Aspectos para Construção de Serviços Web

Cristiano Amaral Maffort e Marco Túlio de Oliveira Valente

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

Abstract. *Middleware for web services implementation usually require pervasive and crosscutting code. DAJ is a distributed programming system that decouples functional requirements from middleware specific services. For this purpose, DAJ relies on the combination of three programming technologies: aspects, domain-specific languages and generative programming. In this paper, we present an extension of DAJ with aspects specific to web services implementation.*

Resumo. *Sistemas de middleware para desenvolvimento de serviços Web são invasivos, pervasivos e transversais ao código de negócio de aplicações distribuídas. DAJ é um sistema de programação distribuída que procura resolver esta deficiência das atuais plataformas de middleware. Para isso, DAJ automatiza a modularização em aspectos dos serviços de distribuição providos por plataformas de middleware comuns em ambientes de desenvolvimento em Java. Neste artigo, discute-se o uso de aspectos para introduzir em DAJ suporte a sistemas de middleware para desenvolvimento de serviços Web.*

1. Introdução

Serviços *Web* são aplicações distribuídas que utilizam linguagens e protocolos padrões da Internet [1]. Mais precisamente, tais aplicações interagem usando mensagens formatadas e encapsuladas segundo o protocolo SOAP e transportadas via HTTP. Além disso, serviços *Web* têm suas interfaces definidas em uma linguagem denominada WSDL e são publicados e descobertos usando-se um protocolo chamado UDDI. Atualmente, este conjunto de tecnologias tem sido alvo de grande interesse, tanto da parte de pesquisadores da área de sistemas distribuídos, como da comunidade de desenvolvedores de *software* para a *Web*. Por exemplo, importantes nomes da Internet, como Google e Amazon, já podem ser acessados por meio de serviços *Web*.

Com o sucesso da tecnologia, diversas plataformas de *middleware* foram desenvolvidas para apoiar a implementação de sistemas distribuídos baseados em serviços *Web*. Dentre elas, destacam-se as plataformas Apache Axis [2], JAX-RPC [8] e .NET [11]. No entanto, tais sistemas sofrem de um problema recorrente em sistemas de *middleware* em geral: eles requerem a introdução de código invasivo, pervasivo e transversal ao código de negócio de aplicações distribuídas [5, 14, 12]. Em outras palavras, desenvolvedores de serviços *Web*, ao utilizar qualquer dos sistemas de *middleware* mencionados, devem seguir uma série de convenções de programação. Por exemplo, usualmente cabe a tais desenvolvedores estender classes internas da plataforma, tratar exceções geradas pelo sistema, implementar interfaces remotas, usar determinadas anotações, invocar geradores de *stubs* etc. Como resultado, serviços *Web* construídos usando estas plataformas de

middleware não apresentam graus esperados de reusabilidade, separação de interesses e modularidade.

DAJ (*Distribution Aspects in Java*) é uma ferramenta de apoio à programação distribuída que disponibiliza recursos para encapsular código de distribuição requerido por plataformas de *middleware* utilizadas no desenvolvimento de aplicações distribuídas em Java [10]. Para alcançar este objetivo, DAJ 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).

A Figura 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 qualquer convenção de programação requerida por uma determinada plataforma de *middleware*. Esta característica do desenvolvimento de aplicações 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. Além disso, DAJ evita que este desenvolvedor tenha que conhecer em detalhes convenções de codificação requeridas por plataformas de *middleware*. 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. Os aspectos gerados por DAJ são implementados em AspectJ. 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.

A versão atual de DAJ fornece a opção de gerar código para duas plataformas de *middleware* nativas de ambientes de desenvolvimento Java: Java RMI [19] e Java IDL [7]. Neste artigo, estende-se o artigo original de definição de DAJ [10], descrevendo a incorporação no sistema de uma terceira plataforma de *middleware*: Apache Axis [2], a qual é largamente utilizada para desenvolvimento de aplicações distribuídas baseadas na tecnologia de serviços *Web*. O objetivo final é tornar DAJ compatível com as três principais tecnologias de chamada remota de métodos disponíveis em ambientes Java.

O restante do artigo está estruturado da seguinte forma. A Seção 2 descreve a interface de programação de DAJ. A Seção 3 descreve os aspectos gerados por DAJ para encapsular interesses de distribuição requeridos para comunicação com serviços *Web*. A Seção 4 discute trabalhos relacionados e a Seção 5 conclui apresentando uma discussão sobre as vantagens e desvantagens do emprego de aspectos em DAJ.

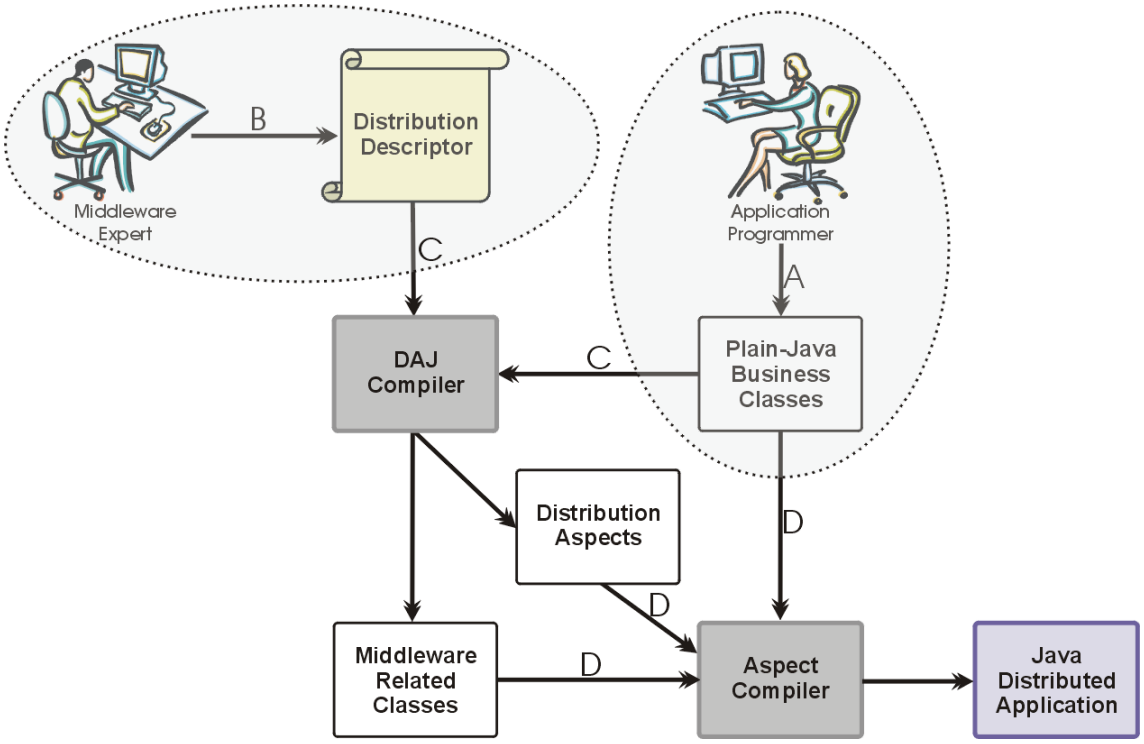

Figura 1. Processo de desenvolvimento de aplicações distribuídas usando DAJ

2. DAJ: Interface de Programação

A fim de descrever a interface de programação de DAJ, utilizaremos como exemplo um sistema para monitoramento dos preços das ações negociadas em uma bolsa de valores. Neste sistema, existe um servidor responsável por manter o preço atual das ações negociadas na bolsa. Clientes podem acessar este servidor tanto para atualizar o valor de uma ação, quanto para obter o valor corrente de ações nas quais esteja interessado. Este sistema é um candidato natural a ser implantado usando-se serviços *Web*, pois com isso permite-se que ele seja acessado não apenas via *browsers*, mas que possa também ser integrado aos sistemas de informação dos clientes desta bolsa de valores.

Mostra-se a seguir a principal interface do Sistema de Controle de Ações. Esta interface inclui métodos para atualizar o preço de uma determinada ação (`update`), assim como para retornar informações sobre o preço de venda de um conjunto de ações (`getStocks`) ou sobre o preço de venda de todas as ações negociadas na bolsa (`getAllStocks`). A classe `StockInfo` é usada para armazenar informações sobre ações.

```
interface StockMarket {
  void update(StockInfo info);
  StockInfo[] getStocks(String[] stockName);
  StockInfo[] getAllStocks();
}

class StockInfo {
  private String stock;
  private double value;
  private String timestamp;
```

```
  public StockInfo (String stock, double value, String ts) {
    this.stock= stock; this.value= value; this.timestamp= ts;
  }
  public String getStock() { return stock; }
  public void setStock(String stock) { this.stock = stock; }
  public double getValue() { return value; }
  public void setValue(double value) { this.value = value; }
  public String getTimestamp() { return timestamp; }
}
```

O campo `timestamp` da classe `StockInfo` possui apenas método `get`. O objetivo é evitar alterações no `timestamp` de objetos que contenham informações sobre preços de ações, após a criação dos mesmos.

Descritor de Distribuição: Suponha que a classe `StockMarketImpl` implementa a interface `StockMarket`. Mostra-se a seguir um possível descritor de distribuição para o Sistema de Controle de Ações. Por meio deste descritor, define-se a interface de um objeto remoto (nodo `interface`), a classe que implementa esta interface (nodo `class`), o *middleware* de comunição a ser usado no sistema (nodo `protocol`) e a URI na qual o serviço deverá ser disponibilizado (nodo `uri`). Define-se ainda que a classe `StockInfo` deve ser serializável (nodo `serializable`), já que seus objetos são usados como parâmetro e como retorno de chamadas remotas.

```
<server id="StockMarket">
   <interface>stockService.StockMarket</interface>
   <class>stockService.StockMarketImpl</class>
   <protocol>ApacheAxis</protocol>
   <uri>http://patofu.pucmg.br/axis/services/StockMarket</uri>
</server>
<serializable>
   <class>stockService.StockInfo</class>
</serializable>
```

Codificação de Clientes: Em DAJ, clientes devem utilizar o método `getReference` 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 deste servidor, sem precisar estar ciente do *middleware* que suporta esta comunicação.

Mostra-se a seguir um fragmento de código de um cliente do Sistema de Controle de Ações. Inicialmente, o cliente obtém, através do método `getReference`, uma referência para o serviço *Web* identificado pelo nome `StockMarket` (linha 2). Em seguida, o cliente chama o método `update` passando um objeto do tipo `StockInfo` como parâmetro (linha 4). Por fim, o cliente solicita, por meio do método `getStocks`, o valor corrente das ações das companhias `bb` e `cef` (linha 6).

```
1: StockMarket sm;
2: sm= (StockMarket) ServiceLocator.getReference("StockMarket");
3:   ...
4: sm.update(new StockInfo("cvrd", 124.60, DateTime.now()));
5:   ...
6: StockInfo[] stocks = sm.getStock(new String[]{"bb", "cef"});
```

Ativação de Servidores: Em uma aplicação distribuída construída com suporte de DAJ, desenvolvedores não precisam se preocupar com convenções e procedimentos necessários para ativar e registrar objetos que representam serviços *Web*. O sistema DAJ gera automaticamente todas as classes e arquivos necessários para ativar e registrar tais serviços.

3. Aspectos para Suporte a Serviços *Web*

Nesta seção, descrevem-se as classes e aspectos criados por DAJ, a partir de descritores de distribuição, com o objetivo de transformar uma aplicação em um cliente de serviços *Web*. Suponha a chamada ao método `getReference` do exemplo de cliente de serviço *Web* mostrado na Seção 2. Esta chamada retorna um objeto *proxy* da seguinte classe gerada por DAJ:

```
1: class StockMarketProxy implements stockService.StockMarket {
2:   daj.ws.api.StockMarket stub;
3:   public void update(stockService.StockInfo arg1) {
4:     try {
5:       stub.update((daj.ws.api.StockInfo) arg1);
6:     } catch (Exception e) { ..... }
7:   } ...
8: }
```

Esta classe implementa a interface de negócio da aplicação (`StockMarket`) e possui uma referência para o serviço *Web* remoto (linha 2). O tipo desta referência é uma interface gerada pela ferramenta `wsdl2java`, a qual é utilizada em Apache Axis para gerar *stubs* a partir de interfaces WSDL. Em DAJ, proxies são usados para converter chamadas remotas, realizadas na aplicação usando-se referências de tipos de negócio (como é o caso de `stockService.StockMarket`), em chamadas realizadas usando-se referências cujos tipos são interfaces geradas pela plataforma de *middleware* subjacente (como é o caso de `daj.ws.api.StockMarket`). Estas últimas referências denotam *stubs*, isto é, objetos que encapsulam todos os detalhes de comunicação com o serviço remoto (incluindo criação de envelopes de comunicação SOAP, abertura de conexões HTTP etc).

Conforme mostrado na Seção 2, o método `update` da interface `StockMarket` possui a seguinte assinatura:

```
void update(stockService.StockInfo info)
```

Já o método `update` da interface implementada pelos *stubs* de Apache Axis possui a seguinte assinatura:

```
void update(daj.ws.api.StockInfo info)
```

Como o tipo do parâmetro formal `info` é diferente nestas duas assinaturas, justifica-se o *casting* realizado sobre este parâmetro quando do redirecionamento da chamada de `update` para o respectivo método do objeto *stub* (linha 5 da classe `StockMarket-Proxy`). Além disso, aspectos são usados para evitar que este *casting* venha a falhar em tempo de execução. Essencialmente, estes aspectos são usados para injetar código de serialização requerido por Apaches Axis em classes de negócio da aplicação alvo do sistema DAJ, conforme descrito a seguir.

Aspectos para Introdução de Código de Serialização em Classes de Negócio: As classes serializáveis geradas por Apache Axis (como a classe `daj.ws.api.StockInfo`) incluem apenas atributos públicos e atributos não públicos para os quais existem métodos `get` e `set` nas respectivas classes de negócio (como a classe `stockService.StockInfo`). Assim, na aplicação de exemplo descrita na Seção 2, a classe serializável gerada por Apache Axis possuirá apenas os atributos `stock` e `value`. O atributo de nome `timestamp` não possuirá equivalente na classe gerada por Apache Axis, já que o mesmo não possui método `set`.

Em aplicações baseadas em Apache Axis, a exigência de métodos `get`/`set` para atributos não públicos se deve ao fato desta plataforma de *middleware* reutilizar classes de serialização de EJB [15]. No entanto, como DAJ tem como objetivo central tornar o código de negócio de uma aplicação distribuída independente de convenções de programação requeridas por uma plataforma de *middleware* específica, optou-se por não adotar no sistema as classes de serialização geradas por Apache Axis. Em vez disso, DAJ usa aspectos para introduzir código de serialização nas próprias classes de negócio da aplicação, conforme descrito abaixo:

1. Inicialmente, a classe serializável gerada por Apache Axis (no exemplo, chamada de `daj.ws.api.StockInfo`) é substituída por uma interface de mesmo nome e vazia. Em seguida, usando-se os recursos de transversalidade estática de AspectJ, força-se a classe de negócio da aplicação a implementar esta interface. Com isso, assegura-se que o *casting* realizado na classe `StockMarketProxy` não irá falhar em tempo de execução. Mostra-se a seguir a introdução realizada:

```
declare parents:
  stockService.StockInfo implements daj.ws.api.StockInfo;
```

2. Conforme requerido pelos *stubs* de Apache Axis, introduzem-se métodos `get` e `set` para todos os atributos não públicos de uma classe que serão objeto de serialização. No caso da classe `StockInfo`, como os atributos `stock` e `value` já possuem métodos `get` e `set`, basta introduzir o método `set` para o atributo `timestamp`, conforme mostrado abaixo:

```
public void stockService.Stock.setTimestamp(String timestamp){
   this.timestamp= timestamp;
}
```

3. Conforme requerido por Apache Axis em classes serializáveis, introduz-se um atributo de nome `typeDesc` na classe de negócio da aplicação alvo. Este descritor irá conter informações sobre os atributos serializáveis desta classe (nome, tipo etc). Introduzem-se também métodos `get` e `set` para este atributo, conforme mostrado a seguir:

```
private static TypeDesc stockService.StockInfo.typeDesc =
    new TypeDesc(stockService.StockInfo.class, true);
TypeDesc stockService.StockInfo.getTypeDesc() {
   return typeDesc;
}
void stockService.StockInfo.setTypeDesc(TypeDesc typeDesc) {
   this.typeDesc= typeDesc;
}
```

4. Por meio dos recursos de reflexão computacional de Java, obtém-se informações sobre todos os atributos da classe de negócio. Estas informações são então usadas para inicializar o descritor de serialização da classe, conforme mostrado a seguir:

```
static {
  QName qname= new QName("urn:BeanService", "StockInfo");
  stockService.StockInfo.typeDesc.setXmlType(qname);
  org.apache.axis.description.ElementDesc eD;
  eD = new org.apache.axis.description.ElementDesc();
  eD.setFieldName("stock");
  eD.setXmlName(new javax.xml.namespace.QName("", "stock"));
  qname= new QName("http://www.w3.org/2001/XMLSchema");
  eD.setXmlType(qname,"string"));
  eD.setNillable(true);
  stockService.StockInfo.typeDesc.addFieldDesc(eD);
  ...
}
```

5. Em alguns pontos de seu código, a classe *stub* gerada por Apache Axis usa o nome da classe de serialização originalmente gerada pela plataforma (no exemplo, chamada de `daj.ws.api.StockInfo`). Como esta classe foi substituída pela própria classe de negócio da aplicação, a mesma substituição deve ser feita no código da classe que representa o *stub*. Inicialmente, um *pointcut* captura os pontos de execução do *stub* onde potencialmente se pode referenciar a classe serializável original de Apache Axis. Estes pontos correspondem a chamadas do método `add` de um `Vector` realizadas no interior da construtora da classe *stub*. Um *advice* associado a este *pointcut* verifica se o parâmetro do método `add` referencia o objeto do tipo `Class` da classe serializável original. Se referenciar, troca-se este parâmetro pelo `Class` da classe de negócio da aplicação alvo do sistema DAJ, conforme mostrado a seguir.

```
pointcut refOldSerializedClass(Class param):
  call(boolean java.util.Vector.add(..)) &&
  withincode(daj.ws.api.StockMarketSoapBindingStub.new(..)) &&
  args(param);

boolean around(Class param): refOldSerializedClass(param) {
   if(param == daj.ws.api.StockInfo.class)
     return proceed(stockService.StockInfo.class);
   ...
   return proceed(param);
}
```

4. Trabalhos Relacionados

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.

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. AspectJRMI [17] 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.

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* [4]. Já WSML [18] é um *middleware* que utiliza aspectos para encapsular diversos requisitos transversais típicos de aplicações clientes de serviços *Web*, incluindo autenticação, *logging*, monitoramento, transações etc.

5. Conclusões

Em DAJ, aspectos são usados para modularizar código transversal de comunicação requerido por três plataformas de *middleware*: Java RMI, Java IDL e Serviços *Web*. O suporte às duas primeiras plataformas em DAJ foi descrito com detalhes em [10]. No presente artigo, descreveu-se o uso de aspectos para tornar DAJ compatível com a tecnologia de serviços *Web*, utilizando-se para isso o sistema Apache Axis. Em DAJ, aspectos são usados para tornar o código de negócio de um sistema distribuído livre de quaisquer interesses de distribuição. Com isso, facilita-se o desenvolvimento, o entendimento e o reúso do código de negócio deste sistema. Viabiliza-se também o desenvolvimento de sistemas distribuídos portáveis entre as três tecnologias de *middleware* suportadas por DAJ.

Em vez de aspectos, 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 [9], 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*,

possuíram métodos `get` e `set`, possuíram 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 em ferramentas CASE, as quais normalmente incluem funcionalidades para gerar esqueletos de classes a partir de modelos de mais alto nível, como diagramas UML. No entanto, a mesma possui duas importantes deficiências: (i) o código requerido pelo *middleware*, apesar de não 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 das ferramentas CASE orientadas por objeto com recursos para geração automática de código de distribuição [6].

Em relação à primeira versão de DAJ, com suporte apenas a Java RMI e Java IDL, as principais dificuldades encontradas para introduzir no sistema suporte a serviços *Web* foram conseqüência direta da impossibilidade de se reutilizar as classes de serialização geradas por Apache Axis, conforme descrito na Seção 3. Com isso, foi necessário usar extensivamente os recursos de declaração inter-tipos de AspectJ para migrar para as classes serializáveis da aplicação código originalmente implementado nas classes geradas por Apache Axis. Esta tarefa exigiu considerável esforço, por demandar um perfeito entendimento das rotinas de serialização de Apache Axis. Além disso, a mesma implica em um maior acoplamento entre os aspectos de DAJ e a implementação de Apache Axis. Assim, evoluções futuras no esquema de serialização de classes adotado em Apache Axis podem vir a impactar os aspectos descritos neste artigo.

No caso de se optar por servicos *Web*, a versão implementada não inclui suporte a passagem de objetos com semântica de referência, normalmente utilizada em sistemas distribuídos para modelar *callbacks* realizados por objetos servidores em objetos clientes. Esta deficiência, no entanto, é comum à maioria dos sistemas de *middleware* para desenvolvimento de serviços *Web*, já que o suporte a *callbacks* requer a existência de um servidor *Web* executando em estações clientes. Em [13], Ruth e colegas propõem exatamente uma solução para *callbacks* em aplicações distribuídas baseadas na tecnologia de serviços *Web* que faz uso de tais servidores. Como trabalho futuro, pretende-se incorporar em DAJ soluções semelhantes a esta, bem como investigar o emprego de DAJ no desenvolvimento e/ou reestruturação de aplicações distribuídas maiores que o Sistema de Controle de Ações utilizado como exemplo neste artigo.

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

Referências

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

[2] Apache Axis. http://ws.apache.org/axis/.

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

[5] 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.

[6] 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.

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

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

[9] Byte Code Engineering Library. http://jakarta.apache.org/bcel/.

[10] Cristiano Amaral Maffort and Marco Túlio Valente. Aspectos para construção de aplicações distribuídas. In *XX Simpósio Brasileiro de Engenharia de Software*, October 2006.

[11] Microsoft .NET Web Services Technology. http://msdn.microsoft.com/webservices/.

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

[13] Michael Ruth, Feng Lin, and Shengru Tu. A framework for applications utilizing web services with callbacks. In *International Conference on Web Services*, pages 829–830, 2005.

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

[15] Sun Microsystem. Enterprise Java Beans specification (version 2.1), November 2003.

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

[17] 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.

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

[19] 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.