Aspectos para Construção de Aplicações Distribuídas

Cristiano Amaral Maffort e Marco Túlio de Oliveira Valente

Instituto de Informática
Pontifícia Universidade Católica de Minas Gerais
{maffort,mtov}@pucminas.br

Abstract. *Software engineers often rely on middleware systems to improve the productivity in the design of distributed systems. However, middleware functionality is usually invasive, pervasive and tangled with business-specific concerns. In this paper, we describe a distributed programming system that encapsulates services provided by two middleware technologies: Java RMI and Java IDL. The proposed system is flexible and has enough expressive power to support common abstractions provided by object oriented middleware platforms. The system is also compatible with different distributed systems architectures and configurations.*

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. Neste artigo, apresenta-se um sistema de apoio à programação distribuída 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.*

1 Introdução

Atualmente, 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, serialização de dados, sincronização, localização de serviços etc. No entanto, os sistemas atuais de *middleware* são, em geral, invasivos, pervasivos e transversais ao código de negócio de aplicações distribuídas [4, 14, 16, 11]. 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 *middleware* não apresentam graus esperados de reusabilidade, separação de interesses e modularidade, o que dificulta o desenvolvimento, manutenção e evolução dos mesmos.

Neste artigo, apresenta-se um sistema de apoio à programação distribuída, chamado DAJ (*Distribution Aspects in Java*). O sistema DAJ permite isolar a funcionalidade de distribuição – provida por plataformas de *middleware* – da lógica de negócio

de aplicações implementadas em Java. Mais especificamente, DAJ inclui um *framework* para implementação de distribuição, o qual define classes e aspectos que encapsulam diversos detalhes de programação usualmente requeridos por plataformas de *middleware*. Além disso, o sistema DAJ inclui uma ferramenta de geração de código, responsável por gerar aspectos que especializam os aspectos abstratos do *framework* de forma a introduzir distribuição em uma determinada aplicação alvo. Ou seja, em DAJ, a adaptação e instanciação do *framework* de distribuição é feita de modo automático. O gerador de aspectos de DAJ gera código para duas plataformas de *middleware* nativas de ambientes de desenvolvimento Java: Java RMI [18] e Java IDL [5]. Java RMI é um *middleware* utilizado com freqüência por aplicações distribuídas cujos módulos são integralmente implementados em Java. A plataforma Java IDL, por sua vez, é uma implementação do padrão CORBA [13] e, portanto, permite que aplicações distribuídas implementadas em Java acessem sistemas desenvolvidos em outras linguagens de programação.

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 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 e classes que modularizam o código de distribuição requerido pela aplicação base. Os aspectos gerados por DAJ são implementados em AspectJ.

Acredita-se que o sistema proposto proporciona pelo menos três benefícios a um engenheiro de sistemas distribuídos. Primeiro, e mais importante, este engenheiro pode se concentrar exclusivamente no desenvolvimento do código funcional de sua aplicação, o qual não será mais entrelaçado com código de distribuição. Segundo, este engenheiro não precisa dominar detalhes e convenções de codificação requeridos por plataformas de *middleware*. Terceiro, este engenheiro não precisa dominar 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.

O restante deste artigo está organizado conforme descrito a seguir. Inicialmente, na Seção 2, apresenta-se uma aplicação distribuída destinada a controlar e monitorar o preço das ações negociadas em uma bolsa de valores. Esta aplicação será usada ao longo de todo o artigo para motivar, apresentar e e validar a arquitetura do sistema DAJ. Em seguida, na Seção 3, descreve-se a interface de programação proposta por DAJ, com ênfase nos parâmetros que podem ser configurados por meio de descritores de distribuição. A Seção 4 apresenta de forma detalhada as classes e aspectos do *framework* de distribuição proposto por DAJ, bem como as classes e aspectos gerados automaticamente pelo sistema de forma a especializar este *framework* e, conseqüentemente, introduzir código de

distribuição em uma determinada aplicação. A Seção 5 avalia o projeto de DAJ. Nesta seção, compara-se também o sistema com uma outra solução para implementação de distribuição em AspectJ, proposta por Soares, Borba e Laureano [14]. A Seção 6 discute outros trabalhos relacionados e a Seção 7 conclui o presente artigo.

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 artigo 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 dois objetivos: (i) comunicar uma alteração no preço de venda de uma determinada ação e (ii) manifestar interesse em receber notificações de alterações no preço de determinada ação. Neste segundo 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 valor como por referência. 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.

Descreve-se a seguir as principais interfaces e classes utilizadas na implementação deste sistema.

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);
}
```

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.

```
class StockInfo {
  String stock;
  double value;
  String 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 a seguir:

```
interface StockListener {
  void update(String stock, double value);
}
```

Implementação: Na implementação do sistema, a classe `StockMarketImpl` fica responsável por implementar a interface `StockMarket`. Por sua vez, a classe `StockListenerImpl` implementa a interface `StockListener` e, portanto, o método `update`. 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.

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.

3.1 Descritores de Distribuição

Um dos objetivos do *framework* 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 *framework* a implementação dos aspectos relacionados a distribuição. No entanto, este projetista precisa explicitar quais objetos da aplicação serão acessados remotamente. Para isso, utiliza-se um arquivo XML, chamado *descritor de distribuição*, por meio do qual são definidos diversos parâmetros de distribuição da aplicação.

Em um descritor de distribuição, são configurados os servidores que compõem a aplicação. Além disso, são definidos os tipos que, em chamadas remotas de métodos, são passados por referência e por valor. No caso de servidores (nodo `server` do arquivo XML), 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 (nodo `remote`), deve-se informar a interface e a classe deste tipo. No caso de tipos passados por valor (nodo `serializable`), deve-se informar a classe deste tipo[1].

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

[1]Por questões de espaço, este artigo não apresenta o esquema do documento XML usado como descritor de distribuição. No entanto, acredita-se que o exemplo a seguir transmite uma idéia bastante completa das informações que devem ser codificadas neste arquivo.

```
 1:  <server id="StockMarketA">
 2:    <interface>stockMarket.StockMarket</interface>
 3:    <class>stockMarket.StockMarketImpl</class>
 4:    <protocol>corba</protocol>
 5:    <nameserver>skank.pucminas.br</nameserver>
 6:  </server>
 7:  <server id="StockMarketB">
 8:    <interface>stockMarket.StockMarket</interface>
 9:    <class>stockMarket.StockMarketImpl</class>
10:    <protocol>javarmi</protocol>
11:    <nameserver>patofu.pucminas.br:1530</nameserver>
12:  </server>
13:  <remote>
14:    <interface>stockMarket.StockListener</interface>
15:    <class>stockMarket.StockListenerImpl</class>
16:  </remote>
17:  <serializable>
18:    <class>stockMarket.StockInfo</class>
19:  </serializable>
```

Por meio deste descritor, configura-se uma implantação do sistema com dois servidores: o primeiro deles utilizará Java IDL para comunicação e será registrado com o nome `StockMarketA` (linhas 1 a 6); o segundo utilizará Java RMI para comunicação e será registrado com o nome `StockMarketB` (linhas 7 a 12). Além disso, define-se que, em chamadas remotas de métodos destes servidores, objetos do tipo `StockListenerImpl` serão passados por referência (linhas 13 a 16) e que objetos do tipo `StockInfo` serão passados por valor (linhas 17 a 19).

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 (quando se passa o *stub* do parâmetro de chamada). Em DAJ, um cliente deve utilizar o método `getReference` do *framework* 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 precisar estar ciente do *middleware* que suporta esta comunicação.

Exemplo: Mostra-se a seguir um fragmento de código de um cliente do Sistema de Controle de Ações.

```
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, "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`, este cliente obtém referências para os servidores de nome `StockMarketA` e `StockMarketB`, definidos no descritor de distribuição mostrado no exemplo da Seção 3.1. Em seguida, o cliente chama o método `update` do primeiro servidor passando um objeto do tipo `StockInfo` por valor (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 sempre por referência, receberá notificações do primeiro servidor via Java IDL e do segundo servidor via Java RMI.

3.3 Codificação de Servidores

Conforme afirmado na Seção 2, as interfaces e classes de negócio de uma aplicação distribuída construída sobre a plataforma DAJ não possuem código de distribuição. 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 3.1. Para instanciar, ativar e registrar os servidores especificados neste descritor, o sistema DAJ gera automaticamente as seguintes classes: `ServerStockMarketA` e `ServerStockMarketB`.

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 *framework* responsáveis por transformar classes de negócio em classes remotas, obter referências para objetos remotos e ativar servidores.

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 se evitar colisões, interfaces remotas são geradas em um pacote separado.

Interfaces Remotas em Java RMI: As 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 [14, 16].

Outra diferença entre interfaces de negócio e interfaces remotas diz respeito a parâmetros passados por referência. Mais especificamente, se um método de uma interface de negócio possui um parâmetro 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[2].

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 2:

```
interface StockMarket extends Remote {
  void update(StockInfo info) throws RemoteException;
  void subscribe(String stock, daj.sca.rmi.StockListener obj)
       throws RemoteException;
  void unsubscribe(String stock, daj.sca.rmi.StockListener obj)
       throws RemoteException;
}
```

A interface remota apresentada estende `Remote`. Além disso, todos os seus métodos ativam `RemoteException`. Por fim, o tipo do parâmetro `obj` dos métodos `subscribe` e `unsubscribe` foi substituído pelo tipo da interface remota associada (a qual, apesar de não mostrada no exemplo, também é gerada automaticamente por DAJ).

Interfaces Remotas em Java IDL: Como usual em aplicações baseadas em CORBA, interfaces remotas devem ser codificadas em IDL (isto é, em uma linguagem neutra proposta por CORBA especificamente para definição de interfaces). Portanto, o sistema DAJ assume que já existe uma especificação em IDL das interfaces de negócio definidas no descritor de distribuição do sistema. Esta suposição é bastante razoável, pois a definição de interfaces IDL é normalmente o ponto de partida de qualquer desenvolvimento baseado em CORBA. 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.

[2]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.

```
interface StockMarketOperations {
 void update(daj.sca.corba.StockInfo info);
 void subscribe(String stock, daj.sca.corba.StockListener obj);
 void unsubscribe(String stock,daj.sca.corba.StockListener obj);
}
```

É interessante notar que, apesar de criada automaticamente pelas ferramentas de geração de código de Java IDL, esta interface também adota a substituição de tipos de negócio por tipos remotos, conforme proposto para interfaces remotas em Java RMI.

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 é 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 em chamadas remotas de métodos. Como exemplo, podemos citar a classe `StockListenerImpl`.

A seguir, descreve-se como estes dois tipos de classes remotas são instrumentadas com código de distribuição requerido por Java IDL. A introdução de código de comunicação requerido por Java RMI segue metodologia semelhante à que será proposta para Java IDL e, portanto, não será discutida no presente artigo.

Classes Remotas usadas para Instanciação de Servidores 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 um parâmetro `T` que seja do tipo de uma interface remota, deve-se introduzir na classe um método correspondente. 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 abaixo o aspecto gerado por DAJ para transformar a classe de negócio `StockMarketImpl` em uma classe remota.

```
 1: aspect RemoteStockMarketImpl {
 2:  declare parents:
 3:     StockMarketImpl implements StockMarketOperations;
 4:
 5:  public void StockMarketImpl.subscribe(String stock,
 6:       daj.sca.corba.StockListener obj) {
 7:    subscribe(stock, new StockListenerAdapter(obj));
 8:  }
 9:  ...
10: }
```

Nas linhas 2 e 3, indica-se que a classe de negócio `StockMarketImpl` deve implementar a interface remota `StockMarketOperations` (mostrada na Seção 4.1).

Como o método `subscribe` desta interface possui um parâmetro formal `obj` cujo tipo é uma interface remota, cuida-se de introduzir uma implementação para este método na classe `StockMarketImpl` (linhas 5 a 8). Esta introdução é necessária, pois 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`).

Na linha 7 do aspecto mostrado anteriormente, o código do método `subscribe` redireciona a chamada para o método de negócio de mesmo nome (o qual foi codificado pelo desenvolvedor da classe `StockMarketImpl`). Para que esse redirecionamento ocorra, no entanto, deve-se instanciar um objeto adaptador de uma classe similar à mostrada abaixo:

```
1: class StockListenerAdapter implements StockListener {
2:   daj.sca.corba.StockListener adaptee;
3:   StockListenerAdapter (daj.sca.corba.StockListener adaptee){
4:     this.adaptee= adaptee;
5:   }
6:   public void update(String stock, double value) {
7:      adaptee.update(stock,value);
8:   }
9: }
```

Este adaptador adapta a interface `StockListener` à sua respectiva interface remota.

Classes Remotas usadas para Instanciação de Objetos Remotos em Java IDL: Neste caso, realiza-se o mesmo processo de introdução de requisitos transversais descrito anteriormente. No entanto, adicionalmente, são introduzidos dois novos membros nas classes de negócio associadas: um método responsável por realizar a ativação de objetos destas classes e uma referência para o *stub* retornado pelo processo de ativação[3]. Basicamente, o método introduzido consulta o valor desta referência. Se ela for nula, realiza-se a ativação do objeto remoto e o *stub* associado ao mesmo é atribuído à referência.

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

```
 1: aspect RemoteStockListenerImpl {
 2:   declare parents:
 3:      StockListenerImpl implements StockListenerOperations;
 4:
 5:   StockListener StockListenerImpl.refIDL= null;
 6:
 7:   StockListener StockListenerImpl.export2IDL() {
 8:     // ativa objeto segundo as convenções de Java IDL
 9:   }
10: }
```

[3]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.

Nas linhas 2 e 3, indica-se que a classe de negócio `StockListenerImpl` deve implementar a interface remota `StockListenerOperations` (gerada automaticamente pelo compilador `idlj`). Na linha 5, introduz-se a referência para o *stub* associado a instâncias desta classe. Nas linhas 7 a 9, introduz-se o método `export2IDL`, o qual é responsável por realizar a ativação de objetos desta classe.

4.3 Obtenção de Referências Remotas

Conforme afirmado na Seção 3.2, clientes conseguem referências para servidores remotos 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 para o objeto retornado pela consulta ao 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 têm duas funções: tratar exceções remotas lançadas pela platatorma de *middleware* em caso de falhas de comunicação 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 4.2.

Mostra-se a seguir o *proxy* gerado quando o cliente solicita uma referência para um servidor da classe `StockMarketImpl` e que utiliza Java IDL para comunicação.

```
 1: class StockMarketImplProxy implements StockMarket {
 2:   daj.sca.corba.StockMarket server;
 3:   .....
 4:   void subscribe(String stock, StockListener obj) {
 5:     try {
 6:       daj.sca.corba.StockListener _obj;
 7:       _obj= ((StockListenerImpl) obj).export2IDL();
 8:       server.subscribe(stock, _obj)
 9:     }
10:     catch(Exception e) { ... }
11:   }....
12: }
```

No método `subscribe` da classe *proxy* mostrada, ativa-se o parâmetro `obj` (linha 7), antes de efetivamente utilizá-lo na chamada remota (linha 8). Esta ativação é requerida por Java IDL, já que este parâmetro é remoto.

4.4 Ativação de Objetos Servidores

Conforme afirmado na Seção 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: abstract aspect RMIServer {
 2:   abstract pointcut ServerMainExecution();
 3:   abstract String getServerName();
 4:   abstract String getRegistry();
 5:   abstract Remote getInstance();
 6:
 7:   void around(): ServerMainExecution() {
 8:       // instancia, ativa e registra objetos remotos em RMI
 9:   }
10: }
```

Neste aspecto, o *pointcut* abstrato `ServerMainExecution` denota os pontos do programa onde deve-se instanciar um objeto servidor (linha 2). 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 3). O *host* do servidor de nomes é retornado pelo método `getRegistry` (linha 4). Já o método `getInstance` é usado para se obter uma instância do servidor (linha 5). Por fim, um *advice* associado ao *pointcut* `ServerMainExecution` se encarrega de instanciar, registrar e ativar um objeto remoto segundo as convenções de Java RMI (linhas 7 a 9).

Em DAJ, existe ainda um aspecto abstrato chamado `CORBAServer`, o qual é semelhante ao aspecto `RMIServer`. No entanto, neste aspecto, o *advice* associado ao *pointcut* `ServerMainExecution` instancia e registra um objeto remoto segundo a interface de programação de Java IDL.

Exemplo: Mostra-se a seguir o aspecto concreto `RMIServerStocketMarketB` gerado automaticamente pelo sistema DAJ para instanciar e registrar o servidor `StockMarketB` (definido no descritor de distribuição mostrado na Seção 3.1).

```
 1: aspect RMIServerStockMarketB extends RMIServer {
 2:  pointcut ServerMainExecution:
 3:   execution(public static void ServerStockMarketB.main(..));
 4:  String getServerName() {
 5:    return "StockMarketB";
 6:  }
 7:  String getRegistry() {
 8:    return "patofu.pucminas.br:1530";
 9:  }
10:  StockMarket getInstance() {
11:    return new StockMarketImpl();
12:  }
13: }
```

Como pode ser observado, os parâmetros que diferenciam este código (tais como, nome do servidor, endereço do *registry* e nome da classe do servidor) são todos obtidos do descritor de distribuição.

5 Avaliação

O desenvolvimento de DAJ foi inspirado no trabalho de Soares, Borba e Laureano, visando a implementação de aspectos de distribuição em AspectJ. Em [14], estes autores

descrevem uma experiência bem sucedida de reestruturação de uma aplicação real, chamada *HealthWatcher*, usada por cidadãos para enviar reclamações sobre restaurantes e similares a órgãos públicos responsáveis pela vigilância sanitária destes estabelecimentos. 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. Na versão original do sistema, este código encontrava-se espalhado e entrelaçado nas classes de negócio da aplicação. Além de distribuição, os autores apresentam também uma solução para modularização de persistência.

Assim, inicia-se esta seção de avaliação com uma descrição das principais contribuições que DAJ introduz no método de implementação proposto em *HealthWatcher*:

- DAJ oferece uma solução para modularização de código de distribuição requerido tanto por Java RMI como por Java IDL, enquanto que em *HealthWatcher* propõe-se uma solução apenas para Java RMI. 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 simétrica e ortogonal para implementação modularizada de distribuição.

- DAJ permite passagem de objetos por referência 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[4]. Já o método de implementação proposto em *HealthWatcher* inclui suporte apenas a passagem de objetos por valor.

- DAJ permite a geração de aspectos de distribuição independentemente da arquitetura de *software* adotada pela aplicação base. Por outro lado, a solução proposta em *HealthWatcher* é restrita e limitada pela arquitetura deste sistema. Por exemplo, assume-se em *HealthWatcher* 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 *HealthWatcher* 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 [15], Soares descreve uma ferramenta que automaticamente gera aspectos que concretizam os aspectos abstratos propostos em *HealthWatcher*. No entanto, a ferramenta

[4] A fim de avaliar a freqüência com que aplicações distribuídas utilizam passagem por referência, foi realizada uma pesquisa com os oito projetos de maior atividade que utilizam Java RMI hospedados no repositório de código livre `sourceforge.net`. Esta pesquisa mostrou que quatro destes aplicativos fazem uso de passagem por referência: BlackJackRMI, ChatX, Java3D Distributed Environment e RMI-MessageQueue. Não fazem uso de passagem por referência os seguintes projetos: Load4J, m-e-c Scheduler, RMIJavaChat e Virtual Sharing. Esta pesquisa acessou a versão mais recente destes projetos em 17/04/2006.

proposta é específica para sistemas que seguem a mesma arquitetura de *software* adotada em *HealthWatcher*.

A seguir, são discutidas e avaliadas outras características consideradas como contribuições importantes em DAJ:

- Suporte simultâneo a duas plataformas de *middleware*: DAJ permite que uma determinada classe servidora seja instrumentada tanto com código de comunicação Java RMI quanto com código requerido por Java IDL. Assim, uma mesma instância desta classe pode responder a chamadas remotas provenientes tanto de sistemas RMI como de sistemas CORBA. Um exemplo desta situação foi mostrado no exemplo da Seção 3.2, onde um mesmo objeto do tipo `StockListener` é usado para receber *callbacks* provenientes de um servidor Java RMI e de um servidor Java IDL. Este suporte simultâneo a múltiplos protocolos de comunicação foi alcançado graças ao projeto dos aspectos abstratos do sistema, no qual se conseguiu evitar interferências entre aspectos responsáveis por interesses transversais demandados por plataformas de *middleware* distintas.

- Reconfigurações em clientes e servidores: DAJ permite que parâmetros de distribuição usados por clientes e servidores sejam reconfigurados sem requerer manutenções ou recompilações dos módulos envolvidos. As reconfigurações possíveis incluem alterações no endereço do servidor de nomes e no protocolo de comunicação utilizado por um servidor. Estas reconfigurações afetam apenas o descritor de distribuição da aplicação.

- *Obliviousness* e Transparência: Plataformas de *middleware* têm como objetivo tornar transparentes detalhes de programação em redes. Já o sistema DAJ tem como objetivo modularizar o código invasivo e intrusivo, requerido pelas próprias plataformas de *middleware*. Portanto, 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? Felizmente, o grau de dependência e entrelaçamento do sistema é nulo, no caso de módulos servidores. No caso de módulos clientes este grau é mínimo, resumindo-se ao uso do método `getReference`.

No entanto, a dependência introduzida por DAJ em clientes, mencionada no último item, pode ser eliminada criando-se um aspecto adicional, conforme ilustra o seguinte exemplo:

```
 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= ServiceLocator.getReference("StockMarketA");
 9:     c.s2= ServiceLocator.getReference("StockMarketB");
10:   }
11: }
```

O aspecto proposto, a exemplo do que ocorre em *frameworks* de injeção de dependência [8], é responsável por obter as referências remotas associadas às variáveis de instância `s1` e `s2`. Com isso, a classe cliente `MyClient` não possui mais qualquer referência a componentes do sistema DAJ.

6 Trabalhos Relacionados

Diversos projetos de pesquisa têm investigado o uso de aspectos para modularizar serviços transversais providos por plataformas de *middleware*. Por exemplo, já foram desenvolvidas ferramentas baseadas em aspectos para auxiliar na implementação de aplicações EJB, tais como Gotech [16] e AspectJ2EE [3]. 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.

Outros trabalhos têm como objetivo incorporar abstrações para programação distribuída em linguagens orientadas por aspectos. DJcutter [9] 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 [2] é uma extensão de AspectJ com suporte a injeção de dependência.

Ghosh e colegas descrevem uma metodologia para implementar de forma independente interesses de negócio e interesses de distribuição providos por plataformas de *middleware* [4]. Assim, considera-se que esta metodologia e o sistema DAJ compartilham dos mesmos objetivos. A idéia dos autores é incorporar conceitos e técnicas de desenvolvimento orientado por modelos e, mais especificamente, conceitos propostos pela arquitetura MDA [10], 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 plataformas de *middleware*, a fim de se obter um MSD (*middleware specific design*), isto é, um modelo incluindo requisitos funcionais e de distribução. Consideramos que DAJ é um sistema que põe em prática os princípios e conceitos da proposta de Ghosh e colegas, disponibilizando uma ferramenta que automatiza e torna transparente detalhes de programação distribuída requeridos por Java IDL e Java RMI.

Os descritores de distribuição propostos em DAJ podem ser considerados como uma linguagem orientada por aspectos de domínio específico para representação de requisitos transversais introduzidos por tecnologias de *middleware*. Alguns autores, como Wand [17] e Lieberherr [12], 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 sujeitas às críticas normalmente feitas a linguagens orientadas por aspectos de propósito geral, como violação de encapsulamento e impossibilidade de se raciocinar localmente sobre módulos [17]. 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 [7].

Ceccato e Tonella descrevem um *framework* que inclui geração automática de

aspectos de distribuição e interfaces remotas [1]. No entanto, a solução proposta é restrita a Java RMI. Além disso, a especificação dos parâmetros de distribuição é feita por meio de uma simples lista com o nome das classes remotas do sistema.

RMIX [6] é um sistema que permite a instanciação de servidores que aceitam chamadas remotas provenientes de Java RMI, CORBA ou SOAP. Para prover esta flexibilidade, RMIX reimplementa diversos componentes internos comuns a sistemas de *middleware*. Em DAJ, a mesma flexibilidade é obtida de forma mais simples e não-invasiva.

7 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. 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.

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 se beneficia da sinergia gerada pela combinação de três tecnologias utilizadas na sua implementação: 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).

As principais contribuições e funcionalidades da ferramenta DAJ foram validadas por meio de uma aplicação distribuída simples, mas que faz uso de diversas abstrações normalmente providas por plataformas de *middleware*. Com o uso de DAJ, foi possível encapsular todo código de distribuição deste sistema. Com isso, as classes de negócio da aplicação 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 deste sistema.

Como trabalho futuro, pretende-se investigar o emprego de DAJ no desenvolvimento e/ou reestruturação de outras aplicações distribuídas. Pretende-se ainda adicionar suporte a aplicações baseadas em serviços *Web*.

Agradecimentos: Este trabalho foi desenvolvido como parte de um projeto de pesquisa financiado pela FAPEMIG (processo CEX-817/05 - Edital Universal 2005).

Referências

[1] 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.

[2] 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.

[3] 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.

[4] S. Ghosh, R. B. France, A. Bare, B. Kamalalar, R. P. Shankar, D. M. Simmonds, G. Tandon, P. Vile, and S. Yin. A middleware transparent approach to developing distributed applications. *Software Practice and Experience*, 35(12):1131–1154, October 2005.

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

[6] 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.

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

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

[9] 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.

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

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

[12] 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.

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

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

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

[16] 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.

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

[18] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232, 1996.