“Um Mecanismo de Monitoramento e Adaptação Dinâmica de Provedores de Serviços com Base em Atributos de Qualidade”

Por

Tarcisio Coutinho da Silva

Trabalho de Graduação

Universidade Federal de Pernambuco
Centro de Informática
secgrad@cin.ufpe.br
http://www.cin.ufpe.br/ secgrad

Recife, dezembro/2011

Universidade Federal de Pernambuco
Centro de Informática
Graduação em Ciência da Computação

Tarcisio Coutinho da Silva

"Um Mecanismo de Monitoramento e Adaptação Dinâmica de Provedores de Serviços com Base em Atributos de Qualidade"

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Nelson Souto Rosa*

Recife, dezembro/2011

Dedicado à minha Família,
especialmente a meus pais e minha irmãzinha, por todo o
incentivo e carinho oferecidos.

Agradecimentos

A Deus, por tudo que fez e tem feito em minha vida.

À minha Família, pelo apoio, incentivo e paciência ao longo dos anos, sempre me dando forças e coragem para que eu seguisse meus sonhos.

Ao professor Nelson Rosa, pela oportunidade de realizar este trabalho e por sua orientação sempre clara e objetiva.

A Fábio Souza, por sua ajuda, dedicação e sobretudo paciência, acompanhando todo o desenvolvimento deste trabalho sempre com boas idéias.

A todos os amigos que fiz durante esses quatro anos e meio de graduação, sem vocês as viradas de noite ou os finais de semana no CIn não seriam divertidos como foram.

A todos os amigos que fiz no NTI, obrigado por tornar meu primeiro estágio uma experiência interessante. Aprendi muito com todos.

E a todos que direta ou indiretamente contribuíram durante minha graduação, meus sinceros agradecimentos.

Oozora ni egaita
Chiisana yume wo taisetsu ni shiyou
Sousureba itsumo shiawase ni nareru
Kitto.

Dê importância ao pequeno sonho que você pintou em direção ao céu
Se fizer isto, você sempre será feliz
Certamente.

—YOKO ISHIDA (Shiawase no Iro)

Resumo

Arquitetura Orientada a Serviços, ou do inglês *Service-Oriented Architecture* (SOA), tem surgido ao longo dos últimos anos como uma das abordagens preferidas para construção de sistemas. Com objetivos alinhados aos objetivos do paradigma SOC *(Service-Oriented Computing)*, SOA possibilita a criação de novas aplicações com maior coerência, rapidez e diminuição nos custos, tudo isso com excelente aproveitamento do legado. Deste modo, a construção de aplicações orientadas a serviços tende a ter maior agilidade, flexibilidade e reuso de componentes pré-existentes. Neste contexto, módulos podem consumir serviços providos por tecnologias diferentes, apenas identificando o serviço requerido e realizando o *binding*, estabelecendo assim um contrato entre o consumidor e o provedor do serviço, esse definido através de SLAs *(Service Level Agreement)*. Porém, garantir que esses contratos são devidamente cumpridos é uma tarefa importante e complexa para provedores de serviços.

Tendo em vista esse cenário, o objetivo deste trabalho é especificar e construir um mecanismo de gerenciamento dinâmico de provedores de serviço, focado na prevenção de possíveis quebras de contrato em função do contexto em que o serviço provido é executado.

Palavras-chave: Arquitetura Orientada a Serviços, Service Level Agreement, Gerenciamento Dinâmico

Abstract

Service Oriented Architecture (SOA) has emerged as one of the preferred approaches for building systems. With Goals aligned to the goals of the SOC paradigm, SOA enables the creation of new applications with greater consistency, speed and decrease in costs, all with excellent use of the legacy. Therefore, the construction of service-oriented applications tend to have greater agility, flexibility and reuse of pre-existing components. In this context, modules can consume services provided by different technologies, only identifying the required service and performing the binding, thus establishing a contract between the consumer and the service provider, defined by SLAs (Service Level Agreement). However, ensure that these contracts are properly carried out is an important and complex task for service providers.

This work aims to specify and build a mechanism for dynamic management of service providers, focused on prevention of possible breaches of contract depending on the context in which the service provided is performed.

Keywords: Service Oriented Architecture, Service Level Agreement, Dynamic Management

Sumário

Lista de Figuras **xi**

Lista de Tabelas **xii**

Lista de Abreviaturas **xiii**

1 Introdução **1**
1.1 Contexto . . . 1
1.2 Objetivos . . . 4
1.3 Organização do Trabalho . . . 4

2 Conceitos Básicos **5**
2.1 Orientação a Serviços . . . 5
2.1.1 *Service Oriented Architecture* . . . 6
2.2 Orientação a Componentes . . . 6
2.2.1 *Component Elements* . . . 7
Modelo de Componente . . . 7
Instância de Componente . . . 7
Pacote de Componente . . . 7
2.2.2 *Deployment Descriptors* . . . 8
2.3 OSGi . . . 8
2.3.1 Arquitetura . . . 8
Module Layer . . . 9
Lifecycle Layer . . . 9
Service Layer . . . 9
2.3.2 iPOJO . . . 9
2.4 JMX . . . 10
2.4.1 Arquitetura . . . 11
Nível de Instrumentação . . . 12
Nível Agente . . . 13
Nível de Gerenciamento . . . 14
2.4.2 Mecanismo de Notificação . . . 14
2.5 Ciclo PDCA . . . 16
2.5.1 Etapas . . . 17

Plan - Planejar 17
Do - Executar 17
Check - Verificar 17
Act - Agir 17
2.6 Considerações Finais 18

3 DSOA **19**
3.1 Contexto e Objetivos 19
3.2 Arquitetura 20
3.2.1 Componentes 21
Distribution Service 21
Adaptation Manager 21
Broker Service 22
Mediator Service 22
Notification Service 22
Event Processing Service 22
Monitoring Service 23
Composition Service 23
Management Tool 23
3.3 Considerações Finais 24

4 Mecanismo de Gerenciamento Automático de Provedores de Serviços **25**
4.1 Visão Geral 25
4.2 Arquitetura 26
4.2.1 Módulo de Monitoração de Recursos 26
4.2.2 Módulo de Gerenciamento de Provedor 28
4.3 Aplicação do Ciclo PDCA 29
4.3.1 *Plan* - Planejar 29
4.3.2 *Do* - Executar 30
4.3.3 *Check* - Verificar 30
4.3.4 *Act* - Agir 30
4.4 Avaliação Experimental 31
4.4.1 Análise dos Resultados 32
4.5 Considerações Finais 35

5 Conclusão e Trabalhos Futuros **36**
5.1 Contribuições 36
5.2 Trabalhos Futuros 37

Referencias Bibliográficas **42**

Apêndices **43**

A *Provider Manager* **44**
A.1 XML Provider 44
A.1.1 Definição dos SLOs 45
A.1.2 Definição dos Profiles 45

Lista de Figuras

1.1 Triângulo SOA 2

2.1 Arquitetura OSGi 8
2.2 Handlers Ipojo 10
2.3 Arquitetura JMX 11
2.4 Acesso a atributos e operações do MBean. 13
2.5 Modelo de notificação JMX 16
2.6 Ciclo PDCA 16

3.1 Sistema de Monitoração 20
3.2 Arquitetura em Camadas DSOA 21

4.1 Visão Geral da Proposta na Arquitetura 26
4.2 Resource Monitoring Service 27
4.3 QoS Provider Manager 28
4.4 Aplicação do Ciclo PDCA 29
4.5 Componentes da Arquitetura x Fases PDCA 31
4.6 Consumo de Memória 50 Clientes 32
4.7 Consumo de Memória 100 Clientes 33
4.8 Consumo de Memória 200 Clientes 33
4.9 Carga CPU 200 Clientes 34

Lista de Tabelas

2.1 Componentes do Mecanismo de Notificação 15

4.1 Configuração do Ambiente de Experimentos 31
4.2 Média Tempo de Resposta dos Cenários (ms) 35

Lista de Abreviaturas

SOA Service Oriented Architecture

SOC Service Oriented Computing

DBC Desenvolvimento Baseado em Componentes

TI Tecnologia da Informação

UFPE Federal University of Pernambuco

NFR Non-Functional Requirement

SLA Service Level Agreement

WSDL Web Service Description Language

QoS Quality of Service

OSGi Open Services Gateway Initiative

JMX Java Management Extensions

JVM Java Virtual Machine

JCP Java Community Process

JSR Java Specification Request

API Application Programming Interface

SNMP Simple Network Manager Protocol

POJO Plain Old Java Objects

XML eXtensible Markup Language

1 Introdução

1.1 Contexto

Atualmente, o mercado cada vez mais acirrado e competitivo, faz com que as empresas constantemente modifiquem e redesenhem suas estratégias de negócio de maneira ágil, buscando inovação e agregação de valor a seus produtos e serviços.

Do ponto de vista computacional, esse tipo de cenário requer adaptação constante dos sistemas e processos relacionados, tornando os custos necessários à essa adaptação um grande obstáculo [1] [2], já que, a maioria desses custos são provenientes de implementações de novas soluções para cada caso específico. Além disso, o constante incremento de novas aplicações torna o sistema ainda mais complexo, dificultando o processo de desenvolvimento e aumentando os custos relacionados ao gerenciamento [2] [3].

Assim, torna-se clara a necessidade de uma solução que suporte esse novo contexto, no qual os sistemas constantemente evoluem, interagem e trocam informações entre si, onde a idéia de se desenvolver uma aplicação nova e complexa deve ser evitada e substituída pela composição de aplicações pequenas, já existentes (quando possível) [2]. A aplicação agora deve ser composta por um conjunto de módulos independentes, especializados, interoperáveis e sobretudo simples, o que aumenta a reutilização e facilita a manutenção do módulo [4].

Em função disto, temos nos princípios de SOA *Service Oriented Architecture*, um modelo arquitetural modular e interoperável que se mostra como uma alternativa viável à resolução dessa necessidade.

SOA tem surgido ao longo dos últimos anos como uma das abordagens preferidas para construção de sistemas distribuídos [5]. Com objetivos alinhados aos objetivos do paradigma SOC *(Service-Oriented Computing)* [6], SOA possibilita a criação de novas aplicações com maior coerência, rapidez e diminuição nos custos, tudo isso com excelente

aproveitamento do legado [5].

Baseada em padrões abertos e na visão onipresente da Internet, SOA tem como princípio fundamental a idéia de serviços como unidades que representam módulos do negócio ou funcionalidades da aplicação [5] [7] [8]. Na visão de SOA, um serviço é um componente que implementa uma função de negócio. Ele pode responder a requisições ocultando os detalhes de sua implementação e é descrito através de contratos que expressam seu objetivo e suas capacidades.

Buscando maior agilidade, flexibilidade e reuso de componentes pré-existentes no desenvolvimento de aplicações, diversas empresas de tecnologia da informação vêm adotando SOA na construção de seus sistemas. O modelo básico de SOA, consiste em 3 elementos principais apresentados na Figura 1.1.

O **provedor do serviço** (*Service Provider*) implementa o negocio e publica uma descrição do serviço no registro. O **registro** (*Service Registry*) dá suporte à capacidade de descobrimento de serviços (*Discoverability*), já que, provedores tem que publicar seus serviços para que estes possam ser consumidos por potenciais consumidores que realizam uma busca pelo serviço que necessitam no registro. E o **consumidor do serviço** (*Service Requestor*) que busca no registro por um determinado serviço, caso este seja encontrado, a localização do serviço é recuperada e o consumidor se conecta ao *endpoint* do serviço, podendo assim invocar as operações do serviço [9] [10].

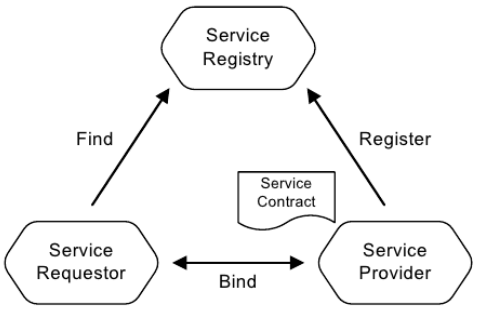


Figura 1.1: Elementos de SOA [11].

A Figura 1.1 mostra de maneira implícita alguns dos princípios essenciais de SOC, a flexibilidade e o baixo acoplamento entre serviços, uma vez que, um provedor de serviços pode muito bem fazer o papel consumidor de serviços. Essa dinâmica tem impactos ainda maiores quando tratamos de arquitetura de software, pois, ao trocarmos aplicações monolíticas por serviços ou composições de serviços, estes pequenos módulos podem ser facilmente substituídos ou atualizados, sem que haja um impacto maior no sistema.

Nesse contexto, temos a plataforma de serviços OSGi (*Open Services Gateway initiative*) [12]. OSGi provê suporte à modelagem e desenvolvimento de sistemas modulares na linguagem Java [13]. A idéia básica é resolver o problema de criar softwares monolíticos, ou seja, softwares projetados sem modularidade, facilitando a reutilização e manutenção de componentes, o que torna a solução mais robusta, barata e confiável [14].

OSGi introduz um modelo de programação orientada a serviço, o que alguns autores apresentam como *SOA in a Virtual Machine* [13], separando de fato a interface da implementação. Isso mostra um grande potencial para a construção de aplicações orientadas a serviço.

Outra tecnologia que compartilha vários princípios de SOC é a tecnologia de *Web Services*. *Web Services* fornecem uma forma padrão de interoperabilidade entre diferentes aplicações, capaz de executar em uma variedade de plataformas e *frameworks* [15] simplificando o desenvolvimento de aplicações distribuídas.

Assim, em uma arquitetura orientada a serviços, módulos podem consumir serviços providos por tecnologias diferentes, apenas identificando o serviço requerido e realizando o *binding*, que representa a realização de um contrato entre o consumidor e o provedor do serviço [4].

Esses contratos são especificados através de SLAs (*Service Level Agreements*). Em um SLA temos basicamente uma descrição de um acordo entre o provedor e consumidor de um determinado serviço. Esse acordo define: responsabilidades de ambas as partes; propriedades de como o consumidor terá acesso ao serviço (geralmente relacionadas a requisitos não-funcionais, como: *response time* e disponibilidade); duração do contrato, entre outros [16] [17].

Porém, uma tarefa complexa neste cenário é garantir que os acordos a nível de serviço entre o consumidor e provedor são realmente respeitados em tempo de execução. Além disso, prevenir uma quebra de contrato, onde o provedor não cumpre com os requisitos não-funcionais definidos nos SLAs, é outro ponto complexo a ser atacado, pois, na maioria dos casos, quando é caracterizada uma quebra de contrato o provedor pode sofrer algum tipo de penalidade, ou mesmo, perder um potencial consumidor de seus serviços.

Deste modo, o monitoramento dos serviços aliado à análise dos dados monitorados, surge como uma possível solução para o problema de garantir o cumprimento dos contratos entre provedores e consumidores de serviços [1]. Além disto, o resultado dessa análise aliada a um mecanismo de auto-gerenciamento do ambiente onde estes serviços são executados, surge como uma alternativa viável à manutenção dos contratos por parte dos provedores, uma vez que, o ambiente de execução do provedor tem impacto direto na

qualidade do serviço provida [18].

1.2 Objetivos

O objetivo desse trabalho é especificar e construir um mecanismo de gerenciamento automático de provedores de serviço, focado na prevenção de possíveis quebras de contrato.

A prevenção da quebra será realizada através do monitoramento de atributos de qualidade relacionados à invocações a um determinado serviço e do balanceamento de carga em função do contexto em que o serviço provido é executado.

Assim, as ações de gerenciamento são realizadas dinamicamente (em tempo de execução), sem que o serviço provido torne-se indisponível.

1.3 Organização do Trabalho

Visando atingir o objetivo proposto na Seção 1.2, o trabalho foi organizado da seguinte maneira:

- **Capítulo 2 - Conceitos Básicos**

  Neste capítulo serão apresentados os principais conceitos necessários ao entendimento do trabalho. Discutiremos sobre orientação a serviços, orientação a componentes, OSGi e iPOJO, o ciclo PCDA e gerenciamento com JMX. A apresentação destes conceitos é imprescindível para o entendimento da proposta.

- **Capítulo 3 - DSOA**

  No Capítulo 3 é apresentada a plataforma DSOA, na qual este trabalho está incluso. Abordaremos a motivação para seu desenvolvimento, seus objetivos e uma visão geral dos componentes presentes em sua arquitetura.

- **Capítulo 4 - Mecanismo Proposto**

  Este capítulo apresenta uma visão geral do mecanismo proposto, identificando onde cada componente desenvolvido se encaixa na plataforma DSOA. Discutiremos também a arquitetura, o funcionamento e detalhes importantes de sua implementação.

- **Capítulo 5 - Conclusão**

  O Capítulo 5 conclui o trabalho, apresentando as potencialidades e limitações da solução. E por fim, os possíveis trabalhos futuros abertos a desenvolvimento.

2
Conceitos Básicos

Neste capítulo discutiremos alguns conceitos básicos necessários ao entendimento da solução proposta. Abordaremos inicialmente orientação a serviços e componentes, apresentando as principais características de cada abordagem. Apresentaremos também a plataforma OSGi e o Framework iPOJO, que são as tecnologias que formam o núcleo deste trabalho. Em seguida, discutiremos sobre gerenciamento de recursos e serviços, sua importância no contexto de arquiteturas orientadas a serviços, e uma visão geral do ciclo de desenvolvimento PDCA, que é um acrônimo das etapas do ciclo (*Plan*, *Do*, *Check*, *Act*). Por fim, apresentaremos uma visão geral de JMX, a tecnologia utilizada no trabalho para realizar o monitoramento dos recursos.

2.1 Orientação a Serviços

Orientação a Serviços é um paradigma de desenvolvimento de aplicações distribuídas que utiliza serviços como unidades que representam funcionalidades da aplicação atendendo as necessidades do negócio [5] [8].

Desta forma, serviços descrevem suas capacidades através de interfaces bem definidas e utilizam-se de uma descrição de serviço (*Service Description*) para expor tais capacidades e eventuais propriedades específicas, por exemplo, propriedades relacionadas a requisitos de qualidade ou localização do serviço. Um exemplo conhecido de uma *service description* é o documento WSDL, utilizado para descrever *Web Services* expondo, além da interface do serviço, um conjunto extra de propriedades.

Deste modo, no mundo de serviços, temos diversas implementações distintas para um mesmo serviço, onde o consumidor pode não estar vinculado a nenhuma delas, podendo substituir o serviço consumido de acordo com suas necessidades, através da definição de novos contratos de serviço com base nessas descrições [8].

2.1.1 *Service Oriented Architecture*

Alinhado a essas idéias, temos em SOA, um modelo de arquitetura baseada em serviços com suporte à descoberta dinâmica de serviços. O modelo arquitetural de SOA baseia-se nos 3 elementos fundamentais da orientação a serviços.

- **Provedor de Serviços** (*Service Provider*)
- **Consumidor de Serviços** (*Service Requestor*)
- **Registro de Serviços** (*Service Registry*)

A interação destes elementos compõe o tradicional triângulo SOA, ver Figura 1.1. A interação dos elementos é relativamente simples. Provedores de serviço publicam suas capacidades através de *Service Descriptions* no registro de serviços. O consumidor do serviço executa consultas a um determinado serviço, com base em sua descrição, através do registro. Caso exista um provedor que atenda as necessidades do consumidor, então o registro devolverá uma referência do provedor do serviço ao consumidor, possibilitando a realização do *binding* entre eles. Então, o provedor disponibiliza um objeto que implementa a interface do serviço e representa o objeto remoto, em tempo de execução (*Servant* [19]), finalizando a interação com o provedor.

Quando a interação entre o consumidor e o *servant* é finalizada, o mesmo é liberado para utilização em alguma outra requisição ou simplesmente destruído. Nesse contexto, o consumidor não tem conhecimento de como o serviço é implementado, onde o mesmo se localiza, nem mesmo se o *servant*, pelo qual ele tem acesso ao serviço, é o mesmo a cada invocação. Esse grau de transparência provê um grande potencial de dinamismo e reuso, que são duas das principais características de arquiteturas orientadas a serviços [14].

2.2 Orientação a Componentes

Segundo Szyperski [20], "Componentes de software são unidades binárias de produção, aquisição e implantação independente, que interagem formando sistemas". Componentes possuem interfaces e dependências através de um modelo de componente. Esses modelos, assim como classes da orientação a objetos, descrevem as características do componente (ver 2.2.1).

A orientação a componentes também define o conceito de *containers*, que gerenciam o ciclo de vida e encapsulam os componentes, intermediando sua interação com o mundo externo.

Um componente possui 3 propriedades características:

1. **Pode ser implantado de forma independente**: Diz respeito a implantação independente de ambiente ou de outros componentes;
2. **Não deve possui estado observável**: Diz respeito as instâncias de componentes, que não devem ser distinguíveis entre si, ou seja, não há o conceito de estado observável presente na orientação a objetos. Terceiros não conseguem diferenciar cópias de um componente; e
3. **Unidade de composição com terceiros** Essa propriedade traz consigo algumas características semelhantes as da orientação a serviços, uma vez que, para a realização de uma composição, o componente deve ser coeso, auto-contido e possuir interfaces bem definidas. Um componente pode ser visto como uma caixa preta, que encapsula detalhes de sua implementação e interage com o mundo externo através de sua(s) interface(s).

Essas propriedades são realizadas por 3 elementos distintos que são base da orientação a componentes.

2.2.1 *Component Elements*

Modelo de Componente

O conceito de modelo de componente é, de certa forma, similar ao conceito de classe na orientação a objetos [8]. Ele abstrai e descreve as características dos componentes (interfaces, propriedades, dependências) e pode definir mecanismos pelos quais estes são implantados.

Instância de Componente

Uma instância de componente, retomando nossa analogia a orientação a objetos, é um elemento que representa uma entidade "física"do modelo. Diferente do modelo, uma instância de componente possui estado e pode fazer parte de composições [8] [20].

Pacote de Componente

Um pacote de componente pode ser definido como um componente pronto para ser implantado, ou seja, um componente auto-contido que possui todas as suas dependências resolvidas de maneira a prover suas funcionalidades de forma independente [8].

2.2.2 *Deployment Descriptors*

Outro elemento importante no mundo de componentes é o chamado *Deployment Descriptor*, que é um arquivo de configuração que descreve um conjunto de propriedades que definem como o componente será implantado na plataforma [21] [22].

Geralmente essas propriedades são descritas em documentos XML [23] devido a sua natureza semi-estruturada, representatividade e a facilidade de entendimento e escrita.

2.3 OSGi

Como é amplamente conhecido, Java fornece a portabilidade necessária para suportar aplicativos em diferentes plataformas, porém, não dá suporte explícito à construção de sistemas modulares [13]. A plataforma OSGi provê um conjunto de especificações permitindo que aplicações sejam construídas de maneira colaborativa, a partir de pequenos componentes reutilizáveis [24], simplificando o desenvolvimento e a manutenção do código [13].

O principal componente da especificação OSGi é o Framework OSGi, ele fornece um ambiente padronizado onde as aplicações, denominadas *bundles*, podem, em tempo de execução, ser instaladas, desinstaladas, ativadas ou desativadas local ou remotamente.

2.3.1 Arquitetura

A arquitetura do OSGi define um conjunto de camadas, vistas na Figura 2.2.

Iremos nos concentrar nas principais camadas do framework, apresentando-as nas próximas sub-seções.

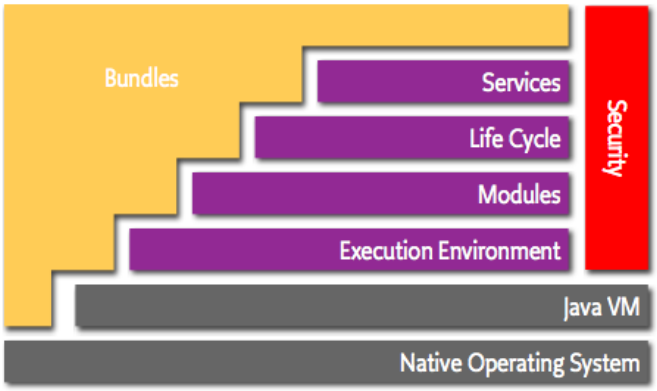


Figura 2.1: Arquitetura OSGi [24].

Module Layer

Define o conceitos de módulos, providos através dos *Bundles*. Um *bundle* é um arquivo JAR com um conjunto de metadados definidos em um *Manifest*. De fato, a figura do *bundle* é extremamente simples, porém, ao mesmo tempo poderosa, uma vez que, diferente de um simples arquivo JAR, um *bundle* define um módulo lógico que pode ser combinado para a composição de uma aplicação [13].

Além disso, *bundles* podem declarar explicitamente dependências externas e pacotes exportados, ampliando o mecanismo de controle de acesso padrão de Java. Essa capacidades de declarar explicitamente pacotes importados e exportados, traz um grande benefício ao desenvolvedor, uma vez que, o próprio framework fica responsável por gerenciar e verificar automaticamente a consistência da aplicação, através de um processo chamado *bundle resolution* [13] [24].

Lifecycle Layer

A camada de ciclo de vida controla o ciclo de vida dos *bundles*, definindo como os mesmos são gerenciados (instalados, desinstalados,parados, etc.) dinamicamente em tempo de execução.

Ela também define como os *bundles* obtém acesso ao *bundle context*, além de ser responsável por controlar a interação dos *bundles* com o framework.

Nesse contexto temos a figura do ativador do *bundle*, que se reposabiliza pela inicialização/parada o *bundle*. Provido através da interface *BundleActivator*, que realiza uma função semelhante ao *main* de um programa Java.

Service Layer

A camada de serviços incorpora características de SOA no OSGi, com as figuras do registro, provedor e consumidor de serviço, visto na Seção 2.1. Ela expande o dinamismo provido pelos *bundles* na camada de ciclo de vida, combinando-o ao dinamismo provido por arquiteturas orientadas a serviço, onde serviços podem aparecer ou desaparecer a qualquer momento [13].

2.3.2 iPOJO

Outra tecnologia importante para a compreensão deste trabalho, é o framework iPOJO [25]. O iPOJO implementa um modelo componentes orientados a serviço, e tem como principal objetivo simplificar o desenvolvimento de aplicações OSGi. O iPOJO utiliza o

conceito de POJO (*Plain Old Java Objects*) para definir componentes, onde um POJO é basicamente um objeto java simples, que não possui nenhuma dependência do ambiente de execução.

No iPOJO, os diversos POJO's são encapsulados em *containers*, responsáveis pelo gerenciamento das interações entre o POJO e o mundo externo ao *container* e podem ser estendidos através da utilização de *handlers* [25].

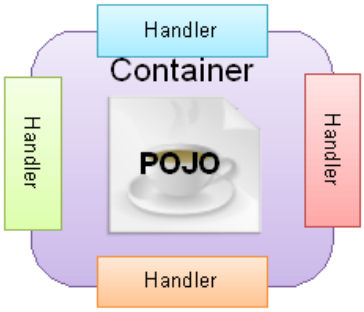


Figura 2.2: Handlers iPOJO [25].

O iPOJO se baseia nos conceitos de orientação a componentes, onde temos, além da figura do *container*, a visão de modelo e instância de componente. No iPOJO, o modelo de componente, chamado de *component type*, especifica: um conjunto de propriedades do componente, sua classe de implementação e políticas de criação de instâncias. Cada instância do componente herda todas as características do *component type* e pode ter seu próprio conjunto de propriedade [25].

O iPOJO abstrai o uso de OSGi através do conceito de *containers* que controlam o acesso e gerenciam o ciclo de vida do POJO, e de *handlers* "plugados" aos *containers* estendendo suas capacidades.

2.4 JMX

Java Management Extensions (JMX) é uma API que fornece uma maneira simples e padrão de gestão e monitoramento de recursos para a plataforma Java. Estes recursos podem ser aplicações, dispositivos, serviços ou a própria JVM (*Java Virtual Machine*) e são instrumentados por um ou mais *Managed Beans*, ou simplesmente MBeans, responsáveis por adquirir, manipular ou enviar informações acerca destes recursos [26].

A especificação de JMX define, além da arquitetura, padrões de projeto, API's e um conjunto de serviços de gerenciamento e monitoramento, que possibilitam o desenvolvimento de aplicações gerenciáveis local ou remotamente através do processo de

instrumentação, onde atributos, configurações e capacidades da aplicação são expostos. Isso aumenta a robustez e extensibilidade da aplicação, uma vez que, é possível construir soluções de gerenciamento inteligentes, interoperáveis e independentes da infra-estrutura de gestão [27].

2.4.1 Arquitetura

A arquitetura JMX é definida em três níveis, ver Figura 2.3:

- Nível de Instrumentação;
- Nível de Agente;
- Nível de Gerenciamento.

JMX foi definida segundo duas JSRs , JSR 3 e JSR 160. Os dois primeiros níveis da arquitetura (Instrumentação e Agente) foram definidos na JSR 3, enquanto o Nível de Gerenciamento foi definido na JSR 160. Isso mostra, de certa forma, o potencial de extensibilidade da tecnologia.

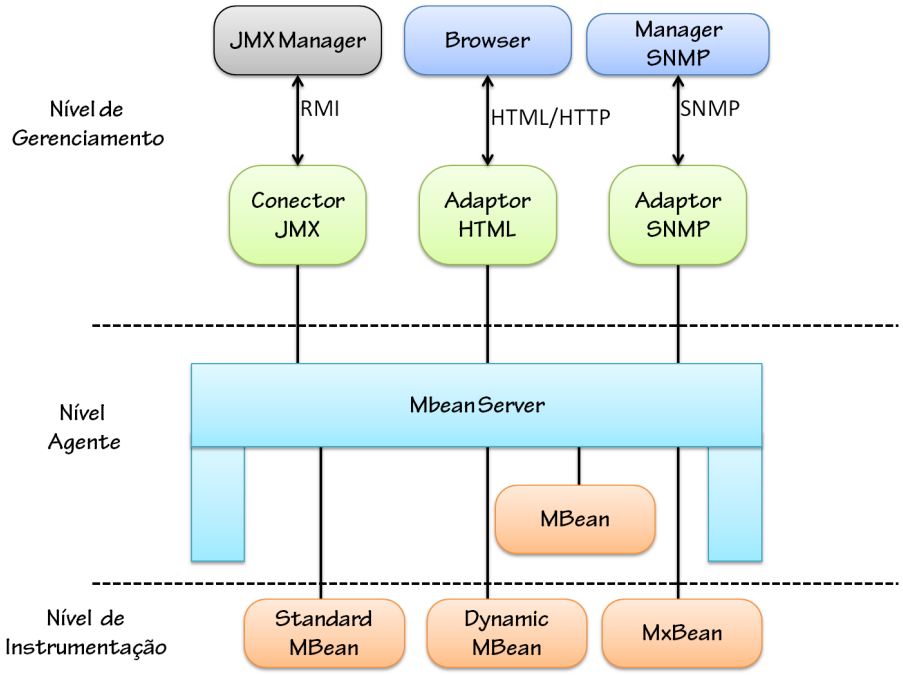


Figura 2.3: Arquitetura JMX.

Nível de Instrumentação

O nível de instrumentação é responsável por expor as funcionalidades e configurações das aplicações através da criação e registro de MBeans [27]. Estes MBeans coletam e manipulam informações dos recursos gerenciáveis, repassando-as aos agentes JMX do nível superior.

Existem diferentes tipos de MBeans em JMX. Nas próximas sub-seções descreveremos alguns dos mais importantes.

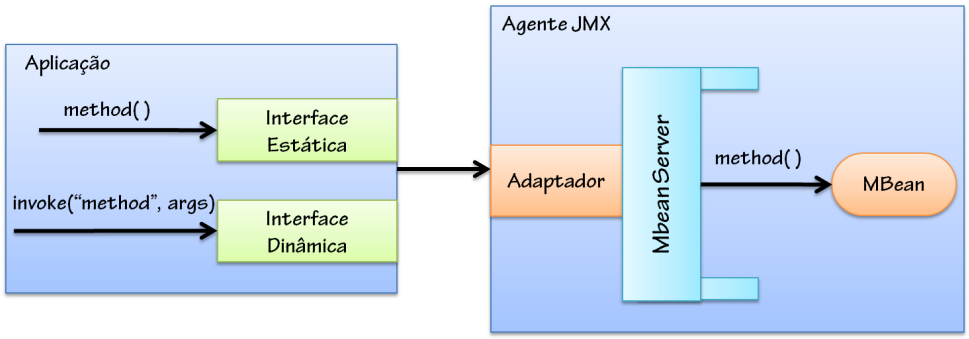
Standard MBean: São os tipos mais simples de *Managed Bean*. Eles interagem com os recursos gerenciáveis através da definição de uma interface de gerenciamento, que descreve os atributos e operações do MBean. Estas interfaces são definidas explicitamente e os atributos e métodos descobertos por meio de reflexão. Além disso, *Standard MBeans* seguem a convenção do nome da classe acrescido do prefixo `MBean` e todos os seus atributos são acessados por métodos *getters* e *setters*. *Standard Mbeans* possuem uma limitação quanto aos tipos de dados que podem ser utilizados, não permitindo o uso de tipos complexos definidos pelo usuário [28].

Uma limitação da arquitetura JMX é que não é permitido a implementação de duas interfaces de gerenciamento para o mesmo MBean, mesmo através de herança. Isto evita a implementação explícita de duas interfaces, pois o agente JMX sempre irá utilizar a interface de gerenciamento mais próxima da classe, ou seja, a interface implementada pela própria classe e não pela classe ancestral. Uma alternativa a esse problema é estender a interface de gerenciamento.

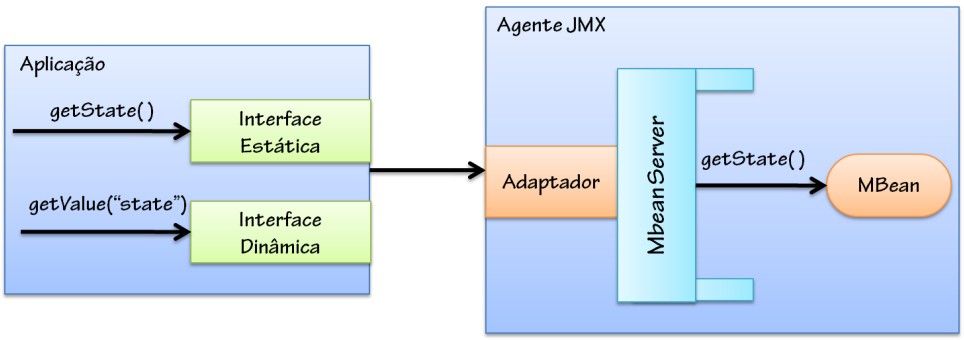
MxBean: Mxbean é uma versão melhorada do *Standard MBean* que possui uma solução ao problema relacionado ao conjunto limitads de tipos de dados utilizáveis pelo MBean comum. Ele provê suporte à utilização de tipos de dados complexos para representação de atributos do MBean, através de mapeamentos de tipos complexos para um tipo padrão `CompositeDataSupport` [28].

Dynamic MBean: *Dynamic MBeans* se diferenciam do *Standard MBeans* pelo fato de que aqueles descrevem seus atributos e operações através de uma interface genérica. Assim, as propriedades do MBean são descobertas em tempo de execução, já que, através dessa interface, os agentes JMX obtém a descrição do MBean por meio da classe `MBeanInfo` [27]. Na prática, o diferencial é que no caso dos *Standard MBeans*, os metadados que descrevem a interface são obtidos por meio de reflexão, e no *Dynamic*

MBean, as classes de metadados são construídos por elas mesmas em tempo de execução. Desde modo, o acesso ao atributos e operações em MBeans dinâmicas não é realizado diretamente, mas sim por meio de métodos genéricos, semelhante ao mecanismo utilizado pelo padrão de projeto *Requestor* [19]. Esse método de acesso é ilustrado na figura 2.4.



(a) Invocação de Operação no MBean



(b) Acesso a atributo no MBean

Figura 2.4: Acesso a atributos e operações do MBean.

Exitem outros tipos de MBeans dinâmicos, como *Model MBean*, *Open MBean* e *AbtractDynamicMBean* que basicamente adicionam novas *features* ao *Dynamic MBean* tradicional.

Nível Agente

O nível agente é responsável por gerenciar todos os *Mbeans* e através delas controlar diretamente todos os recursos, tornando-os acessíveis a aplicativos de gerenciamento remoto.

A gerência dos *MBeans* é provida através do principal componente do agente JMX, o *MBean Server*. Além disso, agentes JMX incluem um conjunto de serviços de gerenciamento de *MBeans* e pelo menos um adaptador de comunicações ou conector para permitir o acesso ao agente por aplicativos de gerenciamento [27]. Basicamente, um agente JMX

consiste em um *MBean Server*, um conjunto de serviços básicos de gerência e pelo menos um adaptador de comunicação ou conector JMX.

MBean Server: O *MBean Server* é o principal componente do nível agente. Ele é o intermediário entre o sistema de gerenciamento e os objetos gerenciáveis (recursos), uma vez que, é responsabilidade do *MBean Server* registrar e manipular os *MBeans* [27].

Para registrar um *MBean* no servidor, além do próprio objeto *MBean*, é necessário que seja registrado junto ao *MBean* um *ObjectName* que identifica unicamente aquele *MBean* no servidor. Além de identificar o *MBean*, ele permite a seleção de objetos a partir de máscaras no formato [domínio]:[par chave/valor].

```
org.meudominio:local=UFPE,nome=MBean
```

A manipulação de *MBeans* é feita através do *MBean Server* que provê um conjunto de operações comuns aos diferentes tipos de *MBeans*, uma vez que, os atributos e operações são descobertos através de classes de metadados comuns.

Nível de Gerenciamento

O nível de gerenciamento define um mecanismo baseado em adaptadores e conectores que tornam os agente JMX acessíveis a aplicativos de gerenciamento remoto fora da JVM em que o agente encontra-se [27]. De certa forma, o nível de gerenciamento define uma interface de acesso a agentes JMX para o mundo externo, através de protocolos proprietários ou existentes (*e.g.* SNMP - *Simple Network Management Protocol* [29], definindo um conjunto de conectores e adaptadores que disponibilizam o acesos aos agentes JMX para diferentes tecnologia de gerenciamento.

A principal diferença entre conectores e adaptadores é que adaptadores surgem como uma solução de integração a sistemas que não dão suporte direto a JMX, por exemplo, sistemas de gerenciamento baseados em SNMP ou HTTP como Zabbix [30] ou Nagios [31], fornecendo uma visão de todos os MBeans através de um determinado protocolo. Conectores, por sua vez, fornecem uma interface de gerenciamento remoto transparente e independente de protocolo [27].

2.4.2 Mecanismo de Notificação

Além dos três níveis de arquitetura, JMX provê um mecanismo de notificação que permite que aplicações ou *MBeans* recebam eventos ou notificações com informações sobre o

estado dos recursos gerenciados, podendo gerar estatísticas ou mesmo processar eventos relacionados ao funcionamento dos recursos [26].

O modelo de notificação JMX é semelhante ao mecanismo de notificações de Java, que define um conjunto de geradores de notificações, onde para receber uma notificação, o receptor deve registrar-se como *listener* do gerador. No JMX, o MBean que gera as notificações é representado pelo *(broadcaster MBean)*, responsável por enviar eventos de notificação a todos os *listeners* registrados [26].

O mecanismo de notificações JMX possui os seguintes componentes:

Componente	**Descrição**
Notification	Representa um tipo genérico de notificação
NotificationListener	Interface que permite o recebimento de notificações
NotificationFilter	Interface que permite a filtragem de notificações. Assim, apenas notificações relevantes ao Listener são recebidas
NotificationBroradcaster	Interface que permite que notificações sejam enviadas aos listeners registrados

Tabela 2.1: Componentes do Mecanismo de Notificação

O modelo de notificações JMX segue os seguintes passos, apresentados na Figura 2.5

Funcionamento

1. MBean consumidor registra-se através da interface *NotificationBroadcaster*
2. MBean gerador emite uma notificação
3. A notificação é enviada aos MBeans registrados
4. O MBean consumidor recebe e filtra as notificações
5. As notificações não descartadas são tratadas

Na prática, JMX é uma tecnologia de monitoramento e notificação flexível, dinâmica e extensível, sendo utilizada em diversas ferramentas comerciais (*e.g.* JBoss Application Server - JBoss [33], JOnAS Application Server - OW2 Consortium [34], Tivoli Composite Application Manager - IBM [35]).

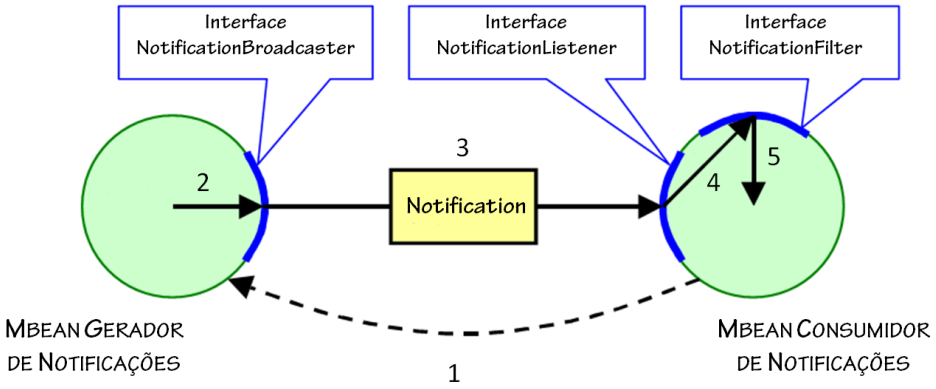


Figura 2.5: Modelo de notificação JMX [32].

2.5 Ciclo PDCA

O ciclo PDCA (***Plain Do Check Act***), ou ciclo de Demming, é um ciclo de desenvolvimento e gestão de processos com foco na melhoria contínua [36]. O ciclo é dividido em 4 etapas, ver Figura 2.6, que visam garantir o alcance de metas pré-estabelecidas, através da utilização de informações como fator chave na tomada de decisões.

O ciclo é aplicável a diversas atividades de gestão, incluindo o gerenciamento de performance [37]. Ele é um método eficaz de controle de atividades relacionadas à melhorias e compartilha dos conceitos de gerência de redes e recursos em geral, a medida que, a tomada de decisão é fortemente influenciada pelas informações coletadas.

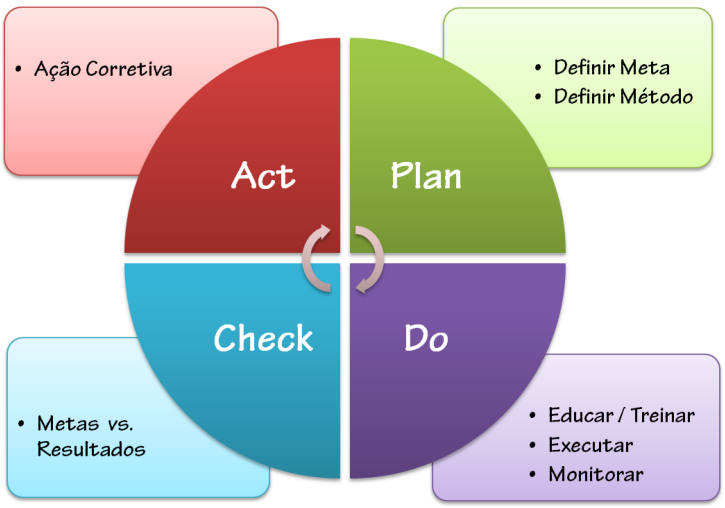


Figura 2.6: Ciclo PDCA de Desenvolvimento com Foco na Melhoria Contínua.

2.5.1 Etapas

Plan - Planejar

A fase de planejamento consiste no estabelecimento de metas e objetivos com base nas políticas adotadas pela organização [37]. Nesta etapa, é de extrema importância a análise de informações para a identificação e o entendimento do problema a ser resolvido. Esta é a fase mais importante do processo, uma vez que, um planejamento bem elaborado reflete no resultado como um todo. Evitando falhas e perda de tempo em fases futuras [38] [37].

A fase de planejamento, resulta na elaboração de um plano de ação, que define os métodos necessários para atingir as metas estabelecidas.

Do - Executar

É nesta fase em que ocorre a execução do plano de ação e coleta dos dados relacionados, para posterior análise [37].

Nesta etapa é importante que a execução seja coerente com o plano de ação definido na fase anterior, uma vez que, enquanto a fase anterior busca maior eficácia e coesão, a fase de execução é voltada à eficiência do processo.

Check - Verificar

Nesta etapa, é realizada a verificação dos resultados obtidos da execução realizada na fase anterior. Com base nas metas e estratégias definidas na fase de planejamento em conjunto com os dados monitorados na fase de execução, a etapa de verificação foca na análise sistemática dos dados monitorados a fim de verificar a efetividade das ações tomadas [37]. A diferença entre o desejável, o planejamento, e o resultado alcançado constitui um problema a ser resolvido.

Act - Agir

Nesta fase são realizadas as ações corretivas, com base na análise do resultados da fase anterior. Essas ações tem o intuito de evitar que o problema verificado na etapa anterior volte a acontecer [37]. Essas ações também estão definidas no plano de ação e além de buscar corrigir problemas, podem estar envolvidas na busca por melhoria contínua do processo [39].

A aplicação dessas etapas resulta no aprendizado do processo, o que repercute na tomada de decisão, uma vez que, utiliza de informações relevantes extraídas da execução do processo.

2.6 Considerações Finais

Este capítulo apresentou os conceitos essenciais ao entendimento dos próximos capítulos e do trabalho como um todo. Apresentamos os paradigmas de orientação a serviços e componentes, assim como as plataforma OSGi e iPOJO, que são bases tecnológicas do mecanismo proposto. Por fim, demos uma visão geral do ciclo de desenvolvimento PDCA e da tecnologia de monitoramento de recursos utilizada neste trabalho, a tecnologia JMX.

3
DSOA

Neste capítulo apresentaremos a plataforma DSOA (*Dynamic SOA*), dando uma visão geral da motivação e dos objetivos da plataforma. Apresentaremos também uma rápida descrição dos principais componentes que constituem a arquitetura da plataforma, contextualizando a realização deste trabalho.

3.1 Contexto e Objetivos

A natureza dinâmica e distribuída do ambiente SOA e o não-determinismo dos atributos de qualidade sugerem a necessidade de um sistema de monitoração contínua.

Nesse contexto, a plataforma DSOA estende as capacidades fornecidas pelas arquiteturas orientadas a componentes baseados em serviços atuais, com a capacidade de adaptação dos componentes em função dos atributos de qualidade.

Esse tipo de arquitetura é na verdade um modelo de composição de serviços, onde temos as figuras do consumidor e do provedor de serviços, nitidamente separadas.

Neste cenário, a adaptação faz sentido em ambos os lados. No lado do consumidor de serviços, a adaptação consiste na capacidade de selecionar serviços de funcionalidade equivalente, com ciência da qualidade provida. No lado do provedor, essa adaptação faz sentido a medida que existe a necessidade de garantir aspectos de qualidade anunciados.

Assim, o objetivo de DSOA é propor uma plataforma orientada a componentes baseados em serviços que estende a capacidade do *container*, de forma a tratar com aspectos de QoS, em particular, aspectos relacionados a desempenho.

Neste sentido, tratar aspectos de qualidade relacionados a desempenho é um objetivo desafiador, pois, a maioria das plataformas existentes, não dão suporte a isso. O mais próximo disso, podem ser vistos no iPOJO [25], *Declarative Services* [40] e BluePrints [41]. Porém, tais plataformas tratam apenas de aspectos relacionados a questão da

disponibilidade do serviços, não considerando aspectos de desempenho.

3.2 Arquitetura

Como vimos na Seção 3.1, no contexto da adaptação dinâmica, a consideração de atributos de qualidade ligados a desempenho traz a necessidade de monitoramento constante de tais atributos. Para isso, torna-se necessária a figura de um sistema de monitoração *online*. Basicamente, um sistema de monitoração é composto por uma linguagem de especificação (para definir configurações de monitoração), um conjunto de sensores (para coletar dados), analisadores e processadores de eventos (para realização de ações com base na análise dos dados monitorados).

Uma representação desse tipo sistema pode ser vista na Figura 3.1.

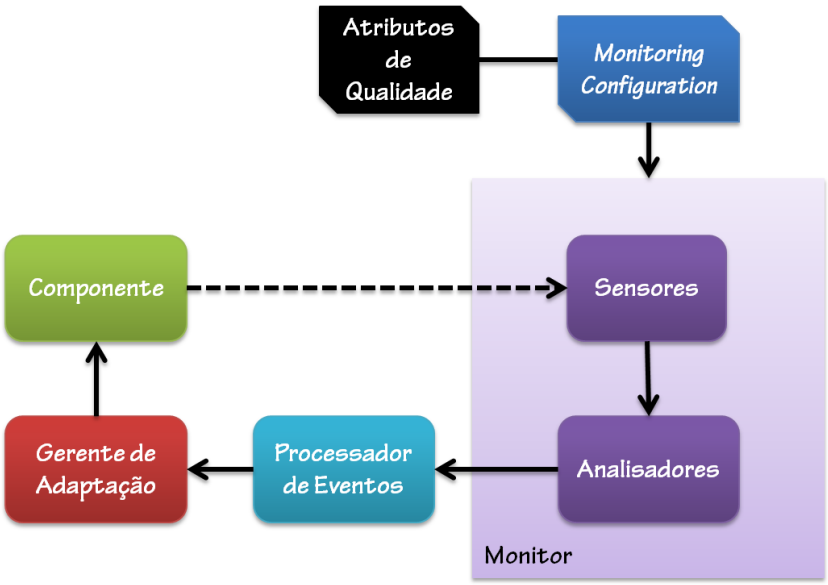


Figura 3.1: Sistema de Monitoração

Porém, para prover adaptação dinâmica, o *container* deve ser capaz de modificar-se sem provocar a interrupção do serviço. Deste modo, a plataforma especifica a inclusão de uma camada de serviços de utilidade sob a camada de composição de serviços e uma camada de adaptação sobre a mesma, visando estender suas capacidades, conforme a Figura 3.2.

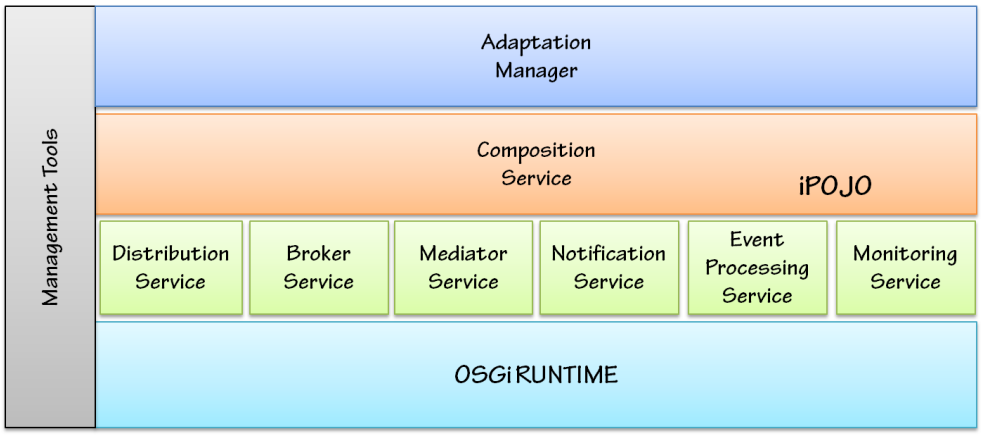


Figura 3.2: Arquitetura em Camadas DSOA.

3.2.1 Componentes

Nesta seção, daremos uma visão geral de cada componente presente na arquitetura, identificando sua funcionalidade e suas características principais.

Distribution Service

O serviço de distribuição é responsável por permitir que serviços disponibilizados no registro do *runtime* OSGi local sejam distribuídos como serviços remotos a outros *runtimes*. Esse serviço é provido através da utilização do DOSGi [42], que implementa um conjunto de serviços que provêem distribuição de serviços OSGi. Seu funcionamento baseia-se na idéia de um registro centralizado de serviços remotos, descritos como *endpoints*, onde esses serviços são disponibilizados localmente através de *proxies* que permitem o acesso ao *endpoint*.

Adaptation Manager

É o principal componente da arquitetura e é responsável por gerenciar dinamicamente as dependências das aplicações, com base em atributos de qualidade.

No ambiente DSOA, uma aplicação pode ser vista como uma composição de serviços sendo especificada de forma declarativa, ou seja, cada composição possui um descritor, no qual são indicadas as interfaces dos serviços necessários e os atributos de qualidade esperados. Nesse contexto, cabe ao gerente de adaptação interpretar este descritor e buscar garantir que os serviços selecionados para a composição atendam aos critérios de qualidade especificados. Caso isso não ocorra, cabe a ele, adaptar dinamicamente a

aplicação substituindo o serviço em uso, por um mais adequado.

Outra responsabilidade do *Adaptation Manager* é solicitar ao serviço de monitoração a verificação da qualidade do serviço provida pelos elementos da composição, através da definição de configurações de monitoração (*monitoring configurations*).

Broker Service

O *broker service* provê um mecanismo de seleção de serviços com base em atributos de qualidade. Ele abstrai a busca ao registro para o gerente de adaptação e através de um conjunto de serviços de seleção especializados em atributos de qualidade, retorna os serviços que melhor atendem as necessidades do consumidor.

As necessidades do cliente são definidas em tempo de *deployment* através de propriedades do componente. Porém, todo o processo de seleção e adaptação é realizado dinamicamente em tempo de execução.

Mediator Service

Fornece um serviço de integração e mediação de serviços distribuídos, com funcionalidades equivalentes em diversas tecnologias, a partir de uma interface única, que media a invocação a estes serviços. O esquema de mediação é definido através de uma interface gráfica que realiza o cadastro das interfaces do serviço concreto e das interfaces de integração de serviços.

Notification Service

O serviço de notificação é responsável por sinalizar as quebras de contrato ao gerente de adaptação, permitindo que este tome as medidas necessárias para garantir os aspectos de qualidade esperados. As notificações são realizadas através do envio de eventos assíncronos, baseado no padrão de troca de mensagens *publish-subscribe*, definindo tópicos para a publicação e entrega de notificações por meio de eventos.

Event Processing Service

O serviço de processamento de eventos é o módulo responsável pelo processamento dos eventos que ocorrem na plataforma. Em particular, a própria infra-estrutura gera eventos relacionados ao ciclo de vida dos serviços, representando, por exemplo, o envio de requisições e respostas, a publicação e a remoção de serviços, além de eventos derivados que indicam o nível de qualidade do serviço percebido pela aplicação. Ele também é

responsável pela verificação de quebras de contrato, através da análise dos dados de qualidade recebidos.

Em termos de complexidade computacional, esta é uma tarefa extremamente custosa, sobretudo pela quantidade de eventos que devem ser processados. Deste modo, o serviço é implementado sob uma *engine* de processamento de eventos complexos, *Esper* [43], que fornece um mecanismo de processamento de *stream* de eventos e uma linguagem de consulta própria (`EQL`), facilitando o processamento configurável dos eventos em tempo real, através da criação de *statements* que selecionam ou correlacionam dados provenientes do processamento dos eventos.

Monitoring Service

O serviço de monitoração é responsável por inicializar a monitoração dos serviços utilizados pela aplicação. Ele recebe as configurações de monitoração, extraídas através das especificações do serviço pelo *adaptation manager*, e configura o *Event Processing Service* de maneira a processar os eventos gerados pela monitoração dos atributos de qualidade da aplicação.

É função do serviço de monitoração, definir os *statements* utilizados no processamento de eventos. Estes *statements* são construídos com base nas *monitoring configurations* repassadas pelo *adaptation manager*.

Composition Service

DSOA utiliza o iPOJO [25] como infra-estrutura à composição de serviços. Ele implementa um *container* que realiza a injeção de dependências na aplicação, ou seja, indica quais serviços devem ser utilizados na composição, sob uma pespectiva puramente funcional. Assim, temos na figura do *adaptation manager* e dos demais serviços, uma extensão da infra-estrutura de suporte à composição, permitindo o uso de atributos de qualidade na seleção dos serviços que deverão compor a aplicação.

Management Tool

Este componente fornece um conjunto de ferramentas administrativas para gerenciar a plataforma e os serviços que estão em execução, integradas a VisualVM [44], uma ferramenta visual para gerenciamento de aplicações na plataforma Java SE.

3.3 Considerações Finais

Este capítulo apresentou a plataforma DSOA, que um conjunto de componentes na forma de serviços, para lidar com atributos de qualidade visando a adaptação dinâmica da aplicação. Assim, os conceitos apresentados neste capítulo são essenciais para a contextualização deste trabalho, uma vez que, este tem como um dos objetivos, estender a plataforma conforme veremos no próximo capítulo.

4

Mecanismo de Gerenciamento Automático de Provedores de Serviços

Neste capítulo, apresentaremos os componentes desenvolvidos para a realização dos objetivos propostos, bem como, detalhes de implementação considerados importantes para o melhor entendimento técnico do mecanismo. Também será mostrada uma análise do processo de gerenciamento automático, tendo em vista as idéias do ciclo PDCA, apresentado na Seção 2.5 deste trabalho. Essa análise irá relacionar as etapas definidas no ciclo com os sub-processos definidos pelo mecanismo.

4.1 Visão Geral

O contexto ao qual este trabalho está inserido foi apresentado no Capítulo 3, através da plataforma DSOA, que provê um ambiente de composição dinâmica baseado em componentes orientados a serviços, com percepção de requisitos não-funcionais mapeados em atributos de qualidade.

Porém, o estado atual da plataforma não dá suporte ao gerenciamento e monitoramento do provedor de serviços. O provedor basicamente define um conjunto de atributos de qualidade, que são utilizados pelos consumidores na seleção de serviço e na elaboração de contratos. Solucionar tais limitações mostra-se relevante, a medida que a qualidade do serviço é um fator chave tanto seleção do provedor quanto na manutenção dos contratos.

Buscando resolver esse problema, o objetivo deste trabalho é especificar e construir um mecanismo de gerenciamento automático de provedores de serviço, focado na prevenção de possíveis quebras de contrato. Para isso, a plataforma deve ser capaz de monitorar e adaptar dinamicamente o provedor de serviços em função do contexto em que o mesmo executa.

4.2 Arquitetura

Como vimos na Seção anterior, a plataforma DSOA, não prevê o gerenciamento do provedor de serviços com base em atributos de qualidade, deste modo, propomos uma estender a plataforma para suprir tais necessidades. Para isso, foram desenvolvidos e implantados 2 módulos à plataforma. Um módulo de monitoração de recursos de hardware e um módulo de gerenciamento de provedores de serviços (ver Figura 4.1).

O módulo de monitoração de recursos consiste em um sensor que captura informações referentes ao estado dos recursos da máquina. É instrumentado através de JMX e foi disponibilizado como um serviço da plataforma. O módulo de gerenciamento de provedor de serviço estende um *container* fornecendo suporte à reconfiguração dinâmica de serviços. A reconfiguração é baseada em atributos de qualidade e no contexto em que o provedor executa.

A figura 4.1 destaca onde o trabalho desenvolvido está situado na plataforma DSOA.

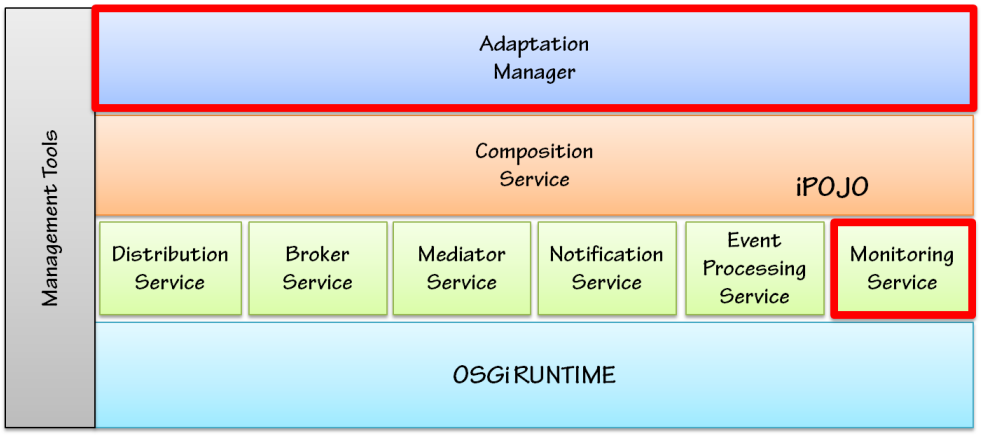


Figura 4.1: Visão Geral da Proposta na Arquitetura.

A combinação destes 2 módulos, junto aos serviços providos pela plataforma, surge como uma solução para o problema de gestão de provedores de serviço. Detalharemos seu funcionamento e arquitetura nas próximas seções.

4.2.1 Módulo de Monitoração de Recursos

Conforme o Capítulo 3, um elemento importante de um sistema de monitoração corresponde a figura dos sensores. Sensores são responsáveis por coletar dados referentes aos recursos monitorados e, na plataforma DSOA, estão representados pelos serviços de monitoração.

Originalmente a plataforma suporta um conjunto de sensores responsáveis pela coleta de dados referente à qualidade do serviço. Contudo, do ponto de vista do provedor, dois fatores impactam diretamente na qualidade: a demanda e a limitação dos recursos de hardware e software [18].

Neste contexto, é vital que os recursos sejam monitorados, de forma a evitar que o consumo elevado degrade a qualidade do serviço provido, prevenindo eventuais quebras de contratos. Essa necessidade nos levou a proposição de um novo conjunto de sensores com a finalidade coletar dados relacionados ao contexto de execução do serviço, em termos de consumo de recursos de hardware, em particular, consumo de CPU e memória.

Esse conjunto de sensores, foi incorporado à plataforma DSOA, através do serviço de monitoração de recursos (*Resource Monitoring Service*), que é parte integrante do serviço de monitoração (*Monitoring Service*), conforme a Figura 4.2.

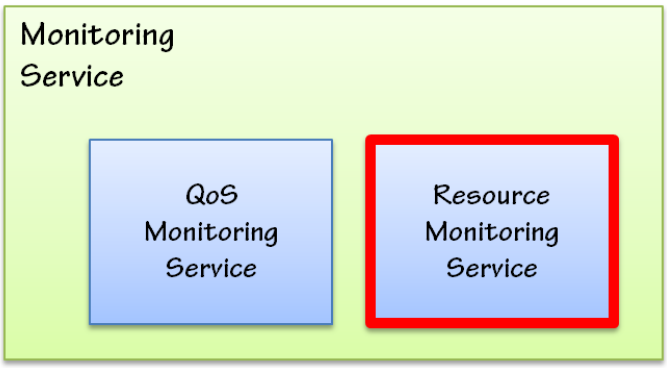


Figura 4.2: Resource Monitoring Service.

Como foi dito na Seção 4.2, o serviço de monitoração de recursos foi desenvolvido utilizando JMX para realizar a coleta de dados acerca dos recursos da máquina. Esse processo é realizado através de uma técnica de *polling* [45] sobre os dados de memória e utilização de CPU.

Os dados do monitoramento são enviados, através de eventos, ao serviço de processamento de eventos (*Event Processing Service*), que gera notificações ao gerenciador de adaptação (*Adaptation Manager*) quando o provedor encontra-se em um "Estado de Alerta".

Um estado de alerta pode ser visto como uma iminência de quebra de contrato. Neste estado, o provedor chegou a um ponto crítico de utilização dos recursos e logo não poderá garantir a qualidade devida, necessitando adaptar-se ao novo contexto de execução.

4.2.2 Módulo de Gerenciamento de Provedor

Em linhas gerais, um processo de garantia de qualidade envolve um ciclo de atividades compreendendo além da coleta e análise dados, uma intervenção que representa uma ação ou conjunto de ações realizadas com base nessa análise.

Na plataforma DSOA, as etapas de coleta e análise são tratadas respectivamente pelo serviço de monitoração e pelo serviço de processamento de eventos. A ação é de responsabilidade do gerente de adaptação, que no contexto do provedor de serviços, atualmente não realiza nenhum tipo de tratamento.

Deste modo, verificou-se a necessidade de estender o gerente de adaptação para atuar também no contexto do provedor, uma vez que, o monitoramento de recursos é restrito apenas à identificação de estados de alerta.

Assim, o módulo de gerenciamento do provedor (*QoS Provider Manager*) implementa uma extensão do *container* com o objetivo de permitir que este se adapte dinamicamente, em função do estado atual de consumo de recursos e da qualidade do serviço provido. Essa extensão é apresentada na Figura 4.3.

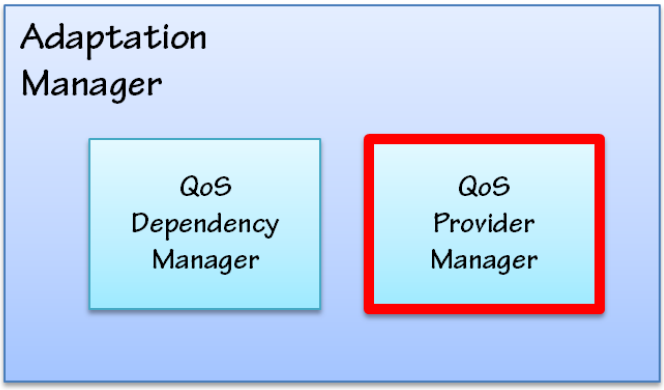


Figura 4.3: QoS Provider Manager.

O módulo de gerenciamento do provedor é capaz de adaptá-lo a seu contexto de execução, visando manter a qualidade de serviço, através do gerenciamento das políticas de criação de instâncias, que definem como os objetos remotos são construídos e acessados quando invocados. A partir disto, definimos um conjunto de políticas para cenários específicos de uso dos recursos. Porém, devido a diversidade enorme de cenários existentes, uma vez que, o uso de recursos é extremamente variável de aplicação para aplicação, a melhor maneira de contemplar a maioria dos casos é disponibilizar a política de criação de instâncias como um serviço.

4.3 Aplicação do Ciclo PDCA

Todo o processo de monitoramento e adaptação será baseado nas etapas definidas pelo ciclo PDCA. Como vimos na Seção 2.5, o ciclo foca em manter ou melhorar a qualidade de processos, com base nas informações referentes à sua execução.

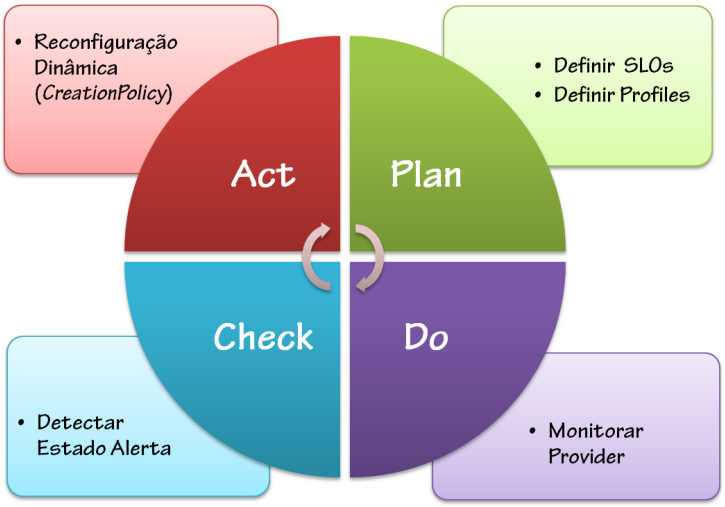


Figura 4.4: Aplicação do Ciclo PDCA.

4.3.1 *Plan* - Planejar

Na etapa de planejamento, as metas serão definidas por meio de um documento XML [23] que descreve as capacidades que o provedor afirma garantir. Essas propriedades são utilizadas para definir configurações de monitoração (*Monitoring Configuration*), relacionadas a atributos de qualidade do provedor (*e.g* tempo de processamento).

Essas metas são definidas através da *tag* `slo`, que descreve uma métrica a ser medida, um valor de *threshold* e uma relação de limite(máximo ou mínimo) que o valor do *threshold* pode ultrapassar. Em alguns casos, temos também o identificador da operação provida. Essa informação é útil em casos de configurações de monitoração relacionadas à invocações de determinados métodos do provedor (e.g. *throuhput* de uma operação).

Além disso, também são definidos no mesmo XML, métodos que visam atuar sobre estados de alerta, buscando garantir que as capacidades apresentadas sejam devidamente providas. Esses métodos são descritos pela *tag* `profile`, que define um conjunto de `profiles` responsáveis por relacionar as configurações de monitoração às políticas de gerenciamento de instâncias, onde cada política de gerenciamento tem por objetivo evitar

a degradação da qualidade provida.

Um `profile` especifica um conjunto de `resources`, que assim como os `slos`, definem uma configuração de monitoração (*monitoring configuration*).

4.3.2 *Do* - Executar

Definidas as metas de qualidade e as políticas de manutenção de recursos (através de `slos` e `profiles`), o próximo passo é monitorar o provedor em execução. Cabe ao *Monitoring Service* realizar esta tarefa.

A monitoração de recursos é de responsabilidade do *Resource Monitoring Service*, que coleta dados referentes ao estado atual da máquina (através de um processo de *polling*) e os envia à central de processamento de eventos.

A monitoração de atributos de qualidade é realizada no próprio gerenciador de adaptação, através da utilização da interceptação das invocações ao serviço.

Um conjunto de técnicas e padrões para captura de atributos relacionados a desempenho são discutidos em [46] [47]. Deste modo, foi adotado o padrão *QoS Wrapper* [46] para monitoramento do provedor, uma vez que, este é extremamente simples, não gera um *overhead* elevado e é o que melhor se aplica ao nosso problema.

4.3.3 *Check* - Verificar

A etapa de verificação difere um pouco do processo original, pois ela também ficou responsável por agregar os dados da monitoração uma vez que os dados da monitoração são enviados à cada invocação ao serviço (no caso de dados relacionados à qualidade) ou em intervalos de tempo (no caso de dados relacionados ao estado da máquina).

Além disso, é nesta etapa que os estados de alerta são identificados a partir da análise dos dados monitorados em relação aos `slos` e `profiles`. Essa análise é realizada pelo serviço de processamento de eventos que envia notificações ao gerenciador de adaptação através do serviço de notificação.

4.3.4 *Act* - Agir

A última fase do ciclo é responsável por garantir o dinamismo da solução através do módulo de gerenciamento do provedor (*QoS Provider Manager*). O *QoS Provider Manager* recebe notificações referentes ao nível de qualidade do serviço provido e ao estado dos recursos da máquina. Com base nessas informações, reconfigura o serviço visando evitar eventuais quebras de contrato.

A reconfiguração é realizada a partir da modificação da política de criação de instâncias, como foi visto na Seção 4.2.2.

A figura 4.5 mostra onde cada etapa do ciclo está situada na plataforma DSOA.

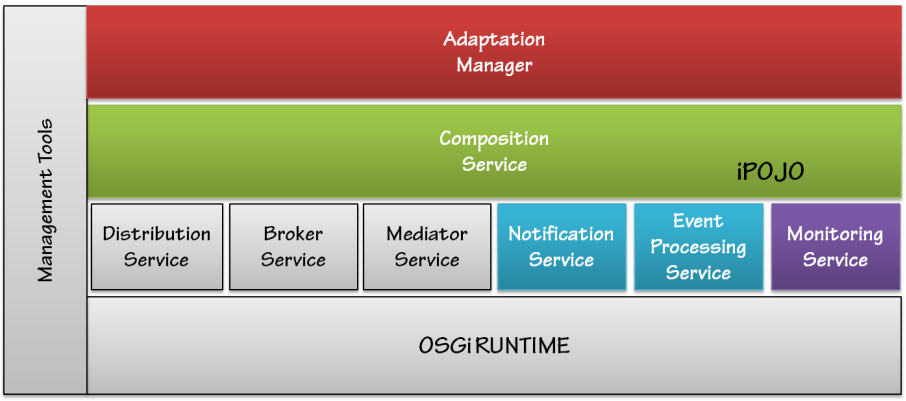


Figura 4.5: Componentes da Arquitetura x Fases PDCA.

4.4 Avaliação Experimental

Para avaliar o mecanismo proposto, utilizamos a ferramenta JMeter [48] para simulação de carga, através de requisições. O ambiente de experimentos contou com o maquinário descrito na Tabela 4.1.

Máquina	**S.O.**	**Memória RAM**	**Processador**
Máquina 1	Windows 7 Professional	2G	Intel Core 2 Duo 2.8 Ghz
Máquina 2	Windows 7 Professional	8G	Intel Core i7 2.8 Ghz
Máquina 3	Slackware Linux 13.37	2G	Intel Dual Core 2.0 Ghz

Tabela 4.1: Configuração do Ambiente de Experimentos

O provedor de serviços executa na `máquina 3`, enquanto os consumidores (gerados pelo JMeter) executam nas `máquinas 1` e `2`.

Inicialmente iremos executar com 3 configurações distintas para avaliar o impacto do número de clientes (consequentemente o número de requisições) no consumo de recursos.

1. **50 clientes - 50 requisições**
2. **100 clientes - 50 requisições**
3. **200 clientes - 50 requisições**

4.4.1 Análise dos Resultados

Devido às limitações de tempo e a percepção ainda não muito clara das melhores configurações de `Profiles` e `Resources` , realizamos uma comparação entre duas políticas: *default* e a adaptativa (que varia entre as duas *default* em função do contexto).

Inicialmente utilizaremos a política de criação *singleton*, que retorna sempre a mesma instância, analisando o impacto no consumo de recursos de memória e CPU, além do tempo de resposta percebido pelo cliente. Em seguida, realizaremos o mesmo teste com a política de criação *per-request*, que devolve uma nova instância a cada requisição. Por fim, utilizaremos estas duas políticas sob a gerência do *QoS Provider Manager*.

A utilização da política *singleton* deverá apresentar um menor consumo de memória, uma vez que, temos apenas uma instância do objeto remoto para todas as requisições ao serviço. Porém, caso a invocação de algum método do objeto remotos tenha de ser sincronizado, esta política terá uma carga de CPU elevada devido ao processo de sincronização das diversas *threads* concorrentes.

A política *per-request* não apresenta o mesmo problema de carga de CPU quando existe sincronização de invocação ao método do objeto remoto, uma vez que, cada requisição (pertencente a mesma *thread* ou não) é realizada a um objeto diferente. Porém, o consumo de memória apresentado é diretamente proporcional a demanda ao serviço.

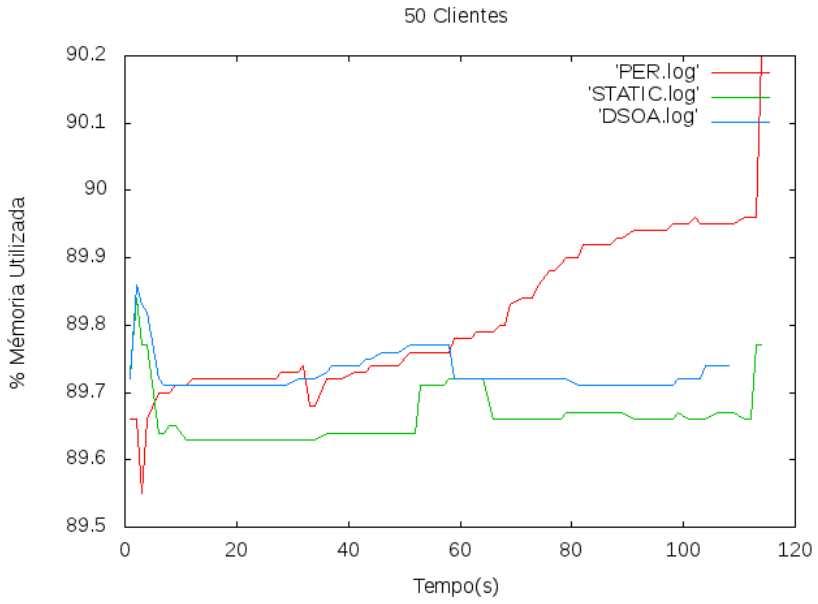


Figura 4.6: Consumo de Memória 50 Clientes.

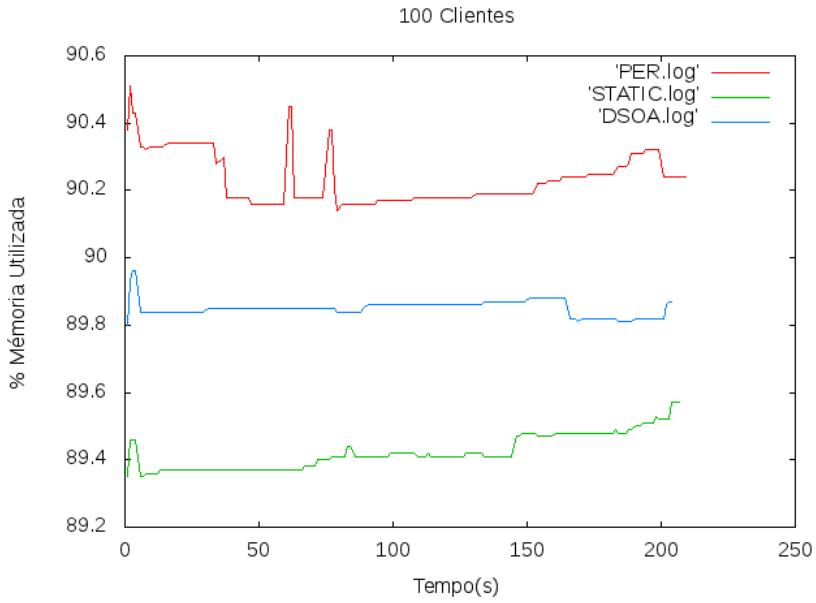


Figura 4.7: Consumo de Memória 100 Clientes.

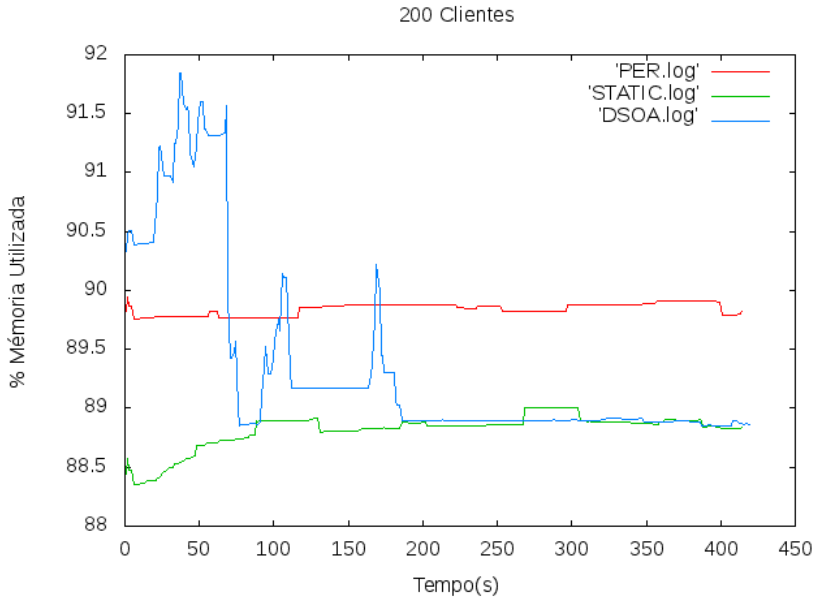


Figura 4.8: Consumo de Memória 200 Clientes.

Nos 3 casos, percebemos que o consumo de memória não varia bruscamente, identificando que a política com adaptação dinâmica ficou estabilizada em umas das políticas *default*. Porém, na Figura 4.8, que apresenta o cenário com um número maior de usuários, consequentemente maior uso dos recursos (tanto de memória, quando de CPU), percebemos o impacto da variação da política no consumo destes recursos.

Analisando o gráfico, para a política com adaptação dinâmica, temos inicialmente um consumo alto de memória, identificando estados de alerta. Deste modo, o provedor foi adaptado ao contexto, por meio da modificação da política de criação, o que fez com que o consumo de memória decaísse gradativamente.

Como a CPU estava com uma carga elevada durante todo o processo (ver Figura 4.9), assim que a memória era estabilizada, ocorria a adaptação visando diminuir a carga de CPU, fazendo com que o consumo de memória crescesse novamente, evidenciando estados de alerta e consequentemente necessitando de adaptação.

Por conta disto, alguns picos são percebidos na política com adaptação dinâmica, até que a utilização dos recursos se estabiliza e a política com adaptação dinâmica não realiza mais trocas.

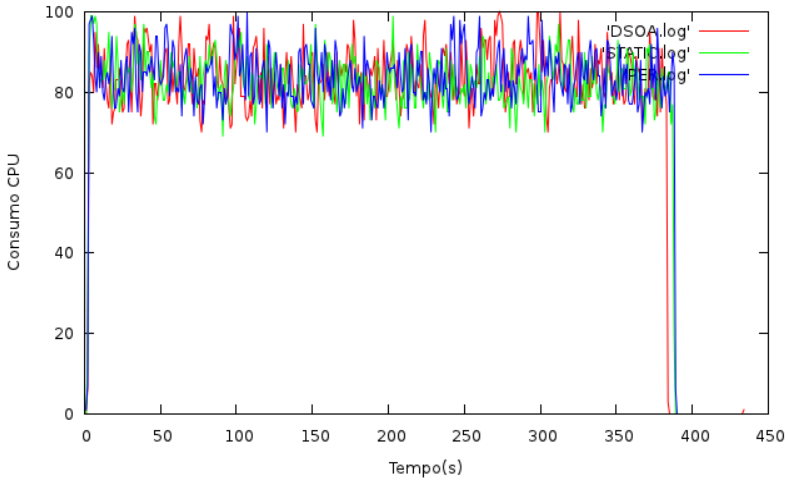


Figura 4.9: Carga CPU 200 Clientes.

Tanto na carga de CPU, quanto no tempo de resposta percebido pelo cliente, não aprestaram variações bruscas, como podemos ver na Tabela 4.2.

Cenário	Static (Singleton)	Per Request	Adaptativa
50 clientes	1,93	1,91	1,90
100 clientes	3,80	3,83	3,85
200 clientes	7,69	7,80	7,81

Tabela 4.2: Média Tempo de Resposta dos Cenários (ms)

4.5 Considerações Finais

Assim, concluímos a avaliação experimental do mecanismo, identificando a necessidade de um estudo mais aprofundado da configuração dos parâmetros, de maneira a melhor aproveitar do dinamismo oferecido pela adaptação dinâmica. Uma vez que, nos testes realizados, mesmo com a modificação da política, o consumo de CPU mostrou-se relativamente independente, tendo uma variação mínima entre as políticas, em todos os cenários.

5

Conclusão e Trabalhos Futuros

Neste trabalho foi projetado e implementado um mecanismo que permite o monitoramento de recursos e atributos de qualidade, visando à adaptação dinâmica do provedor de serviços, em função do contexto em que esta executa.

O principal objetivo por traz desse mecanismo, foi evitar eventuais quebras de contrato devido à demanda ou a limitação de recursos no contexto em que o serviço é provido.

Foram apresentados os principais conceitos relacionados ao trabalho, como orientação a componentes, orientação a serviços e o ciclo PDCA para gestão de qualidade e melhoria de processos de desenvolvimento. Além das principais tecnologias utilizadas na construção dos componentes do mecanismo.

Em seguida, descrevemos a plataforma DSOA, à qual este trabalho foi incorporado, apresentando seus objetivos, sua arquitetura e os principais componentes da plataforma.

Por fim, discutimos sobre a proposta deste trabalho, apresentando uma visão geral do contexto, identificando uma possível extensão à plataforma DSOA, definindo os componentes necessários à realização do trabalho, onde a estratégia de execução foi baseada nas etapas definidas pelo ciclo PDCA, incorporando suas características de busca por melhoria ao mecanismo.

5.1 Contribuições

As principais contribuições deste trabalho são apresentadas a seguir:

- **Suporte à adaptação dinâmica do provedor em função do contexto de execução**

  Suporte à adaptação dinâmica do provedor em função do contexto de execução através do desenvolvimento do gerente de adaptação do provedor (*QoS Provider*

Manager), que adapta o provedor de serviços em função da análise de atributos de qualidade e do contexto de execução. A adaptação é provida através de um conjunto de políticas de criação de instâncias padrão, disponibilizadas como serviços OSGi, o que possibilita a adição de novas políticas de criação em tempo execução.

- **Módulo de monitoramento de recursos**

  Um módulo de monitoramento de recursos de hardware foi construído com base em JMX, garantindo à plataforma uma maneira simples de obtenção de dados relacionados aos recursos de hardware da máquina em que esta executa.

- **Suporte à utilização de diferentes políticas de criação de instâncias em um ambiente OSGi distribuído**

  As políticas de criação de instâncias *default* do iPOJO não dão suporte nativo a execução em um ambiente OSGi, utilizando, apenas a política *singleton*, mesmo que o usuário tenha definido a criação de uma instância diferente por *bundle*. Isso ocorre porque o D-OSGi disponibiliza serviços remotos localmente através de *proxies*. Deste modo, o *endpoint* remoto é visto localmente como um *bundle*, e este "mascara" as invocações locais, de diferentes bundles, ao serviço remoto.

5.2 Trabalhos Futuros

Como pudemos perceber, toda a definição das políticas de gerenciamento de instâncias é feita em tempo de *deployment*.

Assim, uma possível extensão a este trabalho, seria incorporar um componente inteligente que definisse qual a política que melhor atende ao contexto da aplicação, dentre as possíveis, em tempo de execução.

Outra possibilidade de extensão interessante, seria embutir a capacidade de predição de possíveis quebras de contrato, com base na análise da variação da utilização dos recursos. Para isso, seria necessário um estudo mais aprofundado dos padrões existentes de uso de recursos para cada caso específico. Uma solução baseada em redes neurais, aplicação de técnicas de mineração de dados ou mesmo armazenar tais dados em bases consolidadas para análises mais elaboradas realizadas por ferramentas OLAP, podem ser uma extensão interessante deste trabalho.

Uma limitação atual do mecanismo está em tratar quando um estado de alerta, dispara dois profiles ao mesmo tempo, assim, seria necessária a definição de um modelo de

prioridades concorrentes para os `profiles` e `resources` e um estudo focado no cenário de aplicação de cada política.

Referências Bibliográficas

[1] M.P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, 17(02), 2008.

[2] R. J. Rabelo. Arquiteturas orientadas a serviços, 2006. Departamento de Automação e Sistemas.

[3] Ada Diaconescu. *Automatic Performance Optimisation of Component-Based Enterprise Systems via Redundancy*. PhD thesis, School of Electronic Engineering, 2006.

[4] Qusay H. Mahmoud. Service-oriented architecture (soa) and web services: The road to enterprise application integration (eai), 2005. `http://www.oracle.com/technetwork/articles/javase/soa-142870.html`. Último acesso em Setembro/2011.

[5] T. Erl and Safari Tech Books Online (Online service). *Soa: principles of service design*. Prentice Hall, 2008.

[6] Mike P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *4th International Conference on Web Information Systems Engineering (WISE'03)*, pages 3–12, 2003.

[7] Boris Lublinsky. Defining soa as an architectural style, 2007. `http://www.ibm.com/developerworks/library/ar-soastyle`. Último acesso em Setembro/2011.

[8] H. Cervantes and R.S. Hall. Technical concepts of service orientation. *Service-oriented software system engineering: challenges and practices*, pages 1–12, 2005.

[9] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar. Towards recovering the broken soa triangle: a software engineering perspective. In *2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, pages 22–28. ACM, 2007.

[10] M.N. Huhns and M.P. Singh. Service-oriented computing: Key concepts and principles. *Internet Computing, IEEE*, 9(1):75–81, 2005.

[11] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. End-to-end support for qos-aware service selection, invocation and mediation in vresco. *Technical University Vienna, Tech. Rep. TUV-184-2009-03, Jun*, 2009.

[12] O.S.G. Alliance. Osgi service platform, core specification, release 4, version 4.1. *OSGi Specification*, 2007.

[13] R.S. Hall and D. Savage. *Osgi in action*. Manning Publications Co, 2010.

[14] J. Davis and Safari Tech Books Online. *Open source SOA*, volume 8. Manning, 2009.

[15] W3C. Web services activity, 2002. `http://www.w3.org/2002/ws/`. Último acesso em Setembro/2011.

[16] L. Jin, V. Machiraju, and A. Sahai. Analysis on service level agreement of web services. *HP June*, 2002.

[17] SLA@SOI Project. Reference architecture for an sla management framework, 2011. `http://sla-at-soi.eu`. Último acesso em Novembro/2011.

[18] N. Ye. Models of quality of service and quality of information assurance towards their dynamic adaptation. Technical report, DTIC Document, 2011.

[19] M. Völter, M. Kircher, and U. Zdun. *Remoting patterns: foundations of enterprise, Internet and realtime distributed object middleware*, volume 5. Wiley, 2005.

[20] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*. Addison-Wesley Professional, 2002.

[21] Java ee : Xml schemas for java ee deployment descriptors. `http://java.sun.com/xml/ns/javaee/index.html`. Último acesso em Dezembro/2011.

[22] Solution deployment descriptor specification 1.0 - oasis standard. `http://docs.oasis-open.org/sdd/v1.0/os/sdd-spec-v1.0-os.html`. Último acesso em Dezembro/2011.

[23] Extensible markup language (xml) 1.0. `http://www.w3.org/TR/xml`. Último acesso em Dezembro/2011.

[24] Qusay H. Mahmoud. Osgi alliance. `http://www.osgi.org/About/Technology`. Último acesso em Setembro/2011.

[25] ipojo -a flexible and extensible service component model. http://felix.apache.org/site/apache-felix-ipojo.html. Último acesso em Dezembro/2011.

[26] J. Lindfors and M. Fleury. *JMX: Managing J2EE with Java Management Extensions*. Sams Publishing, 2002.

[27] Java management extensions (jmx) technology overview, 2004. http://download.oracle.com/javase/1.5.0/docs/guide/jmx/overview/JMXoverviewTOC.html. Último acesso em Novembro/2011.

[28] Eamonn McManus. What is an mxbean?, 2006. http://weblogs.java.net/blog/emcmanus/archive/2006/02/what_is_an_mxbe.html. Último acesso em Novembro/2011.

[29] M. Douglas and K. SCHMIDT. *Essential SNMP*. O'Reilly, 2005.

[30] Zabbix, the ultimate open source monitoring solution. http://www.zabbix.com. Último acesso em Novembro/2011.

[31] Nagios, the industry standard in it infrastructure monitoring. http://www.nagios.org. Último acesso em Novembro/2011.

[32] F.A. Péricas and L.N. Júnior. Gerência dinâmica tmn baseado na tecnologia java jmx.

[33] Jboss application server. http://www.jboss.org. Último acesso em Novembro/2011.

[34] Jonas opensource java ee application server. http://jonas.ow2.org. Último acesso em Novembro/2011.

[35] Tivoli composite application manager. http://www-01.ibm.com/software/tivoli/products/composite-application-mgrproductline. Último acesso em Novembro/2011.

[36] Walter andrew shewhart. http://pt.scribd.com/doc/35531336/Walter-Shewhart. Último acesso em Novembro/2011.

[37] JingFeng Ning, Zhiyu Chen, and Gang Liu. Pdca process application in the continuous improvement of software quality. In *Computer, Mechatronics, Control and Electronic Engineering (CMCE), 2010 International Conference on*, volume 1, pages 61–65, aug. 2010.

[38] Fábio Felippe De Andrade. O método de melhorias pdca, 2003.

[39] Ana Paula Reusing Pacheco, Bertholdo Werner Salles, Marcos Antônio Garcia, and Osmar Possamai. O ciclo pdca na gestão do conhecimento: Uma abordagem sistêmica. pages 1–5.

[40] Felix Meschberger. Declarative services - dependency injection osgi style. `http://www.eclipsecon.org/2011/sessions/sessions?id=2133`. Último acesso em Dezembro/2011.

[41] The apache aries project. `http://aries.apache.org`. Último acesso em Dezembro/2011.

[42] The apache cxf distributed osgi. `http://cxf.apache.org/distributed-osgi.html`. Último acesso em Dezembro/2011.

[43] Esper - java event stream processor. `http://esper.codehaus.org/esper-1.0.5/doc/reference/en/html/index.html`. Último acesso em Dezembro/2011.

[44] Visualvm - all-in-one java troubleshooting tools. `http://visualvm.java.net`. Último acesso em Novembro/2011.

[45] D.A. Patterson and J.L. Hennessy. *Organização e projeto de computadores*. Elsevier, 3 edition, 2005.

[46] E. Oberortner, S. Sobernig, U. Zdun, and S. Dustdar. Monitoring of performance-related qos properties in service-oriented systems: A pattern-based architectural decision model. 2011.

[47] E. Oberortner, U. Zdun, and S. Dustdar. Patterns for measuring performance-related qos properties in distributed systems. 2010.

[48] Apache jmeter™. `http://jmeter.apache.org`. Último acesso em Dezembro/2011.

Apêndices

A

Provider Manager

Este apêndice mostra um exemplo da definição do arquivo XML necessário à utilização do componente pelo provedor de serviços.

Nele, são expostos os parâmetros utilizados na geração das *Monitoring Configuration*, provenientes do planejamento realizado com objetivo de prevenir possíveis quebras de contrato.

São planejados basicamente 2 tipos de configuração de monitoração. A configuração de monitoração de recursos (*e.g* Carga Total CPU) e a configuração de monitoração de atributos de qualidade (*e.g* Tempo de Processamento).

A.1 XML Provider

```
<ipojo xmlns:qos="br.ufpe.cin.dsoa.manager">
    <component classname="english.EnglishDictionary">
        <provides strategy="br.ufpe.cin.dsoa.strategy.Strategy"/>
        <qos:provider-manager pid="provider.service.english.EnglishDictionaryImpl" name="QoSProvider"
            duration.value="86500000" duration.unit="ms">

            <slo metric="processingTime" statistic="min" threshold="10000"  operation="checkWord" expression="EQ" />
            <slo metric="throughput"     statistic="min" threshold="40000"  operation="checkWord" expression="LT" />
            <slo metric="processingTime" statistic="min" threshold="50000"  operation="translate" expression="LT" />
            <slo metric="throughput"     statistic="min" threshold="70000"  operation="translate" expression="LT" />
            <slo metric="availability"   statistic="min" threshold="0.99"   expression="LT"/>

            <profiles>
                <profile policy="StaticInstance">
                    <resource type="cpu"    attribute="SystemLoadAverage" threshold="0.80" expression="GT"/>
                    <resource type="memory" attribute="FreePhysicalMemorySize" threshold="60" expression="GT"/>
                    <resource type="memory" attribute="FreeSwapSpaceSize" threshold="90" expression="GT"/>
                </profile>
                <profile policy="PerRequestInstance">
                    <resource type="cpu"    attribute="SystemLoadAverage" threshold="0.20" expression="GT"/>
                    <resource type="memory" attribute="CommittedVirtualMemorySize" threshold="750" expression="GT"/>
                </profile>
            </profiles>
        </qos:provider-manager>
    </component>
    <instance component="english.EnglishDictionary" />
</ipojo>
```

A.1.1 Definição dos SLOs

```
<slo metric="processingTime" statistic="min" threshold="10000"
    operation="checkWord" expression="EQ" />
<slo metric="throughput"     statistic="min" threshold="40000"
    operation="checkWord" expression="LT" />
<slo metric="processingTime" statistic="min" threshold="50000"
    operation="translate" expression="LT" />
<slo metric="throughput"     statistic="min" threshold="70000"
    operation="translate" expression="LT" />
<slo metric="availability"   statistic="min" threshold="0.99"
    expression="LT"/>
```

A.1.2 Definição dos Profiles

```
<profiles>
    <profile policy="StaticInstance">
        <resource type="cpu"    attribute="SystemLoadAverage"
            threshold="0.80" expression="GT"/>
        <resource type="memory" attribute="FreePhysicalMemorySize"
            threshold="60" expression="GT"/>
        <resource type="memory" attribute="FreeSwapSpaceSize"
            threshold="90" expression="GT"/>
    </profile>
    <profile policy="PerRequestInstance">
        <resource type="cpu"    attribute="SystemLoadAverage"
            threshold="0.20" expression="GT"/>
        <resource type="memory" attribute="CommittedVirtualMemorySize"
            threshold="750" expression="GT"/>
    </profile>
</profiles>
```