Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação
Mestrado e Doutorado em Ciência da Computação

Dissertação de Mestrado

AdaptiveRME e AspectCompose: Um *Middleware* Adaptativo e um Processo de Composição Orientado a Aspectos para o Desenvolvimento de *Software* Móvel e Ubíquo

Lincoln Souza Rocha

Fortaleza – Ceará
2007

Lincoln Souza Rocha

AdaptiveRME e AspectCompose: Um *Middleware* Adaptativo e um Processo de Composição Orientado a Aspectos para o Desenvolvimento de *Software* Móvel e Ubíquo

Dissertação de Mestrado submetida à Coordenação do Programa de Pós-graduação em Ciência da Computação (MDCC) da Universidade Federal do Ceará (UFC) como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Rossana Maria de Castro Andrade, PhD.

Fortaleza – Ceará
2007

AdaptiveRME e AspectCompose: Um *Middleware* Adaptativo e um Processo de Composição Orientado a Aspectos para o Desenvolvimento de *Software* Móvel e Ubíquo

Dissertação de Mestrado submetida à Coordenação do Programa de Pós-graduação em Ciência da Computação (MDCC) da Universidade Federal do Ceará (UFC) como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Aprovado em ___/___/______

Banca Examinadora

Profa. Rossana Maria de Castro Andrade, PhD. (Orientadora)
Universidade Federal do Ceará – UFC

Profa. Cláudia Maria Lima Werner, DSc.
Universidade Federal do Rio de Janeiro – UFRJ

Prof. Francisco Heron de Carvalho Junior, DSc.
Universidade Federal do Ceará – UFC

Prof. Javam de Castro Machado, DSc.
Universidade Federal do Ceará – UFC

Dedico esta dissertação

Aos meus pais Luiz Rocha Barros e Maria Tereza de Souza Rocha pelo exemplo de amor, união, perseverança e humildade.

As minhas irmãs Karina e Marina pelo espírito fraterno e por sempre me apoiarem na jornada rumo ao saber.

A minha esposa Pollyana, pela demonstração constante de amor, tolerância, paciência e companheirismo.

Agradecimentos

Agradecer é difícil, pois sempre corremos o risco de esquecermo-nos de alguém. Por este motivo, vou começar agradecendo a todos os que me ajudaram durante estes dois anos de mestrado, deste modo, aqueles que não forem citados diretamente já estarão referenciados em parte.

Em primeiro lugar quero agradecer a Deus por construir um laboratório fantástico que é o universo, o qual representa a maior fonte de pesquisa e inspiração de que se tem notícia. Agradeço-o, também, por conceder-me o dom da inquietude que me conduz pelos caminhos desconhecidos do conhecimento.

Aos meus pais, Luiz e Tereza, pelos exemplos de carinho e respeito que só o verdadeiro amor é capaz de proporcionar, meus agradecimentos. As minhas irmãs, Karina e Marina, pelo incentivo constante e pelas poucas, porém prazerosas, horas em família que tivemos ao longo destes dois anos. A minha amada esposa, Pollyana, pela prova de paciência, confiança e amor que me dedicou durante todos este tempo.

Deixar a família é sempre difícil, entretanto, habitar a "Bringelandia" com Antoine, Flávio e Bringel não tem preço. Obrigado amigos pela cordial e enriquecedora conivência que tivemos.

De maneira especial, agradeço a Profa. Rossana, minha orientadora, pelas horas dedicadas às discussões e revisões deste trabalho, pelo auxílio financeiro nos momentos difíceis e pelo exemplo de foco e perseverança. Ao Prof. Heron, meus agradecimentos pelas discussões e ajuda na compreensão de determinados conceitos, principalmente, sobre Componentes de *Software*. A todos os professores e servidores do MDCC e do GREat que, gentilmente, me proporcionaram momentos de discussão, reflexão, alegria e descontração, meu sincero obrigado.

Todo pesquisador em início de carreira objetiva ter publicações para que possa ser referenciado pelos seus pares. Sendo assim, vou agradecer aos meus colegas de mestrado, amigos de doutorado e alunos de IC que, além da minha gratidão, terão seus nomes citados nesta dissertação (em ordem alfabética): Bruno Sabóia, Carlos Pontual, Diana, Ériko, João Gustavo, Máiquel, Márcio, Marcos Dantas, Paula Cibele, Rute e Windson.

Por fim, agradeço a CAPES pelo generoso fomento, imprescindível a realização deste trabalho de dissertação.

“Ciência da computação tem tanto a ver com o
computador como a Astronomia com o
telescópio, a Biologia com o microscópio ou a
Química com os tubos de ensaio. A Ciência não
estuda ferramentas. Ela estuda como nós as
utilizamos e o que descobrimos com elas.”

Edsger Wybe Dijkstra

Resumo

A computação ubíqua é um paradigma computacional de grande abrangência, com aplicabilidades tanto para o cotidiano de um cidadão comum quanto para o tratamento de informações complexas em ambientes hospitalares. Este paradigma propõe uma nova forma de interação homem-computador baseada na proatividade dos computadores para facilitar a vida dos usuários. Entretanto, tal forma de interação demanda um alto grau de colaboração e comunicação entre os elementos computacionais que compõem os ambientes móveis e ubíquos. Um dos principais desafios está relacionado a maneira de como conceber sistemas de *software* capazes de lidar com a alta variação de recursos e serviços, além de garantir a interoperabilidade entre os diversos elementos computacionais que compõem estes ambientes. Este trabalho propõe, então, um *middleware* adaptativo para ambientes móveis e ubíquos com o objetivo de solucionar problemas de heterogeneidade e dinamicidade. Para diminuir o acoplamento entre o *middleware* proposto e as aplicações que o utilizam, esta dissertação também propõe um processo de composição de *software*. O *middleware* utiliza mecanismos de adaptação dinâmica e suporte ao desenvolvimento de *software* sensível ao contexto, e o processo de composição faz uso de técnicas de Programação Orientada a Aspectos (POA). Um estudo de caso é desenvolvido para avaliar a funcionalidade do *middleware* adaptativo e demonstrar a utilização do processo de composição. Além disso, uma análise de desempenho é apresentada para medir o impacto provocado pelo uso do *middleware* proposto em um ambiente de rede sem fio.

Palavras-chave: Computação Ubíqua, *Middleware* Adaptativo, Sensibilidade ao Contexto, Composição de *Software*, Programação Orientada a Aspectos.

Abstract

Ubiquitous computing is an extensive computational paradigm, which can provide solutions to regular citizens or can be useful to handle complex medical environment data, for example. This paradigm proposes a new human-computer interaction approach based on the proactivity of computers that makes users' life easier. However, this approach demands a high level of collaboration and communication among the computational elements that compose the mobile and ubiquitous environments. One of the main challenges is related to the way of conceiving software systems capable of dealing with the high variation of resources and services, along with guaranteeing the interoperability among the diversity of computational elements that compose these environments. Therefore, this work proposes an adaptive middleware for mobile and ubiquitous environments that intends to solve heterogeneity and dynamicity problems. This dissertation also proposes a software composition process to reduce coupling between the proposed middleware and applications. The middleware uses dynamic adaptation and support for context-aware software development mechanisms, and the composition process uses Aspect-Oriented Programming (AOP) techniques. A case study is developed to evaluate the adaptive middleware's functionalities and demonstrate the composing process. Furthermore, a performance analysis is presented to measure the proposed middleware's impact in a wireless network.

Key-words: Ubiquitous Computing, Adaptive Middleware, Context-Aware, Software Composition, Aspect-Oriented Programming.

Índice

Agradecimentos .... **iv**
Resumo .... **vi**
Abstract .... **vii**
Índice .... **viii**
Lista de Figuras .... **x**
Lista de Tabelas .... **xi**
Lista de Abreviaturas e Siglas .... **xii**
Capítulo 1 - Introdução **1**
1.1 Contextualização e Caracterização do Problema 1
1.2 Motivação 3
1.3 Objetivos e Contribuições 4
1.4 Organização da Dissertação 5
Capítulo 2 - Computação Ubíqua **6**
2.1 Princípios e Desafios 6
2.2 Contexto 8
2.3 Adaptação de *Software* 9
2.3.1 Maneiras de Adaptar 10
2.3.2 Adaptação Dinâmica 11
2.3.2.1 Separação de Interesses 11
2.3.2.2 Reflexão Computacional 13
2.3.2.3 Desenvolvimento Baseado em Componentes 14
2.3.2.4 Sistema de *Middleware* 14
2.4 Alguns Problemas e Soluções em Computação Ubíqua 16
2.5 Conclusões 18
Capítulo 3 - Sistemas de *Middleware* Adaptativos **19**
3.1 Definição e Classificação 19
3.2 Exemplos 21
3.2.1 DynamicTAO 22
3.2.2 UIC 23
3.2.3 FlexiNet 24
3.2.4 Open ORB 26
3.2.5 MoCA 27
3.2.6 RME 29
3.2.7 Análise Comparativa 32
3.3 Conclusões 33
Capítulo 4 - AdaptiveRME e AspectCompose **34**
4.1 AdaptiveRME: Visão Geral e Arquitetura 34
4.1.1 Camada de Aquisição de Contexto 37
4.1.1.1 Sensores 38
4.1.1.2 Gerente de Sensores 39
4.1.2 Camada de Interpretação de Contexto 41
4.1.2.1 Interpretadores de Contexto 41
4.1.2.2 Notificadores de Contexto 43
4.1.3 Camada de Serviços 44
4.1.3.1 Serviço de Invocação Remota de Métodos Sensível ao Contexto - SIRMSC 44
4.1.3.1.1 Sincronização 45
4.1.3.1.2 Reconfiguração Dinâmica 46
4.1.3.1.3 Adaptação de Conteúdo 49
4.1.3.2 Serviço de Notificação de Contexto - SNC 49
4.1.3.2.1 Criando Serviços de Contexto 50
4.1.3.2.2 Produtores de Informação Contextual 52

4.1.3.2.3 Consumidores de Serviços de Contexto 53
4.2 O Processo AspectCompose 55
4.2.1 Convenções e Pré-condições 56
4.2.2 Passos do Processo 57
4.3 Conclusões 60
Capítulo 5 - Avaliação 62
5.1 Estudo de Caso 62
5.1.1 UbiPEP: Prontuário Eletrônico do Paciente Ubíquo 62
5.1.1.1 Configuração do SIRMSC 64
5.1.1.2 Serviço de Monitoramento Cardíaco 66
5.1.2 AspectCompose no UbiPEP 69
5.2 Análise de Desempenho 76
5.2.1 Configuração para o Experimento 78
5.2.2 Resultados Obtidos 78
5.3 Conclusões 81
Capítulo 6 - Conclusão 83
6.1 Contribuições 83
6.2 Trabalhos Futuros 84
Referências Bibliográficas 86

Lista de Figuras

Figura 2.1: Processo de desenvolvimento orientado a aspectos. 13
Figura 2.2: Reflexão computacional. 13
Figura 2.3: Composição de componentes. 14
Figura 2.4: Camadas de *middleware* (adaptado de [Sadjadi e Mckinley, 2003]). 15
Figura 3.1: Classificação por tipo de adaptação (adaptado de [Sadjadi e Mckinley, 2003]). 20
Figura 3.2: Classificação por domínio de aplicação (adaptado de [Sadjadi e Mckinley, 2003]). 21
Figura 3.3: Arquitetura de DynamicTAO (Adaptado de [Kon et al., 2000a]). 22
Figura 3.4: Personalizações de UIC (Adaptado de [Román et al., 2001]). 24
Figura 3.5: Pilha de protocolos reflexivos de FlexiNet (Adaptado de [Hayton et al., 1998]). 25
Figura 3.6: Modelo de meta-espaço do Open ORB (Adaptado de [Blair et al., 2001]). 26
Figura 3.7: Arquitetura típica de aplicações MoCA (Adaptado de [Viterbo Filho et al., 2006]). 28
Figura 3.8: Principais módulos de Arcademis (Adaptado de [Pereira, 2006]). 30
Figura 3.9: RME/ORB e algumas fábricas de objetos (Adaptado de [Pereira, 2006]). 31
Figura 4.1: Arquitetura de AdaptiveRME. 35
Figura 4.2: Diagrama de classe do Sensor. 38
Figura 4.3: Diagrama de classe do Gerente de Sensor. 40
Figura 4.4: Arquivo de configuração XML Sensor Descriptor. 40
Figura 4.5: Diagrama de classes do Interpretador de Contexto. 41
Figura 4.6: Associação entre os arquivos XML Context Descriptor e XML Policy Descriptor. 42
Figura 4.7: Diagrama de classes dos Notificadores de Contexto. 44
Figura 4.8: Serviço Invocação Remota de Métodos Sensível ao Contexto. 45
Figura 4.9: Diagrama de classes do Sincronizador. 46
Figura 4.10: Diagrama de classes dos Decoradores de Componentes. 47
Figura 4.11: Diagrama de classes do Configurador de Componentes. 48
Figura 4.12: Diagrama de classes do Adaptador. 49
Figura 4.13: Arquitetura de um Serviço de Contexto. 50
Figura 4.14: Diagrama de classes do Serviço de Contexto. 51
Figura 4.15: Digrama de classes do Produtor. 52
Figura 4.16: Arquivo de configuração XML Provider Descriptor. 53
Figura 4.17: Diagrama de classes do Consumidor. 54
Figura 4.18: Ilustração do processo realizado pelo AspectCompose. 56
Figura 5.1: Diagrama de classes do UbiPEP. 63
Figura 5.2: (a) Autenticação, (b) listagem de paciente e (c) detalhamento de informações em UbiPEP. 63
Figura 5.3: (a) Listagem de exames, (b) detalhamento de exame e (c) imagens em UbiPEP. 64
Figura 5.4: Configuração de contextos e políticas em UbiPEP. 65
Figura 5.5: Arquitetura do serviço de contexto em UbiPEP. 67
Figura 5.6: Arquivo de configuração XML Provider Descriptor para UbiPEP. 67
Figura 5.7: Diagrama de classes do serviço de monitoramento cardíaco. 68
Figura 5.8: Interface Ibenchmark. 77
Figura 5.9: Cenário de execução do experimento. 77
Figura 5.10: Médias do RTT de (a) RME e (b) AdaptiveRME. 79
Figura 5.11: Relação do RTT médio de RME e AdaptiveRME com *overhead* de rede fixo e tempo de processamento variável no servidor. 81
Figura 5.12: Relação do RTT médio de RME e AdaptiveRME com *overhead* de rede variável e tempo de processamento fixo no servidor. 81

Lista de Tabelas

Tabela 3.1: Quadro comparativo. 32
Tabela 4.1: Decoradores concretos. 47
Tabela 5.1: RTT de RME. 80
Tabela 5.2: RTT de AdaptiveRME. 80

Lista de Abreviaturas e Siglas

API	Application Programming Interface
CORBA	Common Object Request Broker Architeture
DCOM	Distributed Component Object Model
GPRS	General Packet Radio Service
IoC	Inversion of Control
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edidion
MAC	Media Access Control
MoCA	Mobile Collaboration Architecture
MVC	Model View Controller
ORB	Object Request Broker
PDA	Personal Digital Assistant
POA	Programação Orientada a Aspectos
QoS	Quality of Service
RME	Remote Method Invocation for J2ME
RMI	Remote Method Invocation
RTT	Round Trip Time
SIRMSC	Serviço de Invocação Remota de Métodos Sensível ao Contexto
SNC	Serviço de Notificação de Contexto
UIC	Universally Interoperable Core

Capítulo 1

Introdução

Esta dissertação apresenta AdaptiveRME, um *middleware* adaptativo para ambientes móveis e ubíquos, e AspectCompose, um processo de composição de *software* orientado a aspectos. Este capítulo introduz, na Seção 1.1, uma visão geral sobre a computação ubíqua e algumas problemáticas relacionadas ao tema. Em seguida, a Seção 1.2 expõe a motivação para o desenvolvimento deste trabalho. Na Seção 1.3, são descritos os principais objetivos desta dissertação. Por fim, a Seção 1.4 apresenta a estrutura organizacional desta dissertação.

1.1 Contextualização e Caracterização do Problema

A computação ubíqua idealizada por *Weiser* [Weiser, 1991] é considerada um dos mais atuais e abrangentes paradigmas da computação. *Weiser* dizia que a computação residiria nos mais inusitados objetos (e.g., etiquetas de roupas, xícaras de café, interruptores de luz e canetas) tornando-se imperceptível aos olhos dos homens [Weiser, 1993]. Neste contexto, onde a computação torna-se pervasiva[1] ao cotidiano, as pessoas passam a conviver com os computadores ao invés de simplesmente utilizá-los.

De certo modo, pode-se entender a computação ubíqua como sendo uma computação distribuída, dinâmica e heterogênea realizada por dispositivos computacionais que atuam de maneira discreta e colaborativa nos ambientes onde estão inseridos com o intuito de auxiliar os usuários na execução de suas tarefas.

Avanços contínuos nas áreas de microeletrônica e telecomunicação têm contribuído significativamente para redução dos preços e proliferação dos dispositivos móveis (i.e., dispositivos portáteis capazes de se comunicar através de alguma tecnologia de comunicação sem fio). Sendo assim, a possibilidade de processamento e acesso a informação de qualquer lugar e a qualquer instante tem se tornado cada vez mais presente no cotidiano das pessoas. Neste contexto, a indústria tem buscado produzir dispositivos móveis diversificados, porém, com funcionalidades semelhantes e interoperáveis. Por exemplo, não é difícil encontrar em uma loja

[1] Neste trabalho o termo pervasivo ou computação pervasiva é utilizado como sinônimo do termo inglê *Pervasive Computing*.

especializada, um PDA com funcionalidades de telefone celular, ou ainda, diferentes dispositivos (e.g., celulares, *Palms* e *mp3 players*) capazes de trocar informação através de diferentes tecnologias de comunicação (e.g., *bluetooth*, *wi-fi* e GPRS - *General Packet Radio Service*). Todos esses adventos podem ser vistos como um meio para a concretização da computação ubíqua.

Por permitir processamento e acesso ubíquo (de qualquer lugar e a qualquer instante) à informação, a computação ubíqua tem se mostrado uma alternativa para diversos domínios de aplicação (e.g., ambientes domésticos, turismo, transportes e *healthcare*). Em particular, as clínicas médicas e os hospitais apresentam características peculiares que os tornam ambientes ideais para a utilização da computação ubíqua [Favela, 2004], dentre estas características destacam-se: necessidade de coordenação e colaboração entre especialistas de áreas diferenciadas, intensa troca de informação e alta mobilidade de equipe, pacientes, documentos e equipamentos. Desta forma, a Telemedicina[2] tem se tornado um cenário fértil para as aplicabilidades da computação ubíqua (e.g., telemonitoração, telediagnóstico e acesso ubíquo a registros médicos).

Contudo, além do alto grau de descentralização, heterogeneidade (de *hardware* e *software*) e dinamicidade dos ambientes ubíquos, o "desaparecimento" do computador do cotidiano das pessoas tem conduzido os engenheiros de *software* a uma reflexão sobre a forma como sistemas de *software* devem ser desenvolvidos para estes ambientes. De acordo com *Bardram* [Bardram, 2004], sensibilidade ao contexto é a palavra chave para computação ubíqua. Contexto é toda e qualquer informação que pode ser usada para caracterizar uma situação de uma pessoa, um lugar, ou um objeto que é considerado relevante para a interação entre o usuário e a aplicação, incluindo o próprio usuário e a própria aplicação [Dey e Abowd, 1999]. Assim, um sistema sensível ao contexto[3] é um sistema que utiliza contexto para prover informação ou serviço relevante para usuário, onde a relevância depende da tarefa do usuário [Dey e Abowd, 1999].

Sendo assim, sistemas ubíquos devem ser capazes de capturar o contexto e utilizá-lo para guiar o seu comportamento e auxiliar os usuários na execução de suas tarefas. Neste cenário, este trabalho busca prover mecanismos que abstraiam para o engenheiro de *software* as tarefas relacionadas com aquisição e tratamento de informações contextuais (e.g., largura de banda e memória disponível) e provisão de ações adaptativas (e.g., reconfiguração dinâmica e adaptação de conteúdo) em função de variações no contexto, com o intuito de facilitar o desenvolvimento de *software* ubíquo.

[2] Segundo a Organização Mundial de Saúde, Telemedicina é a oferta de serviços ligados aos cuidados com a saúde, nos casos em que a distância é um fator crítico. Tais serviços são providos por profissionais da área de saúde, usando Tecnologias de Informação e Comunicação (TICs) para fazer o intercâmbio de informações válidas para diagnósticos, prevenção e tratamento de doenças, para a contínua educação de prestadores de serviços em saúde e para fins de pesquisas e avaliações.

[3] Neste trabalho o termo sensível ao contexto é utilizado como tradução do termo em inglês *Context-Aware*.

1.2 Motivação

Como mencionado anteriormente, a heterogeneidade e a dinamicidade são características bastante comuns em sistemas ubíquos. Garantir a interoperabilidade entre dispositivos com diferente poder computacional tem sido o objeto de estudo de vários pesquisadores na última década [Kon et al, 2000a][Kon et al, 2000b][Grimm et al, 2001a][Grimm et al, 2001b][Román et al, 2002][Garlan et al, 2002][Sadjadi e Mckinley, 2003][Mckinley et al, 2004][Sacramento et al., 2004][Grimm et al, 2004][Pereira et al., 2006][Viterbo Filho et al., 2006]. Além disso, neste cenário dinâmico, aspectos como movimentação, variação na disponibilidade de recursos e serviços devem ser levados em consideração pelo engenheiro de *software* durante o processo de desenvolvimento de *software* móvel e ubíquo [Endler e Silva, 2001].

Entretanto, capturar, interpretar e representar cada possível contexto de execução e utilizá-los para delinear o comportamento do *software* não é uma atividade trivial, o que constitui um desafio para o desenvolvimento de *software* para ambientes móveis e ubíquos. Distribuição, adaptação de conteúdo, reconfiguração dinâmica e sensibilidade ao contexto representam requisitos não-funcionais altamente desejáveis em sistemas móveis e ubíquos.

Neste cenário, sistemas de *middleware* adaptativos [Sadjadi e Mckinley, 2003] surgem como uma alternativa viável para facilitar o desenvolvimento de *software* ubíquo, separando e encapsulando interesses específicos e provendo serviços adaptativos para as aplicações. Contudo, vale ressaltar que soluções adotadas por sistemas de *middleware* adaptativos para ambientes distribuídos, em geral, fazem uso extensivo de reflexão computacional e algoritmos específicos que tratam de controle de rotas, tolerância a falhas, segurança e políticas de transações [Coulouris et al., 2005]. Tais soluções pressupõem um poder computacional mínimo por parte dos elementos que compõem o sistema, condição esta que não pode ser garantida em ambientes heterogêneos e dinâmicos como os que caracterizam a computação ubíqua. Sendo assim, este é um dos desafios na construção de sistemas de *middleware* adaptativos para ambientes móveis e ubíquos.

Embora a utilização de sistemas de *middleware* adaptativos contemple a separação de interesses e proporcione um alto nível de abstração e encapsulamento para o engenheiro de *software*, esta abordagem por si só não se mostra eficiente em evitar o entrelaçamento e espalhamento de código, pois, em geral, sistemas de *middleware* são intrusivos e transversais à lógica das aplicações. Por exemplo, chamadas à API (*Application Program Interface*) do *middleware* permeiam o código por diversos módulos, o que dificulta a manutenibilidade e, por conseqüência, diminui a qualidade do *software* produzido.

Com o mesmo intuito de fazer a separação de interesses, a POA (Programação Orientada a Aspectos) proposta por *Kiczales* [Kiczales et al., 1997] tem por objetivo modularizar decisões de projeto que a programação orientada a objetos não consegue tratar de maneira adequada (e.g., distribuição, segurança e adaptação). De acordo com *Soares* e *Borba* [Soares e Borba, 2002], o tratamento inadequado destas decisões de projeto pode resultar num entrelaçamento e espalhamento de código que dificultam o desenvolvimento e comprometem o reúso e a manutenção. Assim, trabalhos como o de *Vaysse* [Vaysse et al., 2005] propõem a utilização da POA como ferramenta de composição de *software* para compor aplicações utilizando os serviços fornecidos pelos sistemas de *middleware*.

Contudo, embora a literatura apresente vários sitemas de *middlware* adaptativos, é importante salientar que parte deles está focada na heterogeneidade de *software* e *hardware*; parte na reconfiguração dinâmica para atendimento a redefinição de requisitos; e parte no suporte ao desenvolvimento de aplicações sensíveis ao contexto. Entretanto, um *middleware* adaptativo para o ambiente ubíquo deve ser capaz deve ser capaz de lidar com essas três questões. Além de prover mecanismos que permitam capturar, representar e tratar informações contextuais.

1.3 Objetivos e Contribuições

O presente trabalho tem como objetivo principal propor um *middleware* adaptativo para facilitar o desenvolvimento de *software* para ambientes móveis e ubíquos. Como objetivo secundário, um processo de composição de *software* orientado a aspectos é criado para diminuir o acoplamento entre o *middleware* proposto e as aplicações que o utilizam.

AdaptiveRME, o *middleware* adaptativo proposto, é uma extensão do *middleware* RME (*Remote Method Invocation for* J2ME) [Pereira et al., 2006]. AdaptiveRME provê mecanismos de reconfiguração dinâmica e adaptação de conteúdo através de um Serviço de Invocação Remota de Métodos Sensível ao Contexto (SIRMSC) e suporte ao desenvolvimento de aplicações ubíquas sensíveis ao contexto através de um Serviço de Notificação de Contextos (SNC). Com estes serviços, AdaptiveRME lida com a dinamicidade e heterogeneidade dos ambientes móveis e ubíquos.

Com a finalidade de diminuir o acoplamento e a transversalidade impostos pela utilização de AdaptiveRME, também é proposto um processo de composição *software*, denominado AspectCompose, que faz uso de técnicas de POA para compor as aplicações usando o SIRMSC de AdaptiveRME. AspectCompose diminui o acoplamento e o entrelaçamento gerados com o intuito de aumentar a manutenibilidade e o reúso das aplicações.

Para avaliar a proposta, um estudo de caso e uma análise de desempenho são apresentados. O estudo de caso mostra o uso de AdaptiveRME e de AspectCompose no desenvolvimento de uma versão ubíqua e simplificada do Prontuário Eletrônico do Paciente [Costa, 2001] para um ambiente móvel. Já a análise de desempenho, avalia o desempenho de AdaptiveRME em um ambiente de rede sem fio.

De uma maneira geral, a principal contribuição desta dissertação é facilitar o desenvolvimento de *software* móvel e ubíquo e promover o reúso e a manutenibilidade através do uso conjunto de AdaptiveRME e AspectCompose.

1.4 Organização da Dissertação

Esta dissertação está organizada em seis capítulos. O presente capítulo fez uma ligeira introdução ao tema, contextualizando o assunto abordado neste trabalho. Além disso, o Capítulo 1 apresentou a motivação, o objetivo e a principal contribuição deste trabalho de dissertação.

O Capítulo 2 aborda os principais conceitos relacionados a temática abordada por esta dissertação. Inicialmente são introduzidos os princípios e principais desafios da computação ubíqua, no contexto deste trabalho. Em seguida, são apresentados os conceitos de contexto, sensibilidade ao contexto, adaptação de *software* e soluções para alguns problemas em computação ubíqua.

O Capítulo 3 mostra definições, classificações e modelos de sistemas de *middleware* adaptativos. Ainda neste capítulo, um estudo comparativo entre os modelos de sistemas de *middleware* adaptativos apresentados é fornecido.

No Capítulo 4 são apresentadas as principais contribuições desta dissertação. Inicialmente, detalhes sobre o modelo de manipulação de contexto do *middleware* proposto é apresentado e, em seguida, características de arquitetura, projeto e implementação são melhor detalhadas. Posteriormente, o processo de composição proposto é descrito.

O Capítulo 5 descreve a avaliação deste trabalho de dissertação. Inicialmente, um estudo de caso é apresentado e, em seguida, tanto a descrição quanto os resultados de uma análise de desempenho são mostrados.

Por fim, o Capítulo 6 apresenta as conclusões e possíveis linhas de pesquisa a serem investigadas como trabalhos futuros.

Capítulo 2

Computação Ubíqua

Este capítulo apresenta uma visão geral sobre a computação ubíqua com foco no desenvolvimento de *software* para este ambiente. Inicialmente, na Seção 2.1 são introduzidos os princípios e os desafios mais relevantes da computação ubíqua no contexto deste trabalho. Na Seção 2.2 são postulados conceitos sobre contexto e aplicações sensíveis ao contexto. A Seção 2.3 introduz e discute características de adaptabilidade de *software* além de classificar e apresentar as principais tecnologias utilizadas no desenvolvimento de *software* adaptativo. Já a Seção 2.4 traz uma descrição de soluções para problemas em computação ubíqua abordados por pesquisadores em projetos científicos. Por fim, a Seção 2.5 apresenta algumas conclusões e considerações sobre este capítulo.

2.1 Princípios e Desafios

A história da computação pode ser dividida em função dos ambientes computacionais predominantes em três épocas distintas: a passada, dos mainframes, caracterizada pela relação de "vários usuários por máquina"; a atual, dos computadores pessoais, cuja relação predominante é a de "um usuário por máquina"; e a futura, da computação ubíqua, caracterizada principalmente pela relação "um usuário para várias máquinas".

A computação ubíqua, idealizada por *Mark Weiser* [Weiser, 1991], tem como objetivo principal tornar o uso do computador mais amigável, fazendo com que muitos computadores estejam disponíveis a todo instante, mas de forma invisível para o usuário. Para alcançar tal objetivo, a computação ubíqua prevê um mundo com vários tipos de dispositivos interconectados através de enlaces de redes sem fio em todos os lugares. Tal "onipresença" computacional outorga aos usuários a possibilidade de acesso a informação de todo lugar a todo instante.

A idéia central da computação ubíqua é criar um novo tipo de relacionamento entre pessoas e computadores, marcado pela proatividade dos computadores na resolução de problemas e no provimento de informação relevante e contextualizada para os usuários, mas sem

que estes percebam que estão sendo ajudados. Entretanto, "desaparecer" com os computadores do cotidiano das pessoas representa um grande desafio para a computação ubíqua.

Para *Streitz* e *Nixon* [Streitz e Nixon, 2005], o "desaparecimento" do computador pode ser de duas naturezas: física e mental. O "desaparecimento" de natureza física está relacionado com o grau de miniaturização dos dispositivos computacionais, ao passo que, o "desaparecimento" de natureza mental tem a ver com a maneira como os usuários interagem e percebem os computadores no seu cotidiano. Por isso, os computadores ubíquos, pequenos ou não, devem prover interfaces de comunicação casuais que não requeiram do usuário muita disponibilidade e alto grau de concentração.

De certo modo, pode-se entender a computação ubíqua como sendo uma computação distribuída, realizada por dispositivos computacionais que atuam de forma discreta nos ambientes onde estão inseridos. Tais dispositivos podem estar embutidos em objetos de baixa ou nenhuma mobilidade (e.g., quadros, lâmpadas e aparelhos de televisão) e em objetos com alto grau de mobilidade (e.g., automóveis, aparelhos celulares e PDAs).

Ambientes ubíquos podem ser espaços físicos quaisquer (casas, salas de aula, escritórios, hospitais, universidades e bairros) repletos de dispositivos computacionais ou eletro-eletrônicos (e.g., sensores de umidade, computadores, celulares e televisores) capazes de realizar algum processamento útil para os freqüentadores destes ambientes. Tal capacidade pressupõe que os dispositivos estejam intimamente ligados ao cotidiano dos freqüentadores e em harmonia com os demais objetos presentes no ambiente. Por isso, ambientes ubíquos são altamente sensíveis ao contexto. Por exemplo, informações sobre localização de pessoas e objetos, umidade relativa do ar e níveis de sinais vitais de um paciente internado são tipos de informações que podem ser utilizadas por aplicações para auxiliar pessoas na realização das mais diversas atividades.

A heterogeneidade e a dinamicidade são características típicas de ambientes ubíquos. A grande variedade de dispositivos com diferentes tamanhos, capacidade de processamento, armazenamento e tecnologia de comunicação dificulta a interoperabilidade. Tudo isso, aliado a necessidade de gerenciamento dinâmico e auto-organização impostos pela dinamicidade do ambiente, representam entraves no desenvolvimento de *software* ubíquo.

Outro fator que deve ser levado em consideração no desenvolvimento de aplicações ubíquas é a maneira como as informações sobre o ambiente são capturadas, processadas pelos dispositivos e utilizadas para auxiliar o homem na execução de suas tarefas. Neste contexto, técnicas que permitam capturar, representar e tratar informações contextuais bem como tecnologias que habilitem o *software* a adaptar-se em função destas informações são requisitos

fortemente desejáveis em sistemas de *software* ubíquos.

2.2 Contexto

Dey e *Abowd* [Dey e Abowd, 1999] dizem que o primeiro trabalho a utilizar o termo sensível ao contexto foi o de *Schilit* e *Theimer* [Schilit e Theimer, 1994], o qual se referia a contexto como localização, identidades de pessoas e objetos e as mudanças desses objetos. Já *Pascoe* em [Pascoe, 1998] diz que contexto é o subconjunto de estados físicos e conceituais de interesse de uma entidade particular.

Schilit [Schilit et al., 1994] diz que os principais aspectos do contexto são: "onde você está", "quem está com você" e "quais recursos estão próximos" argumentando que o contexto pode ser dividido em três categorias:

- **Contexto computacional** está relacionado diretamente com o dispositivo. Capacidade de processamento, memória disponível, custos associados à comunicação, conectividade de rede, largura de banda e recursos disponíveis (e.g., impressoras, fax e serviços) são informações que caracterizam este tipo de contexto;
- **Contexto do usuário** está relacionado diretamente com o usuário. Informações como perfil, localização, pessoas próximas, variação de humor e estado de saúde são exemplos de informações relevantes para este tipo de contexto;
- **Contexto físico** está relacionado com o ambiente físico onde o dispositivo e/ou o usuário possam estar inseridos. Luminosidade, nível de poluição sonora, condições do trânsito, temperatura e umidade são exemplos de informações importantes para esse tipo de contexto.

Entretanto, *Dey* e *Abowd* [Dey e Abowd, 1999] e *Dey* [Dey, 2001] argumentam que as definições de *Schilit* e *Theimer* [Schilit e Theimer, 1994] e de *Pascoe* [Pascoe, 1998] são muito específicas e sugerem um novo conceito para contexto. Para *Dey* e *Abowd*, contexto é qualquer informação que pode ser usada para caracterizar uma situação de uma entidade. Nesse caso, uma entidade pode ser uma pessoa, um lugar, ou um objeto que é considerado relevante para a interação entre um usuário e uma aplicação, incluindo o próprio usuário e a própria aplicação.

A definição de contexto utilizada nesta dissertação estende a definição de *Dey* e *Abowd*, apenas adicionando a idéia de que o estado de uma entidade pode ser relevante para a interação entre os módulos de uma mesma aplicação ou para a interação entre diferentes aplicações.

Dey e *Abowd* [Dey e Abowd, 1999] argumentam que um sistema sensível ao contexto é um sistema que utiliza contexto para prover informação relevante e/ou serviços ao usuário, onde a relevância depende da tarefa do usuário. Assim, aplicações sensíveis ao contexto devem ser capazes de capturar e interpretar informações de contexto de maneira transparente ao usuário, disponibilizando-as num formato que possa ser entendido por um ambiente computacional.

Para diferenciar cada informação de contexto, *Dey* e *Abowd* [Dey e Abowd, 1999] sugerem separá-las em cinco dimensões, de modo que respondendo às seguintes perguntas é possível ter ciência do contexto: *who* (quem) - está relacionada com as informações de identidade do usuário; *where* (onde) - está relacionada à localização do usuário; *when* (quando) - está relacionada com o contexto temporal; *what* (o quê) - está relacionada com a identificação de atividades do usuário; e, por fim, *why* (por que) - está relacionada com as informações que provocaram determinadas ações do usuário.

De maneira complementar, *Henricksen* em [Henricksen et al, 2002] apresenta uma discussão sobre como modelar o contexto levando em consideração características temporais das informações contextuais. *Henricksen* argumenta que dependendo do tipo de informação o contexto pode ser estático ou dinâmico. O contexto estático é composto por informações que não variam seus valores (e.g., números de CPF e tamanho do *display*) com o decorrer do tempo de vida de uma entidade (e.g., uma pessoa e um aparelho celular). Já o contexto dinâmico·é composto por todas as demais informações e podem ser classificadas em três tipos: sentidas, explícitas e interpretadas.

Informações de contexto dinâmicas sentidas são informações capturadas por sensores físicos e lógicos (e.g., localização e luminosidade do ambiente). Já as informações de contexto dinâmicas explícitas são as informações fornecidas diretamente pelo usuário (e.g., senhas e demais informações pessoais). Diferente das demais, as informações dinâmicas interpretadas são informações derivadas de uma ou mais informações contextuais por meio de regras de inferência simples ou por algoritmos complexos de Inteligência Artificial. São exemplos de informações de contexto dinâmicas sentidas as condições de saúde de um paciente aferidas do seu batimento cardíaco e da pressão sangüínea.

2.3 Adaptação de *Software*

Na literatura, a adaptabilidade de sistemas de *software* é, em geral, referenciada utilizando como sinônimo das palavras **adaptável** e **adaptativo**. Entretanto, no escopo deste trabalho, um *software* adaptável é entendido como sendo um sistema que permite adaptar facilmente uma

estrutura completa ou partes específicas devido a mudanças nos requisitos [Tekinerdogan et al., 1997], ao passo que um *software* adaptativo é entendido como um sistema capaz de modificar dinamicamente seu comportamento em função de variações no contexto do ambiente em que executa [Lieberherr, 1995]. Sendo assim, a adaptabilidade pode ser atingida de maneira estática (adaptável) ou dinâmica (adaptativa).

2.3.1 Maneiras de Adaptar

Para que se possa empregar a adaptabilidade no desenvolvimento de *software* é necessário conhecer as maneiras de se fazer adaptação. *Carvalho* e *Andrade* utilizam em [Carvalho e Andrade, 2006] alguns modelos de adaptação, destacando em [Carvalho et al., 2005] os seguintes:

- **Adaptação à descrição** é a capacidade de descrever uma aplicação permitindo sua adequação ou transformação em diferentes linguagens ou plataformas de programação. Nesse caso, a aplicação é descrita em uma metalinguagem e é transformada em uma linguagem de programação (e.g., Java e C++) ou em uma linguagem de marcação de hipertexto (e.g., XHTML e WML);

- **Adaptação ao dispositivo** é a habilidade que uma aplicação possui de adequar seu modo de execução às características do dispositivo. Essas características podem ser estáticas, como número de cores e dimensões da tela, e/ou dinâmicas, como a quantidade de memória e bateria disponíveis;

- **Adaptação ao contexto** é a propriedade de uma aplicação de adequar-se a mudanças no contexto em que executa. As mudanças no contexto podem ser decorrentes, por exemplo, das alterações da localização do dispositivo, do interesse do usuário e da largura de banda de comunicação. Este tipo de adaptação exige um alto nível de dinamicidade.

Dependendo do modelo de adaptação escolhido, a aplicação pode ser adaptada em tempo de compilação, inicialização ou execução. Se a adaptação em tempo de compilação é considerada, o engenheiro de *software* deve desenvolver uma aplicação generalizada que será disponibilizada na forma de versões adequadas às características do ambiente de execução alvo. Por outro lado, se a adaptação em tempo de inicialização for adotada, o engenheiro de *software* deve desenvolver uma aplicação que seja passível de ajustes por meio de parâmetros (e.g., variáveis de ambiente e arquivos de configuração). Diferente dos dois modelos citados, se a adaptação em tempo de execução é adotada, o engenheiro de *software* deve desenvolver apenas uma única aplicação que seja capaz de identificar o ambiente de execução onde esta inserida e adaptar seu comportamento

a ele.

Contudo, é importante salientar que a adaptação não está restrita apenas ao comportamento da aplicação em função do ambiente de execução alvo, mas também à adequação dos dados aos quais ela pode acessar. Esta maneira de fazer adaptação é denominada **adaptação de conteúdo**.

Desta forma, para que o engenheiro de *software* consiga desenvolver aplicações adaptativas é necessário que ele utilize técnicas que permitam às aplicações capturar e interpretar informações sobre o contexto corrente e utilizá-las para delinear o comportamento do *software*.

2.3.2 Adaptação Dinâmica

Segundo *Mckinley* [Mckinley et al., 2004], uma variedade de técnicas permite ao *software* se adaptar ao ambiente de execução. Tais técnicas possibilitam que a estrutura do *software* seja mudada para reparar erros, melhorar o desempenho, aumentar a disponibilidade e a segurança em resposta a ataques. *Mckinley* aponta duas abordagens gerais para fazer adaptação dinâmica no *software*, são elas: adaptação parametrizada e adaptação composicional. A adaptação parametrizada envolve a modificação de variáveis de ambiente que determinam o comportamento do *software*. Já a adaptação composicional possibilita que módulos sejam adicionados ou substituídos, dinamicamente, com o intuito de melhor ajustar o *software* às variações do ambiente em que executa.

É importante observar que a adaptação parametrizada permite apenas ajustar parâmetros ou configurar o *software* para usar uma estratégia diferente, previamente implementada, não possibilitando a incorporação de novas estratégias. Por isso, a adaptação composicional se mostra mais eficiente, pois permite ao *software* se ajustar de acordo com suas novas necessidades. *Weiser* denomina de *calm computing* [Weiser e Brown, 1996] a adaptação sem a ação direta do homem, sendo este um dos principais objetivos no desenvolvimento de *software* ubíquo, foco deste trabalho.

De acordo com *Mckinley*, as principais abordagens/tecnologias utilizadas para fazer adaptação composicional são: separação de interesses (*separation of concerns*), reflexão computacional, desenvolvimento baseado em componentes e sistema de *middleware*.

2.3.2.1 Separação de Interesses

A separação de interesses [Dijkstra, 1974] preconiza a identificação e o tratamento individualizado dos diferentes interesses envolvidos no desenvolvimento de um *software*. Um

interesse pode ser compreendido como parte de um problema que se deseja tratar de maneira isolada. Interesses podem ser requisitos, propriedades ou características que um *software* deve possuir ou implementar. A separação de interesses acredita que a decomposição do *software* em módulos independentes simplifica o desenvolvimento, facilita a manutenção e promove o reuso.

Uma das abordagens mais atuais utilizada na separação de interesses é a Programação Orientada a Aspectos (POA). Proposta por *Kiczales* [Kiczales et al., 1997], a POA tem por objetivo modularizar decisões de projeto que a programação orientada a objetos não consegue tratar de maneira adequada. Isto se deve ao fato de que alguns requisitos violam a modularização natural do código [Kiczales et al., 2001]. A POA propõe que esses requisitos, ditos transversais (*crosscutting concerns*), sejam tratados em um plano diferente ao do sistema.

Soares e *Borba* em [Soares e Borba, 2002] afirmam que o tratamento inadequado de requisitos transversais pode resultar num espalhamento (*scattering*) e entrelaçamento (*tangling*) de código com diferentes propósitos. O espalhamento está relacionado com o grau de dispersão do código do requisito transversal pelos diversos módulos do programa, ao passo que o entrelaçamento está relacionado com a confusão gerada pela mistura de código entre requisitos funcionais e transversais em um mesmo módulo do programa. Tal entrelaçamento e espalhamento podem impactar diretamente na qualidade do *software* produzido.

A POA propõe que os requisitos funcionais sejam implementados separadamente dos requisitos transversais. De acordo com [Tirelo et al., 2004], o processo de desenvolvimento de *software* orientado a aspectos pode ser dividido em três fases distintas (Figura 2.1):

- **Identificação de interesses**: nesta fase é feita a identificação e a separação dos requisitos funcionais e transversais da aplicação;
- **Implementação de interesses**: nesta fase é feita a implementação de cada um dos interesses identificados, utilizando-se recursos tradicionais da programação orientada a objetos (e.g., encapsulamento e herança);
- **Recomposição**: nesta fase são definidas as regras de recomposição do sistema. Elas definem como os requisitos deverão ser compostos para formar o sistema final, através de um processo denominado combinação (*weaving*).

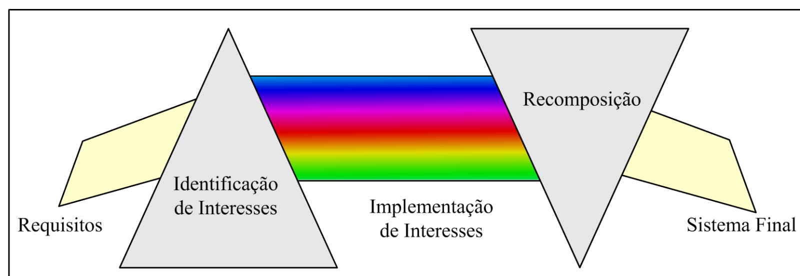


Figura 2.1: Processo de desenvolvimento orientado a aspectos.

2.3.2.2 Reflexão Computacional

A reflexão computacional [Maes, 1987] é a capacidade de um sistema de observar ou até mesmo modificar a sua estrutura ou comportamento. A idéia de se utilizar reflexão em sistemas computacionais é permitir que um sistema faça uso de parte do seu processamento em beneficio próprio, para monitorar ou modificar sua própria estrutura ou comportamento em função de algum evento ou situação especial (e.g., erros, mudança de requisitos e variações no contexto e execução).

Quando utilizada, a reflexão computacional subdivide os sistemas em dois níveis (Figura 2.2): o nível-base, responsável pelo processamento das funcionalidades básicas do sistema, e o meta-nível, responsável pelo processamento das características não-funcionais do sistema.

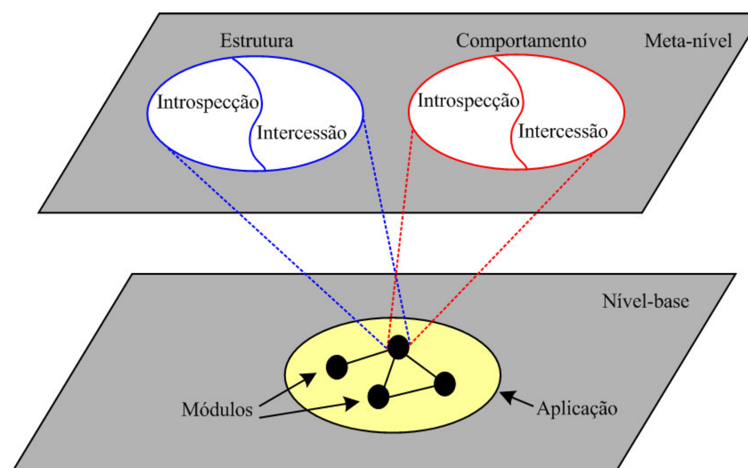


Figura 2.2: Reflexão computacional.

A reflexão computacional pode ser dividida nas atividades de introspecção e intercessão (Figura 2.2). Na atividade de introspecção o meta-nível é utilizado para inspecionar e manipular a estrutura e o comportamento interno do sistema como, por exemplo, identificar tipos de dados e monitorar execuções de métodos em linguagens orientadas a objetos. Já na atividade de intercessão o meta-nível é utilizado para alterar a estrutura interna do sistema e controlar o seu

comportamento funcional. São exemplos de ações de intercessão, a adição de membros de classes e interceptação de chamada de métodos para realização de meta-computação.

2.3.2.3 Desenvolvimento Baseado em Componentes

Um componente pode ser definido como uma unidade de *software* independente e autocontida que encapsula seu projeto e implementação e oferece uma interface bem definida que especifica o que ele requer e o ele que provê para o meio externo. Através de interfaces públicas, um componente pode ser conectado a outros para compor um novo componente ou um sistema completo. Dessa forma, o desenvolvimento baseado em componentes permite a reutilização de unidades de *software* que foram projetadas, desenvolvidas, compostas, testadas e disponibilizadas por outros engenheiros de *software* [Wang e Qian, 2005].

Sistemas baseados em componentes podem ser compostos de maneira estática ou dinâmica (Figura 2.3). Na composição estática, o engenheiro de *software* pode combinar diversos componentes, em tempo de projeto ou compilação, gerar um novo componente ou um sistema completo. A principal desvantagem deste tipo de composição é que uma vez gerado, o sistema não mais pode ser alterado em tempo de execução, neste caso, toda alteração requer que o sistema seja interrompido, atualizado e depois reiniciado. Já na composição dinâmica, o engenheiro de *software* pode adicionar, remover, substituir e reconfigurar componentes do sistema, em tempo de execução, permitindo a este adequar-se dinamicamente a seus novos interesses e necessidades.

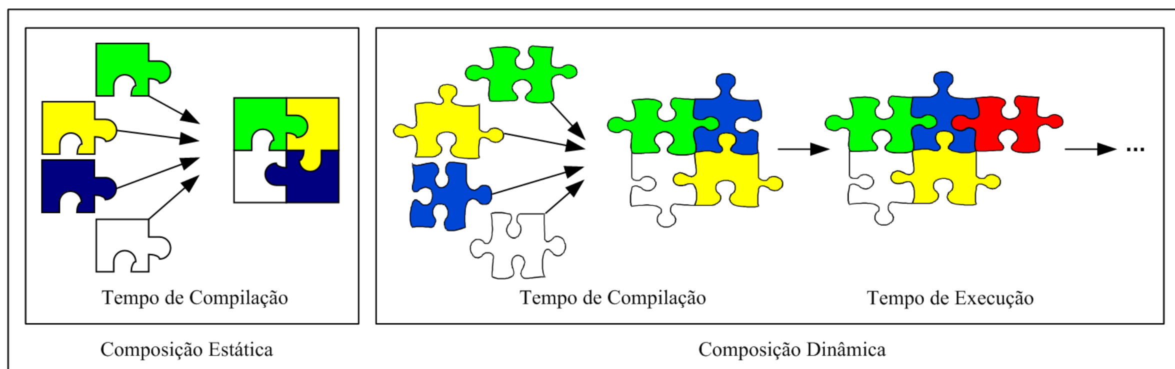


Figura 2.3: Composição de componentes.

2.3.2.4 Sistema de *Middleware*

No âmbito dos sistemas distribuídos pode-se definir sistema de *middleware* como sendo uma camada de *software* intermediária entre as aplicações e o sistema operacional que abstrai a

heterogeneidade de *software* e *hardware* [Geihs, 2001]. Diferentemente das demais abordagens utilizadas para fazer adaptação dinâmica, sistemas de *middleware* fornecem uma camada de abstração onde os engenheiros de *software* podem inserir comportamento adaptativo de maneira direta ou simplesmente fazendo uso das demais abordagens anteriormente descritas (separação de interesses, reflexão computacional e desenvolvimento baseado em componentes).

Schmidt [Schmidt, 2002] propõe que os sistemas de *middleware* sejam decompostos em quatro camadas: infra-estrutura de *host* (*host-infrastructure*), distribuição (*distribution*), serviços comuns (*common-services*) e serviços de domínio (*domain-services*). A Figura 2.4 ilustra a organização destas camadas.

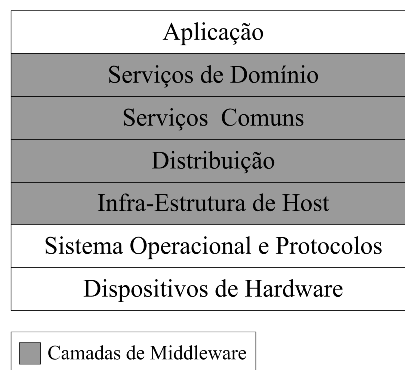


Figura 2.4: Camadas de *middleware* (adaptado de [Sadjadi e Mckinley, 2003]).

A camada de infra-estrutura de *host* fornece uma API de alto nível que encapsula a heterogeneidade de dispositivos de *hardware*, sistema operacional e alguns protocolos de rede. Já a camada de distribuição fornece um alto nível de abstração para o desenvolvimento de aplicações, permitindo que engenheiros de *software* construam aplicações distribuídas de maneira similar aos programas não distribuídos. CORBA [OMG Website], DCOM [Chung et al., 1997] e Java RMI [RMI Website] são exemplos de sistemas de *middleware* focados nesta camada. A camada de serviços comuns provê funcionalidades específicas, tais como tolerância a falhas, segurança, persistência, escalonamento em tempo real e gerenciamento de transações. Os serviços de alto nível providos por esta camada podem ser utilizados por diferentes tipos de aplicações. Por fim, a camada de serviços de domínio fornece abstrações para uma classe particular de aplicações de um domínio específico (e.g., controle de processamento distribuído e processamento de radar).

Entretanto, um dos desafios no desenvolvimento de sistemas de *middleware*, em geral, é conceber como integrá-los às aplicações de uma maneira eficiente e com um custo aceitável para o engenheiro de *software*. De acordo com *Vaysse* [Vaysse et al., 2005], integrar sistemas de *middleware* e aplicações requer uma grande quantidade de código que, geralmente, é genérico e disseminado. Em seu trabalho, *Vaysse* propõe a utilização da POA como uma alternativa para

fazer composição de aplicações usando sistemas de *middleware*.

2.4 Alguns Problemas e Soluções em Computação Ubíqua

Nesta seção são apresentadas soluções para alguns problemas em computação ubíqua que se relacionam com este trabalho de dissertação. Os problemas apresentados nesta seção foram tratados por pesquisadores no escopo dos projetos Gaia, One.Word, Aura e Context ToolKit que, de maneira geral, buscam facilitar o desenvolvimento de *software* ubíquo e automatizar o gerenciamento de recursos e serviços nestes ambientes.

O projeto Gaia[4] é desenvolvido na Universidade de *Illinois* em *Urbana-Champaign* e procura criar um ambiente computacional genérico capaz de mapear ambientes físicos (e.g., casas, escolas e hospitais) e seus dispositivos ubíquos em um ambiente computacional homogêneo, denominado espaço ativo. O objetivo principal das pesquisas realizadas no projeto Gaia é criar um sistema operacional intermediário capaz de gerenciar todos os recursos de um espaço ativo, da mesma forma que os recursos computacionais são geridos pelos sistemas operacionais tradicionais [Román et al, 2002]. Para atingir este objetivo, foi criado um sistema operacional de *middleware* baseado em componentes que executa em diferentes sistemas operacionais (e.g., *Windows2000*, *WindowsCE* e *Solaris*) e utiliza CORBA como plataforma de comunicação. A idéia desta infra-estrutura é deixar transparente para as aplicações a maneira como o *hardware* é acessado dentro de um espaço ativo. O sistema operacional de *middleware* do Gaia originou-se do sistema operacional distribuído 2K [Kon et al, 2000b] que buscou resolver problemas de heterogeneidade de *hardware* e *software* através de sistemas *middleware* flexíveis, portáteis e dinâmicos.

Já o projeto One.World[5], que é um subprojeto do projeto Portolano[6] desenvolvido na Universidade de *Washington*, procura resolver três problemas de um ambiente ubíquo: modularização, dinamicidade e heterogeneidade [Grimm et al, 2001b][Grimm et al, 2004]. O primeiro problema está relacionado com a maneira como os dados e operações são modularizados em objetos, segundo os pesquisadores do One.Word, o forte acoplamento entre dados e código imposto pela modularização dos objetos dificultam a escalabilidade. Para resolver este problema, dados e código são mantidos separados e uma nova abstração é criada para fazer o agrupamento dos dois preservando o isolamento. Já o segundo problema está relacionado com a dinamicidade e a intermitência dos dispositivos e serviços dos ambientes ubíquos. Para resolver

[4] http://gaia.cs.uiuc.edu
[5] http://one.cs.washington.edu
[6] http://portolano.cs.washington.edu

este problema os pesquisadores do projeto One.Word afirmam que os desenvolvedores devem criar sistemas que sejam sensíveis às variações contextuais e não simplesmente as ignorem. Por fim, o terceiro problema está relacionado com a heterogeneidade de *hardware* e *software* do ambiente ubíquo. Para resolver este problema, os pesquisadores do One.Word propõem a criação de uma API padrão para o desenvolvimento de aplicações e a geração de um código objeto comum para executar sobre diferentes plataformas de *software* (e.g., *Windows*, *Unix* e *PalmOS*) e *hardware* (e.g., *Intel*, *Sparc* e *Motorola*) [Grimm et al, 2001a].

O projeto Aura[7] da Universidade *Carnegie Mellon* foi desenvolvido com base em situações e cenários de interação de usuários em ambientes ubíquos. Segundo *Garlan* em [Garlan et al, 2002], Aura é um *framework* que tenta lidar com a mobilidade de usuários em ambientes ubíquos envolvendo comunicação sem fio, computadores portáteis e espaços físicos pequenos. Para lidar com a mobilidade dos usuários, Aura propõe maximizar o uso dos recursos computacionais disponíveis e minimizar a distração do usuário, permitindo que este mantenha sua atenção na execução de suas atividades. Segundo os pesquisadores do projeto Aura, a maior fonte de distração do usuário é a necessidade de gerenciar seus recursos computacionais (e.g., telefones celulares e PDAs) nos ambientes para os quais ele se desloca.

Aura possui quatro tipos de componentes: *Task Manager*, *Context Observer*, *Environment Manager* e *Suppliers*. O *Task Manager* é uma descrição dos serviços independente de plataforma que tem como objetivo permitir que usuários movam-se de um ambiente para o outro sem se preocupar com a qualidade de serviço, com o gerenciamento de serviços e as mudanças de contexto. Já o *Context Observer*, provê informação de contexto físico e notifica o *Task Manager* e o *Environment Manager* sobre eventos contextuais relevantes. O *Environment Manager* exerce a função de *gateway* para o ambiente, ele conhece quais recursos estão disponíveis para serem oferecidos em conjunto com os serviços. Por fim, o *Suppliers* provêem serviços que compõem as requisições dos usuários, tais como, edição de texto e acesso à vídeo.

O projeto Context Toolkit[8] é liderado pelo professor *Anind K. Dey* inicialmente na Universidade de *Berkeley*, Califórnia e atualmente na Universidade *Carnegie Mellon*. Não é propriamente um projeto de computação ubíqua, entretanto, seu principal objetivo é criar um conjunto de ferramentas para facilitar o desenvolvimento de aplicações sensíveis ao contexto, classe de aplicações que compõe ambientes ubíquos [Dey et al, 1999][Dey, 2000]. Para isso, o Context Toolkit provê um *framework* para o desenvolvimento e execução de aplicações sensíveis ao contexto. O *framework* possui como principais elementos os *Widgets*, os *Aggregators* e os

[7] http://www.cs.cmu.edu/~aura

Interpreters, para prover às aplicações transparência na aquisição e manipulação das informações contextuais.

Widgets fazem o interfaceamento entre o usuário e o ambiente e são responsáveis por encapsular as informações de uma determinada forma de contexto. Já os *Aggregators* são responsáveis por coletar as informações de contexto relacionadas a uma entidade física (e.g., pessoa e dispositivo móvel). O *Aggregator* recebe e agrupa os dados fornecidos por todos os *Widgets* e provêem informações relevantes para uma determinada entidade. Por fim, *Interpreters* são responsáveis por transformar uma forma de contexto em outra. Por exemplo, obter o número telefônico a partir do nome de um usuário ou ainda como determinar se uma aula esta sendo ministrada a partir da quantidade de pessoas presentes e da direção em que cada uma das pessoas estiver olhando.

2.5 Conclusões

Neste capítulo foi apresentada uma visão geral sobre a computação ubíqua abordando seus princípios, principais desafios (no contexto desta dissertação) e soluções para alguns de seus problemas. Ainda neste capítulo foram introduzidos conceitos sobre contexto e aplicações sensíveis ao contexto, bem como foram enumeradas as abordagens utilizadas no desenvolvimento de *software* adaptativo.

As principais dificuldades encontradas no desenvolvimento de *software* para ambientes ubíquos derivam de características típicas destes ambientes como heterogeneidade e dinamicidade. Além disso, requisitos como sensibilidade ao contexto demandam a criação de *softwares* cada vez mais autoadaptáveis e autogerenciáveis. Nesta vertente foram apresentadas as abordagens mais utilizadas para o desenvolvimento de *software* adaptativo e, dentre elas, a que mais despertou interesse dentro deste trabalho de dissertação foi a de sistemas de *middleware*. Isto se deve ao fato de que sistemas de *middleware*, além de resolverem problemas de heterogeneidade e disponibilizar uma camada de abstração que permite a inserção de comportamento adaptativo, possibilitam a incorporação de todas as outras abordagens apresentadas.

O capítulo seguinte conceitua, classifica e apresenta alguns sistemas de *middleware* adaptativos existentes que utilizam técnicas de adaptação dinâmica para facilitar o desenvolvimento de *software* adaptativo.

---

[8] http://www.cs.berkeley.edu/~dey/context.html

Capítulo 3

Sistemas de *Middleware* Adaptativos

Este capítulo define e classifica *middleware* adaptativo apresentando alguns modelos existentes. A seção 3.1 introduz o conceito de *middleware* adaptativo e descreve abordagens para classificá-los. A Seção 3.2 discorre sobre alguns sistemas de *middleware* adaptativos existentes, apresentando ao final uma comparação entre estes. Por fim, na Seção 3.3 são apresentadas as principais conclusões e algumas considerações sobre o capítulo.

3.1 Definição e Classificação

Sistemas de *middleware* adaptativos permitem que aplicações distribuídas possam modificar seu comportamento em resposta a mudanças de requisitos ou variações no ambiente de execução. De acordo com *Sadjadi* e *Mckinley* [Sadjadi e Mckinley, 2003] os sistemas operacionais também podem oferecer serviços adaptativos para as aplicações, entretanto, estes serviços por serem, na maioria das vezes, de baixo nível comprometem a portabilidade. Um *middleware* adaptativo, por outro lado, pode explorar os recursos de um sistema operacional fornecendo uma abstração de alto nível que garanta a portabilidade de serviços para as aplicações.

Sadjadi e *Mckinley* [Sadjadi e Mckinley, 2003] propõem que além dos critérios definidos por *Schmidt* [Schmidt, 2002], descritos na Seção 2.3.2.4, os sistemas de *middleware* adaptativos devem ser classificados de acordo com o tipo de adaptação que eles fornecem. A Figura 3.1 ilustra a classificação de sistemas de *middleware* adaptativos, proposta por *Sadjadi* e *Mckinley*, onde cada tipo de adaptação é mapeado em uma das fases do tempo de vida da aplicação. Desta forma, se o sistema de *middleware* permitir adaptação apenas em tempo de compilação ou inicialização, ele é denominado *middleware* estático. Por outro lado, se o *middleware* permitir adaptação em tempo de execução ele é denominado *middleware* dinâmico.

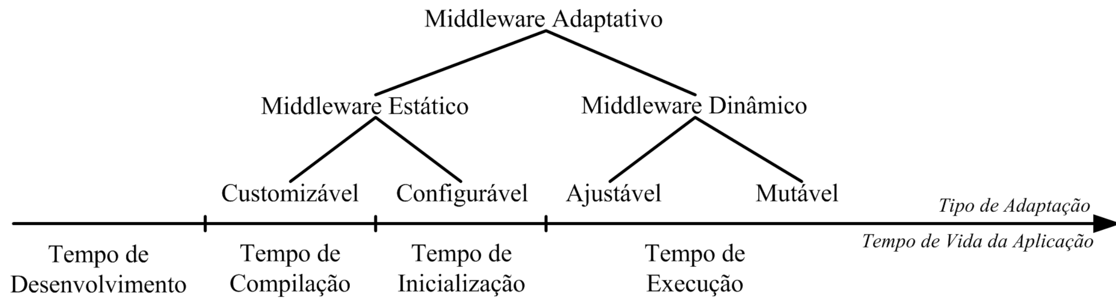


Figura 3.1: Classificação por tipo de adaptação (adaptado de [Sadjadi e Mckinley, 2003]).

Os sistemas de *middleware* estáticos se dividem em dois subtipos (Figura 3.1): customizáveis e configuráveis. O *middleware* customizável permite fazer a adaptação em tempo de compilação ou ligação (*link*), de modo que o engenheiro de *software* pode gerar versões (adaptadas) customizadas do *middleware*. Cada versão customizada é gerada em função das mudanças de requisitos funcionais e do ambiente de execução alvo. Na realidade este tipo de *middleware* pode ser chamado de adaptável. Por outro lado, o *middleware* configurável permite fazer adaptação em tempo de inicialização. Neste modelo o administrador inicializa o *middleware* adicionando parâmetros de configuração que fazem com que o *middleware* se ajuste para atender novos requisitos ou variações no ambiente de execução.

Os sistemas de *middleware* dinâmicos se dividem em dois subtipos (Figura 3.1): ajustáveis e mutáveis. O *middleware* ajustável permite fazer adaptação após o tempo de inicialização da aplicação, mas antes que a aplicação seja realmente usada, possibilitando ao administrador fazer pequenos ajustes em resposta a variações no ambiente de execução que ocorrem depois que a aplicação é inicializada. O *middleware* mutável é o tipo mais poderoso de *middleware* adaptativo, pois permite fazer adaptação em tempo de execução. O *middleware* mutável se adapta dinamicamente ao contexto de execução sem a intervenção do administrador. A principal diferença entre sistemas de *middleware* ajustáveis e mutáveis é que as adaptações que ocorrem nos sistemas de *middleware* ajustáveis não alteram seus núcleos, ao passo que em sistemas de *middleware* mutáveis até mesmo o núcleo do *middleware* pode ser reconfigurado em resposta a eventuais redefinições de requisitos e variações no ambiente de execução.

Outra forma de categorizar sistemas de *middleware* adaptativos é fazer a classificação levando em consideração o domínio de aplicação que eles suportam. De acordo com os autores de [Mckinley et al., 2004] e [Sadjadi e Mckinley, 2003], a maioria dos projetos de sistemas de *middleware* adaptativos suportam um dos seguintes domínios de aplicação (Figura 3.2): aplicação orientada a QoS, aplicação confiável e aplicação embarcada.

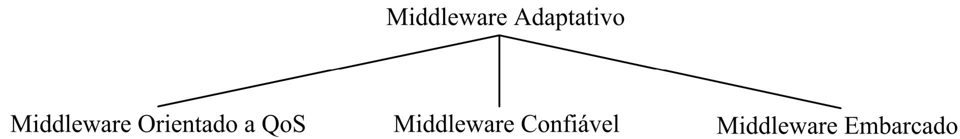


Figura 3.2: Classificação por domínio de aplicação (adaptado de [Sadjadi e Mckinley, 2003]).

O *middleware* orientado a QoS (*Quality of Service*) suporta aplicações de tempo-real e aplicações multimídia (e.g., vídeo conferência e voz sobre IP) que necessitam de requisitos de QoS. Já o *middleware* confiável dá suporte a aplicações distribuídas críticas que requerem precisão de operações, como controle de comandos militares e aplicações médicas. Diferentemente dos outros dois, o *middleware* embarcado dá suporte à aplicações que tem que executar em ambientes com baixo poder computacional.

Outra visão de *middleware* adaptativo é descrita por Costa [Costa, 2004], a qual sugere que um sistema de *middleware* adaptativo deve prover adaptação de duas formas: por invocação, onde apenas a chamada remota alvo sente os efeitos da adaptação; e global, onde todas as chamadas remotas sentem os efeitos da adaptação.

A adaptação por invocação permite que clientes que estejam executando requisições remotas simultaneamente possam ser atendidos pelo mesmo servidor de maneira distinta. Por exemplo, se dois clientes com largura de banda diferente estiverem executando requisições remotas simultâneas o servidor poderia comprimir os dados do cliente que possui menor largura de banda para que este não espere muito tempo para receber a resposta da requisição.

Já a adaptação global permite que servidores remotos modifiquem seu comportamento independentemente da invocação remota que esteja sendo executada. Por exemplo, um servidor poderia abandonar sua política de atendimento de requisições de uma *thread* por requisição para usar um *pool threads* em função de uma sobrecarga de requisições.

3.2 Exemplos

Nesta seção são apresentados os sistemas de *middleware* adaptativos que serviram de modelo para implementação de partes do *middleware* adaptativo proposto nesta dissertação. Nas seções subseqüentes são descritos os sistemas de *middleware* DynamicTAO, UIC, FlexiNet e Open ORB, classificados por *Sadjadi* e *Mckinley* [Sadjadi e Mckinley, 2003], e MoCA e RME, classificados durante esta pesquisa seguindo o mesmo modelo de classificação de *Sadjadi* e *Mckinley* descrito na Seção 3.1.

3.2.1 DynamicTAO

O DynamicTAO [Kon et al., 2000a] é uma plataforma de *middleware* reflexiva implementada como uma extensão de TAO [Schmidt e Cleeland, 1999]. Embora TAO seja uma plataforma portável e flexível, ela não permite reconfigurações dinâmicas. Uma vez configurada, a plataforma TAO não permite que novos algoritmos e componentes sejam modificados ou substituídos enquanto aplicações estiverem sendo executadas. DynamicTAO, por outro lado, suporta reconfigurações dinâmicas ao mesmo tempo em que assegura que o estado da plataforma e das aplicações distribuídas permanece consistente.

DynamicTAO é uma plataforma reflexiva porque é capaz de realizar auto-inspeção e auto-modificação em sua estrutura interna. Isso é possível porque o sistema mantém uma representação de seu próprio estado interno e de quais dependências existem entre seus componentes. Tal fato possibilita que componentes possam ser substituídos sem a necessidade de reiniciar a execução das aplicações que executam sobre ele.

Por ser uma extensão da plataforma TAO, DynamicTAO faz uso de seus componentes que se encontram agrupados em um repositório persistente. Estes componentes são agrupados em categorias que representam os diferentes aspectos de um sistema de *middleware*. Uma vez que um componente é implantado no repositório persistente, ele pode ser dinamicamente escolhido para fazer parte do *middleware*.

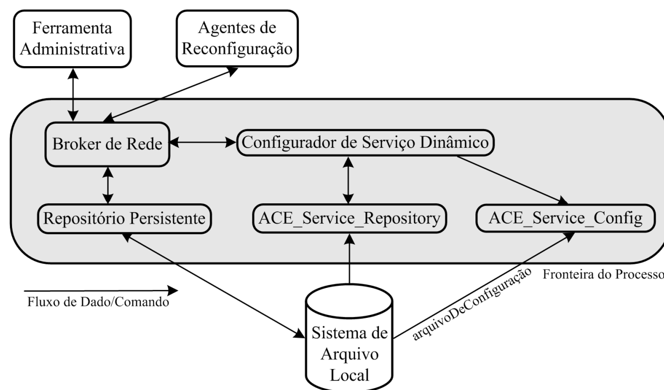


Figura 3.3: Arquitetura de DynamicTAO (Adaptado de [Kon et al., 2000a]).

Os principais elementos da arquitetura de DynamicTAO são mostrados na Figura 3.3 e descritos a seguir:

- **Ferramenta Administrativa**: é uma interface gráfica para gerenciar coleções distribuídas de ORBs (*Object Request Broker*);
- **Agentes de Reconfiguração**: são agentes móveis responsáveis por fazer a

reconfiguração de cada um dos ORBs distribuídos de maneira hierárquica;

- **Repositório Persistente**: armazena as implementações de componentes no sistema de arquivo local em categorias hierárquicas;
- ***Broker* de Rede**: recebe as requisições de reconfiguração e as encaminha para o Configurador de Serviço Dinâmico;
- **Configurador de Serviço Dinâmico**: recebe requisições de reconfiguração do *Broker* de Rede e inicializa um novo serviço/estratégia e o inclui no *ACE_Service_Repository*;
- **ACE_Service_Config**: é responsável por inicializar os arquivos de configuração e gerenciar a ligação dinâmica dos componentes;
- **ACE_Service_Repository**: é responsável por gerenciar a carga das implementações dos componentes.

DynamicTAO fornece ao engenheiro de *software* dois níveis diferentes de reconfiguração. No primeiro nível, os componentes que podem ser alterados estão associados às instâncias de entidades denominadas configuradores de componente (*ComponentConfigurator*). Configuradores são componentes que mantêm um grafo de dependências entre os componentes do *middleware*, e entre componentes do *middleware* e a aplicação. Quando é necessário alterar um dos componentes da plataforma, seu configurador é examinado, de modo a garantir a consistência do sistema. Já no segundo nível, DynamicTAO utiliza componentes denominados interceptadores para possibilitar que desenvolvedores de aplicações tenham como alterar aspectos da plataforma do *middleware*. Interceptadores são elementos de *software* que podem ser inseridos em pontos particulares do *middleware*, como por exemplo, entre o *stub* e a camada de transporte, com o intuito de realizar algum processamento previamente definido pelo engenheiro de *software*.

3.2.2 UIC

UIC (*Universally Interoperable Core*) [Román et al., 2001] não é propriamente um *middleware*, mas uma infra-estrutura formada por componentes abstratos inter-relacionados. UIC permite que componentes concretos sejam combinados estática ou dinamicamente para construir um *middleware* específico. Por ser baseado em componentes, UIC permite que diferentes características de sistemas de *middleware* possam ser configuradas (e.g., protocolo de rede e de transporte, estabelecimento de conexões e semântica da invocação remota).

UIC faz uso de reflexividade para possibilitar a construção de *middlewares* dinamicamente configuráveis. A arquitetura de UIC destina-se ao ambiente heterogêneo e dinâmico da

computação ubíqua. UIC tenta resolver problemas relacionados com heterogeneidade de dispositivos, dinamicidade do ambiente e limitação de recursos.

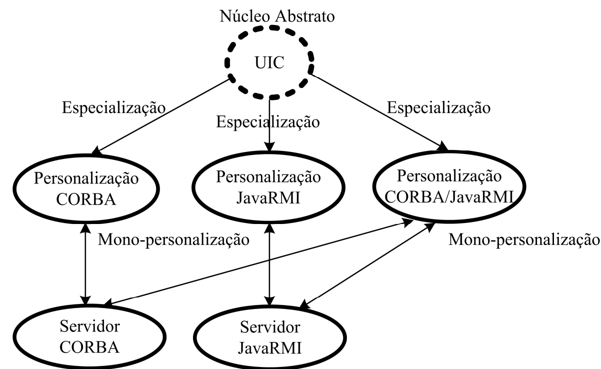


Figura 3.4: Personalizações de UIC (Adaptado de [Román et al., 2001]).

Cada instância de *middleware* derivada de UIC é denominada de personalização (Figura 3.4). Uma personalização pode ser configurada como cliente (i.e., envia pedidos e recebe respostas), servidora (i.e., recebe pedidos e envia respostas) ou *middleware* híbrido (i.e., envia e recebe pedidos e envia e recebe respostas). Uma instância de UIC pode ainda ser classificada como mono-personalizada, ou multi-personalizada. Uma plataforma mono-personalizada é capaz de interagir com somente um tipo de *middleware*, ao passo que uma plataforma multi-personalizada é capaz de interagir com vários sistemas de *middleware* diferentes. Uma instância multi-personalizada não é equivalente a um conjunto de instâncias mono-personalizadas. Uma instância multi-personalizada pode interagir com diversas plataformas de *middleware* sem a interferência da aplicação. Já em uma coleção de instâncias mono-personalizadas, cabe a aplicação escolher qual personalização usar a cada interação.

3.2.3 FlexiNet

O FlexiNet [Hayton et al., 1998] é um *middleware* reflexivo desenvolvido em Java, que dá às aplicações o poder de controlar as decisões do *middleware* em tempo de execução. As aplicações podem controlar protocolos de comunicação e serviços como segurança, replicação ou mobilidade. FlexiNet permite ao engenheiro de *software* escolher quais componentes farão parte do *middleware*, que dessa forma, é composto apenas pelos módulos necessários para um determinado domínio de aplicação.

Para prover reconfiguração dinâmica, FlexiNet dispõe de uma pilha (Figura 3.5) de protocolos dinamicamente configurável em um nível denominado meta. Esta pilha reflexiva é composta por meta-objetos que se utilizam de recursos de reflexão, disponibilizados pela linguagem Java, para fazer transformações em uma invocação remota. Cada meta-objeto da pilha

pode ser visto como um componente independente, que recebe requisições de níveis superiores, as quais são tratadas e repassadas para os níveis inferiores. O protocolo da pilha manipula a chamada tratando de requisitos não funcionais como replicação, centralização e protocolos de chamada de procedimento remoto.

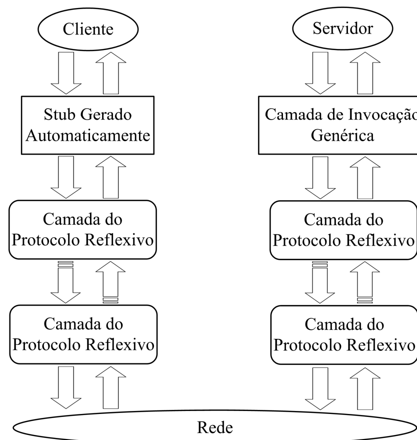


Figura 3.5: Pilha de protocolos reflexivos de FlexiNet (Adaptado de [Hayton et al., 1998]).

Um *stub*, no lado cliente, transforma a chamada em uma invocação genérica, mas fortemente tipada, que é enviada ao topo da pilha. Os *stubs* são gerados dinamicamente, sob demanda, a partir da definição das interfaces dos servidores remotos, utilizando os recursos de *runtime linking* de Java. A partir do topo da pilha, a chamada é manipulada por cada meta-objeto que compõe a pilha e então é enviada através da rede para o servidor.

No servidor, a chamada sofre um processo inverso ao ocorrido no lado cliente. Em seguida, a requisição é enviada ao componente da pilha chamado Camada de Invocação Genérica, que recebe o nome da interface do objeto servidor e os parâmetros para então realizar a invocação através de reflexão.

FlexiNet leva em consideração a noção de seção, que possibilita manter o estado entre um determinado número de chamadas de um cliente a um mesmo servidor. Podem ser mantidas informações que dizem respeito a uma conexão ou informações para manutenção de uma comunicação segura (e.g., chave criptográfica).

O sistema FlexiNet faz uso do conceito de cluster para prover, de maneira transparente, serviços de persistência, transação, re-alocação e migração. Além disso, FlexiNet pode determinar que protocolo é mais apropriado para uma dada comunicação. Uma vez que sejam utilizados *binders* é possível realizar configuração dinâmica por invocação.

3.2.4 Open ORB

O Open ORB [Blair et al., 1998] e [Blair et al., 2001] é um ORB CORBA que se utiliza dos princípios de desenvolvimento baseado em componentes e reflexão computacional para prover reconfiguração dinâmica. Open ORB é capaz de se adaptar em resposta a variações no ambiente de execução. Ele é formado por um conjunto de componentes dinamicamente configuráveis através de meta-computação. Open ORB faz uso de reflexão para inspecionar e modificar a estrutura e comportamento de seus componentes.

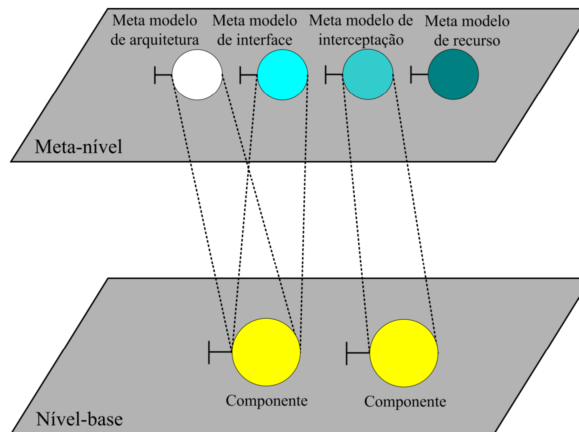


Figura 3.6: Modelo de meta-espaço do Open ORB (Adaptado de [Blair et al., 2001]).

Open ORB possui uma estruturação de meta-espaço, dividido em quatro modelos complementares, cada um se referenciando a diferentes aspectos estruturais ou comportamentais do *middleware*. Os aspectos estruturais do meta-nível são representados pelos meta-modelos de interface e arquitetura, já os aspectos comportamentais, são representados pelos meta-modelos de interceptação e de recursos. A Figura 3.6 ilustra cada um dos quatro meta-modelos que são descritos a seguir:

- O meta-modelo de interface possibilita a inspeção da estrutura de um componente permitindo a identificação de sua interface;
- O meta-modelo de arquitetura dá acesso à implementação de um componente através do grafo de componentes, dos quais ele é composto, e de um conjunto de restrições arquiteturais. O grafo de componentes é formado por um conjunto de componentes conectados através de *bindings*. Os *bindings* podem ser de dois tipos: *local bindings*, ligações entre as interfaces de componentes em um único espaço de endereçamento, e *distributed bindings*, ligações entre as interfaces de componentes em diferentes espaços de endereçamento. O meta-modelo de arquitetura permite a modificação deste grafo em tempo de execução;

- O meta-modelo de interceptação permite a inserção dinâmica de interceptores, que manipulam a chamada de métodos de interfaces inserindo pré-processamento e pós-processamento. Interceptores são úteis para introduzir comportamentos não-funcionais relacionados a aspectos como segurança e controle de concorrência;
- O meta-modelo de recursos, por sua vez, provê acesso aos recursos do *middleware*, como memória, *threads* e *buffers*. Há um meta-modelo de recursos por espaço de endereçamento, que dá acesso a um conjunto de componentes responsáveis pelo gerenciamento de recursos, capazes de inspecionar e adaptar as características destes recursos.

Os primeiros protótipos do Open ORB foram desenvolvidos em *Python*, posteriormente, foi desenvolvido um protótipo em C++ com o objetivo de otimizar o desempenho do *middleware*.

A adaptação de Open ORB sobre componentes é global, de maneira que a configuração dinâmica de um componente afeta qualquer chamada subseqüente, visto que os modelos se referem à estruturação e ao comportamento dos componentes. Portanto, Open ORB não fornece suporte apropriado à adaptação de mensagens individuais.

3.2.5 MoCA

MoCA (*Mobile Collaboration Architecture*) [Sacramento et al., 2004] e [Viterbo Filho et al., 2006] é um sistema de *middleware* que oferece suporte ao desenvolvimento de aplicações distribuídas sensíveis ao contexto que envolvem dispositivos móveis interconectados através de redes sem fio infra-estruturadas (IEEE 802.11b/g). Os serviços disponibilizados pelo MoCA provêem meios para coletar, armazenar e processar informações de contexto computacional (e.g., o estado dos recursos dos dispositivos e da rede sem fio) obtidas diretamente dos dispositivos móveis. Além disso, MoCA engloba um conjunto de APIs para o desenvolvimento de aplicações que interagem com esses serviços como consumidores de informações de contexto.

Os elementos que compõe a arquitetura do MoCA são ilustrados na Figura 3.7 e descritos a seguir:

- ***Context Information Service*** (CIS): é um serviço distribuído que recebe e processa dados de contexto obtidos de dispositivos móveis por meio dos seus Monitores. Cada servidor CIS processa consultas diretas sobre variáveis de contexto específicas de dispositivos móveis. Além disso, servidores CIS utilizam a *Event-Based Communication Interface* para aceitar subscrições no formato de expressões lógicas sobre variáveis de contexto, para serem notificadas quando tais expressões forem avaliadas como verdadeiras. Um dos clientes deste serviço é o *Location Inference Service* (LIS), que consulta

periodicamente os servidores CIS sobre os conjuntos de intensidade de sinal medidos em todos os dispositivos monitorados;

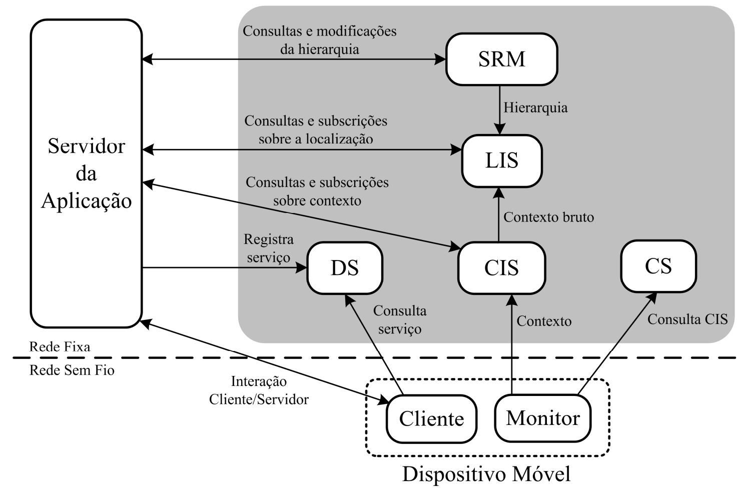


Figura 3.7: Arquitetura típica de aplicações MoCA (Adaptado de [Viterbo Filho et al., 2006]).

- ***Location Inference Service*** (LIS): é um serviço responsável por inferir e disponibilizar a localização simbólica de dispositivos móveis em áreas cobertas por Pontos de Acesso de redes IEEE 802.11. O LIS utiliza a intensidade de sinais coletados pelo CIS de todos os dispositivos móveis para inferir a localização de cada dispositivo móvel;

- ***Symbolic Region Manager*** (SRM): é um serviço que permite estabelecer uma relação entre as regiões atômicas definidas pelo LIS, descrevendo uma hierarquia onde regiões podem ser subordinadas ou estar contidas em outras regiões;

- ***Configuration Service*** (CS): é um serviço responsável por armazenar e manter informações de configurações de todos os dispositivos móveis. As informações de configuração são armazenadas em tabelas *hash* onde cada entrada na tabela (indexada pelo endereço MAC do dispositivo) guarda os endereços do servidor CIS e de um *Discovery Service*, e a periodicidade com a qual o Monitor deverá enviar informações do dispositivo para o CIS;

- ***Discovery Service*** (DS): é encarregado de guardar informações como nome, propriedades e endereços de qualquer aplicação ou serviço registrado no *middleware* MoCA, para que sejam descobertos por seus clientes automaticamente.

MoCA não se preocupa com o atendimento a requisitos não funcionais como qualidade de serviço, pois este não é o seu foco, no entanto, ele fornece um *framework*, denominado *ProxyFramework*, que possibilita que determinadas ações sejam executadas de acordo com o contexto corrente do dispositivo móvel cliente. As ações podem ser de dois tipos: de adaptação

de mensagens (*adapters*) e de mudança de estado (*listeners*). Ações do tipo *adapter* são acionadas no momento do envio de uma mensagem a um cliente, dependendo de seu contexto corrente. Já as ações do tipo *listeners* reagem às mudanças no estado de clientes, no momento em que elas ocorrem.

3.2.6 RME

Arcademis [Pereira et al., 2006] é um *framework* utilizado para facilitar o desenvolvimento de sistemas de *middleware* baseados em objetos distribuídos que utilizam a invocação remota de métodos como meio de comunicação. Sistemas de *middleware* derivados de Arcademis podem ser customizados de diferentes formas, porém todos eles possuem um conjunto comum de classes e interfaces que definem a estrutura geral de um sistema baseado em invocação remota de métodos. Dois destes componentes comuns a qualquer instância de Arcademis são *stubs* e *skeletons*. O `Stub` é responsável por transmitir os parâmetros de uma invocação remota para o objeto remoto alvo, bem como, receber o resultado (caso exista) da invocação. Já o `Skeleton` é responsável por receber invocações remotas e encaminhá-las para o objeto que irá efetivamente processá-las. Além disso, o `Skeleton` é responsável também por enviar o resultado (caso exista) da requisição remota para o cliente que a disparou.

Além de *stubs* e *skeletons*, um sistema de *middleware* derivado de Arcademis utiliza diversos outros módulos que visam garantir a comunicação entre objetos remotos e aplicações clientes. Os principais módulos de Arcademis são ilustrados na Figura 3.8. Alguns módulos representam componentes individuais (e.g., `Stub`, `Skeleton` e `Scheduler`), enquanto outros representam conjunto de componentes (e.g., `Protocolo do Middleware` e `Protocolo de Serialização`) que interagem para fornecer funcionalidades especificas para o engenheiro de *software*. As linhas entre módulos representam colaborações entre componentes.

Observando a Figura 3.8, invocações remotas não são emitidas diretamente por componentes do tipo `Stub`, mas por meio de componentes do tipo `Invoker`. Do lado da aplicação servidora, o componente análogo ao `Invoker`, que é responsável por receber invocações da rede e enviá-las para o objeto remoto, é denominado `Dispatcher`. Chamadas remotas podem ser ordenadas de acordo com sua prioridade, o que é feito por um componente denominado `Scheduler`. A camada de rede propriamente dita é representada por componentes que compõem o `Protocolo do Middleware`, o `Protocolo de Serialização` e o `Protocolo de Transporte`. Para que o cliente possa estabelecer

conexões, ele utiliza um componente de nome `Connector`, ao passo que do lado da aplicação servidora, conexões são recebidas por um componente conhecido como `Acceptor`. A interface entre `Invokers` e a camada de rede é feita por dois componentes: `RequestSender` e `ResponseReceiver`. Já a interface entre componentes do tipo `Dispatcher` e a rede é feita pelos componentes `RequestReceiver` e `ResponseSender`. Finalmente, `Activator` é o componente responsável por inicializar objetos distribuídos e torná-los aptos a receberem invocações remotas.

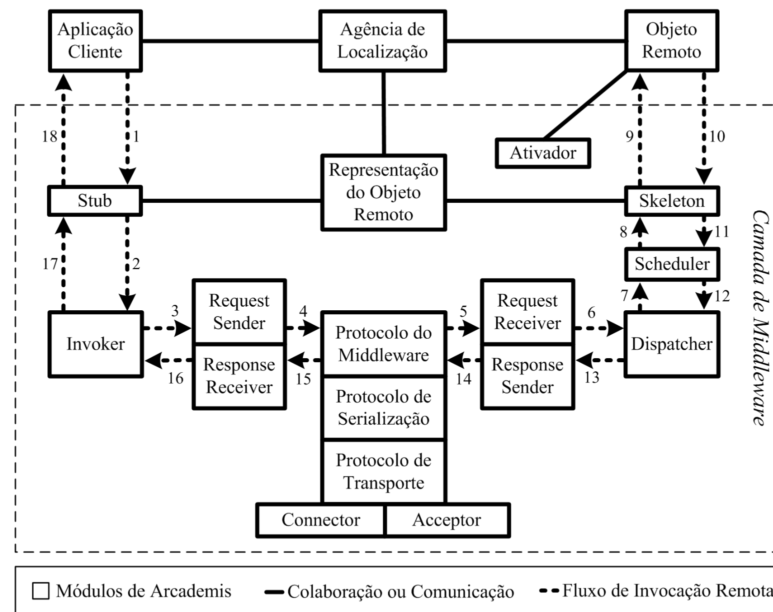


Figura 3.8: Principais módulos de Arcademis (Adaptado de [Pereira, 2006]).

O RME (*Remote Method Invocation for* J2ME) é um sistema de *middleware* orientado a objetos, que permite a invocação remota de métodos segundo uma sintaxe semelhante à de Java RMI [Pereira et al., 2006]. RME foi derivado de Arcademis, sendo assim uma instância do mesmo. Existem duas versões de RME implementadas. Uma delas destina-se à plataforma J2SE da linguagem Java, e permite que aplicações distribuídas usufruam das características e serviços providos por esta plataforma. A outra versão foi desenvolvida para ser utilizada na configuração CLDC da plataforma J2ME de Java e contém somente as funcionalidades necessárias a aplicações clientes, isto é, fornece às aplicações distribuídas a capacidade de invocar métodos sobre objetos remotos.

Assim como Java RMI, RME apresenta uma arquitetura orientada a serviços [Baresi et al., 2003]. Em RME, fornecedores de serviços são representados pela implementação dos objetos remotos. A agência de localização é representada por um serviço de resolução de nomes, no qual objetos remotos podem registrar-se e aplicações clientes podem fazer consultas em busca de

determinados nomes. Por fim, toda aplicação capaz de utilizar o diretório de nomes para obter informações sobre um objeto remoto e solicitar a execução de métodos remotos sobre esse objeto é denominado cliente.

RME permite que aplicações desenvolvidas sobre a plataforma J2ME realizem invocações remotas de métodos, porém tais aplicações não podem fornecer serviços, possuindo apenas as funcionalidades necessárias a programas clientes. Fornecedores de serviços, em RME, são executados sobre a plataforma J2SE porque geralmente programas servidores demandam maior quantidade de recursos que, geralmente, é um requisito não encontrado em dispositivos móveis.

Com o intuito de facilitar a tarefa de reconfiguração, a maior parte dos componentes de Arcademis, e conseqüentemente de RME, são criados a partir de fábricas de objetos, seguindo os padrões de projeto *Abstract Factory* e *Factory Method* [Gamma et al., 1994]. O acesso a cada uma das fábricas de objetos que compõem Arcademis se dá via uma estrutura denominada *broker*, a qual é representada pela classe ORB (Figura 3.9), cuja implementação segue o padrão de projeto *Singleton* [Gamma et al., 1994]. Todas as aplicações distribuídas que fazem uso de sistemas de *middleware* derivados de Arcademis devem fornecer uma configuração para o *broker* antes de iniciarem sua execução. A configuração do *broker* consiste na definição de quais fábricas farão parte dele, o que, de certa forma, determina que instância de *middleware* é gerada. Uma vez definida a configuração do *broker*, ela não mais poderá ser modificada durante a execução da aplicação. Para a configuração do *broker*, Arcademis define a interface `Configurator`.

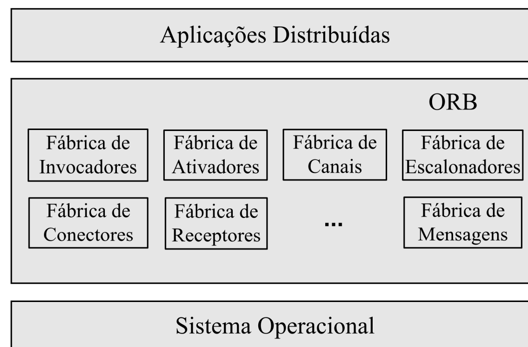


Figura 3.9: RME/ORB e algumas fábricas de objetos (Adaptado de [Pereira, 2006]).

A classe ORB possui dois estados bem definidos: aberto e fechado para configuração. O estado inicial é sempre aberto. No entanto, uma vez ativado o estado fechado, esta situação não pode mais ser modificada. Caso uma tentativa de reconfiguração aconteça quando o estado fechado estiver ativo, uma exceção do tipo `ReconfigurationException` é disparada.

Contudo, a utilização de fábricas para a criação de objetos torna a derivação de novos sistemas de *middleware* a partir de sistemas antigos muito simples. Bastando para isso alterar as

fábricas, responsáveis pela criação dos componentes cuja semântica se deseja alterar.

3.2.7 Análise Comparativa

Sadjadi e *Mckinley* classificam em [Sadjadi e Mckinley, 2003] DynamicTAO e FlexiNet como sendo sistemas de *middleware* adaptativos dinâmicos ajustáveis. Já personalizações de UIC são classificadas como sendo sistemas de *middleware* híbridos, pois provêem adaptação em tempo de compilação e em tempo de execução, podendo, então, serem sistemas de *middleware* adaptativos estáticos customizáveis e dinâmicos ajustáveis. No entanto, o *middleware* Open ORB foi classificado por *Sadjadi* e *Mckinley* como *middleware* adaptativo dinâmico mutável. Como mencionado anteriormente, os sistemas de *middleware* MoCA e RME foram avaliados durante esta pesquisa seguindo a classificação definida por *Sadjadi* e *Mckinley*.

O primeiro *middleware* avaliado foi o MoCA, cujo principal objetivo é prover uma arquitetura distribuída que facilite o desenvolvimento de aplicações colaborativas sensíveis ao contexto. O MoCA não apresenta nenhum mecanismo que permita fazer reconfiguração dinâmica de seus componentes internos. Entretanto, MoCA permite que através do *ProxyFramework* desenvolvedores criem adaptadores de mensagens para adaptar o conteúdo acessado por uma aplicação em função do contexto corrente do dispositivo móvel. Contudo, a instanciação do *framework* e a criação dos adaptadores são feitas em tempo de projeto. Desta forma, o MoCA pode ser classificado como um *middleware* adaptativo estático customizável, uma vez que a instanciação do *framework* e adição de adaptadores se dá em tempo de projeto, e estático configurável, pois ele usa informações de contexto para fazer a adaptação das mensagens.

Já o RME tem como objetivo principal prover uma infra-estrutura de comunicação para ambientes móveis distribuídos. Sua capacidade de reconfiguração está intimamente relacionada com a maneira com que seu *broker*, representado pela classe ORB, é configurado. Desta forma, uma aplicação pode utilizar um configurador definido em tempo de projeto ou selecionar um configurador em tempo de *startup* através, por exemplo, de um arquivo de configuração. Assim, RME pode ser classificado como um *middleware* adaptativo estático customizável ou configurável.

A Tabela 3.1 apresenta um resumo comparativo dos sistemas de *middleware* adaptativos apresentados neste capítulo com relação ao tipo de adaptação e ao domínio de aplicação suportada por cada um deles.

Tabela 3.1: Quadro comparativo.

Domínio de Aplicação	**Tipo de Adaptação**			
	Estática		**Dinâmica**	
	Customizável	**Configurável**	**Ajustável**	**Mutável**
Orientado a QoS	-	-	DynamicTAO e FlexiNet	Open ORB

Confiável	-	-	-	-
Embarcado	UIC*, MoCA* e RME*	MoCA* e RME*	UIC*	-

(*)*Middleware* de adaptação híbrida.

É importante observar que parte dos sistemas de *middleware* apresentados está focada na heterogeneidade de *software* e *hardware*; parte na reconfiguração dinâmica para atendimento a redefinição de requisitos; e parte no suporte ao desenvolvimento de aplicações sensíveis ao contexto. Sendo que nenhum deles se preocupa com o atendimento simultâneo às três questões.

Neste cenário, um *middleware* adaptativo para o ambiente ubíquo deve ser capaz de lidar com a heterogeneidade e ajustar seu comportamento em função de possíveis redefinições de requisitos e possíveis variações no contexto de execução. Além disso, um *middleware* adaptativo ubíquo deve prover mecanismos que permitam capturar, representar e tratar informações contextuais e dar suporte ao desenvolvimento de aplicações ubíquas sensíveis ao contexto.

Entretanto, é importante salientar que quanto mais autonomia possui um *middleware* mais complexo ele se torna. Todavia, sabe-se que sistemas mais complexos realizam mais operações que sistemas estáticos equivalentes. Por este motivo, dinamismo e flexibilidade devem ser balanceados de tal forma que o desempenho de um *middleware* adaptativo não seja comprometido.

3.3 Conclusões

Neste capítulo foram apresentadas definição, classificação e alguns modelos de sistemas de *middleware* adaptativos existentes, além de uma análise comparativa entre os mesmos. O principal objetivo deste capítulo foi introduzir uma classificação para sistemas de *middleware* adaptativos e mostrar como as abordagens para fazer adaptação dinâmica, descritas no Capítulo 2, foram incorporadas a sistemas de *middleware* para dar suporte ao desenvolvimento de *softwares* adaptativos.

O capítulo seguinte apresenta AdptiveRME, um *middleware* adaptativo para ambientes móveis e ubíquos, e AspectCompose, um processo de composição de *software* orientado a aspectos. AdaptiveRME foi concebido para lidar com heterogeneidade, adaptação dinâmica e suporte ao desenvolvimento de aplicações móveis e ubíquas sensíveis ao contexto. Já AspectCompose tem como objetivo diminuir o acoplamento gerado pelo uso de AdaptiveRME pelas aplicações.

Capítulo 4

AdaptiveRME e AspectCompose

Este capítulo apresenta as principais contribuições desta dissertação. No início da Seção 4.1 um visão geral de AdaptiveRME e detalhes de projeto e implementação de cada módulo de sua arquitetura são descritos. Posteriormente, uma descrição detalhada do processo AspectCompose é fornecida na Seção 4.2. Por fim, na Seção 4.3 um resumo das principais conclusões deste capítulo é mostrado.

4.1 AdaptiveRME: Visão Geral e Arquitetura

Como mensionado na seção 3.2.7, é importante salientar que um *middleware* adaptativo para o ambiente ubíquo deve ser capaz de lidar com a heterogeneidade e ajustar seu comportamento em função de possíveis redefinições de requisitos e variações no contexto de execução. Bem como prover mecanismos que permitam capturar, representar e tratar informações contextuais e dar suporte ao desenvolvimento de aplicações ubíquas sensíveis ao contexto.

Neste senário, esta dissertação propõe AdaptiveRME, que é um *middleware* adaptativo dinâmico implementado como extensão de RME para prover um Serviço de Invocação Remota de Métodos Sensível ao Contexto (SIRMSC), o qual é responsável por reconfigurar dinamicamente o *middleware* durante uma invocação remota, e um Serviço de Notificação de Contexto (SNC), o qual fornece uma infra-estrutura para a publicação e acesso à serviços contextuais. AdaptiveRME possui uma arquitetura orientada a serviços seguindo o modelo de RME. Tanto em RME quanto em AdaptiveRME serviços podem ser acessados através de invocações remotas de método sobre objetos distribuídos.

Serviços publicados sobre AdaptiveRME podem ser de dois tipos: serviços funcionais, responsáveis por compor a lógica de negócio da aplicação (e.g., serviço de impressão, serviço de acesso a internet e serviço de persistência) e serviços de contexto, responsáveis por notificar as aplicações sobre o estado ou condição de uma ou mais entidades de contexto (e.g., serviço de localização). Uma entidade de contexto é toda e qualquer entidade cujo comportamento, estado, propriedades e atributos são relevantes para o sistema. São exemplos de tipos de entidades de

contexto pessoas, dispositivos computacionais, eletroeletrônicos em geral e o próprio ambiente físico.

Embora RME disponibilize uma API reduzida e rotinas otimizadas para lidar com limitações de memória (persistente e volátil) e baixo poder de processamento dos dispositivos móveis, ele não permite reconfigurações dinâmicas, necessárias ao dinamismo dos ambientes móveis e ubíquos. Uma vez configurado, seu *broker* não permite que novos algoritmos e componentes sejam incorporados ou substituídos dinamicamente. Já AdaptiveRME foi concebido para ser capaz de se adaptar dinamicamente ao contexto de execução durante uma invocação remota. Para isso, AdaptiveRME permite que seus componentes sejam reconfigurados, em tempo de execução, em resposta a eventuais variações no contexto. Além disso, AdaptiveRME permite que desenvolvedores utilizem adaptadores de conteúdo para adaptar o conteúdo acessado às capacidades do dispositivo ou às preferências do usuário durante uma invocação remota.

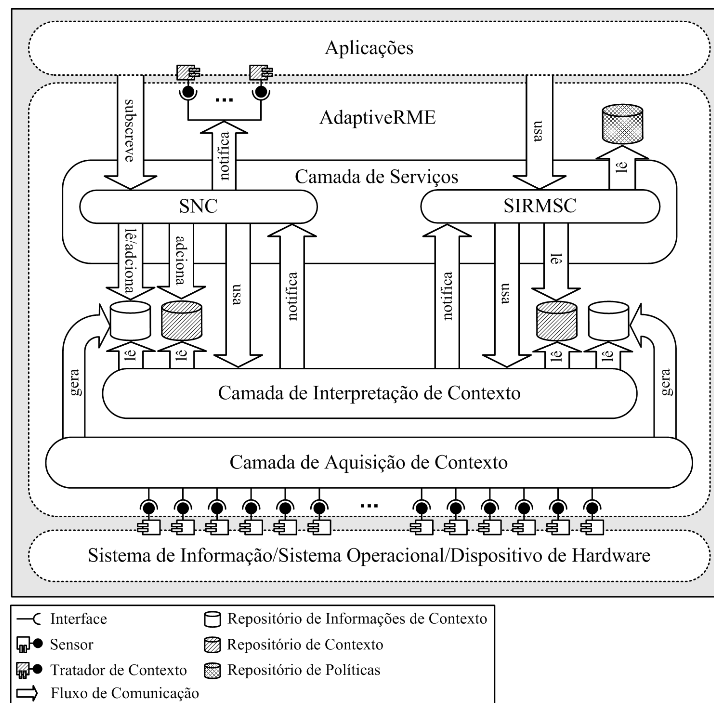


Figura 4.1: Arquitetura de AdaptiveRME.

Para melhor manipular o contexto, AdaptiveRME propõe uma arquitetura dividida em três camadas (Figura 4.1): Camada de Aquisição de Contexto, Camada de Interpretação de Contexto e Camada de Serviços. Cada uma destas camadas é responsável por uma parte do processamento do contexto, desde sua aquisição até o momento em que o *middleware* é reconfigurado e/ou as aplicações são notificadas.

A Camada de Aquisição de Contexto exerce um papel fundamental na arquitetura proposta fornecendo uma abstração sobre como as informações contextuais são capturadas e processadas, para, posteriormente, serem manipuladas pelas camadas superiores. Informações de contexto são informações que dizem respeito a uma entidade de contexto, podendo ser estáticas ou dinâmicas. Uma informação de contexto é dita estática se o seu valor não varia com o decorrer do tempo (e.g., o tamanho e a escala de cores do *display* de um celular). Diferentemente das estáticas, uma informação de contexto dinâmica pode variar seu valor com o decorrer do tempo (e.g., largura de banda, qualidade do sinal, localização e memória de execução livre).

É também responsabilidade da Camada de Aquisição de Contexto gerenciar o ciclo de vida dos sensores. Sensores são componentes de *software* que têm por objetivo coletar informações de contexto e disponibilizá-las num formato que possa ser entendido pelo *middleware*. Informações de contexto podem ser obtidas de diferentes fontes, por exemplo, de um Sistema de Informação, de um Sistema Operacional ou diretamente de algum dispositivo de *hardware*. Sensores podem ser implantados em AdaptiveRME através de arquivos de configuração, que descrevem quais informações contextuais são providas por cada sensor e com qual freqüência eles as atualizam.

Já a Camada de Interpretação de Contexto é responsável por verificar quais contextos estão ativos num dado momento. Em AdaptiveRME, um contexto pode ser descrito por meio de expressões lógicas que relacionam informações de contexto de uma ou mais entidades de contexto. Tais expressões caracterizam um estado que se deseja observar. Um exemplo de expressão lógica que caracteriza um contexto de um dispositivo móvel é: "((SIGNAL_STRENGTH >= 80) && (FREE_MEMORY >= 500))". Neste exemplo "SIGNAL_STRENGTH" representa a porcentagem da qualidade do sinal de comunicação de um dispositivo móvel num dado instante e "FREE_MEMORY" representa a quantidade em *bytes* de memória de execução livre no dispositivo. Os símbolos ">=" e "&&" representam, respectivamente, o sinal de maior ou igual e o operador lógico de conjunção. Um contexto é dito ativo se a avaliação da expressão lógica que o representa resultar num valor lógico verdadeiro. Contextos podem ser criados via API de AdaptiveRME ou a partir de arquivos XML de configuração. Contextos criados utilizando a API são destinados aos serviços de contexto publicados sobre o SNC. Por outro lado, contextos criados a partir de arquivos de configuração destinam-se ao SIRMSC.

Para aferir o contexto, AdaptiveRME faz uso de elementos denominados interpretadores de contexto. Tais elementos avaliam todas as expressões lógicas que representam cada contexto em função das informações de contexto disponíveis. As informações de contexto, por sua vez,

podem ser providas diretamente pela Camada de Aquisição de Contexto ou indiretamente por algum serviço de contexto publicado sobre o SNC. Após avaliar todas as expressões, os interpretadores de contexto fazem a notificação dos contextos ativos para a Camada de Serviços.

Por fim, a Camada de Serviços, é responsável por fazer o interfaceamento entre as aplicações e o *middleware*. Nesta camada se encontram o SIRMSC e o SNC.

No SIRMSC a reconfiguração é realizada por meio de elementos denominados configuradores de componente aplicando o padrão de projeto *Decorator* [Gamma et al., 1995] aos componentes internos do *middleware*, de acordo com estratégias definidas por políticas. Uma política está relacionada com o atendimento ou não a um determinado requisito não-funcional durante uma invocação remota, ao passo que as estratégias representam a maneira como o requisito não-funcional será atendido pelo *middleware* em função do contexto corrente. Por exemplo, uma política de segurança (confidencialidade) pode ser implementada por vários algoritmos criptográficos (estratégias), entretanto, em função do contexto do dispositivo (e.g., memória disponível, nível de bateria e capacidade de processamento) um algoritmo pode ser selecionado em detrimento de outro. Cada política agrupa um conjunto de estratégias que descrevem quais componentes do *middleware* serão decorados e quantos decoradores (*decorators*) irão decorar cada componente. Desta forma, após a reconfiguração cada componente configurado passa a ter novas funcionalidades que permitem ao *middleware* se ajustar dinamicamente ao contexto durante a invocação remota. O SIRMSC permite que várias políticas sejam utilizadas durante uma invocação remota, entretanto, apenas uma estratégia de cada política deve ser selecionada em função do contexto corrente.

Todo serviço de contexto publicado sobre o SNC é responsável por receber subscrições de aplicações na forma de contexto (i.e., expressões lógicas sobre variáveis que representam informações de contexto) e notificá-las quando estes estiverem ativos. Cada contexto subscrito é armazenado no repositório de contexto para ser avaliado pela Camada de Avaliação de Contexto. Para cada subscrição, a aplicação deve associar um ou mais tratadores de contexto que serão notificados quando o contexto subscrito estiver ativo. Tratadores de contexto são elementos responsáveis por realizar algum processamento em função de um dado contexto ativo. Eles são implementados por terceiros e sua lógica de ação depende dos interesses da aplicação para a qual foram projetados.

4.1.1 Camada de Aquisição de Contexto

A Camada de Aquisição de Contexto é composta por dois elementos básicos: sensores e

gerentes de sensores que são explicados com mais detalhes nas sessões subseqüentes.

4.1.1.1 Sensores

Como mencionado anteriormente, um componente sensor é responsável por coletar e processar informações de contexto adquiridas de diferentes fontes (e.g., um Sistema de Informação, um Sistema Operacional ou um dispositivo de *hardware*). Em AdaptiveRME, sensores podem ser implantados por meio de arquivos de configuração. Desta forma, um sensor pode ser adicionado, removido ou substituído por outra implementação mais adequada ao dispositivo no qual o *middleware* será utilizado. A Figura 4.2 ilustra as principais classes envolvidas na criação de componentes sensores.

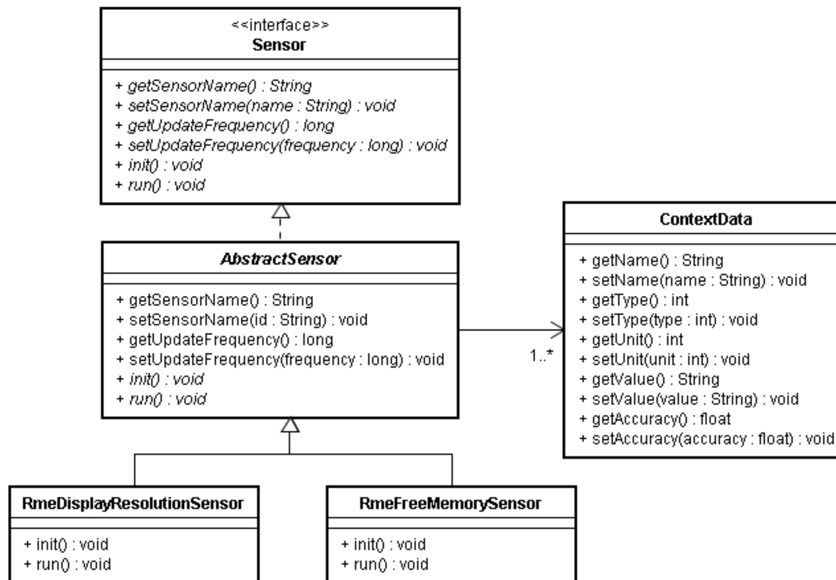


Figura 4.2: Diagrama de classe do Sensor.

Todo componente sensor deve realizar a interface `Sensor`. Entretanto, AdaptiveRME disponibiliza uma implementação abstrata, `AbstractSensor`, para facilitar o desenvolvimento de sensores. A classe `AbstractSensor` realiza a interface `Sensor` e estende a classe `java.lang.Thread` provendo uma implementação para os métodos `getSensorName()`e `setSensorName(String)`, responsáveis por recuperar e atribuir, respectivamente, um nome para o sensor, e os métodos `getUpdateFrequency()` e `setUpdateFrequency(long)`, responsáveis por recuperar e atribuir, respectivamente, o valor que determina com qual freqüência o sensor irá interromper sua execução antes de voltar a coletar novos valores da informação de contexto. Desta forma, para que um desenvolvedor crie um novo sensor basta que ele implemente uma classe que estenda `AbstractSensor` e

codifique nela os métodos `init()` e `run()`.

A Listagem 1 descreve a lógica de funcionamento de um componente sensor. O método `init()` (linha 1) tem como função instanciar os objetos do tipo `ContextData` (responsáveis por encapsular as informações de contexto), preencher seus atributos (e.g., nome da informação de contexto, unidade de medida e grau de pureza) e adicioná-los na sessão. Já o método `run()`, (linha 7) é responsável por capturar as informações de contexto e atribuir seus valores aos objetos instanciados no método `init()`.

```
Sensor
 1  Método Público init() { //Faz a inicialização do sensor.
 2      Instancia um ou mais objetos que encapsula informações de contexto;
 3      Preenche os atributos do(s) objeto(s) instanciado(s);
 4      Adiciona o(s) objeto(s) na sessão;
 5  }
 6
 7  Método Público run() { //Fazer coleta de informações de contexto.
 8      Enquanto (sim) faça { //Laço infinito.
 9          Realiza a coleta da informação contextual;
10          Processa a informação coletada;
11          Atribui o(s) valor(es) coletado(s) ao(s) objeto(s) instanciados em init();
12          Interrompe a execução do sensor por um tempo determinado;
13      }
14  }
```

Listagem 1: Lógica de funcionamento de um componente sensor.

AdaptiveRME disponibiliza junto com sua API duas implementações de componentes sensores 100% portáveis, são elas (Figura 4.2): `RmeMemorySensor` , responsável por capturar a quantidade de memória, livre e total, de um dispositivo móvel, e `RmeDisplayResolutionSensor`, responsável por capturar a altura e a largura da tela de um dispositivo móvel. Cada uma das implementações instancia objetos do tipo `ContextData` e atribui a eles nome, tipo, unidade de medida e grau de pureza com que a informação é coletada, além do valor coletado.

4.1.1.2 Gerente de Sensores

O gerente de sensores é responsável por instanciar, configurar e inicializar cada sensor descrito no arquivo de configuração. A Figura 4.3 ilustra as principais classes envolvidas no processo de gerenciamento dos componentes sensores. A classe `RmeSensorManager`, realização da interface `SensorManager`, gerencia o ciclo de vida dos componentes sensores. Sua implementação segue o padrão de projeto *Singleton*.

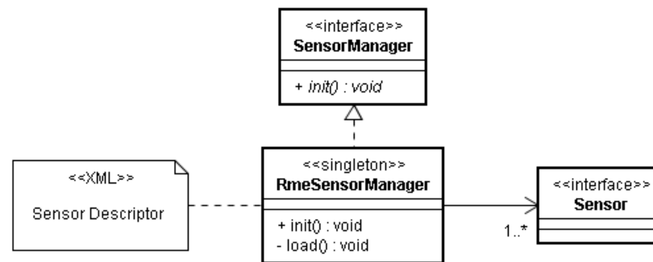


Figura 4.3: Diagrama de classe do Gerente de Sensor.

`RmeSensorManager` é responsável por carregar e interpretar o arquivo `Sensor Descriptor` (Figura 4.4), onde se encontra a descrição de quais componentes sensores irão executar, além de detalhes tais como: o nome do sensor, a freqüência de atualização, a implementação concreta do sensor, e uma lista com os nome das informações de contexto oferecidas.

XML Sensor Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<sensors>
   <sensor active="true|false" name="Nome do Sensor" frequency="Freqüência de Atualização"
       impl="Implementação">
       <context-data-provide>
           <output data="Nome da Informação de Contexto" />
           (...)
       </context-data-provide>
   </sensor>
   (...)
</sensors>
```

Figura 4.4: Arquivo de configuração XML Sensor Descriptor.

Já o método `init()` (linha 1), por sua vez, é responsável por inicializar todos os sensores que foram adicionados na sessão. Entretanto, toda vez que o método `init()` é disparado, ele verifica se os sensores já foram inicializados, em caso positivo, nenhum processamento é realizado.

Gerente de Sensores

```
1  Método Público init() { //Faz a inicialização de todos os sensores.
2      Se (sensoresIniciados = não) então {
3          GerenteSensores.load();
4          Recupera os sensores da sessão;
5          Para (cada um dos sensores recuperados da sessão) faça {
6              Sensor.init();//Inicialização do sensor.
7          }
8          sensoresIniciados ← sim;
9      }
10 }
11
12 Método Privado load(){ //Faz a carga dos sensores.
13     Acessar repositório;
14     Instanciar sensores do repositório de sensores local;
15     Adicionar sensores na sessão;
16 }
```

Listagem 2: Lógica de funcionamento do gerente de sensor.

A Listagem 2 descreve a lógica de execução do gerente de sensores. O método `load()`

(linha 12) é responsável por instanciar todos os sensores descritos no arquivo de configuração `Sensor Descriptor` e adicioná-los na sessão.

4.1.2 Camada de Interpretação de Contexto

A Camada de Interpretação de Contexto tem como principal função avaliar cada contexto, em função das informações de contexto coletadas, e notificar a Camada de Serviços quando um determinado contexto estiver ativo. Os principais elementos desta camada são os interpretadores de contexto e os notificadores de contexto.

4.1.2.1 Interpretadores de Contexto

O interpretador de contexto é responsável por verificar quais contextos estão ativos em um determinado momento, podendo fazer isso de maneira síncrona ou assíncrona. Interpretadores síncronos são usados pelo SIRMSC para avaliar o contexto corrente do dispositivo, antes de efetivamente processar a invocação. Já os tratadores assíncronos são utilizados pelo SNC, uma vez que estes seguem o padrão *Publish and Subscribe* [Buschmann et al., 1996], onde as aplicações fazem subscrições para serem notificadas em um momento posterior. Na Figura 4.5 são ilustradas as principais classes envolvidas no processo de interpretação de contexto.

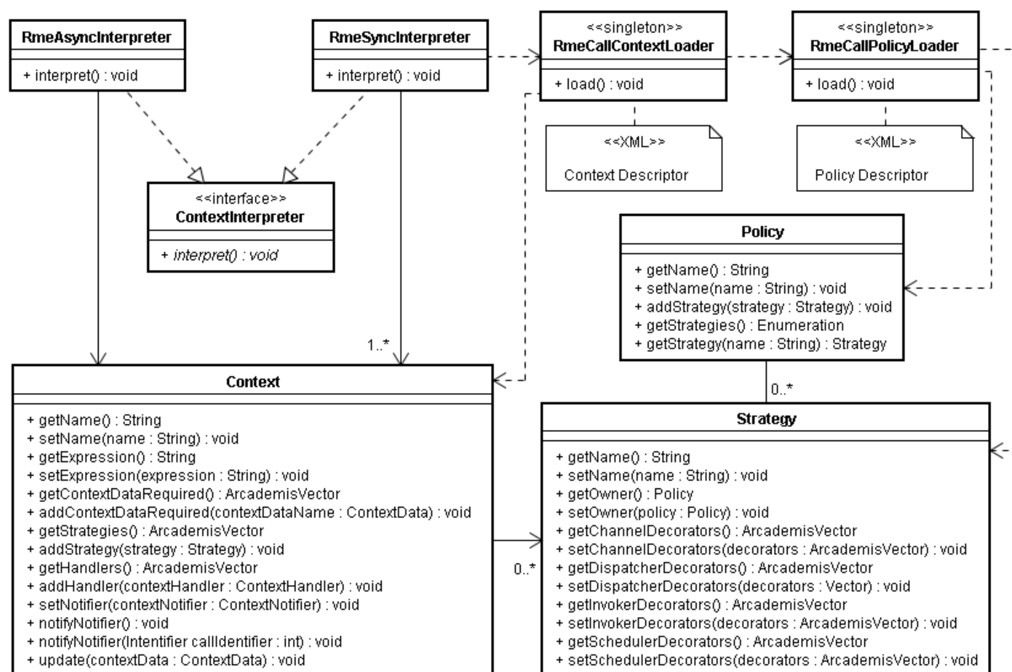


Figura 4.5: Diagrama de classes do Interpretador de Contexto.

A classe `Context` encapsula informações sobre um contexto em particular e guarda

informações como nome do contexto (identificador único), expressão lógica que o representa, informações de contexto requeridas, e referência para os seus tratadores e notificador. As classes `RmeCallContextLoader` e `RmeCallPolicyLoader` são responsáveis, respectivamente, por mapear as informações contidas nos arquivos de configuração `Context Descriptor` e `Policy Descriptor` para uma coleção de instâncias das classes `Context` e `Policy` e depois inseri-las na sessão. A Figura 4.6 ilustra o formato dos arquivos `Context Descriptor` e `Policy Descriptor`. A associação entre contextos e estratégias é feita pelo valor do **ID do Contexto** descritos nos atributos *name* da *tag context* e *context* da *tag strategy*.

XML Context Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<contexts>
   <context active="true|false" name="ID do Contexto" description="...">
       <context-data-required>
           <context-data name="Nome da Informação de Contexto" />
           (...)
       </context-data-required>
       <expression>Expressão de Contexto</expression>
   </context>
   (...)
</contexts>
```

XML Policy Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<policies>
   <policy name="Nome da Política" active="true|false" description="...">
       <strategy name="Nome da Estratégia" description="..." context="ID do Contexto">
           <decorate component="Nome do Componente">
               <decorator impl="Implementação" />
           </decorate>
           (...)
       </strategy>
       (...)
   </policy>
   (...)
</policies>
```

Figura 4.6: Associação entre os arquivos XML Context Descriptor e XML Policy Descriptor.

A classe `RmeSyncInterepreter` é responsável por avaliar o contexto do dispositivo antes da execução de uma chamada remota. A Listagem 3 apresenta de maneira simplificada a execução de um interpretador de contexto síncrono.

Interpretador de Contexto Síncrono

```
1  Método Público interpret() {
2      Recupera os contextos da sessão;
3      Recupera os dados de contexto requeridos da sessão;
4      Faz o match entre os nomes das variáveis e o valor da informação de contexto;
5      Avalia a expressão do contexto;
6      Se (contexto.ativo = sim) então {
7         contexto.notifyNotifier();//Aciona o notificador de contexto.
8      }
9  }
```

Listagem 3: Lógica de funcionamento do interpretador de contexto síncrono.

Já a classe `RmeAsyncInterpreter` é responsável por avaliar cada um dos contextos, subscritos pelas aplicações através do serviço de notificação de contexto, de maneira assíncrona.

Para cada contexto de interesse da aplicação é criada uma *thread* de `RmeAsyncInterpreter` que se encarrega de avaliar o contexto com uma freqüência definida. A Listagem 4 apresenta a seqüência de passos executados por um interpretador de contexto assíncrono.

```
Interpretador de Contexto Assíncrono
 1  Método Público interpret() {
 2      Enquanto (subscrição.ativa=sim) faça {
 3        Seleciona contexto;
 4        Recupera os dados de contexto requeridos da sessão;
 5        Faz o match entre os nomes das variáveis e o valor da informação de contexto;
 6        Avalia a expressão do contexto;
 7        Se (contexto.ativo = sim) então {
 8           contexto.notifyNotifier();//Aciona o notificador de contexto
 9        }
10        Interrompe a execução por um tempo determinado;
11      }
12  }
```

Listagem 4: Lógica de funcionamento do interpretador de contexto assíncrono.

4.1.2.2 Notificadores de Contexto

Notificadores de contexto são responsáveis por notificar a Camada de Serviço quando um contexto estiver ativo. Existem dois tipos de notificadores de contexto: notificadores de serviço de contexto e notificadores de serviço de invocação. Os notificadores de serviço de contexto são encarregados de notificar os serviços de contexto, publicados sobre o SNC, quando um contexto subscrito estiver ativo. Já os notificadores de serviço de invocação, notificam o SIRMSC, para que este selecione as estratégias que deverão ser utilizadas durante a invocação remota. Desta forma, contextos criados através da API do *middleware* recebem notificadores de serviço de contexto, ao passo que contextos criados a partir de arquivos de configuração recebem notificadores de serviço de invocação.

As classes `ContextServiceNotifier` e `InvocationServiceNotifier` (Figura 4.7) são implementações concretas do notificador de serviço de contexto e do notificador de serviço de invocação, respectivamente. Já as classes `ServerStrategyActivator` e `ClientStrategyActivator` são implementações dos ativadores de estratégias do lado servidor e lado cliente do *middleware*, respectivamente.

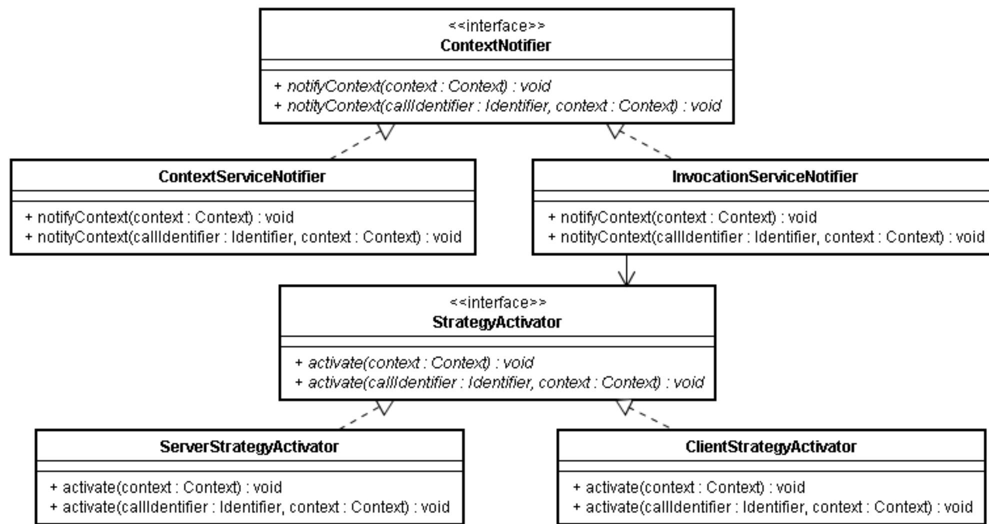


Figura 4.7: Diagrama de classes dos Notificadores de Contexto.

4.1.3 Camada de Serviços

A Camada de Serviços é responsável por prover serviços para as aplicações que utilizam o *middleware*. Os principais elementos que compõem esta camada são: o Serviço Invocação Remota de Métodos Sensível ao Contexto (SIRMSC) e o Serviço de Notificação de Contexto (SNC). Ambos são detalhados nas seções subseqüentes.

4.1.3.1 Serviço de Invocação Remota de Métodos Sensível ao Contexto - SIRMSC

O SIRMSC tem como objetivo reconfigurar o *middleware*, durante uma invocação remota, em função do contexto de execução do dispositivo. Para isso, AdaptiveRME provê abstrações e mecanismos que permitem ao engenheiro de *software* especificar estratégias de reconfiguração e associá-las a determinados contextos de execução do dispositivo. A ação do SIRMSC pode ser dividido em três atividades: sincronização, reconfiguração dinâmica e adaptação de conteúdo. A Figura 4.8 ilustra o fluxo de execução de uma invocação remota sobre o SIRMSC.

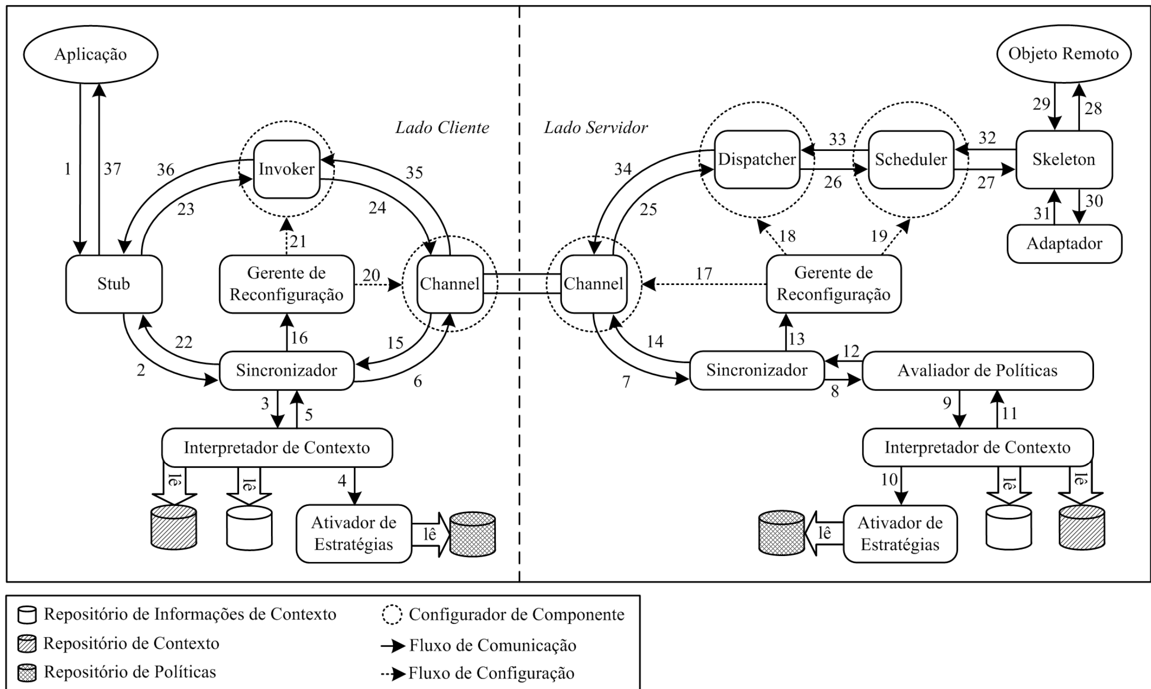


Figura 4.8: Serviço Invocação Remota de Métodos Sensível ao Contexto.

Invocações remotas partem da aplicação e são redirecionadas para o *stub* que, por sua vez, aciona o processo de sincronização. Depois de concluída a sincronização, o gerente de reconfiguração reconfigura os componentes do *middleware* permitindo que a invocação remota seja processada normalmente. Dependendo do método invocado, o *skeleton* aciona os adaptadores de conteúdo passando para eles as informações de contexto enviadas pelo dispositivo móvel necessárias para fazer adaptação do conteúdo acessado.

4.1.3.1.1 Sincronização

Durante uma invocação remota, todas as políticas descritas no repositório de políticas (do lado cliente) são utilizadas, entretanto, cabe ao processo de sincronização garantir que, para cada política utilizada na invocação remota, a mesma estratégia da política seja aplicada tanto no lado cliente quanto no lado servidor.

Acionado pelo *stub*, o sincronizador (cliente) se comunica com o interpretador de contexto que avalia os contextos armazenados no repositório de contexto. Para cada contexto ativo, o interpretador aciona o notificador de serviço de invocação que, por sua vez, notifica os ativadores de estratégias. Ativadores de estratégias são responsáveis por selecionar as estratégias que serão negociadas com o servidor em função dos contextos ativos.

Já no lado servidor, o sincronizador se comunica com o avaliador de políticas que decide se

as estratégias de reconfiguração requeridas pelo cliente serão utilizadas na comunicação. Para cada estratégia solicitada pelo cliente, pode existir algum contexto associado no lado servidor. Caso exista, o avaliador de políticas utiliza o interpretador de contexto para avaliar cada um dos contextos associados para, só então, decidir se a estratégia solicitada pelo cliente será contemplada. Caso a estratégia solicitada pelo cliente não seja contemplada, o próprio avaliador de políticas define qual estratégia será adotada. Contextos avaliados pelo avaliador de políticas podem ser compostos por variáveis de contexto coletadas do ambiente de execução do próprio servidor e/ou por informações de contexto fornecidas pelo cliente durante o processo de sincronização.

A classe `Synchronizer` (Figura 4.9) é responsável por gerenciar o processo de sincronização. Após obter da sessão quais políticas serão utilizadas na invocação remota, `Synchronizer` utiliza o `RmeSyncInterpreter` para avaliar o contexto do dispositivo e definir as estratégias de reconfiguração. No lado servidor, `Synchronizer` utiliza `RmePolicyCallAvaliator` para avaliar as estratégias solicitadas pelo cliente.

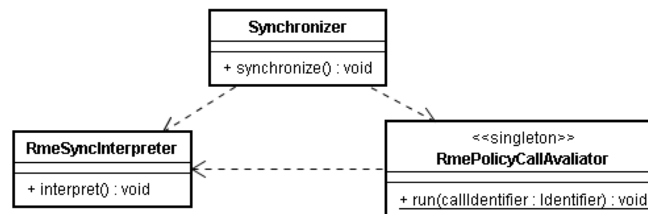


Figura 4.9: Diagrama de classes do Sincronizador.

4.1.3.1.2 Reconfiguração Dinâmica

Após o processo de sincronização, tanto o cliente quanto o servidor estão conscientes de quais estratégias de reconfiguração deverão ser aplicadas ao *middleware* durante a invocação. Neste momento, os configuradores de componente são acionados para decorar os componentes, seguindo as estratégias de reconfiguração estabelecidas pelo processo de sincronização. Cada decorador permite que funcionalidades sejam adicionadas e, posteriormente, removidas dinamicamente de um componente. AdaptiveRME permite que os componentes `Invoker`, `Channel`, `Dispatcher` e `Scheduler`, responsáveis, respectivamente, pela invocação (cliente), manipulação do canal de comunicação (cliente/servidor), pelo despacho de requisições (servidor) e pelo escalonamento das requisições recebidas (servidor) sejam decorados dinamicamente. Para isso, cada um destes componentes possui um decorador abstrato (Figura 4.10).

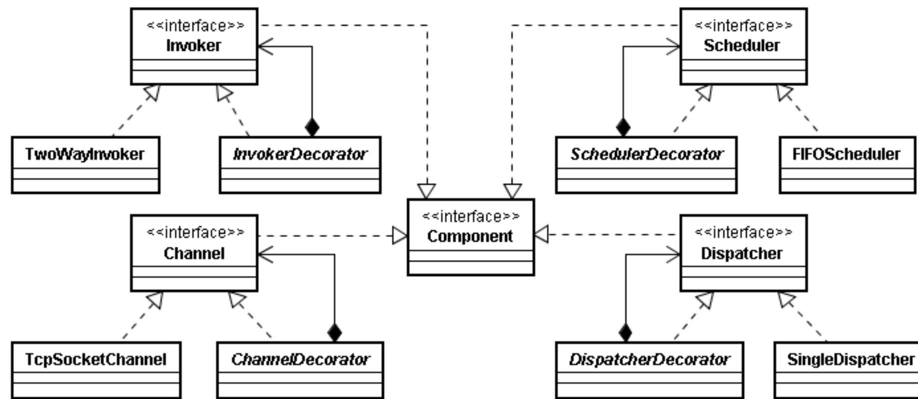


Figura 4.10: Diagrama de classes dos Decoradores de Componentes.

Cada uma das interfaces (i.e., `Invoker`, `Channel`, `Dispatcher` e `Scheduler`) possui uma implementação concreta e um decorador abstrato. Todas elas estendem a interface `Component` que permite que configuradores de componentes ajam sobre suas implementações concretas. A classe `TwoWayInvoker` implementa um invocador que envia requisições e espera o seu retorno de maneira síncrona. A classe `TcpSocketChannel` implementa o canal de comunicação baseado em *sockets* sobre o protocolo TCP/IP. A classe `SingleDispatcher` implementa um *dispatcher monothread*. A classe `FIFOScheduler` implementa o algoritmo FIFO (*First In First Out*), onde a primeira requisição que chega é a primeira a ser atendida.

Para criar um decorador concreto, basta codificar uma classe que estenda o decorador abstrato do componente que se deseja adicionar uma nova funcionalidade, e sobrescrever os métodos cuja funcionalidade se deseja alterar. A Tabela 4.1 apresenta algumas implementações de decoradores concretos disponibilizados junto com a API de AdaptiveRME.

Tabela 4.1: Decoradores concretos.

Nome do Decorador	Componente	Descrição do Decorador
SpeedZipChannelDecorator	Channel	Aplica o nível mais baixo de compressão ZIP nas mensagens
BestZipChannelDecorator	Channel	Aplica o nível mais alto de compressão ZIP nas mensagens
XORChannelDecorator	Channel	Aplica um simples algoritmo XOR nas mensagens
CachedInvokerDecorator	Invoker	Mantém um *cache* de respostas de invocações remotas
ThreadPoolDispatcherDecorator	Dispatcher	Utiliza um *pool* de *threads* para despachar requisições
ThroughputSchedulerDecorator	Scheduler	Escalona requisições em função do *throughput*
FreeMemorySchedulerDecorator	Scheduler	Escalona requisições em função da memória livre do dispositivo

Em AdaptiveRME, configuradores de componentes funcionam como *Adapters* [Gamma et al., 1994] que encapsulam implementações concretas dos componentes e disponibilizam uma interface que permite reconfigurá-los dinamicamente. Quando um módulo do *middleware* requer uma implementação de algum componente, ele a solicita através do *broker*, que por sua vez delega a tarefa à fábrica responsável pela criação do componente requerido. Durante a criação, a fábrica instancia uma implementação do componente e a insere no seu respectivo configurador,

retornando para o módulo que solicitou a criação uma referência para o configurador do componente e não para a implementação do componente de fato. A Figura 4.11 ilustra os configuradores de componentes de AdaptiveRME.

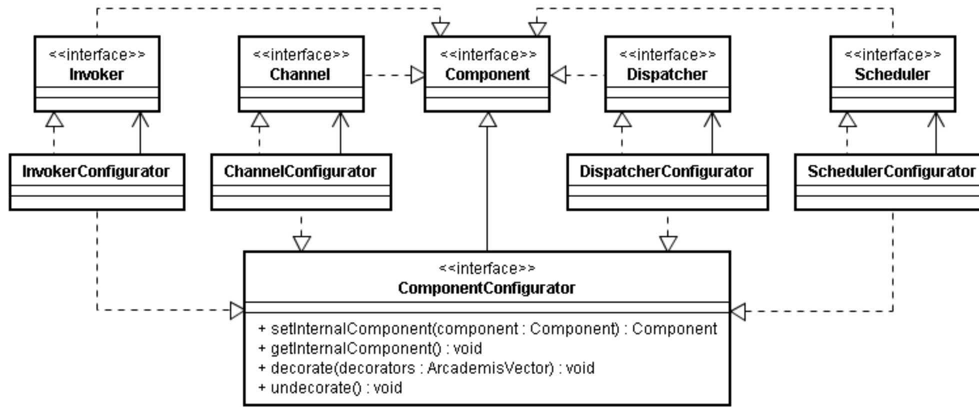


Figura 4.11: Diagrama de classes do Configurador de Componentes.

Cada configurador realiza duas interfaces, a interface `ComponentConfigurator` e a interface do componente que irá configurar. Desta forma, cada configurador de componente possui a assinatura de métodos responsáveis por reconfigurar o componente e os métodos utilizados para acessar os serviços providos pelo componente. Os métodos `setInternalComponent(Component)` e `getInternalComponent()` são responsáveis, respectivamente, por atribuir e obter uma referência para o componente interno. Já os métodos `decorate(ArcademisVector)` e `undecorate()` são responsáveis por decorar e fazer o processo inverso no componente interno.

```
Configurador de Componente
 1  Método Público decorate(ArcademisVector decorators) {
 2      Para (cada um dos decoradores de decorators) faça {
 3        decorador.setInternalComponent(Configurador.internalComp);
 4        Configurador.internalComp ← decorador;
 5      }
 6  }
 7
 8  Método Público undecorate() {
 9      Enquanto (Configurador.internalComp for instância de Decorator) faça {
10        Configurador.internalComp ← Configurador.internalComp.getInternalComponent();
11      }
12  }
```

Listagem 5: Lógica de funcionamento do configurador de componente.

A semântica dos métodos `decorate(ArcademisVector)` e `undecorate()` é descrita na Listagem 5. Os métodos que os configuradores herdam das interfaces Invoker, Channel, Dispatcher e Scheduler são utilizados apenas para redirecionar requisições para o componente interno.

4.1.3.1.3 Adaptação de Conteúdo

A adaptação de conteúdo em AdaptativeRME é feita por elementos denominados adaptadores. Eles são acionados de dentro do *skeleton* (servidor) e utilizam informações de contexto, enviadas pelo dispositivo, para fazer a adaptação. Tais informações podem representar preferências do usuário ou características estáticas e dinâmicas do dispositivo.

Em AdaptiveRME, serviços são, na verdade, métodos de objetos distribuídos, cujo acesso é feito via invocação remota, assim, cabe ao desenvolvedor alterar o código do *skeleton*, do respectivo objeto remoto, para introduzir adaptadores de conteúdo. Em AdaptiveRME, todo *skeleton*, possui trechos de código associados aos métodos remotos do objeto publicado e, nestes trechos, devem ser inseridos os adaptadores necessários para adaptar o conteúdo acessado por aquele método. Adaptadores podem ser implementados seguindo a interface `ContentAdapter` (Figura 4.12).

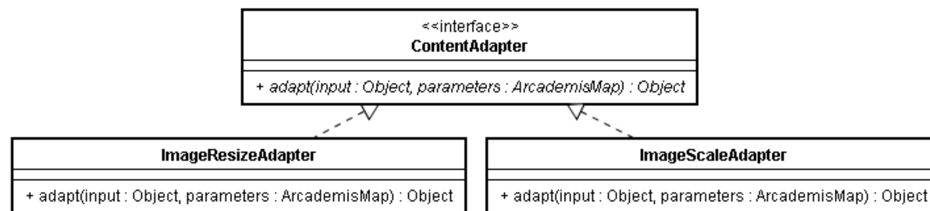


Figura 4.12: Diagrama de classes do Adaptador.

No método `adapt(Object,ArcademisMap),` o argumento `input` representa o conteúdo que se deseja adaptar e `parameters` são as informações de contexto necessárias para fazer a adaptação. AdaptiveRME disponibiliza dois adaptadores: `ImageResizeAdapter` e `ImageScaleAdapter`. `ImageResizeAdapter` busca adequar o tamanho (em *bytes*) de uma imagem à capacidade de memória do dispositivo móvel. Enquanto `ImageScaleAdapter` re-escala uma imagem tentando ajustá-la ao tamanho do *display* do dispositivo móvel.

4.1.3.2 Serviço de Notificação de Contexto - SNC

O SNC tem como principal função permitir a construção de aplicações sensíveis ao contexto. Para isso, o SNC provê uma infra-estrutura que dá suporte à criação e publicação de serviços contextuais. Um serviço de contexto é composto por dois tipos de elementos computacionais: produtores e consumidores. Os elementos produtores são responsáveis por prover as informações de contexto sobre as entidades que se quer observar. Por outro lado, os

consumidores são elementos que registram interesse sobre o estado ou condição destas entidades. Contudo, existem elementos computacionais que podem ser produtores e consumidores ao mesmo tempo.

De uma maneira geral, um serviço de contexto é responsável por avaliar remotamente as expressões contextuais subscritas por aplicações (em função das informações de contexto providas pelos produtores) e notificá-las quando as entidades de contexto monitoradas modificarem seu estado ou comportamento da maneira descrita pelo contexto subscrito. A arquitetura típica de um serviço de contexto é ilustrada na Figura 4.13.

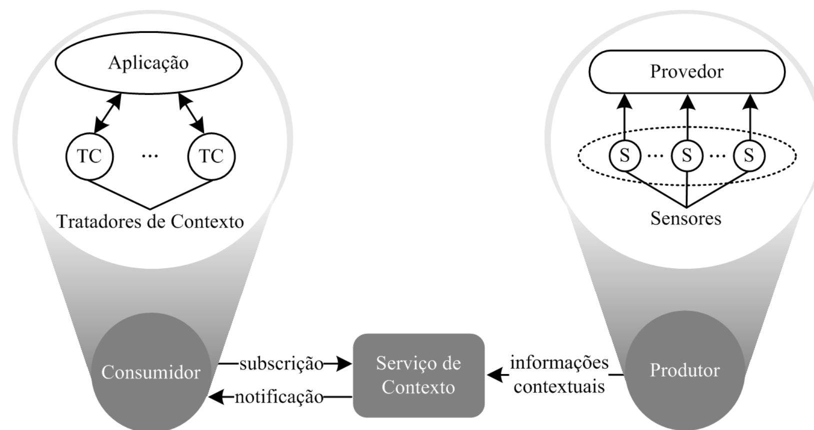


Figura 4.13: Arquitetura de um Serviço de Contexto.

Inicialmente, tanto consumidores quanto produtores localizam o serviço de contexto através da Agência de Localização. Em seguida, os produtores enviam informações de contexto coletadas para o serviço de contexto, que por sua vez as armazena no repositório de informações de contexto. Já os elementos consumidores, representados pelas aplicações que executam em dispositivos móveis, fazem subscrições ao serviço de contexto para que seus tratadores de contexto sejam notificados quando os mesmos estiverem ativos.

4.1.3.2.1 Criando Serviços de Contexto

Para criar um serviço de contexto, o primeiro passo é definir quais entidades de contexto serão monitoradas pelo serviço, bem como quais informações de contexto destas entidades estarão disponíveis para uma eventual subscrição. É importante mencionar que AdaptiveRME não possui um formato rígido para descrição de entidades de contexto. Apenas fornece abstrações que permitem identificar uma entidade de contexto e agrupar suas informações contextuais. Assim, cabe ao engenheiro de *software* definir quais informações contextuais compõem as entidades de contexto monitoradas. A Figura 4.14 ilustra as principais classes de

AdaptiveRME envolvidas na criação de um serviço de contexto.

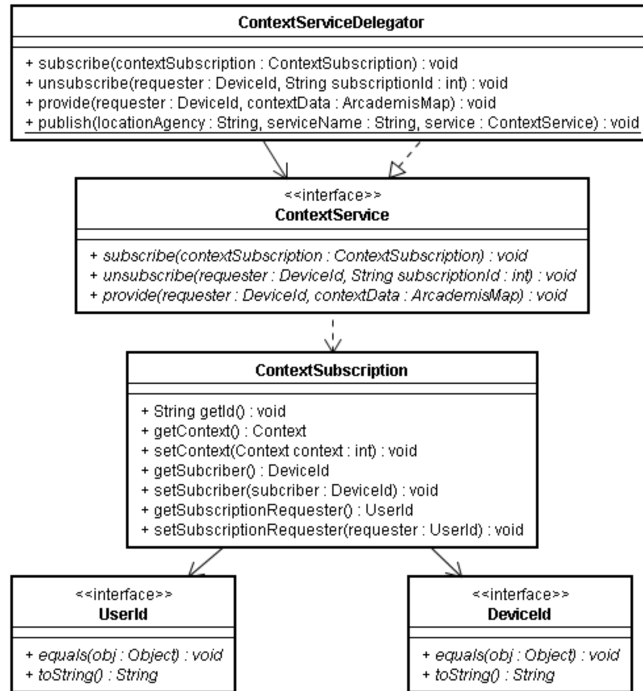


Figura 4.14: Diagrama de classes do Serviço de Contexto.

Todo serviço de contexto em AdaptiveRME deve realizar a interface `ContextService`, que define a assinatura de métodos padrão para a comunicação entre o serviço de contexto e os produtores e consumidores. O método `subscribe(ContextSubscription)` é acessado pelos consumidores passando como argumento uma instância da classe `ContextSubscription` que encapsula o contexto que será subscrito, o identificador do usuário e o identificador do dispositivo que realiza subscrição. Cada usuário e dispositivo possui uma maneira de ser identificado e cabe ao engenheiro de *software* prover uma implementação concreta de `UserId` e `DeviceId`. AdaptiveRME disponibiliza as classes `SimpleUser`, realização da interface `UserId`, e `MACId`, realização da interface `DeviceId`, que encapsulam, respectivamente, o nome de usuário e endereço MAC (*Media Access Control*) do dispositivo móvel.

O método `unsubscribe (DeviceId,String)` é utilizado pelos consumidores para remover uma subscrição. O método `provide (DeviceId,ArcademisMap)` é acessado pelos produtores para enviar informações de contexto da entidade monitorada ao serviço de contexto. É importante observar que todos os métodos requerem a identificação do requisitor. Assim, o serviço de contexto pode utilizar esta informação para autenticar os produtores, autenticar e autorizar consumidores, mapear as informações providas por um

produtor em uma entidade contexto particular e definir níveis de prioridade para consumidores.

Para serem acessados, os serviços funcionais publicados em AdaptiveRME requerem que seus respectivos *stubs* estejam empacotados junto com as aplicações. Desta forma, sempre que um novo serviço for disponibilizado, existe a necessidade de atualizar a aplicação cliente adicionando os *stubs* dos novos serviços para que esta possa utilizá-lo. Entretanto, os serviços de contexto não requerem tal atualização, uma vez que AdaptiveRME provê a classe `ContextServiceDelegator` responsável por delegar requisições a serviços de contexto. Para isso, após instanciar o serviço de contexto, basta publicá-lo através do método `publish(String,String,ContextService)` de `ContextServiceDelegator` fornecendo o endereço da agência de localização em que se deseja publicar o serviço (argumento `agencyLocation`) e um *alias* para o serviço (argumento `serviceName`).

4.1.3.2.2 Produtores de Informação Contextual

As informações contextuais são providas para o serviço de contexto através dos produtores. Produtores podem ser elementos computacionais quaisquer (e.g., PDAs, computadores pessoais, e eletroeletrônicos em geral), capazes se comunicar com o serviço de contexto por meio de alguma infra-estrutura de rede, sem fio ou não. Cada produtor possui um elemento denominado provedor que é responsável por agrupar as informações de contexto, coletadas pelos sensores, e enviá-las para o serviço de contexto associado. Através de arquivos de configuração, o provedor sabe quais informações ele deve enviar para um determinado serviço de contexto e com qual freqüência esse envio deve acontecer. A Figura 4.15 ilustra as principais classes que compõem um provedor.

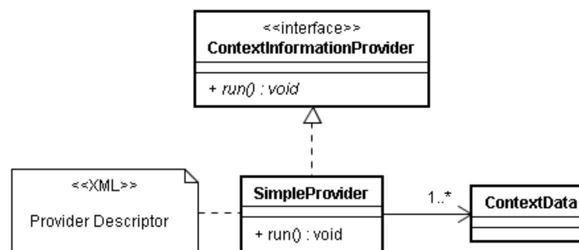


Figura 4.15: Digrama de classes do Produtor.

Em AdaptiveRME, um provedor deve realizar a interface `ContextInformationProvider`. A classe `SimpleProvider` é uma implementação simplificada de um provedor que recupera do repositório de informações de contexto as informações requeridas pelo serviço de contexto associado, conforme descrito no arquivo de

configuração `Provider Descriptor` (Figura 4.16). Entretanto, um provedor pode realizar algum processamento sobre as informações coletadas antes de enviá-las para o serviço de contexto. Por exemplo, novas informações contextuais podem ser derivadas a partir de outras mais primitivas. Provedores podem fornecer informações de contexto para vários serviços de contexto, basta que o arquivo de configuração, `Provider Descriptor`, seja devidamente preenchido com as informações necessárias.

```
XML Provider Descriptor
<?xml version="1.0" encoding="UTF-8"?>
<provider>
   <service name="Nome do Serviço" location="Endereço da Agência de Localização">
      <provide-data frequency="Freqüência de Envio em Milisegundos">
         <context-data name="Nome da Informação de Contexto" />
         (...)
      </provide-data>
   </service>
   (...)
</provider>
```

Figura 4.16: Arquivo de configuração XML Provider Descriptor.

4.1.3.2.3 Consumidores de Serviços de Contexto

Consumidores são elementos capazes de fazer subscrições em serviços de contexto. Em geral, consumidores são representados por aplicações, entretanto, serviços de contexto podem ser acessados por serviços funcionais ou por outros serviços de contexto. Para que um consumidor utilize um serviço de contexto é necessário que ele saiba, previamente, quais entidades de contexto são monitoradas pelo serviço e quais informações de cada entidade estão disponíveis para consulta. Como AdaptiveRME não possui um formato padrão para descrever entidades de contexto, não é possível descobrir tais informações dinamicamente, cabendo ao engenheiro de *software* fazer esta checagem em tempo de projeto.

A Figura 4.17 apresenta as principais classes envolvidas no processo de subscrição de contexto. A classe `ContextSubscriptionFacade` disponibiliza uma fachada que permite aos consumidores realizar subscrições a serviços de contexto. O método `setLocationService(String,String)` é responsável por definir em que agência de localização (argumento `locationAgency`) pode-se obter uma referência para o serviço de contexto requerido (argumento `serviceName`). O método `setRequester(UserId)` é utilizado para associar a subscrição a um determinado usuário. O método `subscriber(String,String)` de `ContextSubscriptionFacade` é responsável por efetivamente realizar a subscrição. Os argumentos `subscriptionId` e `expression` são, respectivamente, os valores de identificação da subscrição e a expressão lógica que representa o contexto.

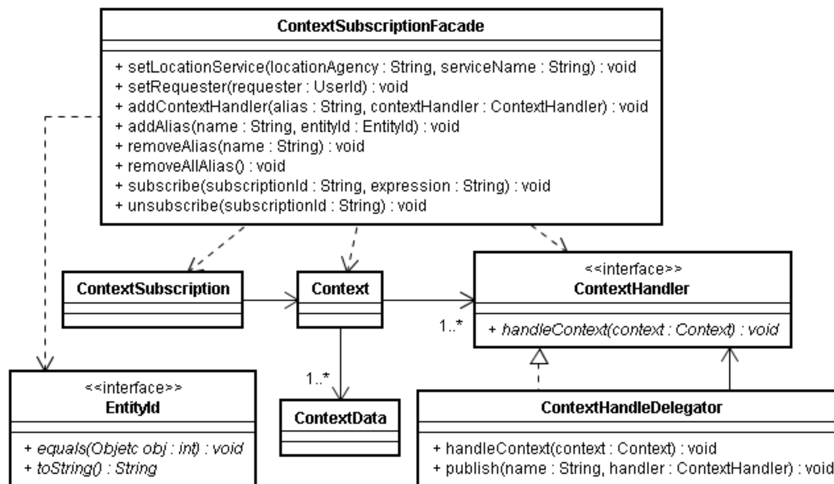


Figura 4.17: Diagrama de classes do Consumidor.

Toda subscrição deve possuir um identificador para que possa ser referenciada no processo de cancelamento de subscrição, feito pelo método `unsubscribe(String)`. A lógica de funcionamento do método `subscribe(String,String)` é descrita na Listagem 6.

Fachada de Subscrição de Contexto

```
 1  Método Público subscribe(String subscriptionId, String expression) {
 2      Faz o match dos aliases de expression com os valores dos EntityId associados;
 3      Cria e povoa um objeto do tipo Context;
 4      Publica os tratadores de contexto no dispositivo móvel;
 5      Associa as referências remotas dos tratadores ao objeto context instanciado;
 6      Cria e um objeto do tipo ContextSubscriber;
 7      Insere o objeto context no objeto ContextSubscriber instanciado;
 8      Insere o objeto do tipo UserId em ContextSubscriber instanciado;
 9      Instancia um objeto do EntityId e o insere no objeto ContextSubscriber instanciado;
10      Obtém uma referencia para o serviço de contexto;
11      Invoca o método remoto subscribe () do serviço de contexto;
12  }
```

Listagem 6: Lógica de funcionamento da Fachada de Subscrição de Contexto.

Como um serviço de contexto pode monitorar mais de uma entidade de contexto, uma expressão que relacione variáveis de contexto de duas ou mais entidades de contexto pode ser escrita da seguinte forma: “((E{1}.VARIAVEL > E{2}.VARIAVEL) && .... (E{n-1}.VARIAVEL == E{n}.VARIAVEL))”, onde E{1}, E{2}, E{n-1} e E{n} são *aliases* para diferentes entidades de contexto. Para fazer o mapeamento dos *aliases*, o método `addAlias(String,EntityId)` deve ser acessado passando como argumento o nome do *alias* (e.g., E{1}, E{2}, E{n-1} ou E{n}) (argumento `name`) e o identificador da entidade que se quer fazer a associação (argumento `entityId`). `EntityId` especifica uma interface para criação de identificadores de entidades, cabendo aos engenheiros de *software* fornecer uma implementação concreta.

Como mencionado anteriormente, para cada contexto subscrito o consumidor deve

associar um ou mais tratadores de contexto. Diferente de RME, AdaptiveRME permite que tratadores de contexto sejam publicados como serviços remotos em dispositivos móveis. Tal funcionalidade permite que quando um contexto subscrito estiver ativo, o notificador de serviço de contexto realize invocações remotas sobre as referências dos tratadores de contextos associados ao contexto ativo.

Tratadores de contexto em AdaptiveRME devem realizar a interface `ContextHandler`. Entretanto, de maneira semelhante aos serviços de contexto, AdaptiveRME disponibiliza a classe `ContextHandlerDelegator` responsável por delegar requisições aos tratadores de contexto. Para isso, basta instanciar um tratador de contexto e publicá-lo através do método `publish(String,ContextHandler)` de `ContextHandlerDelegator` fornecendo um *alias* para o tratador (argumento `name`).

4.2 O Processo AspectCompose

Sistemas de *middleware* são amplamente utilizados por engenheiros de *software* no desenvolvimento de aplicações distribuídas. Isto se justifica pelo fato de que sistemas de *middleware* provêem abstrações sobre primitivas de rede que facilitam o desenvolvimento deste tipo aplicação. Entretanto, eles são, em geral, intrusivos e transversais ao código funcional das aplicações. AdaptiveRME fornece mecanismos que permitem ao engenheiro de *software* publicar e acessar objetos remotos. Contudo, convenções de uso como estender classes internas da API do *middleware*, implementar interfaces remotas e codificar estratégias de serialização proporcionam um alto grau de acoplamento e entrelaçamento entre as aplicações e o *middleware*.

Com o intuito de diminuir o acoplamento e o entrelaçamento ocasionado pelas convenções de uso impostas por AdaptiveRME, este trabalho propõe um processo de composição denominado AspectCompose, que utiliza técnicas de programação orientada a aspectos para compor as aplicações utilizando AdaptiveRME. A Figura 4.18 mostra uma visão geral de AspectCompose.

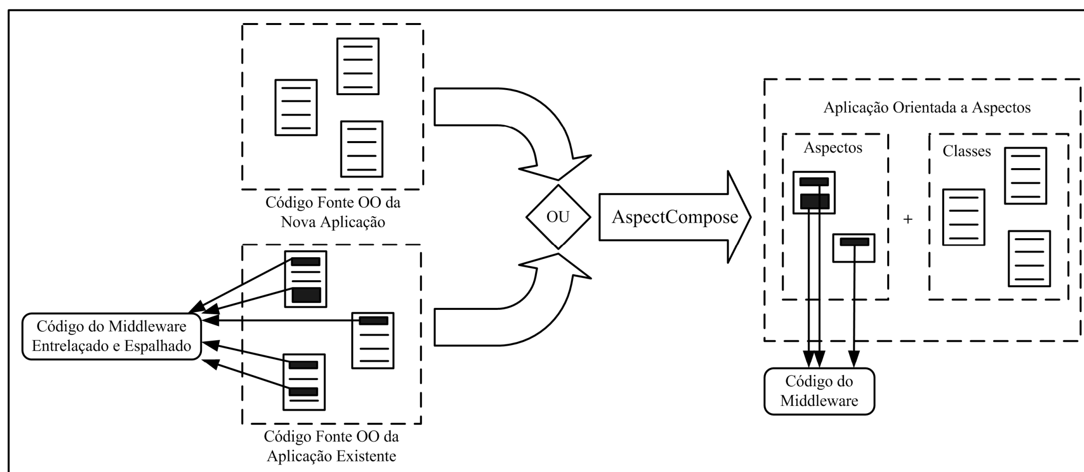


Figura 4.18: Ilustração do processo realizado pelo AspectCompose.

De uma maneira geral, o processo pode ser aplicado sobre uma aplicação existente (que já faz uso do *middleware*) ou sobre uma nova aplicação (que não usa o *middleware*). Caso o processo seja utilizado sobre uma aplicação existente, AspectCompose remove o código transversal, oriundo do *middleware*, para dentro de aspectos. De maneira semelhante, se AspectCompose é usado sobre uma nova aplicação, todo o código que implementa as convenções do *middleware* é codificado dentro de aspectos.

O processo foca em quatro pontos identificados como principais causadores de acoplamento e entrelaçamento: publicação de objetos remotos, serialização de objetos remotos, localização de objetos remotos e configurações iniciais do *middleware*. AspectCompose descreve em uma seqüência passos como compor uma aplicação utilizando AdaptiveRME e técnicas de programação orientada a aspectos. Antes de detalhar os passos do processo, algumas convenções de nomenclatura e pré-condições de codificação serão apresentadas na próxima subseção.

4.2.1 Convenções e Pré-condições

Visando facilitar o entendimento do processo, algumas convenções de termos são adotadas, a seguir:

- Classes que dão origem a objetos remotos, quando instanciadas, são chamadas de classes remotas;
- Interfaces que definem as assinaturas dos métodos que o objeto remoto disponibilizará são chamadas de interfaces remotas;
- Classes que dão origem a objetos que invocam métodos de objetos remotos são chamadas de classes invocadoras;

- Classes que fizerem parte da assinatura de métodos de interfaces remotas são chamadas de classes relacionadas.

As pré-condições definem condições necessárias ao bom funcionamento do processo. Neste tópico são definidas as atividades que devem preceder a aplicação do processo:

- Todas as classes que são ou serão transformadas em classes remotas devem ser identificadas. Em seguida, para cada classe selecionada, deve ser criada uma interface remota, excetuando-se os casos de aplicações existentes onde classes remotas já realizam uma interface remota;
- Após a criação de todas as interfaces remotas, todas as classes invocadoras deverão ser identificadas e os seus relacionamentos com as classes remotas deverão ser substituídos por relacionamentos envolvendo suas respectivas interfaces remotas. Desta forma, onde existir a declaração do tipo "atributo x do tipo ClasseRemota" deve ser substituído por "atributo x do tipo InterfaceRemota". De maneira semelhante, na assinatura de um método, onde existir um argumento ou tipo de retorno do tipo "ClasseRemota" deve ser substituído por "InterfaceRemota". Também deverão ser observados os casos onde houver declarações semelhantes a: "variável local x do tipo ClasseRemota" ou "variável local x do tipo InterfaceRemota". Em ambos os casos, o escopo da variável deve ser aumentado dentro da classe de local para global. Para isso, basta substituir declarações do tipo "variável local x do tipo ClasseRemota" e "variável local x do tipo InterfaceRemota" por declarações do tipo "atributo x do tipo InterfaceRemota".

4.2.2 Passos do Processo

Os passos do processo são apresentados a seguir:

- **Passo 1 (Identificação)**: Nesta etapa são definidos os subpassos que devem ser tomados para possibilitar o início da execução do processo, são eles:
    - **Passo 1.1** – Devem ser identificadas todas as classes remotas;
    - **Passo 1.2** – Devem ser identificadas todas as interfaces remotas;
    - **Passo 1.3** – Devem ser identificadas todas as classes invocadoras;
    - **Passo 1.4** – Devem ser identificadas todas as classes relacionadas.
- **Passo 2 (Publicação)**: Este passo define como o interesse transversal ligado à publicação de objetos remotos deve ser modularizado. Para tanto, são definidos os

seguintes sub-passos:

- **Passo 2.1** – Para cada classe remota e interface remota identificadas nos Passos 1.1 e 1.2, devem ser criados dois aspectos que serão chamados de aspecto de classe remota e aspecto de interface remota, respectivamente. Tais aspectos têm como objetivo adequar o formato estrutural das classes e interfaces remotas as convenções necessárias à publicação de objetos remotos. No caso de aplicações que já usam AdaptiveRME e, portanto, estão sendo evoluídas, as convenções de publicação de objetos remotos encontram-se diluídas pelas suas respectivas classes e interfaces remotas existentes, de onde devem ser removidas e codificadas em seus respectivos aspectos remotos. Em AdaptiveRME, toda classe remota deve estender a classe `RmeRemoteObject` e implementar uma interface remota que especifica o contrato de serviço fornecido pelo objeto remoto. Já uma interface remota deve estender a interface `Remote` de Arcademis e cada método descrito nesta interface deve lançar uma exceção do tipo `ArcademisException`. AdaptiveRME disponibiliza, desde sua versão inicial (RME), um gerador automático de *stubs* e *skeletons* que encapsulam a semântica de invocação e tratamento de chamadas remotas. Desta forma, sistemas baseados em AdaptiveRME devem fazer uso desta ferramenta para gerar *stubs* e *skeletons* de suas classes remotas ao final deste passo;

- **Passo 2.2** – Para cada classe remota, devem ser criados, no respectivo aspecto de classe remota, pontos de junção (*joinpoint*) que interceptem a instanciação de objetos do tipo classe remota. Em seguida, deve ser criado um conjunto de junção (*pointcut*) que agrupe todos os pontos de junção criados no aspecto de classe remota. Este conjunto de junção deve ser associado ao adendo (*advice*) que desvia o fluxo de execução do programa, após a instanciação, e faz a publicação do objeto remoto recém instanciado. Em AdaptiveRME, o seguinte método deve ser invocado para que um objeto remoto seja publicado: `RmeNaming.bind(String, RemoteObject)`, onde o primeiro parâmetro é o *alias* para o objeto remoto e o segundo parâmetro é a instância do objeto a ser publicado. O *alias* é composto pelo endereço da agência de localização (onde o objeto será publicado) e um "apelido" para o objeto remoto. Por exemplo, no *alias* "magneto.great.ufc.br/obj" o fragmento "magneto.great.ufc.br" representa o endereço da agência de localização e "obj" representa o "apelido" do objeto remoto.

- **Passo 3 (Serialização)**: Este passo define como o interesse transversal ligado à serialização de objetos remotos deve ser modularizado. Para tanto é definido o seguinte sub-passo:
  - **Passo 3.1** – Para cada classe relacionada, deve ser criado um aspecto que será chamado de aspecto de serialização. Esse aspecto deve modificar a estrutura da respectiva classe relacionada, inserindo as convenções definidas pelo protocolo de serialização de AdaptiveRME. No caso de aplicações que estão sendo evoluídas, o código relativo às convenções de serialização deve ser removido das classes relacionadas e codificado nos seus respectivos aspectos de serialização. Em AdaptiveRME toda classe relacionada deve implementar a interface `Marshalable` de Arcademis que define os métodos `marshal(Stream)` e `unmarshal(Stream)`, responsáveis, respectivamente, pela serialização de deserialização do objeto.
- **Passo 4 (Localização)**: Este passo define como o interesse transversal ligado à localização de objetos remotos deve ser implementado. Para tanto, são definidos os seguintes sub-passos:
  - **Passo 4.1** – Para cada interface remota deve ser criado um aspecto denominado aspecto de inversão de controle associado;
  - **Passo 4.2** – Devem ser selecionadas todas as classes invocadoras que possuam atributos do tipo interface remota e que façam parte da aplicação cliente. Para cada atributo do tipo interface remota de cada classe invocadora selecionada, deve ser criado um ponto de junção que intercepte a ação de leitura do atributo. Este ponto de junção deve ser codificado dentro do aspecto de inversão de controle, associado ao tipo de interface remota do atributo selecionado;
  - **Passo 4.3** – Deve ser criado um conjunto de junção em cada aspecto de inversão de controle associado que agrupe todos os seus pontos de junção e, em seguida, associe-os ao comportamento transversal, codificado no adendo, que recupera a referência para o objeto remoto antes da ação de leitura ser realizada. Em aplicações que estão sendo evoluídas, o comportamento transversal encontra-se diluído na classe invocadora. Desta forma, este comportamento deve ser removido da classe invocadora e codificado dentro do aspecto de inversão de controle associado ao tipo da interface remota do atributo. Em AdaptiveRME, para se obter a referência de um objeto remoto, deve ser executado o método

`RmeNaming.lookup(String)`, onde o argumento do tipo é o *alias* do objeto que se deseja obter a referência.

- **Passo 5 (Configuração)**: Este passo define como o interesse transversal ligado à configuração inicial de AdaptiveRME deve ser modularizado. Neste passo são definidos os seguintes sub-passos:
  - **Passo 5.1** – Deve ser criado um aspecto denominado de aspecto de configuração;
  - **Passo 5.2** – Para cada classe que exerça a função de classe principal[9], na aplicação cliente, e que a partir dela possa-se fazer invocações remotas, deve ser criado um ponto de junção no aspecto de configuração que intercepte a chamada dos seus respectivos métodos construtores. Após a criação de todos os pontos de junção deve ser criado um conjunto de junção que agrupe todos os pontos de junção declarados dentro do aspecto de configuração;
  - **Passo 5.3** – O conjunto de junção, definido no Passo 5.2, deve ser associado ao adendo que implementa as convenções de configuração inicial definidas por AdaptiveRME. Em aplicações que estão sendo evoluídas as convenções de configuração, geralmente, se encontram dentro da classe principal. Desta forma, este comportamento transversal deve ser removido da classe principal e associado ao conjunto de junção do aspecto de configuração. Em AdaptiveRME antes das aplicações utilizarem o serviço de invocação remota de métodos deve ser feita a configuração inicial do *broker* que garante a sincronia entre as partes cliente e servidor do *middleware*. Para isso, o comando `RmeConfigurator.configure()` deve ser executado.

4.3 Conclusões

Este capítulo descreveu as principais contribuições deste trabalho. A arquitetura de AdaptiveRME foi apresentada e cada uma das suas camadas foi detalhada no decorrer do capítulo. Também neste capítulo foi apresentado o processo AspectCompose que busca diminuir o acoplamento e o entrelaçamento de código nas aplicações que utilizam AdaptiveRME.

O capítulo seguinte tem como objetivo apresentar um estudo de caso demonstrando a funcionalidade de AdaptiveRME e do processo AspectCompose. E, de maneira complementar, ele descreve os resultados de uma análise de desempenho feita sobre o *middleware*, mostrando o

---

[9] Em J2SE uma classe principal possui o método public static void main(String args[]). Já em J2ME, uma classe principal estende a classe MIDlet.

impacto de sua utilização num ambiente de rede sem fio. Detalhes sobre o uso da API de AdaptiveRME na criação de aplicações e serviços de contextos, bem como a aplicação de AspectCompose são abordados no próximo capítulo.

Capítulo 5

Avaliação

Este capítulo descreve como o *middleware* e o processo, propostos nesta dissertação, foram avaliados. Na Seção 5.1 é descrito um estudo de caso que demonstra a utilização de AdaptiveRME no desenvolvimento de uma aplicação e como AspectCompose foi utilizado para fazer o desacoplamento entre a aplicação e o *middleware*. Na Seção 5.2 é apresentada uma análise de desempenho de AdaptiveRME. Por fim, a Seção 5.3 lista as principais conclusões deste capítulo.

5.1 Estudo de Caso

O estudo de caso mostra o funcionamento dos serviços providos por AdaptiveRME e a utilização do processo AspectCompose. Para tanto foi desenvolvido uma aplicação do Prontuário Eletrônico do Paciente denominada UbiPEP.

5.1.1 UbiPEP: Prontuário Eletrônico do Paciente Ubíquo

UbiPEP é uma versão ubíqua e simplificada do Prontuário Eletrônico do Paciente [Costa, 2001], cujo principal objetivo é oferecer uma maneira alternativa de acesso à informações médicas de pacientes internados através de dispositivos móveis como PDAs. UbiPEP permite que médicos visualizem descrições textuais sobre a evolução de seus pacientes internados, além de laudos e imagens de exames realizados pelos mesmos. Além disso, UbiPEP utiliza um serviço de monitoramento de pacientes cardíacos para informar aos médicos quando seus pacientes apresentam sintomas de taquicardia sinusal (freqüência cardíaca superior a 100bpm) ou bradicardia sinusal (freqüência cardíaca inferior a 50bpm).

Projetado segundo o padrão arquitetural MVC (*Model View Controller*) [Buschmann et al., 1996], UbiPEP separa a lógica de acesso aos dados (*Model*), da interface do usuário (*View*) e do fluxo da aplicação (*Controller*). O *view* e o *controller* foram implementados no dispositivo móvel e o *model* (i.e., dados do modelo) implementado no servidor como objeto remoto. Essa aplicação foi codificada em Java, sendo a parte cliente em J2ME e a parte servidora em J2SE. Toda infra-

estrutura de comunicação, adaptação de imagens de exames e notificação sobre os batimentos cardíacos dos pacientes monitorados é provida por AdaptiveRME. O diagrama de classes da Figura 5.1 apresenta as principais classes do modelo do UbiPEP.

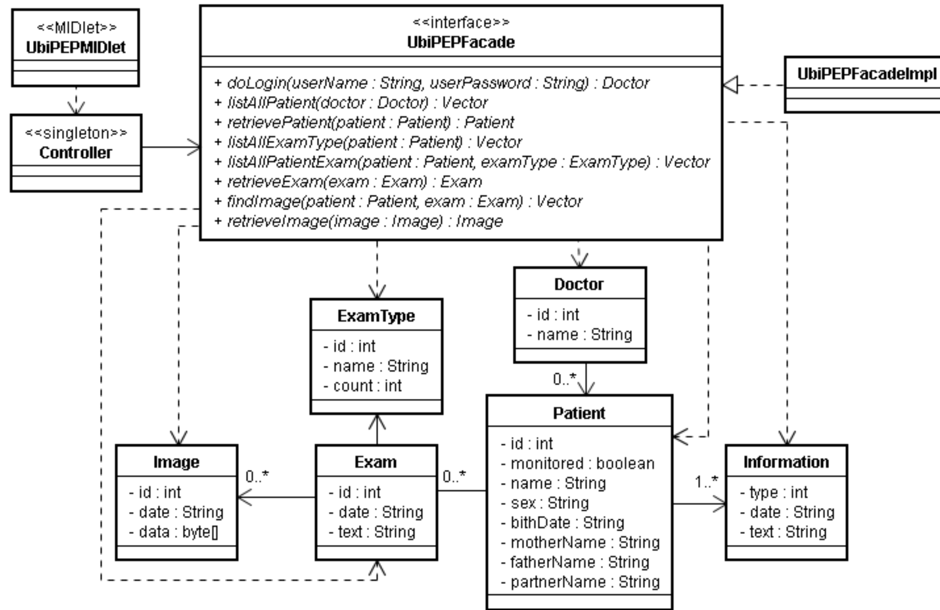


Figura 5.1: Diagrama de classes do UbiPEP.

A classe `UbiPEPMIdlet` é a classe principal da aplicação. `Patient`, `Doctor`, `ExamType`, `Exam`, `Image` e `Information` representam as entidades manipuladas pela aplicação. Já a classe `UbiPEPFacadeImpl`, realização da interface `UbiPEPFacade`, encapsula o acesso aos dados. Instâncias de `UbiPEPFacadeImpl` são publicadas como objetos remotos e acessadas pelo UbiPEP através do SIRMSC de AdaptiveRME. Por fim, a classe `Controller` gerencia o fluxo de execução do UbiPEP.

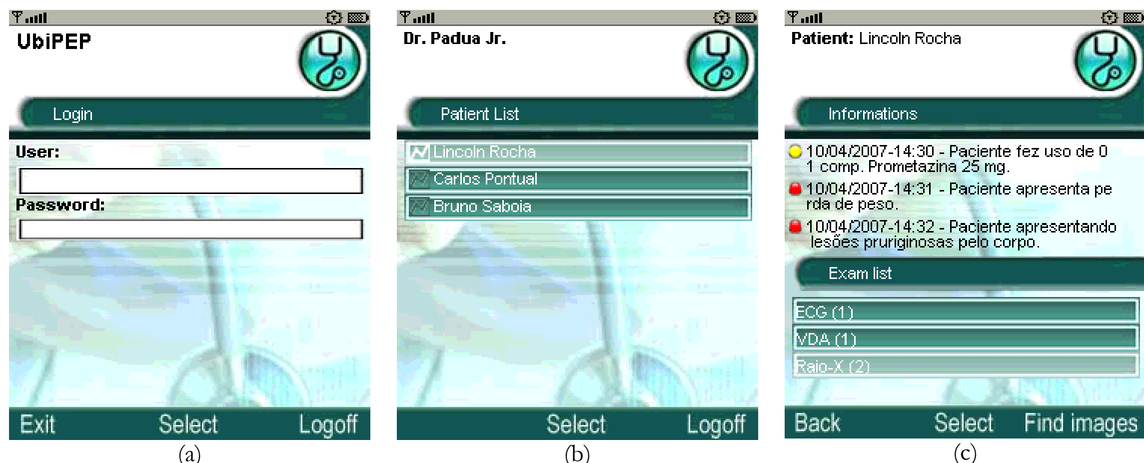


Figura 5.2: (a) Autenticação, (b) listagem de paciente e (c) detalhamento de informações em UbiPEP.

Para ter acesso às informações, os médicos precisam se autenticar no UbiPEP fornecendo

um nome de usuário e senha (Figura 5.2-a), previamente cadastrados. Após a autenticação, o UbiPEP exibe para o médico uma lista de pacientes internados que estão sob seus cuidados indicando quais deles possuem os batimentos cardíacos monitorados (Figura 5.2-b). Detalhes sobre a evolução clínica do paciente, tipos e quantidade de exames realizados podem ser visualizados após a seleção de um determinado paciente da lista (Figura 5.2-c).

A Figura 5.3-a ilustra uma lista de exames do tipo "Raio-X" realizados pelo paciente "Lincoln Rocha" nas datas: "10/04/2007" e "09/04/2007". Na Figura 5.3-b o exame "Raio-X do tórax", datado de "10/04/2007", é detalhado e a imagem associada à este exame é exibida na Figura 5.3-c.

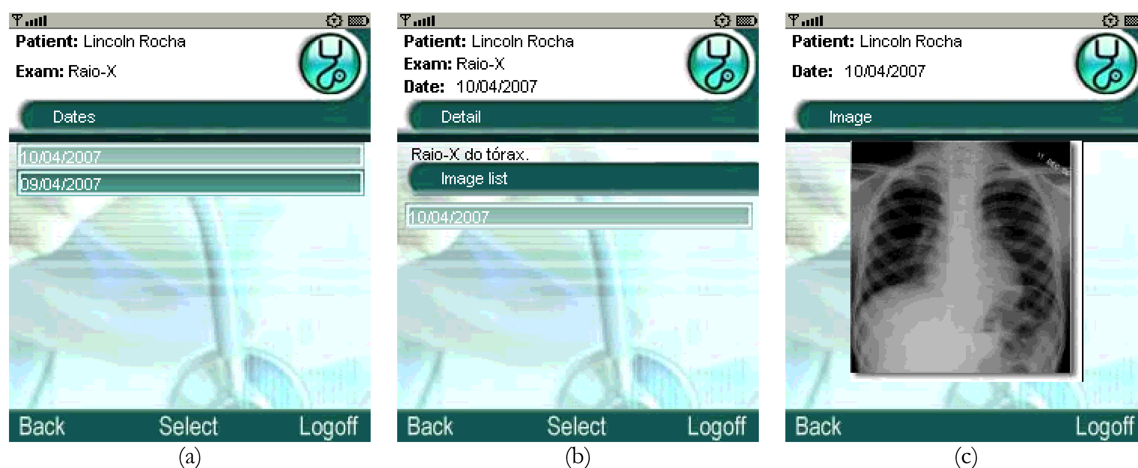


Figura 5.3: (a) Listagem de exames, (b) detalhamento de exame e (c) imagens em UbiPEP.

5.1.1.1 Configuração do SIRMSC

O SIRMSC foi configurado para adaptar o *middleware* às variações no *throughput* da rede. Para isso, foi criada uma política de compressão associada a três possíveis contextos de execução, representados por três intervalos (percentuais) de variação do *throughput*: "THROUGHPUT < 40", "40 ≤ THROUGHPUT ≤ 80" e "THROUGHPUT > 80". Para cada um dos contextos, foi criada uma estratégia de compressão que reconfigura o componente `Channel` adicionando níveis de compressão às mensagens transmitidas em função do contexto. O contexto representado pela expressão "THROUGHPUT < 40" foi associado à estratégia *Best Compression Strategy* que define que o componente `Channel` seja decorado pelo decorador `BestZipChannelDecorator`. Já o contexto "40 ≤ THROUGHPUT ≤ 80" foi associado à estratégia *Speed Compression Strategy* que define que o componente `Channel` seja decorado pelo decorador `SpeedZipChannelDecorator`. Por fim, o contexto "THROUGHPUT > 80" foi associado à estratégia *No Compression Strategy* que simplesmente descarta o uso de compressão. A Figura 5.4 ilustra a configuração dos contextos, políticas e estratégias para o UbiPEP.

```xml
XML Context Descriptor
<?xml version="1.0" encoding="UTF-8"?>
<contexts>
   <context active="true" name="HLT" description="High Level Throughput">
       <context-data-required>
           <context-data name="THROUGHPUT" />
       </context-data-required>
       <expression>(THROUGHPUT > 80)</expression>
   </context>
   <context active="true" name="MLT" description="Medium Level Throughput">
       <context-data-required>
           <context-data name="THROUGHPUT" />
       </context-data-required>
       <expression>((80 >= THROUGHPUT) && (THROUGHPUT >= 40))</expression>
   </context>
   <context active="true" name="LLT" description="Low Level Throughput">
       <context-data-required>
           <context-data name="THROUGHPUT" />
       </context-data-required>
       <expression> (40 > THROUGHPUT) </expression>
   </context>
</contexts>
```

```xml
XML Policy Descriptor
<?xml version="1.0" encoding="UTF-8"?>
<policies>
   <policy name="CP" active="true" description="Compression Policy">
       <strategy name="NCS" description="No Compression Strategy" context="HLT">
       </strategy>
       <strategy name="SCS" description="Speed Compression Strategy" context="MLT">
           <decorate component="Channel">
               <decorator impl="rme.common.decorator.SpeedZipChannelDecorator" />
           </decorate>
       </strategy>
       <strategy name="BCS" description="Best Compression Strategy" context="LLT">
           <decorate component="Channel">
               <decorator impl="rme.common.decorator.BestZipChannelDecorator" />
           </decorate>
       </strategy>
   </policy>
</policies>
```

Figura 5.4: Configuração de contextos e políticas em UbiPEP.

Além de ser configurado para se adaptar às variações do *throughput* da rede, o SIRMSC foi também utilizado no UbiPEP para prover informações de contexto sobre memória livre e resolução do display do dispositivo móvel para os adaptadores `ImageResizeAdapter` e `ImageScaleAdapter` responsáveis por adaptar as imagens dos exames acessados.

A Listagem 7 apresenta o fragmento de código do *skeleton* `UbiPEPFacadeImpl_Skeleton` mostrando como os adaptadores foram utilizados no UbiPEP. É importante mencionar que tanto *stubs* como *skeletons* são gerados automaticamente pela classe `RmeC` de RME. No fragmento de código da Listagem 7, as linhas 14 a 19 foram adicionadas para inserir o código responsável pela adaptação. Na linha 14 é obtido o valor em *bytes* da imagem que será adaptada. Na linha 15 é criada uma instância do adaptador `ImageScaleAdapter`, que será utilizada na linha 16 para fazer a primeira adaptação na imagem. Na linha 17 é criada uma instância de `ImageResizeAdapter` que na linha 18 realiza a segunda adaptação na imagem. Por fim, na linha 19, o valor da imagem adaptada é inserido no objeto que será encaminhado para o dispositivo móvel. Contudo, é importante reinterar que a adaptação só ocorre quando a imagem possui um tamanho (*bytes* e dimensão) incompatível com a

quantidade de memória livre e/ou tamanho do *display* dispositivo móvel.

Classe UbiPEPFacadeImpl_Skeleton

```
1   public class UbiPEPFacadeImpl_Skeleton extends Skeleton {
2
3   public Stream dispatch(RemoteCall r) throws Exception {
4       RmeRemoteCall remoteCall = (RmeRemoteCall) r;
5       Stream returnStream = OrbAccessor.getStream();
6       Stream args = remoteCall.getArguments();
7
8       switch (remoteCall.getOperationCode()) {
9                                                (...)
10        case 2: {
11           Image param0 = (Image) args.readObject();
12           Image retValue = ((UbiPEPFacadeImpl) super.remoteObject).retrieveImage(param0);
13           ArcademisMap parameters = remoteCall.getContextData();
14           byte[] updatedValue = retValue.getBytes();
15           ImageScaleAdapter imageScaleAdapter = new ImageScaleAdapter();
16           updatedValue = (byte[])imageScaleAdapter.adapt(updatedValue, parameters);
17           ImageResizeAdapter imageResizeAdapter = new ImageResizeAdapter();
18           updatedValue = (byte[])imageResizeAdapter.adapt(updatedValue, parameters);
19           retValue.setBytes(updatedValue);
20           returnStream.write(retValue);
21        }
22        break;
23                                               (...)
24      }
25      return returnStream;
26  }
```

Listagem 7: Fragmento de código do *skeleton* da classe UbiPEPFacadeImpl.

5.1.1.2 Serviço de Monitoramento Cardíaco

Com o intuito de validar o SNC foi criado um serviço de monitoramento de pacientes cardíacos. Tal serviço simula o monitoramento da freqüência cardíaca dos pacientes internados, permitindo que aplicações façam subscrições a fim de serem notificadas quando a freqüência cardíaca do paciente variar conforme a descrição da expressão lógica que representa o contexto de interesse da aplicação.

Para garantir o funcionamento do serviço de contexto, foi criada uma infra-estrutura composta por um PDA e dois computadores *desktops* (Figura 5.5). O UbiPEP, instalado no PDA, funciona como consumidor do serviço de contexto. Por outro lado, um dos computadores *desktop* funciona como produtor e o outro como servidor (onde são avaliadas as subscrições). De maneira geral, o serviço de monitoramento recebe os valores da freqüência cardíaca dos pacientes internados e avalia as subscrições em função destes valores.

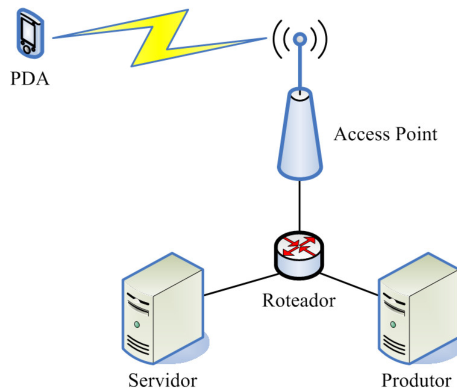


Figura 5.5: Arquitetura do serviço de contexto em UbiPEP.

Na ausência de um frequencímetro (equipamento utilizado para medir a freqüência cardíaca) para prover os valores da freqüência cardíaca, foi utilizado um arquivo texto com valores que simulam a freqüência cardíaca de um paciente ao longo do tempo. Tais freqüências são lidas e processadas pelo sensor `HeartFrequencySensor` e, posteriormente, enviadas pelo provedor (`SimpleProvider`) ao serviço de contexto. O arquivo de configuração do provedor é ilustrado na Figura 5.6.

XML Provider Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<provider>
   <service name="HFM" location=" magneto.great.ufc.br">
       <provide-data frequency="50000">
           <context-data name="HEART_FREQUENCY" />
       </provide-data>
   </service>
</provider>
```

Figura 5.6: Arquivo de configuração XML Provider Descriptor para UbiPEP.

O serviço de monitoramento cardíaco foi implementado na classe `HeartFrequencyMonitoring` (Figura 5.7) que recebe subscrições do UbiPEP e valores das freqüências cardíacas de `SimpleProvider`. `HeartFrequencyMonitoring` associa o endereço do dispositivo produtor ao paciente monitorado através de um vetor cujas entradas estão no formato **chave/valor**. A **chave** representa o identificador do dispositivo (`MACId`) e o **valor** representa o identificador do paciente (`PatientId`). Assim, quando um produtor envia os valores da freqüência cardíaca, o serviço consegue associar o valor da freqüência cardíaca ao paciente monitorado. Além disso, `HeartFrequencyMonitoring` também possui um vetor contendo o identificador de todos os dispositivos que estão autorizados a fazer subscrições. Tal fato garante que apenas os dispositivos autorizados podem ter acesso ao serviço.

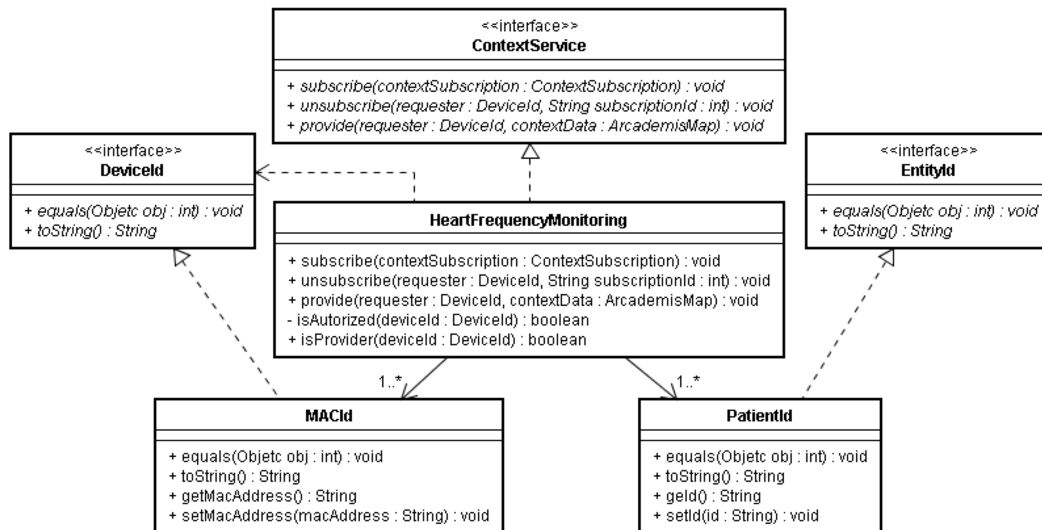


Figura 5.7: Diagrama de classes do serviço de monitoramento cardíaco.

No UbiPEP, a classe `Controller` é responsável por fazer as subscrições ao serviço de monitoramento. O fragmento de código da Listagem 8 ilustra uma subscrição feita ao serviço de monitoramento. Na linha 8 é informado o endereço da agência de localização (`magneto.great.ufc.br`) e o nome do serviço de contexto desejado (HFM). Em seguida, na linha 9 é criado um *alias* para o identificador da entidade de contexto. Na linha 10 é instanciado o tratador de contexto `TachycardiaHandler` e na linha 13 o tratador de contexto `BradycardiaHandler`, ambos ao serem notificados fazem com que o PDA emita um sinal sonoro e exiba uma tela de alerta informando o nome e a freqüência cardíaca do paciente monitorado.

Classe Controller

```
 1  Vector inmatesPatients = this.facade.listAllPatient(doctor);
 2  Enumeration e = inmatesPatients.elements();
 3  while (e.hasMoreElements()) {
 4      Patient patient = (Patient) e.nextElement();
 5      if (patient.isInmate()) {
 6         PatientId patientId = new PatientId(patient.getId());
 7         service = new ContextSubscriptionFacade();
 8         service.setLocationService("magneto.great.ufc.br", "HFM");
 9         service.addAlias("E{" + i + "}", patientId);
10         TachycardiaHandler tachycardiaHandler = new TachycardiaHandler();
11         service.addContextHandler("tachycardia", tachycardiaHandler);
12         service.subscribe("Tach" + i, "E{" + i + "}.HEART_FREQUENCY > 100");
13         BradycardiaHandler bradycardiaHandler = new BradycardiaHandler();
14         service.addContextHandler("tachycardia", tachycardiaHandler);
15         service.subscribe("Brad" + i, "E{" + i + "}.HEART_FREQUENCY < 50");
16         i++
17      }
18                                    (...)
19  }
```

Listagem 8: Fragmento de código da classe Controller.

5.1.2 AspectCompose no UbiPEP

Como mencionado anteriormente, o simples uso de AdaptiveRME provoca acoplamento e entrelaçamento às aplicações que o utilizam. Desta forma, esta seção descreve como AspectCompose foi utilizado no UbiPEP para eliminar o acoplamento e o entrelaçamento. Para aplicar os conceitos da POA, requeridos pelo processo, foi utilizada a extensão orientada a aspectos de Java, AspectJ. Os tópicos a seguir descrevem a aplicação de AspectCompose no UbiPEP, mostrando o estado do código fonte do UbiPEP antes e depois da execução de cada um dos passos do processo:

- **Aplicação das Pré-condições**: Por tratar-se de uma aplicação existente não foi necessário criar interfaces remotas, alterar declarações de atributos e aumentar o escopo de variáveis locais.
- **Aplicação do Passo 1 (Identificação)**:
  - **Passo 1.1**: A única classe remota identificada foi `UbiPEPFacadeImpl`;
  - **Passo 1.2**: A única interface remota identificada foi `UbiPEPFacade`;
  - **Passo 1.3**: A única classe invocadora identificada foi `Controller`;
  - **Passo 1.4**: As classes relacionadas identificadas foram `Doctor`, `Patient`, `ExamType`, `Exam`, `Image` e `Information`.
- **Aplicação do Passo 2 (Publicação)**:
  - Neste passo foram criados dois aspectos, `UbiPEPFacadeImplRCA`, relacionado com a classe remota `UbiPEPFacadeImpl`, e `UbiPEPFacadeRIA`, relacionado com a interface remota UbiPEPFacade. Os aspectos criados são responsáveis por encapsular todo o código transversal responsável pela publicação do objeto remoto. Tal código encontra-se diluído na classe `UbiPEPFacadeImpl` e na interface UbiPEPFacade.

  A Listagem 9 apresenta o código fonte da classe remota `UbiPEPFacadeImpl` antes da aplicação do processo. As declarações “`extends RmeRemoteObject`” e “`implements UbiPEPFacade`” representam convenções de AdaptiveRME relacionadas a publicação de objetos remotos que devem ser modularizadas pelo aspecto `UbiPEPFacadeImplRCA`.

```
Classe UbiPEPFacadeImpl
 1  public class UbiPEPFacadeImpl extends RmeRemoteObject implements UbiPEPFacade {
 2     public Doctor doLogin(String username, String userPassword){...}
 3     public Vector listAllPatient(Doctor doctor){...}
 4     public Patient retrievePatient(Patient patient){...}
 5     public Vector listAllExamType(Patient patient){...}
 6     public Vector listAllPatientExam(Patient patient, ExamType examType){...}
 7     public Exam retrieveExam(Exam exam){...}
 8     public Vector findImage(Patient patient, Exam exam){...}
 9     public Image retrieveImage(Image image){...}
10  }
```

Listagem 9: Fragmento de código fonte da classe UbiPEPFacadeImpl antes do processo.

A Listagem 10 apresenta o código fonte do aspecto de classe remota `UbiPEPFacadeImplRCA`. Na linha 2 é feita a declaração intertipo, a qual define que a classe `UbiPEPFacadeImpl` estende a classe `RmeRemoteObject`. Na linha 3 é definido que `UbiPEPFacadeImpl` realiza a interface `UbiPEPFacadeImpl`. Além disso, o conjunto de junção da linha 4 intercepta toda instanciação da classe `UbiPEPFacadeImpl`, fazendo com que toda instância criada seja publicada na agência de localização "`magneto.ufc.br`" com o "apelido" de "`ubiPEP`" (linha 7).

```
Aspecto UbiPEPFacadeImplRCA
 1  public aspect UbiPEPFacadeImplRCA {
 2     declare parents: UbiPEPFacadeImpl extends RmeRemoteObject;
 3     declare parents: UbiPEPFacadeImpl implements UbiPEPFacade;
 4     pointcut publish(UbiPEPFacadeImpl s):execution(*.new(..)) && target(s);
 5     after(UbiPEPFacadeImpl s) : publish(s){
 6        try {
 7           RmeNaming.bind("magneto.ufc.br/ubiPEP", s);
 8           s.activate();
 9        } catch (ArcademisException e) {
10        } catch (MalformedURLException e) {
11        } catch (AlreadyBoundException e) {
12        }
13  }
```

Listagem 10: Fragmento de código fonte do aspecto UbiPEPFacadeImplRCA.

```
Classe UbiPEPFacadeImpl
 1  public class UbiPEPFacadeImpl {
 2     public Doctor doLogin(String userName, String userPassword){...}
 3     public Vector listAllPatient(Doctor doctor){...}
 4     public Patient retrievePatient(Patient patient){...}
 5     public Vector listAllExamType(Patient patient){...}
 6     public Vector listAllPatientExam(Patient patient, ExamType examType){...}
 7     public Exam retrieveExam(Exam exam){...}
 8     public Vector findImage(Patient patient, Exam exam){...}
 9     public Image retrieveImage(Image image){...}
10  }
```

Listagem 11: Fragmento de código fonte da classe UbiPEPFacadeImpl após o processo.

A Listagem 11 mostra o código fonte da classe remota `UbiPEPFacadeImpl` após a aplicação do processo. É importante observar que todo o código transversal foi removido para o aspecto `UbiPEPFacadeImplRCA`.

A Listagem 12 mostra o código fonte da interface remota `UbiPEPFacade` antes da aplicação do processo. As declarações "`extends Remote`" da linha 1 e demais declarações do tipo "`throws ArcademisException`", que ocorrem ao longo do código, representam convenções de AdaptiveRME relacionadas a publicação do objeto remoto. Tais convenções devem ser modularizadas pelo aspecto `UbiPEPFacadeRIA`.

```
Classe UbiPEPFacade
 1  public class UbiPEPFacade extends Remote {
 2    public Doctor doLogin(String userName, String userPassword) throws
 3            ArcademisException;
 4    public Vector listAllPatient(Doctor doctor) throws ArcademisException;
 5    public Patient retrievePatient(Patient patient) throws ArcademisException;
 6    public Vector listAllExamType(Patient patient) throws ArcademisException;
 7    public Vector listAllPatientExam(Patient patient, ExamType examType) throws
 8            ArcademisException;
 9    public Exam retrieveExam(Exam exam) throws ArcademisException;
10    public Vector findImage(Patient patient, Exam exam) throws ArcademisException;
11    public Image retrieveImage(Image image) throws ArcademisException;
12  }
```

Listagem 12: Fragmento de código fonte da classe UbiPEPFacade antes do processo.

A Listagem 13 apresenta o código fonte do aspecto de interface remota `UbiPEPFacadeRIA`. Na linha 2 é feita a declaração intertipo, a qual define que a interface `UbiPEPFacade` estende a interface `Remote`. Na linha 3 é definido que todos os métodos da interface `UbiPEPFacade` e de suas descendentes lançam a exceção `ArcademisException`, como requerido por AdaptiveRME.

```
Aspecto UbiPEPFacadeRIA
 1  public aspect UbiPEPFacadeImplRCA {
 2    declare parents: UbiPEPFacade extends Remote;
 3    declare soft: ArcademisException : execution(* +UbiPEPFacade.*(..));
 4  }
```

Listagem 13: Fragmento de código fonte do aspecto UbiPEPFacadeRIA.

```
Classe UbiPEPFacade
 1  public class UbiPEPFacade {
 2    public Doctor doLogin(String userName, String userPassword);
 3    public Vector listAllPatient(Doctor doctor);
 4    public Patient retrievePatient(Patient patient);
 5    public Vector listAllExamType(Patient patient);
 6    public Vector listAllPatientExam(Patient patient, ExamType examType);
 7    public Exam retrieveExam(Exam exam);
 8    public Vector findImage(Patient patient, Exam exam);
 9    public Image retrieveImage(Image image);
10  }
```

Listagem 14: Fragmento de código fonte da classe UbiPEPFacade após o processo.

A Listagem 14 mostra o código fonte da interface remota UbiPEPFacade após a aplicação do processo. É importante salientar que todo o código transversal foi removido para o aspecto UbiPEPFacadeRIA.

- **Aplicação do Passo 3 (Serialização)**:
  - **Passo 3.1**: Neste passo foram criados seis aspectos para modularizar o protocolo de serialização imposto por AdaptiveRME as classes relacionadas `Doctor`, `Patient`, `ExamType`, `Exam`, `Image` e `Information`.

A Listagem 15 apresenta o código fonte da classe `Doctor`. A declaração "`implements Marshalable`", feita na linha 1, é imposta por AdaptiveRME para que objetos da classe relacionada `Doctor` possam ser serializadas e transportadas por um canal de comunicação. Tal declaração obriga que a classe `Doctor` implemente os métodos `marshal(Stream b)` e `unmarshal(Stream b)` responsáveis, respectivamente, pela ordem com que os atributos internos da classe `Doctor` serão serializados e deserializados.

Classe Doctor

```
 1  public class Doctor implements Marshalable {
 2    private int id;
 3    private String name;
 4    public int getId(){...}
 5    public void setId(int id){...}
 6    public String getName(){...}
 7    public void setName(String name) {...}
 8    public void marshal(Stream b) throws MarshalException {
 9       b.write(this.id);
10       b.write(this.name);
11    }
12    public void unmarshal(Stream b) throws MarshalException {
13       this.id = b.readInt();
14       this.name = (String) b.readObject();
15    }
16  }
```

Listagem 15: Fragmento de código fonte da classe Doctor antes do processo.

Como o procedimento é similar para todas as classes relacionadas, a Listagem 16 ilustra apenas o código fonte do aspecto de serialização `DoctorSA` que está associado à classe relacionada `Doctor`. Na linha 2 é feita a declaração intertipo que define a realização da interface `Marshalable` por parte da classe `Doctor`. Na linha 3 é feita a codificação do método `marshal(Stream b)` responsável por fazer a serialização de instâncias de `Doctor`. Na linha 7 é feita a codificação do método `unmarshal(Stream b)` responsável pela deserialização de objetos do tipo `Doctor`.

Aspecto DoctorSA

```
 1  public privileged aspect DoctorSA {
 2    declare parents: Doctor implements Marshalable;
 3    public void Doctor.marshal(Stream b) throws MarshalException {
 4       b.write(this.id);
 5       b.write(this.name);
```

```
 6      }
 7      public void Doctor.unmarshal(Stream b) throws MarshalException {
 8         this.id = b.readInt();
 9         this.name = (String) b.readObject();
10      }
11   }
```

Listagem 16: Fragmento de código fonte do aspecto DoctorSA.

A Listagem 17 apresenta o código fonte da classe `Doctor` totalmente livre do código transversal após a aplicação do processo.

```
Classe Doctor
 1   public class Doctor {
 2      private int id;
 3      private String name;
 4      public int getId(){...}
 5      public void setId(int id){...}
 6      public String getName(){...}
 7      public void setName(String name) {...}
 8   }
```

Listagem 17: Fragmento de código fonte da classe Doctor após o processo.

- **Aplicação do Passo 4 (Localização)**:
  - **Passo 4.1**: Após a execução deste passo foi criado o aspecto de inversão de controle, `UbiPEPFacadeIoCA`, associado à interface remota `UbiPEPFacade`.

```
Classe Controller
 1   public class Controller {
 2      private static Controller instance;
 3      private UbiPEPFacade facade = null;
 5                                         (...)
 6      public synchronized static Controller getInstance() {...}
 7      public synchronized static Controller getInstance(UbiPEPMIDlet owner) {...}
 8      private Controller() {
 9         this.referenceStake = new Vector();
10         Session.init();
11         try {
12            this.facade = (UbiPEPFacade) RmeNaming.lookup("magneto.ufc.br/ubiPEP");
13         } catch (MalformedURLException e) {
14         } catch (ArcademisException e) {
15         } catch (NotBoundException e) {
16         }
17      }
18                                         (...)
19   }
```

Listagem 18: Fragmento de código fonte da classe Controller antes do processo.

  - **Passo 4.2**: Neste passo apenas a classe `Controller` foi selecionada. A Listagem 18 apresenta parte do código fonte da classe `Controller` antes da aplicação do processo. A linha 3 ilustra a declaração do atributo `facade` do tipo `UbiPEPFacade`. Já na linha 8 é apresentado o construtor da classe onde se encontra o código transversal relativo à localização do objeto remoto (linhas 11 à

16).

A Listagem 19 mostra o código do aspecto de inversão de controle `UbiPEPFacadeIoCA`. Na linha 3 é apresentado o conjunto de junção responsável pela interceptação da ação de leitura do atributo `facade` do tipo `UbiPEPFacade` da classe invocadora `Controller`.

```
Aspecto UbiPEPFacadeIoCA
 1  public privileged aspect UbiPEPFacadeIoCA {
 2     private Remote reference = null;
 3     pointcut intercept(Controller c):get(* Controller.facade) && target(c);
 4     before(Controller c): intercept(c){
 5        if(this.reference == null){
 6          try {
 7             this.reference= RmeNaming.lookup("magneto.ufc.br/ubiPEP");
 8          } catch (MalformedURLException e) {
 9          } catch (ArcademisException e) {
10          } catch (NotBoundException e) {
11          }
12        }
13        c.facade = (UbiPEPFacade) this.reference;
14     }
15  }
```

Listagem 19: Fragmento de código fonte do aspecto UbiPEPFacadeIoCA.

- **Passo 4.3**: Após a execução deste passo, o código transversal responsável pela localização do objeto remoto foi removido da classe invocadora `Controller` (Listagem 18, da linha 11 à 16) e codificado no aspecto `UbiPEPFacadeIoCA`. (Listagem 19, da linha 6 à 11).

A Listagem 20 apresenta o código fonte da classe `Controller` após a aplicação do processo. Pode ser observado que o código transversal ligado à localização do objeto remoto foi removido do construtor da classe (linha 8).

```
Classe Controller
 1  public class Controller {
 2     private static Controller instance;
 3     private UbiPEPFacade facade = null;
 5                                       (...)
 6     public synchronized static Controller getInstance() {...}
 7     public synchronized static Controller getInstance(UbiPEPMIDlet owner) {...}
 8     private Controller() {
 9        this.referenceStake = new Vector();
10        Session.init();
11     }
12                                       (...)
13  }
```

Listagem 20: Fragmento de código fonte da classe Controller após o processo.

- **Aplicação do Passo 5 (Configuração)**:
  - **Passo 5.1**: Após a execução deste passo foi criado o aspecto de configuração `ConfigurationA`.

- **Passo 5.2**: Apenas a classe `UbiPEPMIDlet` exerce o papel de classe principal no UbiPEP. A Listagem 21 ilustra o código transversal (linhas de 4 à 8) relacionado com a configuração inicial do *middleware* na classe principal `UbiPEPMIDlet` antes da aplicação do processo.

```
Aspecto UbiPEPMIDlet
 1  public class UbiPEPMIDlet extends MIDlet {
 2    private Controller controller = null;
 3    public UbiPEPMIDlet() {
 4       try {
 5          RmeConfigurator configurator = new RmeConfigurator();
 6          configurator.configure();
 7       } catch (ReconfigurationException e) {
 8       }
 9    }
10                                      (...)
11  }
```

Listagem 21: Fragmento de código fonte da classe UbiPEPMIDlet antes do processo.

A Listagem 22 apresenta o código fonte do aspecto de configuração `ConfigurationA`, interceptando o construtor da classe principal `UbiPEPMIDlet` (linha 2).

```
Aspecto ConfigurationA
 1  public aspect ConfigurationA {
 2    pointcut init(): initialization(UbiPEPMIDlet.new(..));
 3    after() : init(){
 4       try {
 5            RmeConfigurator configurator = new RmeConfigurator();
 6            configurator.configure();
 7       } catch (ReconfigurationException e) {
 8       }
 9    }
10  }
```

Listagem 22: Fragmento de código fonte do aspecto ConfigurationA.

- **Passo 5.3**: Neste passo foi implementado o código transversal responsável pela configuração inicial de AdaptiveRME no aspecto de configuração `ConfigurationA`. O adendo (Listagem 22, linha 3) apresenta o código transversal relacionado a configuração inicial de AdaptiveRME.

```
Aspecto UbiPEPMIDlet
 1  public class UbiPEPMIDlet extends MIDlet {
 2    private Controller controller = null;
 3    public UbiPEPMIDlet() {
 4    }
 5                                      (...)
 6  }
```

Listagem 23: Fragmento de código fonte da classe UbiPEPMIDlet após o processo.

Por fim, a Listagem 23 apresenta o código fonte da classe principal `UbiPEPMIDlet` totalmente desprovida de código transversal oriundo da configuração inicial do *middleware*.

5.2 Análise de Desempenho

Para tornar RME um *middleware* adaptativo dinâmico, várias funcionalidades foram incorporadas a sua versão adaptativa, AdaptiveRME. Dentre elas, um Serviço de Invocação Remota de Métodos Sensível ao Contexto (SIRMSC) que utiliza informações de contexto para reconfigurar o *middleware* durante uma invocação. Contudo, SIRMSC insere uma sobrecarga de processamento devido a necessidade de avaliação de contextos, sincronização de políticas e reconfiguração dos componentes do *middleware*. Tal sobrecarga pode impactar no desempenho do *middleware* e, conseqüentemente, nas aplicações que o utilizam. Com o objetivo de mensurar o impacto no desempenho provocado pelo SIRMSC, esta seção descreve uma análise de desempenho onde é feita uma comparação entre RME e AdaptiveRME.

Para avaliar o desempenho, a técnica escolhida foi a medição e a métrica foi o RTT (*Round Trip Time*), tempo de duração de uma chamada remota desde a sua invocação pelo cliente, passando pelo processamento do servidor, até o recebimento da resposta pelo cliente. A métrica foi aplicada sobre RME e AdaptiveRME. Ao calcular o RTT de RME obteve-se uma base para o cálculo do *overhead* imposto pelo SIRMSC de AdaptiveRME.

A comparação dos dois sistemas de *middleware* foi feita utilizando a técnica *paired observation*, a mesma usada por *Costa* [Costa, 2004] e descrita em [Jain, 1992]. Esta técnica permite comparar dois sistemas e afirmar, baseado nas medições, com um determinado nível de confiança, qual dos sistemas tem melhor desempenho. Para isso, os experimentos realizados devem possuir correspondência um para um nos dois sistemas. Assim, se *n* experimentos são realizados em dois sistemas X e Y, deve haver uma correspondência de um para um entre os *n workloads* utilizados em cada sistema.

Uma vez obtidas as médias dos experimentos realizados nos dois sistemas, é calculada a diferença entre cada par de médias que possuam correspondência um para um. Em seguida, é calculada a média destas diferenças e o intervalo de confiança. Caso este intervalo inclua zero, pode-se afirmar que os dois sistemas não são diferentes com um determinado nível de confiança. Caso o intervalo não inclua zero, um dos sistemas possui um desempenho superior.

Na avaliação foi escolhido como *benchmark* o usado em [Costa, 2004], que é um subconjunto daquele usado em [Juric et al., 2000], cujas operações estão definidas na interface `IBenchmark` (Figura 5.8), a qual possui métodos que aceitam (`accept`) ou retornam (`return`) tipos primitivos ou um objeto do tipo `String`. Os métodos cujos nomes iniciam com `accept` recebem como argumento um determinado tipo primitivo ou `String` e não retornam nenhum valor. Por exemplo, o método `acceptBoolean` recebe um `boolean`

(verdadeiro ou falso) como argumento. Os métodos cujos nomes iniciam com `return` não recebem argumentos e retornam um determinado tipo primitivo ou `String`. Por exemplo, o método `returnBoolean` retorna um `boolean` como resultado.

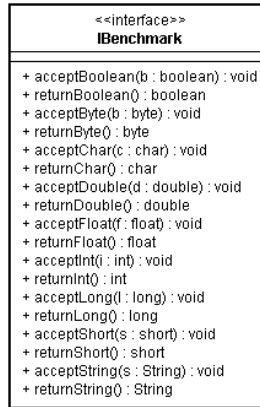


Figura 5.8: Interface Ibenchmark.

A implementação da interface `IBenchmark` utilizada no experimento possui métodos que não realizam processamento. Assim, eles praticamente não influenciam na medição do RTT, que deste modo representa o tempo da rede somado ao tempo de processamento da invocação no *middleware*. No experimento, para cada *middleware*, cada método da implementação da interface `IBenchmark` foi invocado 10.000 vezes. O RTT de cada invocação remota foi medido utilizando o método estático `currentTimeMillis()` da classe `java.lang.System()`.

O ambiente de execução do experimento (Figura 5.9) foi composto por: um computador *Pentium IV* de 3GHz, 1GB de RAM, *Windows XP* e máquina virtual Java da *Sun Microsystems* versão 1.5; um *Access Point* (AP) SENAO SL-3054 CB3 *Deluxe*, com velocidade de transmissão de 54Mbps, e interfaces *Ethernet* e IEEE 802.11b/g; e um iPAQ h4155 (h4100 series) com um processador PXA255 400MHz, 64MB de memória SDRAM, tecnologia de comunicação IEEE 802.11b, sistema operacional *Microsoft Pocket* PC 2003 e máquina virtual Java J9 (versão 2.2) da IBM.

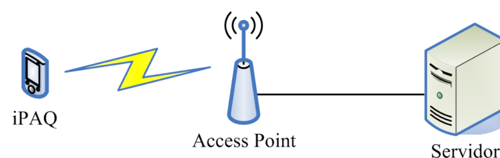


Figura 5.9: Cenário de execução do experimento.

5.2.1 Configuração para o Experimento

O SIRMSC de AdaptiveRME foi configurado para atender requisitos de qualidade de serviço em função de variações do *throughput* e da qualidade do sinal. Para isso formam criados três contextos no dispositivo móvel baseados em combinações de intervalos de valores envolvendo as duas variáveis de contexto: *throughput* e qualidade do sinal. Tais contextos modelam situações onde a relação entre *throughput* e qualidade do sinal podem ser ótimas, medianas ou ruins. Cada contexto criado foi associado a uma estratégia de compressão da política de QoS (*Quality of Service*) definida. Neste caso, uma relação ruim entre *throughput* e qualidade do sinal implica na utilização da estratégia *Best Compression Strategy*, a qual define que o componente `Channel` seja decorado pelo decorador `BestZipChannelDecorator`. Já para uma relação mediana entre *throughput* e qualidade do sinal, a estratégia *Speed Compression Strategy*, a qual define que o componente `Channel` seja decorado pelo decorador `SpeedZipChannelDecorator`, deve ser utilizada. Por outro lado, se a relação entre *throughput* e qualidade do sinal for ótima, a estratégia a ser empregada é *No Compression Strategy*, a qual simplesmente descarta o uso de compressão.

Já do lado servidor, outros três contextos criados foram associados a cada uma das estratégias da política de QoS adotada. Entretanto, estes contextos não modelam nenhum evento específico e sempre são avaliados como verdadeiros. Isto é feito para forçar o avaliador de estratégias a sempre utilizar o interpretador de contexto durante o processo de sincronização, garantindo que todos os passos de execução do SIRMSC (exibidos na Figura 4.8) sejam executados.

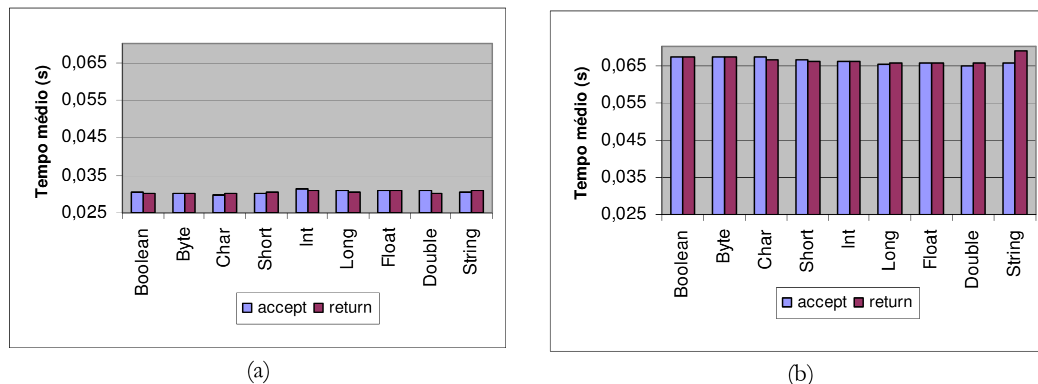
Numa avaliação de desempenho, os sistemas comparados devem possuir iguais condições de execução, por isso o contexto de execução foi mantido fixo de modo que apenas a estratégia de não fazer compressão fosse utilizada. Esta medida tornou-se necessária, pois em determinados cenários, o fato de AdaptiveRME utilizar compressão poderia favorecê-lo. Contudo, é importante salientar que para todas as 10.000 execuções de cada um dos métodos do *benchmark* sobre AdaptiveRME, todos os contextos são avaliados com o intuito de decidir qual é a melhor estratégia a ser utilizada. Tal impacto representa o *overhead* do SIRMSC que se quer avaliar.

5.2.2 Resultados Obtidos

De cada *middleware* a média do RTT de cada método do *benchmark*, o desvio padrão e o intervalo de confiança das médias foram medidos. O desvio padrão da média indica a variação dos valores individuais da amostra, de maneira que, quanto maior o desvio padrão, maior a

variação destes valores. O intervalo de confiança da média informa, com certa probabilidade, o conjunto de valores que a média calculada de outra amostra do mesmo experimento poderá ter. Esta probabilidade é denominada de coeficiente de confiança. O coeficiente de confiança, quando representado em forma de percentagem, tem o nome de nível de confiança. Em toda a avaliação de desempenho, o nível de confiança utilizado para determinação dos intervalos de confiança foi 95%. Um nível de confiança de 95% implica em um nível de significância de 5% (α=5%).

As Figuras 5.10-a e 5.10-b apresentam os tempos médios das 10.000 invocações de cada método do *benchmark* sobre RME e AdaptiveRME, respectivamente. Observa-se que a diferença entre as médias dos métodos `accept` e `return` sobre ambos os sistemas de *middleware* é mínima. Entretanto, como já era esperado, os tempos médios na execução do *benchmark* sobre AdaptiveRME são maiores que os de RME. Isso é justificável, pois para cada execução de um método do *benchmark* sobre AdaptiveRME, o interpretador de contexto é acionado, tanto do lado cliente quanto do lado servidor, para analisar qual será a melhor estratégia a ser utilizada durante a execução da chamada remota. Além disso, o fato de AdaptiveRME fazer a negociação de estratégias impacta no número e no tamanho das mensagens trafegadas na rede.



(a) (b)

Figura 5.10: Médias do RTT de (a) RME e (b) AdaptiveRME.

Na Tabela 5.1 são mostrados as médias, os desvios padrões e os intervalos de confiança calculados para a execução do *benchmark* sobre RME. Observa-se que os valores das médias não se diferenciaram muito de um método para outro, apesar da diferença do número de *bits* necessários para transportar cada tipo de dado através da rede.

Na Tabela 5.2 são mostrados as médias, os desvios padrões e os intervalos de confiança obtidos para a execução do *benchmark* sobre AdaptiveRME. De maneira similar a RME, os valores das médias não se diferenciaram muito de um método para outro, apesar de AdaptiveRME adicionar *overhead* relacionado a políticas e dados de contexto trafegados na sincronização e invocação remota.

Tabela 5.1: RTT de RME.

Métodos	M	DP	IC (α=0,05)
acceptBoolean	0,0306	0,2585	(0,0256 , 0,0357)
acceptByte	0,0303	0,2579	(0,0252 , 0,0353)
acceptChar	0,0295	0,2503	(0,0246 , 0,0345)
acceptShort	0,0302	0,2466	(0,0253 , 0,0350)
acceptInt	0,0312	0,2634	(0,0260 , 0,0363)
acceptLong	0,0311	0,2731	(0,0257 , 0,0364)
acceptFloat	0,0311	0,2764	(0,0257 , 0,0365)
acceptDouble	0,0311	0,2658	(0,0259 , 0,0363)
acceptString	0,0307	0,2487	(0,0258 , 0,0355)
returnBoolean	0,0300	0,2459	(0,0252 , 0,0348)
returnByte	0,0303	0,2428	(0,0255 , 0,0350)
returnChar	0,0300	0,2462	(0,0251 , 0,0348)
returnShort	0,0304	0,2576	(0,0253 , 0,0354)
returnInt	0,0311	0,2761	(0,0256 , 0,0365)
returnLong	0,0306	0,2622	(0,0254 , 0,0357)
returnFloat	0,0307	0,2791	(0,0253 , 0,0362)
returnDouble	0,0302	0,2550	(0,0252 , 0,0352)
returnString	0,0309	0,2650	(0,0257 , 0,0361)
[M = Média] [DP = Desvio Padrão] [IC = Intervalo de Confiança]			

Tabela 5.2: RTT de AdaptiveRME.

Métodos	M	DP	IC (α=0,05)
acceptBoolean	0,0671	0,4092	(0,0591 , 0,0751)
acceptByte	0,0674	0,3941	(0,0597 , 0,0751)
acceptChar	0,0672	0,4077	(0,0592 , 0,0752)
acceptShort	0,0664	0,3933	(0,0587 , 0,0741)
acceptInt	0,0661	0,3861	(0,0586 , 0,0737)
acceptLong	0,0653	0,3580	(0,0583 , 0,0723)
acceptFloat	0,0655	0,3706	(0,0582 , 0,0728)
acceptDouble	0,0648	0,3573	(0,0578 , 0,0718)
acceptString	0,0657	0,3606	(0,0587 , 0,0728)
returnBoolean	0,0674	0,4183	(0,0592 , 0,0756)
returnByte	0,0672	0,4049	(0,0593 , 0,0751)
returnChar	0,0665	0,3990	(0,0587 , 0,0743)
returnShort	0,0662	0,3743	(0,0588 , 0,0735)
returnInt	0,0659	0,3627	(0,0588 , 0,0730)
returnLong	0,0657	0,3709	(0,0584 , 0,0729)
returnFloat	0,0658	0,3653	(0,0586 , 0,0730)
returnDouble	0,0656	0,3538	(0,0586 , 0,0725)
returnString	0,0666	0,4120	(0,0586 , 0,0747)
[M = Média] [DP = Desvio Padrão] [IC = Intervalo de Confiança]			

Na comparação realizada entre RME e AdaptiveRME, a média calculada para as diferenças entre as médias da execução do *benchmark* foi -0,0357 e o intervalo de confiança (-0,0362 , -0,0352) com α=0,05, não incluindo zero. Desse modo, baseado nas amostras selecionadas e na técnica de comparação utilizada (*paired observation*), conclui-se, com 95% de confiança, que RME possui um desempenho superior a AdaptiveRME, como já era esperado, devido a sobrecarga de processamento e tráfego de rede inserido por ele. Entretanto, a média das diferenças de tempo entre cada *workload* de cada sistema avaliado foi da ordem de 0,03s, sendo este o *overhead* inserido pelo SIRMSC de AdaptiveRME neste experimento, mostrando que para este experimento, mesmo tendo desempenho inferior ao de RME, AdaptiveRME possui um desempenho aceitável quando comparado suas medições com outros sistemas de *middleware* [Juric et al., 2000] e [Costa, 2004].

Contudo, o cenário apresentado no experimento despreza o tempo de processamento da chamada remota no servidor bem como o volume de dados trafegados na rede. Avaliando outro cenário mais realístico onde estes fatores são observados (Figuras 5.11 e 5.12), AdaptiveRME tende para um desempenho similar ao de RME.

Os gráficos apresentados nas Figuras 5.11 e 5.12 foram obtidos da seguinte maneira: para obtenção do gráfico da Figura 5.11 criou-se um objeto remoto que possui como argumento um vetor de *bytes* de tamanho 100 e, para cada uma das 1000 execuções deste método, variou-se o tempo de resposta do mesmo de 0,1 a 0,8 segundos. Já para obtenção do gráfico da Figura 5.12 adotou-se o mesmo objeto remoto, entretanto, fixou-se o tempo de resposta do método remoto em 0,05s e variou-se o valor do argumento do método de 100 à 800 *bytes*.

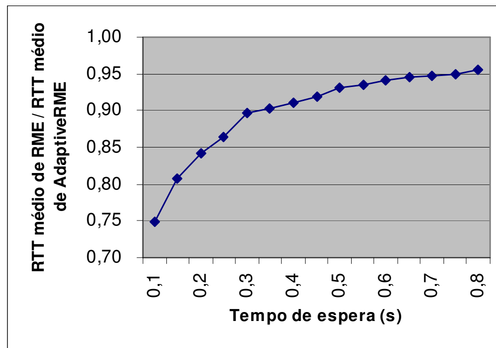


Figura 5.11: Relação do RTT médio de RME e AdaptiveRME com overhead de rede fixo e tempo de processamento variável no servidor.

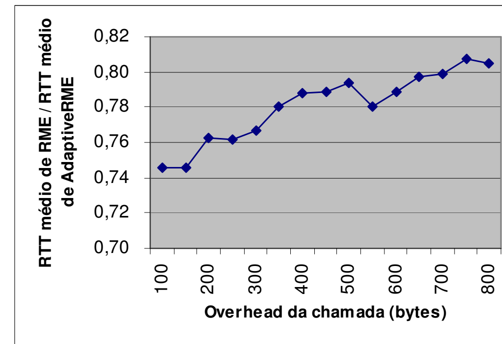


Figura 5.12: Relação do RTT médio de RME e AdaptiveRME com overhead de rede variável e tempo de processamento fixo no servidor.

Observando o gráfico da Figura 5.11 percebe-se que com o aumento no tempo de processamento da chamada remota no lado servidor, a relação entre o RTT médio de RME e o RTT médio de AdaptiveRME tende para 1. Da mesma forma, observando o gráfico da Figura 5.12, percebe-se que com o aumento no volume de dados trafegados na rede, a relação entre o RTT médio de RME e o RTT médio de AdaptiveRME também tende para 1. Isso mostra que em condições mais realísticas RME e AdaptiveRME tendem para um mesmo desempenho.

5.3 Conclusões

Este capítulo descreveu a avaliação dos resultados obtidos nesta dissertação de mestrado. Um estudo de caso foi utilizado para ilustrar o funcionamento de AdaptiveRME e a aplicação do processo AspectCompose. Além disso, foram descritos os resultados de uma análise de desempenho mostrando o impacto do SIRMSC de AdaptiveRME num ambiente de rede sem fio.

O principal objetivo do estudo de caso descrito neste capítulo foi mostrar como AdaptiveRME pode ser utilizado no desenvolvimento de aplicações móveis e ubíquas. Na sua descrição, três pontos foram abordados: a configuração do SIRMSC para atender as variações de *throughput* da rede, a modificação do código fonte do *skeleton* `UbiPEPFacade_Impl` para dar suporte a adaptação de imagens, e o modo como o serviço de monitoramento de pacientes cardíacos foi construído e utilizado por UbiPEP. Outro ponto abordado no estudo de caso foi a aplicação do processo AspectCompose sobre o UbiPEP.

A análise de desempenho, por sua vez, indica que AdaptiveRME possui um desempenho inferior ao de RME, mas que para o cenário avaliado no experimento seu desempenho é aceitável. Além disso, foi mostrado que em condições mais realísticas de comunicação, RME e AdaptiveRME possuem desempenhos parecidos. Contudo, tal análise não avaliou o impacto da execução concorrente dos sensores, sendo esse um trabalho a ser proposto como trabalho futuro.

Outro ponto a ser investigado como trabalho futuro é o impacto do número de contextos que são avaliados durante uma invocação remota.

No próximo capítulo são apresentadas as principais contribuições dessa dissertação e possíveis trabalhos futuros.

Capítulo 6

Conclusão

Esta dissertação propôs um *middleware* adaptativo para ambientes móveis e ubíquos e um processo de composição de *software* orientado a aspectos. O AdaptiveRME utiliza mecanismos de reconfiguração dinâmica, adaptação de conteúdo e notificação de contextos ativos para facilitar o desenvolvimento de aplicações móveis e ubíquas. Já o processo de composição tem como objetivo diminuir o acoplamento e entrelaçamento de código das aplicações que fazem uso do *middleware* proposto. Um estudo de caso foi construído para mostrar a utilização dos recursos fornecidos por AdaptiveRME e ilustrar a aplicação do processo AspectCompose. Além disso, uma análise de desempenho foi realizada para avaliar o desempenho de AdaptiveRME em uma rede sem fio.

O restante deste capítulo descreve as principais contribuições deste trabalho de dissertação (Seção 6.1) e aponta as possíveis linhas de pesquisa a serem investigadas como trabalhos futuros (Seção 6.2).

6.1 Contribuições

Desenvolver sistemas de *software* para ambientes ubíquos é uma tarefa difícil. Características típicas destes ambientes como a alta heterogeneidade e dinamicidade obrigam os engenheiros de *software* a se debruçarem sobre inúmeros requisitos não funcionais durante o processo de desenvolvimento (e.g., distribuição, adaptação e sensibilidade ao contexto). Neste cenário, a principal contribuição desta dissertação é a proposta de AdaptiveRME, que através de sua arquitetura de manipulação de contexto e mecanismos de reconfiguração dinâmica, adaptação de conteúdo e notificação de contextos ativos, facilita o desenvolvimento de aplicações móveis e ubíquas.

AdaptiveRME abstrai a heterogeneidade de *hardware* e *software* do ambiente móvel e ubíquo, permitindo que aplicações executem em dispositivos computacionais distintos com diferentes tamanhos e poder computacional. Além disso, AdaptiveRME utiliza informações de contexto para garantir o atendimento a requisitos não funcionais (especificados pelo engenheiro de *software*)

através de reconfigurações dinâmicas de seus componentes internos. Tal característica possibilita ao *middleware* adaptar-se dinamicamente às condições do ambiente em que executa, diminuindo o impacto da dinamicidade do ambiente sobre as aplicações. Ao mesmo tempo em que, através do serviço de notificação de contexto, as aplicações podem tomar conhecimento de eventos que ocorrem no ambiente e a partir disso realizar algum processamento contextualizado.

Além de propor um *middleware* para facilitar o desenvolvimento de sistemas ubíquos, esta dissertação também apresenta como contribuição o processo AspectCompose, o qual tem como objetivo conferir maiores níveis de reúso e manutenção às aplicações que fazem uso de AdaptiveRME. Para isso, AspectCompose propõe o uso de técnicas de POA para diminuir o acoplamento e o entrelaçamento de código.

Vale ressaltar que a versão adaptativa de RME, proposta nesta dissertação, habilita os dispositivos móveis a atuarem como fornecedores de serviços. Tal característica permite que dispositivos móveis compartilhem seus recursos com outros dispositivos computacionais para os mais variados fins. Câmeras fotográficas, *mp3 players* e acesso a internet são exemplos de recursos disponíveis em dispositivos móveis que podem ser compartilhados para a criação de ambientes ubíquos.

Finalmente, é importante mencionar que uma contribuição secundária deste trabalho foi a criação de um protótipo funcional do Prontuário Eletrônico do Paciente Ubíquo, UbiPEP, o qual pode ser facilmente melhorado e incorporado ao cotidiano de clínicas médicas e hospitais.

6.2 Trabalhos Futuros

Esta dissertação de mestrado possibilita trabalhos futuros relevantes tanto no que diz respeito a evolução de AdaptiveRME quanto a generalização de AspectCompose.

Um ponto importante a ser trabalhado em AdaptiveRME é a padronização na descrição de entidades contextuais. Uma vez definida uma linguagem padrão para descrever e representar entidades contextuais, as aplicações obteriam um ganho considerável no que diz respeito a interoperabilidade. Nessa linha de pesquisa, estudos apontam para o uso de ontologias [Chen et al., 2004] [Tamma et al., 2005] [Kim et al., 2006] como ferramenta de descrição e representação de entidades contextuais.

Outra possível linha de pesquisa a ser investigada como trabalho futuro é a utilização de AdaptiveRME para fazer a reconfiguração de componentes que constituem requisitos funcionais das próprias aplicações. Assim, além dos requisitos não funcionais, os requisitos funcionais das

aplicações passariam a ser sensíveis ao contexto.

Outro possível desdobramento desta dissertação como trabalho futuro seria uma extensão do mecanismo de reconfiguração de AdaptiveRME para possibilitar a substituição e adição de componentes em tempo de execução. Tal extensão possibilitaria a atualização dinâmica, tanto do *middleware* quanto das aplicações, sem a necessidade de interrupção de suas respectivas execuções.

Por fim, no que diz respeito ao processo AspectCompose, esta dissertação descreveu e exemplificou o seu uso apresentando empiricamente seu efeito. Portanto, um possível trabalho futuro é fazer uma avaliação mais precisa do processo através de métricas específicas com o intuito de comprovar sua eficácia. Após esta avaliação, outro possível direcionamento é investigar a aplicabilidade do processo em outros sistemas de *middleware*.

Referências Bibliográficas

BARDRAM, J. E (2004). Applications of ContextAware Computing in Hospital Work – Examples and Design Principles. In: ACM Symposium on Applied Computing, 19., 2004, Nicosia, Cyprus. **Anais do SAC 2004**. New York: ACM Press, 2004. p. 1574-1579.

BARESI, L.; HECKEL, R.; THONE, S.; VARRO, D. (2003). Modeling and validation of service oriented architectures: Application vs. Style. In: European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, 9. e 11., 2003, Helsinki, Finlandia. **Anais do ESEC/FSE 2003**. Nova Iorque: ACM Press, 2003. p. 68-77.

BLAIR, G. S.; COULSON, G.; ROBIN P. e PAPATHOMAS, M. (1998). An Architecture for Next Generation Middleware. In: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, 1998, Londres, Inglaterra. **Anais da Middleware'98**. Springer-Verlag, 1998. p. 191-296.

BLAIR, G. S.; COULSON, G.; ANDERSEN, A.; BLAIR, L.; CLARKE, M.; COSTA, F.; DURAN-LIMON, H.; FITZPATRICK, T.; JOHNSTON, L.; MOREIRA, R.; PARLAVANTZAS, N.; e SAIKOSKI, K. (2001). The Design and Implementation of Open ORB 2. **IEEE Distributed Systems On Line**, v. 2, n. 6, set. 2001.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H; SOMMERLAD, P.; STAL, M. (1996). **Pattern-Oriented Software Architecture: A System of Patterns**. Wiley, 1996, vol. 1, 476 p.

CARVALHO, W. V. de; FERNANDES, P.; TEIXEIRA, R.; ANDRADE, R. M. C. (2005). Mobile Adapter: Uma abordagem para a construção de Mobile Application Servers adaptativos utilizando as especificações CC/PP e UAProf. In: Seminário Integrado de Software e Hardware, 32, 2005. São Leopoldo, RS, Brasil. **Anais do CSBC 2005**. Porto Alegre: SBC, 2005. p. 1914-1929.

CARVALHO, W. V.; ANDRADE, R. M. C. (2006). Uma Proposta para a Geração Semi-automática de Aplicações Adaptativas para Dispositivos Móveis. In: Simpósio Brasileiro de Engenharia de Software, 20, 2006. Florianópolis, SC, Brasil. **Anais do SBES 2006**.

CHEN, H.; PERICH, F., FININ, T.; JOSHI, A. (2004). SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. In: International Conference on Mobile and Ubiquitous Systems: Networking and Services, 1, 2004. Boston, MA, Estados Unidos. **Anais da MobiQuitous 2004**.

CHUNG, P. E.; HUANG, Y.; YAJNIK, S.; LIANG, D.; SHIH, J. C.; WANG, C-Y.; WANG Y-M. (1997). DCOM and CORBA Side by Side, Step by Step, and Layer by Layer. Disponível em: <http://research.microsoft.com/~ymwang/papers/HTML/DCOMnCORBA/S.html>. Acesso em: 03 out. 2006.

COSTA, C.G.A. (2001). **Desenvolvimnto e Avaliação Tecnológica de um Sistema de Prontuário Eletrônico do Paciente, Baseado no Paradigma da Word Wide Web e da Engenharia de Software**. 2001, 288f, Dissertação (Mestrado em Engenharia Elétrica) - Universidade Estadual de Campinas, Campinas, SP, Brasil.

COSTA, M. A. S. (2004). **Um Modelo de Middleware Adaptativo**. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Pernambuco, Recife, PE, Brasil, 2004.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. (2005). **Distributed Systems: Concepts and Design**. Addison Wesley, 4 ed., v. 1, 2005, p. 944.

DEY, A. K.; ABOWD, G. D. (1999). Towards a Better Understanding of Context and Context-Awareness. Relatório Técnico GIT-GVU-99-22, College of Computing, Georgia Institute of Technology, set. 1999.

DEY, A. K. (2000). **Providing Architectural Support for Building Context-Aware Applications**. 2000. 188f. Tese (Ph.D em Ciência da Computação) - Instituto de Tecnologia da Geórgia, Atlanta, Georgia, Estados Unidos, 2000.

DEY, A. K (2001). Understanding and Using Context. **Personal and Ubiquitous Computing**, v. 5, n.1, p. 4-7, fev. 2001.

DIJKSTRA, E. W. (1974). On the role of scientific thought. **Selected writings on Computing: A Personal Perspective**, Springer-Verlag New York, Inc., p. 60–66, ISBN 0-387-90652-5.

ENDLER, M.; SILVA, F. S. (2001). Desenvolvendo Software Adaptável para Computação Móvel. In: Workshop de Comunicação sem Fio e Computação Móvel, 3., 2001. Recife, PE, Brasil. **Anais do III Workshop de Comunicação sem Fio e Computação Móvel**. 2001. p. 93-101.

FAVELA, J.; RODRÍGUEZ, M.; PRECIADO, A. and GONZÁLEZ, V. M. (2004). Integrating Context-Aware Public Displays Into a Mobile Hospital Information System. **IEEE Transactions On Information Technology In Biomedicine**. v. 8, n. 3, set. 2004.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. (1994). **Design Patterns Elements of Reusable Object-Oriented Software**. Addison-Wesley, v. 1, 1994.

GARLAN, D; Siewiorek, D.; Smailagic, A.; Steenkiste, P. (2002). Project Aura: Towards Distraction-Free Pervasive Computing. **IEEE Pervasive Computing**, edição especial Integrated Pervasive Computing Environments, v. 21, n. 2, p. 22-31, 2002.

GEIHS, K. (2001). Middleware Challenges Ahead. **IEEE Computer Magazine**, 34 ed., v. 6, p. 24-31, jun. 2001.

GRIMM, R.; DAVIS, J.; HENDRICKSON, B.; LEMAR, E.; MACBETH, A.; SWANSON, S.; ANDERSON, T.; BERSHAD, B.; BORRIELLO, G.; GRIBBLE, S.; WETHERAL, D. (2001). Systems Directions for Pervasive Computing. In: Workshop on Hot Topics in Operating Systems, 8, 2001. **Anais do HotOS-VIII**.

GRIMM, R.; DAVIS, J.; LEMAR, E.; MACBETH, A.; SWANSON, S.; GRIBBLE, S.; ANDERSON, T.; BERSHAD, B.; BORRIELLO, G.; WETHERAL, D. (2001). Programming for Pervasive Computing Environments. Relatório Tecnico, n. UW-CSE 01-06-01, Universidade de Washington, Junho 2001.

GRIMM, R.; DAVIS, J.; LEMAR, E.; MACBETH, A.; SWANSON, S.; ANDERSON, T.; BERSHAD, B.; BORRIELLO, G.; GRIBBLE, S.; WETHERALL, D. (2004). System Support for Pervasive Applications. **ACM Transactions on Computer Systems**, v. 22, ed. 4, p. 421-486, Novembro 2004.

HAYTON, R.; HEBERT, A.; DONALDSON, D. (1998). FlexiNet: A Flexible Component Oriented Middleware System. In: ACM SIGOPS European Workshop on Support for Composing Distributed Applications, 8., 1998, Sintra, Portugal. **Anais da 8ª ACM SIGOPS European Workshop on Support for Composing Distributed Applications.** 1998. p. 17-24.

HENRICKSEN, K.; INDULSKA, J.; RAKOTONIRAINY, A. (2002). Modeling Context Information in Pervasive Computing Systems. In: International Conference on Pervasive Computing, 1, 2002. Lecture Notes in Computer Science, v. 2414, p. 167-180, Springer, Agosto 2002.

JAIN, R. K. (1992). **The Art of Computer Systems Performance Analysis: Techniques of Experimental Design, Measurement, Simulation and Modeling**. Wiley, 1992.

JURIC, M. B.; ROZMAN, I.; NASH, S. (2000). Java 2 Distributed Object Middleware Performance Analysis and Optimization. **ACM SIGPLAN Notices**, v. 35, n. 8, p. 31-40, Agosto 2000.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C. V.; LOINGTIER, J.; IRWIN, J. (1997). Aspect-Oriented Programming. In: European Conference on Object-Oriented Programming. 11., 1997, Jyväskylä, Finlandia. **Anais da 11ª European Conference on Object-Oriented Programming**. Springer-Verlag, Lecture Notes in Computer Science, v. 1241, p. 220-242, jun. 1997.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. (2001). An Overview of AspectJ. In: European Conference on Object-Oriented Programming, 15., 2001, Budapest, Hungary. **Anais da 15ª European Conference on Object-Oriented Programming**. Springer-Verlag, Lecture Notes in Computer Science, v. 2072, p. 327-353, jun. 2001.

KIM, Y.; KIM, E.; KIM, J.; SONG, E., KO, I. (2006). Ontology Based Software Reconfiguration in a Ubiquitous Computing Environment. In: IEEE International Conference on Computer and Information Technology, 6, 2006. **Anais do CIT'06**, p. 260, v. 00, 2006.

KON, F.; ROMÁN, M.; LIU, P.; MAO, J.; YAMANE, T.; MAGALHÃES, L. C.; and CAMPBELL, R. H. (2000). Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In: IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). 2000, New York. **Anais da Middleware'00**. n. 1795, 2000, p. 121-143.

KON, F., CAMPBELL, R.; MICKUNAS, M. D.; NAHRSTEDT, K.; BALLESTEROS, F. J. (2000). 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In: IEEE International Symposium on High Performance Distributed Computing. 9, 2000. **Anais do HPDC 2000**.

LIEBERHERR, K. (1995). Workshop on Adaptable and Adaptive Software. In: Conference on Object Oriented Programming Systems Languages and Applications. 10, 1995. **Anais da OOPSLA'95**. p.149-154.

MAES, P. (1987). Concepts and Experiments in Computation Reflection. ACM Special Interest Group on Programming Languages (SIGPLAN) Notices, 22 ed., v. 12, p. 147-155, dez. 1987.

MCKINLEY, P. K.; SADJADI, S. M.; KASTEN, E. P.; CHENG, B. H. (2004). A Taxonomy of Compositional Adaptation. Relatório Técnico MSU-CSE-04-17, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, 2004. Disponível em: <http://www.cs.fiu.edu/~sadjadi/Publications/CompositionalAdaptationTaxonomy-TechRep.pdf>. Acesso em: 10 jan. 2006.

MCKINLEY, P. K.; SADJADI, S. M.; KASTEN, E. P.; CHENG, B. H. (2004). Composing Adaptive Software. **IEEE Computer Magazine**, v. 37, n. 7, p. 56-64, jul. 2004.

OMG WEBSITE – Common Object Request Broker Architecture Specification. Disponível em: <http://www.omg.org/technology/documents/corba_spec_catalog.htm>. Acesso em: 03 out. 2006.

PASCOE, J (1998). Adding generic contextual capabilities to wearable computers. In: IEEE International Symposium on Wearable Computers. 2., 1998. **Anais da 2ª IEEE International Symposium on Wearable Computers**. Washington: IEEE Computer Society Press, out. 1998. p. 92-99.

PEREIRA, F. M. Q.; VALENTE, M. T. O.; BIGONHA, R. and BIGONHA, M. (2006). Arcademis: a Framework for Object-Oriented Communication Middleware Development. **ACM Software-Practice & Experience**. v 36 n. 5, p. 495-512, 2006.

ROMÁN, M.; KON, F. e CAMPBELL, R. (2001). Reflective Middleware: From Your Desk to Your Hand. 5 ed., **IEEE Distributed Systems Online**, v. 2, 2001.

ROMÁN, M.; HESS, C. K.; CERQUEIRA, R.; RANGANATHAN, A.; CAMPBELL, R.; NAHRSTEDT, K. (2002). Gaia: A Middleware Infrastructure to Enable Active Spaces. **IEEE Pervasive Computing**, p. 74-83, Outubro 2002.

RMI WEBSITE. Java™ Remote Method Invocation. Disponível em: <http://java.sun.com/j2se/1.4.2/docs/guide/rmi/>. Acesso em: 03 out. 2005.

SACRAMENTO, V.; ENDLER, M.; RUBINSZTEJN, H. K.; LIMA, L. S.; GONÇALVES, K.; NASCIMENTO, F. N.; E BUENO, G. A. (2004). MoCA: A middleware for developing collaborative applications for mobile users. **IEEE Distributed Systems Online**, v.5, n. 10, Outubro de 2004.

SADJADI, S. M.; MCKINLEY, P. K. (2003). A survey of adaptive middleware. Relatório Técnico MSU-CSE-03-35, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, dez. 2003. Disponível em: <http://35.9.20.31/~sadjadis/Publications/AdaptiveMiddlewareSurvey.pdf>. Acesso em: 10 jan. 2006.

SCHILIT, B.; ADAMS, N.; WANT, R. (1994). Context-aware computing applications. In: IEEE Workshop on Mobile Computing Systems and Applications. 1., 1994, Santa Cruz, California. **Anais do 1º IEEE Workshop on Mobile Computing Systems and Applications**. IEEE Computer Society Press, 1994. p 85-90.

SCHILIT, B.; THEIMER, M. (1994). Disseminating Active Map Information to Mobile Hosts. **IEEE Network Special Issue on Personal Nomadic Communications**. v. 8, n. 5, p. 22-32, set. 1994.

SCHMIDT, D.; CLEELAND, C. (1999). Applying Patterns to Develop Extensible ORB Middleware. **IEEE Communications Magazine**. v. 37, n. 4, p. 54-63, abr. 1999.

SCHMIDT, D. C.; STAL, M.; ROHNERT, H.; BUSCHMANN, F. (2000). **Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects**. Wiley & Sons, 2000, vol. 2, 666 p.

SCHMIDT, D. C. (2002). Middleware for real-time and embedded systems. **In Communications of the ACM**. v. 45, p. 43-48, jun. 2002.

SOARES, S.; BORBA, P. (2002). AspectJ - Programação Orientada a Aspectos em Java. Tutorial no SBLP 2002, In: Simpósio Brasileiro de Linguagens de Programação. 6., 2002, Rio de Janeiro, RJ, Brasil. **Anais do VI Simpósio Brasileiro de Linguagens de Programação**. Rio de Janeiro: SBC, jun. 2002. p. 39-55.

STREITZ, N.; NIXON, P. (2005). The Disappearing Computer. **Communication of the ACM**, v. 48, n. 3, 2005, p. 32-35.

TEKINERDOGAN, B.; AKSIT, M. (1997). Adaptability in Object-Oriented Software Development Workshop Report. In: Special Issues in Object-Oriented Programming, M. Muhlhauser (ed.). dpunkt-Verlag, Heidelberg, Germany, p. 7-12.

TAMMA, V.; CRANEFIELD, S.; FININ, T.; WILLMOTT, S. (2005). **Ontologies for Agents: Theory and Experiences**. Whitestein Series in Software Agent Technologies and Autonomic Computing, Birkhäuser Basel, ed. 1, 2005.

TIRELO, F.; BIGONHA, R. S.; BIGONHA, M. A. A.; VALENTE, M. T. O. M. (2004). Desenvolvimento de Software Orientado por Aspectos. In: Jornada de Atualização em Informática. 23., 2004, Salvador, BA, Brasil. **Anais do XXIV Congresso Sociedade Brasileira de Computação**. Porto Alegre: SBC, ed.1, v. 2, p. 57-96, jul. 2004.

VAYSSE, G.; ANDRÉ, F; BUISSON, J (2005). Using Aspects for Integrating a Middleware for Dynamic Adaptation. In: International Middleware Conference. 6., 2005, Grenoble, França. **Anais da Midleware'05**. dez. 2005.

VITERBO FILHO, J.; SACRAMENTO, V.; ROCHA, R. C. A.; ENDLER, M. (2006). MoCA: Uma Arquitetura para o Desenvolvimento de Aplicações Sensíveis ao Contexto para Dispositivos Móveis. In: Simpósio Brasileiro de Redes de Computadores, Sessão de Ferramentas. 5, 2006. Anais do SBRC 2006.

WANG, A.J.A.; QIAN, K. (2005). Component-Oriented Programming. **John Wiley & Sons, Inc**, 1 ed., 2005, p. 336.

WEISER, M. (1991). The Computer for the 21st Century. **Scientific American**. v. 265, n. 3, p. 94-104, fev. 1991.

WEISER, M. (1993). Some Computer Science Issues in Ubiquitous Computing. **Communications of the ACM**. v. 6, n. 7, p. 75-84, mar. 1993.

WEISER, M.; BROWN, J. (1996). Designing Calm Technology. **PowerGrid Journal**. v. 1, n. 1, jul. 1996.