Cristiano Amaral Maffort

Aspectos para Construção de Aplicações Distribuídas

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Informática.

Belo Horizonte

Junho de 2007

PUC Minas
Programa de Pós-graduação em Informática

FOLHA DE APROVAÇÃO

Aspectos para Construção de Aplicações Distribuídas

CRISTIANO AMARAL MAFFORT

Dissertação defendida e aprovada pela seguinte banca examinadora:

Prof. Marco Túlio de Oliveira Valente - Orientador (PUC Minas)
Doutor em ciência da Computação – UFMG

Prof. Roberto da Silva Bigonha (UFMG)
Doutor em Ciência da Computação - University of California, Los Angeles

Prof. Mark Alan Junho Song (PUC Minas)
Doutor em ciência da Computação – UFMG

Belo Horizonte, 12 de junho de 2007.

FICHA CATALOGRÁFICA
Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Maffort, Cristiano Amaral
M187a Aspectos para construção de aplicações distribuídas. / Cristiano Amaral Maffort. – Belo Horizonte, 2007.
VI; 88f.: il.

Orientador: Prof. Dr. Marco Túlio de Oliveira Valente
Dissertação (Mestrado) – Pontifícia Universidade Católica de Minas Gerais, Programa de Pós-Graduação em Informática, Belo Horizonte.
Bibliografia.

1. Processamento eletrônico de dados – Processamento distribuído. 2. Arquitetura de software. 3. Middleware. 4. Engenharia de software. I. Valente, Marco Túlio de Oliveira. II. Pontifícia Universidade Católica de Minas Gerais, Programa de Pós-Graduação em Informática. III. Título.

CDU: 681.3.022

Bibliotecária : Erica Fruk Guelfi – CRB 6/2068

Resumo

Plataformas de *middleware* são largamente empregadas por engenheiros de *software* para tornar mais produtivo o desenvolvimento de aplicações distribuídas. No entanto, os sistemas atuais de *middleware* são, em geral, invasivos, pervasivos e transversais ao código de negócio de aplicações distribuídas. Nesta dissertação, apresenta-se um sistema de apoio à programação distribuída, chamado DAJ, que encapsula em aspectos os serviços de distribuição providos por duas plataformas de *middleware*: Java RMI e Java IDL. O sistema proposto é suficientemente flexível e expressivo de forma a permitir o uso das principais abstrações providas por estas duas tecnologias de *middleware*, além de suportar diversas arquiteturas e configurações típicas de sistemas distribuídos. Para alcançar esse objetivo, o sistema DAJ se beneficia da sinergia gerada pela combinação de três tecnologias: aspectos, linguagens de domínio específico e geração e transformação de código.

DAJ fornece suporte às seguintes funcionalidades:

- Chamadas síncronas de métodos remotos com passagem de parâmetros com semântica tanto por serialização quanto por referência remota.
- Suporte a diversos cenários de distribuição, os quais são descritos por meio de descritores de distribuição. Tais descritores descrevem o papel desempenhado pelas classes de um sistema distribuído que utiliza uma determinada plataforma de *middleware*.
- Geração automática de código de distribuição por meio de uma ferramenta incluída no sistema DAJ, chamada `dajc`, a qual isenta o projetista de uma aplicação distribuída de dominar detalhes e convenções de codificação requeridos por plataformas de *middleware* e de dominar detalhes de programação orientada por aspectos. Desta forma, DAJ permite que esse projetista concentre-se na implementação dos requisitos funcionais de sua aplicação.

Com o intuito de ilustrar o poder de expressão da solução proposta por DAJ, apresenta-se nesta dissertação três estudos de caso realizados com o apoio do sistema. Valendo-se dos recursos oferecidos por DAJ, foi possível encapsular todo código de distribuição desses sistemas. Com isso, as classes de negócio das aplicações permaneceram livres de código entrelaçado demandado pelas plataformas de *middleware* subjacentes.

Abstract

Software engineers often rely on middleware platforms to design and implement distributed systems. However, middleware functionality is usually invasive, pervasive and tangled with business-specific concerns. In this master thesis, we describe a system, called DAJ, that encapsulates in aspects distribution services provided by two middleware platforms: Java RMI and Java IDL. The proposed system is flexible and has enough expressive power to support the main abstractions provided by these two middleware technologies. Moreover, DAJ provides support to different software architectures employed in distributed systems implementation. To reach its objectives, DAJ relies on the synergistic combination of three technologies: aspects, domain-specific languages and generative programming.

DAJ provides support to the following functionalities:

- Synchronous remote calls using call by-serialization and call by-remote-reference semantics.
- Different distribution and software architecture scenarios, which are described by means of distribution descriptors. Such descriptors are used to declare the role that plain Java classes and objects play in a particular distributed system.
- Automatic generation of distribution code by a tool included in the DAJ system. Thus, distributed application developers do not need to dominate details and conventions required by middleware platforms and concepts of aspect-oriented programming. Instead, DAJ users can focus on the functional concerns of the system they are building.

In order to illustrate the expressiveness of the proposed solution, we have evaluate the use of DAJ in three legacy distributed applications. Using the resources provided by DAJ, it was possible to encapsulate all distribution concerns of these systems. In this way, business classes of the evaluated systems have remained agnostic to underlying middleware code.

À Amanda,
minha maior riqueza.

Agradecimentos

Agradecer não é uma tarefa simples. Sempre há a possibilidade de esquecer alguém. Desta forma, começarei por agradecer a todos aqueles que me ajudaram, direta ou indiretamente. Assim ninguém estará completamente esquecido, na hipótese de alguém que me ajudou não ser citado.

Feito isso, gostaria de agradecer, de forma especial, a algumas pessoas que colaboraram para que eu chegasse a este ponto na minha "caminhada". Agradeço aos meus pais por permitirem que eu pudesse caminhar em passos firmes. Meu Pai (sempre presente) deve estar orgulhoso agora. À Sra Carla Lazarotti, por me mostrar que existem vários caminhos e por me indicar aquele pelo qual eu encontraria a minha aptidão profissional. À Sra Inês e Sra Laudieme, por terem fornecido subsídios, ainda que indiretamente, para que eu me mantivesse nesse caminho. Ao Prof. Paulo Sérgio, pelas lições aprendidas, em especial por me mostrar que o melhor caminho é sempre aquele percorrido de forma honrosa.

Ao Prof. Marco Túlio, por ter confiado em mim e ter me direcionado na estrada que me conduziria a este ponto da caminhada. Pela dedicação, disponibilidade, responsabilidade e compreensão. Por ter quebrado as "convenções protocolares" entre mestre e discípulo, me apoiando em assuntos que vão além do âmbito acadêmico.

Aos amigos do mestrado, por compartilharem seus conhecimentos, experiências, inseguranças, ansiedades, o doce de leite e a pizza. Em especial a Anne, Carol, José Geraldo, Leonardo, Michelle, Pedro e Sandro.

Aos professores do mestrado, pelo incentivo e orientação para formação do conhecimento. Em especial a Raquel Mini, Lucila Ishitani, Silvio Jamil, Mark Alan e Clodoveu Davis.

À Giovanna, secretária acadêmica, pela atenciosidade e carinho.

Aos amigos da PUC Minas, em especial aqueles do Apoio às Coordenações de Cursos.

Ao meu irmão, por sempre ter me apoiado. À Amanda, por fornecer cor e luz à minha vida. À Fabiana, pelo amor, paciência e inspiração. À Patrícia, por estar sempre presente.

A todos vocês, o meu sincero agradecimento e gratidão.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

A evolução nas tecnologias de redes de computadores, associada ao crescimento acelerado do número de usuários da Internet, alavancou a demanda por aplicações distribuídas de vários níveis de complexidade, afetando desde simples aplicações para troca de mensagens a complexos sistemas bancários.

A diversidade de cenários em que uma aplicação distribuída pode ser executada é considerável, incluindo ambientes heterogêneos, constituídos por computadores que apresentam arquiteturas e sistemas operacionais diferentes, e aplicações implementadas em diferentes linguagens de programação. Isso torna o desenvolvimento de aplicações distribuídas consideravelmente mais complexo do que o desenvolvimento de aplicações centralizadas [Emm00]. Com o objetivo de amenizar os problemas oriundos desse ambiente heterogêneo, foi proposta a introdução de uma camada de *software*, chamada *middleware*, entre as aplicações e seus sistemas operacionais subjacentes, fornecendo aos desenvolvedores de aplicações distribuídas uma interface de programação de alto nível [CK02], a qual objetiva ocultar as dificuldades inerentes a esse ambiente. Com o sucesso da tecnologia, diversas plataformas de *middleware* foram desenvolvidas para apoiar a implementação de sistemas distribuídos. Dentre elas, destacam-se as plataformas Java RMI [WRW96], CORBA [Gro02], .NET *Remoting* [OH01], JAX-RPC [JAX] e Apache Axis [AXI].

Assim, plataformas de *middleware* são largamente empregadas por engenheiros de *software* para simplificar e tornar mais produtivo o desenvolvimento de aplicações distribuídas. Basicamente, estes sistemas encapsulam diversos detalhes inerentes à programação em ambientes de rede, incluindo protocolos de comunicação, heterogeneidade de arquiteturas, *marshalling* e *unmarshalling* de dados, sincronização, localização de serviços, etc.

No entanto, os sistemas atuais de *middleware* caracterizam-se por serem, em ge-

ral, invasivos, pervasivos e transversais ao código de negócio de aplicações distribuídas [GFS+05, TUSF03, POM03, SBL06]. Desta forma, o grau de modularidade de sistemas distribuídos baseados em arquiteturas de *middleware* está começando a tornar-se complexo devido às interações de vários interesses transversais impostos por uma grande variedade de requisitos de aplicações [ZJ04]. Normalmente, usuários de um sistema de *middleware* devem seguir uma série de convenções de programação, como por exemplo estender classes internas da plataforma, tratar exceções geradas pelo sistema, codificar interfaces remotas, invocar geradores de *stubs*, etc. Como resultado, sistemas distribuídos baseados em plataformas de *middleware* não apresentam graus esperados de reusabilidade, separação de interesses e modularidade, o que dificulta o desenvolvimento, manutenção e evolução dos mesmos.

Este cenário é paradoxal, pois a variedade de funcionalidades, a inflexibilidade, a característica monolítica e o grau de entrelaçamento e espalhamento de código imposto pelas plataformas de *middleware* atuais acaba por dificultar a utilização desses sistemas, contrariando seu principal objetivo, isto é, simplificar o desenvolvimento de aplicações distribuídas. Além disso, a interface de programação adotada por plataformas de *middleware* para desenvolvimento de aplicações distribuídas em Java, dificulta a utilização simultânea de múltiplas plataformas de *middleware* na mesma aplicação. Isso ocorre em razão, por exemplo, de tais interfaces requererem que classes de negócio estendam classes internas da plataforma de *middleware*. Como Java não suporta herança múltipla, limita-se nesse caso o uso de apenas uma plataforma de *middleware*.

1.2 Objetivos

Nesta dissertação pretende-se investigar uma solução para modularização, em aspectos, de interesses de distribuição requeridos por plataformas de *middleware* utilizadas no desenvolvimento de aplicações distribuídas em Java. Desta forma, todo o código de distribuição será agrupado em módulos independentes, mantendo o núcleo da aplicação livre de qualquer espalhamento ou entrelaçamento de código. Além disso, a solução proposta deverá atender aos seguintes requisitos:

- **Geração automática de código de distribuição**: a solução proposta deve incluir uma ferramenta que produzirá aspectos responsáveis por introduzir distribuição em uma determinada aplicação alvo. Esta ferramenta isentará o projetista da aplicação de dominar detalhes e convenções de codificação requeridos por plataformas de *middleware*. Além disso, esse projetista também estará isento de dominar conceitos de programação orientada por aspectos, já que não será responsável por codificar

os aspectos responsáveis por modularizar o código de distribuição. Desta forma, pretende-se que o projetista concentre-se exclusivamente no desenvolvimento do código funcional da aplicação.

- **Compatibilidade com mais de uma tecnologia de *middleware***: a ferramenta de geração de código deve produzir código para as duas principais plataformas de *middleware* nativas de ambientes de desenvolvimento Java: Java RMI e Java IDL.
- **Configurabilidade dos cenários de distribuição**: a solução proposta deve ser compatível com diversas arquiteturas de *software* empregadas na construção de aplicações distribuídas.
- **Flexibilidade de transmitir objetos tanto por serialização quanto por referência remota**: pretende-se que a solução proposta permita, em chamadas remotas de métodos, a passagem/retorno de objetos tanto por serialização quanto por referência remota. O recurso de passagem/retorno de referências para objetos remotos é normalmente usado por aplicações distribuídas interessadas em serem notificadas, via *callback*, da ocorrência de determinados eventos [Fra98].

1.3 Visão Geral da Solução Proposta

Tendo em vista os problemas decorrentes da utilização das atuais plataformas de *middleware* relatados na Seção 1.1, esta dissertação apresenta um sistema de apoio à programação distribuída que isola a funcionalidade de distribuição da lógica de negócio de aplicações implementadas em Java. Desta forma, pretende-se que o desenvolvedor da aplicação concentre seus esforços apenas nos requisitos funcionais da mesma, mantendo o seu núcleo livre de código de distribuição que se entrelaça e se espalha pelos módulos do sistema.

O sistema proposto, chamado DAJ (*Distribution Aspects in Java*) [MV06a], inclui um sistema para implementação do interesse de distribuição, o qual define classes e aspectos que encapsulam e modularizam, de forma independente da aplicação alvo, diversos detalhes de programação usualmente requeridos por plataformas de *middleware*. Além disso, o sistema DAJ inclui uma ferramenta de geração e transformação de código, a qual é responsável por gerar aspectos que especializam os aspectos abstratos do sistema. Desta forma, a transformação da aplicação alvo em uma aplicação distribuída é realizada automaticamente por DAJ.

A ferramenta de geração de código de DAJ produz código para as duas principais plataformas de *middleware* nativas de ambientes de desenvolvimento Java: Java RMI e

Java IDL. Java RMI é um *middleware* fortemente integrado à linguagem Java, seguindo o modelo de programação e o sistema de tipos adotados por essa linguagem. A plataforma Java IDL é uma implementação do padrão CORBA [Sie00], permitindo, portanto, que aplicações distribuídas implementadas em Java acessem sistemas desenvolvidos em outras linguagens de programação.

DAJ fornece ainda suporte simultâneo a estas duas plataformas de *middleware*, isto é, permite-se que um objeto remoto receba chamadas provenientes tanto de clientes codificados em Java RMI como em Java IDL. DAJ também fornece, para Java RMI e Java IDL, a possibilidade de passar objetos tanto por serialização quanto por meio de uma semântica de referência remota.

Em DAJ, classes de negócio, que compõem o núcleo da aplicação, não precisam seguir quaisquer convenções de codificação, ou seja, não precisam estender classes ou implementar interfaces oriundas das plataformas de *middleware*, não precisam implementar métodos `get` e `set`, não precisam ativar ou tratar exceções pré-definidas, não precisam se preocupar com a instanciação, ativação e registro de objetos remotos, etc.

Em DAJ, descritores de distribuição são usados para configurar as características de distribuição desejadas pelo usuário, de acordo com o ambiente computacional em que a aplicação será inserida. Nestes descritores, é possível configurar as propriedades de um objeto remoto, incluindo a interface implementada pelo objeto, a classe que implementa a lógica de negócio, a plataforma de *middleware* que será usada para comunicação com o objeto remoto e informações necessárias para localizar e registrar o objeto remoto junto a um servidor de nomes. A ferramenta de geração de código de DAJ, a partir das especificações contidas em descritores de distribuição, se encarrega de gerar automaticamente aspectos, implementados em AspectJ, e classes que modularizam o código de distribuição requerido pela plataforma de *middleware* especificada.

Com a finalidade de validar e demonstrar as potencialidades do sistema proposto, foram utilizados, além de um exemplo motivador, três estudos de caso que fazem uso das principais abstrações providas por sistemas de *middleware* orientados por objetos. O poder de expressão de DAJ foi demonstrado através da diversidade de cenários nos quais os estudos de casos foram aplicados.

1.4 Estrutura da Dissertação

O restante desta dissertação está organizado da seguinte forma. No Capítulo 2, realiza-se uma revisão bibliográfica, onde são descritas as principais plataformas de *middleware* para construção de aplicações distribuídas em Java. Além disso, são apresentados traba-

lhos relacionados com o projeto de DAJ, assim como conceitos de Programação Orientada por Aspectos.

No Capítulo 3, descreve-se a interface de programação e a arquitetura interna de DAJ. Neste capítulo, são abordados em detalhes tanto os aspectos necessários para introduzir o interesse de distribuição em uma aplicação alvo, quanto as adaptações necessárias para que múltiplas plataformas de *middleware* sejam utilizadas em uma mesma aplicação.

No Capítulo 4, demonstram-se as potencialidades de DAJ através da utilização do sistema para introdução do interesse de distribuição em três aplicações. Essas aplicações incluem um sistema para controle de ações negociadas em uma bolsa de valores, uma aplicação usada por cidadãos para enviar reclamações sobre restaurantes e similares a órgãos públicos responsáveis pela vigilância sanitária destes estabelecimentos e uma aplicação para controle de uma biblioteca.

O Capítulo 5 conclui esta dissertação, apresentando uma avaliação de DAJ. Além disso, esse capítulo compara o sistema proposto com soluções relacionadas, buscando ressaltar os pontos positivos e negativos da utilização de DAJ. Por fim, são apresentadas possíveis linhas de pesquisa que podem ser desenvolvidas como trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

2.1 *Middleware*

Middleware, em computação distribuída, designa uma camada de *software* posicionada entre uma aplicação e o sistema operacional, que objetiva ocultar do engenheiro de *software* as complexidades do desenvolvimento de aplicações distribuídas, em especial àquelas oriundas da heterogeneidade do ambiente em que essas aplicações são utilizadas. Para tanto, as plataformas de *middleware* oferecem uma interface uniforme de programação aos desenvolvedores de aplicações distribuídas, permitindo que a comunicação entre os componentes da aplicação seja realizada de forma transparente para o desenvolvedor. A Figura 2.1 ilustra como plataformas de *middleware* orientadas por objeto integram aplicações distribuídas em ambientes heterogêneos.

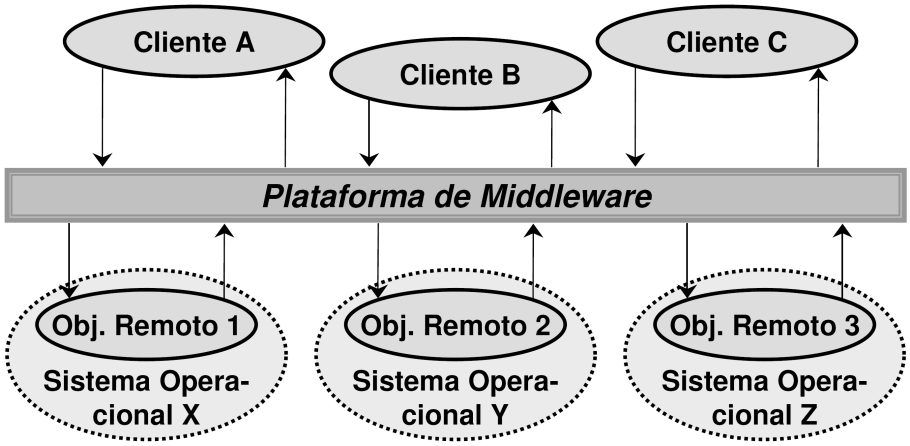

Figura 2.1: *Middleware* como mediador em ambientes heterogêneos.

Os benefícios trazidos pela utilização de plataformas de *middleware* no desenvolvimento de sistemas distribuídos despertaram o interesse de consórcios formados por gran-

des companhias, as quais vislumbraram criar um padrão para arquiteturas distribuídas. A especificação da arquitetura CORBA [Gro02, Vin97, Vin98] surgiu como resultado dos esforços de um desses consórcios, chamado OMG (*Object Management Group*) [OMGa].

Diversas outras plataformas de *middleware* surgiram logo depois. Algumas delas também patrocinadas por grandes companhias: como COM/DCOM [PS98], Java RMI [WRW96] e Apache Axis [AXI]. DCOM é o modelo de objetos distribuídos de propriedade da Microsoft, permitindo interoperabilidade entre programas desenvolvidos em diferentes linguagens. Entretanto, este modelo é incompatível com o ambiente de desenvolvimento Java e, portanto, não será detalhado nessa dissertação. Java RMI estende o modelo de objetos da linguagem Java, por isso é utilizado com freqüência por aplicações distribuídas cujos módulos são integralmente implementados nessa linguagem. Apache Axis é um *middleware* para desenvolvimento de sistemas distribuídos baseado em Serviços Web [ACKM04].

No restante desta seção, abordam-se as duas principais plataformas de *middleware* para desenvolvimento de aplicações distribuídas em Java: Java RMI e Java IDL.

2.1.1 Java RMI

Java RMI é uma plataforma de *middleware* para desenvolvimento de sistemas distribuídos em Java. Ela permite que objetos sendo executados em uma Máquina Virtual Java possam chamar métodos de objetos em execução em outra JVM, ou seja, um cliente pode invocar métodos de servidores localizados em um espaço de endereçamento diferente.

Java RMI é completamente integrado à linguagem Java, sendo, por isso, fortemente dependente do modelo de programação adotado por essa linguagem. Esse grau de dependência, associado aos benefícios oriundos do uso da JVM, como portabilidade e segurança, faz com que inexistam problemas de heterogeneidade entre plataformas, criando um ambiente homogêneo de desenvolvimento de aplicações distribuídas, que dispensa a utilização de outras tecnologias, como por exemplo de uma linguagem neutra para definição de interfaces, como ocorre, por exemplo, em CORBA. Essas características permitem que classes, *stubs* e interfaces para objetos remotos sejam distribuídos dinamicamente entre os nós da rede, fazendo com que os *bytecodes* dos objetos distribuídos não precisem ser pré-instalados no sistema de arquivos das aplicações clientes [Per04]. Além disso, a plataforma RMI também implementa detalhes de sincronização, concorrência e aproveita-se dos mecanismos de segurança nativos de Java. Esses mecanismos de segurança fornecem suporte à verificação automática de *bytecodes* e o controle do acesso à memória. Essas facilidades tornam mais simples o uso de objetos remotos, pois permite que esses objetos

possam ser acessados de uma forma muito semelhante a invocações realizadas localmente.

Pode-se distinguir dois tipos de aplicação em Java RMI: servidores e clientes. Servidores fornecem métodos que podem ser remotamente invocados. Já clientes invocam esses métodos. Apesar dessa distinção, *hosts* podem se comportar ao mesmo tempo como servidor e cliente para objetos diferentes.

Interface de programação

Java RMI disponibiliza às aplicações uma série de mecanismos que procuram tornar mais transparente as tarefas de localização, registro e ativação de objetos remotos. Para tanto, Java RMI possui um serviço de nomes, denominado de *registry*, no qual são armazenadas referências para objetos remotos.

Os clientes acessam o servidor de nomes com dois objetivos: localizar ou registrar objetos remotos. Essas tarefas são realizadas através da classe `Naming`, a qual oferece os métodos `lookup` e `bind` para localização e registro de objetos remotos, respectivamente. O método `lookup` é utilizado pela aplicação cliente para localização de objetos remotos, através da `URI` (*Uniform Resource Identifier*) do servidor de nomes, concatenada ao nome que identifica o objeto remoto neste servidor.

O método `bind` (ou `rebind`) permite que um determinado objeto remoto seja registrado em um servidor de nomes. Para tanto devem ser informados uma referência para o objeto remoto e um nome que identificará este objeto no servidor, o qual será usado para localização desse objeto. A Figura 2.2 mostra o relacionamento entre o servidor de nomes, objetos remotos e aplicações clientes.

Em um objeto remoto, nem todos os métodos são visíveis para serem invocados, por serem privados, por exemplo. Por isso, os métodos que podem ser invocados remotamente são definidos em uma interface particular, denominada interface remota, a qual contém a assinatura de todos os métodos que podem ser invocados sobre o objeto remoto, que deve implementar essa interface remota. A referência que o cliente obtém do serviço de nomes é do tipo dessa interface remota, portanto o cliente a utiliza para identificar os métodos que poderão ser invocados sobre o objeto remoto.

Arquitetura interna

Clientes de Java RMI efetuam chamadas sobre objetos remotos como se eles estivessem localizados no mesmo espaço de endereçamento dos objetos locais. Para fornecer essa abstração, uma nova camada é interposta entre a aplicação cliente e o objeto remoto. Desta forma, o cliente não referencia diretamente o objeto remoto, mas sim um representante

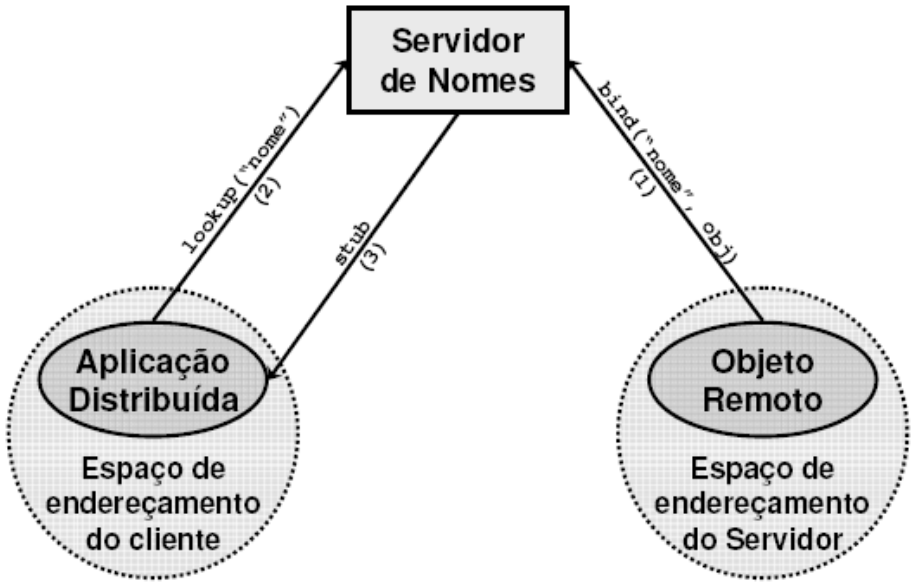

Figura 2.2: Interação entre servidor de nomes, objeto remoto e aplicação cliente

local do mesmo, denominado *stub*. O *stub*, por desempenhar papel de *proxy* [GHJV00], implementa a mesma interface que o objeto remoto, sendo, por isso, compatível com a referência remota utilizada pelo cliente. O *stub* é responsável por interceptar a chamada efetuada pelo cliente, redirecionando-a para o objeto remoto, de onde receberá os resultados do método invocado e os retornará para o cliente. Para tanto, o *stub* serializa os argumentos, assim como desserializa o resultado da chamada remota.

Java RMI fornece suporte a duas semânticas de passagem de parâmetros em chamadas remotas: por serialização e por referência remota [WRW96, CT04]. Na passagem por serialização, Java RMI utiliza o mecanismo de serialização de Java para converter o parâmetro, ou o retorno da chamada, em uma seqüência de *bytes* que poderá ser transmitida através da rede. Na passagem por referência remota, ou seja, quando uma referência para um objeto remoto é passada como parâmetro de uma chamada remota, seu *stub* é enviado como argumento. Em ambos os casos, o *stub* se encarrega de serializar os parâmetros passados por cópia, de enviar o *stub* dos parâmetros passados por referência remota e, também, de desserializar o resultado retornado pelo método invocado.

No lado do servidor, um outro componente denominado *skeleton* se encarrega de desserializar os argumentos do método invocado pelo cliente, de efetivamente chamar este método e, por fim, de enviar o resultado da chamada para o cliente.

O *stub* e o *skeleton* interagem com a camada de transporte, por onde trafegam os dados resultantes da comunicação entre cliente e servidor. A Figura 2.3 ilustra as relações entre os diversos componentes de uma aplicação distribuída usando Java RMI. De acordo

com essa figura, quando o cliente realiza uma chamada a um método remoto, ocorrem as seguintes operações:

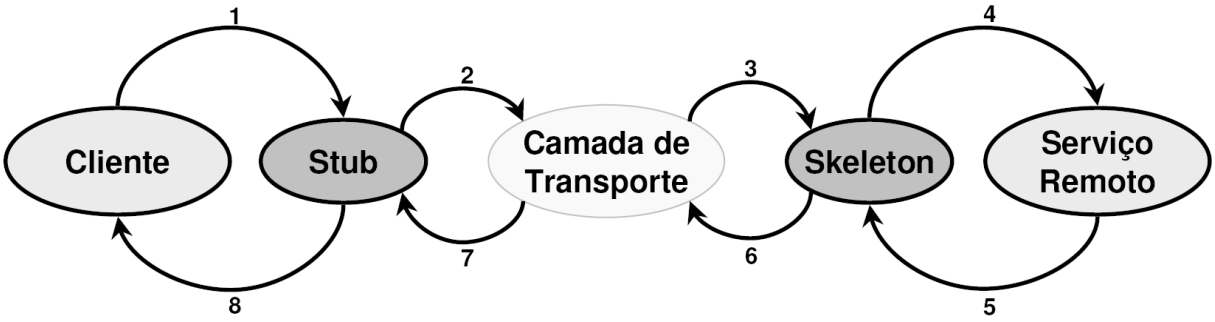

Figura 2.3: Interação entre componentes de Java RMI

1. O cliente efetua, de forma transparente, chamadas remotas de métodos sobre o *stub*.
2. Ao receber a chamada do cliente, o *stub* estabelece uma conexão com o *skeleton* no lado do servidor. Em seguida serializa e transmite os argumentos da chamada;
3. O *skeleton*, ao receber a solicitação de conexão, cria uma *thread* para tratar as chamadas remotas efetuadas pelo cliente e, em seguida, desserializa os argumentos enviados;
4. O método é efetivamente invocado sobre o objeto remoto;
5. O resultado da chamada é retornado para o *skeleton*;
6. O *skeleton* serializa o resultado da chamada e o transmite para o *stub*;
7. O *stub* desserializa o resultado enviado pelo *skeleton*;
8. Por fim, o resultado é retornado ao cliente que efetuou a chamada remota.

Plataformas de *middleware* baseadas em Java RMI

A partir de Java RMI, surgiram outras implementações mais sofisticadas com vistas a suprir deficiências desta plataforma de *middleware*, como por exemplo: Jini [Wal99], JNDI [JND] e EJB [EJB]. Jini fornece suporte a sistemas distribuídos capazes de se configurarem de forma autônoma. Já JNDI (*Java Naming and Directory Interface*) fornece uma interface comum para acesso a diversos tipos de serviços de nomes. Por fim, EJB (*Enterprise Java Beans*) fornece módulos que encapsulam a implementação de diversos requisitos não-funcionais, incluindo distribuição, segurança, persistência e controle de transações.

2.1.2 CORBA

A especificação CORBA (*Common Object Request Broker Architecture*) [Gro02] surgiu de um esforço iniciado em 1989 por um consórcio de empresas denominado OMG (*Object Management Group*) [OMGa]. Esse consórcio reúne projetistas e usuários de sistemas de *middleware*, com o objetivo de padronizar tecnologias para integração de objetos distribuídos segundo um modelo de programação orientado por objetos. CORBA é um sistema de *middleware* que oferece abstrações para o desenvolvimento desses tipos de sistemas, permitindo integrar diversas aplicações em ambientes heterogêneos [Vin98].

Aplicações distribuídas baseadas em CORBA são independentes tanto de linguagem de programação quanto de plataformas de *software* ou *hardware*. Diferindo assim de RMI, a qual apesar de ser independente de plataforma, é restrita ao ambiente Java.

CORBA não é um produto, mas sim uma especificação que define requisitos necessários para o desenvolvimento de plataformas de *middleware* orientadas por objetos. Assim, diferentes empresas implementam plataformas de *middleware* baseadas nessa especificação, as quais interoperam entre si de forma transparente para o usuário.

Arquitetura interna de CORBA

CORBA constitui parte de uma arquitetura para objetos distribuídos denominada OMA (*Object Management Architecture*) [OMG00]. A Figura 2.4 ilustra os principais elementos que fazem parte dessa arquitetura.

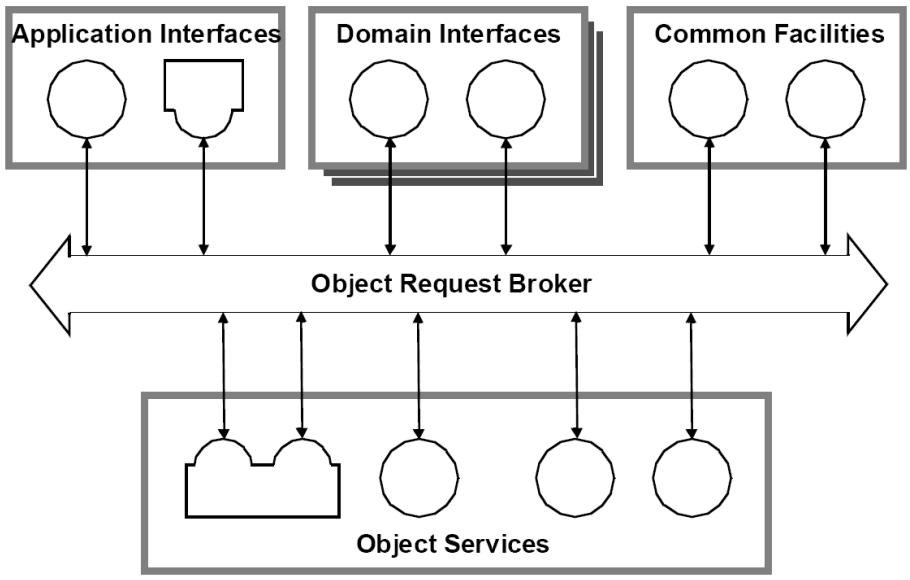

Figura 2.4: Arquitetura de gerenciamento de objetos da OMG [OMA]

Serviços de objeto (*Object Services*): representam funcionalidades de propósito geral que são fundamentais para o desenvolvimento de aplicações de objetos distribuídos baseadas em CORBA. Algumas dessas funcionalidades são descritas a seguir [CK02]:

- O *Serviço de Nomes* permite que objetos remotos possam ser registrados com nomes lógicos, através dos quais os clientes localizarão o serviço remoto.
- O *Serviço de Negociação* (*trader*) permite que objetos sejam localizados através de suas propriedades, ou seja, a localização do objeto pode ser feita a partir das categorias ou classificações de serviços que o objeto oferece, ao invés do processo de localização por nome como ocorre tradicionalmente.
- O *Serviço de Transações* fornece suporte a diversos modelos de transações e permite a execução de transações distribuídas envolvendo múltiplos objetos.
- O *Serviço de Persistência* fornece mecanismos para salvar em memória secundária o estado de objetos persistentes, de forma transparente às aplicações.
- O *Serviço de Controle de Concorrência* permite que múltiplos clientes acessem recursos compartilhados de forma segura.
- O *Serviço de Segurança* fornece mecanismos para identificação, autenticação, autorização, controle de acesso, auditoria e comunicação segura em ambientes distribuídos heterogêneos.
- O *Serviço de Eventos* fornece mecanismos de comunicação assíncrona através de canais para transmissão de eventos. Clientes se inscrevem para receber notificações de certos tipos de eventos. Quando o evento é publicado, o cliente recebe uma notificação através do canal que ele criou.

Facilidades Horizontais (*Common Facilities*): definem uma série de interfaces aplicáveis à maioria dos domínios de aplicação. Como, por exemplo, facilidades para gerenciamento de sistemas e para definição de interface com o usuário.

Interfaces de Domínio (*Domain Interfaces*): são interfaces direcionadas para determinados domínios de aplicação, tais como aplicações médicas e aplicações financeiras.

Interfaces de Aplicação (*Application Interfaces*): são constituídas por interfaces não padronizadas, as quais são direcionadas para uma aplicação específica.

CORBA define um padrão para um dos principais componentes da arquitetura OMA, denominado ORB. A Figura 2.5 ilustra a organização geral de um sistema CORBA, a qual é descrita nos parágrafos seguintes.

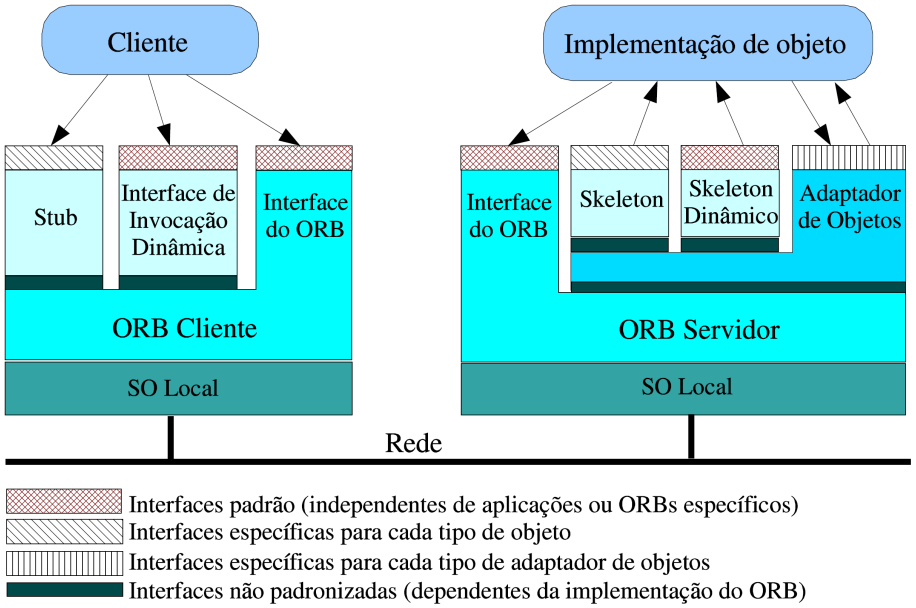

Figura 2.5: Arquitetura da plataforma CORBA[CK02]

Object Request Broker (ORB): constitui o núcleo da plataforma CORBA. A principal função do ORB é enviar requisições de chamadas remotas para objetos e retornar para os clientes os resultados decorrentes dessas chamadas [Vin97]. A principal característica do ORB é tornar transparente para o usuário os detalhes da programação distribuída, permitindo que o desenvolvedor possa abstrair-se de questões como localização, troca de mensagens, comunicação, criação e ativação de objetos remotos.

Linguagem de Definição de Interfaces (IDL): fornece uma linguagem neutra para descrever os serviços disponibilizados por uma aplicação distribuída. Por ser independente de qualquer linguagem de programação, ela permite que objetos sejam implementados em diferentes linguagens. Essa propriedade é importante em sistemas heterogêneos, pois fornece uma interface comum às diversas linguagens. A definição de IDL favoreceu o sucesso de CORBA como uma tecnologia de integração [Vin97].

Stub: componente que efetivamente realiza as invocações de métodos remotos em nome do cliente. Os *stubs* são criados automaticamente a partir das definições especificadas em IDL e são responsáveis por converter as requisições, emitidas pelos clientes, em um formato apropriado para transmissão em uma conexão de rede até o objeto remoto. Essa

conversão, envolve tanto a serialização dos parâmetros do método invocado, como também a desserialização do resultado da chamada. Além disso, esse processo de transformação depende da linguagem na qual o cliente foi implementado. Por isso, o gerador de *stubs* deve mapear as definições especificadas em IDL para a linguagem do cliente.

Skeleton: componente que repassa a requisição do cliente para o objeto remoto. Ou seja, ele efetivamente realiza a invocação sobre o objeto remoto no espaço de endereçamento do servidor. Para isso, ele recebe as requisições do ORB e desserializa os argumentos do método invocado. Após efetuar a chamada do método remoto, o *skeleton* serializa o seu resultado e o envia para o cliente. Da mesma forma que ocorre com o *stub*, o *skeleton* também é produzido por uma ferramenta de geração de código, a qual produz o código na linguagem na qual o servidor foi implementado.

Interface de Invocação Dinâmica (DII): componente que permite que chamadas de métodos remotos possam ser transmitidas ainda que inexista o *stub* que representa localmente o objeto remoto, ou seja, uma aplicação cliente pode realizar chamadas remotas sem conhecer, em tempo de compilação, a interface do objeto alvo. Nesta forma de chamada, a obtenção dos tipos envolvidos em uma requisição é realizada por meio de consultas ao repositório de interfaces (IR), o qual é uma estrutura que armazena de forma persistente informações sobre as interfaces IDL definidas em um sistema distribuído.

Skeleton Dinâmico (DSI): componente análogo à DII, porém usado no lado do servidor. Assim como DII permite aos clientes realizar chamadas remotas sem necessitar dos *stubs*, DSI permite que servidores recebam chamadas remotas sem possuir o *skeleton* do objeto sendo invocado.

Objeto Adaptador (OA): responsável por diversas funcionalidades, como por exemplo: criar, ativar e desativar objetos, gerar referências interoperáveis para objetos (IORs) e despachar invocações para o objeto apropriado.

Plataformas de middleware baseadas em CORBA

Com o sucesso da especificação CORBA, diversas plataformas de *middleware* foram desenvolvidas baseadas nessa especificação. Como por exemplo: JacORB [Jac], OpenORB [BCA+01], minimumCORBA [MOR], Java IDL [Jav] e CORBA Messaging [COR].

2.2 Programação Orientada por Aspectos

Tanto o paradigma procedural quanto o paradigma orientado por objetos não são completamente eficientes para modelar determinadas decisões de projeto que um programa deve implementar [KLM+97]. No paradigma procedural, a modularização é restrita à implementação de abstrações dos procedimentos necessários à realização de uma tarefa. Já a programação orientada por objetos fornece um nível mais elevado de abstração, permitindo representar tipos abstratos de dados.

Em ambos os paradigmas, os requisitos que compõem um sistema podem ser classificados como funcionais e não-funcionais. Os requisitos funcionais constituem as funcionalidades que se espera que o sistema disponibilize, ou seja, são os requisitos que capturam a funcionalidade central de um módulo. Já os requisitos não-funcionais compreendem elementos específicos de projeto, como restrições nas quais o sistema deve operar ou propriedades do sistema (como serviços de distribuição, *logging*, segurança e persistência). Via de regra, orientação por objetos permite modularizar de forma eficiente requisitos funcionais. No entanto, não é capaz de modularizar adequadamente requisitos que não são mapeáveis diretamente em uma ou poucas classes de um sistema [KLM+97]. Esses requisitos que se espalham e se entrelaçam pelo código funcional da aplicação são denominados requisitos transversais (*crosscutting concerns*). Requisitos não-funcionais constituem o tipo mais comum de requisito transversal.

2.2.1 Entrelaçamento e Espalhamento de Código

O problema da modularização de requisitos transversais pode ser classificado em duas categorias: espalhamento e entrelaçamento de código [Lad03]. O espalhamento (*scattering*) ocorre quando o código para implementação do requisito transversal fica disperso no programa, ou seja, fica espalhado em múltiplos módulos do sistema. Já o entrelaçamento (*tangling*) ocorre quando um módulo implementa simultaneamente múltiplos requisitos. Esses problemas dificultam a reusabilidade e a manutenibilidade do código do programa, além do código apresentar baixa coesão modular e alto grau de acoplamento entre módulos [TBBV04], comprometendo dessa forma a qualidade interna, a legibilidade e a evolução da aplicação. A Figura 2.6 ilustra como módulos de um sistema são formados pela composição de requisitos que se entrelaçam com o código de negócio da aplicação.

Programação Orientada por Aspectos (POA) [KLM+97] propõe uma nova abstração, denominada aspecto, que tem por objetivo modularizar decisões de projeto que não podem ser adequadamente definidas por meio de programação orientada por objetos. Para tanto, POA fornece mecanismos de composição que permitem descrever de forma modular

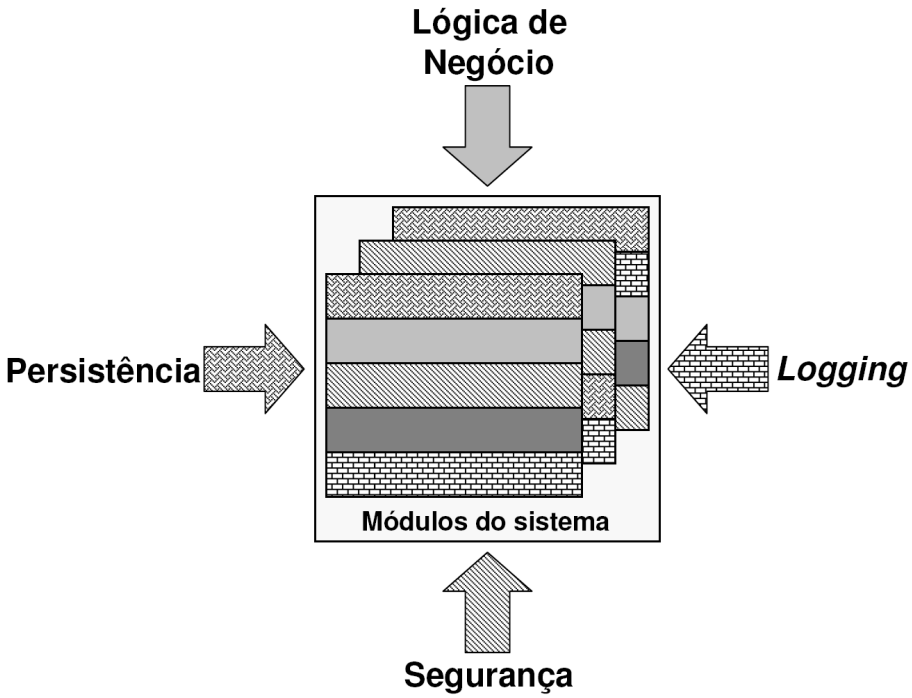

Figura 2.6: Sistema formado pela composição de múltiplos requisitos, adaptado de [Lad03]

propriedades que se encontram espalhadas e entrelaçadas em vários pontos de um sistema orientado por objetos. Além disso, POA fornece mecanismos para instrumentação de código, permitindo inclusive alterar a estrutura interna das classes de um sistema.

Na programação orientada por aspectos, requisitos de sistemas são modelados por meio de classes e aspectos. Estes últimos implementam requisitos transversais do sistema, tais como geração de registro de operações, tratamento de erros, segurança, persistência e comunicação [TBBV04].

Programação orientada por aspectos não se configura como uma alternativa à programação orientada por objetos, mas sim como um complemento desta. Juntas elas permitem que a estruturação do código seja realizada de forma mais eficiente, tornando-o mais legível e reutilizável, além de permitir que diversos requisitos sejam modularizados independentemente de sua natureza, seja ela funcional ou não funcional [RS05]. Para isso, primeiramente deve-se identificar os requisitos do sistema, através de um mecanismo de decomposição de requisitos. Uma das formas de se fazer essa decomposição consiste em considerar que cada requisito transversal forma uma dimensão independente das demais, a qual pode evoluir sem afetar os demais requisitos.

A Listagem 1, extraída de [Lad03], exemplifica um possível uso de orientação por aspectos. Nela podemos identificar os seguintes problemas:

1. Os atributos para suporte aos requisitos transversais não pertencem à lógica de

negócio da classe.

2. O método `someOperation` executa mais operações do que somente realizar a operação objetivo.

3. Os métodos `save()` e `load()` não realizam operações relacionadas à lógica de negócio da classe.

Listagem 1 Lógica de negócio com requisitos transversais entrelaçados

```
public class SomeBusinessClass extends OtherBusinessClass {
 "Atributos associados à lógica de negócio"
 "Atributos para suporte aos requisitos transversais"

 public void someOperation(ParameterOperation p) {
   "Garante autorização"
   "Garante que p satisfaça contrato"
   "Bloqueia o objeto para garantir controle de concorrência"
   "Garante que a cache está atualizada"
   "Faz o logging do início da operação"
   "EXECUTA AS OPERAÇÕES DE NEGÓCIO"
   "Faz o logging do término da operação"
   "Desbloqueia o objeto"
 }

 "Mais operações similares à operação anterior"

 public void save(PersistenceStorage p) {
   ...
 }

 public void load(PersistenceStorage p) {
   ...
 }
}
```

O problema de entrelaçamento é ilustrado no método `someOperation()` da Listagem 1, no qual os requisitos para controle de autorização, adequação a contratos, controle de *cache* e *logging* se entrelaçam com o requisito funcional do método. Já o problema de espalhamento de código ocorre quando um requisito transversal está presente em diversos módulos do sistema. Como exemplo, podemos citar o requisito de *logging*.

Programação orientada por aspectos permite que esses requisitos sejam encapsulados

em um único módulo denominado aspecto, o qual será costurado (*weaving*) nas classes usuárias deste requisito, de forma automática.

2.2.2 Principais Abstrações

Programação orientada por aspectos possibilita criar regras de costura de requisitos transversais ao núcleo de uma aplicação, sendo este formado basicamente pelos módulos que implementam sua lógica de negócio. *Pontos de junção* (*joinpoints*) compõem a base da programação orientada por aspectos. Um ponto de junção é um ponto bem definido no fluxo de execução de um programa, como chamadas e execução de métodos. Associados aos pontos de junção estão os conjuntos de junção (*pointcuts*) e as *regras de junção* (*advices*). Conjuntos de junção são construções do programa que contêm pontos de junção. Eles têm a função de reunir informações contextuais a respeito desses pontos. Ou seja, um conjunto de junção declara os pontos onde se deseja interceptar a execução de um programa para introduzir um dado comportamento ou requisito transversal. As regras de junção são responsáveis por definir o código que deve ser executado no momento em que um ponto de junção for alcançado. Uma regra de junção pode ser executada antes (*before*), depois (*after*) ou no lugar (*around*) do ponto de junção interceptado. Juntos, conjuntos de junção e regras de junção formam os mecanismos para transversalidade dinâmica de linguagens de programação orientada por aspectos.

Além dos recursos de transversalidade dinâmica, programação orientada por aspectos também fornece recursos de transversalidade estática. Esses recursos permitem modificar a estrutura das classes, interfaces e aspectos do sistema, possibilitando, por exemplo, inserir novos métodos e atributos em classes, alterar a hierarquia de classes, fazendo com que uma classe estenda outra ou fazendo com que uma classe implemente determinada interface.

2.3 AspectJ

Descrevem-se a seguir as principais abstrações utilizadas por AspectJ [KHH+01, Lad03, ASP] para modularizar requisitos transversais.

Pontos de junção: são pontos bem definidos no fluxo de execução de um programa, como, por exemplo, a chamada de um método ou uma atribuição a um membro de um objeto. Pontos de junção podem também possuir contexto associado. Por exemplo, o contexto associado à chamada de um método contém o alvo da chamada e sua lista de

parâmetros. Os mecanismos de transversalidade dinâmica de AspectJ giram em torno de pontos de junção, pois eles indicam onde os requisitos transversais serão costurados.

Conjuntos de junção: são formados por conjuntos de pontos de junção e também pelas informações contextuais destes pontos. Um conjunto de junção declara os pontos de um programa onde se deseja inserir um comportamento transversal. Esse comportamento pode se valer de informações contextuais para implementação das regras de junção, como, por exemplo, os valores dos argumentos de um método.

AspectJ permite a coleta de pontos de junção em inúmeros contextos, conforme descrito a seguir:

- **Chamadas de métodos e construtores:** capturam pontos de execução após a avaliação dos argumentos, no momento em que é realizada a chamada efetiva de um método, ou seja, antes da mudança do fluxo para o método a ser chamado. O construtor utilizado para esse tipo de conjunto de junção é o operador `call`, passando a assinatura do método a ser interceptado. Para interceptar chamadas de construtores, basta substituir o nome do método pela palavra reservada `new`.
- **Execução de métodos e construtores:** capturam a execução do corpo do método chamado. O construtor utilizado para esse tipo de conjunto de junção é o operador `execution`.
- **Acesso a atributos:** capturam as operações de leitura e escrita em atributos, por meio dos operadores `get` e `set` respectivamente.
- **Tratadores de exceções:** capturam o tratamento de exceções, ou seja, capturam a execução do bloco `catch`, sendo definidos por meio do operador `handler`.
- **Definições baseadas no fluxo de controle:** capturam pontos de junção que ocorrem no fluxo de controle gerado a partir de um ponto de junção. AspectJ define dois operadores para isso: `cflow` e `cflowbelow`. Em ambos os casos, deve-se especificar um conjunto de junção como argumento da operação. O primeiro operador captura os pontos de junção que ocorrem no fluxo de controle do conjunto de junção especificado, incluindo este próprio conjunto de junção. O segundo operador é semelhante ao primeiro, exceto pelo fato de excluir o ponto de junção capturado pelo conjunto de junção especificado.
- **Definições baseadas na estrutura léxica:** capturam pontos de junção que ocorrem no escopo léxico de classes, métodos ou aspectos. AspectJ define dois operadores

para isso: `within` e `withincode`. O primeiro operador captura todos os pontos de junção que ocorrem no escopo léxico de uma classe ou aspecto. Já o segundo operador captura todos os pontos de junção que ocorrem no escopo léxico de um método ou construtor.

Regras de junção: são operações que são executadas no momento em que um ponto de junção, selecionado por um conjunto de junção, for alcançado. Essas regras de junção podem ser executadas antes (`before`), após (`after`) ou no lugar (`around`) do ponto de junção interceptado. A implementação de uma regra de junção é semelhante à implementação de um método.

Aspectos: o principal construtor da linguagem AspectJ é chamado *aspecto*. Um aspecto pode incluir declarações de conjuntos de junção, declarações de regras de junção, assim como todas as construções permitidas para declaração de classes. Aspectos podem ser estendidos ou especializados, podendo-se estender somente aspectos abstratos. Além disso, não é permitida a definição de hierarquias múltiplas de aspectos. As regras para redefinição de métodos ou conjuntos de junção em aspectos são similares às regras para redefinição de métodos em Java.

Informações reflexivas sobre pontos de junção: AspectJ fornece suporte a reflexão, ainda que de forma limitada, para acessar informação estática ou dinâmica associada a pontos de junção. Para isso, AspectJ define três objetos especiais, chamados `thisJoinPoint`, `thisJoinPointStaticPart` e `thisEnclosingJoinPointStaticPart`, os quais fornecem informações sobre o ponto de junção interceptado.

Transversalidade estática: os elementos que fornecem suporte a transversalidade dinâmica, discutidos anteriormente, permitem modificar o fluxo de execução de um programa. No entanto, muitas vezes também é necessário modificar a estrutura estática dos tipos, classes, interfaces e aspectos, inclusive para fornecer recursos que serão instrumentalizados pelos mecanismos de transversalidade dinâmica. Os tipos de transversalidade estática suportados por AspectJ são os seguintes:

- **Introdução de membros:** freqüentemente há a necessidade de introduzir atributos e métodos em uma classe ou interface de forma a permitir a modularização do requisito transversal implementado pelo aspecto. A linguagem AspectJ provê um mecanismo denominado introdução (`introduction`), o qual é usado para inserir membros em classes e interfaces de maneira transversal.

- **Alteração da hierarquia de classes:** AspectJ fornece construtores que possibilitam alterar a hierarquia de classes, de forma a definir superclasses, fazer com que classes implementem determinadas interfaces e também fazer com que uma interface estenda outra.

2.4 Distribuição e Aspectos

Atualmente, distribuição é um requisito presente na maioria dos projetos de desenvolvimento de sistemas. Isso resultou no sucesso de tecnologias de *middleware*, as quais têm como objetivo encapsular detalhes inerentes à programação em ambientes de redes. No entanto, paradoxalmente, os sistemas atuais de *middleware* também acrescentam detalhes, convenções e protocolos próprios de programação, os quais devem ser dominados por engenheiros de *software*. Mais importante, estes detalhes, via de regra, possuem um comportamento transversal, se entrelaçando e se espalhando pelo código funcional de uma aplicação. O resultado final é que aplicações distribuídas baseadas em *middleware* não apresentam graus desejados de modularidade, reusabilidade e separação de interesses.

Desta forma, desde o final da década de 90, distribuição vêm sendo citada e estudada como um dos principais interesses que podem se beneficiar de implementações orientadas por aspectos, em função do comportamento transversal desse requisito no núcleo da aplicação. Diversos projetos de pesquisa têm investigado o uso de aspectos para modularizar serviços transversais providos por plataformas de *middleware*. A seguir, são descritos alguns destes projetos.

2.4.1 Sistema Health Watcher

Em [SBL06, SLB02], Soares, Borba e Laureano descrevem uma experiência bem sucedida de reestruturação de uma aplicação real, chamada *Health Watcher*, usada por cidadãos para enviar reclamações sobre restaurantes e similares a órgãos públicos responsáveis pela vigilância sanitária destes estabelecimentos. O sistema foi implementado usando uma arquitetura de camadas associada a padrões de projeto. O sistema inclui uma interface de usuário baseada em HTML e Java script, a qual interage com *servlets* Java registrados em um servidor Web, que por sua vez interagem com objetos remotos usando Java RMI.

No referido trabalho, os autores mostram como este sistema foi reestruturado de forma a modularizar em aspectos o interesse de distribuição, representado por código de comunicação via Java RMI, assim como outros interesses subjacentes a esse interesse, como,

por exemplo, os mecanismos de serialização e tratamento de exceções.

Na solução proposta, os autores implementaram uma hierarquia de aspectos, composta por aspectos abstratos, os quais são independentes da arquitetura do sistema, e aspectos concretos, que estendem os aspectos abstratos e que são específicos para a arquitetura do sistema *Health Watcher*. Esses aspectos são agrupados em aspectos para introdução de distribuição nas classes da aplicação cliente e em aspectos para introdução de distribuição nas classes da aplicação servidora.

O serviço a ser acessado remotamente possui uma fachada que controla o acesso ao mesmo. Desta forma, os aspectos no lado do servidor têm a função de tornar essa fachada remotamente acessível. Além disso, esses aspectos também devem tornar serializáveis tanto os tipos dos parâmetros formais dos métodos dessa fachada, quanto os tipos de retorno desses métodos. No lado do cliente, os aspectos interceptam as chamadas efetuadas pelo cliente sobre a fachada, e as redirecionam para uma referência remota, a qual é instanciada e encapsulada em um aspecto.

A abordagem utilizada no sistema *Health Watcher* possui algumas restrições, descritas a seguir:

- A solução proposta inclui suporte apenas a passagem de parâmetros em chamadas remotas por serialização, ainda que Java RMI também forneça suporte a passagem de objetos com a semântica de referência remota.

- A solução é restrita e limitada pela arquitetura do sistema, na qual assume-se a existência de uma única instância do objeto servidor. Desta forma, essa solução não permite que clientes possuam múltiplas referências remotas do mesmo tipo, cada uma referenciando objetos remotos distintos.

- A solução proposta é restrita a Java RMI, o que impede que a aplicação servidora receba chamadas remotas de clientes implementados em outras linguagens de programação.

2.4.2 RMIX

RMIX [KWSS03] é um arcabouço que permite a instanciação de servidores que aceitam chamadas remotas provenientes de Java RMI, CORBA ou SOAP. A Figura 2.7 ilustra como RMIX abstrai o protocolo de comunicação usado por aplicações distribuídas. Para prover esta flexibilidade, RMIX reimplementa diversos componentes internos comuns a sistemas de *middleware*.

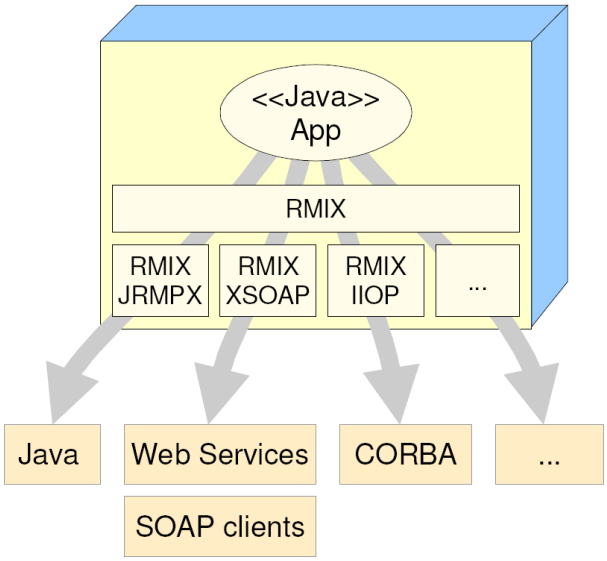

Figura 2.7: Arquitetura do arcabouço RMIX [KWSS03]

Apesar da flexibilidade provida por RMIX, o uso do arcabouço requer uma API proprietária, cujo código fica espalhado e entrelaçado ao código funcional da aplicação. Além disso, o mecanismo para serialização de RMIX requer que objetos passados como argumentos de chamadas remotas, ou retornados como resultado dessas chamadas, sigam determinadas convenções, como, por exemplo, implementar métodos `get` e `set`.

2.4.3 Model-Driven Architecture

Ghosh e colegas descrevem uma metodologia para implementar de forma independente interesses de negócio e interesses de distribuição providos por plataformas de *middleware* [GFS+05]. A idéia de os autores é incorporar conceitos e técnicas de desenvolvimento orientado por modelos e, mais especificamente, conceitos propostos pela arquitetura MDA (*Model-Driven Archictecture*) [OMGb], ao desenvolvimento de aplicações distribuídas. Na metodologia proposta, designa-se como MTD (*Middleware Transparent Design*) um modelo que captura apenas os interesses funcionais de um sistema. Este modelo é então combinado com aspectos específicos de uma plataforma de *middleware*, a fim de se obter um MSD (*Middleware Specific Design*), isto é, um modelo incluindo requisitos funcionais e de distribuição.

Apesar dos autores salientarem que a abordagem orientada por aspectos pode facilitar a evolução de uma aplicação distribuída, a metodologia proposta não foi efetivamente colocada em prática e, para isso, os autores sugerem a criação de uma ferramenta de geração e transformação de código como um recurso para agilizar o processo de produção dos

aspectos necessários para introdução do interesse de distribuição no núcleo de aplicações.

2.4.4 Sistema FreeTTS

Ceccato e Tonella descrevem um *framework* que inclui geração automática de aspectos de distribuição e interfaces remotas [CT04]. A abordagem proposta foi validada através da utilização do *framework* em uma aplicação que produz um arquivo de áudio a partir de um arquivo de texto.

A solução proposta possui as seguintes restrições:

- Suporte apenas a Java RMI como protocolo de comunicação, fazendo com que a aplicação servidora se comunique apenas com aplicações clientes também implementadas em Java.
- Semântica de passagem de parâmetros somente por referência remota, a qual produz uma sobrecarga desnecessária em aplicações que não necessitam dessa funcionalidade.
- A especificação dos parâmetros de distribuição é feita por meio de uma simples lista com o nome das classes remotas do sistema.

O sistema também possui mecanismos que tratam do problema de migração, permitindo que uma aplicação seja distribuída dinamicamente entre diversos nodos da rede. Além disso, o código de distribuição é produzido automaticamente por uma ferramenta de geração de código, isentando o usuário da tarefa de implementar manualmente esse requisito.

2.4.5 Outros Trabalhos

Já foram desenvolvidas ferramentas baseadas em aspectos para auxiliar na implementação de aplicações EJB, tais como Gotech [TUSF03] e AspectJ2EE [CG04]. Gotech é um *framework* que gera aspectos capazes de transformar classes Java em componentes EJB. AspectJ2EE é uma implementação orientada por aspectos de um *container* que disponibiliza serviços não-funcionais típicos de servidores EJB.

AspectJRMI [VTLP05] é um sistema de chamada remota de métodos que, por meio de conceitos de orientação por aspectos, disponibiliza uma plataforma de *middleware* configurável em tempo de compilação. Em AspectJRMI, programadores podem agregar funcionalidades transversais ao núcleo do sistema, o qual oferece basicamente um serviço de chamadas síncronas de métodos remotos.

Aspectos já foram empregados também na implementação de aplicações baseadas em Serviços *Web*. Por exemplo, Courbis e Finkelstein propõem o uso de aspectos dinâmicos para customizar e adaptar aplicações usadas para coordenar a execução de chamadas a serviços *Web* [CF05]. Já WSML [VVJ06] é um *middleware* que utiliza aspectos para encapsular diversos requisitos transversais típicos de aplicações clientes de serviços *Web*, incluindo autenticação, *logging*, monitoramento, transações, etc.

Outros trabalhos têm como objetivo incorporar abstrações para programação distribuída em linguagens orientadas por aspectos. DJcutter [NCT04] é uma linguagem que acrescenta em AspectJ o conceito de *pointcuts* remotos, os quais viabilizam a captura de pontos de junção pertencentes à execução de programas em máquinas remotas. GluonJ [CI05] é uma extensão de AspectJ com suporte a injeção de dependência.

2.5 Conclusão

A disseminação de redes de computadores proporcionou um crescimento vertiginoso na demanda por aplicações distribuídas. A heterogeneidade do ambiente em que estas aplicações são executadas tornou mais complexo o desenvolvimento desses sistemas, em relação aos sistemas centralizados. Plataformas de *middleware* foram propostas com o objetivo de facilitar o desenvolvimento de aplicações distribuídas, tornando transparente para o desenvolvedor as complexidades inerentes ao ambiente em que estas aplicações estão inseridas.

Esses benefícios colaboraram para que o requisito de distribuição esteja atualmente presente na grande maioria dos projetos de desenvolvimento de *software*. No entanto, os sistemas atuais de *middleware* acrescentam detalhes, convenções e protocolos próprios de programação, os quais se entrelaçam no código funcional de uma aplicação, podendo se espalhar entre diversos módulos do sistema. Desta forma, aplicações distribuídas baseadas em plataformas de *middleware* apresentam deficiências de modularidade, reusabilidade e separação de interesses, o que dificulta a legibilidade e a evolução dessas aplicações.

Programação Orientada por Aspectos é um novo paradigma de programação que permite que requisitos transversais, como o requisito de distribuição, possam ser eficientemente modularizados em uma única unidade de programação, chamada aspecto. Assim, surgiram diversas propostas que se valem de aspectos para isolar o requisito de distribuição da lógica de negócio de sistemas distribuídos. No entanto, essas propostas possuem limitações, como a utilização de um único protocolo de comunicação, ou possuem deficiências de configurabilidade do cenário de distribuição da aplicação.

Capítulo 3

O Sistema DAJ

3.1 Introdução

Nessa dissertação, apresenta-se um sistema de apoio à programação distribuída, denominado DAJ [MV06a] (*Distribution Aspects in Java*). Esse sistema permite modularizar o interesse de distribuição provido pelas duas principais plataformas de *middleware* para desenvolvimento de sistemas distribuídos em Java: Java RMI [WRW96] e Java IDL [Jav]. O sistema se encarrega de gerar módulos que isolam o requisito de distribuição, resultante da utilização dessas plataformas de *middleware*, da lógica de negócio de sistemas implementados em Java.

O sistema DAJ é formado por classes e aspectos que encapsulam diversos detalhes de programação usualmente requeridos por plataformas de *middleware*. Essas classes e aspectos são produzidos por uma ferramenta de geração de código, e são combinados ao núcleo do sistema, de forma a produzir a versão de implantação da aplicação distribuída. Desta forma, a instanciação do sistema de distribuição proposto é feita de modo automático.

Em aplicações distribuídas implementadas com o apoio do sistema DAJ, classes de negócio não precisam seguir quaisquer protocolos de codificação, isto é, não precisam estender outras classes ou interfaces, não precisam implementar métodos `get` e `set`, não precisam ativar ou tratar exceções pré-definidas, etc. Em vez disso, no sistema proposto, descritores de distribuição são usados para descrever o papel desempenhado por tais classes em um sistema de objetos distribuídos baseado em uma determinada tecnologia de *middleware*. Na solução proposta, descritores de distribuição são usados para declarar propriedades de um serviço (ou objeto) remoto, incluindo a interface implementada pelo serviço, a classe usada para instanciar o serviço, a plataforma de *middleware* usada para comunicação com o serviço e informações necessárias para localizar e registrar o serviço junto a um servidor de nomes. A partir das informações contidas em um descritor de

distribuição, a ferramenta de geração de código integrante do sistema DAJ se encarrega de gerar automaticamente aspectos, que são implementados em AspectJ, e classes que modularizam o código de distribuição requerido pela aplicação base.

Apesar de DAJ requerer que a classe que codifica a lógica de negócio implemente uma determinada interface, denominada interface de negócio, não se considera que esse requisito implique em uma convenção a ser seguida apenas em razão da utilização do sistema. Na verdade, a definição de uma interface para especificação dos métodos que podem ser invocados constitui, antes de tudo, uma boa prática no desenvolvimento de *software*.

A Figura 3.1 descreve o processo de desenvolvimento de aplicações distribuídas usando DAJ. Inicialmente (etapa A da figura), o engenheiro de *software* implementa os requisitos funcionais de sua aplicação, sem a obrigatoriedade de seguir quaisquer convenções de programação requeridas por uma determinada plataforma de *middleware*. Essa característica do desenvolvimento de aplicações distribuídas em DAJ é fundamental, pois permite que o desenvolvedor concentre-se apenas nos requisitos funcionais da aplicação, a qual permanece livre de código entrelaçado, espalhado e responsável por interesses de distribuição. Em um segundo momento (etapa B), um engenheiro de *software* especialista em sistemas de *middleware* define, por meio de um descritor de distribuição, a configuração de distribuição da aplicação construída na etapa anterior. De forma geral, este descritor especifica quais objetos serão acessados remotamente, assim como informações necessárias para registro, ativação e localização dos mesmos. Em seguida (etapa C), o núcleo da aplicação e o descritor de distribuição são fornecidos como entrada para a ferramenta de geração de código do sistema DAJ. Esta ferramenta se encarrega de gerar classes e aspectos que encapsulam código de comunicação responsável por transformar a aplicação construída na primeira etapa em uma aplicação distribuída. Por fim (etapa D), o compilador de aspectos efetua o *weaving* do núcleo da aplicação com o código gerado por DAJ e produz a versão de implantação da aplicação distribuída.

Em relação aos benefícios introduzidos por DAJ, podemos destacar os seguintes:

- O desenvolvedor implementa o núcleo da aplicação sem se preocupar com questões relativas ao interesse de distribuição do sistema. Assim, esse desenvolvedor concentra-se exclusivamente na implementação dos requisitos funcionais de sua aplicação. Desta forma, o uso de DAJ permite que a aplicação seja implementada mais rapidamente, uma vez que o desenvolvedor ignora o requisito de distribuição, reduzindo, desta forma, a quantidade de código a ser implementada manualmente. Além disso, como o desenvolvedor não se preocupa com outros interesses, o código da aplicação fica propenso a ter menos erros de implementação.

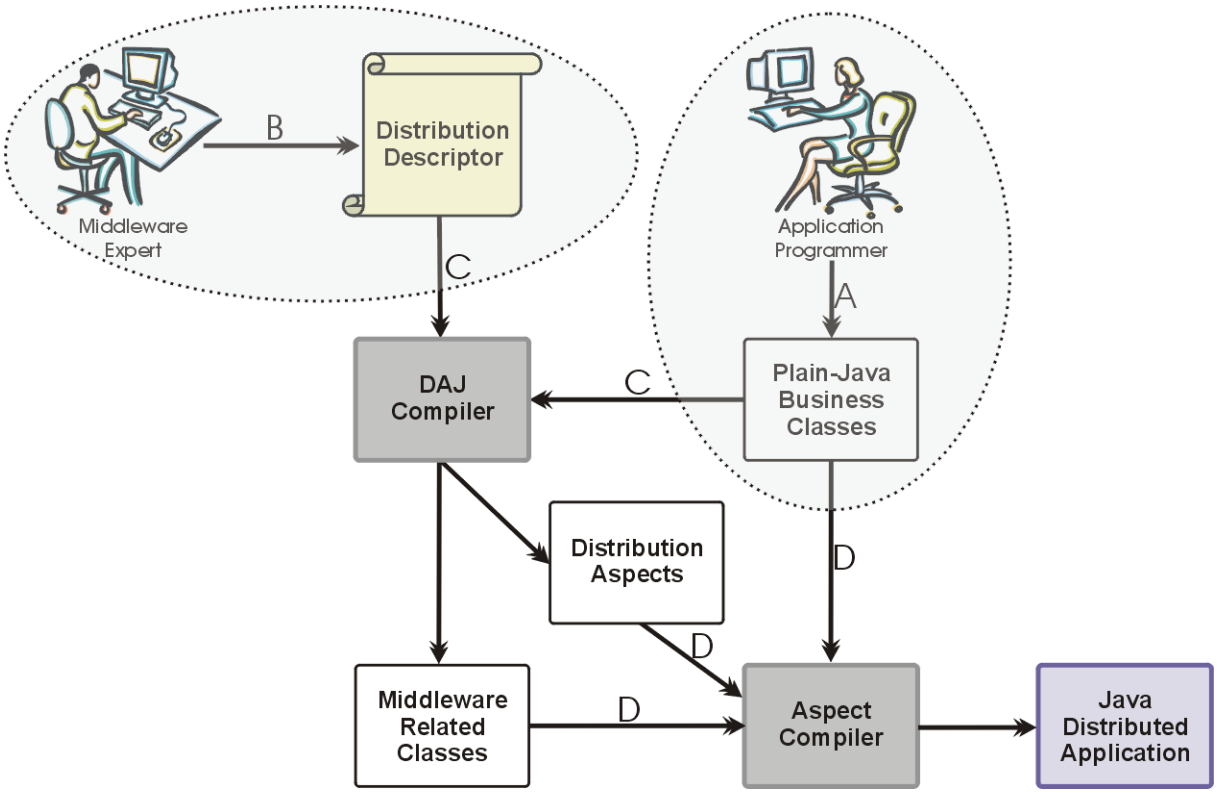

Figura 3.1: Processo de desenvolvimento de aplicações distribuídas usando DAJ

- O código destinado à introdução da funcionalidade de distribuição não fica mais entrelaçado e espalhado pelo código funcional do núcleo da aplicação. Desta forma, o uso de DAJ permite que um sistema seja construído de forma mais independente, com baixo acoplamento, melhorando assim a legibilidade, a reusabilidade e a modularidade do código.

- Como o código é gerado automaticamente por uma ferramenta, o desenvolvedor da aplicação distribuída não precisa dominar detalhes e convenções de codificação requeridos por plataformas de *middleware*, nem conceitos de programação orientada por aspectos, já que não será responsável por codificar de forma manual aspectos responsáveis por modularizar código de distribuição, o que quase sempre é uma tarefa tediosa e sujeita a erros.

3.2 Exemplo Motivador: Sistema para Controle de Ações

A fim de ajudar a descrever e avaliar o funcionamento do sistema DAJ, será utilizado como exemplo no restante deste capítulo uma aplicação distribuída para monitoramento e controle do preço de ações negociadas em uma determinada bolsa de valores. Este sistema inclui um servidor que armazena o preço corrente das ações negociadas nesta bolsa de valores. Clientes acessam este servidor com três objetivos: (i) comunicar uma alteração no preço de venda de uma determinada ação, (ii) obter o preço corrente de venda de uma ação e (iii) manifestar interesse em receber notificações de alterações no preço de determinada ação. Nesse último caso, o servidor notifica o cliente por meio de um *callback*.

Apesar de simples, esta aplicação faz uso das principais abstrações providas por sistemas de *middleware* orientados por objetos. Essencialmente, ela utiliza chamadas remotas de métodos, passando objetos tanto por serialização como por referência remota. Além disso, ela admite diversas configurações de implantação. Por exemplo, um cenário de implantação pode incluir um único servidor, utilizando Java RMI e diversos clientes Java. Em outro cenário, este servidor deve utilizar CORBA, já que o mesmo será acessado por sistemas implementados em outras linguagens de programação. Em um terceiro cenário, pode existir um servidor RMI e um servidor CORBA, cada um deles controlando um subconjunto das ações negociadas na bolsa.

A Figura 3.2 ilustra as classes e interfaces que constituem o núcleo do Sistema de Controle de Ações, as quais serão descritas em detalhes a seguir:

Interfaces: O serviço provido pelo servidor que armazena o preço corrente das ações negociadas na bolsa de valores possui a seguinte interface:

```
interface StockMarket {
  void update(StockInfo info);
  void subscribe(String stock, StockListener obj);
  void unsubscribe(String stock, StockListener obj);
  StockInfo getStock(String stockName) throws StockNotFoundException;
}
```

Clientes publicam uma alteração no preço das ações de uma determinada companhia chamando o método `update`. A classe `StockInfo`, mostrada a seguir, é usada para armazenar o valor de uma ação em uma determinada data e hora.

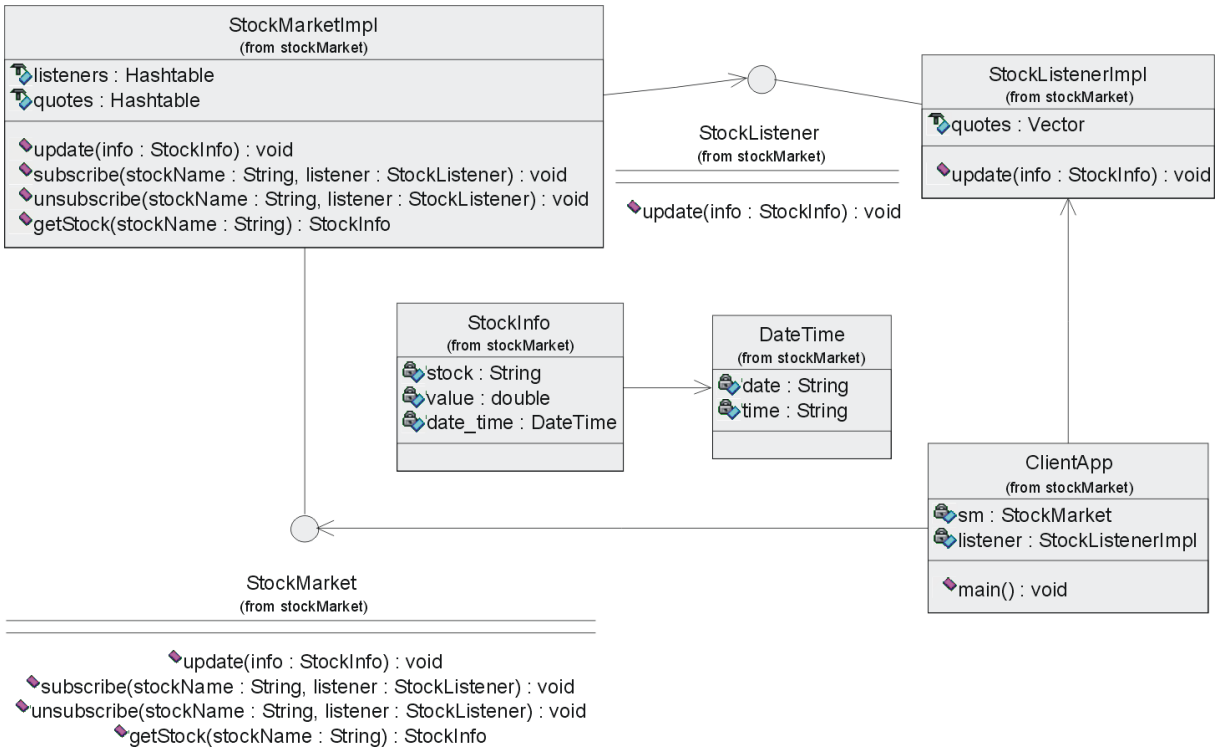

Figura 3.2: Diagrama de classes dos objetos que formam o núcleo do sistema

```
class StockInfo {
  String stock;
  double value;
  DateTime date_time;
}
```

Clientes podem ainda registrar interesse em serem notificados de alterações no preço das ações de uma determinada companhia. Para isso, devem utilizar o método `subscribe`, fornecendo como argumento o nome da ação e um objeto que receberá via *callback* a notificação. O método `unsubscribe` é usado para cancelar a assinatura realizada por meio do método `subscribe`. Em ambos os métodos, o objeto usado para receber a notificação deve implementar a interface `StockListener`, mostrada abaixo.

```
interface StockListener {
  void update(StockInfo info);
}
```

Além disso, usuários que apenas desejarem consultar o valor corrente de uma ação devem usar o método `getStock`, fornecendo o nome da ação como argumento. Neste

último método, caso não existam informações sobre a ação no servidor, uma exceção do tipo `StockNotFoundException` será ativada.

Implementação: Na implementação do sistema, a classe `StockMarketImpl` é responsável por implementar a interface `StockMarket`. Basicamente, esta classe utiliza duas tabelas *hash* privadas: a primeira tabela, chamada `quotes`, é usada para armazenar o preço corrente de cada uma das ações negociadas na bolsa; a segunda tabela, chamada `listeners`, é usada para mapear uma ação para uma lista de objetos observadores de clientes que manifestaram interesse em serem notificados sobre alterações no preço desta ação. A classe `StockMarketImpl` implementa ainda os métodos `update`, `subscribe`, `unsubscribe` e `getStock`.

Por sua vez, a classe `StockListenerImpl` implementa a interface `StockListener` e, portanto, o método `update`. A implementação deste método simplesmente exibe uma mensagem na console do sistema informando o novo preço da ação.

Ao contrário do que aconteceria se o sistema tivesse sido implementado diretamente sobre Java RMI ou Java IDL, as interfaces e classes de negócio descritas não possuem qualquer código de distribuição entrelaçado. Em outras palavras, com a utilização do sistema DAJ, permite-se que um projetista de *software* se concentre totalmente na lógica de negócios de sua aplicação.

Na próxima seção, descreve-se a interface de programação e os descritores de distribuição utilizados pelo sistema DAJ.

3.3 Interface de Programação

Descreve-se nesta seção a interface de programação proposta por DAJ. Particularmente, são descritas as convenções que devem ser seguidas para especificação de descritores de distribuição e para codificação de clientes e servidores.

3.3.1 Descritores de Distribuição

Um dos objetivos do sistema de distribuição proposto é ocultar do projetista de *software* a camada de *middleware* subjacente à aplicação distribuída que está sendo desenvolvida. Com isso, este projetista pode se concentrar na lógica de negócio do seu sistema, deixando a cargo do sistema a implementação dos aspectos relacionados à distribuição. No entanto, este projetista precisa explicitar quais objetos da aplicação serão acessados remotamente. Para isso, utiliza-se de um conjunto de documentos XML, chamados *descri-*

tores de distribuição, por meio dos quais são definidos diversos parâmetros de distribuição da aplicação.

Um descritor de distribuição é formado por três documentos XML: um descritor dos serviços remotos, nos quais são configurados os servidores que compõem a aplicação; um descritor de objetos que, em chamadas remotas de métodos, são enviados com a semântica de passagem por referência remota e um descritor dos objetos enviados com a semântica de passagem por serialização. No caso de serviços remotos, deve-se definir o identificador do servidor, o nome de sua interface de negócio, o nome da classe que implementa esta interface, o protocolo de comunicação a ser utilizado e o nome e a porta do servidor de nomes onde o servidor será registrado. No caso de tipos passados por referência remota, deve-se informar a interface e a classe deste tipo. No caso de tipos passados por serialização deve-se informar a classe deste tipo.

A seguir são apresentados os esquemas dos documentos XML, usados nos descritores de distribuição, para objetos servidores (i), para objetos passados por referência remota (ii) e para objetos passados por serialização (iii). Esses esquemas são utilizados para validar os documentos XML antes de iniciar o processo de geração de código, por meio da ferramenta `dajc` (descrita na Seção 3.5).

i)

```
<!ELEMENT services (server+)>
<!ELEMENT server (interface, class, protocol, serverName?)>
<!ELEMENT interface (#PCDATA)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT protocol (#PCDATA)>
<!ELEMENT serverName (#PCDATA)>
<!ATTLIST server id CDATA #REQUIRED>
```

ii)

```
<!ELEMENT remoting (remote+)>
<!ELEMENT remote (class+)>
<!ELEMENT class (#PCDATA)>
<!ATTLIST remote interface CDATA #REQUIRED>
```

iii)

```
<!ELEMENT serializable (class+)>
<!ELEMENT class (#PCDATA)>
```

Exemplo: Mostra-se a seguir um exemplo de descritor de distribuição para o Sistema de Controle de Ações.

a)

```
 1: <services>
 2:   <server id="StockMarketA">
 3:     <interface>stockMarket.StockMarket</interface>
 4:     <class>stockMarket.StockMarketImpl</class>
 5:     <protocol>javaidl</protocol>
 6:     <nameserver>skank.pucminas.br</nameserver>
 7:   </server>
 9:   <server id="StockMarketB">
10:     <interface>stockMarket.StockMarket</interface>
11:     <class>stockMarket.StockMarketImpl</class>
12:     <protocol>javarmi</protocol>
13:     <nameserver>patofu.pucminas.br:1530</nameserver>
14:   </server>
15: <services>
```

b)

```
 1: <remoting>
 2:   <remote interface="stockMarket.StockListener">
 3:       <class>stockMarket.StockListenerImpl</class>
 4:   </remote>
 5: </remoting>
```

c)

```
 1: <serializable>
 2:   <class>stockMarket.StockInfo</class>
 3:   <class>stockMarket.DateTime</class>
 4: </serializable>
```

Por meio deste descritor, configura-se uma implantação do sistema com dois servidores (documento a): o primeiro deles utilizará Java IDL para comunicação e será registrado com o nome `StockMarketA` (linhas 2 a 7); o segundo utilizará Java RMI e será registrado com o nome `StockMarketB` (linhas 9 a 14). Além disso, define-se que, em chamadas remotas de métodos destes servidores, objetos do tipo `StockListenerImpl` serão passados por referência remota (documento b) e que objetos do tipo `StockInfo` e `DateTime` serão passados por serialização (documento c).

3.3.2 Codificação de Clientes

Em sistemas distribuídos apoiados por plataformas de *middleware* orientadas por objetos, aplicações cliente obtêm referências para objetos remotos e então chamam métodos dos mesmos. Nestas chamadas, parâmetros podem ser passados por cópia (via serialização) ou então por referência remota (quando se passa o *stub* do parâmetro de chamada). Em DAJ, um cliente deve utilizar o método `getReference` do sistema para obter uma referência para um dos servidores configurados no descritor de distribuição do sistema. A partir da referência obtida, o cliente pode chamar métodos remotos deste servidor, sem estar ciente do *middleware* que suporta esta comunicação.

Exemplo: A Figura 3.3 ilustra um possível cenário de distribuição do Sistema de Controle de Ações.

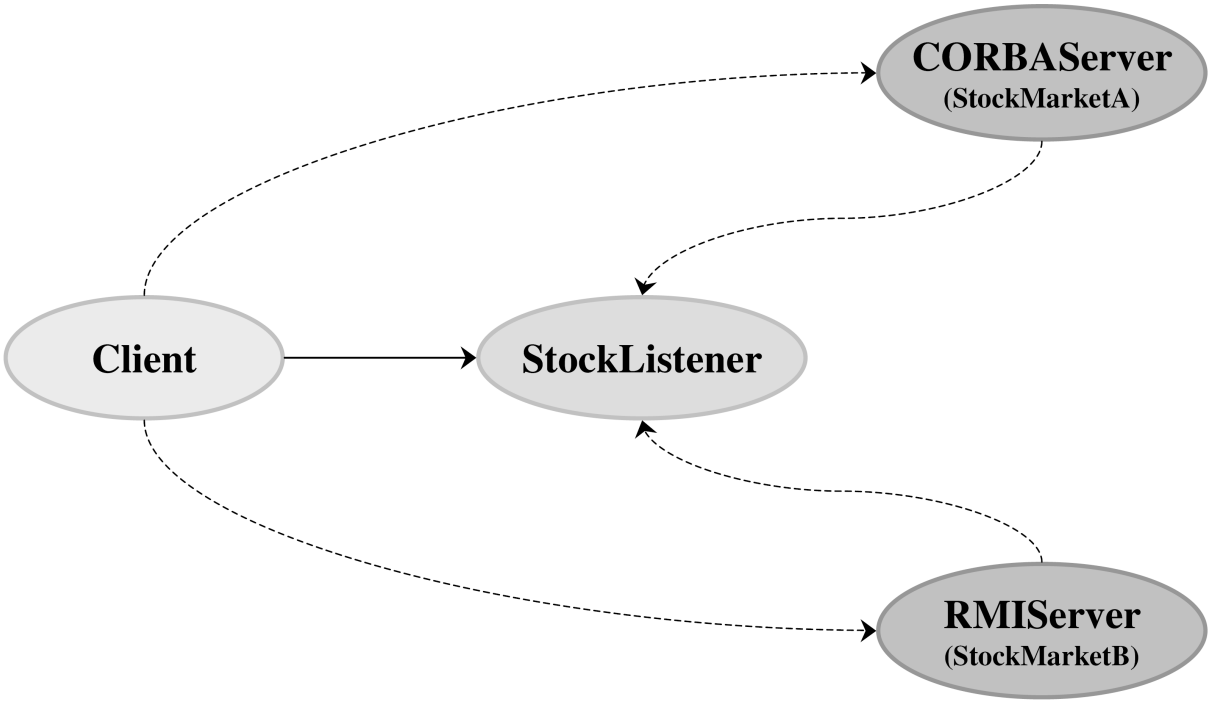

Figura 3.3: Cenário de distribuição do Sistema de Controle de Ações

Mostra-se a seguir um fragmento de código cliente baseado no cenário descrito na Figura 3.3.

```
1: StockMarket s1,s2;
2: s1=(StockMarket)ServiceLocator.getReference("StockMarketA");
3: s2=(StockMarket)ServiceLocator.getReference("StockMarketB");
4:   ....
```

```
 5: StockInfo info;
 6: info= new StockInfo("cvrd", 124.60, new DateTime("10/04/2006", "18:21"));
 7: s1.update(info);
 8: .......
 9: StockListener listener = new StockListenerImpl();
10: s1.subscribe("bb", listener);
11: s2.subscribe("cef", listener);
```

Nas linhas de 1 a 3, por meio do método `getReference`, o cliente obtém referências para os servidores de nome `StockMarketA` e `StockMarketB`, definidos no descritor de distribuição mostrado no exemplo da Seção 3.3.1. Em seguida, o cliente chama o método `update` do primeiro servidor passando um objeto do tipo `StockInfo` por serialização (linhas 5 a 7). Por fim, o cliente manifesta seu interesse em receber notificações sobre alterações nos preços de duas ações (linhas 9 a 11). Veja que em ambas chamadas do método `subscribe`, o cliente informa o mesmo objeto do tipo `StockListener`. Este objeto, passado por referência remota, receberá notificações do primeiro servidor via Java IDL e do segundo servidor via Java RMI.

3.3.3 Codificação de Servidores

Conforme afirmado na Seção 3.2, as interfaces e classes de negócio de uma aplicação distribuída construída com suporte da plataforma DAJ não possuem código de distribuição espalhado e entrelaçado. Assim, seus desenvolvedores não precisam se preocupar com convenções de programação requeridas por uma determinada plataforma de *middleware*. Além disso, para cada servidor definido em um descritor de distribuição, o sistema DAJ gera automaticamente uma classe de ativação, a qual possui um método `main` contendo código para instanciar, ativar e registrar o respectivo objeto remoto. Esta classe possui o nome do servidor, prefixado com a palavra `Server`.

Exemplo: Suponha o descritor de distribuição mostrado na Seção 3.3.1. Para instanciar, ativar e registrar os servidores especificados neste descritor, o sistema DAJ gera automaticamente as seguintes classes: `ServerStockMarketA` e `ServerStockMarketB`.

Ainda que DAJ gere automaticamente classes para ativação e registro de objetos servidores, algumas aplicações demandam que instâncias de objetos remotos sejam registradas explicitamente, em função de condições cuja satisfação somente pode ser determinada em tempo de execução. Isso ocorre em aplicações em que a localização e o registro de objetos remotos constituem parte integrante da lógica de negócio, como por exemplo em sistemas *peer-to-peer* e sistemas de grades computacionais (*grids*). Desta forma, DAJ fornece um

método para ativação e registro de objetos remotos, chamado `registryRemoteObject`. Nesse método, o objeto remoto será registrado com o nome passado como argumento, o qual deve corresponder a um identificador de um serviço remoto especificado no descritor de distribuição.

3.4 Arquitetura Interna

Descreve-se nesta seção a arquitetura interna do sistema DAJ, enfatizando os aspectos abstratos propostos pelo sistema e os aspectos concretos, gerados para estender estes últimos a partir das informações contidas em descritores de distribuição. Inicia-se a seção descrevendo-se o processo de geração de interfaces remotas e, em seguida, são descritos os aspectos do sistema responsáveis por transformar classes de negócio em classes remotas, obter referências para objetos remotos, ativar servidores e tratar exceções.

3.4.1 Interfaces Remotas

Em DAJ, interfaces remotas são interfaces com código de distribuição entrelaçado. A partir das informações lidas do descritor de distribuição, o sistema DAJ gera automaticamente interfaces remotas conforme requerido pelas plataformas Java RMI ou Java IDL. Em ambos os casos, estas interfaces têm o mesmo nome das interfaces de negócio especificadas no descritor de distribuição. No entanto, para evitar colisões, interfaces remotas são geradas em um pacote separado.

Interfaces Remotas em Java RMI: Interfaces remotas requeridas por Java RMI são semelhantes às suas respectivas interfaces de negócio, exceto pelo fato de estenderem `java.rmi.Remote` e de todos os seus métodos declararem que podem ativar uma exceção do tipo `java.rmi.RemoteException`. O código destas interfaces remotas deve ser gerado pelo sistema DAJ já que os recursos de transversalidade estática de AspectJ não permitem acrescentar uma cláusula `throws` na assinatura de métodos. Esta mesma dificuldade já foi reportada em outros trabalhos [SBL06, TUSF03, CT04].

Outra diferença entre interfaces de negócio e interfaces remotas diz respeito a parâmetros passados por referência remota ou referências remotas retornadas como resultado de chamadas de métodos remotos. Mais especificamente, se um método de uma interface de negócio possui um parâmetro ou um retorno de um tipo `T` declarado como remoto no descritor de distribuição do sistema, então, na interface remota gerada, o tipo `T` deve ser trocado pelo tipo de sua respectiva interface remota. Em chamadas de métodos, o *run-time* de Java RMI se encarrega de passar o *stub* do parâmetro de chamada quando

o parâmetro formal é de um tipo remoto. Como o *stub* gerado automaticamente pela implementação de Java RMI implementa a interface remota, ele não seria compatível para atribuição com um parâmetro formal declarado como sendo do tipo de uma interface de negócio. Daí a necessidade da substituição descrita neste parágrafo.

Suponha que `A` e `ARMI` são, respectivamente, interfaces de negócio e remotas. Uma possível alternativa para a referida substituição seria, via declaração intertipos, fazer `A` estender `Remote` e `ARMI` estender `A`. No entanto, esta alternativa é inviável, pois em Java não se permite que um método redefinido em uma sub-interface acrescente exceções verificadas à cláusula `throws` do método sobrescrito. É importante ressaltar que a ativação da exceção `java.rmi.RemoteException` na interface remota é indispensável tanto para registro do objeto remoto em um servidor de nomes quanto para geração do `stub` do objeto passado por referência remota ou retornado pelo método remoto invocado.

A fim de ilustrar a geração de interfaces em RMI, mostra-se a seguir a interface remota gerada para a interface de negócio `StockMarket`, apresentada na Seção 3.2:

```
public interface StockMarket extends java.rmi.Remote {
  void update(stockMarket.StockInfo arg0) throws java.rmi.RemoteException;
  void subscribe_daj_javarmi(String arg0, daj.javaidl.api.StockListener arg1)
       throws java.rmi.RemoteException;
  void unsubscribe_daj_javarmi(String arg0, daj.javaidl.api.StockListener arg1)
       throws java.rmi.RemoteException;
  stockMarket.StockInfo getStock(String arg0)
       throws stockMarket.exceptions.StockNotFoundException,
              java.rmi.RemoteException;
}
```

A interface remota apresentada estende `Remote`. Além disso, todos os seus métodos ativam `RemoteException`. Por fim, o tipo do parâmetro `arg1` dos métodos `subscribe` e `unsubscribe` foi substituído pelo tipo da interface remota associada, mostrada abaixo, a qual também é gerada automaticamente por DAJ.

```
public interface StockListener extends Remote {
  void update(stockMarket.StockInfo arg0) throws RemoteException;
}
```

Nesse caso, os métodos cujas assinaturas tornam-se incompatíveis, em razão da incompatibilidade de tipos dos parâmetros do método ou do tipo de retorno, têm o sufixo `_daj_javarmi` adicionado ao nome do método. A adição desse sufixo foi necessária para

diferenciar explicitamente métodos que tiveram suas assinaturas modificadas, uma vez que sobrecarga de métodos em Java não considera nem o tipo de retorno do método nem as exceções ativadas pelo mesmo. Assim, um método remoto com nome e parâmetros idênticos aos do seu respectivo método de negócio, mas que retorna uma referência remota, não poderia ser introduzido em uma classe de negócio.

Além disso, o método `getStock` ativa uma exceção do tipo `StockNotFoundException`, a qual também deve ser ativada pela interface remota para que seja enviada para o cliente do método remoto. Apesar de a exceção ser transferida entre servidor e cliente, não há necessidade de fazer com que a exceção implemente a interface `java.io.Serializable`, pois ambos os tipos de exceção de Java, `Exception` e `RuntimeException`, implementam nativamente essa interface.

Interfaces Remotas em Java IDL: Como usual em aplicações baseadas em CORBA, interfaces remotas devem ser codificadas em IDL, uma linguagem neutra proposta especificamente para definição de interfaces. Portanto, há a necessidade da existência de uma especificação em IDL das interfaces de negócio definidas no descritor de distribuição do sistema. Com o objetivo de isentar o desenvolvedor da aplicação de conhecer os detalhes de codificação dessa interface, o que pode resultar em erros e dificultar a utilização do sistema, DAJ fornece uma funcionalidade para geração da especificação IDL a partir das informações contidas no descritor de distribuição.

Em seguida, o sistema se encarrega de chamar o compilador `idlj`, o qual integra a plataforma Java IDL. Este compilador gera diversas classes e interfaces utilizadas na implementação de aplicações distribuídas baseadas em Java IDL. Em implementações que não utilizam DAJ, o uso destas classes fica entrelaçado com a implementação das classes de negócio do sistema.

A fim de ilustrar a geração de interfaces em Java IDL, mostra-se a seguir a interface `StockMarketOperations`, gerada pelo compilador `idlj`. Em sistemas baseados em Java IDL, esta interface deve ser implementada por uma classe de negócio.

```
public interface StockMarketOperations {
  void update_daj_javaidl(daj.javaidl.api.stockMarket.StockInfo info);
  void subscribe_daj_javaidl(String stock,
    daj.javaidl.api.stockMarket.StockListener obj);
  void unsubscribe_daj_javaidl(String stock,
    daj.javaidl.api.stockMarket.StockListener obj);
  daj.javaidl.api.stockMarket.StockInfo getStock_daj_javaidl (String stockName)
    throws daj.javaidl.api.stockMarket.exceptions.StockNotFoundException;
```

```
}
```

É interessante notar que esta interface também adota a substituição de tipos de negócio por tipos remotos, conforme proposto para interfaces remotas em Java RMI. No entanto, o tipo de exceção ativada pelo método remoto é diferente entre os sistemas Java RMI e Java IDL. Em Java RMI, a exceção ativada é do mesmo tipo, tanto na interface de negócio como na interface remota equivalente. Em Java IDL, a ferramenta de geração de código `idlj` produz uma exceção que é utilizada apenas na interface remota. Além disso, esta exceção, a qual chamamos de exceção remota, estende uma classe interna da plataforma Java IDL, o que inibe sua substituição pela exceção definida pelo desenvolvedor, a qual chamamos de exceção de negócio. Maiores detalhes sobre o tratamento dado às exceções são discutidos na Seção 3.4.6.

3.4.2 Classes Remotas

Em DAJ, uma classe remota é aquela resultante da introdução, em uma classe de negócio, de código de distribuição requerido por uma determinada plataforma de *middleware*. Dentre este código, por exemplo, encontra-se aquele responsável por informar a interface remota que deve ser implementada pela classe. Na implementação de DAJ, faz-se uma distinção entre classes remotas cujas instâncias serão registradas em um servidor de nomes e classes remotas cujas instâncias não são registradas em um servidor de nomes. As primeiras são sempre declaradas em um nodo `server` de um descritor de distribuição. Como exemplo, podemos citar a classe `StockMarketImpl`. As últimas são declaradas em um nodo `remote`, sendo usadas, portanto, para instanciar objetos que serão passados por referência remota em chamadas de métodos remotos. Como exemplo, podemos citar a classe `StockListenerImpl`.

Apesar desta distinção, a transformação necessária em ambos os tipos de classes remotas é o mesmo. A seguir, descreve-se como estes dois tipos de classes remotas são adaptadas com código de distribuição requerido tanto por Java IDL quanto por Java RMI.

Classes remotas em Java IDL: Como qualquer classe remota, estas classes devem implementar a interface remota gerada por DAJ. Além disso, para cada método desta interface que tenha como parâmetro ou o retorno um tipo `T` que seja uma interface remota, deve-se introduzir na classe um método correspondente, o qual possuirá o mesmo nome do método de negócio seguido pelo sufixo `_daj_javaidl`. O código deste método deve chamar o método já implementado na classe contendo como parâmetro o tipo da interface

de negócio associada a `T`.

Para ilustrar, mostra-se a seguir o aspecto gerado por DAJ para transformar a classe de negócio `StockMarketImpl` em uma classe remota.

```
 1: public privileged aspect Remoting_StockMarketImpl {
 2:
 3:   declare parents: stockMarket.StockMarketImpl
 4:     implements daj.javaidl.api.stockMarket.StockMarketOperations;
 5:
 6:   public void stockMarket.StockMarketImpl.subscribe_daj_javaidl(
 7:     String arg0, daj.javaidl.api.stockMarket.StockListener arg1) {
 8:       subscribe(arg0, new daj.javaidl.api.stockMarket.StockListenerAdapter(arg1));
 9:   }
10:
11:   public void stockMarket.StockMarketImpl.unsubscribe_daj_javaidl(
12:     String arg0, daj.javaidl.api.stockMarket.StockListener arg1) {
13:       unsubscribe(arg0, new daj.javaidl.api.stockMarket.StockListenerAdapter(arg1));
14:   }
15:
16:   public void stockMarket.StockMarketImpl.update_daj_javaidl(
17:     daj.javaidl.api.stockMarket.StockInfo arg0) {
18:       update((stockMarket.StockInfo)arg0);
19:   }
20:
21:   public daj.javaidl.api.stockMarket.StockInfo
22:     stockMarket.StockMarketImpl.getStock_daj_javaidl (String arg0)
23:       throws daj.javaidl.api.stockMarket.exceptions.StockNotFoundException {
24:         try {
25:           return (daj.javaidl.api.stockMarket.StockInfo)getStock(arg0);
26:         }
27:         catch (stockMarket.exceptions.StockNotFoundException e) { }
28:         throw new RuntimeException("");
29:   }
30:
31:   private daj.javaidl.api.stockMarket.StockMarket
32:     stockMarket.StockMarketImpl.refIDL = null;
33:
34:   public daj.javaidl.api.stockMarket.StockMarket
35:     stockMarket.StockMarketImpl.export2IDL() {
36:       if (refIDL == null) {
37:         try {
38:           String[] settings = {"ORBInitialHost","localhost"};
39:           org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(settings, null);
40:
```

```
41:           org.omg.PortableServer.POA rootPOA =
42:             org.omg.PortableServer.POAHelper.narrow(
43:               orb.resolve_initial_references("RootPOA"));
44:           rootPOA.the_POAManager().activate();
45:
46:           daj.javaidl.api.stockMarket.StockMarketPOATie tie =
47:             new daj.javaidl.api.stockMarket.StockMarketPOATie(this, rootPOA);
48:
49:           refIDL = tie._this(orb);
50:         }
51:         catch (org.omg.CORBA.ORBPackage.InvalidName e) {
52:           e.printStackTrace();
53:         }
54:         catch (org.omg.PortableServer.POAManagerPackage.AdapterInactive e) {
55:           e.printStackTrace();
56:         }
57:       }
58:       return refIDL;
59:   }
60: }
```

Na linha 1, o aspecto é declarado como privilegiado (`privileged`) para permitir a adaptação da classe `StockMarketImpl`, mesmo que a classe ou qualquer membro dela sejam declarados como privados ou protegidos . Nas linhas 3 e 4, indica-se que a classe de negócio `StockMarketImpl` deve implementar a interface remota `StockMarketOperations` (mostrada na Seção 3.4.1). O método `subscribe_daj_javaidl` desta interface não é implementado na classe de negócio, apesar dela possuir um método semanticamente equivalente, mas com parâmetros diferentes. Assim, torna-se necessária a introdução desse método na classe de negócio. Nesse caso, o método de negócio `subscribe` implementado pelo usuário na classe `StockMarketImpl` espera um parâmetro do tipo de negócio `StockListener`, e não do tipo remoto associado a `StockListener`, conforme especificado em `StockMarketOperations`. De forma similar ao método `subscribe`, os métodos `unsubscribe` e o método `update` têm seus argumentos adaptados antes da chamada ser redirecionada para o método equivalente que implementa a lógica de negócio. Já o método `getStock`, além de redirecionar a chamada e adaptar o tipo de retorno, captura uma eventual exceção ativada pelo método de negócio (linha 27). O tratamento dado à exceção capturada será descrito na Seção 3.4.6.

Na linha 8 do aspecto mostrado anteriormente, o código do método `subscribe_daj_javaidl` redireciona a chamada para o método de negócio semanticamente equivalente, codificado pelo desenvolvedor da classe `StockMarketImpl`. Para que esse redirecionamento

ocorra deve-se instanciar um objeto adaptador da seguinte classe:

```
 1: public class StockListenerAdapter implements stockMarket.StockListener {
 2:
 3:   daj.javaidl.api.stockMarket.StockListener adaptee;
 4:
 5:   public StockListenerAdapter(daj.javaidl.api.stockMarket.StockListener adaptee) {
 6:       this.adaptee = adaptee;
 7:   }
 8:
 9:   public void update (stockMarket.StockInfo arg0) {
10:     adaptee.update_daj_javaidl ((daj.javaidl.api.stockMarket.StockInfo)arg0);
11:   }
12:
13:   public daj.javaidl.api.stockMarket.StockListener export2IDL() {
14:      return adaptee;
15:   }
16: }
```

Este adaptador implementa a interface de negócio (linha 1) e internamente possui uma referência para o *stub* do objeto remoto (linha 3), a qual é informada ao adaptador através do seu construtor (linhas 5 a 7). Na implementação dos métodos da interface de negócio, o adaptador redireciona a chamada para um método remoto equivalente, invocado sobre a referência remota encapsulada em seu interior, efetuando, quando necessário, ajustes nos argumentos ou no tipo de retorno do método remoto.

O aspecto mostrado anteriormente também introduz dois novos membros nas classes de negócio: um método, chamado `export2IDL`, responsável por realizar a ativação de objetos destas classes (linhas 34 a 59) e uma referência para o *stub* retornado pelo processo de ativação[1] (linhas 31 e 32). Basicamente, o método introduzido consulta o valor desta referência (linha 36). Se ela for nula, realiza-se a ativação do objeto remoto (linhas 38 a 47) e o *stub* associado ao mesmo é atribuído à referência (linha 49). No caso de objetos servidores, esta referência será utilizada para registrar o objeto remoto em um servidor de nomes através do método `registryRemoteObject`, invocado pelo usuário da aplicação. Outra forma de se registrar um servidor é através da classe de ativação, que será descrita na seção 3.4.4.

Algumas aplicações podem trocar referências remotas entre seus *hosts*, por exemplo, quando um *host* A envia uma referência remota para um *host* B. Se o *host* B enviar essa

[1]Em Java IDL, objetos remotos devem ser sempre ativados após serem instanciados, independentemente de serem ou não registrados em um servidor de nomes. Objetos são ativados chamando-se o método `activate`, o qual retorna uma referência para o *stub* associado ao objeto. Esta referência deve, no restante do programa, ser usada no lugar da referência original para o objeto.

mesma referência para um terceiro *host*, ele irá se valer do mesmo mecanismo usado por A para obter a referência remota, ou seja, ele invocará o método `export2IDL`. Por isso, esse método é inserido na classe adaptadora mostrada (linhas 13 a 15).

Classes remotas em Java RMI: De forma semelhante à abordagem usada no código de distribuição de Java IDL, a classe que codifica a lógica de negócio do serviço remoto deve implementar a interface remota gerada por DAJ. Métodos desta interface que esperam como parâmetro ou retornam como resultado tipos remotos devem ser introduzidos na classe de negócio. Além disso, também são adicionados mecanismos para geração, armazenamento e recuperação do *stub* do objeto remoto.

Mostra-se abaixo o aspecto gerado por DAJ para transformar a classe de negócio `StockMarketImpl` em uma classe remota.

```
 1: public privileged aspect Remoting_StockMarketImpl {
 2:
 3:   declare parents: stockMarket.StockMarketImpl
 4:     implements daj.javarmi.api.stockMarket.StockMarket;
 5:
 6:   public void stockMarket.StockMarketImpl.subscribe_daj_javarmi(
 7:     String arg0, daj.javarmi.api.stockMarket.StockListener arg1) {
 8:       subscribe(arg0,
 9:         new daj.javarmi.api.stockMarket.StockListenerAdapter(arg1));
10:   }
11:
12:   public void stockMarket.StockMarketImpl.unsubscribe_daj_javarmi(
13:     String arg0, daj.javarmi.api.stockMarket.StockListener arg1) {
14:       unsubscribe(arg0,
15:         new daj.javarmi.api.stockMarket.StockListenerAdapter(arg1));
16:   }
17:
18:   private daj.javarmi.api.stockMarket.StockMarket
19:     stockMarket.StockMarketImpl.refRMI = null;
20:
21:   public daj.javarmi.api.stockMarket.StockMarket
22:     stockMarket.StockMarketImpl.export2RMI() {
23:       if (refRMI == null) {
24:         try {
25:           refRMI = (daj.javarmi.api.stockMarket.StockMarket)
```

```
26:             java.rmi.server.UnicastRemoteObject.exportObject(this, 0);
27:         } catch(Exception e) {
28:           e.printStackTrace();
29:         }
30:       }
31:       return refRMI;
32:   }
33: }
```

Inicialmente, o aspecto mostrado, através dos mecanismos de transversalidade estática de AspectJ, faz com que a classe de negócio implemente a interface remota `StockMarket` (linhas 3 e 4). Em seguida, os métodos incompatíveis, em relação à interface de negócio, são introduzidos na classe de negócio, com a respectiva adaptação dos argumentos incompatíveis e posterior redirecionamento da chamada para o método que implementa a lógica de negócio do sistema (linhas 6 a 10 e 12 a 16). Por fim, também é introduzida uma referência remota para o objeto (linhas 18 e 19) e um método para geração, ativação e recuperação dessa referência (linhas 21 a 32). Esse método introduzido, será utilizado tanto para registro de serviços através do método `registryRemoteObject`, quanto para ativação de objetos que receberão chamadas remotas através de *callback*.

3.4.3 Obtenção de Referências Remotas

Conforme afirmado na Seção 3.3.2, clientes obtêm referências para servidores declarados em um descritor de distribuição chamando o método `getReference`. A implementação deste método realiza o *parser* do documento XML que representa o descritor de distribuição, identifica o servidor para o qual está sendo solicitada uma referência e efetua uma consulta ao servidor de nomes de Java RMI ou Java IDL.

No entanto, o método `getReference` devolve para o cliente não a referência retornada pelo servidor de nomes, mas uma referência para um *proxy* deste objeto. Este *proxy*, cuja classe é gerada pela implementação de DAJ, funciona como um representante local do objeto remoto e tem três funções: capturar as exceções remotas lançadas pela plataforma de *middleware* em caso de falhas de comunicação, adaptar tanto os argumentos enviados pelos métodos remotos quanto o resultado esperado pelo cliente e ativar o objeto remoto, quando o mesmo é utilizado como argumento de uma chamada remota de método. Para realizar esta ativação, o *proxy* utiliza o método `export2IDL` ou `export2RMI` introduzido em classes remotas pelo aspecto descrito na Seção 3.4.2.

Mostra-se a seguir o *proxy* instanciado quando o cliente solicita uma referência para um servidor da classe `StockMarketImpl`, utilizando Java IDL como protocolo de comunicação.

```
 1: public class StockMarketProxy implements stockMarket.StockMarket {
 2:
 3:   daj.javaidl.api.stockMarket.StockMarket server;
 4:
 5:   public StockMarketProxy(daj.javaidl.api.stockMarket.StockMarket server) {
 6:     this.server = server;
 7:   }
 8:
 9:   public void subscribe(java.lang.String arg0, stockMarket.StockListener arg1) {
10:     server.subscribe_daj_javaidl(arg0, (daj.javaidl.api.stockMarket.StockListener)
11:       daj.solver.SolveReference.javaidlSolve(arg1));
12:   }
13:
14:   public void unsubscribe(java.lang.String arg0, stockMarket.StockListener arg1) {
15:     server.unsubscribe_daj_javaidl(arg0, (daj.javaidl.api.stockMarket.StockListener)
16:       daj.solver.SolveReference.javaidlSolve(arg1));
17:   }
18:
19:   public stockMarket.StockInfo getStock(java.lang.String arg0)
20:     throws stockMarket.exceptions.StockNotFoundException {
21:       try {
22:         return (stockMarket.StockInfo)server.getStock_daj_javaidl(arg0);
23:       }
24:       catch (daj.javaidl.api.stockMarket.exceptions.StockNotFoundException e) { }
25:       throw new RuntimeException("");
26:   }
27:
28:   public void update(stockMarket.StockInfo arg0) {
29:     server.update_daj_javaidl((daj.javaidl.api.stockMarket.StockInfo)arg0);
30:   }
31: }
```

Da mesma forma que ocorre nos objetos adaptadores vistos na Seção 3.4.2, a classe *proxy* implementa uma interface de negócio, que é a mesma utilizada pelo cliente para especificação dos métodos oferecidos pelo objeto remoto (linha 1). Além disso, ela encapsula uma referência para um objeto remoto (linha 3) e é responsável por interceptar as chamadas efetuadas pelo cliente e de redirecioná-las para este objeto remoto. Em alguns casos, há a necessidade de efetuar o *type casting* dos argumentos passados por serialização, como no caso dos métodos `update` e `getStock`, em outros há a necessidade de substituir o argumento por sua referência remota. A geração dessa referência remota é realizada pelo método `javaidlSolve` para Java IDL e pelo método `javarmiSolve` para Java RMI. Estes métodos, através dos recursos de reflexão computacional de Java, executam o método

`export2IDL` ou `export2RMI`, e obtém a referência remota do objeto.

3.4.4 Ativação de Objetos Servidores

Conforme afirmado na Seção 3.3.3, em DAJ existem aspectos responsáveis pela instanciação, ativação e registro de objetos servidores, de forma que usuários do sistema não precisam se preocupar com a codificação destas tarefas.

No caso específico de Java RMI, o seguinte aspecto abstrato é usado para definir *pointcuts*, *advices* e métodos relacionados com a implementação de tais funções:

```
 1: public abstract aspect RMIServerSide {
 2:
 3:   public abstract pointcut registryRemoteObject();
 4:   public abstract String getServerName();
 5:   public abstract int getHostPort();
 6:   public abstract Remote getInstanceObject();
 7:
 8:   void around(): registryRemoteObject() {
 9:     try {
10:       Remote objInstance = getInstanceObject();
11:
12:       java.rmi.Remote stub = (java.rmi.Remote)
13:         UnicastRemoteObject.exportObject(objInstance, getHostPort());
14:
15:       Registry registry = LocateRegistry.getRegistry();
16:       registry.rebind(getServerName(), stub);
17:     } catch(Exception e) {
18:        e.printStackTrace();
19:     }
20:   }
21: }
```

Neste aspecto, o *pointcut* abstrato `registryRemoteObject` denota os pontos do programa onde deve-se instanciar um objeto servidor (linha 3). O método abstrato `getServerName` é usado para obter o nome com o qual o objeto servidor será registrado no servidor de nomes de Java RMI (linha 4). A porta TCP/IP de acesso ao serviço de transporte do servidor de nomes é obtida pelo método `getHostPort` (linha 5). Já o método `getInstanceObject` é usado para obter uma instância do servidor (linha 6). Por fim, um *advice* associado ao *pointcut* `registryRemoteObject` se encarrega de instanciar, registrar e ativar um objeto remoto segundo as convenções de Java RMI (linhas 8 a 20).

Exemplo: Mostra-se a seguir o aspecto concreto `RMIServerSide_StocketMarketB` gerado automaticamente pelo sistema DAJ para instanciar e registrar o servidor `StockMarketB`, definido no descritor de distribuição mostrado na Seção 3.3.1.

```
 1: public aspect RMIServerSide_StockMarketB
 2:   extends RMIServerSide {
 3:
 4:   public pointcut registryRemoteObject():
 5:     execution(public static void Server_StockMarketB.main(..));
 6:
 7:   public String getServerName() {
 8:     return "StockMarketB";
 9:   }
10:
11:   public int getHostPort() {
12:     return 0;
13:   }
14:
15:   public java.rmi.Remote getInstanceObject() {
16:     return new stockMarket.StockMarketImpl();
17:   }
18: }
```

Em DAJ, existe ainda um aspecto abstrato chamado `IDLServerSide`, o qual é semelhante ao aspecto `RMIServerSide`. Entretanto, neste aspecto, o *advice* associado ao *pointcut* `registryRemoteObject` instancia e registra um objeto remoto segundo a interface de programação de Java IDL. Este aspecto é mostrado a seguir:

```
 1: public abstract aspect IDLServerSide {
 2:
 3:   abstract pointcut registryRemoteObject();
 4:   abstract String getServerName();
 5:   abstract int getHostPort();
 6:   abstract org.omg.CORBA.Object getInstanceObject(ORB orb, POA rootPOA);
 7:
 8:   void around(): registryRemoteObject() {
 9:     try {
10:       String [] settings = null;
11:        if (getHostPort() > 0)
12:          settings = new String[] {"-ORBInitialPort", String.valueOf(getHostPort())};
13:          ORB orb = ORB.init(settings, null);
14:
15:          POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
```

```
16:          rootPOA.the_POAManager().activate();
17:
18:          org.omg.CORBA.Object objRef = getInstanceObject(orb, rootPOA);
19:
20:          org.omg.CORBA.Object nsRef =
21:            orb.resolve_initial_references("NameService");
22:          NamingContextExt ncRef = NamingContextExtHelper.narrow(nsRef);
23:          NameComponent path[] = ncRef.to_name(getServerName());
24:
25:          ncRef.rebind(path, objRef);
26:          orb.run();
27:      }
28:      catch(Exception e) {
29:          e.printStackTrace();
30:      }
31:  }
31: }
```

Exemplo: Mostra-se a seguir o aspecto concreto `IDLServerSide_StocketMarketA`, responsável por instanciar e registrar o servidor `StockMarketA`, definido no descritor de distribuição mostrado na Seção 3.3.1.

```
 1: public aspect IDLServerSide_StockMarketA
 2:   extends IDLServerSide {
 3:
 4:   public pointcut registryRemoteObject():
 5:     execution(public static void Server_StockMarketA.main(..));
 6:
 7:   public String getSystemName() {
 8:     return "StockMarketA";
 9:   }
10:
11:   public int getHostPort() {
12:     return 0;
13:   }
14:
15:   public org.omg.CORBA.Object getInstanceObject(ORB orb, POA rootPOA) {
17:       stockMarket.StockMarketImpl objInstance = new stockMarket.StockMarketImpl();
18:       daj.javaidl.api.stockMarket.StockMarketPOATie tie  =
19:         new daj.javaidl.api.stockMarket.StockMarketPOATie(objInstance, rootPOA);
20:       return tie._this(orb);
21:   }
22: }
```

Como pode ser observado, em ambos os aspectos concretos (`RMIServerSide_StockMarketB` e `IDLServerSide_StockMarketA`), os parâmetros que diferenciam o código, tais como, nome do servidor, porta de acesso ao *registry* e nome da classe do servidor, são obtidos do descritor de distribuição.

3.4.5 Serialização de Objetos

Em plataformas de *middleware* orientadas por objetos, objetos podem ser passados por cópia entre os nodos de uma rede, como argumentos de métodos remotos ou como resultado da chamada desses métodos. Ocorre que, efetivamente, nas camadas internas de um protocolo de rede, esses objetos necessitam ser convertidos em uma seqüência de *bytes*, que pode ser transferida entre essas camadas, devendo o receptor dessa seqüência efetuar a reconstituição do objeto. Esse processo de transformação é denominado serialização e desserialização. Plataformas de *middleware* adotam políticas e mecanismos distintos de serialização. A seguir descreveremos esse processo para as plataformas Java IDL e Java RMI.

Serialização de Objetos em Java IDL: Em Java IDL, objetos passados por serialização devem implementar a interface `org.omg.CORBA.portable.StreamableValue`, a qual define os métodos que uma classe deve implementar para serialização de seus objetos. Estes métodos serão invocados pelas camadas de mais baixo nível da plataforma. Em Java IDL, o mecanismo de serialização consiste em fazer com que a classe serializável implemente uma interface específica da plataforma e introduzir na mesma a implementação dos métodos dessa interface.

A ferramenta `idlj` produz classes com o código necessário para serializar e desserializar instâncias de classes serializáveis. Nas classes geradas são introduzidos, inclusive, seus atributos. No entanto, esses atributos também foram especificados na classe de negócio. Além disso, os atributos gerados são especificados como sendo públicos ou protegidos, o que pode impactar na lógica de negócio do sistema quando da necessidade de se especificar atributos privados. Essas classes geradas são utilizadas pelos *stubs*, também gerados por `idlj`, não sendo possível, por isso, excluí-las do projeto. A abordagem adotada por DAJ consiste em remover as classes geradas e, em seu lugar, gerar interfaces vazias, as quais implementam a interface de serialização requerida por Java IDL. As interfaces geradas reproduzem a hierarquia de herança das classes sobrescritas.

Exemplo: A seguir apresenta-se a classe `StockInfo`, produzida por DAJ, que substituiu a classe de mesmo nome gerada por `idlj`.

```
1: public interface StockInfo
2:   extends org.omg.CORBA.portable.StreamableValue { }
```

Mostra-se a seguir o aspecto `StockInfo_Serializer`, o qual é responsável por tornar a classe `StockInfo` serializável segundo as convenções de Java IDL.

```
1: public privileged aspect StockInfo_Serializer {
2:
3:   declare parents: stockMarket.StockInfo implements
4:     daj.javaidl.api.stockMarket.StockInfo;
5:
6:   private static String[] stockMarket.StockInfo._truncatable_ids = {
7:     daj.javaidl.api.stockMarket.StockInfoHelper.id()
8:   };
9:
10:  public String[] stockMarket.StockInfo._truncatable_ids() {
11:    return _truncatable_ids;
12:  }
13:
14:  public void stockMarket.StockInfo._write(
15:    org.omg.CORBA.portable.OutputStream ostream) {
16:    ostream.write_string(this.stock);
17:    ostream.write_double(this.value);
18:    daj.javaidl.api.stockMarket.DateTimeHelper.write(ostream, this.date_time);
19:  }
20:
21:  public void stockMarket.StockInfo._read(
22:    org.omg.CORBA.portable.InputStream istream) {
23:    this.stock = istream.read_string();
24:    this.value = istream.read_double();
25:    this.date_time = (stockMarket.DateTime)
26:      daj.javaidl.api.stockMarket.DateTimeHelper.read(istream);
27:  }
28:
29:  public org.omg.CORBA.TypeCode stockMarket.StockInfo._type() {
30:    return daj.javaidl.api.stockMarket.StockInfoHelper.type();
31:  }
32: }
```

O aspecto é definido como privilegiado para permitir que atributos privados da classe de negócio sejam acessados, tanto para leitura quanto para escrita (linha 1). Nas linhas 3 e 4, indica-se que a classe de negócio deve implementar a interface de serialização. Assim, a classe de negócio pode ser manipulada pelos *stubs* específicos da plataforma. Por

implementar a interface de serialização, são introduzidos nela os métodos definidos nessa interface (linhas 6 a 31). Dentre os métodos introduzidos, o método `_write` é responsável pela serialização do objeto e o método `_read` é responsável pela desserialização. Observe que a ordem na qual os atributos da classe são serializados (linhas 16 a 18) é a mesma na qual eles devem ser desserializados (linhas 23 a 26). Além disso, os atributos de tipos primitivos são serializados (linhas 16 e 17) e desserializados (linhas 23 e 24) diretamente. Já os atributos que referenciam objetos são enviados para uma classe responsável por efetuar a sua serialização (linha 18) e desserialização (linhas 25 e 26) daquele objeto. A Tabela 3.1 ilustra os tipos primitivos suportados por Java IDL e o seu respectivo mapeamento para a linguagem Java.

Tipo Java	**Tipo Java IDL**
boolean	boolean
char	char, wchar
byte	octet
String	string, wstring
short	short, unsigned short
int	long, unsigned long
long	long long, unsigned long long
float	float
double	double

Tabela 3.1: Mapeamento de tipos de Java para Java IDL

Serialização de Objetos em Java RMI: O processo de serialização em Java RMI se vale das vantagens de se apoiar no sistema de tipos da linguagem Java. Por isso, o processo é mais simples do que o de Java IDL. Em Java RMI, para tornar uma classe serializável basta fazer com que ela implemente a interface `java.io.Serializable`.

Exemplo: Mostra-se a seguir o aspecto `StockInfo_Serializer`, o qual é responsável por tornar a classe `StockInfo` serializável segundo as convenções de Java RMI.

```
1: public privileged aspect StockInfo_Serializer {
2:   declare parents: stockMarket.StockInfo implements java.io.Serializable;
3: }
```

Da mesma forma que em Java IDL, o aspecto é definido como privilegiado para possibilitar visibilidade integral de atributos da classe `StockInfo`, ainda que eles sejam privados ou protegidos (linha 1). Na linha 2, define-se que a classe `StockInfo` deve implementar a interface de serialização de Java RMI.

3.4.6 Tratamento de Exceções

Em um ambiente distribuído, plataformas de *middleware* são responsáveis por transmitir exceções ativadas em servidores para aplicações cliente. A seguir, descreve-se como essas exceções são tratadas nas plataformas Java IDL e Java RMI.

Tratamento de Exceções em Java IDL: Em Java IDL, as exceções ativadas por um método devem ser especificadas na definição de interfaces IDL. A partir dessa definição, a ferramenta `idlj` produz classes específicas para serialização dessas exceções. Na Seção 3.4.1 podemos observar que o método `getStock`, da interface remota `StockMarketOperations`, ativa uma exceção do tipo `StockNotFoundException`, produzida por `idlj`. Essas classes de exceções estendem uma classe específica da plataforma Java IDL, chamada `org.omg.CORBA.UserException`. Como afirmado anteriormente, e conforme ilustrado pela classe `StockNotFoundException`, as exceções de negócio estendem uma classe específica de Java, assim não é possível fazer com que a exceção de negócio estenda a exceção remota, uma vez que Java não possui herança múltipla.

```
1: public class StockNotFoundException extends Exception {
2:   public StockNotFoundException(String dsc) {
3:     super(dsc);
4:   }
5: }
```

A solução proposta por DAJ para contornar as restrições descritas anteriormente, consiste em criar um aspecto que capture a ocorrência de uma exceção, e que realize a transformação da exceção ativada numa exceção suportada pelo ambiente na qual será transmitida. Ou seja, se uma exceção de negócio for ativada no servidor, ela deverá ser transformada em uma exceção remota, a qual é serializável, para que esta possa ser enviada até o lado da aplicação cliente. Ao chegar no lado do cliente, a exceção remota deverá ser, novamente, transformada em uma exceção de negócio, a qual será repassada à aplicação cliente.

Exemplo: Mostra-se a seguir o aspecto `ExceptionHandler_StockNotFoundException`, o qual é responsável por transformar exceções remotas em exceções de negócio e vice-versa, em Java IDL.

```
1: public privileged aspect ExceptionHandler_StockNotFoundException {
2:   before(stockMarket.exceptions.StockNotFoundException e)
3:     throws daj.javaidl.api.stockMarket.exceptions.StockNotFoundException:
4:       handler(stockMarket.exceptions.StockNotFoundException)
```

```
 5:       && args(e)
 6:       && (withincode(* *.*_daj_javaidl(..))) {
 7:         daj.javaidl.api.stockMarket.exceptions.StockNotFoundException _e =
 8:           new daj.javaidl.api.stockMarket.exceptions.StockNotFoundException();
 9:         _e.detailMessage = e.getMessage();
10:         _e.setStackTrace(e.getStackTrace());
11:         throw _e;
12:   }
13:
14:   before(daj.javaidl.api.stockMarket.exceptions.StockNotFoundException e)
15:     throws stockMarket.exceptions.StockNotFoundException:
16:       handler(daj.javaidl.api.stockMarket.exceptions.StockNotFoundException)
17:       && args(e)
18:       && (withincode(* daj.javaidl.api.stockMarket.StockMarketProxy.*(..))) {
19:         stockMarket.exceptions.StockNotFoundException _e =
20:           new stockMarket.exceptions.StockNotFoundException(e.detailMessage);
21:         _e.setStackTrace(e.getStackTrace());
22:         throw _e;
23:   }
24: }
```

Define-se o aspecto como privilegiado para permitir visibilidade integral da estrutura das classes associadas (linha 1). Nesse aspecto, existem dois *advices* do tipo `before` que capturam exceções ativadas no lado do servidor (linhas 2 a 12) e exceções ativadas no lado do cliente (linhas 14 a 23). Além disso, esses *advices* ativam exceções compatíveis com o contexto em que são capturadas, ou seja, no lado do servidor são ativadas exceções remotas (linha 3), já que elas serão enviadas para o cliente, e no lado do cliente são ativadas exceções de negócio (linha 15), que serão enviadas para o invocador do método remoto. O processo de transformação consiste, em ambos os casos, em capturar a exceção e transferir os valores de seus atributos para uma exceção equivalente. Assim, no lado do servidor (primeiro *advice*), o construtor `handler` captura o bloco `catch` da exceção ativada (linha 4). Já o construtor `args` captura a exceção lançada (linha 5) e o *joinpoint* da linha 6 especifica os pontos de junção onde essa interceptação deve ocorrer, ou seja, em todos os métodos de classes de negócio em que foram introduzidos métodos para redirecionamento de chamadas remotas para seus respectivos métodos de negócio. Após a captura da exceção, ela será transformada em uma exceção de tipo compatível com o ambiente em que será lançada, nesse caso, uma exceção serializável (linhas 7 a 10) e, por fim, será ativada. No lado do cliente será realizado um processo semelhante ao utilizado no lado do servidor.

Tratamento de Exceções em Java RMI: Em Java RMI, inexistem as complexidades descritas para Java IDL, pois Java RMI se aproveita do fato de utilizar o mesmo sistema de tipos da linguagem Java. Assim, em Java RMI, ao contrário do ocorre em Java IDL, não há a necessidade de realizar qualquer intervenção em relação às exceções ativadas pelos métodos da interface de negócio, uma vez que essas exceções já são serializáveis.

Apesar disso, Java RMI utiliza uma exceção verificável própria para tratamento de erros de comunicação, adicionada na interface remota mostrada na Seção 3.4.1. Essa exceção, chamada `java.rmi.RemoteException`, deve ser tratada pela aplicação cliente. Como o cliente não introduz o código de tratamento dessa exceção no núcleo da aplicação, a abordagem proposta por DAJ consiste em transformar essa exceção verificável em uma exceção não-verificável, isentando assim o tratamento explícito desse tipo de exceção por parte da aplicação cliente.

Exemplo: Mostra-se a seguir os aspectos responsáveis por transformar a exceção `RemoteException` de verificável para não-verificável.

```
1: public aspect StockMarketProxy_Exception {
2:   declare soft: java.rmi.RemoteException :
3:     withincode(* daj.javarmi.api.stockMarket.StockMarketProxy.*(..));
4: }
5:
6: public aspect StockListenerAdapter_Exception {
7:   declare soft: java.rmi.RemoteException :
8:     withincode(* daj.javarmi.api.stockMarket.StockListenerAdapter.*(..));
9: }
```

O primeiro aspecto, chamado `StockMarketProxy_Exception`, captura as exceções ativadas na classe *proxy* utilizada pelo cliente. O construtor `declare soft` é responsável por suavizar a exceção definida na linha 2, quando a mesma for ativada nos pontos de junção definidos pelo conjunto de junção especificado na linha 3. Ou seja, todas as exceções do tipo `java.rmi.RemoteException` que forem ativadas no interior de qualquer método da classe `StockMarketProxy` serão transformadas em exceções não-verificáveis.

De forma similar, o segundo aspecto também realiza essa transformação. No entanto ela ocorre no lado do servidor, no interior dos adaptadores que encapsulam os parâmetros transmitidos com semântica de referência remota.

3.5 Ferramenta `dajc`

A geração manual dos componentes necessários para introdução do requisito de distribuição em uma aplicação, ainda que beneficiada pela sistemática proposta por DAJ, é uma tarefa tediosa e sujeita a erros. Além disso, o programador necessita dominar detalhes e convenções requeridas por plataformas de *middleware*, além de conceitos de programação orientada por aspectos. Com o objetivo de sanar esses problemas, o sistema DAJ inclui uma ferramenta, chamada `dajc`, que produz todo o código necessário para introdução do requisito de distribuição em uma aplicação. Para tanto, essa ferramenta recebe como entrada tanto o núcleo da aplicação quanto os descritores de distribuição.

Na inicialização da ferramenta `dajc`, configura-se a localização dos descritores de distribuição e a relação dos descritores utilizados, pois o sistema pode não requerer descritores de serialização ou de objetos que recebem chamadas remotas por *callback*. Além disso, pode ser solicitada a produção de código de distribuição para ambas as plataformas de *middleware* suportadas por DAJ, independentemente do protocolo especificado no descritor de distribuição. Essa opção permite a troca do protocolo de comunicação utilizado pelo sistema distribuído, através do descritor de servidores remotos, sem a necessidade de recompilar o sistema.

A geração dos componentes de distribuição é controlada por módulos independentes, para que o processo de produção seja coordenado de forma mais eficiente, produzindo apenas o código necessário para tornar uma aplicação distribuída, independentemente de qualquer configuração realizada de forma desnecessária nos descritores de distribuição. Por exemplo, se no descritor de distribuição for configurado um objeto como serializável, mas esse objeto não for utilizado em nenhum método remoto, então não será produzido código para serialização desse objeto.

3.5.1 Geração de Código de Distribuição para Java RMI

A seguir são apresentados os módulos da ferramenta `dajc` responsáveis por gerar os componentes necessários para introdução do requisito de distribuição em uma aplicação segundo as convenções requeridas pela plataforma Java RMI.

Gerador de interfaces remotas (`RemoteInterfaceGenerator`): É responsável por produzir interfaces remotas segundo as convenções requeridas por Java RMI. Durante o processo de geração, esse módulo preserva a hierarquia de herança da interface sendo gerada. Além disso, esse gerador avalia os tipos usados nos métodos da interface para verificar a necessidade de realizar adaptações na assinatura dos mesmos, conforme descrito

na Seção 3.4.1.

Gerador de classes *proxy* (`ProxyGenerator`): As classes *proxy* constituem os representantes locais do objeto remoto utilizado pelo usuário. Assim, elas são geradas a partir da especificação dos servidores remotos. Nesse caso, o gerador deve avaliar os argumentos do método invocado, o tipo de retorno e as exceções ativadas pelo método para realizar as adaptações necessárias, conforme descrito na Seção 3.4.3.

Gerador de classes adaptadoras (`AdapterGenerator`): As classes adaptadoras são semelhantes às classes *proxy*, exceto pelo fato de serem produzidas a partir das interfaces remotas de objetos que são passados com semântica de referência remota. Além disso, elas são utilizadas para encapsular as referências para esses objetos remotos antes de eles serem enviados para métodos que implementam regras de negócio da aplicação, conforme descrito na Seção 3.4.2.

Gerador de aspectos para serialização (`SerializableAspectGenerator`): Em Java RMI, objetos passados com semântica de serialização devem implementar uma interface da API de Java chamada `java.io.Serializable`, conforme descrito na Seção 3.4.5. Esse módulo de geração de código produz aspectos que, através de recursos de transversalidade estática de AspectJ, introduzem nas classes de negócio essa convenção requerida por Java RMI. Além disso, o gerador não produz aspectos de serialização para subclasses de classes para as quais já foram produzidos tais aspectos.

Gerador de aspectos para registro e ativação de objetos servidores (`ActivateAspectGenerator`): Esse gerador de código produz aspectos concretos, a partir do descritor de objetos servidores, que estendem o aspecto abstrato `RMIServerSide`. Além disso, conforme descrito na Seção 3.4.4, esse gerador produz uma classe que, juntamente com esses aspectos, instancia, registra e ativa o objeto remoto em um servidor de nomes.

Gerador de aspectos para adaptação de classes remotas (`RemoteAspectGenerator`): O aspecto para adaptação das classes remotas descrito na Seção 3.4.2 é produzido por esse gerador de código. Nele são avaliados os métodos introduzidos na classe de negócio para permitir chamadas de métodos remotos com passagem de parâmetros via uma semântica de referência remota.

3.5.2 Geração de Código de Distribuição para Java IDL

Nessa seção são apresentados os módulos que produzem código requerido por Java IDL para introdução do requisito de distribuição em uma aplicação.

Gerador da especificação IDL (`IDLFileGenerator`): A interface IDL é o ponto de partida de qualquer desenvolvimento baseado em CORBA. Como um dos principais objetivos do sistema DAJ é ocultar dos desenvolvedores detalhes e convenções requeridos por Java IDL, esse gerador de código produz a especificação da interface IDL de acordo com as especificações contidas nos descritores de distribuição.

Gerador de interfaces serializáveis (`SerializableClassGenerator`): A ferramenta `idlj` produz classes com código para serialização e desserialização de objetos, as quais são utilizadas pelos *stubs* também produzidos por `idlj`. A ferramenta `idlj` também introduz atributos nas classes geradas, introduzindo assim redundância de dados, uma vez que a classe de negócio equivalente também possui tais atributos. Para evitar essa redundância e facilitar o acessos às informações armazenadas nos atributos das classes de negócio, `dajc` remove as classes de serialização produzidas por `idlj` e, em seu lugar, introduz interfaces vazias que implementam a interface de serialização requerida por Java IDL, conforme discutido na Seção 3.4.5.

Gerador de fábricas de objetos serializáveis (`SerializableDefaultFactoryGenerator`): A ferramenta `idlj` também produz classes fabricadoras de objetos serializáveis. No entanto, nessas classes são criadas instâncias de classes produzidas pela ferramenta `idlj`, as quais são incompatíveis com as classes de negócio equivalentes. Assim, esse gerador remove essas classes e introduz classes equivalentes para instanciação de objetos serializáveis do núcleo da aplicação.

Gerador de classes *proxy* (`ProxyGenerator`): De forma similar a Java RMI, as classes *proxy* de Java IDL também são responsáveis por interceptar chamadas realizadas pelos clientes. Em seguida, quando necessário, realizam a adaptação dos argumentos e redirecionam a chamada para o objeto remoto. Além disso, elas também são responsáveis por capturar as exceções remotas para permitir que as mesmas possam ser convertidas em exceções de negócio. Na Seção 3.4.3, mostra-se uma classe *proxy* produzida pela ferramenta `dajc`, chamada `StockMarketProxy`.

Gerador de classes adaptadoras (`AdapterGenerator`): As classes adaptadoras são semelhantes às classes *proxy*, exceto pelo fato de serem produzidas a partir das interfaces

dos objetos que recebem chamadas remotas por *callback*. Uma classe adaptadora produzida por `dajc` para Java IDL, chamada `StockListenerAdapter`, é mostrada na Seção 3.4.2.

Gerador de aspectos para serialização (`SerializableAspectGenerator`): Como as classes de serialização produzidas por `idlj` são substituídas por interfaces vazias, as classes de negócio equivalentes tomam o lugar das classes serializáveis no momento da passagem de parâmetros. Para isso, são introduzidas nessas classes as convenções de serialização requeridas por Java IDL, conforme descrito na Seção 3.4.5. Os aspectos que introduzem essas convenções são produzidos por esse gerador.

Gerador de aspectos para registro e ativação de objetos servidores (`ActivateAspectGenerator`): De forma similar a Java RMI, esse gerador de código cria componentes que instanciam, registram e ativam objetos remotos segundo as convenções requeridas por Java IDL.

Gerador de aspectos para adaptação de classes remotas (`RemoteAspectGenerator`): Esse gerador produz aspectos que adaptam as classes de negócio segundo as convenções requeridas por Java IDL para que suas instâncias possam receber chamadas remotas, conforme descrito na Seção 3.4.2.

Gerador de aspectos para transformação de exceções (`ExceptionHandlerGenerator`): Conforme afirmado, as exceções produzidas pela ferramenta `idlj` não são compatíveis com exceções de negócio. Assim, esse gerador produz os aspectos de adaptação de exceções discutidos na Seção 3.4.6.

3.6 Conclusão

Java RMI e Java IDL requerem que classes de negócio de uma aplicação distribuída sigam determinadas convenções e protocolos de codificação, os quais possuem um comportamento transversal. Pela sistemática proposta por DAJ, demonstrada nessa seção, é possível modularizar todo o requisito de distribuição demandado por essas duas plataformas de *middleware* em módulos independentes dos requisitos funcionais do sistema.

Capítulo 4

Estudo de Caso

Neste capítulo são avaliadas as potencialidades do sistema DAJ através da utilização do sistema para introdução do interesse de distribuição em três aplicações. Essas aplicações incluem um sistema, chamado *Health Watcher*, usado por cidadãos para enviar reclamações sobre restaurantes e similares a órgãos públicos responsáveis pela vigilância sanitária destes estabelecimentos, um sistema para controle de ações negociadas em uma bolsa de valores, chamado NPS, e uma aplicação para controle do acervo de uma biblioteca, chamado *Library*.

4.1 Sistema Health Watcher

O sistema *Health Watcher* é um sistema para gerenciamento de reclamações realizadas por cidadãos acerca das condições sanitárias de estabelecimentos comerciais da área de alimentação, como, por exemplo, restaurantes e lanchonetes. As reclamações são agrupadas em categorias e podem ser utilizadas por instituições preocupadas com a saúde da população. Essas instituições podem utilizar as informações coletadas para investigar riscos iminentes à saúde, podendo, desta forma, atuar de forma preventiva em relação a esse problema.

4.1.1 Arquitetura do Sistema

O sistema é implementado em Java e organizado em uma estrutura de camadas e padrões de projeto, que são usados para prover uma arquitetura modular e extensível. Desta forma, a arquitetura do sistema ajuda a separar os requisitos de gerenciamento de dados, negócio, distribuição e interface com o usuário. Assim, o código desses requisitos

torna-se menos entrelaçado e espalhado entre módulos do sistema, mas não completamente.

O acesso ao sistema é realizado através de uma interface baseada na Web, na qual as reclamações são registradas ou consultadas. Essa interface realiza o papel de uma camada de apresentação do sistema, sendo utilizada somente para requisição de serviços e exibição de resultados. As requisições submetidas através dessa interface são enviadas para *servlets*, os quais realizam a chamada ao servidor remoto. Esses *servlets* acessam o sistema através de uma fachada chamada `HWFacade`, a qual fornece acesso aos módulos que implementam a lógica de negócio do sistema.

A versão orientada por objetos do sistema *Health Watcher* utiliza a plataforma de *middleware* Java RMI para prover comunicação entre cliente e servidor e possui código de distribuição entrelaçado e espalhado por diversas classes do sistema. Esse entrelaçamento é mais significativo na fachada de acesso ao sistema (`HWFacade`), na codificação da aplicação cliente (*servlets*) e na interface que o cliente usa para invocar os métodos remotos (`IRemoteFacade`). Além disso, há código espalhado nas classes cujos objetos são utilizados como parâmetros formais de métodos remotos ou como tipo de retorno desses métodos.

A Tabela 4.1 apresenta algumas métricas do sistema. O núcleo do sistema *Health Watcher* possui 78 classes e 13 interfaces. Além disso, alguns requisitos de distribuição, como serviço de localização, exceções remotas e sincronização, são implementados por meio de 4 classes e 2 interfaces. Apesar do agrupamento dessas métricas em categorias, também existe código de distribuição disperso nas classes que constituem o núcleo da aplicação.

	Classes	**Interfaces**	**Linhas de Código (LOC)**
Núcleo da aplicação	78	13	4976
Código específico de distribuição	4	2	153
Total	82	15	5129

Tabela 4.1: Métricas da versão orientada por objetos do sistema *Health Watcher*.

4.1.2 Aplicação de DAJ no Sistema Health Watcher

Para que o sistema DAJ fosse aplicado ao sistema *Health Watcher* houve, inicialmente, a necessidade de se refatorar o código do sistema de forma a remover todo o código de distribuição requerido pela plataforma Java RMI. Após essa etapa, foram identificados os objetos que constituem os serviços remotos do sistema e os objetos passados como parâmetros de métodos ou retornados como resultado dos mesmos. A partir dessa análise,

foram codificados os descritores de distribuição do sistema, os quais são mostrados abaixo:

Servidores: No descritor de distribuição do sistema, foram definidos dois servidores: o primeiro utiliza Java RMI para comunicação, da mesma forma que a versão original do sistema, sendo registrado com o nome `HealthWatcher`; o segundo utiliza Java IDL para comunicação, como uma alternativa a Java RMI, sendo registrado com o nome `HealthWatcherIDL`.

```
<services>
    <server id="HealthWatcher">
        <interface>gui.IFacade</interface>
        <class>controllers.HealthWatcherFacade</class>
        <protocol>JavaRMI</protocol>
        <serverName>localhost</serverName>
    </server>
    <server id="HealthWatcherIDL">
        <interface>gui.IFacade</interface>
        <class>controllers.HealthWatcherFacade</class>
        <protocol>JavaIDL</protocol>
        <serverName>localhost</serverName>
    </server>
</services>
```

Objetos passados por serialização: Em *Health Watcher*, objetos passados por serialização são definidos por meio do seguinte descritor de distribuição:

```
<serializable>
    <class>addresses.Address</class>
    <class>complaint.Complaint</class>
    <class>complaint.IComplaintRepository</class>
    <class>complaint.AnimalComplaint</class>
    <class>complaint.FoodComplaint</class>
    <class>complaint.SpecialComplaint</class>
    <class>complaint.ComplaintRepositoryArray</class>
    <class>complaint.DiseaseType</class>
    <class>complaint.IDiseaseRepository</class>
    <class>complaint.DiseaseTypeRepositoryArray</class>
    <class>complaint.Symptom</class>
```

```
13:     <class>complaint.ISymptomRepository</class>
14:     <class>complaint.SymptomRepositoryArray</class>
15:     <class>employee.Employee</class>
16:     <class>employee.IEmployeeRepository</class>
17:     <class>employee.EmployeeRepositoryArray</class>
18:     <class>healthguide.ISpecialityRepository</class>
19:     <class>healthguide.HealthUnit</class>
20:     <class>healthguide.IHealthUnitRepository</class>
21:     <class>healthguide.HealthUnitRepositoryArray</class>
22:     <class>healthguide.MedicalSpeciality</class>
23:     <class>healthguide.SpecialityRepositoryArray</class>
24:     <class>util.IteratorDsk</class>
25:     <class>util.IIterableRepository</class>
26:     <class>util.ConcreteIterator</class>
27:     <class>util.Date</class>
28:     <class>util.Schedule</class>
29: </serializable>
```

No Sistema *Health Watcher*, não existem objetos passados por referência remota.

Por fim, tanto o núcleo do sistema, isento de código de distribuição, quanto os descritores de distribuição foram usados como entrada da ferramenta `dajc`, que produz classes e aspectos de distribuição para as duas plataformas de *middleware* especificadas no descritor de distribuição. A Tabela 4.2 apresenta as métricas apuradas para o sistema *Health Watcher* após a utilização da ferramenta `dajc`.

	Classes	**Interfaces**	**Aspectos**	**LOC**
1. Núcleo da aplicação	78	13	0	4566
2. Módulos internos de DAJ	6	0	0	244
3. DAJ - Módulos gerados para Java RMI	2	1	21	191
4. DAJ - Módulos gerados para Java IDL	103	29	38	5671
5. Total	189	43	59	10672

Tabela 4.2: Métricas da versão DAJ do sistema *Health Watcher*.

No núcleo da versão de *Health Watcher* baseada em DAJ, ainda que o número de classes e interfaces tenha sido mantido, houve uma redução no número de linhas de código (8.24%), decorrente da remoção de 410 linhas de código de distribuição que se encontrava espalhado pelo núcleo da versão orientada por objetos do sistema. A seguir, serão descritas versões de implantação do sistema *Health Watcher*.

Versão do Sistema *Health Watcher* com suporte a Java RMI: Na versão do sistema

com suporte a Java RMI, a introdução do requisito de distribuição é realizada através da combinação do núcleo da aplicação (linha 1 da Tabela 4.2) com classes, interfaces e aspectos responsáveis por realizar as adaptações necessárias para tornar a aplicação distribuída (linha 3). Além disso, são usadas algumas classes internas de DAJ, as quais são essenciais ao funcionamento do sistema (linha 2).

O código gerado para suporte a comunicação via Java RMI inclui uma interface remota, uma classe *proxy* e uma classe para registro do objeto remoto. Além disso, DAJ gera 21 aspectos, dos quais 17 são para serialização de objetos, um para tratamento de exceções, um para adaptação do objeto remoto e dois para ativação e registro do objeto remoto.

Versão do sistema *Health Watcher* com suporte a Java IDL: De forma semelhante ao que ocorre com a versão com suporte a Java RMI, na versão Java IDL a introdução do requisito de distribuição consiste na combinação do núcleo da aplicação (linha 1 da Tabela 4.2) com classes e aspectos específicos de Java IDL (linha 4) que transformarão o núcleo em uma aplicação distribuída. Da mesma forma, é necessário introduzir algumas classes internas de DAJ (linha 2).

Nesta versão, DAJ introduz uma quantidade maior de classes e aspectos em relação à versão do sistema com suporte a Java RMI, em virtude das convenções requeridas pela plataforma Java IDL.

Das 132 classes e interfaces geradas, apenas duas foram produzidas por `dajc`: uma que implementa o *proxy* utilizado pelo cliente para invocar métodos remotos e outra responsável por instanciar, ativar e registrar o objeto remoto, as 130 classes restantes foram geradas através da ferramenta `idlj`. Destas, 27 classes de serialização foram substituídas por interfaces vazias produzidas por `dajc` (conforme descrito na Seção 3.4.5). Em relação aos 38 aspectos gerados, 27 são utilizados para serialização de objetos, 8 para tratamento de exceções, um para adaptação do objeto remoto e dois para ativação e registro do objeto remoto.

Versão de DAJ com suporte múltiplo a Java RMI e Java IDL: A versão integrada do sistema, com suporte simultâneo a Java RMI e Java IDL, requer a combinação do núcleo da aplicação (linha 1 da Tabela 4.2) com classes e aspectos específicos de Java RMI (linha 3) e de Java IDL (linha 4). Para o funcionamento do sistema, há também a necessidade de se introduzir classes específicas da API de DAJ (linha 2).

Para essa versão integrada são geradas todas as classes, interfaces e aspectos descritos anteriormente nessa Seção. Portanto, são utilizadas 189 classes, 43 interfaces e 59 aspectos, os quais totalizam 10.672 linhas de código.

4.1.3 Avaliação do uso de DAJ no Sistema Health Watcher

A ferramenta de geração de código de DAJ produziu classes e aspectos tanto para Java RMI quanto para Java IDL. A partir do código gerado, testes foram realizados com objetivo de se verificar o correto funcionamento da aplicação distribuída em relação a estas duas plataformas de *middleware*. Em todos os testes realizados, os resultados foram aqueles esperados. Foi possível também validar a adaptação da aplicação para uma segunda plataforma de *middleware* (Java IDL), para a qual não havia suporte inicial.

Foram feitas modificações no sistema para que fosse possível a um cliente acessar dois servidores, cada um respondendo a chamadas remotas usando plataformas de *middleware* distintas. Além disso, foram feitos testes de reconfiguração do sistema, através dos descritores de distribuição, como, por exemplo, a mudança do protocolo de comunicação do serviço. Em todos esses cenários, os resultados obtidos foram satisfatórios e confirmaram o poder de expressão do sistema DAJ.

4.2 Sistema NPS

O sistema NPS (*Network Pricing System*) é um sistema financeiro para controle de ações negociadas em uma bolsa de valores, usado como objeto de estudo no livro *Programming with Java IDL* [LBS97]. Neste sistema, clientes podem manifestar interesse em serem notificados de alterações nos preços de determinadas ações negociadas em uma bolsa de valores. Além disso, o sistema fornece funcionalidades para submissão de atualizações nos preços de ações e também para controle de autenticação de clientes.

Em linhas gerais, NPS é semelhante ao exemplo motivador descrito na Seção 3. No entanto, como foi originalmente implementado usando Java IDL, decidiu-se utilizá-lo como estudo de caso nesta Seção para se analisar a viabilidade do emprego de DAJ em sistemas desenvolvidos nesta plataforma de *middleware*.

4.2.1 Arquitetura do Sistema

No sistema NPS, existem dois servidores remotos, configurados no descritor de distribuição como `Legacy` e `QuoteFeedAdmin`. O primeiro é utilizado por clientes que atualizam valores de ações. Para isso eles utilizam uma interface chamada `Feed`. O segundo servidor é utilizado por clientes interessados em serem notificados de alterações nos preços de ações. Nesse caso, esses clientes devem, primeiramente, serem autenticados, para então registrarem as companhias para as quais desejam receber as notificações de alterações nos preços de suas ações. Para o recebimento dessas notificações, esses clientes informam o

nome da companhia e um objeto responsável por receber as notificações, o qual deverá ser passado com semântica de referência remota. A classe deste objeto deve implementar uma interface chamada `FeedFactory`.

A versão original do Sistema NPS utiliza a plataforma de *middleware* Java IDL. Nessa versão, o código de distribuição requerido por Java IDL encontra-se espalhado e entrelaçado em diversas classes do sistema: em todas as classes remotas, em todas as classes cujos objetos são enviados com semântica de serialização e em todas as classes clientes de serviços remotos.

Na Tabela 4.3 são apresentadas algumas métricas da versão orientada por objetos do sistema NPS. Ainda que existam classes responsáveis apenas por código específico de distribuição, o núcleo da aplicação também possui código de distribuição espalhado e entrelaçado por suas classes.

	Classes	**Interfaces**	**LOC**
Núcleo da aplicação	18	1	1279
Código específico de distribuição	61	14	3138
Total	79	15	4417

Tabela 4.3: Métricas da versão orientada por objetos do sistema *NPS*.

4.2.2 Aplicação de DAJ no Sistema NPS

Inicialmente, foi necessário remover todo o código de distribuição existente no sistema. Após essa etapa, houve a necessidade de se incorporar novas classes no sistema, uma vez que algumas classes, em especial as classes dos objetos enviados com semântica de serialização, foram produzidas pela ferramenta `idlj`. Em outras palavras, o núcleo da versão orientada por objetos do sistema NPS não pode ser compilado separadamente, pois utiliza classes produzidas pela ferramenta `idlj`. Após a introdução dessas classes, o sistema pode ser compilado e testado localmente antes da introdução do requisito de distribuição. Após essa etapa de refatoração, foram identificadas as classes dos objetos servidores do sistema, as classes dos objetos enviados com semântica de referência remota e dos objetos enviados com semântica de serialização. Baseando-se nessa classificação, foram codificados os descritores de distribuição requeridos por DAJ, os quais são apresentados a seguir.

Servidores: conforme afirmado, o sistema possui dois servidores: um utilizado por clientes que querem ser notificados de alterações nos valores de determinadas ações e outro utilizado por clientes que submetem atualizações nos valores de ações. Com o objetivo de

avaliar as potencialidades e flexibilidades do sistema DAJ, para cada servidor do sistema foram configurados dois servidores remotos: um utilizando Java IDL, da mesma forma que na versão orientada por objetos do sistema, e outro utilizando Java RMI:

```
<services>
    <server id="Legacy">
        <interface>legacy.Feed</interface>
        <class>legacy.FeedImpl</class>
        <protocol>JavaIDL</protocol>
        <serverName>localhost</serverName>
    </server>
    <server id="LegacyRMI">
        <interface>legacy.Feed</interface>
        <class>legacy.FeedImpl</class>
        <protocol>JavaRMI</protocol>
        <serverName>localhost</serverName>
    </server>
    <server id="QuoteFeedAdmin">
        <interface>nps.FeedFactory</interface>
        <class>server.FeedFactoryImpl</class>
        <protocol>JavaIDL</protocol>
        <serverName>localhost</serverName>
    </server>
    <server id="QuoteFeedAdminRMI">
        <interface>nps.FeedFactory</interface>
        <class>server.FeedFactoryImpl</class>
        <protocol>JavaRMI</protocol>
        <serverName>localhost</serverName>
    </server>
</services>
```

Objetos passados por referência remota: apresenta-se a seguir o descritor de distribuição por meio do qual são definidas as interfaces e classes dos objetos enviados com a semântica de referência remota:

```
<remoting>
    <remote interface="nps.QuoteReceiver">
        <class>client.TestQuoteReceiverImpl</class>
        <class>client.QuoteReceiverImpl</class>
```

```
 5:     </remote>
 6:     <remote interface="legacy.Receiver">
 7:         <class>legacy.ReceiverImpl</class>
 8:     </remote>
 9:     <remote interface="npsprivate.QuoteFilter">
10:         <class>server.QuoteFilterImpl</class>
11:     </remote>
12:     <remote interface="nps.FilteredQuoteFeed" />
13: </remoting>
```

Objetos passados por serialização: Na versão refatorada do sistema NPS, objetos enviados por serialização são definidos por meio do seguinte descritor de distribuição:

```
1: <serializable>
2:     <class>auth.AuthPacket</class>
3:     <class>auth.Profile</class>
4:     <class>legacy.Quote</class>
5:     <class>nps.Quote</class>
6: </serializable>
```

Após a codificação dos descritores de distribuição, tanto esses descritores quanto o núcleo do sistema foram usados como entrada da ferramenta `dajc`, que produz classes e aspectos de distribuição. A Tabela 4.4 apresenta as métricas aferidas para o sistema NPS após a utilização da ferramenta `dajc`.

	Classes	**Interfaces**	**Aspectos**	**LOC**
1. Núcleo da aplicação	18	1	0	1104
2. Módulos adicionados ao núcleo	7	7	0	106
3. Módulos internos de DAJ	6	0	0	244
4. DAJ - Módulos gerados para Java RMI	7	7	18	355
5. DAJ - Módulos gerados para Java IDL	65	18	15	3744
6. Total	103	33	33	5553

Tabela 4.4: Métricas da versão DAJ do sistema NPS.

Considerando apenas as classes e interfaces do núcleo da versão orientada por objetos, houve uma redução no número de linhas de código de 13,68%. Se levarmos em consideração os componentes introduzidos no núcleo pela versão baseada em DAJ, ainda que o número de componentes (classes, interfaces e aspectos) aumente em 73,68%, há uma redução do número de linhas de código em 5,39%. A seguir, serão descritas as versões de implantação do sistema NPS.

Versão do sistema NPS com suporte a Java IDL: Na versão do sistema com suporte a Java IDL, o requisito de distribuição é introduzido a partir da combinação do núcleo da aplicação (linhas 1 e 2 da Tabela 4.4) com classes e aspectos específicos da plataforma de *middleware* Java IDL (linha 5). A partir dessa combinação, tem-se uma versão de implantação do sistema, a qual utiliza também algumas classes internas da plataforma DAJ (linha 3).

Nesta versão do sistema, ainda que tenha sido removido todo o código de distribuição do núcleo da aplicação, existem diversas classes e interfaces comuns à versão orientada por objetos. Isto ocorre devido à utilização, tanto na versão orientada por objetos quanto na versão DAJ, da ferramenta `idlj` para geração de código específico dessa plataforma de *middleware*.

Em relação à versão orientada por objetos do sistema, houve aumento no número de componentes em 45,74%, resultante, principalmente, da introdução de: 14 componentes adicionados ao núcleo (linha 2), 6 classes específicas de DAJ (linha 3), 15 aspectos de distribuição (linha 5), 4 classes *proxy* e 6 classes adaptadoras. Apesar disso, o número de linhas de código aumentou apenas 17,68%. Ressalte-se, no entanto, que estas classes e aspectos são gerados automaticamente pela ferramenta `dajc`.

Versão do sistema NPS com suporte a Java RMI: Nessa versão do sistema, a introdução do requisito de distribuição requer a combinação do núcleo da aplicação (linhas 1 e 2 da Tabela 4.4) com classes e aspectos responsáveis por adaptar o núcleo e tornar a aplicação distribuída (linha 4). São necessárias ainda classes internas de DAJ, as quais são essenciais ao funcionamento do sistema (linha 3).

Nessa versão, o código gerado para suporte a comunicação através de Java RMI requer a introdução de 7 classes (2 classes *proxy*, 3 classes adaptadoras e 2 classes para ativação e registro do objeto remoto) e de 7 interfaces remotas. Além disso, são introduzidos 18 aspectos, os quais são responsáveis pelos requisitos de serialização (4 aspectos), tratamento de exceções (5 aspectos), ativação e registro de objetos remotos (3 aspectos) e adaptação do objeto remoto às convenções requeridas por Java RMI (6 aspectos).

Em relação à versão orientada por objetos do sistema NPS, houve uma redução no número de componentes em 24,47%, com respectiva redução no número de linhas de código em 59,04%. Essa redução deve-se ao modelo de desenvolvimento requerido por Java RMI, que beneficia-se do fato de operar no mesmo ambiente no qual o núcleo da aplicação foi implementado.

Versão do sistema NPS com suporte múltiplo a Java RMI e Java IDL: A versão

do sistema NPS com suporte múltiplo requer a combinação de todas as classes, interfaces e aspectos apresentados na Tabela 4.4. Nessa versão, são utilizadas 103 classes, 33 interfaces e 33 aspectos, totalizando 5.553 linhas de código.

4.2.3 Avaliação do uso de DAJ no Sistema NPS

De forma semelhante ao descrito para o Sistema *Health Watcher*, todo o código produzido por DAJ foi testado tanto para Java RMI quanto para Java IDL. Em todos os testes realizados os resultados obtidos estavam em consonância com os esperados. Assim, foi possível criar uma versão da aplicação compatível com uma segunda plataforma de *middleware*, neste caso a plataforma Java RMI, para a qual não havia suporte na versão original do sistema.

Foram realizadas reconfigurações nas versões de implantação do sistema, de forma a permitir que clientes pudessem acessar os servidores utilizando plataformas de *middleware* distintas, ou seja, o cliente que publica alterações no valor de ações utiliza Java IDL e o cliente que recebe as notificações dos valores das ações utiliza Java RMI, ou vice-versa. Além disso, foram realizados testes com ambos os tipos de clientes utilizando uma única plataforma de *middleware*. Em todos os cenários avaliados, os resultados obtidos foram aqueles esperados, confirmando tanto as potencialidades do sistema DAJ quanto o poder de expressão oferecido pelos descritores de distribuição.

Outro benefício derivado do uso de DAJ no sistema NPS foi a possibilidade de se realizar testes funcionais do núcleo da aplicação antes mesmo de se introduzir qualquer requisito de distribuição. Tal facilidade não é viável em sistemas que utilizam diretamente Java IDL como plataforma de *middleware*, uma vez que eles se tornam fortemente dependentes das classes geradas pela ferramenta `idlj`.

4.3 Sistema *Library*

O sistema *Library* provê um conjunto de funcionalidades para controle de rotinas de uma biblioteca. O sistema foi usado como estudo de caso em [BLL04] para validar um sistema para engenharia reversa de diagrama de seqüência UML. Esse sistema possui três categorias de usuários: administradores, que são responsáveis por cadastrar os funcionários da biblioteca; funcionários, que são responsáveis por gerenciar tanto o acervo bibliográfico quanto os clientes e, por fim, os clientes, que solicitam reservas de exemplares e realizam consultas.

4.3.1 Arquitetura do Sistema

O sistema *Library* possui três servidores remotos, os quais são configurados no descritor de distribuição como `AdministratorControl`, `EmployeeControl` e `CustomerControl`. O servidor `AdministratorControl` é utilizado pelos usuários responsáveis por cadastrar os funcionários da biblioteca. Para tanto, eles utilizam uma interface chamada `AdministratorControlIF`. Já o servidor `EmployeeControl` é utilizado por funcionários da biblioteca para realizar manutenções no acervo, incluindo tarefas como controle de reservas, empréstimos e pagamentos. Para isso eles utilizam uma interface chamada `EmployeeControlIF`. Por fim, o servidor `CustomerControl` é utilizado pelos clientes da biblioteca para consultas no acervo, assim como para realização de reservas. Nesse caso, os clientes utilizam uma interface chamada `CustomerControlIF`.

A versão original do sistema *Library* utiliza Java RMI como *middleware* de comunicação. Na Tabela 4.5 são apresentadas as métricas apuradas para esta versão do sistema. Como pode ser observado, não existem classes específicas para o requisito de distribuição. Assim, diferentemente do sistema *Health Watcher* e do sistema NPS, as métricas apresentadas referem-se apenas ao núcleo da aplicação. Apesar disso, esse núcleo também possui código de distribuição espalhado e entrelaçado por suas classes e interfaces.

	Classes	**Interfaces**	**LOC**
Núcleo da aplicação	66	4	4997

Tabela 4.5: Métricas da versão orientada por objetos do sistema *Library*.

4.3.2 Aplicação de DAJ no Sistema *Library*

Em AspectJ, o processo de costura (*weaving*) dos requisitos, introduzidos através de mecanismos de transversalidade, é dependente do código fonte das classes nas quais esses requisitos são introduzidos. Assim, a utilização de classes compiladas inviabiliza a introdução do requisito de distribuição no sistema. Portanto, é condição necessária para garantir a devida introdução do requisito de distribuição em um sistema, que o código fonte das classes cujos objetos estão presentes no processo de comunicação, como por exemplo classes remotas e classes serializáveis, esteja ao alcance do compilador de AspectJ.

Desta forma, além de remover todo o código de distribuição existente no sistema *Library*, foi necessário também identificar classes compiladas, como por exemplo àquelas da API de Java, que são utilizadas no processo de comunicação do sistema. Nesta etapa foram identificadas as classes `Vector`, `GregorianCalendar` e `Exception`, para as quais foram implementadas classes equivalentes. Após essa refatoração, foram identificados os

objetos servidores e os objetos enviados com semântica de serialização. A partir dessa classificação, foram codificados os descritores de distribuição requeridos por DAJ, que são apresentados a seguir.

Servidores: Conforme afirmado, o sistema *Library* possui três servidores: um é utilizado por usuários administradores para gerenciar contas de funcionários, outro é utilizado por funcionários para controlar o acervo e as contas dos clientes da biblioteca e, por fim, um terceiro servidor é utilizado pelos clientes para consulta e reserva de exemplares. Da mesma forma que nos sistemas descritos anteriormente, para cada servidor do sistema foram configurados dois servidores remotos: um utilizando Java RMI e outro utilizando Java IDL.

```
 1: <services>
 2:     <server id="AdministratorControl">
 3:         <interface>server.AdministratorControlIF</interface>
 4:         <class>server.AdministratorControl</class>
 5:         <protocol>JavaRMI</protocol>
 6:         <serverName>localhost</serverName>
 7:     </server>
 8:     <server id="AdministratorControlIDL">
 9:         <interface>server.AdministratorControlIF</interface>
10:         <class>server.AdministratorControl</class>
11:         <protocol>JavaIDL</protocol>
12:         <serverName>localhost</serverName>
13:     </server>
14:     <server id="EmployeeControl">
15:         <interface>server.EmployeeControlIF</interface>
16:         <class>server.EmployeeControl</class>
17:         <protocol>JavaRMI</protocol>
18:         <serverName>localhost</serverName>
719:     </server>
20:     <server id="EmployeeControlIDL">
21:         <interface>server.EmployeeControlIF</interface>
22:         <class>server.EmployeeControl</class>
23:         <protocol>JavaIDL</protocol>
24:         <serverName>localhost</serverName>
25:     </server>
26:     <server id="CustomerControl">
27:         <interface>server.CustomerControlIF</interface>
```

```
28:         <class>server.CustomerControl</class>
29:         <protocol>JavaRMI</protocol>
30:         <serverName>localhost</serverName>
31:     </server>
32:     <server id="CustomerControlIDL">
33:         <interface>server.CustomerControlIF</interface>
34:         <class>server.CustomerControl</class>
35:         <protocol>JavaIDL</protocol>
36:         <serverName>localhost</serverName>
37:     </server>
38: </services>
```

Objetos passados por serialização: Na versão refatorada do sistema *Library*, foram identificados 27 objetos passados por serialização, os quais são definidos no descritor de distribuição a seguir.

```
 1: <serializable>
 2:     <class>server.Active</class>
 3:     <class>server.Available</class>
 4:     <class>server.Barcode</class>
 5:     <class>server.BusinessRules</class>
 6:     <class>server.Copy</class>
 7:     <class>server.CopyState</class>
 8:     <class>server.Customer</class>
 9:     <class>server.CustomerState</class>
10:     <class>server.Employee</class>
11:     <class>server.Error</class>
12:     <class>server.Holding</class>
13:     <class>server.Id</class>
14:     <class>server.Index</class>
15:     <class>server.Loan</class>
16:     <class>server.LoanSuspended</class>
17:     <class>server.OnReserve</class>
18:     <class>server.Reservation</class>
19:     <class>server.ReservationState</class>
20:     <class>server.ReserveSuspended</class>
21:     <class>server.Suspended</class>
22:     <class>server.Title</class>
23:     <class>server.Unavailable</class>
```

```
24:     <class>server.Waiting</class>
25:     <class>util.DateTime</class>
26:     <class>util.Value</class>
27:     <class>util.VectorString</class>
28:     <class>util.VectorValue</class>
29: </serializable>
```

No sistema *Library* não existe descritor de distribuição para objetos enviados com semântica de referência remota, uma vez que nesse sistema objetos são passados apenas por serialização.

Os descritores de distribuição apresentados anteriormente, juntamente com o núcleo da versão refatorada do sistema *Library*, foram fornecidos como entrada para a ferramenta `dajc`, a qual gerou então classes e aspectos de distribuição em conformidade com as especificações definidas. A Tabela 4.6 apresenta informações sobre o código gerado por `dajc`.

	Classes	**Interfaces**	**Aspectos**	**LOC**
1. Núcleo da aplicação	66	4	0	5011
2. Módulos adicionados ao núcleo	4	1	0	179
3. Módulos Internos de DAJ	6	0	0	244
4. DAJ - Módulos gerados para Java RMI	6	3	26	309
5. DAJ - Módulos gerados para Java IDL	98	32	34	5427
6. Total	180	40	60	11170

Tabela 4.6: Métricas da versão DAJ do sistema *Library*.

Observa-se que, em relação ao núcleo da versão orientada por objetos (Tabela 4.5), há um aumento de 0,28% no número de linhas de código. Esse aumento é resultante da introdução de uma classe para tratamento de exceções (chamada `Error`). Na versão original do sistema, objetos remotos ativam exceções do tipo `java.lang.Exception`, a qual é específica da API de Java, o que, como descrito anteriormente, inviabiliza a introdução das convenções requeridas para serialização dessa exceção. Considerando-se os componentes introduzidos no núcleo pela versão baseada em DAJ, há um aumento no número de componentes (7,14%) e no número de linhas de código (3,86%).

A seguir, são apresentadas as versões de implantação do sistema *Library*.

Versão do sistema *Library* com suporte a Java RMI: Em Java RMI, a introdução do requisito de distribuição requer a combinação do núcleo do sistema (linhas 1 e 2 da Tabela 4.6) com as classes e aspectos que adaptam esse núcleo e tornam a aplicação distribuída

(linha 4). Classes internas de DAJ também são incorporadas a essa versão de implantação do sistema.

Essa versão do sistema requer a introdução de 3 classes *proxy*, 3 classes para ativação e registro de objetos remotos, 3 interfaces remotas e 26 aspectos, dos quais 16 são responsáveis pelo requisito de serialização, 4 são usados para ativação e registro de objetos remotos, 3 são destinados ao tratamento de exceções e 3 realizam a adaptação de classes remotas.

Versão do sistema *Library* com suporte a Java IDL: Já a versão do sistema com suporte a Java IDL requer a combinação do núcleo refatorado (linhas 1 e 2 da Tabela 4.6) com classes e aspectos produzidos pela ferramenta `dajc` para Java IDL (linha 5), além de classes internas da plataforma DAJ (linha 3).

Nessa versão do sistema, são introduzidos 127 componentes específicos de Java IDL, distribuídos entre *stubs*, *proxies*, interfaces remotas e outras classes dessa plataforma de *middleware*. Além disso, são introduzidas 3 classes *proxy* e 34 aspectos destinados a adaptar o núcleo da aplicação ao protocolo de programação requerido por Java IDL. Todas essas classes e aspectos são produzidos automaticamente pela ferramenta `dajc`, dispensando o usuário de conhecer qualquer convenção requerida por esta plataforma de *middleware*.

Versão do sistema *Library* com suporte simultâneo a Java RMI e Java IDL: A versão do sistema *Library* com suporte múltiplo requer a combinação de todas as classes e aspectos apresentados na Tabela 4.6.

4.3.3 Avaliação do uso de DAJ no Sistema *Library*

O código de distribuição produzido por DAJ foi testado para as duas plataformas de *middleware* suportadas: Java RMI e Java IDL. De forma semelhante a que ocorreu nos sistemas avaliados anteriormente, os resultados obtidos foram aqueles esperados.

Da mesma forma, reconfigurações nas versões de implantação do sistema foram realizadas para permitir que clientes pudessem acessar os servidores utilizando diferentes plataformas de *middleware*. Em todas as versões avaliadas, os resultados foram satisfatórios.

Ainda que a necessidade de refatoração do código tenha requerido a substituição de algumas classes compiladas, essa exigência não representa uma convenção a ser seguida em razão da utilização do sistema DAJ, mas das plataformas de *middleware* utilizadas. Por exemplo, Java RMI requer que objetos passados com semântica de serialização imple-

mentem a interface `java.io.Serializable`. Se uma determinada classe compilada, que esteja presente no processo de comunicação, não implementar nativamente essa interface, então essa convenção requerida por Java RMI não poderá ser introduzida nessa classe, pois: (i) ela não pode ser introduzida manualmente haja vista, para isso, a necessidade da disponibilidade do código fonte dessa classe; (ii) não é possível realizar essa introdução por meio de mecanismos de transversalidade de AspectJ, pois o compilador de AspectJ necessita do código fonte das classes para realizar as introduções durante o processo de *weaving*.

4.4 Conclusões

Por meio dos estudos de caso apresentados, foi possível validar e comprovar as potencialidades de DAJ. Basicamente, foi possível verificar o uso DAJ em aplicações distribuídas de médio porte. Além disso, as aplicações avaliadas são completas o suficiente para cobrir a maioria dos possíveis cenários envolvidos no processo de distribuição de uma aplicação real. Em todos os casos avaliados, os resultados obtidos foram satisfatórios.

Capítulo 5

Conclusões

Atualmente, distribuição é um interesse presente na grande maioria dos projetos de desenvolvimento de *software*. Daí o sucesso de tecnologias de *middleware*, as quais têm como objetivo encapsular detalhes inerentes à programação em ambientes de redes, tornando o requisito de distribuição transparente para o desenvolvedor e assim aumentando a produtividade. No entanto, paradoxalmente, os sistemas atuais de *middleware* também acrescentam detalhes, convenções e protocolos próprios de programação, os quais devem ser dominados por engenheiros de *software*. Mais importante, estes detalhes possuem um comportamento transversal, se entrelaçando e se espalhando pelo código funcional de uma aplicação. O resultado final é que aplicações distribuídas baseadas em *middleware* não apresentam graus desejados de modularidade, reusabilidade e separação de interesses.

Portanto, desde o final da década de 90, distribuição vêm sendo citada e estudada como um dos principais interesses que podem se beneficiar de implementações orientadas por aspectos. O presente trabalho contribui com estas iniciativas ao propor uma ferramenta orientada por aspectos cujo objetivo final é automatizar e encapsular código de distribuição requerido por duas plataformas de *middleware* bastante utilizadas por desenvolvedores de aplicações em Java. Para alcançar este objetivo, a ferramenta DAJ, que executa o papel de um "*middleware* de *middleware*", se beneficia da sinergia gerada pela combinação de três tecnologias: aspectos (para modularização de código transversal de distribuição), linguagens de domínio específico (para descrição e configuração de parâmetros de distribuição) e geração e transformação de código (para automatizar a criação de aspectos).

5.1 Avaliação de DAJ

As principais contribuições e funcionalidades da ferramenta DAJ foram validadas através da refatoração de três aplicações distribuídas, as quais fazem uso de diversas abstrações normalmente providas por plataformas de *middleware*. Com o uso de DAJ, foi possível encapsular todo código de distribuição destes sistemas. Com isso, as classes de negócio das aplicações permaneceram livres de código entrelaçado demandado pelas plataformas de *middleware* subjacentes. Os descritores de distribuição utilizados em DAJ se mostraram ainda flexíveis e expressivos para modelar diversas (re)configurações de implantação destes sistemas.

A seguir, avalia-se a solução proposta por DAJ, levando-se em consideração critérios como: modularização do requisito de distribuição, portabilidade de plataforma de *middleware*, *obliviousness*. Por fim, descrevem-se tecnologias alternativas à sistemática proposta por DAJ.

5.1.1 Modularização

A sistemática proposta por DAJ, por meio da combinação de padrões de projeto e programação orientada por aspectos, foi capaz de isolar completamente o requisito de distribuição de todos os sistemas avaliados e descritos na Seção 4. Inicialmente, o núcleo de todas essas aplicações foi refatorado, onde o requisito de distribuição foi completamente removido. Entretanto, para alcançar um núcleo independente de qualquer plataforma de *middleware*, houve, em alguns casos, a necessidade de se substituir componentes concretos gerados pela ferramenta `idlj`, produzidos especificamente para a plataforma Java IDL, por componentes independentes de qualquer tecnologia de *middleware*. A mesma abordagem foi adotada para métodos remotos que utilizam componentes previamente compilados, para os quais o código fonte não está ao alcance do compilador de AspectJ. Como exemplos desses componentes pré-compilados podemos citar aqueles da API de Java.

A adoção de um núcleo independente de plataformas de *middleware* é mais simples de se entender, testar e evoluir. Particularmente, testes podem ser realizados sem levar em consideração aspectos de distribuição do sistema, o que torna-se uma característica relevante no caso de desenvolvimento orientados por testes.

5.1.2 Portabilidade

Através do uso de DAJ é possível gerar versões distintas de sistemas distribuídos, utilizando diferentes plataformas de *middleware*, ou ainda utilizando simultaneamente duas plataformas de *middleware*, tornando ainda mais versátil a aplicação. A geração dessas versões requer somente a configuração adequada de um descritor de distribuição.

Entretanto, DAJ fornece suporte apenas a um conjunto limitado de serviços providos pelas plataformas Java IDL e Java RMI. Basicamente, DAJ torna possível a realização de chamadas síncronas de métodos remotos, com semântica de passagem de parâmetros tanto por serialização quanto por referência remota. Desta forma, DAJ não fornece suporte a funcionalidades disponíveis somente em sistemas CORBA, como chamadas assíncronas de métodos, chamadas `oneway`, interfaces de invocação dinâmica etc.

5.1.3 *Obliviousness*

Embora o sistema DAJ isole completamente o núcleo da aplicação do código requerido pelas plataformas de *middleware*, a pergunta que resta responder é a seguinte: DAJ também não introduz novas convenções invasivas de programação, que tornem aplicações distribuídas dependentes do próprio sistema DAJ? Se isto ocorrer, pode-se argumentar que DAJ substitui o entrelaçamento de código decorrente do uso de plataformas de *middleware* por seu próprio código. Felizmente, o grau de dependência e entrelaçamento do sistema é mínimo e pode ser eliminado através da criação de um aspecto extra, como ilustrado abaixo por um cliente hipotético do sistema *StockWatcher*:

```
 1: class MyClient {
 2:   StockMarket s1,s2;
 3:   ....
 4: }
 5: aspect MyClientDependencyInjection {
 6:  after(MyClient c):
 7:   execution(void MyClient.new(..)) && this(s) {
 8:   c.s1= (StockMarket)
 9:         ServiceLocator.getReference("StockMarketA");
10:   c.s2= (StockMarket)
11:         ServiceLocator.getReference("StockMarketB");
12:   }
13: }
```

O aspecto proposto, a exemplo do que ocorre em *frameworks* de injeção de dependência [Mar, CI05], é responsável por obter as referências remotas associadas às variáveis de instância `s1` and `s2`. Desta forma, a classe cliente `MyClient` não possui mais qualquer referência a componentes do sistema DAJ.

5.1.4 Tecnologias Alternativas

Em vez de aspectos, pelo menos duas outras tecnologias poderiam ser usadas para automatizar a geração de código de distribuição requerido por plataformas de *middleware*:

- Ferramentas para manipulação de *bytecodes*: a idéia consiste em usar uma biblioteca para análise e transformação de *bytecodes*, como BCEL [Dah01], para introduzir código de distribuição diretamente nos arquivos resultantes da compilação das classes de negócio de um sistema. Assim, conforme ocorre em DAJ, estas classes permaneceriam livres de código de comunicação. No entanto, linguagens orientadas por aspectos, como AspectJ, oferecem abstrações de mais alto nível para realizar estas mesmas transformações. Estas abstrações contribuíram para simplificar e tornar mais produtivo o projeto e a implementação de DAJ.

- Geração de esqueletos de classes: a idéia consiste em gerar automaticamente esqueletos de classes de negócio com todo o código demandado por uma determinada plataforma de *middleware*. Em outras palavras, dependendo da plataforma escolhida, tais classes estenderiam classes internas do sistema de *middleware*, possuiriam métodos `get` e `set`, possuiriam anotações pré-definidas etc. No caso de métodos de negócio, o código gerado incluiria apenas o cabeçalho do método e uma implementação vazia. Caberia então ao desenvolvedor prover implementação para os métodos com corpo vazio, de acordo com os requisitos funcionais de seu sistema. Esta abordagem é tradicionalmente usada por ferramentas CASE, as quais normalmente incluem funcionalidades para gerar esqueletos de classes a partir de modelos de mais alto nível, como diagramas de classe e de seqüência da UML. No entanto, a mesma possui duas importantes deficiências: (i) o código requerido pelo *middleware*, apesar de não ser implementado manualmente, continua invasivo e espalhado pelo sistema, prejudicando seu entendimento e evolução; (ii) uma vez implementado em uma determinada plataforma de *middleware*, dificulta-se a migração de um sistema distribuído para uma outra plataforma, já que seu código de negócio estaria inserido diretamente no esqueleto das classes geradas para o primeiro *middleware* escolhido. Esta deficiência já foi apontada como um dos principais fatores que levaram ao

insucesso de ferramentas CASE orientadas por objeto com recursos para geração automática de código de distribuição [GSNW02].

5.2 Comparação com Outros Trabalhos

O desenvolvimento de DAJ foi inspirado no trabalho de Soares, Borba e Laureano, através de suas experiências na utilização de AspectJ para prover uma versão orientada por aspectos do sistema *Health Watcher* [SBL06, SLB02]. Os autores mostram como o sistema *Health Watcher* foi reestruturado de forma a modularizar em aspectos o interesse de distribuição, representado por código de comunicação através da plataforma Java RMI. Além de distribuição, os autores apresentam também uma solução para modularização de persistência. Em relação a esse trabalho, DAJ apresenta pelo menos as seguintes contribuições:

- DAJ oferece uma solução para modularização de código de distribuição requerido tanto por Java RMI como por Java IDL, enquanto que em *Health Watcher* propõe-se uma solução apenas para Java RMI. Suporte a Java IDL é importante para garantir interoperabilidade com aplicações construídas em outras linguagens de programação. Do ponto de vista de projeto, o principal desafio enfrentado para prover suporte a uma segunda plataforma de *middleware* foi evitar incompatibilidades e colisões originadas por aspectos específicos de cada uma das plataformas. Por exemplo, no caso de declarações intertipos, optou-se sempre por usar declarações que definem que classes da aplicação alvo devem implementar interfaces e não estender classes, já que Java não possui herança múltipla. O resultado final foi uma solução ortogonal para implementação modularizada de distribuição.

- DAJ permite passagem de objetos usando semântica de serialização e de referência remota em chamadas remotas de métodos. Permite também que chamadas remotas retornem referências para objetos remotos. Estes recursos são normalmente usados por aplicações distribuídas interessadas em serem notificadas via *callback* da ocorrência de determinados eventos. Já o método de implementação proposto por Soares, Borba e Laureano inclui suporte apenas a passagem de objetos com semântica de serialização, sendo esta a única estratégia de passagem de parâmetros usada no sistema *Health Watcher*. Entretanto, chamada remota de métodos com passagem de parâmetros com semântica de referência é usada em outros sistemas distribuídos apoiados por plataformas de *middleware*, em especial aqueles que necessitam receber chamadas remotas por *callbacks* [Fra98]. Por exemplo, o sistema NPS,

descrito na Seção 4.2, se vale de passagem de parâmetros por referência remota.

- DAJ permite a geração de aspectos de distribuição independentemente da arquitetura de *software* adotada pela aplicação base. Por outro lado, a solução proposta em *Health Watcher* é restrita pela arquitetura deste sistema. Por exemplo, assume-se em *Health Watcher* que existe um único objeto servidor (do tipo `HWFacade`). Além disso, chamadas remotas codificadas em clientes são sempre capturadas usando-se este tipo. O resultado é que o *framework* proposto em *Health Watcher* não é compatível, por exemplo, com clientes que possuam múltiplas referências remotas do mesmo tipo, cada uma delas referenciando objetos remotos distintos. Em DAJ, esta situação pode ser modelada facilmente em um descritor de distribuição.

Em [Soa04], Soares descreve uma ferramenta que automaticamente gera aspectos que concretizam os aspectos abstratos propostos em *Health Watcher*. No entanto, a ferramenta proposta é específica para sistemas que seguem a mesma arquitetura de *software* adotada em *Health Watcher*. Já em DAJ, descritores de distribuição adicionam flexibilidade e expressividade ao processo de geração de aspectos, permitindo a modelagem de sistemas distribuídos com as mais diversas arquiteturas e configurações.

Ceccato e Tonella descrevem um *framework* que fornece suporte para migração de uma aplicação não distribuída para uma arquitetura distribuída [CT04]. De forma semelhante ao que ocorre com DAJ, o núcleo da aplicação permanece livre do requisito de distribuição introduzido e todo o código necessário para introdução desse requisito é gerado automaticamente. Entretanto, a solução proposta é restrita à plataforma Java RMI. Além disso, o arquivo de configuração usado no processo de geração de código é basicamente uma lista contento os nomes das classes do sistema, desconsiderando outras informações como estratégias de passagem de parâmetros, servidores de nomes, etc. Na solução proposta por eles, objetos são passados como parâmetros de métodos remotos apenas com semântica de referência remota. Infelizmente, esta abordagem não é adequada para aplicações distribuídas que dependem da realização de chamadas remotas de métodos com semântica de serialização. Além disso, o uso de chamadas remotas com passagem parâmetros apenas por referência remota impactam o desempenho do sistema, pois a execução de um método remoto resulta em uma chamada por *callback* ao processo cliente.

Os descritores de distribuição propostos em DAJ podem ser considerados como uma linguagem orientada por aspectos de domínio específico, para representação de requisitos transversais, típicos de sistemas distribuídos apoiados por tecnologias de *middleware*. Alguns autores, como Wand [Wan03] e Lieberherr [SLS03], consideram que linguagens de domínio específico são veículos mais adequados para modelar e expressar aspectos, já

que as mesmas utilizam vocabulários e abstrações mais próximos do interesse transversal que pretendem modularizar. Além disso, as mesmas são menos propensas às críticas normalmente feitas a linguagens orientadas por aspectos de propósito geral, como violação de encapsulamento e impossibilidade de se raciocinar localmente sobre módulos [Wan03, DW06, Ste06]. Certamente, tais críticas não se aplicam a linguagens orientadas por aspectos de domínio específico. É interessante observar ainda que as primeiras gerações de linguagens orientadas por aspectos eram linguagens de domínio específico. Dentre elas, podemos citar COOL (para sincronização e coordenação) e RIDL (para especificação de interfaces remotas), ambas propostas por Lopes em seu trabalho de doutoramento [Lop97].

5.3 Trabalhos Futuros

Diversas linhas de pesquisa podem ser desenvolvidas a partir de DAJ. Inicialmente, pretende-se investigar o uso de DAJ no desenvolvimento e/ou reestruturação de outras aplicações distribuídas. Pretende-se ainda adicionar suporte a aplicações baseadas em serviços *Web*. Uma primeira iniciativa neste sentido foi descrita em [MV06b].

Pretende-se também investigar o *overhead* introduzido nas aplicações distribuídas apoiadas pelo sistema DAJ em relação às versões originais dessas aplicações, as quais possuem código de distribuição espalhado e entrelaçado pelos requisitos funcionais.

Diversas variantes do sistema podem ser desenvolvidas. Por exemplo, pode-se introduzir suporte a funcionalidades disponíveis somente em uma plataforma de *middleware*. Em relação à CORBA, pode-se introduzir suporte, por exemplo, a chamadas assíncronas de métodos e chamadas `oneway`.

Outro ponto que pode vir a ser pesquisado é a introdução em DAJ de suporte a sistemas de *middleware* que não sejam baseados em invocação de métodos remotos, tais como TSpaces [LMW99], JavaSpaces [FHA99] ou Lime [PMR99].

Bibliografia

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[ASP] *The AspectJ™ Project*. `http://eclipse.org/aspectj/`.

[AXI] *Apache Axis*. `http://ws.apache.org/axis/`.

[BCA+01] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fábio M. Costa, Hector A. Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui S. Moreira, Nikos Parlavantzas, and Katia B. Saikoski. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[BLL04] L. C. Briand, Y. Labiche, and J. Leduc. Towards the Reverse Engineering of UML Sequence Diagrams for Distributed, Multithreaded Java software. Technical report, Carleton University, September 2004.

[CF05] Carine Courbis and Anthony Finkelstein. Towards aspect weaving applications. In *27th International Conference on Software Engineering*, pages 69–77, October 2005.

[CG04] Tal Cohen and Joseph Gil. AspectJ2EE = AOP + J2EE. In *18th European Conference on Object-Oriented Programming*, volume 3086 of *LNCS*, pages 219–243. Springer-Verlag, 2004.

[CI05] Shigeru Chiba and Rei Ishikawa. Aspect-oriented programming beyond dependency injection. In *19th European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 121–143. Springer, 2005.

[CK02] Fábio M. Costa and Fábio Kon. Novas Tecnologias de Middleware: Rumo à Flexibilização. In *Simpósio Brasileiro de Redes de Computadores*, Rio de Janeiro, RJ, Brasil, 2002.

[COR] *CORBA Messaging Specification.* `http://www.omg.org/docs/formal/04-03-09.pdf`.

[CT04] Mariano Ceccato and Paolo Tonella. Adding distribution to existing applications by means of aspect oriented programming. In *4th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 107–116. IEEE Computer Society, 2004.

[Dah01] M. Dahm. Bytecode engeeniering with the BCEL API. Technical report, Freie Universitt - Institut fur Informatik, 2001.

[DW06] Daniel S. Dantas and David Walker. Harmless advice. In *33rd ACM Symposium on Principles of Programming Languages*, pages 383–396. ACM Press, 2006.

[EJB] *Enterprise JavaBeans Technology.* `http://java.sun.com/products/ejb/`.

[Emm00] Wolfgang Emmerich. Software Engineering and Middleware: a roadmap. In *Conference on The Future of Software Engineering*, pages 117–129. ACM Press, 2000.

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley Professional, 1999.

[Fra98] Frantisek Plasil and Michael Stal. An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. *Software Concepts and Tools*, 19(1):14–28, 1998.

[GFS+05] Sudipto Ghosh, Robert B. France, Devon M. Simmonds, Abhijit Bare, Brahmila Kamalakar, Roopashree P. Shankar, Gagan Tandon, Peter Vile, and Shuxin Yin. A Middleware-Transparent Approach to Developing Distributed Applications: Research Articles. *Software-Practice & Experience*, 35(12):1131–1154, 2005.

[GHJV00] Erich Gamma, Richard Helm, Ralph Jahnson, and John Vlissides. *Padrões de Projeto: soluções reutilizáveis de software orientado a objetos.* Bookman, Porto Alegre, 2000.

[Gro02] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 3.0.2*, 2002.

[GSNW02] Aniruddha S. Gokhale, Douglas C. Schmidt, Balachandran Natarajan, and Nanbor Wang. Applying model-integrated computing to component middleware and enterprise applications. *Communications of the ACM*, 45(10):65–70, 2002.

[Jac] JacORB. http://www.jacorb.org.

[Jav] Java IDL. http://java.sun.com/products/jdk/idl.

[JAX] JAX-WS. http://java.sun.com/webservices/.

[JND] *Java Naming and Directory Interface.* `http://java.sun.com/products/jndi/`.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[KWSS03] Dawid Kurzyniec, Tomasz Wrzosek, Vaidy S. Sunderam, and Aleksander Slominski. RMIX: A multiprotocol RMI framework for Java. In *17th International Parallel and Distributed Processing Symposium*, pages 140–146. IEEE Computer Society, April 2003.

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning, 2003.

[LBS97] Geoffrey Lewis, Steven Barber, and Ellen Siegel. *Programming with Java IDL* . John Wiley & Sons, 1997.

[LMW99] Tobin J. Lehman, Stephen W. McLaughry, and Peter Wyckoff. TSpaces: The Next Wave. In *Hawaii International Conference on System Sciences (HICSS-32), January 1999.* Computer Society Press, 1999.

[Lop97] Cristina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, December 1997.

[Mar] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html.

[MOR] *Minimum CORBA Specification*. `http://www.omg.org/technology/documents/formal/minimum_CORBA.htm`.

[MV06a] Cristiano Amaral Maffort and Marco Túlio Oliveira Valente. Aspectos para Construção de Aplicações Distribuídas. In *XX Simpósio Brasileiro de Engenharia de Software*, pages 271–286, 2006.

[MV06b] Cristiano Amaral Maffort and Marco Túlio Oliveira Valente. Aspectos para Construção de Serviços Web. In *III Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos*, pages 11–20, 2006.

[NCT04] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote Pointcut: A Language Construct for Distributed AOP. In *3rd International Conference on Aspect-Oriented Software Development*, pages 7–15. ACM Press, 2004.

[OH01] Piet Obermeyer and Jonathan Hawkins. Microsoft .NET Remoting: a technical overview. Technical report, Microsoft Corporation, 2001.

[OMA] *Object Management Arquitecture*. `http://www.omg.org/oma/`.

[OMGa] *Object Management Group*. `http://www.omg.org/`.

[OMGb] OMG Model Driven Architecture. http://www.omg.org/mda.

[OMG00] OMG. *Discussion of the Object Management Architecture Guide*. Object Management Group, 2000.

[Per04] Fernando Magno Quintão Pereira. Arcademis: Um Arcabouço para Construção de Sistemas de Objetos Distribuídos em Java. Master's thesis, Universidade Federal de Minas Gerais, 2004.

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda Meets Mobility. In *International Conference on Software Engineering*, pages 368–377, 1999.

[POM03] Roman Pichler, Klaus Ostermann, and Mira Mezini. On aspectualizing component models. *Software Practice and Experience*, 33(10):957–974, August 2003.

[PS98] Frantisek Plasil and Michael Stal. An Architectural View of Distributed Objects and Components in CORBA, Java RMI and COM/DCOM. *Software - Concepts and Tools*, 19(1):14–28, 1998.

[RS05] Antônio Maria Pereira Resende and Claudiney Calixto Silva. *Programação Orientada a Aspectos em Java*. Brasport, 2005.

[SBL06] Sergio Soares, Paulo Borba, and Eduardo Laureano. Distribution and persistence as aspects. *Software Practice and Experience*, 36(7):711–759, 2006.

[Sie00] Jon Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, 2nd edition, 2000.

[SLB02] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *17th ACM Conference on Object-Oriented programming systems, languages, and applications*, pages 174–190. ACM Press, 2002.

[SLS03] Macneil Shonle, Karl J. Lieberherr, and Ankit Shah. Xaspects: an extensible system for domain-specific aspect languages. In *OOPSLA Companion*, pages 28–37. ACM Press, October 2003.

[Soa04] Sergio Soares. *An Aspect-Oriented Implementation Method*. PhD thesis, Federal Universisty of Pernambuco, Recife, Brazil, October 2004.

[Ste06] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *21st Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 481–497, 2006.

[TBBV04] Fabio Tirelo, Roberto S. Bigonha, Mariza A. S. Bigonha, and Marco Túlio Oliveira Valente. Desenvolvimento de Software Orientado por Aspectos. In *XXIII Jornada de Atualização em Informática, XXIV Congresso da Sociedade Brasileira da Computação*, 2004.

[TUSF03] Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis, and Marc Fleury. Aspectizing server-side distribution. In *Automated Software Engineering Conference*, pages 130–141. IEEE Press, October 2003.

[Vin97] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[Vin98] Steve Vinoski. New Features for CORBA 3.0. *Commun. ACM*, 41(10):44–52, 1998.

[VTLP05] Marco Tulio Valente, Fabio Tirelo, Diana Campos Leao, and Rodrigo Palhares. An aspect-oriented communication middleware system. In *International Symposium on Distributed Objects and Applications*, volume 3761 of *LNCS*, pages 1115–1132. Springer-Verlag, October 2005.

[VVJ06] B. Verheecke, W. Vanderperren, and V. Jonckers. Unraveling crosscutting concerns in web services middleware. *IEEE Software*, 23(1):42–50, January 2006.

[Wal99] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[Wan03] Mitchell Wand. Understanding aspects: extended abstract. In *8th International Conference on Functional Programming*, pages 299–300. ACM Press, August 2003.

[WRW96] Jim Waldo, Roger Riggs, and Ann Wollrath. A Distributed Object Model for the Java System. *Computing Systems*, 9(4):265–290, 1996.

[ZJ04] Charles Zhang and Hans-Arno Jacobsen. Resolving Feature Convolution in Middleware Systems. In *19th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 188–205. ACM Press, 2004.