Estudo de Caso Sobre Programação Orientada a Aspectos Utilizando Protocolo RMI

Felipe da Silva Borges
André Luis Castro de Freitas[1]

[1]Centro de Ciências Computacionais – Universidade Federal do Rio Grande (FURG)
Caixa Postal 474 – 96201.900 – Rio Grande – RS – Brasil
dmtalcf@furg.br

Abstract. *This paper presents a case study concerning Aspect-Oriented Programming and a comparison with Object-Oriented Programming. the program used for the study combines remote methods invocation, using RMI, a distribution protocol used in programs that involve remote methods or remote objects. In this paper, the main features that guide Aspect-Oriented Programming will be adressed, its advantages and disadvantages regarding Object-Oriented Programming and the results with the implementation of proper testing by the two forms of programming proposals in this case study.*

Resumo. *Este trabalho apresenta um estudo de caso sobre Programação Orientada a Aspectos e uma comparação com a Programação Orientada a Objetos. O aplicativo utilizado para o estudo combina invocações remotas de métodos, utilizando o RMI (Remote Method Invocation) um protocolo de distribuição muito utilizado em programas que envolvam métodos ou objetos remotos. São abordadas as características básicas que norteiam a Programação Orientada a Aspectos, suas vantagens e desvantagens em relação à Orientação a Objetos e os resultados obtidos com a realização dos devidos testes pelas duas formas de programação propostas nesse estudo de caso.*

Palavras-chave: *cliente, servidor, RMI, Programação Orientada a Aspectos, Programação Orientada a Objetos.*

1. Introdução

A Programação Orientada a Aspectos, por ser um objeto de estudo recente, está sendo muito abordada por engenheiros de software, pesquisadores e programadores interessados em utilizar essa técnica que busca reduzir a complexidade e aumentar a produtividade no desenvolvimento de software. A Orientação a Objetos é um método eficaz até certo ponto. Precisava de uma evolução que não foi obtida, o encapsulamento dos requisitos que produzem efeitos em todo o sistema, pois não passou por revisões e adaptações profundas nas suas descrições. Por isso, a Orientação a Aspectos (OA) ganhou força na comunidade científica e tecnológica como a provável solução para a lacuna deixada pela Orientação a Objetos, com certa eficácia na mudança de pensamento e do conhecimento na área de desenvolvimento de sistemas. Muitas empresas têm incorporado rapidamente a OA por não exigir nenhuma adaptação de seus equipamentos e ser de fácil compreensão aos desenvolvedores e testadores de software.

Por outro lado, o protocolo de distribuição RMI da linguagem Java tem uma boa aceitação e é um dos métodos remotos mais utilizados no mundo, por ser nativo de Java e ser de fácil implementação e configuração.

Este trabalho tem por objetivo comparar os métodos de orientação para o desenvolvimento de programas propiciando facilidade de entendimento tanto pelo desenvolvedor e pelo testador desse software. Visa também apresentar conceitos e características da Programação Orientada a Aspectos e, juntamente com o protocolo de distribuição RMI, demonstrar uma aplicação efetiva de uso, com um carrinho de compras de um mercado fictício com um objeto cliente e um objeto servidor remoto.

Este trabalho está organizado nas seções 2, 3, 4 e 5: A seção 2 descreve os conceitos da programação orientada a objetos. Na seção 3 aborda-se a teoria da programação orientada a aspectos. A seção 4 faz um estudo sobre o protocolo de distribuição RMI e, finalmente, a seção 5 demonstra o aplicativo estudado nas duas formas de programação com o uso do protocolo RMI. Após procede-se a conclusão onde será feita uma

avaliação das metodologias e resultados obtidos nos testes. Finalmente, são apresentados os trabalhos futuros.

2. Programação Orientada a Objetos

A Programação Orientada a Objetos (POO) é o padrão mais utilizado na comunidade científica, acadêmica e comercial. A base da POO é o objeto, que é uma estrutura organizada que limita o uso, a inserção e a alteração das informações contidas nesse objeto devendo seguir as regras da classe em que o objeto está incluso. Para ROYCE [1998], a tecnologia orientada a objetos permite a construção de modelos de domínios complexos, consistindo em um grande número de ocorrências, por meio de entidades computacionais denominadas objetos.

Pode-se afirmar que a POO foi o primeiro método que permitiu a herança de atributos e objetos de classes ditas superclasses para outras classes criadas que necessitam dessas operações, chamadas subclasses, o que ajuda a reutilização de código. Além da herança, a POO permitiu o uso do polimorfismo para permitir que funcionalidades sejam dinamicamente selecionadas durante a execução dos programas.

2.1 Classes & Objetos

Para MELLO [2002], classes são definidas como uma estrutura de dados que contém métodos e atributos. No paradigma da orientação a objetos, os procedimentos ou funções (presentes em linguagens estruturadas, tais como C e Java) são chamados de métodos de uma classe. As variáveis, que são declaradas dentro de uma classe, são chamadas de atributos.

Segundo SANTOS [2003], se a classe é usada para que várias instâncias sejam criadas a partir dela, cada uma destas instâncias terá um conjunto dos campos definidos na classe. Após a definição das classes e seus respectivos atributos e métodos que irão compor a aplicação, será possível criar essas classes, em uma linguagem com suporte a Orientação a Objetos.

Assim são apresentados com mais detalhes os componentes formadores de uma classe, baseando-se na descrição UML (*Unified Modeling Language*) RICARTE [2001]: Nome da Classe: é o meio de identificação de uma classe para referência posterior; Conjunto de Atributos: é o elemento que contém as propriedades das classes e as descreve por um nome e um tipo associado. Essa atribuição é geralmente o nome da classe ou tipos primitivos como inteiro, char, ou float (real), de cada atributo e, finalmente, Métodos: definem o que se pode fazer com os objetos de uma classe, são especificados por assinaturas que são o nome do método, o tipo para retorno e uma lista de argumentos, identificada pelo tipo de cada método e nome.

Pode-se dizer que um objeto representa uma entidade, concreta ou abstrata que interage com os outros objetos no domínio da solução do problema analisado e, por similaridade, são agrupados em classes. São pelos objetos que ocorrem o processamento de sistemas na Programação Orientada a Objetos, mas necessita-se que haja um uso racional dos objetos para não sobrecarregar as implementações orientadas a objetos. Pode-se também classificar um objeto como uma variável, pois ambos adquirem espaço em memória para armazenamento de estado e um conjunto de operações aplicadas a cada caso. Um programa orientado a objetos é composto por um conjunto de objetos que interagem por meio de aplicação de métodos a objetos, sendo esses métodos os mais privativos possíveis, conhecendo-se apenas sua especificação.

2.2 Herança & Polimorfismo

A herança permite que se crie uma classe usando outra como base e descrevendo ou implementando as diferenças e adições da classe usada como base, reutilizando os campos e métodos não-privados da classe base, também chamada de superclasse.

Para os relacionamentos em herança, temos a extensão ou herança estrita, com a extensão da superclasse pela subclasse com a adição de novos atributos e/ou métodos, permanecendo a superclasse inalterada, a especificação, que não é implementada nenhuma funcionalidade, apenas com a herança da interface da superclasse pela subclasse, a combinação de extensão e especificação ou herança polimórfica, que a subclasse herda a interface e uma implementação padrão de alguns métodos da superclasse e a subclasse pode redefinir os métodos de modo que reflita as características para o funcionamento correto da subclasse e, ainda há a contração que elimina alguns métodos da superclasse para fazer métodos mais simples, com redefinição dos métodos eliminados com corpo vazio, mas essa técnica não é indicada por se ter uma subclasse alterando métodos de uma superclasse, o que vai contra os princípios da orientação a objetos.

Segundo [SANTOS, 2003], “polimorfismo (“muitas formas”) permite a manipulação de instâncias de classes que herdam de uma mesma classe ancestral de forma unificada: podemos escrever métodos que recebam instâncias de uma classe C, e os mesmos métodos serão capazes de processar instâncias de qualquer

classe que herde da classe C, já que qualquer classe que herde de C é-um-tipo-de C". Pode-se notar que as classes herdeiras que usam polimorfismo terão comportamentos distintos, permitindo a abstração dos detalhes dos objetos quando não são necessários.

3. Programação Orientada a Aspectos

A Programação Orientada a Aspectos surgiu com o objetivo de melhorar a modularização dos interesses transversais, envolvendo requisitos não-funcionais, que tem influência em todo o programa, por meio de abstrações que possibilitam a separação e composição destes interesses na construção de sistemas de software. Desde então são realizadas pesquisas sobre o assunto para se determinar se a POA é ou não um mecanismo eficaz para o desenvolvimento de software, com o seu foco voltado à *separação de interesses*. ELRAD [2001] afirma que a POA foi proposta com o objetivo de facilitar a modularização dos interesses transversais, complementando a POO.

As três etapas distintas de desenvolvimento na POA são:

• *Decompor os interesses (aspectual decomposition):* identificar e separar os interesses transversais dos interesses do negócio;

• *Implementar os interesses (concern implementation):* implementar cada um dos interesses identificados separadamente;

• *Recompor os aspectos (aspectual recomposition):* nesta etapa, tem-se o integrador de aspectos que especifica regras de recomposição para criação de unidades de modularização. A esse processo de junção da codificação dos componentes e dos aspectos é denominada combinação (*weaving*).

Na Figura 1 são apresentadas as etapas de desenvolvimento de software orientado a aspectos, extraída de LADDAD [2003]:

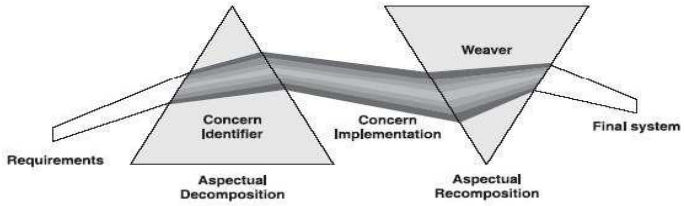

Figura 1 - Etapas de Desenvolvimento de Software Orientado a Aspectos

3.1 Aspectos

Um aspecto é o elemento básico de encapsulamento dos outros elementos que dão suporte ao uso da Programação Orientada a Aspectos, os quais são os pontos de corte, os adendos e as declarações inter-tipos que são os atributos, os métodos e as classes internas, que podem alterar a estrutura estática (*static crosscutting*), adicionando membros a uma classe; e dinâmica, interceptando pontos de junção e da adição de comportamento antes e depois desses pontos de junção ou ainda por meio da obtenção de total controle sobre o ponto de execução, de um programa, mudando a hierarquia do sistema, SOARES E BORBA [2002].

Para a declaração de um aspecto segue-se a forma geral:

```
[privileged] [modifiers] aspect Id [extends Type] [implements TypeList]
[PerClause]
{
 Body
}
```

Os *modifiers* têm a mesma definição que nas classes de programação Java. As regras de visibilidade são as mesmas. O aspecto é *público*, o mesmo é acessível e visível para todos. O aspecto é *privado* quando o mesmo é visível apenas para o aspecto que o pertence, sem exceções para os pacotes que se encontra ou pelos subaspectos. O aspecto é *protegido* quando tem a mesma visibilidade do privado e diferenciando-se do privado por poder ser visto dentro do pacote em que se encontra ou pelos subaspectos. A declaração *privileged* é utilizada quando não há outra maneira de se acessar as partes privadas das classes ou se não há a possibilidade de acesso pelos métodos *gets* e *sets*. *PerClause* é para definir a disponibilidade da instância do aspecto e o momento de se instanciar.

3.2 AspectJ

O AspectJ é um *plugin* desenvolvido para adaptação da linguagem Java para a orientação a aspectos. Essa adaptação envolve a inserção de aspectos (*aspects*), pontos de corte (*pointcuts*), pontos de junção (*joinpoints*),

adendos (*advices*) e declarações inter-tipos (*inter-type declarations*), que são os atributos, métodos e construtores.

3.2.1 Ponto de Junção (*JoinPoint*)

Um Ponto de Junção na Programação Orientada a Aspectos são pontos no fluxo de execução de um programa baseado em componentes onde os aspectos serão aplicados. Para GRADECKI [2003], os pontos de junção são utilizados principalmente para: chamada e execução de métodos*;* chamada e execução de construtores*;* execução de inicialização*;* execução de inicialização estática*;*pré-inicialização de objetos*;* inicialização de objetos; referência a campos e execução de tratamento de exceções.

A Figura 2 mostra um exemplo identificando os pontos de junção e o fluxo de execução do programa.

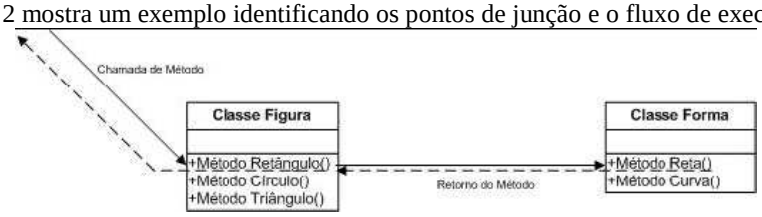

Figura 2 - Fluxo de Execução do Programa

Para demonstrar como funcionam os pontos de junção em uma execução de uma aplicação Orientada a Aspectos, considera-se que o primeiro ponto de junção no programa é a chamada de um método ou objeto qualquer, como o método *Retângulo* da Figura acima, por exemplo. Essa chamada pode ser satisfeita ou levantar uma exceção. Em seguida, o próximo ponto de junção é a execução do método *Retângulo.* Essa execução pode chamar outro método, terminar o seu trabalho com sucesso ou levantar uma exceção.

Na Figura, citada anteriormente, a execução do método *Retângulo* resultou em uma chamada a um método *Reta*, da classe *Forma*, esse método é chamado e se a chamada for completada, o ponto de junção seguinte é a execução do método *Reta* que, se concluir o seu trabalho com sucesso ou levantar uma exceção, obrigatoriamente precisa entregar o fluxo de execução de volta a execução do método *Retângulo,* se este não estiver concluído, ou esperar algum valor ou dado de retorno do método *Reta*. Como não estava concluída a sua execução, o método *Reta*, após a sua conclusão, retorna o controle para a continuação da execução do método *Retângulo,* que após terminar a sua execução e não chamar nenhum novo método, sinaliza o fim da sua execução a classe que pertence e se a classe não contiver nenhum método para executar, pode terminar a sua execução.

3.2.2 Pontos de Corte (*PointCuts*)

Os pontos de corte são os atributos existentes dentro de um aspecto declarando os pontos de junção aonde se deseja interromper a execução de um programa para a interceptação de um aspecto, mudando o comportamento desse programa.

Esses pontos de corte contêm as assinaturas de métodos que podem ser interceptados por ele para alterar o fluxo de execução original do programa e a inserção de novas chamadas, que são os adendos. A declaração de um ponto de corte nomeado deve seguir a seguinte sintaxe:

pointcut <Nome> (Argumentos): <corpo>;

Existem, ainda, os pontos de corte de inicialização de classe, para que o construtor entre em ação para construir a classe que são: *initialization,* que ocorre após a execução do construtor da classe*; preinitialization* que é o ponto de corte de pré-inicialização e que ocorre na chamada do construtor; e *staticinitialization* que é o ponto de corte de inicialização estática, que ocorre antes da chamada do construtor avisando ao sistema quando um construtor será chamado e desviando o fluxo.

3.2.3 Adendos (A*dvices*)

Adendos equivalem aos códigos a serem executados em um ponto de junção que está sendo referenciado pelo ponto de corte. Um adendo vem associado a um ponto de junção utilizando-se três palavras reservadas denominadas *before*, *after* e *around.*

Before determina que o adendo deve ser executado imediamente antes de um ponto de junção ser alcançado e *after* determina que o adendo deve ser executado depois de um ponto de junção ser alcançado. Com o adendo *after*, pode-se fazer uma especificação mais refinada, podendo-se escolher se o adendo que será executado após ocorrer um retorno normal do método (*returning*) ou após ocorrer uma exceção

(*throwing*).

Para GRADECKI [*2003*], *o* adendo pode modificar a execução do código no ponto de junção, pode substituir ou passar por ele. Usando o adendo pode-se "logar" as mensagens antes de executar o código de determinados pontos de junção que estão espalhados em diferentes módulos. O corpo de um adendo é muito semelhante ao de qualquer método, encapsulando a lógica a ser executada quando um ponto de junção é alcançado.

4. Invocação de Métodos Remotos (RMI)

O protocolo de distribuição RMI *(Remote Method Invocation* – Invocação de Métodos Remotos*)* é uma das formas de abordagem da linguagem Java de prover as funcionalidades a objetos distribuídos, até antes não fornecida pela linguagem. Para LEMAY [1999], o protocolo RMI é um mecanismo mais sofisticado para a comunicação entre objetos distribuídos Java do que seria uma simples conexão de *socket*, pois os mecanismos e protocolos por meio dos quais os objetos se comunicam são definidos e padronizados. Assim, é possível acessar um objeto ativo em uma máquina virtual, e que esse objeto virtual possa se comunicar remotamente com objetos de outras máquinas virtuais Java, não importando aonde essas outras máquinas virtuais se encontram.

Consegue-se transparência de localização com o protocolo RMI com a organização de três camadas entre o objeto cliente, que pode ser uma máquina local, e o objeto servidor, que pode ser outra máquina local ou uma máquina remota, baseado em LEMAY [1999]:

1. *Stub/skeleton (esqueleto)*: essas camadas se comportam como objetos substitutos em cada *lado*, *stub* para o objeto cliente e *skeleton* no objeto servidor, ocultando da implementação das classes a característica "remota" da chamada de objeto. São as interfaces para que os objetos da aplicação consigam interagir;
2. R*eferência remota:* trata do empacotamento de uma chamada de método e de seus parâmetros e retorna valores para transporte via camada de rede;
3. *Protocolo de transporte*: representa a conexão de rede real entre um sistema e outro.

Na Figura 3, a seguir, extraída de LEMAY [1999], está demonstrado o funcionamento dessas camadas com o uso do protocolo RMI.

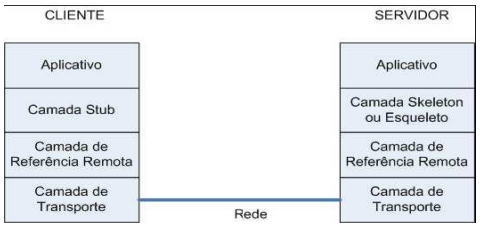

Figura 3 - Funcionamento das Camadas do Protocolo RMI

4.1 Desenvolvimento da Aplicação RMI

Para o perfeito desenvolvimento de uma aplicação cliente-servidor usando Java RMI é essencial que seja definida a interface de serviços que serão oferecidos pelo objeto servidor.

A interface remota é equivalente à definição de qualquer interface na linguagem Java, exceto pelos seguintes detalhes, RICARTE [2001]: sempre a interface de serviços RMI deve estender a interface *Remote* e é necessário declarar no corpo da interface a exceção *RemoteException* (ou uma de suas superclasses) para que possa ser gerada na execução do método.

O motivo da declaração de *RemoteException* para HORSTMANN [2001] é que as chamadas de métodos remotas são inerentemente menos confiáveis do que as chamadas locais, podendo essa chamada remota falhar.

Os serviços deverão ser implementados por meio de uma classe Java onde é necessário indicar quais objetos da classe poderão ser acessados remotamente. Para o funcionamento do serviço é necessário haver a definição da classe que implemente a interface de serviço bem como deve ser feita a inclusão de funcionalidades para a distinção dos objetos remotos dessa classe e os objetos locais. A classe precisa, também, fornecer a implementação para cada um dos métodos especificados na interface. Todas as classes do servidor devem estender a classe abstrata *RemoteServer*, do pacote java.rmi.server, que tem as funcionalidades de um servidor remoto, mas *RemoteServer* é apenas uma classe abstrata que define os mecanismos básicos para a comunicação entre os objetos servidores e seus *stubs* remotos. É a classe concreta *UnicastRemoteObject*, que permite representar um objeto que possui uma única implementação em um

servidor (ou seja, não é replicado em vários servidores) e mantém uma conexão ponto-a-ponto com cada cliente que o referencia, sem a necessidade de se escrever nenhum código.

Para HORSTMANN [2001], o relacionamento de herança entre essas classes remotas é demonstrada na Figura 4, a seguir:

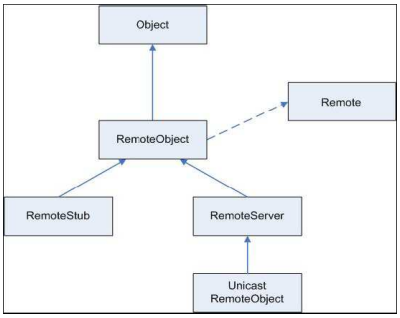

Figura 4 - Diagrama de Herança Entre Classes Remotas

4.2 Clientes e Servidores RMI

Para um cliente RMI é necessária a obtenção de uma referência remota para o objeto que implementa o serviço, com a localização de um primeiro objeto servidor, que não é necessariamente o objeto desejado, e precisa ser localizado de algum modo, o que ocorre por meio do cadastro realizado pelo servidor e, logo após obtida essa referência, não há distinção de objeto local e remoto. Pode-se também encontrar esse primeiro objeto servidor utilizando o serviço de registro de partida (*bootstrap registry service*) fornecido pela biblioteca RMI da Sun.

O aplicativo *rmiregistry* é uma implementação de um serviço de nomes para RMI, que é uma espécie de diretório onde cada serviço disponibilizado é registrado por meio de um nome de serviço sendo executado em plano de fundo aguardando solicitações em uma porta, que pode ou não ser especificada como argumento na linha de comando. Se não for especificada, a porta usada como padrão é a porta 1099, mas pode ser usada qualquer porta com número a partir de 1024.

A partir de uma classe Java, o serviço de nomes tem dois mecanismos básicos, o primeiro utiliza a classe *Naming*, do pacote java.rmi, que descreve o comportamento para se obter referências para objetos remotos baseados pelo método estático *lookup*(String nome). Além de *lookup*() os métodos *bind*(), *rebind*(), *unbind*() e *list*(), descritos na seqüência, são também suportados, todos das classes do pacote java.rmi.registry, o quais oferecem uma classe e uma interface para que classes Java tenham acesso ao serviço de nomes RMI.

Para representar o registro de objetos RMI operando em uma máquina específica é utilizada a interface Registry possibilitando a invocação do método *bind*(), que associa um nome de serviço (uma *String*) ao objeto que o implementa.

As tarefas que devem ser realizadas por meio de um objeto servidor RMI simples são as seguintes: criação de uma instância do objeto que implemente o serviço e disponibilização do serviço por meio do mecanismo de registro.

Pode-se também deixar o programa ativo indefinidamente, criando um objeto de uma classe que estende *UnicastRemoteObject*, iniciando-se uma linha de execução separada para esse propósito, além de poder instalar um gerenciador de segurança para o objeto servidor, para o controle de objetos clientes e quais acessos estão disponíveis para esses objetos clientes.

O cliente RMI é implementado para utilizar a interface remota para a obtenção da referência remota para o objeto (remoto) que implementa o serviço desejado, sendo utilizada geralmente a classe java.rmi.Naming. Após a referência remota ser obtida, precisa-se gerar as camadas de *stubs* e *skeleton*, para os métodos oferecidos remotamente serem invocados e, assim, possam funcionar os dois lados do processo.

Stubs e *skeletons* são classes auxiliares internas responsáveis pela comunicação entre o objeto cliente e o objeto que implementa o serviço para realizar o acesso remoto do protocolo RMI. Um *stub* oferece implementações dos métodos do serviço remoto as quais são invocadas no lado do cliente, onde são empacotados os argumentos que serão enviados ao servidor. *O skeleton* desempacota os dados e invoca o método do serviço no lado servidor.

A disponibilidade do serviço de registro RMI é a base para a execução de uma aplicação utilizando-se o aplicativo *rmiregistry* que, quando disponível, o servidor pode ser executado. Para tanto, essa máquina virtual Java deverá ser capaz de localizar e carregar as classes do servidor, da implementação do serviço e do

skeleton.

Após a ativação da aplicação do servidor, o mesmo deve enviar uma mensagem dizendo que está pronto a receber solicitações dos objetos clientes. Com essa mensagem recebida, o código cliente pode ser executado pela máquina virtual e deverá ser capaz de localizar e carregar as classes com a aplicação cliente, a interface do serviço e o *stub* para a implementação do serviço.

5. O Aplicativo

Nesta seção será abordado o estudo comparativo entre uma aplicação a qual utiliza o protocolo de comunicação RMI Orientado a Objetos e uma aplicação a qual utiliza o mesmo protocolo Orientado a Aspectos. Serão também avaliadas as vantagens e desvantagens de cada um dos modos de implementação.

Esta proposta foi escolhida porque um dos problemas verificados na Programação Orientada a Objetos é que os protocolos de distribuição tendem a dificultar a manutenção, prejudicar o entendimento dos códigos fontes e provocar re-trabalhos na troca de tecnologias. Tais dificuldades geram o chamado de espalhamento de código (*spread code*). O espalhamento de código, segundo RESENDE E SILVA [2005] pode ser resolvido com o uso da POA, separando a lógica de negócio da distribuição.

O aplicativo simula um supermercado onde, após a geração das compras, denominada de "Carrinho" e a colocação de itens no mesmo, o "Carrinho" é enviado remotamente para um "Caixa" o qual soma a compra e retorna seu valor total. A Figura 5 abaixo, demonstra o relacionamento entre as classes e a Tabela 1 apresenta uma descrição sobre cada uma das classes envolvidas.

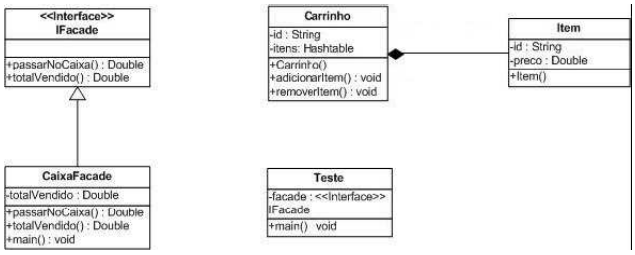

Figura 5 - Diagrama de Classes do Aplicativo Desenvolvido

Tabela 1 - Descrição das Classes Utilizadas no Aplicativo

Elementos	Descrição
Carrinho	Carrinho de compras do mercado.
Item	Cada item ou produto a venda no mercado.
CaixaFacade	Caixa, responsável por somar a compra e retornar seu valor total ao objeto cliente.
IFacade	Interface implementada por "*CaixaFacade*" que especifica quais os métodos podem ser invocados remotamente
Teste	Classe que executa e testa a aplicação

5.1 Implementação – Programação Orientada a Objetos

Na Programação Orientada a Objetos, a implementação do aplicativo precisou de inserções de código referente ao protocolo RMI. Isso pode ser prejudicial ao desenvolvimento do aplicativo, ou pelo menos torna mais difícil uma migração futura para um outro protocolo de distribuição, como CORBA, por exemplo, e sua reusabilidade e manutenibilidade ficam mais complexas. A seguir, comenta-se sobre os detalhes necessários das classes para que a utilização do protocolo RMI seja elaborada.

A interface *Serializable* é incluída nas classes Carrinho e Item para que o compilador da linguagem Java identifique que instâncias dessas classes possam ser divididas em pacotes e enviadas remotamente pela rede. A seguir é necessário estender a Interface *IFacade*, a qual é responsável para que as instâncias da classe "*CaixaFacade*" sejam acessados remotamente e quais métodos remotos devem ser disponibilizados pela classe. Para ser completa a implementação Orientada a Objetos desse supermercado foi necessária a implementação do método principal main(args) para a publicação no servidor remoto uma instância de *CaixaFacade*. Além disso, observa-se que foram necessárias às importações de cinco pacotes RMI para a

utilização da implementação Orientada a Objetos, deixando o código ainda mais interligado com o protocolo RMI, dificultando a sua migração para outro protocolo. Os métodos públicos *passarNoCaixa* e *totalVendido* utilizam a exceção *RemoteException* do pacote *java.rmi.RemoteException*, o qual indica se houve uma exceção no método remoto ou a transação foi concluída com sucesso. Além disso, *passarNoCaixa* utiliza o pacote *java.util.Iterator* para a identificação do método itens de modo interativo pela rede, tratando dos objetos que vão trafegar pela rede.

A interface *IFacade* foi estendida a interface *Remote*, do pacote *java.rmi.Remote*, para servir de ponte de acesso remoto do programa Orientado a Objetos, nas declarações dos métodos *passarNoCaixa* e *totalVendido*, que determinam as funcionalidades fornecidas aos objetos remotos, sendo uma instância da classe *CaixaFacade* neste caso. Se esta interface não fosse definida, seria necessária a invocação da interface *Remote* para cada método, o que tornaria o código do aplicativo mais extenso e confuso ao projetista ou ao usuário desse sistema. Na classe Teste, responsável por testar a aplicação, foi necessário implementar um método remoto *getFacade*(), para a localização do objeto remoto no servidor. No método principal *main(args)*, são definidas as utilizações da interface IFacade usando o método remoto *getFacade*(), os itens existentes no aplicativo, criados com o uso do construtor *new*, os carrinhos também construídos pelo construtor *new* e a colocação de itens dentro do carrinho. Em uma avaliação preliminar enfatiza-se a redundância a qual a aplicação Orientada a Objetos (OO) alcançou: em todas as classes apresentadas foi necessária a inclusão do pacote *java.rmi.RemoteException* para a obtenção da conexão com o servidor ou objeto remoto.

5.2 Implementação – Programação Orientada a Aspectos

Para a implementação do aplicativo por meio da utilização da abordagem Orientada a Aspectos, necessita-se fazer algumas alterações no código do aplicativo, as quais são citadas a seguir:

• A interface *Serializable* é inserida nas classes Carrinho e Item por meio de *Introductions*, alterando a estrutura dessas classes. As *introductions* também podem ser utilizadas para o mesmo fim para a inserção da interface *IFacade* e do método principal *main(args)* que contém os códigos específicos de RMI na classe "*CaixaFacade*";

• Criação de um ponto de corte para a interceptação das chamadas na classe *CaixaFacade* em substituição do método *getFacade*() de Teste. Para completar, é criado um adendo que, antes da execução deste ponto de corte, obtenha uma referência remota ao objeto que se encontra no servidor e;

• Criação de dois adendos para a interceptação de chamadas aos métodos *passarNoCaixa*(Carrinho) e *totalVendido*() os quais foram implementados na classe *CaixaFacade*. Deve-se ainda desviar a execução dos métodos citados para a invocação do equivalente remoto.

A Orientação a Aspectos mostra-se mais enxuta se comparada a Orientação a Objetos no exemplo, pois a distribuição RMI pôde ser canalizada no aspecto *RMIAspect* e na interface *IFacade* a qual não sofreu alterações. Com isso, foi possível abstrair a complexidade da execução das chamadas remotas pela classe Teste. Como essa implementação é intermediária e não a definitiva, faz-se aqui a adaptação inicial do aplicativo orientado a objetos para a orientação a aspectos, por isso, encontrar-se-á implementações relacionadas ao protocolo RMI nas classes Teste e *CaixaFacade*, onde precisa-se fazer algumas alterações para que as classes sejam separadas da parte que envolve as chamadas remotas.

Para esse aplicativo específico é criada uma exceção específica denominada *CaixaException*, a qual substituirá a exceção *RemoteException* da classe *CaixaFacade*. Os adendos do aspecto *RMIAspect* que antes esperavam um *RemoteException* ao invocar remotamente métodos de *CaixaFacade* necessitam de modificações para suportar a nova exceção.

É criada uma nova exceção filha da classe Exception, de controle de exceções na linguagem Java e utiliza a expressão super para dizer que os métodos *message* e *cause* são métodos instanciados da classe pai *Exception*. Agora a classe *CaixaException* pode ser usada especificamente para o aplicativo não necessitando chamar a classe *RemoteException* com todos os métodos possíveis de exceção remota.

Fez-se o aspecto *RMIAspect* o qual é uma junção de todas as possíveis chamadas a classes, métodos e atributos da implementação. Além da declaração de *CaixaFacade* e de Carrinho como sendo as implementações pais do aspecto, no método principal (*main*) de *CaixaFacade* foram declaradas as instâncias para a conexão com o servidor remoto. Existe ainda a importação do pacote *java.rmi.RemoteException* e uma declaração de *RemoteException* e o seu retorno, o que não é desejável, pois aumenta o espalhamento de código.

A declaração da Interface *IFacade*, do ponto de corte (*pointcut*) *facadeExecution*, intercepta todas as

chamadas aos métodos da classe *CaixaFacade* sem alterações realizadas. Um adendo declarando a execução de *prepararFacade*() antes do ponto de corte *facadeExecution*() seja alcançado para a referência ao objeto remoto.

Logo após a execução de *prepararFacade*(), um adendo, com a declaração *around*, toma o controle do fluxo de execução do programa e faz a chamada do método *passarNoCaixa*(Carrinho) da classe *CaixaFacade* para a passagem dos itens do carrinho pelo servidor remoto, se a conexão e a referência ao objeto remoto forem realizadas com sucesso. Se os itens dos carrinhos foram computados com sucesso é retornado pelo método *passarNoCaixa*() que tudo ocorreu bem. Caso contrário será gerada uma exceção.

Por último, há um adendo que retorna o método *totalVendido*() com o valor dos itens de cada carrinho se tudo ocorreu como o esperado, senão retorna a exceção *CxEx* de *CaixaException*. Mesmo com a diminuição do uso da exceção *RemoteException* com as modificações feitas, os aspectos precisam obrigatoriamente incluir no seu código o suporte para a exceção *RemoteException*, para a utilização do protocolo RMI. Mas essa nova alteração é simples de ser feita, pois se necessita apenas alterar a herança da exceção *CaixaException* para *RemoteException* e também silenciar a ocorrência de *RemoteException* no aspecto *RMIAspect*, uma vez que ele invoca diretamente métodos da interface remota *IFacade*.

Conclusões

Em um primeiro momento, avaliando-se apenas a implementação em si, a Programação Orientada a Aspectos é mais atrativa que a Programação Orientada a Objetos, pois o resultado do método orientado a aspectos é mais "limpo" que o método orientado a objetos, traduzido pela maior modularização, significando um menor espalhamento de código, causado principalmente pelos requisitos transversais, que tem influência direta do resultado da implementação e, também, pela possibilidade de uma maior reutilização desses módulos separados em programas com características similares, mas com objetivos diferentes. Além disso, a técnica orientada a objetos tem uma menor adequação ao aplicativo estudado, por ter um requisito transversal que influencia no modo de implementação desse aplicativo, que é o protocolo de distribuição RMI, que entrelaça muito o código com sucessivas chamadas aos métodos externos, tornando-o mais repetitivo e, com o uso da orientação a aspectos, essa característica de entrelaçamento de código é minimizada.

Os testes foram realizados baseando-se no aplicativo Eclipse, versão 3.0, que é um compilador de linguagem Java, já embutido nela o *plugin* AspectJ, que é *o weaver*, ou o interpretador orientado a aspectos e também foi utilizado o aplicativo SDK 1.4, que foi utilizado para a compatibilidade dos aplicativos testados com a biblioteca *RemoteException* do protocolo RMI. Os três aplicativos: o primeiro aplicativo orientado a objetos; o segundo aplicativo utilizando orientação a aspectos utilizando o protocolo a biblioteca *java.rmi.RemoteException* para o levantamento das exceções produzidas e, finalmente, o terceiro utilizando o levantamento de exceção baseado em uma interface *CaixaException*, funcionaram dentro do esperado no teste realizado em uma máquina que serviu como objeto cliente e objeto servidor simultaneamente, retornando o resultado correto, que é o retorno do método *valorVendido* produzido pelo registro dos itens passados pelo caixa desse mercado.

A maior dificuldade encontrada ao ajuste para o perfeito funcionamento dos aplicativos foi a adaptação à orientação a aspectos. Há uma vantagem na utilização da Programação Orientada a Aspectos por se ter uma maior modularização e separação dos componentes do aplicativo, mas na integração desses componentes deve-se ter o maior cuidado com a declaração dos outros módulos pelo módulo principal. Por outro lado, a aplicação orientada a objetos também foi eficiente ao aplicativo proposto, mas há uma possibilidade menor de modularização, com repetitivas chamadas às bibliotecas, principalmente a que envolve o protocolo RMI, não sendo a alternativa mais adequada a implementações com códigos muito extensos, por se ter que usar essa característica repetitiva da implementação.

Trabalhos Futuros

Este trabalho propôs uma avaliação dos métodos orientados a objetos e orientados a aspectos com o protocolo de distribuição RMI, sendo esse protocolo nativo da linguagem Java. Pode-se chegar a resultados diferentes quando compara-se o protocolo de distribuição RMI com o protocolo de distribuição CORBA, que é multi-linguagem, na Programação Orientada a Objetos e na Programação Orientada a Aspectos. Acredita-se que o código possa crescer exageradamente de tamanho ustilizando CORBA. Também faz-se necessário avaliar o comportamento da Programação Orientada a Aspectos utilizando outras linguagens que suportam *plugins* com essas características como C, Python, Smalltalk entre outras. Além dessa nova possibilidade de avaliação, pode-se também considerar o *plugin* instalado, ou seja, pesquisar como seria o comportamento de um outro

plugin que não seja o AspectJ.

Referências

ELRAD, T.; FILMAN, R.E.; BADER, A. *Aspect–oriented Programming*. Communications of the ACM. Vol. 44, n. 10, 2001.

GRADECKI, J.; LESIEC, N. *Mastering AspectJ: aspect-oriented programming in Java*. John Wiley & Sons, 2003.

HORSTMANN, C.S.; CORNELL G. *Core Java 2 – Volume II – Recursos Avançados*. Makron Books, São Paulo, 2001.

LEMAY, L.; CADENHEAD, R. *Aprenda em 21 Dias Java 2*. Campus, São Paulo , 1999.

MELLO, R.; CHIARA, R.; VILLELA, R. *Aprendendo Java 2*. Novatec, São Paulo, 2002.

RESENDE, A.; SILVA, C. *Programação Orientada a Aspectos em Java – Desenvolvimento de Software Orientado a Aspectos*. Brasport, Rio de Janeiro, 2005.

RICARTE, I. (2001) *Programação Orientada a Objetos: Uma Abordagem com Java*. UNICAMP. Disponível em: http://www.dca.fee.unicamp.br/cursos/PooJava/sobre.html.

ROYCE, W. *Software Project Management - A Unified Framework*.Addison- Wesley, 1998.

SANTOS, Rafael. Introdução à Programação Orientada a Objetos Usando Java. Campus, São Paulo, 2003.

SUN MICROSYSTEMS. (1997) Java RMI Tutorial. Revision 1.3, JDK 1.1 FCS, 1997. Disponível em: http://www.metz.supelec.fr/~vialle/course/SI/java-rmi/doc- JavaRMI.pdf