reflexivo sensível ao contexto e em funções de utilidade. Além disso, adota o nível arquitetural, definindo um modelo de componentes hierárquico próprio que inclui alguns metadados a respeito das características do componente. Estes metadados são utilizados, juntamente com o contexto no qual a aplicação se encontra e com as preferências do usuário, para definir a configuração arquitetural que a aplicação

[1] http://www.sintef.no/home/About-us/

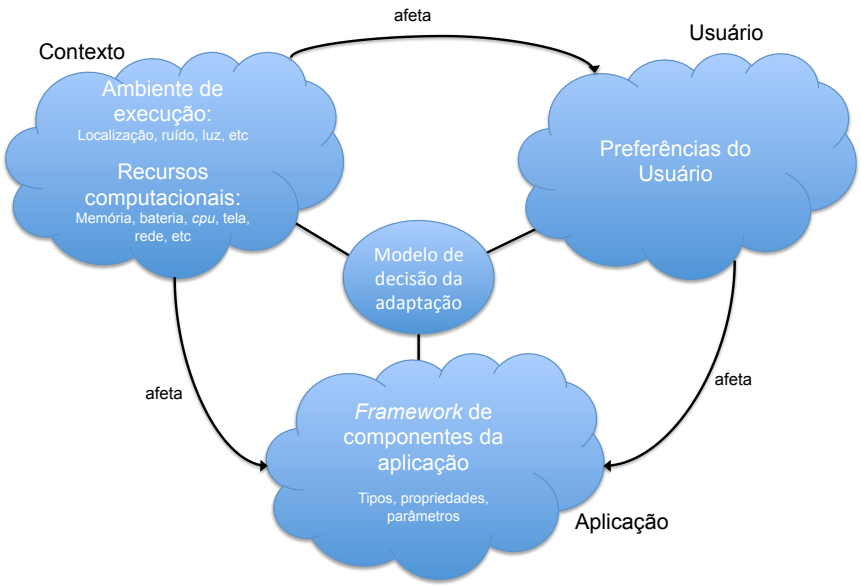

Figura 3.3: Tripé: Usuário, Contexto e Aplicação

deve adotar, através de Funções de Utilidade (WALSH *et al.*, 2004). A Figura 3.3 ilustra os elementos utilizados na definição da adaptação.

Nesta abordagem, uma aplicação é representada por um conjunto de tipos de componentes. Um tipo de componente é uma especificação que é preenchida em tempo de execução pela instância de um componente que possua as características necessárias. Este componente, por sua vez, pode ser formado por vários outros componentes e assim, sucessivamente, de maneira hierárquica. Em tempo de execução, este modelo pode ser representado por uma árvore. Em termos de LPSD, cada tipo de componente representaria um ponto de variabilidade, onde diversos componentes poderiam se encaixar.

Este modelo só se completa em tempo de execução, quando a configuração arquitetural da aplicação é, finalmente, definida. Isto é feito por meio das funções de utilidade que são responsáveis por determinar que componente será escolhido para preencher os tipos de componente da aplicação, dado o contexto de execução, as propriedades de cada componente e as preferências do usuário. Além disso, em tempo de execução, um determinado tipo de componente pode ter a sua implementação (instância) trocada, caso o contexto de execução ou as preferências do usuário sejam alteradas. No nível do modelo, esta troca significaria trocar um ramo da árvore que representa a aplicação.

A avaliação desta abordagem é realizada através de uma prova de conceito, baseada em um Serviço Multimídia Pessoal, utilizando o *framework* QuA (STAEHLI e ELIASSEN, 2002), que implementa o modelo descrito. Este serviço tem o objetivo de otimizar a experiência de assistir a um vídeo ao-vivo a partir de diferentes dispositivos, incluíndo móveis. Para tal, ele constantemente monitora o parâmetro

"disponibilidade de rede" para se adaptar aos diversos cenários possíveis. Baseado neste parâmetro, é escolhido um componente dentre vários que possuem a propriedade "riqueza do dado" em diferentes níveis. O serviço também dá a possibilidade do usuário trocar o dispositivo durante uma transmissão, o que provoca uma adaptação, além de possivelmente, um *re-deploy* de partes dela para o novo dispositivo.

Nesta avaliação, o artigo demonstra os valores das diversas propriedades listadas, levando-se em consideração o contexto (banda) e a preferência do usuário (dispositivo) para determinar a "riqueza do dado" correta a ser utilizada. A partir destes valores, o *framework* define a configuração arquitetural ideal da aplicação, preenchendo os seus pontos de variabilidade visando atender os requisitos do cenários de uso em questão.

Apesar de apresentar uma abordagem focada somente em adaptações antecipadas, este modelo possui uma vantagem em relação aos anteriores que é a utilização de composições de componentes organizados de maneira hierárquica. Neste caso, o modelo prevê uma instância de componente que pode ser formada por um conjunto de tipos de componentes, que possuem suas próprias propriedades e a sua própria função de predição de propriedades.

O grupo utiliza este modelo na implementação de diversos *frameworks*. Um dos *frameworks* que explora esta característica da composição hierárquica de componentes é o MADAM (FLOCH *et al.*, 2006).

O MADAM é um *framework* focado em aplicações móveis, que é uma classe de aplicações que se beneficia bastante das capacidades auto-adaptáveis, já que podem ser executadas em ambientes cujas características mudam constantemente. As principais ideias do MADAM estão reunidas na Figura 3.4.

Quando a aplicação é executada, o *framework* interpreta o modelo especificado e produz uma representação arquitetural da aplicação. Em seguida, analisa as possíveis configurações que atendem à representação arquitetural da aplicação. Este processo é chamado pelos autores de "planejamento". O processo de planejamento é recursivo, pois para cada componente pode haver vários níveis hierárquicos, sendo que no nível mais alto está a composição que representa a aplicação (Figura 3.5). Neste processo, são calculadas todas as alternativas possíveis de configurações, considerando os componentes disponíveis que se encaixam nos tipos definidos na representação arquitetural da aplicação. Isto pode trazer um alto custo para um sistema que é executado em um dispositivo móvel com restrições de recursos e energia limitada por uma bateria.

Uma vez que o processo de planejamento é executado e a configuração arquitetural adequada é escolhida, o *framework* dá início à monitoração do contexto, cujas mudanças podem provocar adaptações na aplicação.

A adaptação em si é realizada através de ações de reconfiguração da aplicação

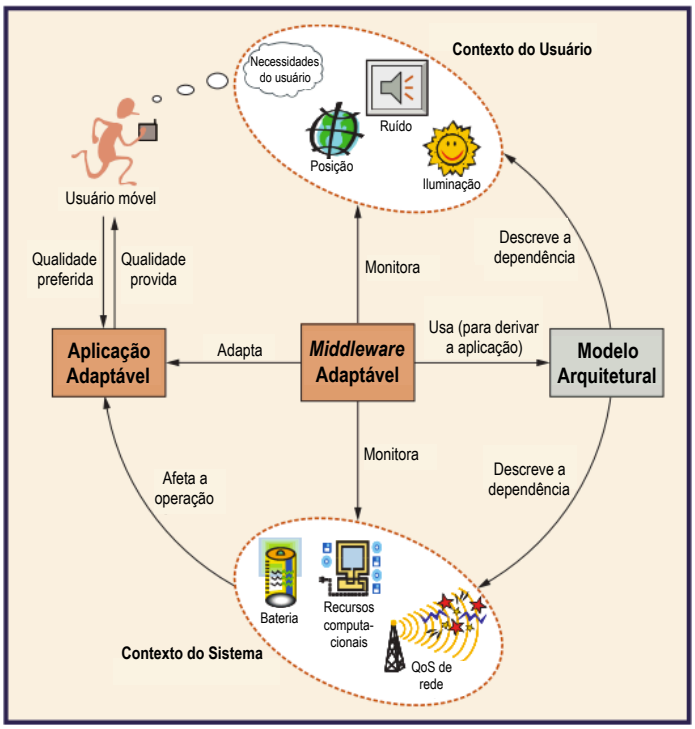

Figura 3.4: Principais ideias do MADAM.

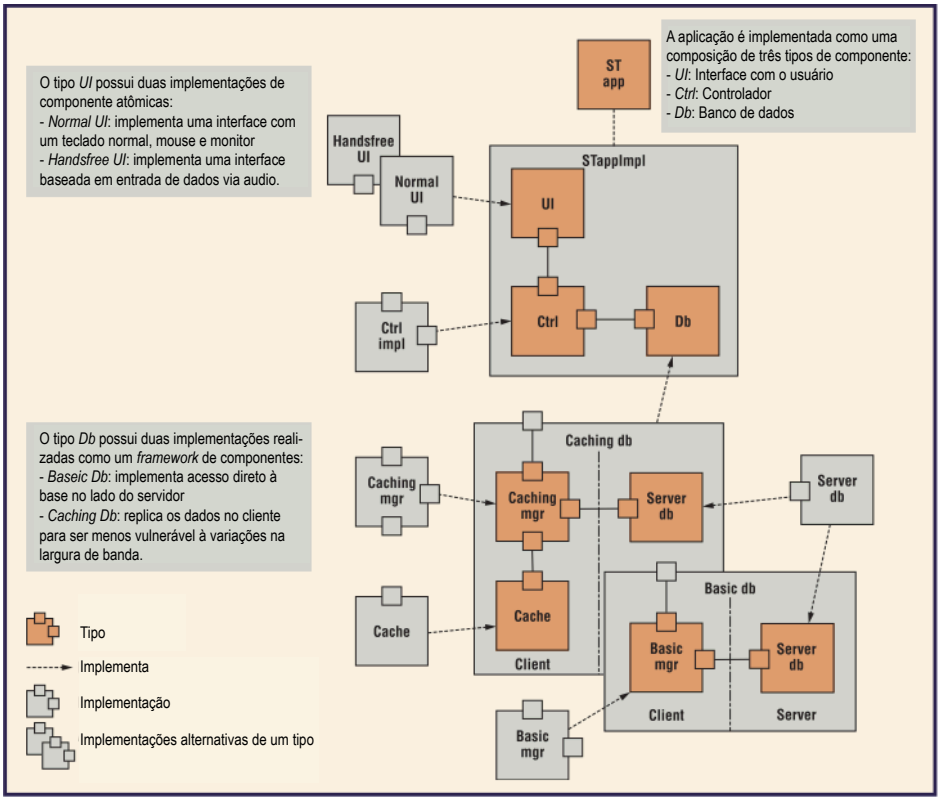

Figura 3.5: Exemplo de uso do MADAM em um sistema móvel utilizado por um técnico de campo.

que podem envolver levar os componentes para um estado "seguro", remover ou instanciar componentes, transferir o estado do componente antigo para o novo, no caso de substituição de instâncias, entre outras. Estas ações são suportadas pelo núcleo do *framework*, que oferece operações de publicação e descobrimento de tipos de componentes e instâncias, instalação e desinstalação de componentes e realizações de ligações entre eles. O núcleo do *framework* ainda oferece acesso à informações de consumo de recursos como memória, CPU e utilização da rede. Todas estas funcionalidades são apresentadas independente de plataforma, porém em um nível mais conceitual. Não fica claro no artigo como esta independência de plataforma é obtida, já que cada plataforma possui suas características e requisitos próprios que dificultam ou facilitam as implementações destas funcionalidades.

Uma característica presente em outras abordagens e ausente no MADAM é a utilização de conceitos SOA. Uma extensão do MADAM criada para suportar estes conceitos é apresentada em (ROUVOY *et al.*, 2008) e (ROUVOY *et al.*, 2009). Nestes artigos, os autores descrevem o MUSIC, um *framework* de componentes orientado a serviço construído em cima dos conceitos utilizados pelo MADAM. O MUSIC preenche, ainda, uma lacuna existente no MADAM, que é a ausência de suporte para aplicações distribuídas.

O MADAM é um *framework* que trabalha basicamente com uma abordagem orientada à qualidade de serviço (QoS). Quando o nível de QoS previamente definido está fora dos padrões, uma nova adaptação é iniciada, visando fazer com que a aplicação volte a atender aos seus requisitos.

No MUSIC, é adicionado o conceito de SLA (*Service Level Agreement*), que está intimamente ligado ao mundo SOA. Nesta extensão, o *framework* deve atender aos SLAs pré-estabelecidos, estando, ainda, compatível com a visão de nível de qualidade de serviço. A Figura 3.6 mostra a arquitetura do MUSIC, enfatizando as mudanças em relação ao MADAM.

Não é objetivo aqui detalhar cada componente descrito na figura. Ressalta-se, no entanto, a diferença entre o MADAM (em cinza claro) e o MUSIC (em cinza escuro). Como pode ser visto, foram adicionados componentes para suportar os conceitos de SLA e Serviços, ligados à Arquitetura Orientada à Serviço.

Um exemplo de utilização do MUSIC é descrito em (ROUVOY *et al.*, 2009), onde a sua implementação foi realizada utilizando-se a plataforma OSGi. Segundo os autores, a utilização do OSGi deu-se pela sua relativa simplicidade, eficiência e portabilidade. Além disso, sua implementação SOA é um dos seus pontos fortes, pois se baseia em chamadas Java diretas, que são bem rápidas, e em um serviço de registro dos serviços disponíveis.

Porém, os serviços OSGi baseados em chamadas Java funcionam somente dentro dos limites da JVM, não sendo adequados para serviços distribuídos. Por este

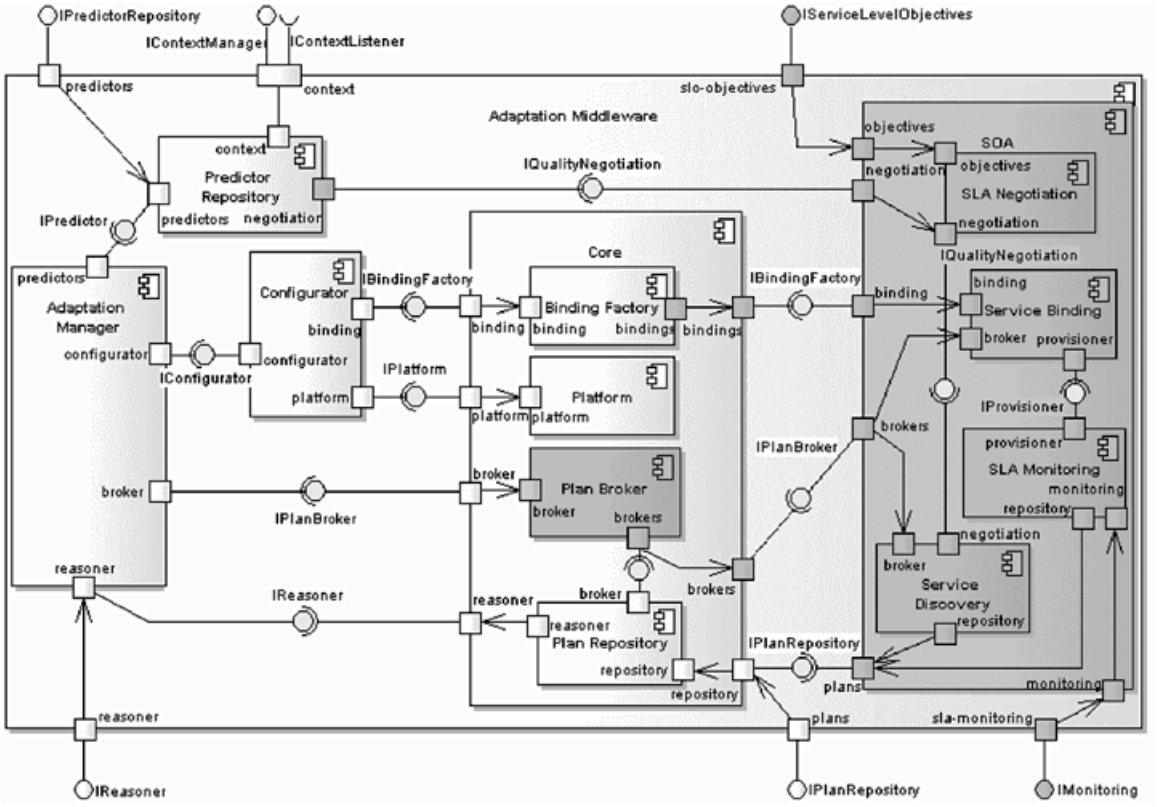

Figura 3.6: Arquitetura do MUSIC enfatizando suas adições (em destaque) em relação à arquitetura do MADAM

motivo, o MUSIC estende o OSGi através do componente "Service Discovery" adicionando a capacidade de comunicação com serviços em diferentes protocolos (não só chamadas diretas à métodos de classes Java).

Novamente, neste trabalho, não são detalhados os aspectos técnicos da execução da adaptação. É descrito apenas que, para a realização da adaptação, os componentes da aplicação são levados a um estado "seguro", onde temporariamente não aceitam requisições, para que possam ser de fato adaptados. Além disso, apesar de citar a transferência de estado entre o componente antigo e o novo, não é detalhado como isto é feito quando a troca é realizada entre componentes simples e compostos, onde, potencialmente, o estado não é compatível. Certamente, esta abordagem traz restrições para a escolha e o projeto destes componentes.

3.3 Abordagens para a Reconfiguração Dinâmica

Nesta seção, são detalhados alguns trabalhos no contexto de Reconfiguração Dinâmica, mais focados em aspectos de execução da adaptação. Em geral, trabalhos que se propõem a definir ações de reconfiguração dinâmica, ou que utilizam tais ações, estão fortemente relacionados à um *framework* de componentes, capaz de implementá-las.

Nesta linha, BURES *et al.* (2006) definem o modelo de *framework* de componentes SOFA 2.0. Segundo os autores, a principal motivação do trabalho foi que os atuais *frameworks* de componentes sofrem de pelo menos um dos seguintes problemas:

1. um suporte limitado para a reconfiguração dinâmica;
2. falta de estrutura para a parte de controle do componente, separando a lógica do negócio da lógica do gerenciamento do componente; e
3. ausência de suporte para múltiplos estilos de comunicação (*pipes and filters*, *shared memory*, troca de mensagens, etc).

Em termos de reconfiguração dinâmica, os autores se colocam contra a abordagem de alterações "descontroladas" na topologia das composições de componentes, pois segundo eles este tipo de prática pode levar à uma "erosão arquitetural". Por isso, permitem apenas 3 padrões de reconfiguração dinâmica (HNĚTYNKA e PLÁŠIL, 2006):

1. Fábrica Aninhada (*Nested Factory*): Este padrão define a adição de um componente à uma composição, além de especificar sua conexão com outros componentes;
2. Remoção de Componente (*Component Removal*): Este padrão define a remoção de um componente de uma composição; e
3. Interface de Utilidade (*Utility Interface*): Este padrão permite a quebra do encapsulamento criado pela composição. Através desta interface, um componente interno à uma composição pode se conectar com um componente externo sem que esta conexão passe pela composição.

Em particular, o 3º padrão abre precedente para que a arquitetura de uma aplicação seja completamente definida usando estas interfaces e, segundo os autores, isto faz com que a arquitetura SOA seja um caso específico deste modelo de componentes.

O SOFA 2.0 define, ainda, um componente de maneira muito semelhante ao iPOJO. Para o SOFA 2.0, um componente é formado por um *frame* que gerencia os aspectos não funcionais, além de arquitetura do componente, que é um conjunto de sub-componentes. Os aspectos não funcionais são implementados por micro-componentes (MENCL e BURES, 2005), semelhantes aos *handlers* do iPOJO, extensíveis e plugáveis no *frame*.

Neste sentido, um *frame* é capaz de gerenciar tanto interfaces inerentes à logica de negócio da aplicação, expondo características funcionais, quanto interfaces

de controle, que são dedicadas aos aspectos não funcionais. Interfaces de controle provêem operações ligadas ao gerenciamento do ciclo de vida, reconfiguração e introspecção da aplicação. Os autores defendem, ainda, que as interfaces de controle devem ser modeladas separadamente e a aplicação deve ter algum tipo de acesso a estas interfaces.

Outro aspecto interessante do SOFA 2.0 é a utilização de conectores como entidades de primeira classe. Estas entidades implementam a comunicação entre os componentes e é através delas que diferentes estilos de comunicação podem ser utilizados. Os estilos de comunicação podem ser de quatro tipos: invocação de método, troca de mensagems, *streaming* e memória compartilhada.

O modelo de componentes do SOFA 2.0 é formalizado através de um metamodelo baseado no MOF[2]. Isto traz algumas vantagens como, por exemplo, a geração automática de um repositório com interfaces padronizadas baseado em XML, utilização de ferramentas já existentes para a edição e instanciação do modelo, etc. A especificação gerada por este metamodelo é utilizada tanto em tempo de projeto quanto em tempo de execução. É importante ressaltar que o modelo de componentes do SOFA 2.0 suporta plenamente composições hierárquicas, criando o conceito de componente e de sub-componente.

Uma reconfiguração dinâmica para o SOFA 2.0 é definida, em termos gerais, como uma modificação arbitrária da estrutura da aplicação. Neste sentido, um caso específico de reconfiguração dinâmica é a atualização dinâmica de um componente, definida como a simples troca de um componente por outro que implemente as mesmas interfaces. Este tipo de ação é, inclusive, o que é feito no Autonomic iPOJO. É, no entanto, uma reconfiguração dinâmica de fato, pois é realizada a troca de uma parte da aplicação em tempo de execução.

A execução da reconfiguração no SOFA 2.0 é suportada por um *framework* de componentes dinâmico e distribuído. Em cada nó existe um *SOFAnode*, que é composto por uma JVM e o ambiente de execução do SOFA 2.0. A ligação entre componentes em nós diferentes é realizada através de uma conexão via "Utility Interface", que implementa conceitos SOA. Cada *SOFAnode* conta também com um repositório que contém as definições dos componentes e conectores.

Em termos de controladores (que implementam as interfaces de controle dos componentes), o SOFA 2.0 conta com dois principais: o gerenciador de ciclo de vida (que é capaz de iniciar, parar e atualizar o componente) e o gerenciador de ligações (que adiciona/remove ligações entre componentes).

Até a publicação do artigo, o ambiente de execução do SOFA 2.0 ainda não havia sido completamente construído.

O SOFA 2.0 é um *framework* de componentes acadêmico (como os autores o de-

[2]http://www.omg.org/mof/

finem) e, em comparação com os trabalhos anteriores, não apresenta um mecanismo completo para o ciclo MAPE-K; seu foco é no suporte às ações de reconfiguração. Possui a vantagem de suportar sistemas distribuídos, no entanto, oferece um conjunto de ações de adaptação limitado por acreditar que muita flexibilidade traria uma "erosão arquitetural". Uma abordagem onde as ações de reconfiguração sejam refletidas no modelo arquitetural do sistema resolveria este problema, sem necessariamente limitar as possibilidades de reconfiguração.

Além disso, ao prover uma maneira de realizar uma ligação entre níveis hierárquicos diferentes do sistema sem realizar a promoção das interfaces correspondentes para o mesmo nível hierárquico, através da "Utility Interface", o *framework* quebra o encapsulamento definido pelo "princípio caixa-preta", definido no Capítulo 2 (Seção 2.1). Isto faz com que sua representação arquitetural deixe de ser uma árvore para ser um grafo mais genérico que pode, inclusive, ter laços fechados. Esta característica pode trazer como consequência um aumento na dificuldade da reconfiguração de componentes compostos que contenham vários níveis hierárquicos.

Em termos de ações de reconfiguração arquitetural, apesar da maioria dos trabalhos atrelarem estas ações à um *framework* de componentes, KETFI *et al.* (2002) realizam um interessante trabalho de levantamento da área, propondo taxonomias para a reconfiguração dinâmica aplicada à adaptação de sistemas, sem necessariamente fazê-las depender de um determinado *framework*.

Inicialmente, os autores definem quatro motivos que levam à uma adaptação:

1. **Adaptação corretiva**: Este tipo de adaptação visa corrigir problemas encontrados no sistema adaptável;

2. **Adaptação adaptativa**: Mesmo quando um sistema executa corretamente, o seu ambiente de execução pode ser alterado e isto pode provocar a demanda para a adaptação do sistema. Este tipo de adaptação visa tratar este cenário;

3. **Adaptação estensiva**: Neste cenário, o sistema adaptável deve receber novas funcionalidades para atender a novos requisitos em tempo de execução;

4. **Adaptação evolutiva**: Por fim, este tipo de adaptação refere-se à alterações no sistema com o objetivo de melhorar uma funcionalidade já existente.

Em seguida, os autores apresentam uma taxonomia da adaptação, listando os tipos de operação mais importantes em termos de reconfiguração dinâmica:

1. **Adaptação na arquitetura da aplicação**: Operações de adaptação arquiteturais podem ser divididas em 3:

- Adição de um novo componente: esta operação consiste na instanciação de um componente e sua posterior ligação com outros componentes já existentes na aplicação. A instanciação do componente pode ainda acontecer em dois cenários: o tipo de componente já existe carregado na aplicação ou o tipo de componente não está disponível na aplicação. No segundo caso, o tipo de componente deve ser carregado de um repositório, ou criado dinamicamente. Além disso, ao ser adicionado à uma aplicação em execução, o componente deve levar em consideração o estado corrente da aplicação, que não será necessariamente o seu estado inicial, e se adaptar para manter a consistência da aplicação como um todo;
- Remoção de um componente já existente: a remoção de um componente existente não deve afetar a execução dos outros componentes. Além disso, o componente deve estar em um estado estável (ou quiescente) para ser removido com segurança.
- Modificação das ligações entre componentes: ao realizar a troca da ligação entre componentes, é necessário considerar a compatibilidade dos tipos de interfaces em questão, além de ter o cuidado necessário para que não sejam perdidas mensagens nesta operação.

2. **Adaptação da implementação do componente**: Este tipo de operação define a troca de uma instância de componente por outra equivalente em termos de interfaces providas e requeridas. Geralmente, é motivada por problemas de performance ou por *bugs* na implementação antiga.

3. **Adaptação da interface do componente**: Esta operação modifica a lista de serviços providos/requeridos pelo componente através da adição/remoção de uma interface da lista de interfaces do componente.

4. **Adaptação da geografia do componente**: Esta operação corresponde à migração de um componente de um ambiente de execução para outro. Isto não afeta a arquitetura do sistema, mas a comunicação entre o componente migrado e outros componentes pode ser alterada. Por exemplo, dois componentes que anteriormente compartilhavam a mesma JVM podem ter que mudar o tipo de conector para que continuem se comunicando quando um deles for migrado para outra JVM.

Apesar de não depender necessariamente de uma determinada implementação de um *framework* de componentes, os detalhes de execução destas ações dependem intrinsecamente de um *framework* que defina os conceitos arquiteturais utilizados. Este modelo é definido pelos autores como uma extensão do CCM[3].

[3] *Corba Component Model* - http://www.omg.org/spec/CCM/

Por fim, outro trabalho que define um *framework* de componentes que provê suporte para a Reconfiguração Dinâmica é o FraSCAti (SEINTURIER *et al.*, 2012). Neste trabalho, o FraSCAti é definido como um "modelo de execução dinamicamente reconfigurável para SOA". Para isso, os autores partiram da especificação SCA[4] e realizaram as seguintes extensões para preencher suas principais lacunas:

1. Prover capacidades de gerenciamento da configuração de um componente em tempo de execução;

2. Prover suporte para características não funcionais dos componentes (por exemplo: segurança, gerenciamento de transações, etc), geralmente atreladas à uma plataforma;

3. Prover um gerenciamento do ciclo de vida de um componente; e

4. Prover capacidades de gerenciamento da plataforma por si só (por exemplo, ser capaz de gerenciar erros, segurança e performance em um ambiente SOA distribuído).

A especificação SCA possui algumas características interessantes como, por exemplo, a definição de um modelo de componentes hierárquico distribuído orientado a serviço. A entidade principal deste modelo é o componente, que pode ser simples ou composto (formado por outros componentes). Componentes são capazes de prover e requerer serviços e expor propriedades. Os serviços requeridos são chamados de referências e tanto eles quanto os serviços providos, se localizados em uma composição, podem ser promovidos para o nível hierárquico superior. A Figura 3.7 ilustra estes conceitos. Outra característica interessante é que a especificação é agnóstica de linguagens de programação e de protocolos de comunicação.

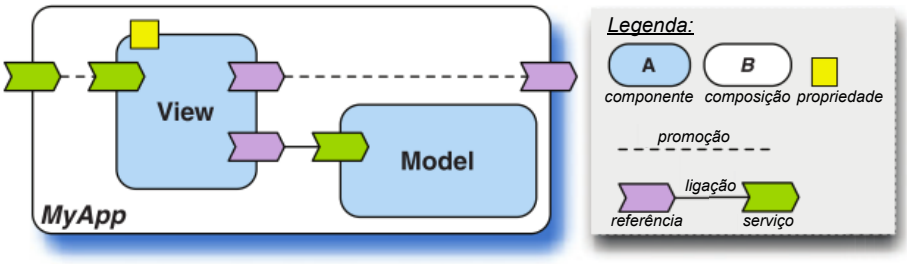

Figura 3.7: Arquitetura orientada a componentes da especificação SCA

A definição de uma aplicação SCA é feita por meio de uma descrição em XML. Esta descrição deve conter todas as características da aplicação em termos de componentes, serviços requeridos, providos e a organização hierárquica dos componentes.

[4] *Service Component Architecture* - http://www.oasis-opencsa.org/sca

O acesso à serviços SCA também pode ser feito utilizando diversas tecnologias diferentes. Existem, inclusive, meios de consumir serviços providos por um ambiente de execução OSGi.

Buscando aproveitar estas qualidades e preencher as lacunas descritas anteriormente, os autores do FraSCAti buscaram juntar a especificação SCA com o modelo de componentes Fractal (BRUNETON *et al.*, 2006). Fractal é um modelo de componentes reflexivo e distribuído, agnóstico de linguagem de programação, criado para a construção de sistemas altamente configuráveis. A principal característica do Fractal é a sua capacidade de customizar a política de execução associada à um componente, chamada de "personalidade".

A implementação de uma personalidade é realizada através de "controladores" e "interceptadores". Estas entidades são responsáveis pela separação entre a lógica de negócio do componente e a sua lógica de gerenciamento dentro do *framework*. Cada controlador cuida de um tipo de capacidade não funcional como, por exemplo, o gerenciamento do ciclo de vida e o gerenciamento de ligações. Já os interceptadores se colocam entre o componente e o mundo externo para interceptar chamadas e realizar ações de acordo com o necessário como, por exemplo, uma escrita em log ou o gerenciamento de uma transação.

O FraSCAti possui algumas ferramentas que ajudam na tarefa de reconfigurar dinamicamente o sistema. Duas delas, herdadas do Fractal, são o FScript e o FPath (DAVID *et al.*, 2009). O FScript[5] é uma ferramenta que facilita o uso da API de reconfiguração dinâmica do Fractal, através da execução de sequências de ações de reconfiguração. Já o FPath é uma DSL[6], embutida no FScript, que provê um meio de realizar a introspecção de arquiteturas Fractal. Com o FPath, é possível localizar os elementos de interesse dentro da arquitetura para que os comandos FScript possam atuar em cima deles.

Em termos de reconfiguração dinâmica, o FScript oferece um conjunto bem extenso de comandos: é capaz de adicionar/remover componente de uma composição, criar novas composições, realizar a ligação entre componentes, alterar propriedades de componentes, exportar/importar interfaces para/de um nível hierárquico superior de um componente composto e transferir o estados compatíveis entre componentes que foram trocados. Além disso, possui comandos específicos para adicionar/remover uma "intenção" (que é uma característica não funcional do componente), e adicionar/remover um "binding" (que são ligações remotas, entre ambientes de execução diferentes do FraSCAti).

[5] http://fractal.ow2.org/fscript/

[6] *Domain Specific Language* - Linguagem Específica de Domínio

3.4 Considerações Finais

Neste capítulo, foram apresentados alguns trabalhos relacionados nas linhas de Sistemas Auto-Adaptáveis e Reconfiguração Dinâmica, com o objetivo de contextualizar o trabalho desenvolvido nesta dissertação.

Ao estudar os trabalhos destas áreas, percebe-se uma divisão entre trabalhos que seguem uma abordagem autonômica, outros que se baseiam nas ideias de Linha de Produtos de Software Dinâmica e, por fim, trabalhos cujo foco está em prover meios de realizar a adaptação, sem se preocupar com a implementação de um ciclo completo de adaptação. Estes últimos estão mais relacionados com os objetivos do trabalho descrito neste dissertação.

Em termos gerais, baseado nos trabalhos apresentados, podemos dizer que as abordagens inspiradas em conceitos de LPSD destacam-se por apresentar um alto nível de controle da adaptação, ou seja, modelam aspectos da decisão da adaptação levando-se em consideração os requisitos do usuário, o contexto de execução e as características dos componentes, que podem ser organizados hierarquicamente. Por outro lado, qualquer alteração neste mecanismo deve ser feita sob medida para a aplicação em questão. Oferecem, deste modo, um baixo nível de suporte para a adaptação não antecipada.

Abordagens examinadas que foram inspiradas no ciclo MAPE-K implementam um ciclo de adaptação em um nível mais baixo, baseando-se nas características técnicas do componente como, por exemplo, o tipo da interface utilizada. Além disso, não oferecem capacidades de composição de componentes de maneira hierárquica. Por outro lado, como o foco destas abordagens é nas características técnicas do componente, novos componentes podem ser adicionados em tempo de execução, o que as confere um certo grau de suporte à adaptação não antecipada.

Por fim, abordagens de reconfiguração dinâmica, apesar de não buscarem implementar um ciclo completo de adaptação, focam, principalmente, em prover meios para que a adaptação seja realizada de maneira controlada. Estas abordagens apresentam um conjunto bem completo de ações de reconfiguração, geralmente em cima de modelos de componentes sofisticados. Deste modo, são mais adequadas para lidar com a adaptação não antecipada.

A Tabela 3.1 compara as principais características dos trabalhos apresentados. Nesta tabela foi incluído o iPOJO, descrito no Capítulo 2, que é a plataforma sobre a qual o trabalho descrito nesta dissertação foi desenvolvido, com o objetivo de pavimentar a discussão sobre as suas contribuições, nos próximos capítulos.

Nesta tabela, a primeira coluna contém as principais características consideradas na comparação. Uma destas características, que encontra-se mais detalhada, são as ações de reconfiguração suportadas. Para estas ações, os trabalhos analizados foram

Tabela 3.1: Resumo das características dos trabalhos apresentados

		Autonômicas		LPSD		RD			
Características		**A-OSGi**	**Autonomic iPOJO**	**MADAM**	**MUSIC**	**SOFA 2.0**	**(KETFI *et al.*, 2002)**	**FraSCAti**	**iPOJO/OSGi**
Modelo de componentes hierárquico		não	não	sim	sim	sim	não	sim	sim
Modelo orientado à serviços		sim	sim	não	sim	sim	não	sim	sim
Suporte à sistemas distribuídos		não	não	não	sim	sim	sim	sim	sim†
Ações suportadas	Criar composição					✓✓		✓✓	✓
	Adicionar componente	✓✓	✓✓	✓	✓	✓✓	✓	✓✓	✓
	Remover componente	✓✓	✓✓	✓	✓	✓✓	✓	✓✓	✓
	Transferir estado			✓	✓			✓✓	
	Alterar propriedade	✓✓	✓✓					✓✓	✓
	Alterar ligação	✓✓	✓‡	✓	✓	✓✓	✓	✓✓	✓
	Adicionar interface					✓✓	✓	✓✓	✓
	Remover interface					✓✓	✓	✓✓	✓
	Alterar conector						✓		

† O suporte à sistemas distribuídos é conseguido por meio do DOSGi (acessível em: http://cxf.apache.org/distributed-osgi.html), uma extensão para o OSGi, plataforma sobre a qual o iPOJO é executado.

‡ A ligação entre componentes pode ser alterada indiretamente, por meio de uma propriedade do componente que requer o serviço.

contemplados com nenhum, um ou dois sinais de visto, cujo significado é detalhado a seguir, juntamente com uma análise de cada uma das características escolhidas.

- Nenhum sinal de visto: neste caso, o *framework* não apresenta suporte à ações de reconfiguração em questão.

- Um sinal de visto (✓): representa suporte limitado à ação ou suporte somente em tempo de inicialização, ou seja, o *framework* instancia uma representação do componente realizando a ação, mas em tempo de execução esta ação não pode ser utilizada para alterar a aplicação.

- Dois sinais de visto (✓✓): nota máxima utilizada, representa o suporte total à ação em questão. Neste caso, o usuário pode executar a ação de maneira independente do momento em que se encontra (inicialização ou execução), através de uma API fornecida para este fim ou de comandos pré-existentes disponíveis no ambiente de execução.

O custo de se modelar o contexto e as preferências do usuário e aplicá-los à decisão da configuração arquitetural ideal trazem para o MADAM e o MUSIC a consequência de não suportarem bem uma adaptação não antecipada. Já o A-OSGi,

Autonomic iPOJO e iPOJO possuem algum suporte à adaptação não antecipada, já que novos componentes podem ser inseridos à aplicação no lugar de componentes já existentes. Estes novos componentes podem conter novas funcionalidades não previstas em tempo de projeto. Já as abordagens baseadas em reconfiguração dinâmica focam nos mecanismos básicos de execução da adaptação, suportando intrinsecamente a inserção de novas funcionalidades não previstas em tempo de projeto.

O suporte à um **modelo de componentes hierárquico** não é pré-requisito para a reconfiguração dinâmica. A arquitetura orientada à serviço, por exemplo, tão utilizada hoje em dia, não suporta composição hierárquica de serviços da maneira como foi definida no Capítulo 2 desta dissertação, já que não cria um encapsulamento dos serviços de mais baixo nível utilizados para compor um serviço de mais alto nível. Dentre as abordagens analisadas, esta característica está bem distribuida entre os grupos, não sendo possível identificar nenhuma tendência.

As abordagens que adotam um **modelo de componentes orientado à serviços** tentam se aproveitar do seu alto nível de desacoplamento, característica muito útil para a reconfiguração dinâmica de uma aplicação. A maioria dos trabalhos considera isto em seus modelos de componentes. Apenas o MADAM, que é um precursor do MUSIC, e a taxonomia implementada em um modelo baseado no CCM de KETFI *et al.* (2002) não suportam conceitos de orientação à serviços.

O **suporte à sistemas distribuídos** é uma característica importante em um *framework* de componentes dinâmico, pois sistemas complexos podem ser divididos em vários ambientes de execução para ter seu desempenho melhorado. Neste sentido, notamos uma tendência de maior suporte desta característica nas abordagens mais próximas à reconfiguração dinâmica.

Em termos de ações suportadas, era de se esperar que as abordagens da linha de reconfiguração dinâmica se mostrassem mais completas do que as demais, o que de fato ocorreu. Podemos diferenciar aqui, entretanto, o nível de suporte às ações e a capacidade das abordagens de prover ao usuário/programador um controle sobre a adaptação. Como já foi dito, o número de sinais de visto mostram um maior grau de suporte à estas ações. Um destaque especial merece ser dado ao FraSCAti, que foi a abordagem que se mostrou mais flexível neste ponto. A única ação que ela não implementa é a alteração do conector em tempo de execução, pois o conector não é tratado como uma entidade de primeira classe, muito embora o *framework* suporte vários tipos de protocolos de comunicação através das suas várias "personalidades".

Ainda em relação ao suporte à ações de reconfiguração, uma característica muito importante não encontrada em nenhum dos trabalhos analisados neste capítulo, é a verificação da aplicabilidade da ação no estado corrente do sistema. Em um sistema verdadeiramente dinâmico, eventos externos podem alterar o estado do sistema durante a decisão da adaptação e posterior produção do seu plano de reconfiguração.

Neste caso, é necessário verificar se a ação determinada ainda possui precondições verdadeiras, antes de executá-las, sob pena de levar o sistema a uma configuração desconhecida e, potencialmente, inválida para o usuário.

Um comentário importante é em relação ao iPOJO. Diferentemente do Autonomic iPOJO, o seu modelo de componentes prevê composições hierárquicas de componentes, instanciadas a partir de uma descrição da composição realizada em tempo de projeto. No entanto, não oferece uma maneira de alterar a topologia destas composições em tempo de execução.

Por fim, uma característica presente nos três trabalhos classificados como abordagens para a reconfiguração dinâmica é que todos eles são iniciativas acadêmicas, sem forte presença na indústria. Apesar de estar atrás no quesito suporte à reconfiguração dinâmica, a plataforma iPOJO/OSGi, entretanto, é utilizada neste trabalho por conta do seu grau de maturidade, pelo seu nível de utilização tanto na academia quanto na indústria e, consequentemente, pela sua grande comunidade de usuários, que oferece uma extensa base de conhecimento por meio de fóruns, emails, blogs, etc. Nenhum outro trabalho, dentro das analizados, consegue reunir estas características no nível obtido pela plataforma iPOJO/OSGi.

No próximo capítulo, as contribuições desta dissertação serão detalhadas. Elas visam prover ao iPOJO capacidades de reconfiguração dinâmica, que são utilizadas na implementação de um mecanismo para a execução de planos de reconfiguração arquitetural de aplicações desenvolvidas sobre a plataforma iPOJO/OSGi.

Capítulo 4

Um Mecanismo para a Execução de um Plano de Reconfiguração Arquitetural para iPOJO/OSGi

Após a descrição dos fundamentos teóricos necessários ao entendimento deste trabalho e da apresentação de alguns trabalhos relacionados, este capítulo visa detalhar o trabalho de pesquisa em si e seus resultados.

Como colocado no Capítulo 1, após um levantamento na área de Sistemas Auto-Adaptáveis, este trabalho concentrou-se na execução da adaptação, utilizando o OSGi como plataforma. A execução da adaptação pode ser conseguida através de técnicas de Reconfiguração Dinâmica, área para a qual a pesquisa foi direcionada.

Utilizando técnicas de Planejamento (Seção 2.2), a execução da adaptação pode ser vista como a execução da um Plano de Reconfiguração Arquitetural (DI BENEDITTO e WERNER, 2012). A realização desta atividade em sistemas altamente dinâmicos traz diversos desafios como, por exemplo, a necessidade de consideração de eventos externos com potencial para a alteração da configuração arquitetural, concorrendo com a execução do plano, e o tratamento do erro de reconfiguração. Estas duas questões são tratadas ao longo deste trabalho.

Um plano de reconfiguração é formado por ações de reconfiguração diretamente relacionadas às capacidades do *framework* de componentes utilizado. No caso deste trabalho, o OSGi, estendido pelo *framework* de componentes iPOJO, provê alguma capacidade de reconfiguração dinâmica, porém não suficientes para a execução de um plano de reconfiguração arquitetural. Sendo assim, para que seja possível atingir o objetivo de execução de um plano de reconfiguração arquitetural no iPOJO, é necessário estendê-lo, adicionando as capacidades de reconfiguração necessárias. Esta extensão também é tratada neste trabalho.

Este capítulo está dividido da seguinte maneira: a Seção 4.1 detalha um pouco

mais o contexto onde este trabalho se encontra, explorando os seus objetivos. Em seguida, a Seção 4.2 define os requisitos deste trabalho para as duas linhas de pesquisa identificadas: execução do plano de reconfiguração arquitetural e extensão das capacidades de reconfiguração dinâmica do iPOJO. Posteriormente, as Seções 4.3 e 4.4 descrevem os detalhes de implementação destas duas linhas. Finalmente, a Seção 4.5 destaca as considerações finais deste capítulo.

4.1 Visão geral e contextualização do problema

Como visto no Capítulo 3, existem várias abordagens propostas pela comunidade de engenharia de *software* para a execução da adaptação através da reconfiguração dinâmica do *software* adaptável.

Em um trabalho anterior do grupo (DI BENEDITTO e WERNER, 2012), os autores abordam a utilização de uma estratégia declarativa, baseada em planejamento automatizado (Seção 2.2), para adaptar o sistema através de um plano de reconfiguração arquitetural, utilizando o planejador JSHOP2[1]. Nesta abordagem, uma reconfiguração dinâmica é modelada como uma transição de estado em um Sistema de Transições de Estado. Cada estado representa uma determinada configuração arquitetural do sistema e cada transição de estado representa uma ação de reconfiguração dinâmica, como mostra a Figura 4.1.

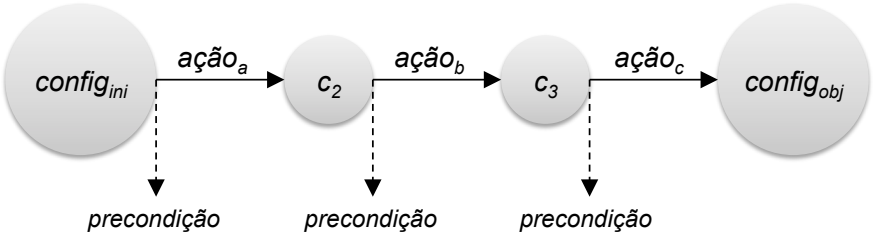

Figura 4.1: Um sistema de transições de estado representando as transições entre as configurações arquiteturais de um sistema adaptável

Um plano que tem o objetivo de levar o sistema da configuração inicial $config_{ini}$ para a configuração objetivo $config_{obj}$ possui, geralmente, configurações arquiteturais intermediárias. As ações que provocam as transições de estado devem ser realizadas somente se as suas precondições forem verdadeiras.

O trabalho propõe ainda o modelo conceitual de um controlador, entidade reponsável por produzir e executar um plano de reconfiguração arquitetural que levará o sistema até a sua configuração objetivo ou final, a partir da sua configuração arquitetural inicial. A Figura 4.2 mostra o controlador e sua estrutura interna.

[1]http://sourceforge.net/projects/shop/files/JSHOP2

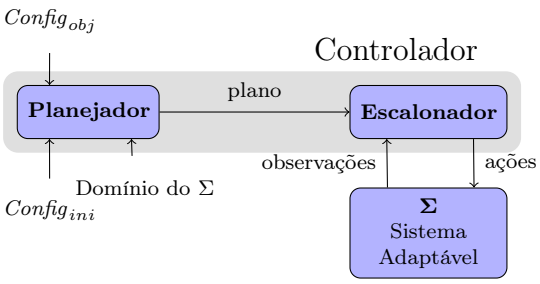

Figura 4.2: Diagrama mostrando o Controlador e o Sistema Adaptável (Σ)

O Controlador é dividido em duas entidades internas, o Planejador e o Escalonador. Estas duas entidades focam nas atividades Planejar e Executar do ciclo MAPE-K. O Controlador parte do princípio que existe outra entidade anterior que decidirá qual é o estado final para o qual se deseja levar o sistema adaptável. Uma vez que o estado final é determinado, o planejador busca um plano de reconfiguração, se ele existir, levando em consideração o domínio (Seção 2.2) e a configuração inicial do sistema que, a cada ciclo de adaptação, é a sua configuração arquitetural corrente.

O Escalonador, por sua vez, tem a responsabilidade de executar o plano de reconfiguração definido. Para isso, ele precisa executar cada ação observando suas precondições. No entanto, dado que o sistema é dinâmico e está exposto à eventos externos, determinada precondição que era verdadeira no momento da elaboração do plano pode não ser verdadeira no momento da sua execução e o Escalonador precisa levar isto em consideração. Uma abordagem é realizar uma verificação das precondições de cada ação no momento da sua execução, evitando, em caso de eventos externos, levar o sistema para um estado desconhecido e, possivelmente, inválido.

Isto, no entanto, não garante a previsibilidade da execução do plano, já que um sistema dinâmico sempre estará exposto à eventos externos. Por isso, o Escalonador deve contar com uma estratégia a ser seguida em casos de erro de reconfiguração.

O *framework* reflexivo utilizado para construir tal mecanismo é o iPOJO, executado em cima do *middleware* OSGi. Como colocado nos capítulos anteriores, o OSGi foi escolhido por conta da sua extensa utilização tanto na academia quanto na indústria e o iPOJO por ser o *framework* de componentes mais completo para OSGi, dentre os analisados.

Apesar de estender o OSGi, provendo um modelo de componentes hierárquico e orientado a serviço, o iPOJO possui uma limitação que impossibilita a implementação do Escalonador: o seu modelo de componentes não é efetivamente dinâmico para componentes compostos. Ou seja, ele não permite o nível desejado de reconfiguração em tempo de execução. Com o modelo de componentes atual do iPOJO, é possível

criar composições, descrevendo-as através de XML ou mesmo instanciando-as através de uma API, mas não é possível modificar uma composição já existente em tempo de execução de formas não previstas em tempo de projeto. Além disso, suas capacidades de reconfiguração atuais não são explícitas, ou seja, não dão ao usuário ou ao desenvolvedor a possibilidade de realizar alterações na configuração arquitetural do sistema de forma direta. Para que a ligação entre componentes iPOJO seja efetuada, é necessário que operações no nível do *bundle* sejam executadas, disponibilizando ou indisponibilizando componentes iPOJO.

Sendo assim, considerando os desafios identificados, podemos dividir este trabalho de pesquisa em duas linhas principais:

1. Investigação e determinação de uma estratégia para evitar a execução de ações de reconfiguração inválidas, considerando um mecanismo para recuperação de erros de reconfiguração; e

2. Determinação de um conjunto de capacidades de reconfiguração dinâmica a serem adicionadas ao iPOJO para formação da base necessária para a execução do plano de reconfiguração arquitetural.

As próximas seções discutem estas duas linhas e definem os requisitos necessários para a implementação das soluções.

4.2 Elaboração dos Requisitos

Esta seção discute questões relativas às duas linhas identificadas anteriormente e define, ao longo da discussão, os requisitos necessários para a fase de implementação das soluções elaboradas. Cada linha é discutida separadamente nas próximas seções.

4.2.1 Mecanismo para a Execução de um Plano de Reconfiguração Arquitetural - Escalonador

Para que seja possível executar um plano de reconfiguração arquitetural na plataforma OSGi/iPOJO, um mecanismo de execução é proposto. Este mecanismo se apoia nas capacidades de reconfiguração dinâmica oferecidas pela plataforma citada e considera as questões relativas à execução de um plano de reconfiguração em um ambiente altamente dinâmico, que é a primeira linha a ser investigada.

Este mecanismo é uma instanciação do Escalonador da Figura 4.2, sendo responsável por executar o plano de reconfiguração gerado pelo Planejador. Podemos, então enunciar o seu primeiro requisito:

R1 O Escalonador deve ser capaz de executar um plano de reconfiguração arquitetural gerado pelo Planejador.

O Planejador utilizado neste trabalho produz planos como o da Listagem 4.1, que são gravados em um arquivo texto. O Escalonador deve, então, ser capaz de ler estes arquivos e extrair as ações de reconfiguração arquitetural que nele constam.

Listagem 4.1: Exemplo de um plano de reconfiguração gerado pelo JSHOP2

```
1 (!install componente)
2 (!add-new instancia componente)
3 (!remove instanciaAntiga)
```

Outra funcionalidade deste Escalonador é a obtenção da configuração arquitetural inicial necessária para a geração do plano pelo Planejador. O Escalonador é um bom lugar para esta funcionalidade, já que estará bem próximo ao sistema adaptável, observando-o constantemente.

R2 O Escalonador deve ser capaz de informar ao Planejador a configuração aquitetural corrente do sistema, que será usada para produzir o plano de reconfiguração.

Para que o Escalonador suporte a execução de um plano de reconfiguração em um sistema dinâmico real, ou seja, exposto à eventos externos que podem alterar o seu estado (configuração arquitetural) a qualquer momento, é necessário entender melhor os estes eventos e o momento no qual eles podem acontecer em relação à execução do plano.

Podemos, de maneira simplista, classificar os eventos externos em "catastróficos", que impedem a execução do sistema como um todo, e "não catastróficos", que indisponibilizam somente parte do sistema. Este último é o foco deste trabalho. Estes eventos alteram o estado do sistema e se traduzem para o ponto de vista arquitetural, como uma possível indisponibilização de parte dos componentes que formam a sua arquitetura. Isto pode levar o sistema à uma configuração arquitetural inválida, deixando-o não disponível, ou pode apenas alterar a sua configuração, deixando-o ainda disponível para requisições, mas com parte das suas funcionalidades desabilitadas.

Em relação ao momento onde um evento externo acontece, este pode ser dividido em:

- Antes da produção do plano: neste caso, o plano é gerado considerando a configuração arquitetural modificada pelo evento externo e, como é correto perante ao modelo, é executável sem problemas no sistema adaptável;

- Depois da produção do plano: neste caso, o plano não é gerado considerando a configuração arquitetural mais atual do sistema adaptável e pode não mais ser válido.

- Durante a execução do plano: novamente, a ocorrência de um evento externo durante a execução do plano pode torná-lo inválido.

Sendo assim, a execução de um plano de reconfiguração deve considerar o momento no qual este é executado. Um evento externo acontecendo antes ou depois de um ciclo de reconfiguração não traz impactos à execução do plano em si. Os dois momentos que trazem impactos e que devem ser considerados são entre a produção do plano e sua execução e durante a execução.

Para evitar que um plano de reconfiguração seja executado em um sistema com uma configuração diferente da considerada na elaboração do plano, é necessário realizar uma validação inicial deste plano face à configuração corrente do sistema. Isto nos leva à mais um requisito para o Escalonador:

R3 Ao receber um plano de reconfiguração dinâmica, antes de iniciar a sua execução, é necessário verificar se a configuração arquitetural corrente é a mesma considerada na geração do plano.

Caso, durante a execução do plano, a configuração arquitetural seja alterada por um evento externo, que invalide o plano, a sua execução não pode mais continuar e alguns cuidados devem ser tomados. Segundo LÉGER *et al.* (2010), uma reconfiguração dinâmica pode ser inválida para o usuário, quando leva o sistema a um estado inconsistente no qual pode ficar indisponível e/ou inutilizável do ponto de vista funcional, e inválida do ponto de vista do modelo de componentes, onde interfaces requeridas ainda não foram atendidas ou uma ligação ainda não foi efetuada. No caso do iPOJO, um estado inconsistente é aquele no qual o sistema fica **inválido** para o usuário, ou seja, o sistema não responde à requisições.

Para evitar que o sistema fique inválido, é necessária uma constante verificação do seu estado durante a execução de uma reconfiguração dinâmica pois, na presença de um evento externo, a sua configuração pode ser alterada e uma determinada ação de reconfiguração planejada pode não ser mais aplicável.

Para isto, cada ação de reconfiguração arquitetural conta com precondições que devem ser definidas para sua correta execução. Estas precondições visam garantir que a ação é aplicável. Isto define mais um requisito para o Escalonador:

R4 Durante a execução do plano, antes da cada ação de reconfiguração, é necessário observar se as precondições daquela ação permanecem válidas. Estas precondições devem ser definidas para cada ação de reconfiguração arquitetural.

Podemos, ainda, considerar um plano de reconfiguração como uma transação a ser realizada no sistema adaptável. Segundo LÉGER *et al.* (2010), uma transação deve seguir as propriedades ACID[2]. No entanto, no contexto de uma reconfiguração de um sistema dinâmico, estas propriedades devem ter suas premissas revistas.

A propriedade de **Atomicidade**, por exemplo, aplica-se ao contexto do Escalonador, que, em caso de erro, deve levar o sistema ao estado inicial através de um mecanismo de compensação, o que define mais um requisito para o Escalonador:

R5 O Escalonador deve considerar a execução do plano de reconfiguração como uma transação, ou seja, deve ser completado com sucesso ou deve ser desfeito, em caso de erros, através de um mecanismo de compensação. Este mecanismo implementa uma das estratégias possíveis, que consiste na execução de um plano de reconfiguração complementar ao original e na ordem inversa, a partir da ação que provocou o erro.

É importante ressaltar, entretanto, que, dependendo do evento externo, não é possível levar o sistema ao seu estado inicial através da execução de ações complementares na ordem inversa, situação na qual o sistema permanece inválido, e um novo ciclo de adaptação deve ser iniciado para produzir um novo plano, compatível com o estado corrente do sistema. Logo, a abordagem de compensação adotada aqui visa, apenas, minimizar os impactos de um plano inválido.

A propriedade de **Consistência**, por sua vez, é um requisito para o Planejador, que deve gerar um plano correto perante o modelo, considerando que o critério de consistência de um sistema desenvolvido com iPOJO é que este deve permanecer no estado "Válido".

Já a propriedade de **Isolamento**, não se aplica neste caso, já que um evento externo pode alterar o estado do sistema a qualquer momento, não existindo maneira de impedir que isto aconteça.

Por fim, a propriedade de **Durabilidade** se aplica parcialmente, ou seja, a reconfiguração é sempre permanente, já que é aplicada ao sistema real, mas não está previsto nenhum mecanismo de persistência da configuração do sistema, já que esta pode ser alterada a qualquer momento.

Por fim, um aspecto importante assumido para o Escalonador é que, no escopo deste trabalho, é assumida uma relação de ordem total para as ações de reconfiguração. Isto significa que o plano é sempre executado de maneira sequencial, ou seja, uma ação só é executada quando a ação anterior do plano for finalizada, não existindo concorrência na execução das ações, por mais que uma não seja pré-requisito para a outra.

[2] *Atomicity, Consistency, Isolation, Durability*

R6 O Escalonador deve ser capaz de executar as ações de reconfiguração de forma sequencial, garantindo que a próxima ação não seja executada enquanto a ação anterior não for finalizada.

A adoção da relação de ordem total simplifica a especificação do Escalonador, ao passo que a outra opção, que é a execução simultânea de ações não relacionadas entre si, poderia aumentar a performance da reconfiguração. Esta última opção, no entanto, traz a necessidade de identificação de que grupos de ações podem ser executados em paralelo entre si, o que foge do escopo deste trabalho de pesquisa e é considerado um trabalho futuro.

4.2.2 DC4iPOJO

Para que o Escalonador possa funcionar na plataforma OSGi+iPOJO, o modelo de componentes adotado deve prover capacidades de reconfiguração dinâmica. Conforme já colocado, o modelo hierárquico do iPOJO não provê tais capacidades no nível adequado. Sendo assim, esta pesquisa define uma extensão para o iPOJO, com o objetivo de adicionar estas capacidades ao seu modelo de componentes hierárquico. A extensão criada chama-se DC4iPOJO (*Dynamic Composite for iPOJO*, em português: Composição Dinâmica para iPOJO).

O DC4iPOJO tem dois objetivos principais:

1. Modificar a estrutura interna do iPOJO para permitir Reconfiguração Dinâmica do seu modelo de componentes hierárquico; e

2. Definir uma API para acesso às ações de reconfiguração das composições dinâmicas criadas.

Para atingir o objetivo 1, as classes internas do iPOJO que definem uma composição e suas características devem ser alteradas. Atualmente, estas classes suportam apenas a criação de uma composição. O DC4iPOJO adiciona suporte para alteração destas composições em tempo de execução.

O objetivo 2 é atingido através da criação de uma camada de acesso aos novos métodos criados. Por meio desta API, os comandos de reconfiguração arquitetural são definidos e, através deles, as ações do plano de reconfiguração são executadas.

Em termos de ações de reconfiguração arquitetural, a seguinte lista obtida do estudo realizado no Capítulo 3 deve ser considerada:

- **Criação de uma composição**: refere-se à capacidade de criação de uma nova composição vazia, para posteriormente ter componentes adicionados. É uma capacidade básica de um *framework* de componentes dinâmico hierárquico;

- **Adição/Remoção de componente a uma composição**: através desta ação, componentes são criados/destruídos de uma composição. Compõe também o conjunto de capacidades básicas;

- **Alteração de ligação entre componentes**: a ligação entre componentes é necessária para o seu correto funcionamento e, por isso, a ação de alteração destas ligações deve ser considerada;

- **Alteração de propriedade de um componente**: componentes podem ser configurados através da alteração de propriedades, que são valores associados a eles. Em determinados componentes, a sua configuração é um pré-requisito para o seu funcionamento e, por isso, esta ação deve ser considerada;

- **Alteração da interface de uma composição**: a interface de um componente é o elemento básico através do qual ele se comunica com outros componentes. Esta ação visa alterar este elemento em uma composição e deve ser considerada;

- **Transferência do estado de um componente para outro**: ao realizar a troca de uma instância de componente por outra semelhante, se os estados forem compatíveis é possível considerar a transferência do estado de execução da instância antiga para a nova instância. Quando a reconfiguração é mais radical adicionando ou removendo requisitos ao sistema adaptável, esta ação não é realizável, visto que o estado pode não ser compatível. Isto pressupõe uma tradução do estado e a lógica para isto varia de componente para componente, motivo pelo qual esta ação não é considerada no *framework* e sim delegada para os componentes individualmente, que devem ser capazes de salvar o seu estado quando estiverem sendo desconectados e recuperarem o seu estado quando estiverem sendo iniciados[3]; e

- **Alteração o conector**: diferentemente de outros modelos de componente, o iPOJO não considera o conector como uma entidade de primeiro nível. Sendo assim, esta ação não é considerada neste trabalho.

Neste sentido, podemos considerar o seguinte requisito para o DC4iPOJO:

R7 Lista de ações de reconfiguração a ser considerada: criação de composição, adição/remoção de componentes de uma composição, alteração da ligação entre componentes, alteração da propriedade de um componente e alteração da interface de uma composição.

[3]O iPOJO provê meios de salvar/recuperar variáveis de um repositório, além de *callbacks* que podem ser conectados aos eventos de iniciação e finalização da execução de uma instância de componente.

A adição destas capacidades ao iPOJO, através do DC4iPOJO, o reposiciona na tabela criada ao final do Capítulo 3, como pode ser visto na Tabela 4.1, que resume as capacidades consideradas.

Tabela 4.1: Capacidades de reconfiguração adicionadas pela extensão proposta

Ações de Reconfiguração	**iPOJO**	**iPOJO+DC4iPOJO**
Criar composição	✓	✓✓
Adicionar componente	✓	✓✓
Remover componente	✓	✓✓
Transferir estado		
Alterar propriedade	✓	✓✓
Alterar ligação	✓	✓✓
Adicionar interface	✓	✓✓
Remover interface	✓	✓✓
Alterar conector		

4.3 Implementação do Escalonador

Para atender os requisitos **R1-R6**, referentes ao Escalonador, a arquitetura da Figura 4.3 foi idealizada. Nela, é possível observar alguns componentes, que são detalhados em seguida.

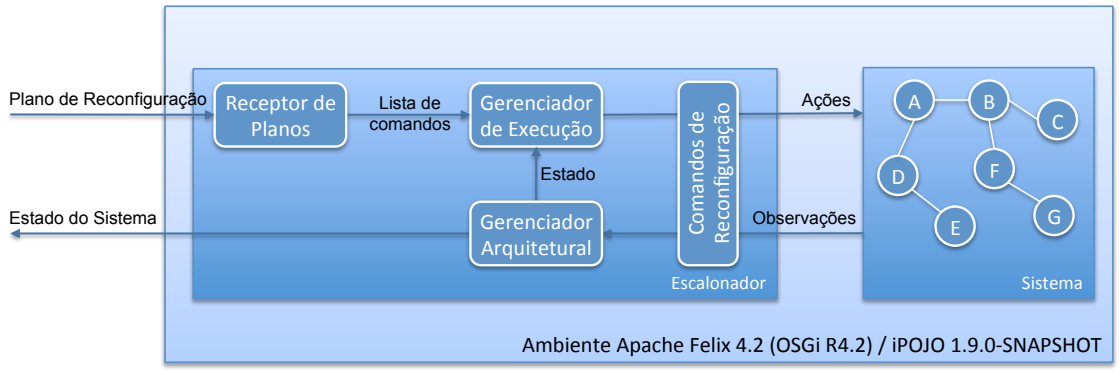

Figura 4.3: Arquitetura do Escalonador

Receptor de Planos Este componente é responsável por receber o plano de reconfiguração e traduzi-lo para uma lista de comandos, considerando a relação entre os operadores do plano e os comandos de adaptação disponíveis.

Gerenciador de Execução O Gerenciador de Execução é o responsável por executar sequencialmente a lista de comandos gerada pelo componente Receptor e, em caso de falhas, realizar a compensação, desfazendo as ações já executadas, por meio de ações complementares.

Gerenciador Arquitetural Este componente mantém uma representação arquitetural do sistema e é o responsável por informar para o Planejador a sua configuração arquitetural inicial, para a produção do plano de reconfiguração. É, também, o responsável por verificar se houve mudanças na configuração arquitetural durante a produção do plano, antes do início da sua execução.

Comandos de Reconfiguração São os componentes que encapsulam as chamadas à API do DC4iPOJO e são os responsáveis por executar as ações de reconfiguração.

A ideia destes componentes é ser extensível. Ou seja, a parte genérica fica implementada em uma classe abstrata, enquanto que as partes específicas da implementação realizada aqui ficam em uma subclasse. As seguintes seções detalham cada um destes componentes.

4.3.1 Receptor de Planos

Este componente é o responsável por receber o plano de reconfiguração gerado pelo Planejador. Esta ação é dependente da tecnologia utilizada. Neste trabalho, a integração com o JSHOP2 não é *online*. O plano gerado é salvo em um arquivo texto que é lido por este componente. Sendo assim, ele é instanciado aqui como um "Receptor de Planos via Arquivo Texto".

Para isto, este componente implementa um ciclo de leitura de um diretório específico, configurado a partir de uma propriedade deste componente. Outra propriedade configurada é o tempo entre uma leitura e a próxima leitura no diretório definido. Por fim, a última propriedade considerada é a extensão dos arquivos que conterão os planos de reconfiguração arquitetural.

O segundo objetivo do Receptor é traduzir os operadores do plano de reconfiguração para comandos de reconfiguração compatíveis com a plataforma em questão, no caso o DC4iPOJO. Os operadores do plano de reconfiguração dependem, por sua vez, do domínio definido para o JSHOP2. Sendo assim, é outro requisito específico que deve ser levado em consideração na hora da sua implementação. Neste trabalho, para simplificar a implementação do "Receptor de Planos via Arquivo Texto", considera-se que os nomes dos operadores são os nomes dos comandos de reconfiguração. Porém, isto nem sempre é verdade para outros domínios.

Por fim, qualquer Receptor precisa de uma implementação do componente "Gerenciador de Execução" para que a lista de comandos gerada seja executada. Logo, o Receptor requer a interface "Executor" para funcionar corretamente.

Em termos de implementação na linguagem Java, o Receptor é uma classe abstrata que delega a implementação de alguns métodos para a classe filha que a estende. A Figura 4.4 mostra o seu diagrama de sequência.

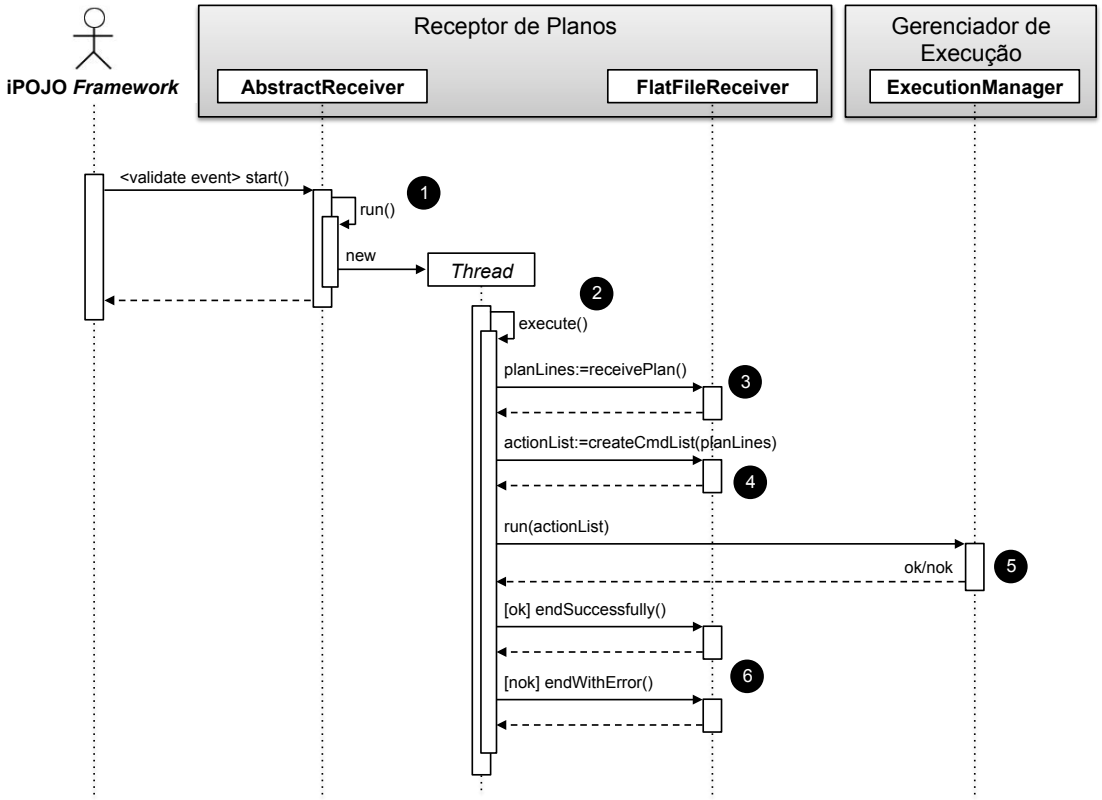

Figura 4.4: Diagrama de sequência do componente "Receptor de Planos"

Sua execução começa, como todo componente iPOJO que tenha cadastrado um *callback* para o evento de validação do componente, no método `start()` da classe `AbstractReceiver` **(1)**. Em seguida, este método cria uma nova *Thread* que é a responsável por chamar o seu próprio método `execute()` **(2)**, dando início ao *loop* principal do receptor, que fica esperando pelo surgimento de planos de reconfiguração através de um mecanismo de *polling*. Este *loop* executa o primeiro método abstrato `receivePlan()` **(3)**, que é implementado pela classe filha `FlatFileReceiver`. Se existir um arquivo contendo um plano de reconfiguração pronto para ser executado, este método retornará uma lista dos operadores deste plano. Em seguida, esta lista é enviada para outro método abstrato, chamado `createCmdList()` **(4)**. Este método faz a tradução da lista de operadores para a lista de comandos. Uma vez traduzida, o método `run()` do componente "Gerenciador de Execução" é chamado **(5)** para executar os comandos. Este método faz parte do escopo do "Gerenciador de Execução" e está detalhado na próxima seção. A execução do plano pode terminar com sucesso ou não. Dependendo do resultado, o método `endSuccessfully`, ou `EndWithError`, é chamado **(6)** e a execução do plano é finalizada.

4.3.2 Gerenciador de Execução

Uma vez que o plano de reconfiguração é recebido e traduzido, o componente "Gerenciador de Execução" é chamado para executá-lo. Este componente percorre a lista de comandos e a executa, comando após comando, respeitando a ordem em que aparecem, de acordo com o requisito **R6**.

Os comandos de reconfiguração dinâmica são implementados como componentes iPOJO (Seção 4.3.4) e são injetados no "Gerenciador de Execução" no momento em que este é instanciado.

Cada ação de reconfiguração executada é colocada em uma pilha e, caso haja um erro de execução ou o estado do sistema seja alterado de forma que não seja possível continuar a execução do plano, o mecanismo de compensação é ativado, conforme requisito **R5**. Este mecanismo lê as ações da pilha e, para cada ação, executa a ação inversa, começando pela última que foi executada até a primeira. Esta estratégia pode levar o sistema de volta ao seu estado inicial, em caso de tentativa de execução de um comando que não tenha as suas precondições atendidas. Entretanto, em um cenário onde um evento externo torna a aplicação inválida, excutar as ações de reconfiguração complementadas em ordem inversa não é garantia de levar o sistema ao seu estado inicial. Este tipo de situação só é corrigida em um novo ciclo de adaptação, que gerará um novo plano de reconfiguração levando em consideração o estado inválido do sistema como estado inicial.

A Figura 4.5 mostra o diagrama de sequência do "Gerenciador de Execução". Este diagrama é um complemento do exibido na Figura 4.4, pois descreve o que acontece do "Gerenciador de Execução" até a execução do comando.

Como pode ser visto, quando a chamada ao método `run()` é realizada, a classe abstrata começa a execução da lista de comandos. Como tanto a execução dos comandos de maneira sequencial quanto o mecanismo de compensação são funcionalidades específicas desta implementação, estes ficam na subclasse. Deste modo, outras implementações do "Gerenciador de Execução" podem ser criadas utilizando outras estratégias.

Assim que é chamado, o método `run()` realiza a validação da lista de ações através de uma chamada ao "Gerenciador Arquitetural" **(1)**. Esta chamada visa identificar se a arquitetura do sistema foi alterada por um evento externo, enquanto o planejador produzia o plano de reconfiguração. Caso o plano não seja válido, a execução é encerrada e o sistema não é reconfigurado. Caso seja possível prosseguir com a execução, o método `run()` delega a execução da lista de ações para a subclasse através da chamada ao `executeActionList()` **(2)**. Este método executa sequencialmente as ações da lista recebida como parâmetro, mas antes disso, gera o comando inverso através de uma chamada ao método `generateInverse()`

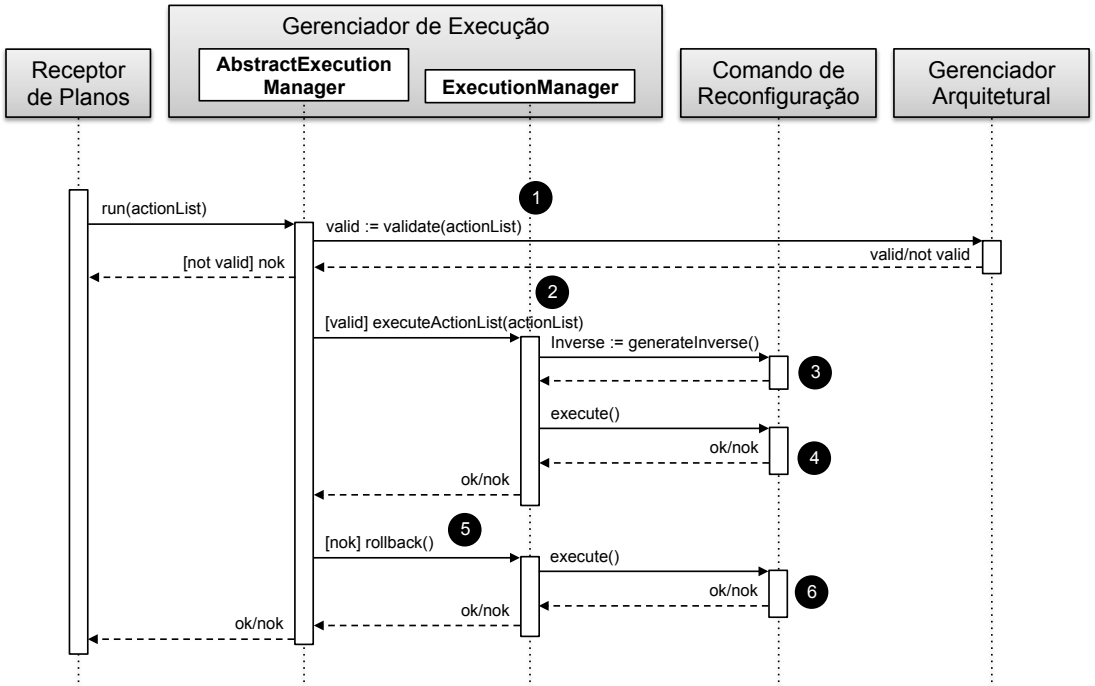

Figura 4.5: Diagrama de sequência do componente "Gerenciador de Execução"

do próprio comando **(3)**. Isto se dá pois o próprio comando deve saber qual é o seu complementar. Deste modo, a classe `ExecutionManager` fica genérica, capaz de utilizar qualquer conjunto de comandos.

Após a obtenção do comando complementar, o comando é, enfim, executado **(4)**. A execução do comando pode ser com sucesso ou com erro. No caso da execução com sucesso, o índice da lista é incrementado e os passos 3 e 4 são executados para o próximo comando da lista, até que não reste mais nenhum comando a ser executado. Caso haja um erro, a execução da lista de comandos é abortada e o método `executeActionList` produz uma exceção indicando que o erro aconteceu.

Neste ponto, a classe abstrata trata a exceção invocando o método `rollback` **(5)**. A estratégia de *rollback* é específica da subclasse. Logo, este método pertence à subclasse. Este método executa os comandos complementares em ordem inversa, a partir do comando que provocou o erro, chamando o método `execute()` de cada um deles **(6)**. Caso haja um erro no processo de *rollback*, este erro é enviado na forma de uma exceção para a classe abstrata, que o devolve para o componente "Receptor de Planos" para que a execução do plano de reconfiguração seja finalizada.

4.3.3 Gerenciador Arquitetural

O "Gerenciador Arquitetural" é o componente do Escalonador responsável por manter uma representação do sistema adaptável, fornecendo-a ao Planejador, no início do ciclo de adaptação. Além disso, é o responsável por verificar se houve mudanças

na configuração arquitetural do sistema, desde que esta foi requisitada pelo Planejador para a produção do plano.

A sua lógica de validação do plano de reconfiguração é bem simples. Para satisfazer o requisito **R3**, ele extrai a configuração arquitetural da aplicação antes da execução do plano e a compara com a configuração enviada para o planejador momentos antes. Para obter a configuração arquitetural do sistema adaptável, ele conta com as capacidades de reflexão do DC4iPOJO. A Figura 4.6 mostra a estrutura deste componente.

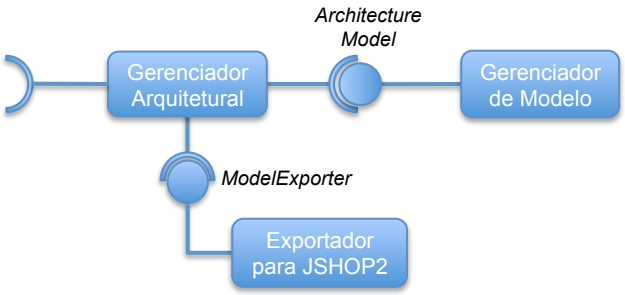

Figura 4.6: Arquitetura do "Gerenciador Arquitetural"

Como pode ser visto, ele conta com dois outros componentes, para os quais delega suas responsabilidades. O "Gerenciador de Modelo" é o responsável por manter o modelo do sistema, utilizando as capacidades de reflexão do DC4iPOJO. Já o "Exportador para JSHOP2" é o responsável por exportar este modelo do sistema para um arquivo texto a ser lido pelo planejador JSHOP2 para a elaboração do plano de reconfiguração dinâmica. Para isto, ele o traduz para o formato esperado pelo JSHOP2. A Listagem 4.2 mostra um exemplo deste arquivo gerado descrevendo o sistema.

Listagem 4.2: Exemplo de um modelo exportado para o JSHOP2

```
1  (componentType ComancheType composite)
2  (componentType FrontendType composite)
3  (componentType RequestReceiverType primitive)
4
5  (interface RequestHandler)
6
7  (requestService RequestReceiverType RequestHandler)
8
9  (componentInstance comanche ComancheType)
10 (componentInstance frontend FrontendType)
11 (incomponent comanche frontend)
12 (componentInstance requestReceiver RequestReceiverType)
13 (incomponent frontend requestReceiver)
14
15 (serviceImport frontend RequestHandler)
```

Esta forma de implementação permite que estes dois componentes auxiliares sejam trocados sem necessidade de alteração do "Gerenciador Arquitetural". Por exemplo, se o formato do arquivo de saída mudar ou se o meio não for mais um arquivo texto, basta codificar a nova maneira de exportar o modelo em uma classe que implemente a interface Java `ModelExporter`. Esta classe, então, deve ser descrita como um componente iPOJO e ligada ao "Gerenciador Arquitetural" no lugar do "Exportador para JSHOP2".

Na Figura 4.7, é possível ver o diagrama de sequência destes componentes. O "Gerenciador Arquitetural" é chamado em dois momentos: para validar um plano e para gerar o modelo do sistema para o planejador.

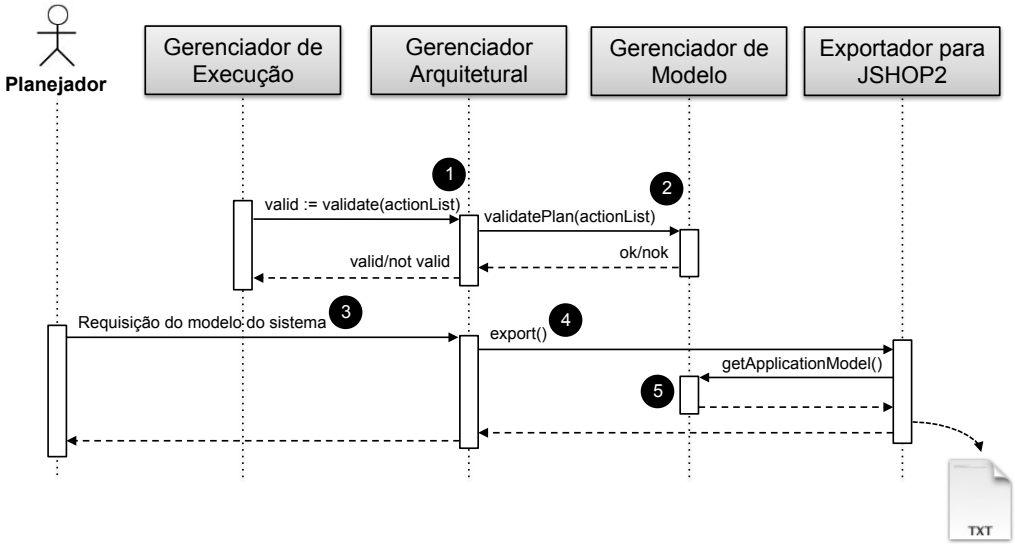

Figura 4.7: Diagrama de sequência do componente "Gerenciador Arquitetural"

A sua primeira função é utilizada pelo "Gerenciador de Execução" que chama o seu método `validate` passando a lista de ações de reconfiguração a serem validadas **(1)**. O método, então, delega esta chamada para o "Gerenciador de Modelo", através de uma chamada ao método `validatePlan()` **(2)** que verifica se o sistema sofreu alguma alteração arquitural durante o tempo em que o plano foi gerado, retornando o resultado da verificação para o "Gerenciador Arquitetural". Este, por sua vez, retorna a informação de validade do plano para o "Gerenciador de Execução.

A segunda função deste conjunto de componentes é exportar o modelo arquitetural do sistema para que o planejador possa utilizá-lo para gerar o plano de reconfiguração. Para isto, o planejador envia uma requisição para o "Gerenciador Arquitetural" **(3)** solicitando o modelo do sistema. Neste trabalho, esta requisição é manual, realizada através de um comando no *shell* do Apache Felix. Uma vez que a requisição seja realizada, o método `export()` do "Exportador para JSHOP2" é chamado **(4)**. Este método, por sua vez, busca o modelo do sistema através de uma chamada ao método `getApplicationModel()` **(5)**, do "Gerenciador de Modelo". O

resultado desta chamada é utilizado para gerar a representação do sistema no arquivo texto que é posteriormente lido pelo JSHOP2, também de maneira manual.

4.3.4 Comandos de Reconfiguração

Cada ação do plano de reconfiguração deve ser executada de forma atômica, alterando a configuração arquitetural do sistema. Para isto, foram desenvolvidos componentes que atuam como comandos, representando estas ações. Estes componentes são acessados pelo "Gerenciador de Execução" no momento da execução do plano. Este tipo de implementação é extensível e permite que novos comandos sejam adicionados através da implementação de uma *interface* Java chamada `ReconfigurationCmd`. Estes comandos acessam a camada de API e as camadas mais baixas do DC4iPOJO para atingir o seu objetivo.

O resultado da execução destes comandos deve ser o mais previsível possível. Não é possível afirmar que eles sempre atingirão o seu objetivo com sucesso, já que, em um sistema verdadeiramente dinâmico, componentes podem ficar indisponíveis a qualquer momento. Por conta desta imprevisibilidade, mesmo que o plano gerado seja correto do ponto de vista do modelo do sistema, cada comando deve verificar se as precondições necessárias para a sua execução estão atendidas naquele momento, de acordo com o requisito **R4**. Por este motivo, a sua execução se divide em três etapas:

1. Verificação de Precondições: Nesta etapa, cada comando executa uma série de validações com o objetivo de verificar se podem ser executados, ou se algo mudou no sistema de forma que a sua execução não levará o sistema à configuração desejada, no âmbito do componente/instância a que ele se refere.

2. Execução do comando: Após a verificação das precondições, o comando pode de fato ser executado.

3. Verificação das Pós-condições: Esta etapa tem a função de verificar se o comando atingiu de fato o seu objetivo, ou se algum evento externo atuou e modificou o resultado esperado. Esta etapa pode ser considerada um teste unitário daquela parte do sistema modificada após a execução da reconfiguração.

Todos os comandos seguem esta sequência de passos, como mostra o diagrama de sequência da Figura 4.8. Caso algum destes passos gere um erro, este erro é propagado para o "Gerenciador de Execução" que adota as estratégias pré-definidas para seu tratamento. A Tabela 4.2 resume as precondições e pós-condições de cada comando, que atendem ao requisito **R7**.

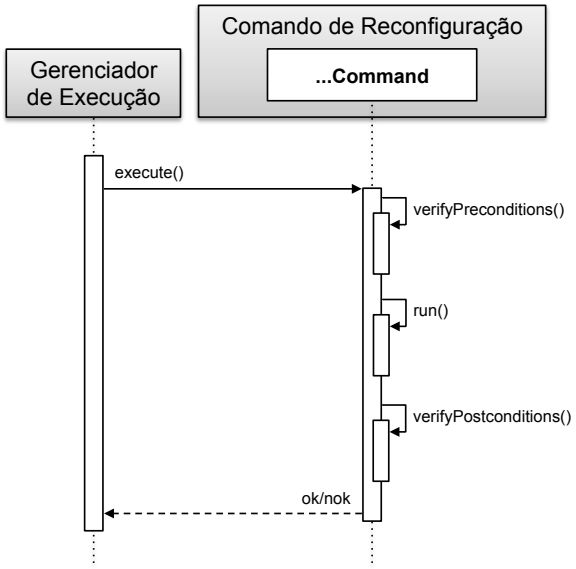

Figura 4.8: Diagrama de sequência dos componentes que representam os comandos de reconfiguração

Tabela 4.2: Comandos e suas precondições e pós-condições

Comando	**Precondições**	**Pós-condições**
CreateCompositeType	O tipo não pode existir	O tipo deve existir
DestroyCompositeType	O tipo deve existir Não deve haver instâncias deste tipo ativas	O tipo não deve existir
CreateInstance	A instância não pode existir Se criada dentro de uma composição, esta deve existir	A instância deve existir no contexto adequado
DestroyInstance	A instância deve existir Se for uma composição não deve haver instâncias internas Não deve estar ligada a outra instância Não deve exportar serviço para o nível superior Não deve importar serviço do nível superior	A instância não pode existir
Bind	A instância de origem deve existir A instância de destino deve existir A interface de origem deve requerer a interface de ligação A instância de destino deve prover a interface de ligação Não deve existir uma ligação entre as instâncias de origem e destino para a interface em questão	A ligação entre a instância de origem e destino deve existir para a interface em questão

Tabela 4.2 – Comandos e suas precondições e pós-condições *(Continuação)*

Comando	**Precondições**	**Pós-condições**
Unbind	A ligação entre a instância de origem e destino deve existir para a interface em questão	Não deve existir uma ligação entre as instâncias de origem e destino para a interface em questão
AddExportedService	A composição não deve exportar o serviço em questão A instância interna deve prover o serviço em questão	A composição deve exportar o serviço em questão
RemoveExportedService	A composição deve exportar o serviço em questão Não deve existir ligação cuja instância de destino seja a composição em questão, para o serviço em questão	A composição não deve exportar o serviço em questão
AddImportedService	A composição não deve importar o serviço A instância externa deve prover o serviço em questão	A composição deve importar o serviço
RemoveImportedService	A composição deve importar o serviço Não deve haver instância interna ligada ao serviço em questão	A composição não deve importar o serviço
SetProperty	A instância deve possuir tal propriedade	A propriedade deve possuir o valor determinado

Estes comandos utilizam chamadas à API do DC4iPOJO, que foi desenvolvida no âmbito desta pesquisa. A próxima seção detalha esta extensão para o iPOJO.

4.4 Implementação do DC4iPOJO

A segunda linha de pesquisa deste trabalho é uma extensão para o iPOJO, com o objetivo de adicionar ao modelo de componentes hierárquico deste *framework* a capacidade de Reconfiguração Dinâmica. Esta seção visa detalhar os aspectos técnicos da implementação do DC4iPOJO.

Como já colocado no Capítulo 2, o iPOJO é formado por um conjunto de *bundles*, que são instalados e executados no ambiente do OSGi. Suas capacidades são providas por vários *bundles* diferentes, pois o iPOJO foi sendo estendido com o tempo, culminando na versão atual, com o conjunto corrente de funcionalidades. O modelo de componentes hierárquico do iPOJO foi uma destas extensões e é instalado como um *bundle* próprio.

O DC4iPOJO deveria ser uma extensão deste *bundle*. Porém, em termos de estratégia de implementação, devido à maneira através da qual o modelo de componentes hierárquico do iPOJO é implementado, não é possível simplesmente estendê-lo por meio de subclasses. Isto se dá pois os métodos principais das classes deste modelo de componentes têm o modificador `private`, fazendo com que não sejam acessíveis às subclasses que estendem a classe pai.

Por este motivo, o DC4iPOJO substitui a implementação do modelo de componentes hierárquico atual, não sendo compatível com o modelo anterior. A implementação do DC4iPOJO modifica a implementação atual do modelo aproveitando o máximo de código possível. O foco principal não é reescrever o modelo, mas aproveitar as funcionalidades existentes atualmente, adicionando a capacidade de reconfiguração dinâmica.

O DC4iPOJO é constituído por três camadas. A camada mais baixa é o seu núcleo, que se comunica diretamente com o núcleo do iPOJO e é onde o contêiner da composição dinâmica é definido. A segunda camada são os *handlers* que implementam o modelo de componentes do iPOJO, definindo os conceitos de serviço importado e exportado, ligação entre componentes e propriedades. Por fim, a terceira camada forma a API que provê acesso aos métodos que instanciam componentes, ligações e serviços. A Figura 4.9 mostra estas camadas em relação aos demais níveis de abstração do OSGi, iPOJO e dos comandos de reconfiguração do Escalonador. As camadas mais escuras formam o DC4iPOJO.

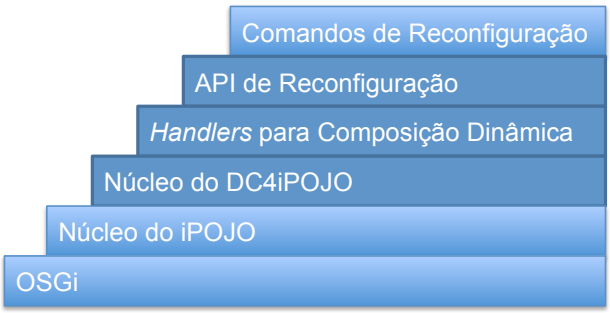

Figura 4.9: Camadas do DC4iPOJO

As próximas seções detalham as três camadas que formam o DC4iPOJO.

4.4.1 Núcleo do DC4iPOJO

Da mesma forma que o modelo de composições do iPOJO, o DC4iPOJO conta com um conjunto de *handlers* para adicionar novas funcionalidades aos componentes iPOJO. Além disso, define um contêiner de componentes que cria um ambiente segregado do ambiente externo, dentro do qual, componentes são adicionados. Estes componentes internos não possuem acesso ao ambiente externo e vice-versa, a menos

que este acesso seja explicitamente permitido. Este contêiner é capaz de ter *handlers* anexados, de forma que a arquitetura estensível do iPOJO é mantida. A Figura 4.10 mostra este contêiner.

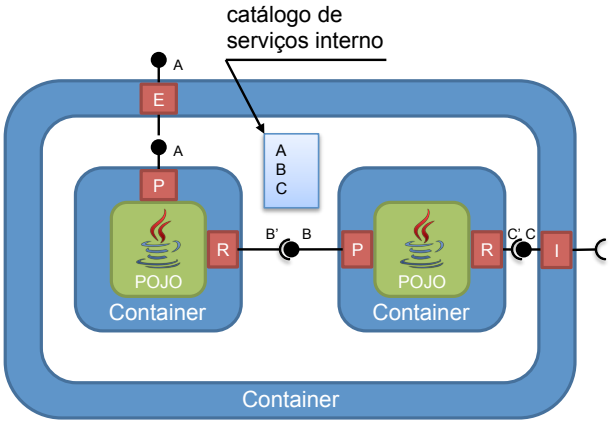

Figura 4.10: Representação de uma composição com componentes internos e o contêiner que a encapsula

Como pode ser visto, o ambiente interno ao contêiner encapsula os componentes, além de contar com um catálogo próprio de serviços. Sendo assim, isola completamente a operação interna da externa, abrindo canais de comunicação através de *handlers*. As classes principais do núcleo do DC4iPOJO podem ser vistas na Figura 4.11.

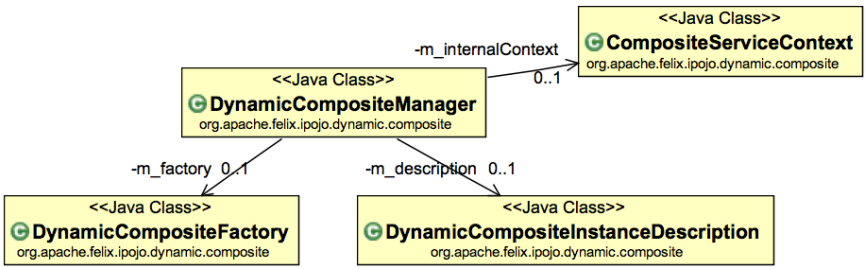

Figura 4.11: Modelo de classes do Core

A principal classe deste modelo é a `DynamicCompositeManager`, que representa o contêiner de componentes. Uma instância desta classe gerencia uma instância de componente que seja uma composição dinâmica. Ela gerencia o seu ciclo de vida e os *handlers* anexados. Porém, sua característica mais importante é estar preparada para permitir alterações na estrutura da composição em tempo de execução. Isto consiste em permitir a adição e remoção de *handlers* de uma composição já existente. Esta característica é a chave para as capacidades de reconfiguração dinâmica do DC4iPOJO.

A classe `CompositeServiceContext` define o catálogo de serviços interno da composição. Este catálogo de serviços próprio da composição permite a definição

de um ambiente interno no qual somente serviços registrados são visíveis e é uma característica importante para garantir o encapsulamento. Serviços não registrados no catálogo interno não são acessíveis e não podem ser consumidos pelas instâncias internas.

Já a classe `DynamicCompositeFactory` implementa o padrão de projeto *Factory*, que tem a responsabilidade de criar instâncias do contêiner da composição. No DC4iPOJO, entretanto, esta classe recebeu novas atribuições. Ela mantém uma lista de instâncias geradas e é capaz de atualizá-las de acordo com as ações de reconfiguração dinâmica invocadas na camada de API.

Por fim, a classe `DynamicCompositeInstanceDescription` faz parte de um conjunto de classes responsável pela funcionalidade de instrospecção do iPOJO e é utilizada para obter a descrição da composição com todos as suas características internas e *handlers* anexados.

4.4.2 *Handlers* para Composição Dinâmica

Para implementar as características do modelo de componentes hierárquico definidas no Capítulo 2, foi definido, no iPOJO, um conjunto de *Handlers*. Como colocado, o contêiner delega para os *handlers* as responsabilidades de gerenciar as características de uma composição. Cada *handler* cuida de uma característica. As próximas seções detalham os *handlers* que formam esta camada do DC4iPOJO.

Handler de Instância de Componente

Este *handler* é responsável por manter e controlar as instâncias internas (simples ou compostas) de uma composição. Sua funcionalidade é implementada pelas classes da Figura 4.12.

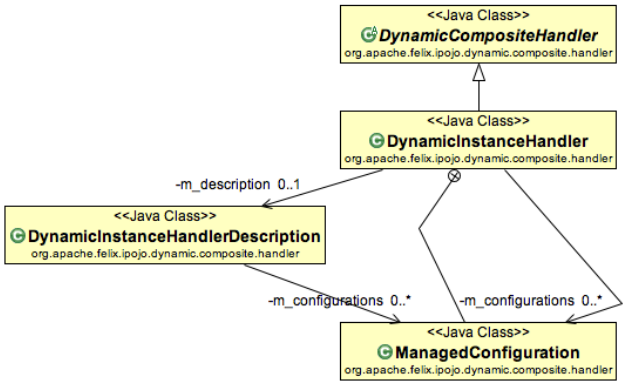

Figura 4.12: Classes que formam o *handler* de instância de componente

A classe `DynamicInstanceHandler` é a responsável por implementar o *handler*. Ela contém uma lista das instâncias internas da composição e possui a capacidade

de criar novas instâncias. Instâncias criadas por esta classe ficam sendo automaticamente controladas por elas. Ela, assim como todos os *handlers* referentes à funcionalidades de uma composição dinâmica do DC4iPOJO, é uma subclasse da classe `DynamicCompositeHandler`. Esta classe é abstrata e implementa as funcionalidades básicas de um *handler* para composições dinâmicas.

Para este *handler*, cada instância fica armazenada em uma lista e é gerenciada por um objeto da classe interna, chamada `ManagedConfiguration`. Esta classe tem o objetivo de manter uma instância de componente e é responsável por armazenar a sua configuração na forma de um dicionário contendo pares de chaves e valores como, por exemplo, o nome da instância, além do objeto da própria instância.

Por fim, o *handler* conta com a classe `DynamicInstanceHandlerDescription`, que é responsável por prover informações para o mecanismo de reflexão do iPOJO.

Handler de Serviço Exportado

Um serviço exportado por um componente composto é uma característica que é gerenciada por um *handler* específico. Um serviço exportado é representado tecnicamente por uma interface Java implementada pela classe que representa o componente. Ao exportar um serviço, um registro desta interface é inserido no catálogo de serviços do nível hierárquico imediatamente acima da composição na qual ele se encontra. Cada serviço exportado é gerenciado pelo conjunto de classes visto na Figura 4.13.

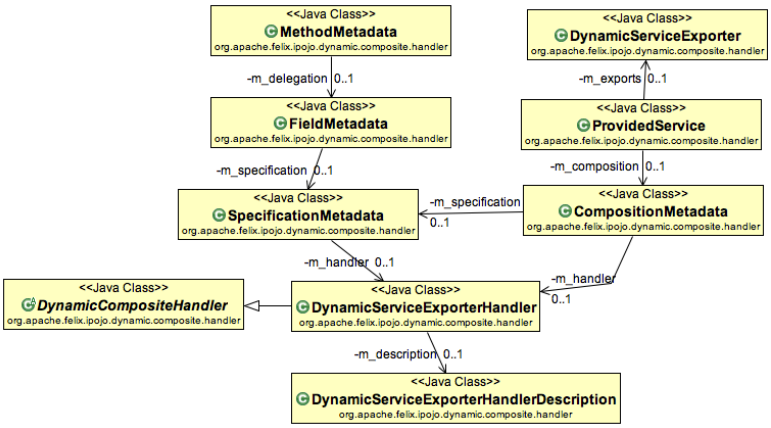

Figura 4.13: Classes que formam o *handler* de serviço exportado

A principal classe deste diagrama é a `DynamicServiceExporterHandler`. Ela contém uma lista das interfaces exportadas para o catálogo de serviços do nível hierárquico superior e é capaz de adicionar e remover serviços exportados em tempo de execução. A classe que realmente exporta o serviço, incluindo-o no catálogo do contexto superior à composição, é a `DynamicServiceExporter`. A classe `DynamicServiceExporterHandlerDescription` é responsável por prover informa-

ções para o mecanismo de reflexão do iPOJO. Por fim, as demais classes do diagrama da Figura 4.13 definem um conjunto de metadados que representa o serviço exportado.

Handler de Serviço Importado

Um serviço importado é o contrário do serviço exportado. Representa uma interface que é definida no nível hierárquico superior e que é importada para ser utilizada no nível hierárquico da composição em questão. Para isto, um registro desta interface é inserido no catálogo de serviços interno. As classes que implementam este *handler* podem ser vistas na Figura 4.14.

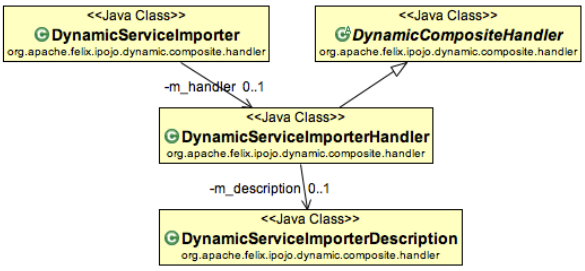

Figura 4.14: Classes que formam o *handler* de serviço importado

A classe que implementa este *handler* é a `DynamicServiceImporterHandler`. Assim como a classe que implementa o *handler* de serviço exportado, ela mantém uma lista interna de serviços importados pela classe `DynamicServiceImporter`. É, também, capaz de realizar adições e remoções de serviços importados em tempo de execução. Assim como os demais *handlers*, possui a sua classe de apoio ao mecanismo de reflexão do iPOJO, a `DynamicServiceImporterDescription`.

Handler de Arquitetura

O *handler* de arquitetura é responsável por prover um ponto de entrada para o mecanismo de introspecção do iPOJO. É por ele que se obtém as descrições do contêiner, de cada *handler* do contêiner, e, com isso, das características da composição. A Figura 4.15 mostra a classe que o implementa.

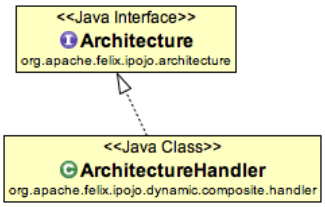

Figura 4.15: Classe que implementa o *handler* de arquitetura

Este *handler* implementa uma interface chamada `Architecture`, ou seja, em termos do iPOJO ele exporta um serviço, já que esta interface é definida como tal na sua descrição de componente. Sendo assim, é possível requerer o serviço representado por esta interface e ter acesso à descrição de todas as instâncias do ambiente de execução do iPOJO.

Uma vez com o objeto deste *handler*, é possível chamar o método `getInstance-Description` que delega a chamada para o contêiner e daí para todos os demais *handlers*. Desta forma, é possível obter a topologia da composição com todas as suas características, incluindo os demais níveis hierárquicos, se existirem, já que o *handler* que controla as instâncias de componente internas à composição também delega a chamada para as suas instâncias.

Handler de Navegação

Enquanto o *handler* de arquitetura é a porta de entrada do processo de introspecção do iPOJO, dando acesso às descrições das instâncias de componente, o *handler* de navegação dá acesso ao objeto que representa a instância, que é o seu contêiner, complementando a introspecção. Ele é utilizado para se obter as instâncias de componente, com o objetivo de apoiar a realização de uma ação de Reconfiguração Dinâmica. A Figura 4.16 mostra a classe que o implementa.

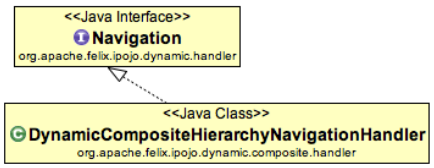

Figura 4.16: Classe que implementa o *handler* de navegação

Da mesma forma que o *handler* de arquitetura, ele também implementa uma interface (chamada de `Navigation`), que o torna acessível para componentes que importem o serviço correspondente.

Handlers para Componentes Simples

Os *handlers* apresentados até aqui são específicos para utilização em um contêiner de uma composição. No entanto, para que seja possível implementar todas as ações de reconfiguração dinâmica definidas, é necessário também definir *handlers* para os componentes simples, já que estes precisam ser acessíveis para o mecanismo de reconfiguração dinâmica e ligados uns aos outros em tempo de execução. A Figura 4.17 mostra as classes que foram criadas para prover as funcionalidades destes novos *handlers*.

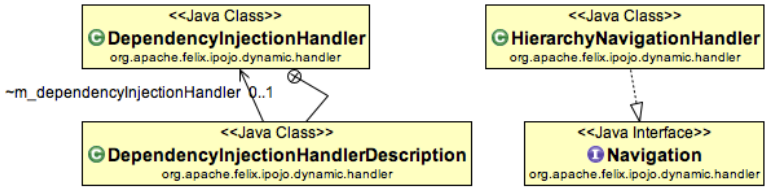

Figura 4.17: Classes que compõem os *handlers* utilizados em componentes simples

Para que seja possível realizar uma ligação entre instâncias de componentes em tempo de execução, componentes precisam prover e requerer serviços. Um componente que requer um serviço é aquele que precisa ser ligado a outro componente para poder funcionar. Por este motivo, o *handler* de serviço requerido do componente simples é também responsável por realizar a sua ligação com outros componentes.

Para isto, foi criado o *handler* de injeção de dependências, baseado no *handler* de serviço requerido original do iPOJO, implementado pela classe `DependencyInjectionHandler`. Esta classe é a responsável por manter os serviços requeridos e realizar as suas ligações com os serviços providos que forem indicados pelas ações de reconfiguração dinâmica que irão criar o sistema a partir dos componentes disponíveis.

Como os demais *handlers*, para suprir o mecanismo de introspecção do iPOJO, este *handler* possui a classe de apoio `DependencyInjectionHandlerDescription`.

O *handler* de navegação para instâncias de componente simples, implementado pela classe `HierarchyNavigationHandler` é responsável por prover um canal para o mecanismo de interseção, assim como a sua versão para composições descrita na seção anterior.

4.4.3 API do DC4iPOJO

A camada de API do DC4iPOJO é a responsável por prover acesso às capacidades de reconfiguração dinâmica do DC4iPOJO, criadas em cima do modelo de componentes hierárquico do iPOJO.

Esta camada conta com um conjunto de classes que guarda o modelo da aplicação como um conjunto de metadados. Este metadado é passado para camadas mais internas do DC4IPOJO e, finalmente, é utilizado pelas *factories* para criar os componentes e suas características.

Este grupo é composto por quatro classes. A Figura 4.18 mostra estas classes. A classe `DynamicCompositeComponentType` define uma composição dinâmica e as classes `Instance`, `ImportedService` e `ExportedService` definem a sua topologia. As próximas seções descrevem estas classes e ilustram a sua utilização.

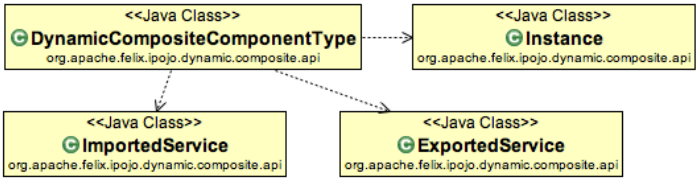

Figura 4.18: Classes que formam a API de reconfiguração dinâmica do DC4iPOJO

Declaração de uma composição

A classe `DynamicCompositeComponentType` representa um modelo de um componente composto. Ela é instanciada quando se deseja criar uma composição, operação realizada pelo comando "CreateCompositeType", da tabela 4.2. O código abaixo exemplifica esta operação.

Listagem 4.3: Exemplo da criação de uma composição e uma instância

```
1  DynamicCompositeComponentType myDynamicCompositeType = new
       DynamicCompositeComponentType()
2     .setBundleContext(m_context)
3     .setComponentTypeName("composite_1");
4  try {
5     ComponentInstance myDynamicCompositeInstance =
          myDynamicCompositeType.createInstance();
6  } catch (ConfigurationException e) {
7     e.printStackTrace();
8  } catch (MissingHandlerException e) {
9     e.printStackTrace();
10 } catch (UnacceptableConfiguration e) {
11    e.printStackTrace();
12 }
```

Logo após a criação do objeto `dynamicCompositeType`, dois métodos são chamados. O método `setBundleContext` serve para registrar no contexto de que *bundle* o componente será criado e o `setComponentTypeName` serve para registrar o nome que o Tipo de Componente terá.

A chamada ao método `createInstance`, feita dentro do bloco `try-catch`, desencadeia o processo de criação da *factory* pela API e, posteriormente, o processo de criação da instância de componente pela *factory*.

Existem dois momentos no ciclo de vida de uma composição onde os métodos da API são chamados. O primeiro momento pode ser entendido como anterior à existência da composição de fato. Neste momento, as *factories* e as instâncias do componente composto não existem ainda, pois o modelo do componente composto ainda não foi definido. Este momento é a configuração da composição.

O segundo momento é aquele onde a composição já está definida, sua *factory* criada com instâncias da composição já existentes. Neste momento, qualquer alteração no modelo da composição gerará uma alteração do metadado da *factory* e, consequentemente, da topologia e/ou das características das instâncias daquele componente composto. Este momento é a reconfiguração da composição.

Sendo assim, em tempo de configuração e reconfiguração da composição, o modelo definido suporta tanto ações de adição de características quanto ações de remoção de características. Para isto, utiliza as demais classes da Figura 4.18, cujas instâncias vão sendo adicionadas (ou removidas) de listas internas à composição. Estas listas vão, posteriormente, ser usadas para formar o metadado que será passado para a *factory* daquela composição.

As características que definem uma composição e os métodos que as definem são:

Instâncias Internas Para adicionar instâncias de componentes internos ao modelo da composição, deve-se utilizar o método `addInstance`. Este método recebe uma instância da classe `Instance` que representa uma instância de componente interno à composição. De forma semelhante, para remover um componente do modelo de uma composição, utiliza-se o método `removeInstance`. Este método, porém, recebe como parâmetro o nome da instância de componente a ser removida da composição.

Serviços Requeridos Para adicionar serviços requeridos ao modelo do componente composto, utiliza-se o método `addServiceImport`. Este método recebe como parâmetro um objeto do tipo `ImportedService`, que representa um serviço importado do contexto no qual a composição se encontra e disponibilizado para ser utilizado internamente à composição. Este serviço importado é provido por uma instância de componente externo à composição e está intimamente ligado à ela. Para a remoção deste serviço, utiliza-se o método `removeServiceImport`, que recebe como parâmetro o nome da instância de componente existente no contexto hierarquicamente superior cuja disponibilidade interna dos serviços deve ser removida.

Serviços Providos A adição de um serviço provido ao modelo do componente composto pode ser feita através do método `addServiceExport`, que recebe como parâmetro um objeto da classe `ExportedService`. Para remover este serviço exportado, o método `removeServiceExport` pode ser utilizado. Da mesma maneira que funciona para serviços requeridos, este método recebe o nome da instância de componente interna à composição, cujos serviços deixarão de estar disponíveis para o contexto hierarquicamente superior.

Definição de Instâncias Internas

Uma composição só faz sentido se dentro dela existirem instâncias de componentes, pois ela nada mais é do que uma abstração para encapsular partes de um sistema que devem ficar juntas e vistas pelo resto do sistema como um componente único. Logo, após a criação de uma composição, devemos adicionar as instâncias de componentes que residirão no contexto de execução que ela cria.

Isto pode ser feito adicionando-se uma instância da classe `Instance` ao tipo recém criado. Esta operação é realizada pelo comando "CreateInstance", definido na Tabela 4.2, e pode ser vista no código da Listagem 4.4:

Listagem 4.4: Exemplo de adição de uma instância à uma composição

```
1 myDynamicCompositeType.addInstance(
2   new Instance("org.test.handler.providerA.ProviderA")
3     .addProperty("instance.name", "providerA")
4 );
```

O método `addInstance` da classe `DynamicCompositeComponentType` adiciona o objeto da classe `Instance` recebido como parâmetro à lista interna de instâncias.

A classe `Instance`, por sua vez, deve ser inicializada passando-se como parâmetro para o seu construtor o nome do componente (simples ou composto) que se deseja adicionar à composição. No caso do exemplo acima, o componente passado tem o mesmo nome da classe que o implementa e podemos deduzir que é um componente simples. Já componentes compostos não são implementados diretamente por classes Java e seus nomes tendem a ser diferentes de nomes de pacotes Java.

Uma vez instanciada, a classe `Instance` deve ser configurada através da adição de propriedades. No exemplo acima, a propriedade `instance.name` define o nome da instância do componente em questão. Estas propriedades serão adicionadas ao metadado a ser passado para a *factory*.

A remoção de uma instância de componente de dentro de uma composição é exemplificada pelo código da Listagem 4.5:

Listagem 4.5: Exemplo de remoção de uma instância de uma composição

```
1 myDynamicCompositeType.removeInstance("providerA");
```

Definição de Serviços Importados

No modelo de componentes orientado à serviços do iPOJO, a ligação entre as instâncias de componentes compostos é viabilizada através dos seus serviços importados e exportados.

Como uma composição é uma abstração que encapsula outros componentes, seus serviços requeridos (e também os providos) devem ser definidos pelos seus compo-

nentes internos. Sendo assim, um serviço requerido de uma composição deve ser um serviço requerido de algum componente interno. A Figura 4.19 ilustra um componente composto formado por dois componentes simples internos.

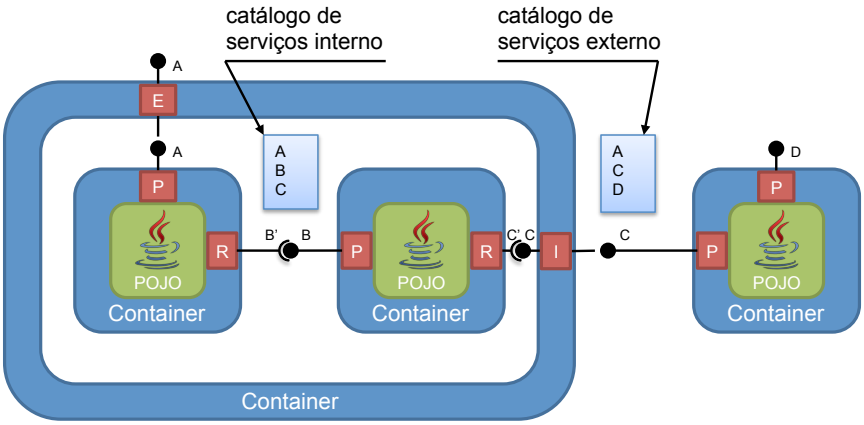

Figura 4.19: Exemplo de serviços requeridos e providos

O componente interno da direita requer o serviço C que não é atendido pelos serviços providos internamente pelo componente da esquerda (que, no caso, provê apenas o serviço A). Sendo assim, o componente composto "importa" o serviço C do componente simples externo à composição para atender o requisito interno. O código que exemplifica um serviço requerido pela composição pode ser visto da Listagem 4.6 e é utilizado pelo comando "AddImportedService", da Tabela 4.2:

Listagem 4.6: Exemplo de adição de um serviço importado à uma composição

```
1 myDynamicCompositeType.addServiceImport(new ImportedService()
2   .setSpecification("org.test.handler.services.InterfaceC")
3   .setInstance("external_provider_instance_name")
4 );
```

Como pode ser visto, duas informações são necessárias para importar um serviço. A primeira é a especificação (ou tipo) da interface que se quer importar. Esta informação é registrada através do método `setSpecification`. A segunda é o nome da instância de componente externa que implementa esta interface ou, em outras palavras, provê o serviço desejado.

Para remover este serviço, importado utiliza-se o método `removeServiceImport` exemplificado pelo código da Listagem 4.7 e utilizado pelo comando "RemoveImportedService", da Tabela 4.2:

Listagem 4.7: Exemplo de remoção de um serviço importado de uma composição

```
1 myDynamicCompositeType.removeServiceImport("
    external_provider_instance_name");
```

Definição de Serviços Exportados

Outra característica de uma composição são serviços exportados (ou interfaces providas). Através deles, a composição poderá ser ligada a outros componentes que requeiram este serviço. Estes serviços estão intimamente ligados à serviços providos por componentes internos à composição. No caso da Figura 4.19, o componente composto exporta o serviço A que, na verdade, é provido pelo componente interno da esquerda.

O código para adicionar um serviço provido à uma composição, utilizado pelo comando "AddExportedService" da Tabela 4.2, pode ser visto na Listagem 4.8.

Listagem 4.8: Exemplo de adição de um serviço exportado à uma composição

```
1 myDynamicCompositeType.addServiceExport(new ExportedService()
2   .setSpecification("org.test.handler.services.InterfaceA")
3   .setInstance("internal_providerA")
4 );
```

De forma semelhante ao serviço requerido, a classe `ExportedService`, que representa um serviço provido pela composição, precisa de duas informações que são a interface a ser provida e o nome da instância de componente interna que a provê.

Para que a composição deixe de prover um determinado serviço provido, basta que o método `removeServiceExport` seja chamado, conforme o exemplo da Listagem 4.9. Este código é utilizado no comando "RemoveExportedService", da Tabela 4.2.

Listagem 4.9: Exemplo de remoção de um serviço exportado de uma composição

```
1 myDynamicCompositeType.removeServiceExport("internal_providerA");
```

4.5 Considerações Finais

Esta seção apresentou as duas linhas de pesquisa deste trabalho: um mecanismo para a execução de um plano de reconfiguração arquitetural em um sistema adaptável (também chamado de Escalonador) e a extensão para o iPOJO para adicionar capacidades de reconfiguração dinâmica, o DC4iPOJO.

O Escalonador é dividido em três componentes internos: o "Receptor de Planos", o "Gerenciador de Execução" e o "Gerenciador Arquitetural", além de contar com vários "Comandos de Reconfiguração", construídos em cima da API fornecida pelo DC4iPOJO. Estes componentes internos dividem as responsabilidades para a execução do plano de reconfiguração dinâmica.

É importante ressaltar que todos estes componentes foram criados tendo em mente a reusabilidade e extensibilidade. Cada um deles possui uma classe abstrata

que implementa o seu funcionamento básico, os *cold spots* e subclasses que implementam os métodos abstratos completando a funcionalidade do *framework*.

Já o DC4iPOJO adiciona ao iPOJO as capacidades de reconfiguração dinâmica requeridas pelo Escalonador. Estas capacidades foram extraídas do estudo realizado no Capítulo 3, que definiu o conjunto básico de ações de reconfiguração utilizado neste trabalho. Com isso, o iPOJO agora ocupa um lugar importante dentre os *frameworks* avaliados, tanto pelas suas capacidades de reconfiguração dinâmica providas pelo DC4iPOJO quanto por utilizar como plataforma o OSGi, *middleware* largamente utilizado na academia e na indústria. Esta plataforma é uma opção madura face às outras essencialmente acadêmicas encontradas.

Capítulo 5

Avaliação

Este capítulo apresenta a avaliação do DC4iPOJO e do mecanismo para execução de planos de reconfiguração arquitetural desenvolvido. A avaliação é realizada na forma de uma Prova de Conceito que exemplifica as funcionalidades desenvolvidas.

Provas de Conceito (do inglês *PoC - Proof of Concept*) representam uma estratégia comum para avaliar abordagens na linha de Reconfiguração Dinâmica. A maioria dos trabalhos analisados no Capítulo 3 utilizam esta estratégia.

Exemplos de utilização de provas de conceito podem ser encontrados nos trabalhos publicados na conferência *Component Based Software Engineering*. Além disso, em (TAJALLI *et al.*, 2010), os autores apresentam uma abordagem para a obtenção da Auto-Adaptação através de um mecanismo de planejamento. O plano de reconfiguração é executado para alterar a configuração arquitetural do sistema de forma a atender aos requisitos estabelecidos. Neste trabalho, foi apresentada uma Prova de Conceito no domínio de robôs, que devem se organizar em um comboio, como forma de avaliação da abordagem.

Sendo assim, este capítulo apresenta uma Prova de Conceito que utiliza um servidor *web* simples, chamado Comanche, para exemplificar cenários de reconfiguração dinâmica utilizando o DC4iPOJO e o mecanismo para execução de planos de reconfiguração arquitural desenvolvidos para alterar a configuração arquitetural do Comanche, de acordo com os requisitos de cada cenário.

A avaliação das capacidades de reconfiguração dinâmica de um *framework* de componentes passa pela demonstração da corretude destas funcionalidades. Nesta demonstração, deve ficar claro que as reconfigurações em tempo de execução funcionam e o sistema é levado a um estado consistente. Para que os efeitos da reconfiguração dinâmica ficassem claros em termos de impactos para o sistema, foi utilizada a estratégia de medição do tempo de atendimento das suas requisições enquanto a reconfiguração é executada. Por isso, cada cenário conta com medições desta variável ao longo do tempo, buscando obter o nível de impacto que a reconfiguração provoca no sistema.

As próximas seções estão organizada da seguinte maneira: a Seção 5.1 detalha a infraestrutura utilizada para a execução da prova de conceito; a Seção 5.2 detalha a prova de conceito em si e, finalmente, a Seção 5.3 apresenta as considerações finais.

5.1 Infraestrutura de Execução

A prova de conceito com exemplos de utilização das contribuições deste trabalho foi executada na implementação do OSGi Apache Felix[1], versão R4.2 (4.2.0).

A versão do iPOJO utilizada foi a 1.9.0-snapshot. Esta versão é a que está em desenvolvimento e se transformará na próxima versão (1.8.6) do iPOJO. A escolha por esta versão se deu devido a um *bug*[2] encontrado e resolvido pelo autor desta dissertação cujo acerto ainda não foi incluído na atual *release* estável (1.8.4) do iPOJO.

A distribuição do Apache Felix conta com um *shell* onde comandos podem ser executados visando interagir com o *framework* e com o ambiente de execução do OSGi. Neste *shell*, *bundles* podem ser instalados, atualizados e iniciados. Além disso, a saída dos comandos, se existir, é exibida na tela deste *shell*. Sendo assim, foi utilizado para verificar o sucesso das execuções dos planos de reconfiguração.

Em termos de *hardware*, as execuções foram realizadas em um MacBook Pro que conta com um processador Intel Core i5 de 2.3GHz e 8GB de memória RAM. O sistema operacional é o Mac OS X Lion 10.7.5.

5.2 Prova de conceito utilizando um servidor *web* simples: Comanche

O sistema usado para demonstrar capacidades de reconfiguração dinâmica de um *framework* é o Comanche[3]. Sua implementação original é em Python, mas possui implementações em Smalltalk[4], C[5] e em Java[6].

O Comanche tornou-se bastante utilizado como exemplo das capacidades de reconfiguração dinâmica do Fractal. É utilizado para demonstrar o uso da Fractal-ADL por Eric Bruneton[7]. Outro exemplo é em (DAVID e LEDOUX, 2006), onde é utilizado para demonstrar o comportamento auto-adaptável em uma abordagem orientada a aspectos. A Figura 5.1 mostra a arquitetura do Comanche.

[1] http://felix.apache.org
[2] https://issues.apache.org/jira/browse/FELIX-3599
[3] http://directory.fsf.org/wiki/Comanche_Server
[4] http://wiki.squeak.org/swiki
[5] http://fractal.ow2.org/cecilia-examples-site/current/comanche/index.html
[6] http://fractal.ow2.org/tutorials/comanche.html
[7] http://fractal.ow2.org/fractal-distribution/comanche-adl/user-doc.html#run

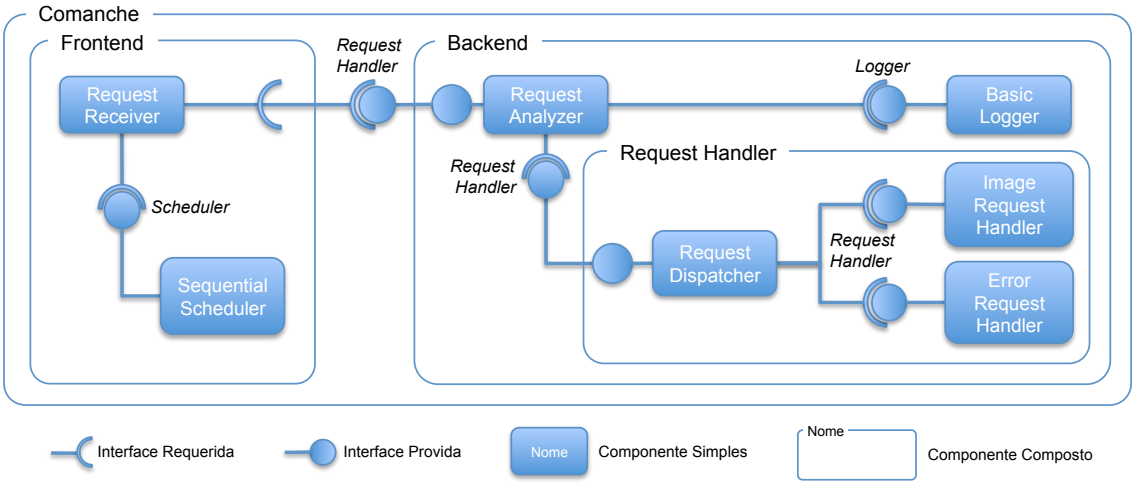

Figura 5.1: Arquitetura do Comanche

O Comanche é formado por sete componentes simples e quatro compostos. Sua arquitetura é hierárquica, onde o componente de mais alto nível, *Comanche*, representa a aplicação. Esta estratégia favorece a reutilização, já que pode ser utilizado como um módulo em um sistema maior.

A composição *Comanche* é formada por duas outras composições: *Frontend* e *Backend*. A composição *Frontend* é formada pelos dois componentes simples *RequestReceiver*, responsável por receber de fato as requisições de uma porta TCP/IP pré-configurada, e o *SequentialScheduler*. Este último implementa a interface *Scheduler* e é o responsável por definir a estratégia para atendimento das requisições que chegam para o *RequestReceiver*. Este componente atende as requisições sequencialmente, sem nenhum nível de concorrência.

O componente *Frontend* requer uma interface chamada *RequestHandler*, que é exportada pelo componente *Backend*. Este, por sua vez, é formado por três componentes, dois simples e um composto. O primeiro componente é o *RequestAnalyzer*, que exporta a interface *RequestHandler* para a composição no qual está inserido. Ele analiza a requisição recebida e a envia para ser processada pelo componente composto *RequestHandler*, com o qual é conectado via interface de mesmo nome. Também possui uma interface requerida chamada *Logger*, que é implementada pelo componente *BasicLogger*.

O componente composto *RequestHandler* é formado por três componentes simples. O primeiro deles é o *RequestDispatcher*, que tanto requer quanto provê a interface *RequestHandler*. Ele exporta esta interface para a composição na qual está inserido. Ele também requer esta interface, com a qual se comunica com os componentes que vão atender de fato a requisição, dependendo do tipo da requisição. É importante ressaltar que esta interface requerida é múltipla, ou seja, várias instâncias de componente podem ser ligadas a ela, o que de fato acontece, como pode ser

visto na figura. Os componentes que estão ligados à ela são o *ImageRequestHandler*, responsável por ler imagens do disco e enviá-las para o navegador *web* e o *ErrorRequestHandler*, que é chamado somente se nenhum outro componente puder atender a requisição de entrada, enviando para o navegador uma mensagem de erro.

Para que este sistema pudesse ser executado sobre o iPOJO, algumas adaptações tiveram que ser realizadas. Além disso, foi necessário criar um XML para cada um dos sete componentes simples, descrevendo-os como componentes iPOJO. Isto é necessário em qualquer aplicação construída sobre o iPOJO. O Apêndice A descreve os XMLs criados e mostra o código Java do Comanche.

Nas próximas seções, são descritos os testes realizados com o objetivo de avaliar a extensão implementada para o iPOJO. É realizada uma medição do impacto no desempenho de execução do Comanche em diferentes cenários enquanto a reconfiguração é realizada, além de exemplificar as capacidades adicionadas ao iPOJO.

5.2.1 Iniciação do Comanche

Utilizando o mecanismo descrito no Capítulo 4, fornecemos para o planejador um estado final que representa a configuração arquitetural do Comanche, conforme a Figura 5.1, tendo como estado inicial aquele onde nenhuma instância de componente existe.

A geração do plano é realizada por um planejador hierárquico (neste caso, o JSHOP2[8]). O planejador executa uma série de tarefas que vão sendo decompostas em tarefas de mais baixo nível até chegar nos operadores, que são os elementos que formam o plano criado. Esta maneira de produzir o plano faz com que as operações apareçam em uma ordem similar à obtida por um algoritmo que percorre a configuração arquitetural do Comanche, como se fosse uma árvore. A Listagem 5.1 representa um fragmento deste plano, mostrando as 10 primeiras linhas. Podemos observar que os componentes vão sendo criados de acordo com a hierarquia. Primeiro os de mais alto nível (compostos e simples), mais próximos à raiz da árvore, e depois os de mais baixo nível na hierarquia, mais próximos às folhas da árvore. Isto é realizado pelos operadores `createCompositeType`, que cria um novo tipo de composição, e `createInstance`, que instancia um tipo de componente, alocando-o no nível hierárquico dado pelo seu último parâmetro. É possível observar que os nomes das instâncias vão ganhando prefixos referentes ao nome da composição da qual fazem parte.

Em seguida, da linha 16 em diante, o plano vai sendo gerado das folhas em direção à raiz, com a realização das ligações entre os componentes, através do operador `bind`. Neste processo, também são realizadas as operações de exportação

[8]http://sourceforge.net/projects/shop/files/JSHOP2

Listagem 5.1: Fragmento inicial do plano de inicialização do Comanche

```
1 (!createCompositeType comancheType)
2 (!createInstance comanche comancheType)
3 (!createCompositeType frontendType)
4 (!createInstance frontend frontendType comanche)
5 (!createInstance requestReceiver org.apache.comanche.
    requestReceiver.RequestReceiver comanche:frontend)
6 (!createInstance sequentialScheduler org.apache.comanche.
    sequentialScheduler.SequentialScheduler comanche:frontend)
7 (!createCompositeType backendType)
8 (!createInstance backend backendType comanche)
9 (!createInstance requestAnalyzer org.apache.comanche.
    requestAnalyzer.RequestAnalyzer comanche:backend)
10 (!createInstance basicLogger org.apache.comanche.basicLogger.
    BasicLogger comanche:backend)
```

e importação de interfaces, de acordo com a necessidade, através dos operadores `addExportedService` e `addImportedService`. A Listagem 5.2 mostra outro fragmento do plano onde isto ocorre. O plano completo pode ser visto no Apêndice B.

Listagem 5.2: Fragmento intermediário do plano de inicialização do Comanche

```
16 (!bind comanche:backend:requestHandler:requestDispatcher comanche:
    backend:requestHandler:imageRequestHandler org.apache.comanche.
    services.RequestHandler)
17 (!bind comanche:backend:requestHandler:requestDispatcher comanche:
    backend:requestHandler:errorRequestHandler org.apache.comanche.
    services.RequestHandler)
18 (!addExportedService comanche:backend:requestHandler
    requestDispatcher org.apache.comanche.services.RequestHandler)
19 (!bind comanche:backend:requestAnalyzer comanche:backend:
    requestHandler org.apache.comanche.services.RequestHandler)
```

Este plano é gravado em um arquivo texto no diretório monitorado pelo componente *Receptor*, parte integrante do *Escalonador*. Ao realizar a leitura do plano, ele produz uma lista de comandos de reconfiguração arquitetural que é executada pelo componente *Executor*. A execução deste plano gera as seguintes linhas no *shell* do OSGi, segundo a Listagem 5.3.

Como pode ser visto, foram executados 24 comandos de reconfiguração, conforme consta no plano, em 211 milisegundos. Logo após esta execução, o Comanche escreve na tela as suas duas linhas iniciais, indicando que está pronto para receber requisições na porta 8088. A partir deste ponto, o Comanche já está atendendo a requisições e um rápido teste em um navegador mostra que ele está funcionando corretamente.

Listagem 5.3: Resultado da execução do plano

```
1 ->
2 Executing plan in file createComanche.plan...
3 Done! Executed 24 reconfiguration commands successfully in 211
     miliseconds.
4 Comanche HTTP Server ready on port 8088.
5 Load http://localhost:8088/gnu.jpg
6
7 ->
```

5.2.2 Adição de um novo *RequestHandler*

O primeiro caso de uso a ser realizado mostra a utilização da capacidade mais básica de reconfiguração dinâmica que é a adição e remoção de componentes em tempo de execução. Na configuração arquitural corrente, existem dois *RequestHandlers*. Estes componentes são responsáveis por, em última instância, buscar o dado que é enviado para o navegador. As duas instâncias existentes são o *ImageRequestHandler*, responsável por carregar imagens JPG e o *ErrorRequestHandler*, que envia uma mensagem de erro e é chamado caso nenhum outro componente consiga atender a requisição. Nesta configuração, o Comache não é capaz de processar arquivos HTML, pois este tipo de arquivo não é uma imagem.

Por este motivo, um novo requisito é adicionado ao sistema. Agora ele deve ser capaz de atender requisições HTML. Para isto, um novo componente deve ser inserido em sua arquitetura, o *HtmlRequestHandler*. A nova configuração arquitetural do Comanche pode ser vista na Figura 5.2, onde o novo componente inserido está em destaque, mais claro do que os demais.

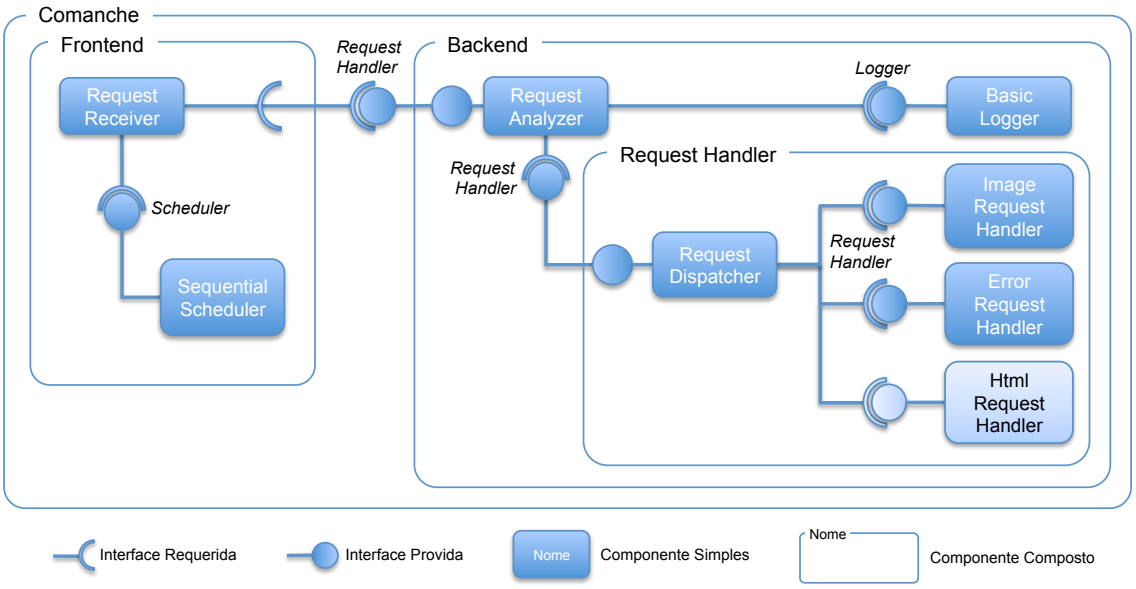

Figura 5.2: Comanche com *HtmlRequestHandler*

Para adicionar esta nova funcionalidade ao sistema, o planejador produziu o plano exibido na Listagem 5.4. A execução dos seus dois passos levou 9ms. A Figura 5.3 mostra o resultado da requisição de um arquivo HTML antes e depois da adição do novo componente.

Listagem 5.4: Plano para inclusão do *HtmlRequestHandler*

```
1 (!createInstance htmlRequestHandler org.apache.comanche.
    htmlRequestHandler.HtmlRequestHandler comanche:backend:
    requestHandler)
2 (!bind comanche:backend:requestHandler:requestDispatcher comanche:
    backend:requestHandler:htmlRequestHandler org.apache.comanche.
    services.RequestHandler)
```

Figura 5.3: (a) Teste realizado sem o *HtmlRequestHandler* e (b) Teste realizado após a adição do *HtmlRequestHandler*

Para ilustrar a operação contrária, um plano de remoção da instância do componente *HtmlRequestHandler* foi executado. Ele encontra-se na Listagem 5.5.

Listagem 5.5: Plano para remoção do *HtmlRequestHandler*

```
1 (!unbind comanche:backend:requestHandler:requestDispatcher comanche:
    backend:requestHandler:htmlRequestHandler org.apache.comanche.
    services.RequestHandler)
2 (!destroyInstance comanche:backend:requestHandler:htmlRequestHandler)
```

A ordem na qual as ações são realizadas é determinada pelo modo como as tarefas foram decompostas na definição do domínio do planejador. No caso da remoção de uma instância de componente de um sistema, inicialmente as suas ligações e serviços importados e exportados devem ser removidos, para depois a instância ser destruída, conforme precondições estabelecidas na Tabela 4.2. Apenas a título de comparação, este plano foi executado em 4ms, o que demonstra que o processo de remoção de uma instância é mais simples do que o de adição, o que era de se esperar.

5.2.3 Troca do *Scheduler*

O segundo caso de uso foca em demonstrar o impacto de uma reconfiguração dinâmica em um sistema enquanto ele está atendendo requisições. Para simular a utilização do Comanche, foi utilizado o Apache JMeter[9], versão 2.7 (r1342410) com o objetivo de disparar automaticamente as requisições. Ele não foi utilizado para medir o tempo gasto nas requisições, pois a menor unidade de tempo que ele é capaz de medir é 1ms e os testes mostraram que o tempo de cada requisição fica na ordem de centenas de microsegundos, já que o cliente e o servidor *web* estão na mesma máquina, o que torna o processo bem rápido.

Neste caso de uso, a requisição realizada foi bem simples, consistindo na visualização de uma imagem em um navegador. Para isto, foi utilizada uma foto que tem 330Kb de tamanho. Para termos uma janela de tempo adequada à realização da reconfiguração, foram criados 10 usuários, cada um enviando 100 requisições, totalizando 1000 requisições.

Para medirmos o impacto da reconfiguração em tempo de execução, podemos imaginar um cenário onde se deseja trocar o *Scheduler* sequencial corrente por um que seja *MultiThread*, capaz de atender a várias requisições em paralelo. Isto confere ao Comanche maior capacidade de atendimento de requisições, consumindo, entretanto, mais recursos em termos de memória.

Para verificar se a suposição em termos de comportamento acima descrita se aplica, inicialmente as duas versões do Comanche foram testadas separadamente. Cada teste teve cinco repetições para assegurar a eliminação de comportamentos aleatórios. O resultado está resumido na Tabela 5.1. A Figura 5.4 mostra a configuração arquitetural do Comanche com o componente *MultiThreadScheduler* no lugar do *SequentialScheduler*.

Tabela 5.1: Resultados dos testes comparativos

Característica	**Comanche Sequencial**	**Comanche *MultiThread***
Média tempo de resposta (ms)	1,8	0,85
Desvio padrão do tempo de reposta (ms)	14,24	1,02
Memória consumida (MB)	5,34	7,6

Como pode ser visto, as suposições estão corretas. O componente *MultiThread* acaba sendo mais eficiente, porém consome 42% a mais de memória. Vale ressaltar que o processador utilizado nestes testes é *multicore*. Certamente, o resultado teria sido diferente em outro *hardware*. Como não é o objetivo deste trabalho comparar os dois componentes, vamos aceitar este resultado e assumir que o cenário de uso proposto é plausível.

[9]http://jmeter.apache.org

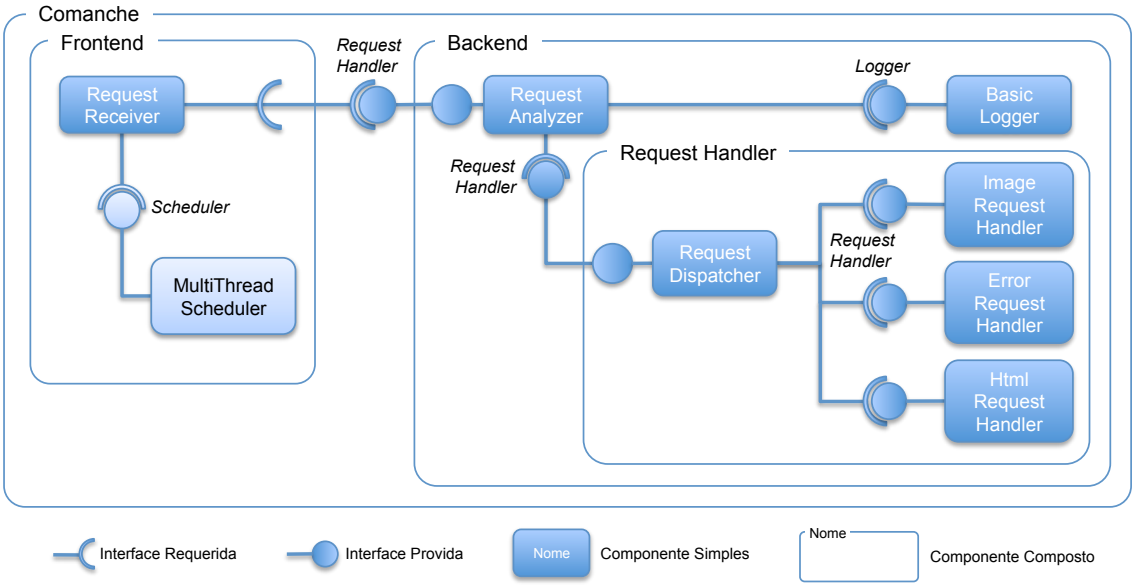

Figura 5.4: Comanche com *MultiThreadScheduler*

Para realizar a reconfiguração, trocando os componentes citados, o plano produzido foi o da Listagem 5.6. Neste plano, primeiramente é criado o novo componente, depois a nova ligação é estabelecida e, por fim, o componente anterior é removido da aplicação. Esta ordem garante que a aplicação não vai ficar inválida em nenhum momento. Este plano deve ser executado durante o atendimento das requisições. Para isto, a mesma infraestrutura foi utilizada, simulando o total de 1000 requisições.

Listagem 5.6: Plano para troca do *SequentialScheduler* pelo *MultiThreadScheduler*

```
1 (!createInstance multiThreadScheduler org.apache.comanche.
    multiThreadScheduler.MultiThreadScheduler comanche:frontend)
2 (!bind comanche:frontend:requestReceiver comanche:frontend:
    multiThreadScheduler org.apache.comanche.services.Scheduler)
3 (!destroyInstance comanche:frontend:sequentialScheduler)
```

O resultado da execução encontra-se na Figura 5.5, que mostra a execução da reconfiguração enquanto as requisições são atendidas. Para que a comparação fique mais fácil, a Figura 5.6 mostra em um mesmo gráfico o tempo de atendimento de requisição para cada uma das configurações, tanto para o componente sequencial quanto para o *MultiThread*. Nestas figuras, o eixo das ordenadas (Y) representa o intervalo de tempo de uma requisição e está representado em uma escala logarítmica.

Como pode ser visto, há uma mudança do comportamento da aplicação que passa a atender as requisições em um tempo menor por conta do componente *MultiThread*. No detalhe da figura, é possível ver um gráfico que representa o momento exato da reconfiguração que, neste caso, levou 13 milisegundos. Neste pequeno gráfico, estão representadas 10 requisições e é possível observar um *overshoot*. Este *overshoot* aparece em todas as execuções efetuadas e, neste caso, representou um aumento

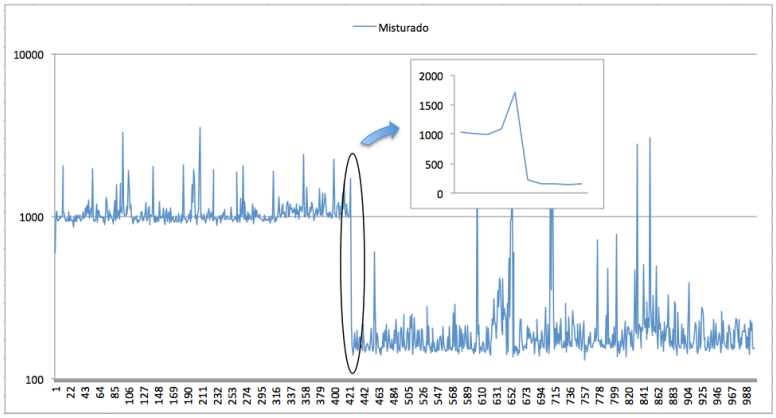

Figura 5.5: Tempos de resposta do Comanche durante a reconfiguração

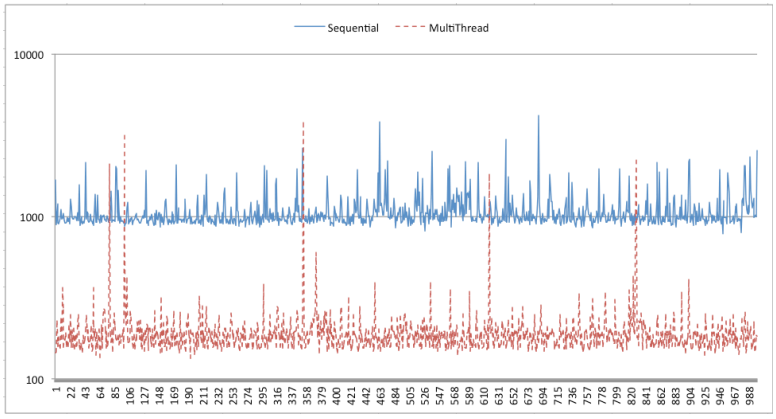

Figura 5.6: Tempos de resposta do Comanche nas configurações sequencial e *MultiThread*

de aproximadamente 0,6 ms em relação à média das execuções com o componente sequencial, que é 60% de aumento. Apesar de percentualmente alto, o aumento do tempo de execução desta requisição não foge dos outros picos, como pode-se ver na figura. Além disso, é importante ressaltar que nenhuma requisição foi perdida, segundo o JMeter.

Este resultado mostra que a troca da configuração arquitetural não afeta a performance da aplicação e isto representa um grande avanço em relação ao que existia sem o DC4iPOJO. Utilizando somente o iPOJO, este cenário pode ser reproduzido, especificando previamente que o componente que será ligado na interface *Scheduler* pode ser trocado, conforme mostrado em um trabalho anterior (FONSECA *et al.*, 2012). A Figura 5.7 mostra o resultado da execução deste cenário.

Pode-se ver na figura que a reconfiguração durou cerca de 55,5 milisegundos e causou uma grande dispersão nos tempos de resposta de várias requisições. Isto se dá porque a reconfiguração dinâmica no iPOJO, além de limitada aos pontos de variabilidade previamente estabelecidos, se dá por meios indiretos. Ou seja, não existe no iPOJO comandos explícitos de reconfiguração dinâmica que permitam ao

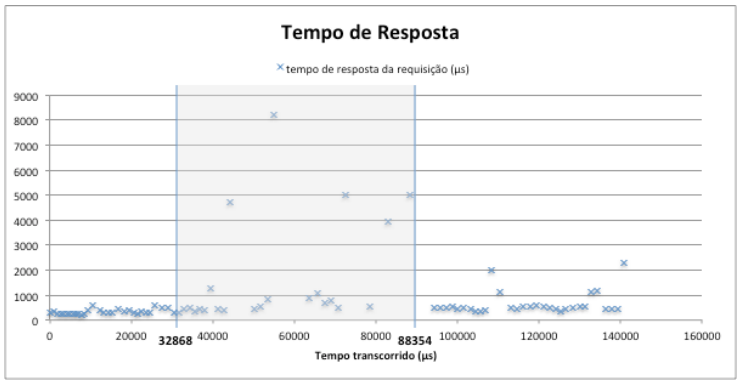

Figura 5.7: Tempo de resposta da reconfiguração indireta utilizando o iPOJO (FONSECA *et al.*, 2012)

desenvolvedor, ou ao usuário, realizar a mudança da configuração arquitetural diretamente. Neste caso, para provocar a alteração arquitetural desejada, é necessário tornar o componente sequencial indisponível para que, através dos mecanismos de eventos internos do OSGi, o iPOJO possa instanciar o componente *MultiThread* e realizar a ligação com o componente *RequestReceiver*, retornado a aplicação a uma configuração capaz de responder à requisições. Com o DC4iPOJO, isto não é necessário, já que ele provê meios diretos de realizar a ação de reconfiguração desejada.

5.2.4 Troca do componente *backend*

O terceiro cenário da prova de conceito é uma alteração mais radical no Comanche, trocando toda a árvore de componentes que se encontra dentro da composição *backend* por um componente não composto, que tem o objetivo de devolver para o navegador uma mensagem padrão, indicando que o servidor está em manutenção. Esta configuração arquitetural do Comanche pode ser vista na Figura 5.8.

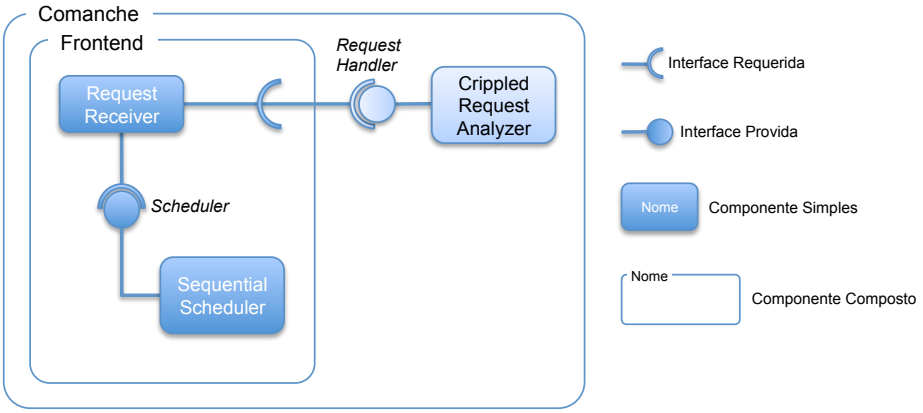

Figura 5.8: Configuração arquitetural do Comanche para manutenção

O resultado de uma requisição quando o Comanche está com esta configuração arquitetural pode ser visto na Figura 5.9. Este cenário não pode ser realizado no

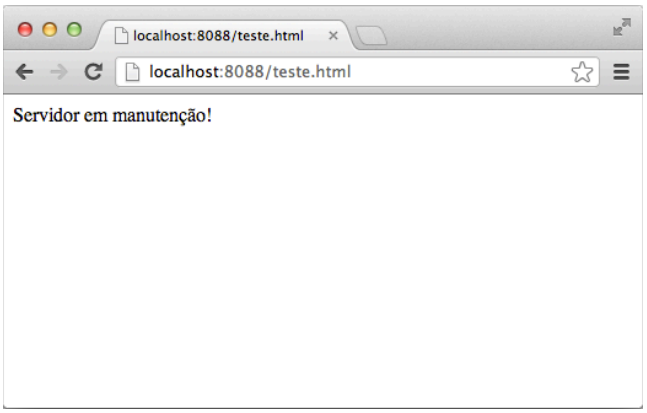

Figura 5.9: Navegador mostrando uma requisição realizada enquanto o Comanche está em manutenção

iPOJO sem as extensões desenvolvidas neste trabalho.

Este cenário de uso é factível considerando uma estrutura distribuída do Comanche, onde uma máquina fica responsável por receber as requisições e distribuí-las para outras várias máquinas que atenderiam a requisição em questão. Nesta topologia, o componente composto *backend* seria instanciado em várias máquinas e o *Scheduler* teria que ser alterado para distribuir as requisições para os vários *backends*, utilizando algum algoritmo de distribuição de carga como, por exemplo, *Round Robin*[10].

Uma estrutura distribuída do Comanche poderia ser conseguida utilizando o DOSGi[11] como camada inferior ao iPOJO. A rigor, como o DC4iPOJO é uma extensão para o iPOJO, deveria funcionar sem problemas sobre o DOSGi, mas este teste foge do escopo desta avaliação e será considerado um trabalho futuro.

Para realizar a reconfiguração necessária para levar o Comanche para a configuração arquitetural mostrada na Figura 5.8, o plano gerado pelo planejador foi o da Listagem 5.7.

As 17 ações deste plano foram executadas em 60 milisegundos, levando o Comanche para a configuração arquitetural de manutenção. Apenas para posicionar esta versão do Comanche em relação às outras, foi feita uma execução com as 1000 requisições e a Tabela 5.2 mostra o resultado em comparação com as demais versões.

Como nos demais cenários, foi realizado um teste capturando a evolução do tempo de resposta do Comanche enquanto ele sofria a reconfiguração da Listagem 5.7. O gráfico que representa esta operação pode ser visto na Figura 5.10.

[10]Neste algoritmo, a distribuição da carga é feita igualmente entre as máquinas, independente de qualquer fator modificador como, por exemplo, a utilização corrente da máquina

[11]http://felix.apache.org/site/apache-felix-ipojo-dosgi.html

Listagem 5.7: Plano para colocar o Comanche em manutenção

```
1  (!createInstance crippledRequestAnalyzer org.apache.comanche.
       crippledRequestAnalyzer.CrippledRequestAnalyzer comanche)
2  (!removeImportedService comanche:frontend org.apache.comanche.services.
       RequestHandler)
3  (!addImportedService comanche:frontend comanche:crippledRequestAnalyzer
        org.apache.comanche.services.RequestHandler)
4  (!bind comanche:frontend:requestReceiver comanche:
       crippledRequestAnalyzer org.apache.comanche.services.RequestHandler
       )
5  (!unbind comanche:backend:requestHandler:requestDispatcher comanche:
       backend:requestHandler:imageRequestHandler org.apache.comanche.
       services.RequestHandler)
6  (!unbind comanche:backend:requestHandler:requestDispatcher comanche:
       backend:requestHandler:errorRequestHandler org.apache.comanche.
       services.RequestHandler)
7  (!destroyInstance comanche:backend:requestHandler:imageRequestHandler)
8  (!destroyInstance comanche:backend:requestHandler:errorRequestHandler)
9  (!unbind comanche:backend:requestAnalyzer comanche:backend:
       requestHandler org.apache.comanche.services.RequestHandler)
10 (!removeExportedService comanche:backend:requestHandler org.apache.
       comanche.services.RequestHandler)
11 (!destroyInstance comanche:backend:requestHandler:requestDispatcher)
12 (!destroyInstance comanche:backend:requestHandler)
13 (!unbind comanche:backend:requestAnalyzer comanche:backend:basicLogger
       org.apache.comanche.services.Logger)
14 (!destroyInstance comanche:backend:basicLogger)
15 (!removeExportedService comanche:backend org.apache.comanche.services.
       RequestHandler)
16 (!destroyInstance comanche:backend:requestAnalyzer)
17 (!destroyInstance comanche:backend)
```

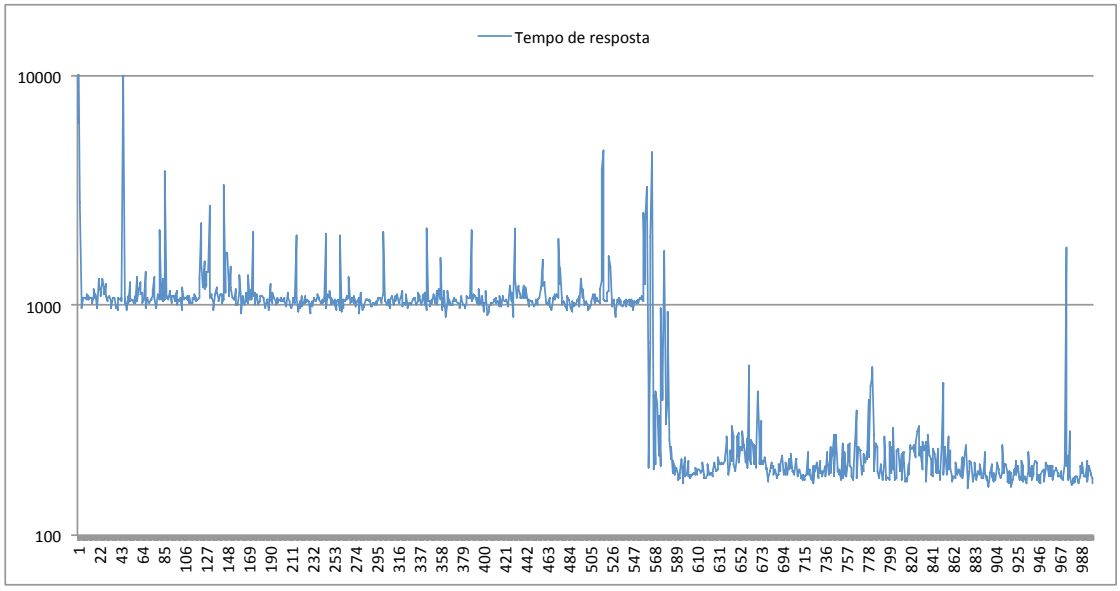

Figura 5.10: Tempo de resposta do Comanche versão Manutenção

Tabela 5.2: Comparação do desempenho das três versões do Comanche

Característica	**Comanche Sequencial**	**Comanche *MultiThread***	**Comanche Manutenção**
Média tempo de resposta (ms)	1,8	0,85	0,22
Desvio padrão do tempo de reposta (ms)	14,24	1,02	0,34
Memória consumida (MB)	5,34	7,6	1,2
Tempo total de execução (ms)	2062,18	2050,67	3051,06

5.2.5 Atuação do mecanismo de compensação

Por fim, este cenário visa demonstrar o funcionamento do mecanismo de compensação, que atua em caso de erros de reconfiguração. Neste cenário, é considerado o Comanche versão "Manutenção", resultado da execução do cenário anterior. A ideia é que, após a realização das manutenções programadas, o componente *backend* seja novamente instanciado, para que o Comanche volte a responder requisições (incluindo arquivos .html) de forma adequada. Para levar o Comanche "Manutenção" descrito pela configuração da Figura 5.8 de volta para a sua configuração completa (Figura 5.2), o plano da Listagem 5.8 é gerado pelo planejador.

Listagem 5.8: Plano para voltar o Comanche à sua configuração inicial

```
1  (!createInstance backend backendType comanche)
2  (!createInstance requestAnalyzer org.apache.comanche.requestAnalyzer.
       RequestAnalyzer comanche:backend)
3  (!createInstance basicLogger org.apache.comanche.basicLogger.
       BasicLogger comanche:backend)
4  (!createInstance requestHandler requestHandlerType comanche:backend)
5  (!createInstance requestDispatcher org.apache.comanche.
       requestDispatcher.RequestDispatcher comanche:backend:requestHandler
       )
6  (!createInstance imageRequestHandler org.apache.comanche.
       requestHandlers.ImageRequestHandler comanche:backend:requestHandler
       )
7  (!createInstance errorRequestHandler org.apache.comanche.
       requestHandlers.ErrorRequestHandler comanche:backend:requestHandler
       )
8  (!createInstance htmlRequestHandler org.apache.comanche.
       htmlRequestHandler.HtmlRequestHandler comanche:backend:
       requestHandler)
9  (!bind comanche:backend:requestHandler:requestDispatcher comanche:
       backend:requestHandler:imageRequestHandler org.apache.comanche.
       services.RequestHandler)
10 (!bind comanche:backend:requestHandler:requestDispatcher comanche:
       backend:requestHandler:errorRequestHandler org.apache.comanche.
       services.RequestHandler)
11 (!bind comanche:backend:requestHandler:requestDispatcher comanche:
       backend:requestHandler:htmlRequestHandler org.apache.comanche.
```

```
    services.RequestHandler)
12 (!addExportedService comanche:backend:requestHandler requestDispatcher
    org.apache.comanche.services.RequestHandler)
13 (!bind comanche:backend:requestAnalyzer comanche:backend:requestHandler
     org.apache.comanche.services.RequestHandler)
14 (!bind comanche:backend:requestAnalyzer comanche:backend:basicLogger
    org.apache.comanche.services.Logger)
15 (!addExportedService comanche:backend requestAnalyzer org.apache.
    comanche.services.RequestHandler)
16 (!bind comanche:frontend:requestReceiver comanche:frontend:
    sequentialScheduler org.apache.comanche.services.Scheduler)
17 (!removeImportedService comanche:frontend org.apache.comanche.services.
    RequestHandler)
18 (!addImportedService comanche:frontend comanche:backend org.apache.
    comanche.services.RequestHandler)
19 (!bind comanche:frontend:requestReceiver comanche:backend org.apache.
    comanche.services.RequestHandler)
20 (!destroyInstance comanche:crippledRequestAnalyzer)
```

Este plano recria a instância do componente *backend* e toda a sua estrutura interna, e o liga ao *RequestReceiver*, instanciado dentro da composição *frontend*. No fim, o plano remove a instância *crippleRequestAnalyzer*.

Para provocar um erro de reconfiguração e acionar o mecanismo de compensação, o tipo *HtmlRequestHandler* será artificialmente removido (através da indisponibilização do *bundle* que o define). Deste modo, a linha 8 deste plano provocará um erro, o que dará início do processo de compensação, que deve desfazer as ações realizadas das linhas 1 a 7. Para mostrar este processo em execução, o mecanismo de *log* do Escalonador foi configurado para o nível mais baixo, fornecendo mensagens de *DEBUG* que permitem a visualização da execução. A Listagem 5.9 mostra a execução do mecanismo de compensação.

Listagem 5.9: Plano para voltar o Comanche à sua configuração inicial

```
1 [ExecutionManager] Executing reconfiguration actions...
2 [ExecutionManager] Executing command createInstance
3 [CreateInstanceCommand] Type backendType found!
4 [ExecutionManager] Executing command createInstance
5 [CreateInstanceCommand] Type org.apache.comanche.requestAnalyzer.
    RequestAnalyzer found!
6 [ExecutionManager] Executing command createInstance
7 [CreateInstanceCommand] Type org.apache.comanche.basicLogger.
    BasicLogger found!
8 [ExecutionManager] Executing command createInstance
9 [CreateInstanceCommand] Type requestHandlerType found!
10 [ExecutionManager] Executing command createInstance
```

```
11 [CreateInstanceCommand] Type org.apache.comanche.requestDispatcher.
       RequestDispatcher found!
12 [ExecutionManager] Executing command createInstance
13 [CreateInstanceCommand] Type org.apache.comanche.requestHandlers.
       ImageRequestHandler found!
14 [ExecutionManager] Executing command createInstance
15 [CreateInstanceCommand] Type org.apache.comanche.requestHandlers.
       ErrorRequestHandler found!
16 [ExecutionManager] Executing command createInstance
17 [CreateInstanceCommand] Type org.apache.comanche.htmlRequestHandler.
       HtmlRequestHandler not found! Throwing an exception!
18 ERROR! Found an error, performing rollback! Error message:
19 ERROR! InvalidPreconditionException during adaptation:
20 ERROR! The type org.apache.comanche.htmlRequestHandler.
       HtmlRequestHandler was not found!
21 [ExecutionManager] Executing rollback...
22 [ExecutionManager] Found rollback command destroyInstance!
23 [ExecutionManager] Rollback command destroyInstance executed
       successfully!
24 [ExecutionManager] Found rollback command destroyInstance!
25 [ExecutionManager] Rollback command destroyInstance executed
       successfully!
26 [ExecutionManager] Found rollback command destroyInstance!
27 [ExecutionManager] Rollback command destroyInstance executed
       successfully!
28 [ExecutionManager] Found rollback command destroyInstance!
29 [ExecutionManager] Rollback command destroyInstance executed
       successfully!
30 [ExecutionManager] Found rollback command destroyInstance!
31 [ExecutionManager] Rollback command destroyInstance executed
       successfully!
32 [ExecutionManager] Found rollback command destroyInstance!
33 [ExecutionManager] Rollback command destroyInstance executed
       successfully!
34 [ExecutionManager] Found rollback command destroyInstance!
35 [ExecutionManager] Rollback command destroyInstance executed
       successfully!
```

Neste listagem, é possível ver na linha 17 o erro provocado pela não existência do tipo de componente *HtmlRequestHandler*. Este erro é descrito nas linhas 18, 19 e 20, e o mecanismo de compensação é executado a partir da linha 21.

5.3 Considerações Finais

Neste capítulo, foi realizada a avaliação das contribuições deste trabalho através de uma Prova de Conceito. Provas de conceito são bastante utilizadas nos trabalhos de

Reconfiguração Dinâmica e visam mostrar as capacidades das abordagens propostas.

Neste trabalho, a prova de conceito consistiu em 3 cenários de reconfiguração dinâmica de uma aplicação chamada Comanche. O Comanche é um servidor *web* simples originalmente escrito em Python, mas portado para Java. É bastante utilizado para demonstrar as capacidades de reconfiguração dinâmica por conta da sua implementação ser componentizada e de fácil modificação.

O primeiro cenário da prova de conceito foi a adição de um componente, um novo *RequestHandler* capaz de processar arquivos HTML, em uma situação hipotética, onde o Comanche não teria este componente pré-carregado. Este cenário mostrou a capacidade mais básica de um *framework* de reconfiguração dinâmica, que é a adição e remoção de instâncias de componente de um sistema em execução.

O segundo cenário focou em mostrar o baixo impacto de performance conferido pela utilização do DC4iPOJO. Este cenário ilustra a troca de um componente enquanto o Comanche atendia a requisições. Foi mostrado que a reconfiguração é executada rapidamente, não produzindo impactos notáveis no tempo de resposta das requisições. Este cenário foi comparado com um trabalho anterior, onde esta mesma reconfiguração era realizada de maneira indireta, utilizando apenas o iPOJO. Mostrou-se que as novas capacidades de reconfiguração dinâmica trazidas pelo DC4iPOJO diminuem muito o impacto no sistema, sendo reconfigurado em comparação ao que havia antes no iPOJO.

O terceiro cenário buscou exemplificar a utilização do DC4iPOJO em uma reconfiguração mais radical, onde mais da metade da aplicação é alterada em tempo de execução. Este cenário também foi acompanhado de uma análise da variação do tempo de resposta provocado pela execução do plano de reconfiguração, que, desta vez, era mais extenso se comparado aos demais. Mais uma vez, mostrou-se que não foram produzidos impactos notáveis. Vale ressaltar que este cenário não é reproduzível no iPOJO.

Através destes cenários ficou claro que as contribuições deste trabalho adicionam interessantes capacidades de reconfiguração à plataforma OSGi/iPOJO e podem vir a ser utilizadas por uma grande comunidade de usuários do OSGi.

Capítulo 6

Conclusão

Este trabalho de pesquisa está inserido no contexto de Reconfiguração Dinâmica aplicada a sistemas adaptáveis, criados de acordo com o paradigma de DBC (Desenvolvimento Baseado em Componentes) (SZYPERSKI, 2002) como uma forma de executar uma adaptação.

Neste sentido, o trabalho abordou inicialmente o tema de Sistemas Auto-Adaptáveis, considerando suas principais características, taxonomias e classificações (SALEHIE e TAHVILDARI, 2009). Sistemas Auto-Adaptáveis são, por natureza, sistemas sensíveis ao contexto de execução, de onde monitoram informações relevantes para a decisão da adaptação. A adaptação, por sua vez, é determinada por um ciclo que compreende as ações de Monitorar, Analizar, Planejar e Executar, tendo como base um conjunto de Conhecimentos sobre o sistema. Este ciclo é conhecido na área de computação autonômica como MAPE-K (KEPHART e CHESS, 2003).

Além disso, a adaptação pode ser conseguida em vários níveis diferentes de abstração (OREIZY *et al.*, 1999), desde abordagens que focam em alterar diretamente o código do sistema, até abordagens composicionais (MCKINLEY *et al.*, 2004) de mais alto nível que buscam a adaptação através da alteração da arquitetura do sistema, descrita por componentes e conectores. Segundo KRAMER e MAGEE (2007), este nível de abstração possui várias vantagens para a obtenção da adaptação.

Após um breve levantamento buscando soluções arquiteturais existentes na área, foi percebido que vários trabalhos foram publicados nesta linha, visando propor diferentes estratégias para atender o ciclo de adaptação como um todo, focando em decidir o melhor momento e a melhor maneira para se chegar à adaptação. Porém, pouca atenção era dada à aspectos ligados à execução da Adaptação, que ainda permanece um desafio (CHENG *et al.*, 2009). Aprofundando mais os estudos neste assunto, chegou-se à área de Reconfiguração Dinâmica.

Reconfigurar um sistema dinamicamente significa alterá-lo em tempo de execução (HOFMEISTER, 1993). Do ponto de vista adotado neste trabalho, a Reconfiguração Dinâmica consiste em alterar a estrutura do sistema, mudando sua arquitetura em

tempo de execução.

Esta pesquisa adotou a abordagem proposta em (DI BENEDITTO e WERNER, 2012), que segue esta linha e utiliza técnicas de Planejamento, uma área da Inteligência Artificial, para resolver o problema de se obter o plano de reconfiguração dada uma configuração objetivo. Nesta abordagem, o Planejador é a entidade responsável por obter, se existir, um plano de reconfiguração, composto por ações de reconfiguração, que leva o sistema do estado inicial até o estado objetivo, sendo o estado uma representação da configuração arquitural do sistema.

Uma vez obtido, o plano é executado por um Escalonador, foco principal desta dissertação de mestrado.

Como plataforma escolhida para a implementação do Escalonador e de todos os seus requisitos, definidos neste trabalho, foi escolhido o OSGi, por ser uma plataforma dinâmica orientada a serviço e largamente utilizada na academia e na indústria. Como extensão do seu modelo de componentes, foi adotado o iPOJO, que é um *framework* de componentes hierárquico que utiliza injeção de dependências e manipulação de *bytecode* Java para possibilitar as ligações entre componentes e seu encapsulamento em composições.

As próximas seções descrevem as contribuições deste trabalho (Seção 6.1), suas limitações (Seção 6.2) e trabalhos futuros (Seção 6.3).

6.1 Contribuições

Considerando os objetivos **O1** e **O2** estabelecidos no Capítulo 1, o resultado deste trabalho pode ser sumarizado pelas seguintes contribuições:

- Definição de um mecanismo para a execução de planos de reconfiguração arquiteturais em sistemas dinâmicos, para o *framework* de componentes iPOJO, que estende as capacidades arquiteturais do OSGi.
  - A abordagem considera o comportamento dinâmico do sistema, que pode ter sua configuração arquitetural alterada ao longo do tempo por eventos externos. Isto deve ser considerado ao executar um plano de reconfiguração, cujas ações podem não ser mais aplicáveis ao sistema, dada sua configuração arquitetural corrente. Uma ação é aplicável se o seu conjunto de precondições é verdadeiro no momento da sua execução.
  - O mecanismo, chamado de Escalonador, foi criado para ser estendido, utilizando conceitos relacionados à área de *framework* da Engenharia de Software, no sentido de ser uma aplicação inacabada (BUSCHMANN *et al.*, 1996). Possui *coldspots* onde reside a funcionalidade básica e *hotspots* que devem ser implementados por meio de extensões das suas classes.

- Extensão para o iPOJO, adicionando a infraestrutura necessária para a reconfiguração dinâmica de aplicações criadas através do seu modelo de componentes hierárquico.
  - Criação de uma API para acesso à estas funcionalidades.
  - Levantamento e definição de um conjunto de ações de reconfiguração dinâmica, através de um levantamento dos trabalhos encontrados.
- Realização de uma prova de conceito contando com um conjunto de cenários que visam demonstrar o correto funcionamento dos artefatos de software construídos.

6.2 Limitações da Pesquisa

Como qualquer pesquisa com um tempo limitado para a sua finalização, alguns itens que representam problemas de pesquisa não foram abordados por restrição do escopo do trabalho. Além disso, algumas atividades não puderam ser realizadas e não foram incluídas no produto final.

- O modelo de componentes do iPOJO e do OSGi não considera o conector como uma entidade de primeiro nível, diferentemente de outros *frameworks* analisados, como é o do SOFA 2.0 (BURES *et al.*, 2006). Por este motivo, a ação de Alteração de Conector não foi considerada;
- Algumas abordagens da linha de Reconfiguração Dinâmica tocam no tema da transferência do estado de execução entre componentes em uma reconfiguração. Este problema envolve a definição da compatibilidade entre estados e uma possível transformação para que o novo componente possa receber o estado do componente anterior. Isto não foi abordado neste trabalho;
- Não foi abordado neste trabalho o cenário de um sistema distribuído, que é bastante relevante considerando sistemas complexos e críticos que requerem evolução em tempo de execução. Este cenário traz algumas dificuldades extras e desafios próprios que não foram considerados no escopo desta dissertação.

6.3 Trabalhos Futuros

A partir desta pesquisa foram identificados alguns trabalhos a serem desenvolvidos futuramente. As próximas seções detalham estas oportunidades.

6.3.1 Conector como uma entidade de primeira ordem

A questão do conector ser uma entidade de primeira ordem traz uma maior flexibilidade para um sistema, que pode alterar o seu estilo de comunicação de maneira transparente para o componente. Isto possibilita, por exemplo, que dois componentes antes no mesmo ambiente possam ser separados em ambientes distribuídos e continuarem a se comunicar sem notar a mudança. Neste caso, apenas o conector seria alterado. No cenário original, o conector faria uma simples ligação entre dois componentes. Já no cenário distribuído, deveria, por exemplo, realizar a invocação de uma operação de um *web-service*.

Alterações de conectores podem trazer desafios próprios. No cenário descrito acima, uma chamada do método de um componente localizado no mesmo ambiente de execução é síncrona. Já a chamada de um método de um componente externo pode passar a ser assíncrona. Uma opção seria fazer com que o conector encapsule esta transformação de chamada síncrona para assíncrona, aguardando a resposta enquanto prende a execução do componente que realizou a chamada. Mas isto pode levar à problemas de performance.

Outra opção seria alterar o comportamento do componente que realiza a chamada para passar a considerar que a resposta da chamada virá em um segundo momento. Mas, neste caso, a reconfiguração não seria apenas no conector e poderia mudar o comportamento de todo o sistema. É preciso analisar opções e verificar soluções já existentes para este caso.

6.3.2 Transferência de estado entre instâncias de componente

A transferência de estado entre componentes já foi tema de alguns trabalhos na linha de Reconfiguração Dinâmica. No Argus (BLOOM, 1983), os autores definem a troca de estado entre componentes através de chamadas específicas à funções destes componentes. Já no Polylith (HOFMEISTER, 1993), é necessário que o componente antigo codifique o estado que, posteriormente, é decodificado e injetado no novo componente. Em (TEWKSBURY *et al.*, 2001), os autores definem um mapeamento entre as variáveis de estado do componente anterior e o novo, o que faz com que seja necessário um trabalho do usuário para definir manualmente a troca de estados.

Em todos estes trabalhos, o nível de automação na transferência de estado é, no máximo, semi-automática, requerendo algum tipo de customização do componente ou do processo para que a reconfiguração seja possível.

Na presente pesquisa, não foi realizada uma revisão extensa deste tema para avaliar que abordagens existem e a sua aplicabilidade no iPOJO. Uma pesquisa futura poderia ir nesta linha, realizando uma revisão dos trabalhos já publicados, verificando sua aplicabilidade para o iPOJO.

6.3.3 Movimentação de instâncias de componente entre diferentes composições

A reconfiguração de um sistema pode consistir apenas em uma redefinição da sua topologia arquitetural, considerando apenas um reagrupamento dos componentes já existentes. A presente pesquisa não abordou este aspecto.

Um trabalho futuro seria a criação de uma ação de "movimentação de componente" que poderia fazer com que componentes migrassem entre composições ou de um nível hierárquico para outro.

Uma extensão desta ação seria a movimentação de componentes entre diferentes ambientes de execução. No caso do iPOJO, entre diferentes JVMs. Isto poderia implicar na criação de uma nova instância de componente na nova localização e posterior transferência do seu estado. Outra opção seria mover de fato a instância de um ambiente para o outro. Seria necessário um estudo aprofundado para entender as implicações desta ação.

6.3.4 Avaliação do DC4iPOJO em um ambiente distribuído

O OSGi, através da sua extensão chamada D-OSGi, no contexto do Apache CXF[1], possui suporte à sistemas distribuídos.

No entanto, estes sistemas possuem desafios próprios com problemas de pesquisa únicos para a área de reconfiguração dinâmica (HOFMEISTER, 1993). Esta área pode ser melhor explorada, dada a sua importância para sistemas complexos que precisem ser distribuídos em diferentes máquinas, seja por requisitos de alta disponibilidade, ou por conta do alto grau de complexidade de seus algoritmos.

6.3.5 Extensões para o componente "Receptor de Planos"

O componente "Receptor de Planos" projetado para o Escalonador realiza a tradução do plano de reconfiguração gerado pelo Planejador em ações de reconfiguração próprias do DC4iPOJO.

Este componente, entretanto, é estensível e pode ser utilizado para outros fins. Por exemplo, poderia ser utilizado em outras abordagens onde a configuração arquitetural do sistema é descrita por uma AML (*Architectural Modification Language*).

6.3.6 Abordagens para tratamento de erro de reconfiguração

O componente "Gerenciador de Execução" também é estensível e pode ser utilizado para implementação de outras abordagens para o tratamento do erro de reconfi-

[1] http://cxf.apache.org/distributed-osgi.html

guração. Uma ideia nesta linha é a utilização de um planejador online, que seria responsável por obter um novo plano de reconfiguração adequado à configuração arquitetural corrente do sistema, considerando o estado de erro.

Esta opção requer um planejador online, devendo o JSHOP2 ser customizado para este fim. Outra opção seria a utilização de outros planejadores que contassem com esta capacidade.

Referências Bibliográficas

ALIA, M., EIDE, V. S. W., PASPALLIS, N., et al., 2007, "A utility-based adaptivity model for mobile applications". In: *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops, AINAW '07*, pp. 556–563, Niagara Falls, Ontario, Canada. IEEE Computer Society. doi: 10.1109/AINAW.2007.64. Disponível em: <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4224162>.

BLOOM, T., 1983, *Dynamic Module Replacement in a Distributed Programming System*. Relatório técnico, Cambridge, MA, USA.

BRUNETON, E., COUPAYE, T., LECLERCQ, M., et al., 2006, "The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems", *Software-Practice & Experience*, v. 36, n. 11-12, pp. 1257–1284. ISSN: 0038-0644. doi: 10.1002/spe.v36:11/12. Disponível em: <http://dl.acm.org/citation.cfm?id=1152345>.

BURES, T., HNETYNKA, P., PLASIL, F., 2006, "SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model", *Fourth International Conference on Software Engineering Research, Management and Applications*, pp. 40–48. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1691359>.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., et al., 1996, *Pattern-oriented software architecture: a system of patterns*. Chichester, England, John Wiley & Sons, Inc. ISBN: 0471958697.

CHENG, B. H., LEMOS, R., GIESE, H., et al., 2009, "Software Engineering for Self-Adaptive Systems: A Research Roadmap". In: *Software Engineering for Self-Adaptive Systems*. Disponível em: <http://portal.acm.org/citation.cfm?id=1573858>.

DAVID, P.-C., LEDOUX, T., 2006, "An aspect-oriented approach for developing self-adaptive fractal components", *Software Composition*, pp.

82–97. Disponível em: <http://www.springerlink.com/index/Y5T247HW88665046.pdf>.

DAVID, P.-C., LEDOUX, T., LÉGER, M., et al., 2009, "FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures", *annals of telecommunications - annales des télécommunications*, v. 64, n. 1-2, pp. 45–63. ISSN: 0003-4347. doi: 10.1007/s12243-008-0073-y. Disponível em: <http://link.springer.com/article/10.1007%2Fs12243-008-0073-y>.

DE LEMOS, R., GIESE, H., MÜLLER, H., et al., 2011, "Software Engineering for Self-Adaptive Systems: A second Research Roadmap". In: de Lemos, R., Giese, H., Müller, H., et al. (Eds.), *Software Engineering for Self-Adaptive Systems*, n. 10431, Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. Disponível em: <http://drops.dagstuhl.de/opus/volltexte/2011/3156>.

DEY, A. K., ABOWD, G. D., SALBER, D., 2001, "A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications", *Human-Computer Interaction*, v. 16, n. 2, pp. 97–166. ISSN: 0737-0024. doi: 10.1207/S15327051HCI16234_02. Disponível em: <http://dl.acm.org/citation.cfm?id=1463110>.

DI BENEDITTO, M. E. M., WERNER, C. M. L., 2012, "A Declarative Approach for Software Compositional Reconfiguration". In: *Proceedings of the 11th International Workshop on Adaptive and Reflective Middleware*, ARM '12, pp. 7:1–7:6, Montreal, Quebec, Canada. ACM. ISBN: 978-1-4503-1609-5. doi: 10.1145/2405679.2405686. Disponível em: <http://doi.acm.org/10.1145/2405679.2405686>.

DIACONESCU, A., BOURCIER, J., ESCOFFIER, C., 2008, "Autonomic iPOJO: Towards Self-Managing Middleware for Ubiquitous Systems", *2008 IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, (out.), pp. 472–477. doi: 10.1109/WiMob.2008.89. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4654284>.

ESCOFFIER, C., BOURCIER, J., LALANDA, P., et al., 2008, "Towards a Home Application Server". In: *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pp. 321–325. doi: 10.1109/ccnc08.2007.78.

ESCOFFIER, C., HALL, R. S., 2007, "Dynamically adaptable applications with iPOJO service components", *Software Composition*, v. 4829/2007, pp. 113–128. doi: 10.1007/978-3-540-77351-1_9. Disponível em: <http://www.springerlink.com/index/9r3u86q510178084.pdf>.

ESCOFFIER, C., HALL, R. S., LALANDA, P., 2007, "iPOJO: An extensible service-oriented component framework". In: *Services Computing, 2007. SCC 2007. IEEE International Conference on*, n. Scc, pp. 474–481, Salt Lake City, Utah, USA. ISBN: 0769529259. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4278693>.

FERREIRA, J., LEITAO, J., RODRIGUES, L., 2009, "A-OSGi: A framework to support the construction of autonomic OSGi-based applications". In: *Proceedings of the Third International ICST Conference on Autonomic Computing and Communication Systems*, pp. 1–16, Limassol, Cyprus. Disponível em: <http://www.springerlink.com/index/u3313v0836276144.pdf>.

FLOCH, J., HALLSTEINSEN, S., STAV, E., et al., 2006, "Using architecture models for runtime adaptability", *IEEE Software*, v. 23, n. 2, pp. 62–70. doi: 10.1109/MS.2006.61.

FONSECA, F. L., DI BENEDITTO, M. E. M., WERNER, C. M. L., 2012, "Um mecanismo extensível para a execução de um plano de reconfiguração arquitetural sob o framework OSGi+iPOJO". In: *III Workshop sobre Sistemas de Software Autônomos (Autosoft)/III Congresso Brasileiro de Software: Teoria e Prática*, pp. 1–10, Natal, RN.

GAMMA, E., HELM, R., JOHNSON, R., et al., 1994, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-63361-2.

GARLAN, D., 2000, "Software architecture: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*, pp. 91–101. ACM. ISBN: 1581132530. Disponível em: <http://portal.acm.org/citation.cfm?id=336512.336537&type=series>.

GHALLAB, M., NAU, D., TRAVERSO, P., 2004, *Automated Planning: Theory & Practice*. 1st ed. San Francisco, California, United States, Morgan Kaufmann Publishers. ISBN: 9781558608566.

HALL, R. S., PAULS, K., MCCULLOCH, S., et al., 2011, *OSGi in Action: Creating Modular Applications in Java*. Manning Pubs Co Series. 1st ed.

Stamford, CT, USA, Manning. ISBN: 9781933988917. Disponível em: <http://books.google.com.br/books?id=y3lPPgAACAAJ>.

HALLSTEINSEN, S., FLOCH, J., STAV, E., 2005, "A middleware centric approach to building self-adapting systems". In: *Proceeding SEM'04 Proceedings of the 4th international conference on Software Engineering and Middleware*, pp. 107–122, Linz, Austria. Springer. Disponível em: <http://dl.acm.org/citation.cfm?id=2136576>.

HALLSTEINSEN, S., HINCHEY, M., PARK, S., et al., 2008, "Dynamic software product lines", *Computer*, v. 41, n. 4, pp. 93–95. ISSN: 0018-9162. doi: 10.1109/MC.2008.123. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4488260>.

HNĚTYNKA, P., PLÁŠIL, F., 2006, "Dynamic Reconfiguration and Access to Services in Hierarchical Component Models". In: *CBSE'06 Proceedings of the 9th international conference on Component-Based Software Engineering*, pp. 352–359, Västeras, Sweden. Disponível em: <http://dl.acm.org/citation.cfm?id=2171395>.

HOFMEISTER, C. R., 1993, *Dynamic Reconfiguration of Distributed Applications*. Relatório técnico, Department of Computer Science, University of Maryland, Maryland, MD, USA, jan.

INVERARDI, P., TIVOLI, M., 2009, "The Future of Software: Adaptation and Dependability", *Software Engineering*, pp. 1–31. Disponível em: <http://www.springerlink.com/index/G624T1466M9V5647.pdf>.

KEPHART, J. O., CHESS, D. M., 2003, "The Vision of Autonomic Computing", *Computer*, v. 36, n. 1, pp. 41–50. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1160055>.

KETFI, A., BELKHATIR, N., CUNIN, P.-Y., 2002, "Automatic adaptation of component-based software - Issues and Experiences". In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '02*, pp. 1365–1371, Las Vegas, Nevada, USA. CSREA Press. Disponível em: <http://adele.imag.fr/Les.Publications/intConferences/PDPTA2002Ket.pdf>.

KRAMER, J., MAGEE, J., 2007, "Self-Managed Systems: an Architectural Challenge", *Future of Software Engineering, 2007. FOSE'07*, pp. 259–268. Disponível em: <http://www.computer.org/portal/web/csdl/doi/10.1109/FOSE.2007.19>.

LADDAGA, R., ROBERTSON, P., SHROBE, H., 2003, "Introduction to Self-adaptive Software: Applications". In: Laddaga, R., Shrobe, H., Robertson, P. (Eds.), *Proceeding IWSAS'01 Proceedings of the 2nd international conference on Self-adaptive software: applications*, v. 2614, *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 1–5. ISBN: 978-3-540-00731-9. doi: 10.1007/3-540-36554-0_1. Disponível em: <http://dl.acm.org/citation.cfm?id=1754789>.

LÉGER, M., LEDOUX, T., COUPAYE, T., 2010, "Reliable dynamic reconfigurations in a reflective component model", *Component-Based Software*, pp. 74–92. Disponível em: <http://www.springerlink.com/index/58HQ758200009817.pdf>.

MAES, P., 1987, "Concepts and Experiments in Computational Reflection", *OOPSLA '87 Conference proceedings on Object-oriented programming systems, languages and applications*, v. 22, n. 12 (out.), pp. 147–155. ISSN: 0885-8985. doi: 10.1145/38765.38821. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1581106>.

MCKINLEY, P. K., SADJADI, S. M., KASTEN, E. P., et al., 2004, "Composing adaptive software", *Computer*, v. 37, n. 7 (jul.), pp. 56–64. ISSN: 0018-9162. doi: 10.1109/MC.2004.48. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1310241>.

MENCL, V., BURES, T., 2005, "Microcomponent-Based Component Controllers: A Foundation for Component Aspects". In: *Proceeding APSEC '05 Proceedings of the 12th Asia-Pacific Software Engineering Conference*, n. 46127110, pp. 729–737, Taipei, Taiwan. Society Press. doi: 10.1109/APSEC.2005.78. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1607215>.

OGATA, K., 1997, *Modern control engineering*. 3rd. ed. Upper Saddle River, NJ, USA, Prentice-Hall, Inc. ISBN: 0-13-227307-1.

OREIZY, P., 1996, *Issues in the runtime modification of software architectures*. Relatório técnico, University of California, Irvine, CA, USA. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.9227&rep=rep1&type=pdf>.

OREIZY, P., MEDVIDOVIC, N., TAYLOR, R. N., 1998, "Architecture-based runtime software evolution". In: *Software Engineering, 1998. Proceedings*

of the 1998 International Conference on, pp. 177–186. IEEE. ISBN: 0818683686. doi: 10.1109/ICSE.1998.671114. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=671114>.

OREIZY, P., GORLICK, M. M., TAYLOR, R. N., et al., 1999, "An architecture-based approach to self-adaptive software", *Intelligent Systems and their Applications, IEEE*, v. 14, n. 3, pp. 54–62. doi: 10.1109/5254.769885. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=769885>.

ROUVOY, R., ELIASSEN, F., FLOCH, J., et al., 2008, "Composing components and services using a planning-based adaptation middleware". In: *Software Composition*, pp. 52–67. Springer. Disponível em: <http://www.springerlink.com/index/b349n14035v824t5.pdf>.

ROUVOY, R., BARONE, P., DING, Y., et al., 2009, "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments", *Software Engineering for Self-Adaptive Systems*, v. 5525, pp. 164–182. doi: 10.1007/978-3-642-02161-9. Disponível em: <http://www.springerlink.com/content/q2835ph217n66n64/>.

SALEHIE, M., TAHVILDARI, L., 2009, "Self-Adaptive Software: Landscape and Research Challenges", *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, v. 4, n. 2. Disponível em: <http://dl.acm.org/citation.cfm?id=1516538>.

SEINTURIER, L., MERLE, P., ROUVOY, R., et al., 2012, "A component-based middleware platform for reconfigurable service-oriented architectures", *Journal Software—Practice & Experience*, v. 42, n. 5, pp. 559–583. doi: 10.1002/spe. Disponível em: <http://onlinelibrary.wiley.com/doi/10.1002/spe.1077/full>.

SHAW, M., GARLAN, D., 1996, *Software Architecture: Perspectives on an Emerging Discipline*. USA, Prentice Hall. ISBN: 0131829572. Disponível em: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0131829572>.

STAEHLI, R., ELIASSEN, F., 2002, *QuA: A QoS-Aware Component Architecture*. Relatório Técnico 2002-12, Simula Research Laboratory.

SZYPERSKI, C., 2002, *Component Software: Beyond Object-Oriented Programming*. 2nd ed. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201745720.

TAJALLI, H., GARCIA, J., EDWARDS, G., et al., 2010, "PLASMA: a plan-based layered architecture for software model-driven adaptation", *on Automated software*, pp. 467–476. Disponível em: <http://dl.acm.org/citation.cfm?id=1859092>.

TAYLOR, R. N., MEDVIDOVIC, N., DASHOFY, E. M., 2009, *Software Architecture: Foundations, Theory, and Practice*. Hoboken, NJ, USA, Wiley. ISBN: 0470167742. Disponível em: <http://portal.acm.org/citation.cfm?id=1538494>.

TEWKSBURY, L. A., MOSER, L. E., MELLIAR-SMITH, P. M., 2001, "Live Upgrade Techniques for CORBA Applications". In: *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pp. 257–272, Deventer, The Netherlands. Kluwer, B.V. ISBN: 0-7923-7481-9. Disponível em: <http://dl.acm.org/citation.cfm?id=648289.754234>.

VANDEWOUDE, Y., 2007, *Dynamically updating component-oriented systems*. Tese de Doutorado, Katholieke Universiteit Leuven.

WALSH, W. E., TESAURO, G., KEPHART, J. O., et al., 2004, "Utility functions in autonomic systems". In: *Autonomic Computing, 2004. Proceedings. International Conference on*, pp. 70–77. doi: 10.1109/ICAC.2004.1301349.

WEISER, M., 1993, "Hot topics-ubiquitous computing", *Computer*, v. 26, n. 10, pp. 71–72. ISSN: 0018-9162. doi: 10.1109/2.237456. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=237456&tag=1>.

Apêndice A

Detalhamento COMANCHE - Código Java e Configuração dos Componentes iPOJO

A.1 Componente *RequestReceiver*

A.1.1 Código

```
1  package org.apache.comanche.requestReceiver;
2
3  import org.apache.comanche.services.RequestHandler;
4  import org.apache.comanche.services.Request;
5  import org.apache.comanche.services.Scheduler;
6  import java.io.IOException;
7  import java.net.ServerSocket;
8  import java.net.Socket;
9  import java.net.SocketException;
10
11 public class RequestReceiver implements Runnable {
12   private Scheduler s;
13   private RequestHandler rh;
14   private ServerSocket ss;
15
16   public void start() {
17     run();
18   }
19
20   public void stop() {
21     stopRequestReceiver();
22   }
23
24   private void myDo() throws IOException {
```

```
25      try {
26        final Socket socket = ss.accept();
27        s.schedule(new Runnable() {
28          public final void run() {
29            try {
30              rh.handleRequest(new Request(socket));
31            } catch (IOException exc) {
32            }
33          }
34        });
35      } catch (NullPointerException e) {
36        System.out.println("ERROR: No scheduler found!");
37        e.printStackTrace();
38      }
39    }
40
41    // ---------------------------------------------------------------
42    // Implementation of the Runnable interface
43
44    public final void run() {
45
46      new Thread() {
47        public void run() {
48          try {
49            ss = new ServerSocket(8088);
50            System.out.println("Comanche HTTP Server ready on port 8088."
                  );
51            System.out.println("Load http://localhost:8088/gnu.jpg");
52            while (true) {
53              // At this point we had to alter the original code.
54              // If the socket was opened in this main thread, it
55              // would lock the shell were this component was
56              // instantiated because of theiPOJO's mechanism of
57              // thread caching.
58              myDo();
59            }
60          } catch (SocketException e) {
61            System.out.println("Socket closed... going out!");
62            e.printStackTrace();
63          } catch (IOException e) {
64            e.printStackTrace();
65          }
66        }
67      }.start();
68    }
69
70    public void stopRequestReceiver() {
```

```
71      System.out.println("Closing socket in port 8088...");
72      try {
73        ss.close();
74      } catch (IOException e) {
75        System.out.println("ERROR: IOException while trying to close the
              socket!");
76        e.printStackTrace();
77      }
78      System.out.println("Port 8088 released!");
79    }
80  }
```

A.1.2 XML iPOJO Component Definition

```
1  <iPOJO xmlns:dc="org.apache.felix.ipojo.dynamic.handler">
2     <component classname="org.apache.comanche.
         requestReceiver.RequestReceiver">
3        <dc:dependency field="s"/>
4        <dc:dependency field="rh"/>
5        <callback transition="validate" method="start"/>
6        <callback transition="invalidate" method="stop"/>
7     </component>
8  </iPOJO>
```

A.2 Componente *SequentialScheduler*

A.2.1 Código

```
1  package org.apache.comanche.sequentialScheduler;
2
3  import org.apache.comanche.services.Scheduler;
4
5  public class SequentialScheduler implements Scheduler {
6
7    // ------------------------------------------------------------------
8    // Implementation of the Scheduler interface
9
10   public final void schedule(Runnable task) {
11     task.run();
12   }
13 }
```

A.2.2 XML iPOJO Component Definition

```
<iPOJO xmlns:dc="org.apache.felix.ipojo.dynamic.handler">
   <component
classname="org.apache.comanche.sequentialScheduler.
   SequentialScheduler">
      <provides />
      <dc:navigation />
   </component>
</iPOJO>
```

A.3 Componente *MultiThreadScheduler*

A.3.1 Código

```
package org.apache.comanche.multiThreadScheduler;

import org.apache.comanche.services.Scheduler;

public class MultiThreadScheduler implements Scheduler {

  // ------------------------------------------------------------
  // Implementation of the Scheduler interface

  public final void schedule(Runnable task) {
    new Thread(task).start();
  }
}
```

A.3.2 XML iPOJO Component Definition

```
<iPOJO xmlns:dc="org.apache.felix.ipojo.dynamic.handler">
   <component classname="org.apache.comanche.
      multiThreadScheduler.MultiThreadScheduler">
      <provides />
      <dc:navigation />
   </component>
</iPOJO>
```

A.4 Componente *RequestAnalyzer*

A.4.1 Código

```
1  package org.apache.comanche.requestAnalyzer;
2
3  import org.apache.comanche.services.RequestHandler;
4  import org.apache.comanche.services.Logger;
5  import org.apache.comanche.services.Request;
6  import org.apache.comanche.services.HandlerType;
7
8  import java.io.IOException;
9  import java.io.InputStreamReader;
10 import java.io.LineNumberReader;
11 import java.io.PrintStream;
12
13 public class RequestAnalyzer implements RequestHandler {
14
15   private static final String GET = "GET ";
16   private RequestHandler rh;
17   private Logger l;
18
19   // ------------------------------------------------------------------
20   // Implementation of the RequestHandler interface
21
22   public final void handleRequest(Request r) throws IOException {
23     r.in = new InputStreamReader(r.s.getInputStream());
24     r.out = new PrintStream(r.s.getOutputStream());
25     String rq = new LineNumberReader(r.in).readLine();
26     if (rq != null) {
27       l.log(rq);
28       if (rq.startsWith(GET)) {
29         r.url = rq.substring(GET.length() + 1, rq.indexOf(' ', GET.
               length()));
30         rh.handleRequest(r);
31       }
32     }
33     r.out.close();
34     r.s.close();
35   }
36
37   public final HandlerType getType() {
38     return HandlerType.DELEGATING;
39   }
40 }
```

A.4.2 XML iPOJO Component Definition

```
<ipojo xmlns:dc="org.apache.felix.ipojo.dynamic.handler">
   <component classname="org.apache.comanche.
      requestAnalyzer.RequestAnalyzer">
      <provides/>
      <dc:dependency field="l"/>
      <dc:dependency field="rh"/>
      <dc:navigation />
   </component>
</ipojo>
```

A.5 Componente *BasicLogger*

A.5.1 Código

```
package org.apache.comanche.basicLogger;

import org.apache.comanche.services.Logger;

public class BasicLogger implements Logger {
  // ------------------------------------------------------------
  // Implementation of the Logger interface

  public final void log(String msg) {
    System.err.println("Logger: " + msg);
  }
}
```

A.5.2 XML iPOJO Component Definition

```
<iPOJO xmlns:dc="org.apache.felix.ipojo.dynamic.handler">
   <component
classname="org.apache.comanche.basicLogger.BasicLogger">
      <provides />
      <dc:navigation />
   </component>
</iPOJO>
```

A.6 Componente *RequestDispatcher*

A.6.1 Código

```
1  package org.apache.comanche.requestDispatcher;
2
3  import java.io.IOException;
4  import java.util.ArrayList;
5  import java.util.List;
6
7  import org.apache.comanche.services.HandlerType;
8  import org.apache.comanche.services.RequestHandler;
9  import org.apache.comanche.services.Request;
10
11 public class RequestDispatcher implements RequestHandler {
12   private RequestHandler[] rh;
13   private List<RequestHandler> sortedRh;
14
15   public final void sort() {
16     // Sort connections : first files, then others, and errors...
17     List<RequestHandler> fileList = new ArrayList<RequestHandler>();
18     List<RequestHandler> errorList = new ArrayList<RequestHandler>();
19     List<RequestHandler> othersList = new ArrayList<RequestHandler>();
20     for (RequestHandler requestHandler : rh) {
21       HandlerType type = requestHandler.getType();
22       if (type.equals(HandlerType.FILE)) {
23         fileList.add(requestHandler);
24       } else if (type.equals(HandlerType.ERROR)) {
25         errorList.add(requestHandler);
26       } else if (!type.equals(HandlerType.DELEGATING)) {
27         othersList.add(requestHandler);
28       }
29     }
30     sortedRh = new ArrayList<RequestHandler>();
31     sortedRh.addAll(fileList);
32     sortedRh.addAll(othersList);
33     sortedRh.addAll(errorList);
34   }
35
36   // ---------------------------------------------------------------
37   // Implementation of the RequestHandler interface
38
39   public final void handleRequest(Request r) throws IOException {
40     // Ensure that the file handler is the first one.
41     // Then others and then error
42     sort();
43     for (RequestHandler requestHandler : sortedRh) {
```

```
44        try {
45          requestHandler.handleRequest(r);
46          return;
47        } catch (IOException exc) {
48        }
49      }
50    }
51
52    public final HandlerType getType() {
53      return HandlerType.DELEGATING;
54    }
55  }
```

A.6.2 XML iPOJO Component Definition

```
1 <ipojo xmlns:dc="org.apache.felix.ipojo.dynamic.handler">
2    <component
3 classname="org.apache.comanche.requestDispatcher.
   RequestDispatcher">
4        <provides/>
5        <dc:dependency field="rh"/>
6        <dc:navigation />
7    </component>
8 </ipojo>
```

A.7 Componente *ImageRequestHandler*

A.7.1 Código

```
1  package org.apache.comanche.requestHandlers;
2
3  import java.io.File;
4  import java.io.FileInputStream;
5  import java.io.IOException;
6  import java.io.InputStream;
7
8  import org.apache.comanche.services.RequestHandler;
9  import org.apache.comanche.services.Request;
10 import org.apache.comanche.services.HandlerType;
11
12 public class ImageRequestHandler implements RequestHandler {
13
```

```
14   // --------------------------------------------------------------------
15   // Implementation of the RequestHandler interface
16
17   public final void handleRequest(Request r) throws IOException {
18     File f = new File(r.url);
19     if (f.exists() && !f.isDirectory() && f.getName().endsWith(".jpg"))
         {
20       InputStream is = new FileInputStream(f);
21       byte[] data = null;
22       try {
23         data = new byte[is.available()];
24         int len = is.read(data);
25         if (len != data.length) {
26           throw new IOException("[ImageRequestHandler] File data not
               totally read");
27         }
28       } finally {
29         is.close();
30       }
31       r.out.print("HTTP/1.0 200 OK\n\n");
32       r.out.write(data);
33     } else {
34       throw new IOException("[ImageRequestHandler] File not found or is
           not an image");
35     }
36   }
37
38   public final HandlerType getType() {
39     return HandlerType.FILE;
40   }
41 }
```

A.7.2 XML iPOJO Component Definition

```
1 <ipojo xmlns:dc="org.apache.felix.ipojo.dynamic.handler">
2    <component
3 classname="org.apache.comanche.requestHandlers.
    ImageRequestHandler">
4       <provides/>
5       <dc:navigation />
6    </component>
7 </ipojo>
```

A.8 Componente *ErrorRequestHandler*

A.8.1 Código

```
1  package org.apache.comanche.requestHandlers;
2
3  import java.io.IOException;
4
5  import org.apache.comanche.services.RequestHandler;
6  import org.apache.comanche.services.Request;
7  import org.apache.comanche.services.HandlerType;
8
9  public class ErrorRequestHandler implements RequestHandler {
10
11   // ---------------------------------------------------------------
12   // Implementation of the RequestHandler interface
13
14   public final void handleRequest(Request r) throws IOException {
15     r.out.print("HTTP/1.0 404 Not Found\n\n");
16     r.out.print("<html>Document " + r.url + " not found.</html>");
17   }
18
19   public final HandlerType getType() {
20     return HandlerType.ERROR;
21   }
22 }
```

A.8.2 XML iPOJO Component Definition

```
1  <ipojo xmlns:dc="org.apache.felix.ipojo.dynamic.handler">
2     <component
3  classname="org.apache.comanche.requestHandlers.
     ErrorRequestHandler">
4        <provides/>
5        <dc:navigation />
6     </component>
7  </ipojo>
```

A.9 Componente *HTMLRequestHandler*

A.9.1 Código

```java
package org.apache.comanche.htmlRequestHandler;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

import org.apache.comanche.services.RequestHandler;
import org.apache.comanche.services.Request;
import org.apache.comanche.services.HandlerType;

public class HtmlRequestHandler implements RequestHandler {

  // ------------------------------------------------------------------
  // Implementation of the RequestHandler interface

  public final void handleRequest(Request r) throws IOException {
    File f = new File(r.url);
    if (f.exists() && !f.isDirectory() && (f.getName().endsWith(".html"
        ) || f.getName().endsWith(".htm"))) {
      InputStream is = new FileInputStream(f);
      byte[] data = null;
      try {
        data = new byte[is.available()];
        int len = is.read(data);
        if (len != data.length) {
          throw new IOException("[HtmlRequestHandler] File data not
              totally read");
        }
      } finally {
        is.close();
      }
      r.out.print("HTTP/1.0 200 OK\n\n");
      r.out.write(data);
    } else {
      throw new IOException("[HtmlRequestHandler] File not found or is
          not an html page");
    }
  }

  public final HandlerType getType() {
    return HandlerType.FILE;
  }
}
```

A.9.2 XML iPOJO Component Definition

```
<ipojo xmlns:dc="org.apache.felix.ipojo.dynamic.handler">
   <component
classname="org.apache.comanche.htmlRequestHandler.
   HtmlRequestHandler">
      <provides/>
      <dc:navigation />
   </component>
</ipojo>
```

A.10 Interfaces

A.10.1 Interface *RequestHandler*

```
package org.apache.comanche.services;

public interface RequestHandler
{
  void handleRequest(Request r) throws java.io.IOException;
  HandlerType getType();
}
```

A.10.2 Interface *Scheduler*

```
package org.apache.comanche.services;

public interface Scheduler
{
  void schedule(Runnable task);
}
```

A.10.3 Interface *Logger*

```
package org.apache.comanche.services;

public interface Logger {
  void log(String msg);
}
```

A.10.4 Interface *HandlerType*

```
1 package org.apache.comanche.services;
2
3 public enum HandlerType {
4   ERROR,
5   FILE,
6   DELEGATING,
7   OTHERS
8 }
```

A.10.5 Interface *Request*

```
1 package org.apache.comanche.services;
2
3 import java.io.PrintStream;
4 import java.io.Reader;
5 import java.net.Socket;
6
7 public class Request
8 {
9   public Socket s;
10  public Reader in;
11  public PrintStream out;
12  public String url;
13
14  public Request(Socket s)
15  {
16    this.s = s;
17  }
18 }
```

Apêndice B

Plano de Iniciação do Comanche

```
1 (!createCompositeType comancheType)
2 (!createInstance comanche comancheType)
3 (!createCompositeType frontendType)
4 (!createInstance frontend frontendType comanche)
5 (!createInstance requestReceiver org.apache.comanche.
    requestReceiver.RequestReceiver comanche:frontend)
6 (!createInstance sequentialScheduler org.apache.comanche.
    sequentialScheduler.SequentialScheduler comanche:frontend)
7 (!createCompositeType backendType)
8 (!createInstance backend backendType comanche)
9 (!createInstance requestAnalyzer org.apache.comanche.
    requestAnalyzer.RequestAnalyzer comanche:backend)
10 (!createInstance basicLogger org.apache.comanche.basicLogger.
    BasicLogger comanche:backend)
11 (!createCompositeType requestHandlerType)
12 (!createInstance requestHandler requestHandlerType comanche:backend
    )
13 (!createInstance requestDispatcher org.apache.comanche.
    requestDispatcher.RequestDispatcher comanche:backend:
    requestHandler)
14 (!createInstance imageRequestHandler org.apache.comanche.
    requestHandlers.ImageRequestHandler comanche:backend:
    requestHandler)
15 (!createInstance errorRequestHandler org.apache.comanche.
    requestHandlers.ErrorRequestHandler comanche:backend:
    requestHandler)
16 (!bind comanche:backend:requestHandler:requestDispatcher comanche:
    backend:requestHandler:imageRequestHandler org.apache.comanche.
    services.RequestHandler)
17 (!bind comanche:backend:requestHandler:requestDispatcher comanche:
    backend:requestHandler:errorRequestHandler org.apache.comanche.
    services.RequestHandler)
```

```
18 (!addExportedService comanche:backend:requestHandler
       requestDispatcher org.apache.comanche.services.RequestHandler)
19 (!bind comanche:backend:requestAnalyzer comanche:backend:
       requestHandler org.apache.comanche.services.RequestHandler)
20 (!bind comanche:backend:requestAnalyzer comanche:backend:
       basicLogger org.apache.comanche.services.Logger)
21 (!addExportedService comanche:backend requestAnalyzer org.apache.
       comanche.services.RequestHandler)
22 (!bind comanche:frontend:requestReceiver comanche:frontend:
       sequentialScheduler org.apache.comanche.services.Scheduler)
23 (!addImportedService comanche:frontend comanche:backend org.apache.
       comanche.services.RequestHandler)
24 (!bind comanche:frontend:requestReceiver comanche:backend org.
       apache.comanche.services.RequestHandler)
```