Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Reestruturando o MobileServer com AspectJ

por
Leonardo Cole Neto

Trabalho de Graduação

Recife, 30 de setembro de 2002

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Leonardo Cole Neto

Reestruturando o MobileServer com AspectJ

Este trabalho foi apresentado à Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito para a disciplina Trabalho de Graduação em Engenharia de Software.

ORIENTADOR:
Prof. Paulo Henrique Monteiro Borba

Agradecimentos

Agradeço primeiramente a Paulo Borba, meu orientador, por toda a ajuda e orientação durante o desenvolvimento deste trabalho, agradeço também a Sergio Soares, pois sem sua cooperação este trabalho não teria se realizado.

Agradeço ainda a Mobile por deixar que um de seus produtos fosse o estudo de caso deste trabalho, assim como ao meu gerente e colegas de trabalho pela compreensão e incentivo nos momentos de dificuldade, especialmente a Flávio Couto e Jones Oliveira, cuja ajuda contribuiu bastante para enriquecimento deste documento.

E por fim, agradeço à minha família e amigos por me ajudarem a relaxar nos momentos defíceis assim como por compreender minha ausência por um longo período.

Resumo

Este trabalho relata uma experiência na utilização de Orientação à Aspectos (AOP - Aspect Oriented Programing) com AspectJ [11], uma extensão orientada à aspectos de Java [7], na implementação dos aspectos de concorrência, persistência e *logging* de um sistema de médio porte anteriormente implementado puramente em Java com Orientação a Objetos.

Neste trabalho, relatamos algumas melhorias alcançadas através do uso de AOP na re-estruturação do sistema originalmente implementado em Java. Discutimos também alguns problemas encontrados.

Alguns dos benefícios agregados ao sistema pelo uso de AOP foram um bom aumento na modularidade e manutenabilidade, através da separação de código não relacionado, e em alguns casos, diminuição do tamanho do código, reduzindo sua complexidade. Houve ainda uma grande melhoria na capacidade de customização do sistema, já que os aspectos implementados podem ser facilmente retirados, fazendo com que o sistema possa executar com ou sem o este aspecto.

Encontramos também alguns pontos negativos como a dificuldade de depuração do código com AspectJ e, principalmente, problemas relacionados ao fato do sistema não haver sido elaborado para uso de AOP, indicando que em muitos casos, o desenvolvimento planejado para uso com AOP pode trazer ainda mais benefícios.

Sumário

Lista de Figuras

1 Introdução

A necessidade de desenvolver software de qualidade aumentou o uso da orientação a objetos (OO) [14, 3] na indústria, trazendo maiores níveis de reuso e manutenibilidade, aumentando a produtividade do desenvolvimento e o suporte a mudanças de requisitos. Entretanto, o paradigma orientado a objetos tem algumas limitações [16, 17], como o entrelaçamento e o espalhamento de código com diferentes propósitos. Por exemplo, o entrelaçamento de código de negócio com código de apresentação, e o espalhamento de código de acesso a dados em vários módulos do sistema, prejudicam a manutenção e a extensão do mesmo, de modo que o desenvolvedor deverá preocupar-se com fatos além das regras de negócio da classe que está implementando ou mantendo.

Parte destas limitações pode ser compensada com o uso de padrões de projetos [4, 6]. Um exemplo do uso de padões de projeto na solução destas limitações seria a utilização do padrão *adapter* [1] para separar todo o código relacionado com a distribuição de uma parte do sistema, de forma a tornar esta distribuição transparente e isolar o código que tem somente o objetivo de realizar esta tarefa. A utilização de um padrão de projeto como esse aumenta a complexidade do sistema, no caso do padrão *adapter*, pelos menos 2 classes são inseridas no sistema e assim mais classes são necessárias para que o sistema seja entendido por completo.

Por outro lado, extensões do paradigma orientado a objetos, como *subject-oriented programming* (SOP) [17], *adaptive programming* (AP) [10] e *aspect-oriented programming* (programação orientada a aspectos) [5], tentam solucionar as limitações do paradigma orientado a objetos. Estas técnicas visam obter uma maior modularidade de software em situações práticas onde a orientação a objetos e padrões de projetos não fornecem o suporte adequado.

A última destas técnicas, programação orientada a aspectos (AOP), parece bastante promissora [15, 5, 12] e ambos SOP e AP podem ser vistos como um caso especial dela [8, 9]. AOP procura solucionar a ineficiência em capturar algumas das importantes decisões de projeto que um sistema deve implementar. Esta dificuldade faz com que a implementação destas decisões de projeto seja distribuída pelo código, resultando num entrelaçamento e espalhamento de código com diferentes propósitos. Este entrelaçamento e espalhamento tornam o desenvolvimento e a manutenção destes sistemas extremamente difíceis. Desta forma, AOP aumenta a modularidade separando código que implementa funções específicas, afetando diferentes partes do sistema, chamadas preocupações ortogonais (*crosscutting concerns*). Exemplos de *crosscutting concerns* são persistência, distribuição, controle de concorrência, tratamento de exceções, *Logging* e depuração. O aumento da modularidade implica em sistemas legíveis, os quais são mais facilmente projetados e mantidos.

Uma das formas mais comuns na aplicação de AOP, permite que a implementação de um sistema seja separada em requisitos funcionais e não-funcionais. Dos requisitos funcionais resulta um conjunto de componentes expressos em uma linguagem de programação atual, por exemplo, a linguagem Java [7]. Já dos requisitos não-funcionais resulta um conjunto de aspectos (*crosscutting concerns*) relacionados às propriedades que afetam o comportamento do sistema. Com o uso da abordagem, os requisitos não-funcionais podem ser facilmente manipulados sem causar impacto no código de negócio (requisitos funcionais), uma vez que estes códigos não estão entrelaçados e espalhados em várias unidades do sistema. Desta forma, AOP possibilita o desenvolvimento de programas utilizando tais aspectos, o que inclui isolamento, composição e reuso do código de implementação dos aspectos.

Este trabalho teve como objetivo, a aplicação de AOP a um sistema real implementado puramente com Java e OO (Orientação a Objetos), na tentativa de agregar as vantagens já

citadas pelo uso de AOP, e então relacionar custos e benefícios da utilização deste paradigma.

A linguagem utilizada para implementação do trabalho foi AspectJ [11], uma extensão orientada a aspectos, de propósito geral, da linguagem Java [7]. Esta foi a linguagem escolhida por sua compatibilidade com Java, sendo capaz de executar em JVM(Java Virtual Machine) já existentes, além de facilitar o uso de AOP por programadores Java.

Preocupações de persistência, concorrência e distribuição são alguns dos grandes problemas a serem considerados no desenvolvimento de um sistema, pois envolvem diversas partes distintas deste e mesmo o uso cuidadoso de padrões de projeto não é suficiente para eliminar a fraqueza da implementação de tais aspectos em linguagens puramente OO. De fato, tais preocupações não podem ser completamente separadas de outras utilizando puramente linguagens OO.

Os aspectos cobertos no MobileServer foram Persistência, Concorrência e *Logging*. Eles foram separados e isolados do código orininal de forma a tornar o sistema mais modular, aumentar o grau de customização e ainda melhorar a manutenabilidade e extensibilidade do sistema.

O restante deste documento está organizado como descrito a seguir: A Seção 2 fala sobre o caso de uso utilizado, o MobileServer, de forma que o leitor se familiarize um pouco com os pontos abordados aqui. A Seção 3 dá uma visão geral da linguagem AspecJ e sua syntaxe, já que algum conhecimento desta é necessário para demonstrar a aplicação de AOP. As três seções seguintes tratam do experimento realizado, cobrindo o aspecto de concorrência na Seção 4, persistência na Seção 5 e *logging* na Seção 6. Cada uma destas seções possui uma sub-seção que comenta os resultados obtidos com cada aspecto. Por fim encerramos com a Seção 7, onde as conclusões deste experimento são enfatizadas.

2 O MobileServer

O estudo de caso utilizado foi o MobileServer, um produto desenvolvido pela unidade de computação móvel da Mobile S.A.. Este sistema tem o objetivo de replicar e sincronizar bases de dados de sistemas legados, entre dispositivos móveis (*hand-helds* e PDA´s). Este projeto foi escolhido por estar em fase de desenvolvimento, além de ser um sistema de médio porte que pode receber grandes benefícios providos pelo uso de AOP.

2.1 Arquitetura do Sistema

A arquitetura do sistema MobileServer pode ser dividida em alguns módulos: o Servidor, o sistema de retaguarda e a parte de sincronização. Estes componentes podem ser vistos na Figura 1. Descrevemos a seguir, cada um deles.

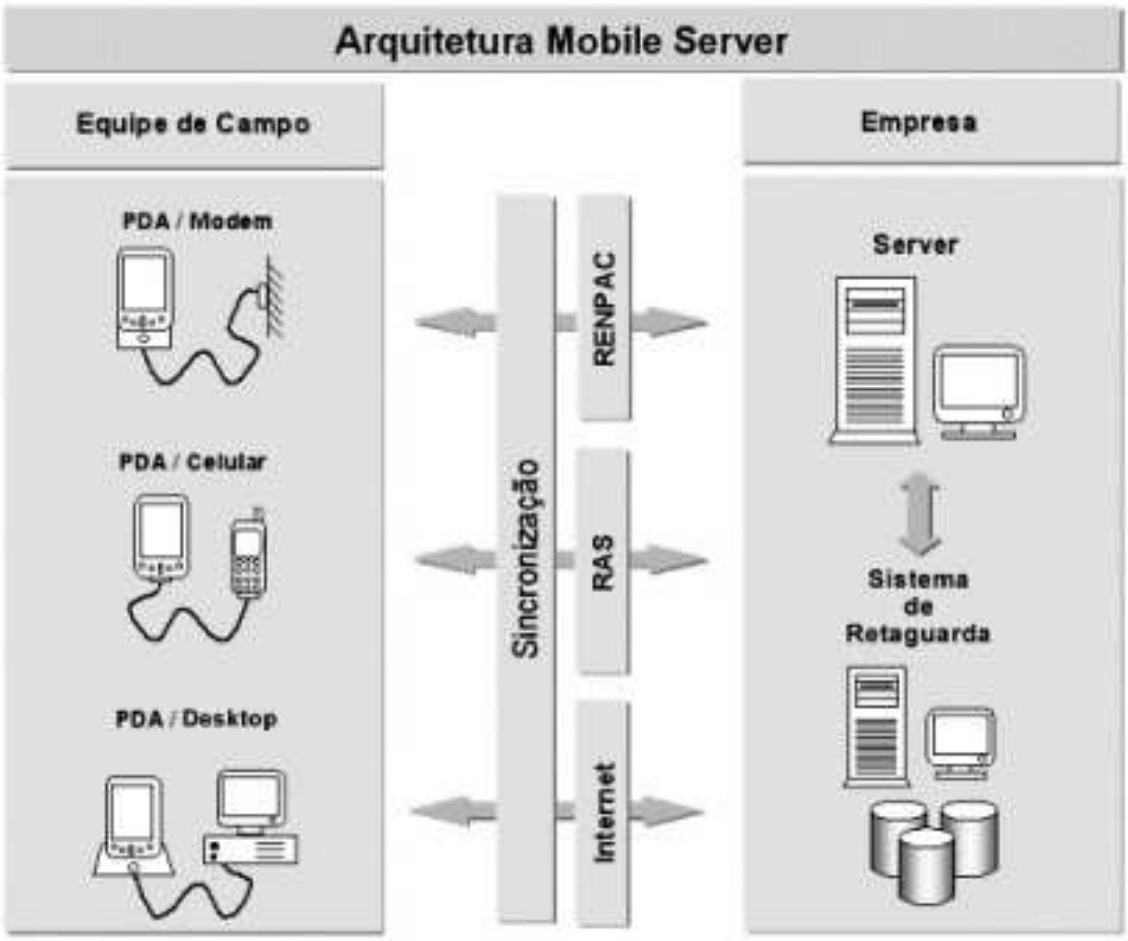

Figura 1: Arquitetura do Sistema.

Servidor: Aplicação servidora responsável pela replicação de dados de um sistema legado (Sistema de Retaguarda) para dispositivos móveis (PDA's). O Servidor é responsável por tornar transparente a utilização e atualização da base de dados do sistema de retaguarda por aplicações executando em algum dispositivo móvel. No processo de comunicação, o Servidor faz a importação dos dados vindos da retaguarda e em seguida os

envia para os PDA's. Também é responsabilidade do servidor fazer a exportação dos dados recebidos dos PDA's para a retaguarda, no intuito de manter as mesmas informações em todas as bases envolvidas.

Sistema de Retaguarda: Sistema legado que deseja possuir uma aplicação distribuída sendo utilizada por dispositivos móveis. Como exemplo temos aplicações de força de vendas, onde cada vendedor possui um PDA com as informações do sistema de retaguarda e assim realiza vendas e pedidos através de seu PDA. Tais vendas são transmitidas e confirmadas ou modificadas pelo sistema de retaguarda. Isto ocorre de forma transparente através do Servidor.

Sincronização: É a forma como os PDA's e o sistema de retaguarda comunicam-se com o Servidor, podendo ser pela Internet, RAS (*Remote Access Services*) e RENPAC (Rede Nacional de Pacotes da EMBRATEL). A conexão entre o PDA e o Servidor, para que utilize um dos meios de comunicação acima, poderá ser feita de duas formas:

1. Utilização de Modem ou Celular: neste caso o PDA estará conectado a um modem ou telefone celular, que permitirá o acesso ao Servidor através da Internet;
2. Através do Desktop: neste caso o PDA acessará o Servidor através da conexão Internet do desktop.

2.2 Arquitetura do Servidor

A Figura 2 ilustra a arquitetura interna do componente Servidor, mostrando todos os módulos do mesmo. O foco deste trabalho encontra-se na base de dados do Servidor e nos processos que interagem com a mesma, pois o acesso concorrente à base deve ser controlado, assim como persistência é uma de suas características inerentes. Descrevemos em seguida os módulos considerados neste documento.

A **Base de Dados** é uma coleção de tabelas que reflete o modelo de dados do sistema de retaguarda, mantendo, no caso geral, uma réplica de cada tabela para cada tipo de dispositivo móvel a ser utilizado. Tais réplicas são necessárias para o cálculo das diferenças entre as tabelas de cada um dos dispositivos, possibilitando assim o envio de uma atualização incremental (somente a informação modificada) para tais dispositivos.

O modelo de classes da base de dados é mostrado na Figura 3. A arquitetura da base de dados segue um padrão de divisão em camadas [2], contendo as camadas de dados e negócio. A camada de dados é formada pela interface `Repository`, que oferece todas as operações que podem ser realizadas sobre a coleção de tabelas. A classe `RepositoryFacade`, juntamente com a estrutura de dados formada pelas interfaces `Table`, `Register` e `Field`, forma a camada de negócio deste repositório de tabelas. Os objetos destas classes representam, respectivamente, tabelas, registros e campos de um banco de dados. A implementação destas interfaces não utiliza necessariamente um banco de dados relacional, podendo utilizar, por exemplo, uma estrutura de aqruivos.

O **Processamento de Retorno** consiste no processamento das informações recebidas pelo Servidor, ou seja, atualização com as novas informações recebidas da réplica correspondente a fonte da informação na base de dados.

Processamento de Remessa é o processo que realiza o cálculo das diferenças para cada uma das réplicas de uma tabela, tendo como saída as informações de atualização a serem

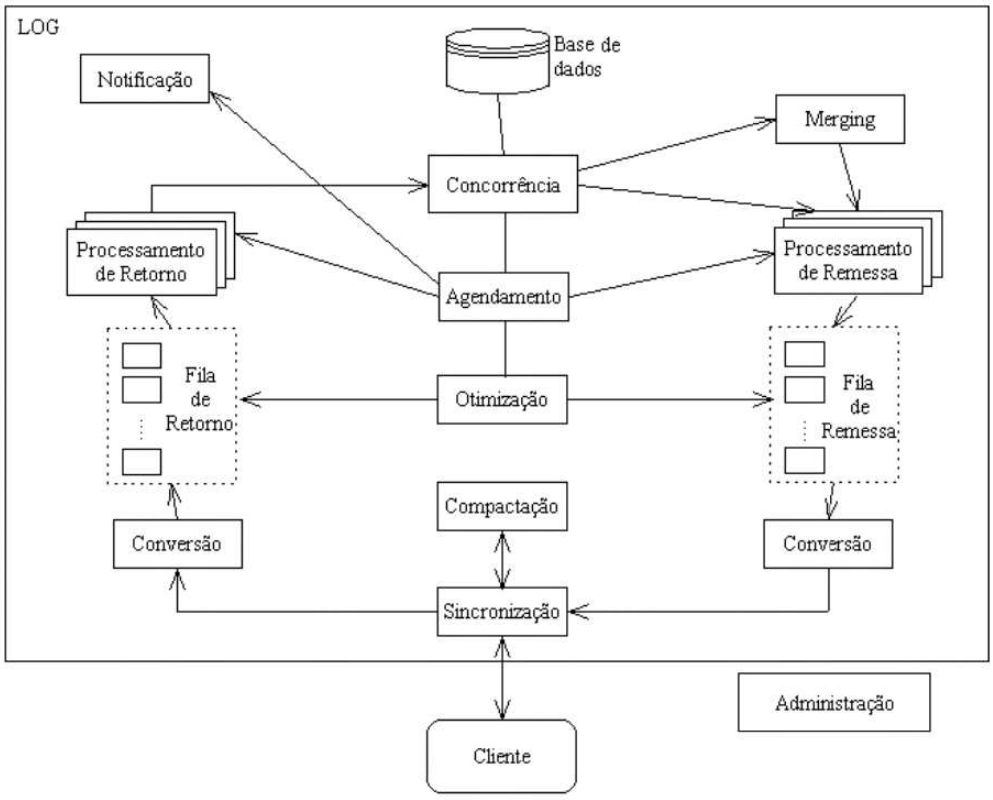

Figura 2: Arquitetura do Servidor.

enviadas para cada dispositivo, com a intenção de tornar idênticas as bases dos dispositivos que fazem uso do MobileServer.

O processo de **Recarga** é um caso particular do processamento de remessa e, por esta razão, não é mostrado como um módulo a parte na Figura 2. Este processo consiste na geração de uma cópia inteira da base de dados para um dispositivo móvel, este processo é necessário somente em casos de perda das informações em tal dispositivo.

A interface de administração possibilita criar, remover e configurar tabelas na base de dados. O operador do sistema é responsável pela realização destas tarefas, mantendo o modelo de tabelas da base de dados atualizado.

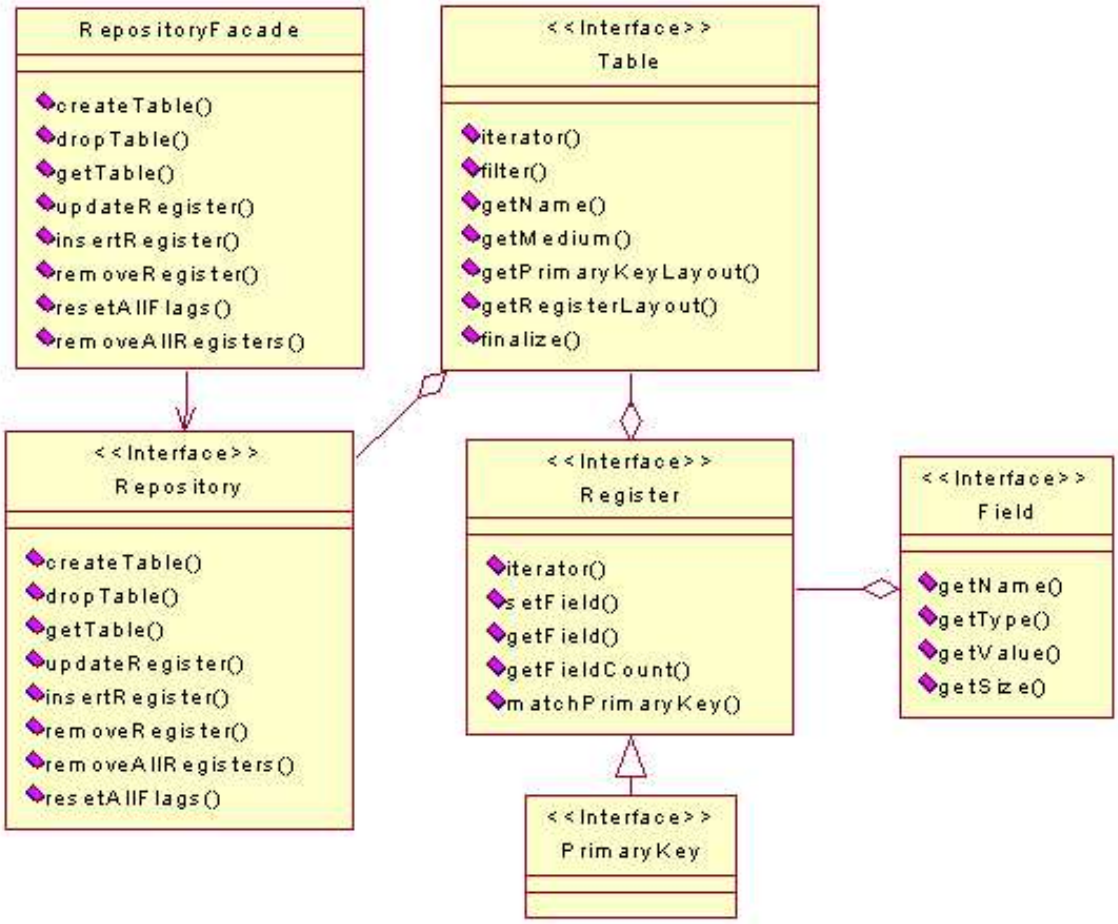

Figura 3: Modelo de classes da Base de Dados.

3 Visão geral de AspectJ

AspectJ [11] é uma extensão, orientada a aspectos de Java. A orientação a aspectos suporta a definição modular de conceitos normalmente entrelaçados pelo sistema, ou seja, conceitos que afetam diversas partes do mesmo. Esta separação de conceitos permite a construção de um sistema mais modular, evitando o espalhamento desnecessário de código, facilitando ainda a manutenabilidade do sistema.

Programar com AspectJ envolve o uso tando de aspectos quanto de classes para separação de conceitos. Conceitos que podem ser bem modelados com classes são implementados com OO; conceitos que entrelaçam objetos são separados usando unidades chamadas aspectos, e estas são unidas às classes através de um processo chamado *weaving*, mostrado na Figura 4. Unindo aspectos definidos com AspectJ e classes definidos com Java, obteremos uma aplicação AspectJ.

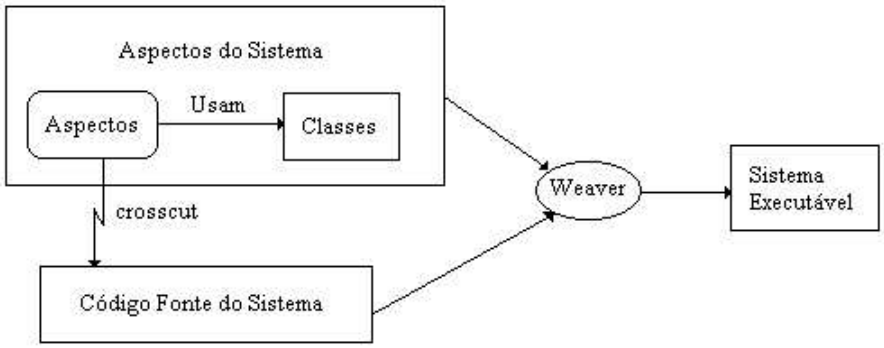

Figura 4: *Weaving*, união de aspectos e classes.

3.1 O Aspecto

A principal unidade de AspectJ [11] é o aspecto (*aspect*). Cada aspecto define uma funcionalidade que afeta diferentes partes do sistema. Um aspecto pode declarar atributos e métodos, além de estender outros aspectos definindo um comportamento concreto para um aspecto definido como abstrato.

Um aspecto pode ser usado para modificar a estrutura estática de programas Java. Este mecanismo permite a introdução de novos métodos e atributos em uma classe já existente. Por exemplo, podemos querer modificar uma dada classe, `MyClass`, para indicar que a mesma possui um novo atributo do tipo inteiro chamado `myNewInt` (linha 1) e que possui um novo método chamado `myNewInt()` (linha 2).

```
1: private int MyClass.myNewInt;
2: public int MyClass.getNewInt() { ... };
```

Uma outra forma de alteração estática permite converter exceções checadas para exceções não checadas. No exemplo a seguir, estamos dizendo que a exceção `MyException` poderá ser levantada pelo método `myMethod()` da classe `MyClass`, sem que outras classes que utilizem este método precisem tratá-la.

```
declare soft: MyException: MyClass.myMethod();
```

Por fim, podemos também modificar a hierarquia de classes fazendo com que uma classe existente herde de uma outra ou implemente uma interface. A linha 1 do exemplo indica que a classe `MyClass` implementa a interface `List` e a linha 2 indica que a classe `MyClass` herda da classe `Vector`.

```
1: declare parents: MyClass implements List;
2: declare parents: MyClass extends Vector;
```

Aspectos também podem modificar a estrutura dinâmica de execução de um programa. Os aspectos podem interceptar certos pontos de execução, chamados *join points*, e modificar o comportamento antes (*before*), depois (*after*) ou executar no lugar (*around*) do código capturado por estes pontos. Exemplos de *join points* são as chamadas dos métodos, execução dos métodos, execução de construtores, acessos a atributos (*get* e *set*), tratamento de exceções, inicializações estáticas, e combinações destes e outros através dos operadores lógicos `!`, `&&` e `||`.

3.2 Pointcut e Advice

Normalmente, aspectos declaram *pointcuts*. *Pointcuts* são as construções de AspectJ que selecionam alguns *join points* e valores no contexto destes. Para definir *pointcuts*, identificando os *join points* a serem afetados, utilizamos construtores de AspectJ chamados designadores de *pointcut* (*pointcut designators*), como os apresentados a seguir.

`call`(*Assinatura*)	Invocação de método/construtor identificado por *Assinatura*
`execution`(*Assinatura*)	Execução de método/construtor identificado por *Assinatura*
`get`(*Assinatura*)	Acesso a atributo identificado por *Assinatura*
`set`(*Assinatura*)	Atribuição de atributo identificado por *Assinatura*
`this`(*PadrãoTipo*)	O objeto em execução é instância de *PadrãoTipo*
`target`(*PadrãoTipo*)	O objeto de destino é instância de *PadrãoTipo*
`args`(*PadrãoTipo*, ...)	Os argumentos são instâncias de *PadrãoTipo*
`within`(*PadrãoTipo*)	O código em execução está definido em *PadrãoTipo*

Onde *PadrãoTipo* é uma construção que pode definir um conjunto de tipos utilizando *wildcards*, como $*$ e $+$. O primeiro é um *wildcard* conhecido, pode ser usado sozinho para representar o conjunto de todos os tipos do sistema, ou depois de caracteres, representando qualquer seqüência de caracteres. O último deve ser utilizado junto ao nome de um tipo para assim representar o conjunto de todos os seus subtipos.

A lista completa de *wildcards* e *pointcut designators* pode ser encontrada no guia de programação de AspectJ [19].

Os exemplos a seguir, mostram definições válidas de *join points* a serem usados com os designadores de *pointcut*. O primeiro exemplo, captura o construtor *default* da classe `Vector`, o seguinte captura todas as chamadas ao método `add` da classe `Vector`, note o uso do *wildcard* '..' para indicar qualquer sequência de parâmetros. Os exemplos seguintes, capturam qualquer chamada de métodos na classe `Vector` e chamadas ao método `remove` na classe `Vector` que recebe um inteiro e retorna um `Object` respectivamente.

```
1) Vector.new()
2) void Vector.add(..)
3) * Vector.*(..)
4) Object Vector.remove(int)
```

A declaração de *advices* em um aspecto especifica o código a ser executado quando o sistema for executar algum *pointcut* declarado. O *advice* indica quando o código deve ser executado através das construções `before` (antes do *pointcut*), `after` (depois do *pontcut*) e `around` (no lugar do *pointcut*).

Como podemos ver no exemplo abaixo, a linha 1 declara um *pointcut* que captura todas as chamadas a métodos da classe `Vector` feitas por métodos da classe `MyClass`. Em seguida a linha 2 declara um *advice* que executa algum código antes dos *join points* capturados pelo *pointcut* `vectorCall()`, ou seja, o código definido no *advice* será executado antes de qualquer chamada a métodos da classe `Vector`, realizada através de métodos da classe `MyClass`.

```
1: pointcut vectorCall():call(* Vector.*(..)) && this(MyClass);
2: before():vectorCall() { ... };
```

Existem casos onde o acesso a alguns parâmetros dos métodos interceptados se faz necessário, assim como a valores de retorno destes, a objetos em execução ou a objetos alvos de chamadas.

AspectJ provê uma forma de expor estes elementos para uso dentro dos *advices*, através do uso de argumentos na declaração de *pointcuts*. A linha 1 do próximo exemplo, declara um *pointcut* que captura as chamadas ao método `add(Object)` da classe `Vector` feitas através de métodos da classe `MyClass`, e expõe a instância da classe `MyClass` em execução, assim como o parâmetro do método chamado. A linha 2 ilustra a declaração de um *advice* utilizando parâmetros para acessar elementos que fazem parte do contexto da chamada capturada. Os parâmetros `c` e `o` podem ser utilizados dentro do código definido para este *advice*, representando respectivamente a instância da classe `MyClass` em execução e o objeto passado como parâmetro no método `add(Object)` da classe `Vector`.

```
1: pointcut vectorCall(MyClass c,Object o):
                        call(* Vector.add(o)) &&
                        this(c);
2: before(MyClass c, Object o):vectorCall(c,o) { ... };
```

Outras construções de AspectJ são explicadas quando utilizadas nas seções a seguir.

4 Aspectos de Concorrência

Controle de concorrência é um exemplo clássico de espalhamento de código por várias partes do sistema que acessam um recurso compartilhado, dificultando muito a manutenção e extensão do sistema. No MobileServer a necessidade de controle de concorrência aparece na Base de Dados do Servidor, um recurso compartilhado por diversos processos, que deve manter um certo nível de integridade e ordem no acesso às suas tabelas.

O controle de concorrência da Base de Dados deve garantir que, dado um processo em execução usando um conjunto de tabelas, um segundo processo poderá utilizar somente tabelas que não façam parte deste conjunto ou tabelas que já tenham sido liberadas pelo primeiro processo. A ordem de utilização das tabelas deve ser a mesma em todos os processos. Em caso de não haverem tabelas disponíveis, o processo deverá aguardar até que as tabelas necessárias sejam liberadas.

Podemos fazer uma analogia com uma linha de montagem, onde a ordem das tabelas utilizadas por todos os processos deverá ser a mesma e cada processo poderá utilizar somente as tabelas que já tenham sido liberadas pelos processos em execução há mais tempo.

Na implementação puramente em Java, este objetivo foi alcançado através do modelo mostrado na Figura 5. A interface `ConcurrencyManager` define o comportamento de uma classe que deverá controlar o início e fim de execução de cada processo, além da alocação[1] e liberação das tabelas necessárias. Todo este controle é feito baseado nas informações do usuário que está solicitando acesso à Base de Dados. A implementação do `ConcurrencyManager` faz uso da interface `RepositoryManagerAdmin` para, de fato, bloquear e liberar as tabelas.

O restante do sistema acessa a Base de Dados através da classe `RepositoryManager`. É responsabilidade do `RepositoryManager` verificar se um determinado usuário pode acessar ou não uma tabela, e levantar exceções nos casos em que a tabela solicitada não possa ser utilizada. Caso a tabela tenha sido previamente alocada para tal usuário, o `RepositoryManager` acessa a base através da classe `RepositoryFacade` (Seção 2.2) para realizar as operações solicitadas.

Com esta estrutura podemos perceber que cada trecho de um processo que necessite fazer uso da base de dados, precisará realizar uma série de chamadas ao `ConcurrencyManager` para alocar e desalocar as tabelas necessárias, além disso, cada chamada ao `RepositoryManager` necessita de um parâmetro a mais para indicar qual usuário estará acessando a base. O trecho do exemplo mostra as chamadas ao `ConcurrencyManager`, linhas 4, 5, 8, 9 e 11.

```
1:     public void process() {
2:         ConcurrencyManager manager = ConcurrencyManager.getInstance();
3:         String id = ConcurrencyManager.OUTPUT_PROCESS;
4:         manager.beginExecution(id,this.tableNames,this.user);
5:         String tableName=manager.getNextTableLock(id,this.user);
6:         while(tableName!=null) {
7:             processTable(tableName); \\Chamada de negócio
8:             manager.releaseTable(id,tableName,this.user);
9:             tableName=manager.getNextTableLock(id,this.user);
10:        }
11:        manager.endExecution(id,this.user);
12:    }
```

[1] Alocar uma tabela implica na alocação de todas as suas réplicas.

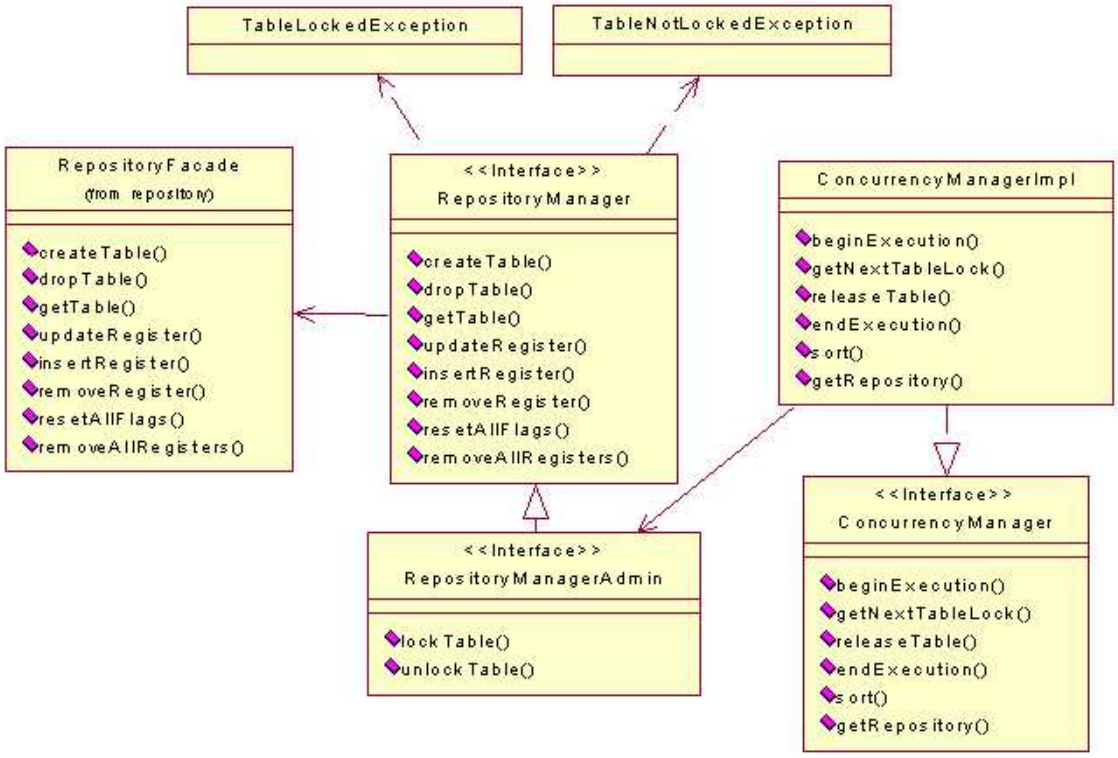

Figura 5: Modelo de classes do controle de concorrência.

O código mostrado faz parte da classe `OutputProcessorThread` e esta pode ser vista no diagrama da Figura 7.

Através do uso de AspecJ, podemos retirar todo este código não relacionado com as regras de negócio do processo em questão, e centralizarmos todo o controle de concorrência em um único local, desta forma aumentando a modularidade do sistema e ao mesmo tempo simplificando o código que realiza as regras de negócio, pois este agora poderá utilizar a Base de Dados diretamente, como se não houvessem processos concorrentes.

4.1 O ConcurrencyManager como aspecto

Uma vez que o sistema tenha sido adaptado para não mais fazer uso do controle de concorrência, toda utilização da Base de Dados é agora feita diretamente através da classe `RepositoryFacade` (Seção 2.2). Desta forma o sistema poderá funcionar perfeitamente de forma serial, ou seja, um processo após o outro.

O aspecto denominado `ConcurrencyManagerAspect` tem a responsabilidade de realizar as devidas chamadas ao `ConcurrencyManager`, agora uma classe auxiliar do aspecto, por todo o sistema, para garantir que os processos possam executar em paralelo. O diagrama da Figura 6 ilustra o aspecto e as classes por ele afetadas.

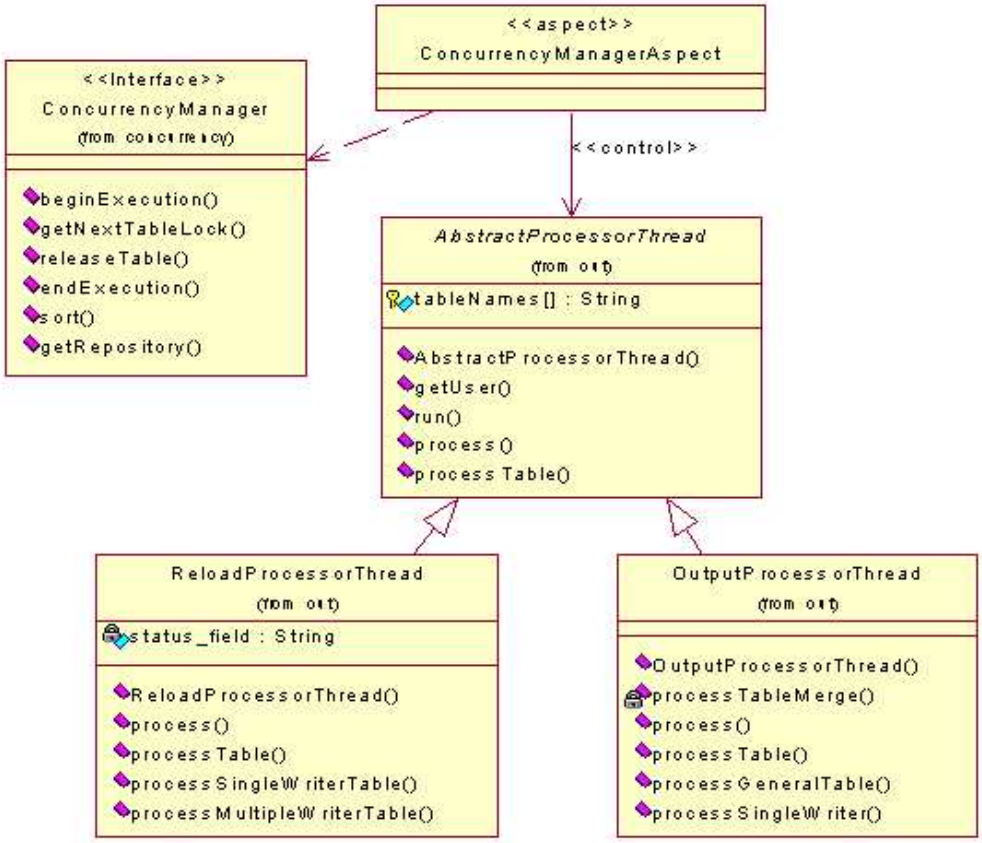

Figura 6: Modelo de classes do aspecto `ConcurrencyManagerAspect`.

A estrutura dos processos concorrentes

A Figura 7 mostra o modelo de classes de dois dos processos que utilizam a base de dados, o processamento de remessa (`OutputProcess`) e a geração de recarga (`ReloadProcess`).

Como visto na Seção 2.2, os processos de remessa e recarga são similares, sendo o segundo um caso particular do primeiro. Assim suas implementações possuem uma estrutura similar, além de terem vários métodos com implementação idêntica, o que nos sugere a utilização de uma estrutura abstrada no topo da hierarquia destes processos. Além disso, ambos processos podem ser executados paralelamente para usuários diferentes, explicando assim o fato de tal estrutura fazer uso de *threads*.

Adaptação do sistema

A adaptação do código originalmente escrito em java, envolve toda a retirada de código relacionado com o controle de concorrência, ou seja, envolve a retirada de todas as chamadas à classe `ConcurrencyManager`.

Para retirada do controle de concorrência foi necessária uma reestruturação da iteração descrita no exemplo do código com controle de concorrência, pois o laço de iteração pelas tabelas dependia diretamente da classe `ConcurrencyManager`, já que a condição da iteração

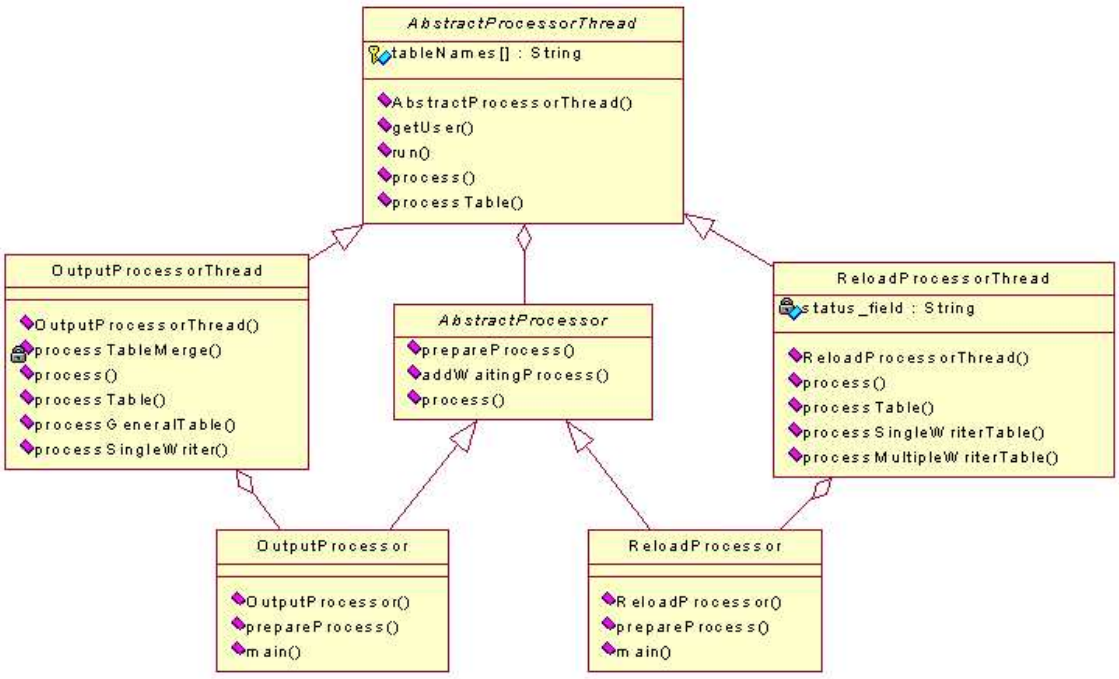

Figura 7: Modelo de classes dos processos de remessa e recarga.

utilizava o retorno de um dos métodos desta. Esta chamada era necessária para que a classe `ConcurrencyManager` controlasse a ordem de utilização das tabelas.

A nova implementação da iteração utiliza um `for` para percorrer o *array* de tabelas, ao invés do `while` original, como ilustrado no código mostrado abaixo. Com a retirada do controle de concorrência, a ordem de utilização das tabelas não é mais importante, logo o laço pode ser feito diretamente através do *array* de tabelas do processo.

```
public void process() {
    for (int i = 0; i < tableNames.length; i++) {
        processTable(tableNames[i]);
    }
}
```

Controle da ordem de utilização das tabelas

Primeiramente, devemos nos preocupar a questão da ordem de utilização das tabelas, já que o código adaptado mostrado na seção anterior não realiza mais qualquer controle na ordem de utilização das tabelas. Precisamos definir omo garantir a ordem no uso das tabelas, se cada processo cria o seu *array* de tabelas independentemente.

Este problema foi solucionado através da captura das atribuições dos *arrays* de tabelas utilizados pelos processos, de forma que estes pudessem ser reordenados antes de serem atribuídos. O *advice* mostrado a seguir realiza estas operações utilizando um método de ordenação da classe `ConcurrencyManager`. A linha 2 captura a atribuição de um *array* de `String` chamado `tableNames` da classe `AbstractProcessorThread`.

```
1: String[] around(String[] tableNames):
2:                     set(String[] AbstractProcessorThread.tableNames)&&
3:                     args(tableNames) {
4:     String[] resp = manager.sort(tableNames);
5:     proceed(resp);
6:     return tableNames;
7: }
```

A chamada ao método `proceed`, faz parte do *advice around* e indica que a execução deve continuar a partir do ponto de captura, podendo utilizar novos valores no lugar daqueles expostos pelo *pointcut*, estes novos valores são indicados pelos parâmetros passados para o método `proceed`. Neste caso, mudando o valor para o novo *array*, devidamente ordenado.

Uma vez que o *array* de tabelas seja ordenado pela classe `ConcurrencyManager` (linha 4) o problema está resolvido e funcionará corretamente para a iteração do código adaptado.

Controle do início e fim de execução dos processos

A segunda medida a ser tomada é a identificação dos pontos que necessitam do controle de concorrência, ou seja, os pontos em cada um dos processos que utilizam a base de dados. Nestes pontos devemos realizar tal controle. Olhando para o código original, puramente escrito em Java, torna-se fácil a identificação destes pontos, já que são exatamente os métodos de onde tiramos as chamadas ao `ConcurrencyManager`.

O *pointcut* definido a seguir, captura a execução dos processos de remessa e recarga, através da classe abstrata que define a estrutura básica de ambos processos, `AbstractProcessorThread`, além disso expõe a instância em execução de tal classe para uso futuro. A captura da execução dos outros processos é feita de forma similar.

```
aspect ConcurrencyManagerAspect {
    pointcut abstractProcessExecution(AbstractProcessorThread apt):
                                                target(apt) &&
                                                call(* process());
    ...
}
```

O próximo passo é a definição de *advices* que realizem corretamente as chamadas necessárias ao `ConcurrencyManager`. De acordo com o código original mostrado na seção anterior, é necessário chamar os métodos `beginExecution` e `endExecution`, da classe `ConcurrencyManager`, respectivamente antes e depois da execução de cada processo.

O método `beginExecution` tem a responsabilidade de informar ao `ConcurrencyManager` qual o conjunto de tabelas a ser utilizado pelo processo. Tal informação é necessária pois um dos papéis do `ConcurrencyManager` é garantir que todos os processos utilizem as tabelas na mema sequência, de forma a impedir um entrelaçamento das tabelas pelos processos. Voltando à analogia da linha de montagem, seria o mesmo que garantir o sentido do fluxo das etapas do processo. Já o método `endExecution` tem a finalidade de liberar qualquer tabela que por ventura tenha permanecido alocada.

```
aspect ConcurrencyManagerAspect {
    private ConcurrencyManager manager = ConcurrencyManager.getInstance();
    public String getProcessID(AbstractProcessorThread apt){ ... }
    ...
    void around(AbstractProcessorThread apt): abstractProcessExecution(apt){
        String id = getProcessID(apt);
        User user = apt.getUser();
        manager.beginExecution(id,apt.getTableNames(),user);
        proceed(apt);
        manager.endExecution(id, user);
    }
    ...
}
```

O *advice* responsável por realizar tais chamadas é mostrado acima. Como mencionado anteriormente, o `ConcurrencyManager` é agora uma classe auxiliar ao aspecto, desta forma tornou-se um atributo deste. O método `getProcessID` identifica qual o real processo em execução a partir do objeto exposto pelo *pointcut*.

Controle do início e fim da utilização de cada tabela

Voltando ao código original, vemos ainda a necessidade de mais chamadas ao `ConcurrencyManager`, mais especificamente antes e após a utilização de uma tabela em particular. Como mostrado, existe uma iteração por todas as tabelas utilizadas pelo processo. A cada passo da iteração, é feita uma chamada ao método `getNextTableLock`, tal método somente deverá retornar quando houver alguma tabela liberada para uso deste processo e seu valor de retorno será tal tabela. Ao fim de cada iteração, é feita uma chamada ao método `releaseTable`, que tem o objetivo de sinalizar a liberação da tabela para que esta possa ser utilizada por outros processos.

Como a questão da ordem de utilização das tabelas já foi resolvida, podemos prosseguir para a realização das chamadas aos métodos de alocação e liberação das mesmas. O *pointcut* definido a seguir captura os pontos de execução para cada tabela em particular, expondo, mais uma vez, a instância do processo em execução, assim como o nome da tabela utilizada. O método `processTable(String)` da classe `AbstractProcessorThread` indica o processamento de uma tabela.

```
pointcut abstractProcessTable(AbstractProcessorThread apt, String table):
                                target(apt) &&
                                args(table) &&
                                call(* processTable(String));
```

Devemos então definir o *advice* que fará uso deste *pointcut* para realizar as chamadas de alocação e liberação da tabela, tal *advice* é mostrado a seguir.

```
boolean around(AbstractProcessorThread apt, String table):
                                        abstractProcessTable(apt,table){
    String id = getProcessID(apt);
    User user = apt.getUser();
    manager.getNextTableLock(id,user);
    boolean ret = proceed(apt,tableName);
    manager.releaseTable(id,tableName,user);
    return ret;
}
```

4.2 O RepositoryManager como aspecto

A Seção 4.1 mostrou a separação de todo código relacionado à classe `ConcurrencyManager`, porém ainda existe código relacionado com o controle de concorrência espalhado pelo sistema.

Este código encontra-se em todo e qualquer acesso à base de dados, feito através da classe `RepositoryManager` (Seção 4). A classe `RepositoryManager` acessa a base de dados através da classe `RepositoryFacade`, descrita na Seção 2.2. O trecho de código a seguir faz parte da classe `OutputProcessorThread` e faz uso da classe `RepositoryManager`.

```
private User user;
public boolean processTable(String tableName) {
    List userTables = null;
    RepositoryManager rep = SystemProperties.getRepositoryManager();
    userTables = rep.getTable(tableName,user);
    ...
}
```

A classe `RepositoryManager` possui exatamente os mesmos métodos da classe `RepositoryFacade`, com uma única diferença, cada método recebe um parâmetro a mais, informando qual o usuário solicitando acesso à base. Através deste parâmetro é feita a verificação se a tabela foi previamente alocada pelo `ConcurrencyManager` para tal usuário. Caso a tabela não esteja devidamente alocada, é levantada uma exceção indicando este fato, caso contrário, a base de dados é acessada normalmente.

Mais uma vez, torna-se necessária a adaptação do sistema para que este deixe de utilizar o `RepositoryManager` e passe a utilizar diretamente a classe `RepositoryFacade`, eliminando assim o restante do código relacionado com o controle de concorrência.

```
public boolean processTable(String tableName) {
    List userTables = null;
    RepositoryFacade rep = SystemProperties.getRepositoryFacade();
    userTables = rep.getTable(tableName);
    ...
}
```

Uma vez adaptado o sistema, devemos nos preocupar em implementar um aspecto que controle o acesso à base de dados dependendo de cada usuário. O aspecto aqui implementado foi denominado `RepositoryManagerAspect` e, assim como ocorrido com o `ConcurrencyManager`, a classe `RepositoryManager` passa a ser uma classe auxiliar ao aspecto, como pode ser visto no diagrama da Figura 8.

No caso deste aspecto, a identificação dos pontos de execução que deve ser interceptados é mais intuitiva e simples. Devemos capturar toda e qualquer chamada à classe `RepositoryFacade` e, antes da execução de seus métodos, verificar se o usuário em execução pode executar esta operação.

No entanto surge um novo problema, pois necessitamos saber qual o usuário atualmente em execução. Felizmente todos os processos que acessam a base de dados, possuem um atributo indicando tal usuário. Assim, devemos procurar uma forma de caracterizar estes processos de maneira mais formal. Esta caracterização pode ser alcançada através das construções de AspectJ que alteram a estrutura estática do programa, de forma a indicar que todos estes processos implementam uma interface em comum, interface esta que possui um único método que deverá expor o usuário em execução. O nome dado à ela foi `UserProcess` e a indição da implementação é ilustrada abaixo.

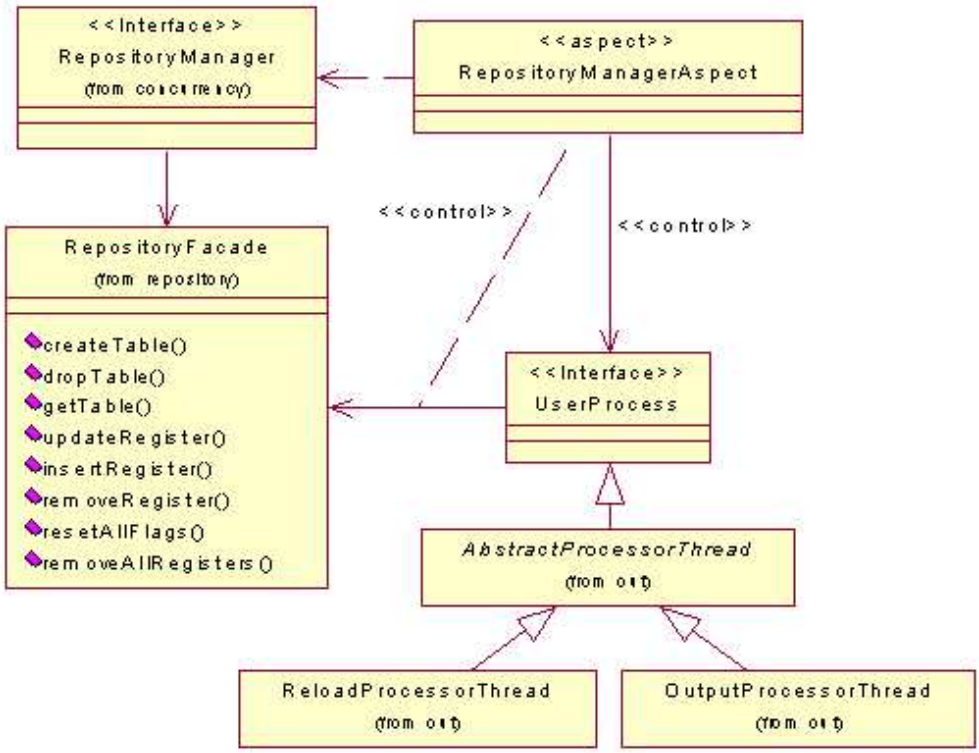

Figura 8: Modelo de classes do aspecto `RepositoryManagerAspect`..

```
public interface UserProcess {
    public User getUser();
}
declare parents: AbstractProcessorThread implements UserProcess;
```

Agora podemos definir todos os pontos de captura deste aspecto, como sendo toda e qualquer chamada à classe `RepositoryFacade`, feita por um objeto do tipo `UserProcess`. Note que a única classe que pode utilizar diretamente a classe `RepositoryFacade` será a própria classe que implementa a interface `RepositoryManager`. Por esta razão existe a limitação imposta pelo designador `withincode` o qual garante que o código capturado estará dentro da classe `RepositoryManager`. O operador + indica qualquer classe abaixo desta na hierarquia.

```
pointcut repositoryCall(UserProcess o):
                        call(* RepositoryFacade.*(..)) &&
                        this(o) &&
                        withincode (* RepositoryManager+.*(..));
```

Como cada método capturado possui uma assinatura diferente e deverá ser acrescido de um parâmetro, o usuário, foi necessário a definição de um *pointicut* para cada método, onde estes *pointcuts* expõem o objeto `UserProcess` em execução assim como os parâmetros de cada método. Um destes *pointcuts* é mostrado a seguir.

```
pointcut getTable(UserProcess o, String table):
                  repositoryCall(o) &&
                  call(List RepositoryFacade.getTable(String)) &&
                  args(table);
```

O *advice* utilizado com o *pointcut* acima é definido abaixo e deverá redirecionar a chamda do método interceptado para o método de mesmo nome na classe `RepositoryManager`, acrescentando o usuário em execução no momento.

```
List around(UserProcess o, String table) throws RepositoryException:
                                         getTable(o,table) {
    return repository.getTable(table,o.getUser());
}
```

A declaração dos demais *pointcuts* e *advices* é similar, mudando somente o método capturado e seus parâmetros. Por este motivo omitimos os detalhes aqui (ver Apêndice A).

4.2.1 Controle de Exceções

Uma outra diferença entre os métodos da classe `RepositoryManager` e da classe `RepositoryFacade` é que os métodos da primeira levantam novas exceções não definidas pela segunda. Sendo assim, devemos fazer o controle destas exceções internamente ao aspecto, garantido que somente as exceções já previstas possam continuar propagando-se. As novas exceções são `TableLockedExceptio` e `TableNotLockedException`, e a exceção já existente é `RepositoryException`.

AspectJ provê um mecanismo para transformar exceções checadas em exceções não checadas como visto na Seção 3. O controle de exceções é então feito de forma a transformar as novas exceções em exceções não checadas, e além disso capturá-las e realizar o tratamento necessário.

O trecho de código a seguir mostra um *pointcut* que captura todos os métodos que necessitam do tratamento de exceções na linha 1. Em seguida as linhas 2 e 3 declaram as novas exceções como não checadas e por fim os *advices* das linhas 4 e 5 fazem o tratamento necessário com as exceções, encapsulando por completo o controle das novas exceções.

```
1: pointcut exceptionHandler(): call(* RepositoryManager.*(..));
2: declare soft: TableLockedException: exceptionHandler();
3: declare soft: TableNotLockedException: exceptionHandler();
4: after() throwing(TableLockedException ex) throws RepositoryException :
                                             exceptionHandler() {
       throw new RepositoryException(ex);
   }
5: after() throwing(TableNotLockedException ex) throws RepositoryException :
                                                exceptionHandler() {
       throw new RepositoryException(ex);
   }
```

4.3 Análise Comparativa

Como mostrado, o código de negócio sem a preocupação com o controle de concorrência tornou-se mais limpo, simples e legível, aumentando a manutenabilidade do sistema. Além disto, a utilização de AspectJ tornou o sistema mais modular, já que todo o controle de concorrência, antes espalhado por diversas classes, agora está centralizado no aspecto.

Com a utilização de AspectJ temos agora a opção de utilizar ou não o controle de concorrência, isto pode ser feito pela simples retirada dos aspectos de concorrência durante o processo de compilação. Este fato nos dá uma maior flexibilidade, que implica em um maior poder de customização, deixando que o sistema possa trabalhar sem o controle de concorrência para melhorar a performance em situações onde o sistema pode funcionar mais eficientemente de forma serial, em um ambiente com poucos usuários por exemplo.

Outra vantagem agregada pelo uso de AspectJ foi uma melhora na extensibilidade do sistema. Esta decorre do fato de que o controle capturado pelo aspecto é baseado em uma classe abstrata. Caso haja necessidade de implementação de um novo processo que utilize a base de dados, basta que este processo herde de tal classe, além de realizar uma pequena alteração no método `getProcessID` para que este reconheça a nova classe. Mais uma vantagem foi a redução do número de linhas de código escritas, devido ao fato do aspecto controlar mais de um processo[2] com um único *pointcut*. Tal redução no código aumenta proporcionalente com o número de pontos capturados por *pointcut*.

A adaptação do sistema escrito puramente com Java, consumiu uma fração considerável (cerca de 40%) do tempo total de desenvolvimento deste aspecto, o que nos indica um problema, já que um custo tão alto para adaptação do sistema poderá tornar a utilização de AOP desinteressante para a reengenharia de sistemas existentes. A existência de ferramentas de refatoramento poderia reduzir este custo.

Por outro lado, caso o sistema houvesse sido planejado para a utilização de AspectJ, o custo de implementação seria praticamente o mesmo e o sistema ainda assim ganharia bastante em termos de qualidade de software, como descrito acima.

Outro problema que decorre do fato do sistema não haver sido planejado para o uso de AspectJ, é que não houve uniformidade na estrutura dos processos que utilizam a base de dados. Esta falta de uniformidade dificultou a implementação de um aspecto simples como o aspecto descrito na Seção 4.1. A criação de outros *pointcuts* e *advices*, similares aos descritos, foi necessária para outros dois processos que fazem uso da base de dados, tornando o código um pouco repetitivo, indicando a necessidade de padrões de AspectJ e ferramentas de geração de código.

Um outro ponto negativo encontra-se no processo de depuração quando da utilização de AspectJ. A depuração do programa utilizando AspectJ é difícil em decorrência do fato que o *bytecode* gerado pelo compilador do AspectJ não reflete o código fonte escrito, isto acontece por conta do processo de *weaving*(Seção 3).

[2]Todos os processos derivados da classe abstrata

5 Aspectos de Persistência

Nesta seção apresentamos uma utilização simples de AOP para configurar a forma de persistência da base de dados do Servidor. Além desta aplicação, AspectJ foi utilizado para modularizar uma parte funcional do sistema, centralizando toda a atualização da base de dados em um único local.

5.1 Configuração

A forma como a base de dados foi modelada [2] já trazia muitas vantagens do ponto de vista de customização, de forma que já era possível configurar o sistema para utilizar uma estrutura de dados persistente ou volátil, sem que fossem necessárias mudanças no código, e sim uma simples mudança em arquivos de configuração. A estrutura modelada pode ser vista na Figura 3 (pág. 6).

Tal estrutura é caracterizada pela existência de uma interface, `Repository`, que define todas as operações necessárias sobre a coleção de dados, e de uma classe de negócio, `RepositoryFacade`, que faz uso desta interface para armazenar dados. Desta forma, através do recurso de reflexão da linguagem Java, pode-se instanciar, em tempo de execução, uma classe que implemente esta interface, podendo esta armazenar as tabelas de forma volátil ou implementar uma forma de persistência diferente, utilizando, por exemplo, um banco de dados ou arquivos.

Por estes motivos, a implementação do aspecto de configuração teve o objetivo de separar, e colocar em um local devido, a inicialização da classe a ser utilizada como coleção de dados. O primeiro passo neste sentido é descobrir qual o momento apropriado para que tal inicialização seja feita. Como esperado, o momento mais apropriado é na construção da classe `RepositoryFacade`. O trecho de código a seguir ilustra o *advice* que captura a construção da classe `RepositoryFacade`. No exemplo, um atributo do aspecto, instância da classe `Repository`, é passado para a classe `RepositoryFacade`.

```
aspect RepositoryAspect {
    private Repository repositoryImplementation;
    after() returning (RepositoryFacade fac): call(RepositoryFacade.new(..)) {
        fac.setRepository(repositoryImplementation);
    }
}
```

5.2 Atualização da Base de Dados

O próximo aspecto discutido tem o objetivo de centralizar toda atualização da base de dados em um único local, na tentativa de deixar o código mais legível, simples e modular. Sendo assim, nosso objetivo é o de retirar todas as chamadas à classe `RepositoryFacade` que atualizam a base de dados. Estas chamadas são indicadas pelos métodos `removeRegister`, `updateRegister` e `insertRegister` como no exemplo a seguir, retirado da classe `OutputProcessorThread`.

```
private void updateRepository(Table[] differences) throws RepositoryException {
    RepositoryFacade fac = SystemProperties.getRepositoryFacade();
    for (int i = 0; i < differences.length; i++) {
        Iterator iter = differences[i].iterator();
        while (iter.hasNext()) {
            Register item = (Register)iter.next();
            char status = item.getField("STATUS").getValue().getChar();
            if (status==Register.STATUS_DELETED) {
                fac.removeRegister( ... );
            } else if (status==Register.STATUS_INSERTED) {
                fac.insertRegister( ... );
            } else {
                fac.updateRegister( ... );
            }
        }
        fac.removeAllRegisters( ... );
        fac.resetAllFlags( ... );
    }
}
```

Neste momento surge um novo problema: estes métodos já são interceptados pelo aspecto definido na Seção 4.2, ou seja, a mudança de local destes métodos implica em uma reestruturação do aspecto de concorrência. Mais uma vez este problema poderia ter sido evitado caso o sistema tivesse sido planejado para uso de AspectJ, de forma que sua arquitetura tornasse possível uma independência maior entre os aspectos. A Figura 9 mostra o modelo de classes da implementação dos aspectos de atualização da base.

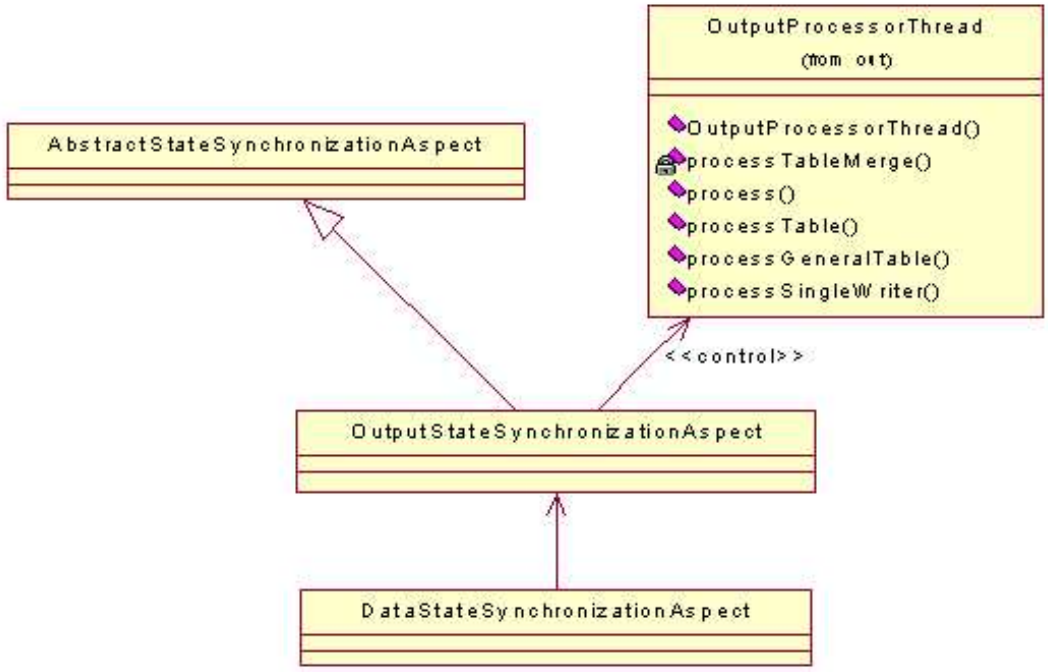

Figura 9: Modelo de classes dos aspectos de atualização da base de dados.

Como o aspecto visto na Seção 4.2 baseia-se em processos que conheçem o usuário em execução, precisemos fazer com que nosso aspecto de atualização possua informação a respeito deste usuário. Uma nova construção de AspectJ foi utilizada para associar uma instância do aspecto de atualização com cada usuário em execução. Esta construção, chamada `pertarget`, aparece na linha 3 do código a seguir e indica que o aspecto declarado não será um *singleton*[3] como é o caso normal, mas existirá uma instância para cada processo indicado pelos *join points* definido nas linhas 3 e 4.

```
1: public aspect OutputStateSynchronizationAspect
2:                   extends AbstractStateSynchronizationAspect
3:                   pertarget (execution(* OutputProcessorThread.*(..))||
4:                              execution(OutputProcessorThread.new(..)))
```

Um aspecto auxiliar foi necessário para capturar a execução de cada processo e associar o usuário deste com a instância do aspecto relacionada ao processo, esta associação pode ser vista a seguir. O método `aspectOf`(...) retorna a instância do aspecto associada com o parâmetro passado, de forma que no código abaixo, retorna a instância do aspecto `OutputStateSynchronizationAspect` associada com o processo em execução.

```
aspect DataStateSynchronizationAspect {
  after() returning (OutputProcessorThread p):
                    call(OutputProcessorThread.new(..)) {
    OutputStateSynchronizationAspect.aspectOf(p).setUser(process.getUser());
  }
  ...
}
```

No Servidor existem somente dois processos que realizam alguma atualização na base de dados. São eles o processamento de remessa e o processamento de retorno. Ambos possuem um método `updateRepository` que recebe como parâmetro um *array* de tabelas e realiza a atualização da base. No entanto, o *array* recebido no processamento de remessa já contém toda a informação necessária para que a base seja atualizada, enquanto que alguma informação extra deve ser fornecida para a atualização no processo de retorno. O processamento de remessa possui uma outra diferença, existem tipos diferentes de tabela e para cada tipo, o processamento é diferente e assim existe um método para cada tipo de processamento. O método utilizado no exemplo é o método `processTableMerge`. A implementação para os outros métodos e para o processamento de retorno é similar.

A implementação deste aspecto consiste em mover o método `updateRepository` para dentro do aspecto e capturar os pontos onde este método deve ser chamado. A captura de um dos pontos de execução para o processamento de remessa pode ser vista nos *pointcuts* definidos abaixo.

```
pointcut outputExecution():
             cflow(execution(void OutputProcessorThread.process()));
pointcut updateMerge():
             outputExecution() &&
             call(Table[] OutputProcessorThread.processTableMerge(..));
```

Vale salientar que foi necessário um refatoramento na classe que realiza o processamento de remessa. Esta adaptação foi necessária, pois o *array* de tabelas a ser passado como parâmetro para o método `update` era gerado no corpo de um outro método, como mostrado a seguir.

[3]Padrão de projeto no qual uma classe possui uma única instância.

```
private void processTableMerge(String tableName, Table[] userTables) {
    if (userTables.length>1) {
        Table[] differences = merger.merge(Arrays.asList(userTables));
        ...
        updateRepository(differences);
    } else {
        ...
    }
}
```

Como não existe uma construção de AspectJ que capture uma variável local a um método, foi feita uma extração do trecho de código que gera o *array* de tabelas, criando um novo método que retorna o *array* de forma que este possa ser interceptado.

```
private Table[] processTableMerge(String tableName, Table[] userTables) {
    Table[] differences = new Table[0];
    if (userTables.length>1) {
        differences = merger.merge(Arrays.asList(userTables));
        ...
        updateRepository(differences);
    } else {
        ...
    }
    return differences;
}
```

Por fim o *advice* mostrado em seguida realiza a chamada ao método `updateRepository` passando como argumento o retorno capturado do método `processTableMerge`.

```
after() returning (Table[] differences) throws RepositoryException:
                                        updateMerge() {
    updateRepository(differences);
}
```

5.3 Análise Comparativa

No caso dos aspectos de persistência como foram tratados nas Seções 5.1 e 5.2, não houve nenhum ganho considerável em termos de qualidade do software.

A solução utilizada para o problema de configuração nos mostra uma forma limpa e modular para resolver o problema com AspectJ, porém o mesmo grau de customização alcançado pode ser obtido com o programa escrito puramente em Java.

Para o problema de atualização da base de dados, podemos notar que não houve uma melhoria notável no sistema, pois os aspectos implementados, como constituem uma parte funcional do sistema, sempre deverão existir juntamente com o código, ou seja, o programa não pode funcinar corretamente sem o aspecto de atualização.

Outro ponto a ser analizado é a quantidade de código escrita. Mais uma vez não obtivemos melhoria. O código incluindo os aspectos é maior do que o código original escrito em Java.

Apesar do código ter ficado um pouco mais organizado e termos isolado toda a parte de atualização da base de dados dentro do aspecto, isto pode levar a uma dificuldade de compreensão do sistema para alguém que venha a mantê-lo no futuro, já que este deverá entender não somente o código que realiza as regras de negócio da base, mas também o aspecto que realiza a atualização da mesma.

Em resumo, não existe uma razão que justifique a implementação das funcionalidades de configuração e atualização das bases como aspectos, já que esta implementação não agrega nenhuma vantagem considerável.

6 Aspectos de Logging

Geralmente o controle de *log* de um sistema qualquer tem por finalidade facilitar a depuração do programa. No Servidor, a informação de *log* é utilizada para geração de relatórios sobre a utilização do sistema. Desta forma as chamadas do *log* devem ser bem localizadas para atenderem a todas as demandas dos relatórios como mostrado a seguir.

```
public void process() {
    if ( ...) {
        Log.getInstance().log(Log.ACTION_PROCESSAMENTO_REMESSA, ...
                              Log.STATUS_INIT);
        boolean error=false;
        ...
        if (!error) {
            Log.getInstance().log(Log.ACTION_PROCESSAMENTO_REMESSA, ...
                                  Log.STATUS_OK);
        } else {
            Log.getInstance().log(Log.ACTION_PROCESSAMENTO_REMESSA, ...
                                  Log.STATUS_ERROR, errorMessage);
        }
    } else {
        ...
    }
}
```

Como podemos perceber, o código de *log* encontra-se espalhado por todo o sistema, inserindo chamadas que não estão relacionadas com o código no qual estão inseridas, ou seja, não são necessárias para o funcionamento correto do mesmo.

Sendo assim, este controle das chamadas ao *log* é um caso bem indicado para o uso de AOP para a modularização e separação de código não relacionado.

Infelizmente, no Servidor, os pontos nos quais encontram-se as chamadas ao *log* não são bem definidos. Este pontos encontram-se normalmente no meio do corpo de métodos, como já mostrado, e estes não possuem qualquer uniformidade ou característica em comum.

Para que a implementação do *log* como aspecto seja possível, será necessário um refatoramaneto de todos os métodos que realizam chamadas ao *log*, no sentido de criar novos métodos a partir de trechos de outros métodos, de forma que as chamadas ao *log* ficassem bem definidas no começo e fim de um método, como mostrado em seguida.

```
public void process() {
    if ( ...) {
        realProcess();
    } else {
        ...
    }
}
```

```
private boolean realProcess() {
    Log.getInstance().log(Log.ACTION_PROCESSAMENTO_REMESSA, ...
                          Log.STATUS_INIT);
    boolean error=false;
    ...
    if (!error) {
        Log.getInstance().log(Log.ACTION_PROCESSAMENTO_REMESSA, ...
                              Log.STATUS_OK);
    } else {
        Log.getInstance().log(Log.ACTION_PROCESSAMENTO_REMESSA, ...
                              Log.STATUS_ERROR, errorMessage);
    }
    return error;
}
```

Uma vez que esta adaptação fosse feita, seria possível criar um aspecto que interceptasse todos os pontos necessários e inserisse as devidas chamadas ao *log*.

6.1 Análise Comparativa

Devido ao grande volume de alterações necessárias para a implementação do *log* como aspecto e ao curto tempo disponível para realização deste trabalho, a implementação do aspecto de *log* não foi realizada.

Vale notar que se o sistema houvesse sido planejado para uso de AOP e a estruturação dos métodos já definisse claramente os pontos onde as chamadas ao log deveriam ser inseridas, a implementação do *log* como aspecto seria simples e rápida, agregando ao sistema algumas das vantagens mencionadas (Seção 1) pelo uso de AOP.

Algumas destas vantagens seriam a modularização e separação de código não relacionado, já que todo o *log* estaria dentro do aspecto. Outra vantagem agregada seria do ponto de vista de customização, pois poderíamos utilizar ou não o *log*. Haveira ainda uma melhoria na manutenabilidade do sistema, já que o código sem as chamadas ao *log* ficaria mais legível, assim como a necessidade de novas chamadas ao *log* implicaria em mudanças no aspecto somente.

7 Conclusões

A utilização de AOP através da linguagem AspectJ atendeu a nossas espectativas, apesar de alguns pontos negativos terem sido encontrados durante o desenvolvimento deste trabalho. Pudemos comprovar boas melhorias no sistema agregadas pelo uso deste paradigma.

Primeiramente, pudemos ver um aumento razoável na moduladiade do sistema, como discutido e mostrado na Seção 4.3. A separação de código que não faz parte do contexto onde está iserido é real e simples de ser implementada.

Muito embora o resultado obtido na implementação do aspecto de concorrência tenha sido satisfatório, este poderia ser ainda melhor caso a arquitetura do sistema fosse desenhada já pensado na aplicação de AOP, de forma que não fosse gasto tempo para adaptação do sistema. Isto tornaria a implementação do aspecto de *log* mais fácil, pois o mesmo não foi implementado por conta de uma estimativa de alto custo para a alteração do sistema (Seção 6.1).

Outra qualidade melhorada foi a manutenabilidade. Esta melhoria deve-se ao fato de que o código sem as preocupações separadas pelos aspectos, tornou-se mais simples e legível, assim como o código, antes espalhado por diversas classes, encontra-se agora contralizado em um único ponto.

Infelizmente estas vantagens não foram obtidas para o aspecto de persistência discutido na Seção 5. Conforme mostrado, a forma como o sistema foi concebido e implementado já estava bem modular e com um bom poder de customização. A tentativa de separar uma preocupação funcional, como foi o caso do aspecto visto na Seção 5.2, não trouxe nenhum benefício pois este aspecto sempre deve existir para o correto funcionamento do sistema, assim como tal separação dificulta a compreensão do código por separar conceitos relacionados.

Estes problemas de implementação de aspectos de persistência não ocorreu em outros trabalhos [18, 13], devido ao sistema utilizado como caso de uso, possuir características diferentes como a necessidade de sincronização de estados, pois objetos criados pelo repositório podem ser alterados em outro lugar e estas alterações devem ser refletidas no repositório.

A facilidade de customização do sistema foi melhorada com o aspecto de concorrência de forma que o sistema poderia agora, funcionar sem tal controle em um ambiente sem processos executando em paralelo, ou com processos em paralelo utilizando o aspecto de concorrência. Esta mesma melhoria poderia ser vista no aspecto de *log*, já que o sistema poderia ser executado utilizando ou não a parte de *log*.

A extensibilidade do sistema também obteve melhorias. Como visto na Seção 4.3, um novo processo que faça uso da base de dados precisaria somente herdar da classe abstrata utilizada pelo aspecto de concorrência para estar totalmente dentro do esquema do controle de concorrência. Em decorrência deste fato, pudemos também perceber uma redução no tamanho do código escrito em Java, em comparação com o código coberto pelo aspecto.

Uma dificuldade encontrada na implementação de todos os aspectos foi no processo de depuração do sistema. A depuração do sistema é complicada dado o fato de que o *bytecode* gerado pelo compilador do AspectJ não reflete a estrutura do código fonte original. Desta forma fica difícil identificar o ponto de ocorrência de uma exceção. Esta diferença entre o *bytecode* e o código fonte ocorre por causa do processo de *weaving* comentado na Seção 3

Para finalizar, reiteramos que um cuidado na estruturação da aplicação, já pensando em uma implementação usando AOP, pode facilitar consideravelmente o desenvonvimento, juntamente com a melhoria do software como um todo, já que uma parte maior da aplicação poderá fazer uso das vantagens providas pela orientação a aspectos.

A Código fonte dos Aspectos

A.1 ConcurrencyManagerAspect

```
package br.com.mobile.ms.concurrency;

import br.com.mobile.ms.business.User;
import br.com.mobile.ms.data.out.AbstractProcessorThread;
import br.com.mobile.ms.data.out.OutputProcessorThread;
import br.com.mobile.ms.data.out.ReloadProcessorThread;
import br.com.mobile.ms.data.in.InputProcessorThread;
import br.com.mobile.ms.gui.web.handler.SetupTableHandler;
import br.com.mobile.ms.util.SystemProperties;
import br.com.mobile.ms.util.log.Log;

privileged aspect ConcurrencyManagerAspect {
    private static ConcurrencyManager manager;

    public static ConcurrencyManager getConcurrencyManager() {
        return manager;
    }

    public ConcurrencyManagerAspect() {
        if (manager==null) {
            try {
                String className = SystemProperties.getSystemParameters()
                                   .getParam("CONCURRENCY_MANAGER_CLASS_NAME");
                manager = (ConcurrencyManager)Class.forName(className)
                                                   .newInstance();
            }catch (Exception ex) {
                Log.getInstance().logException("ERRO: Na criacao do "+
                                               "repositorio",ex);
            }
        }
    }

    public String[] AbstractProcessorThread.getTableNames() {
        return this.tableNames;
    }

    public String getProcessID(AbstractProcessorThread apt) {
        String resp="";
        if (apt instanceof OutputProcessorThread) {
            resp = ConcurrencyManager.OUTPUT_PROCESS;
        } else if (apt instanceof OutputProcessorThread) {
            resp = ConcurrencyManager.RELOAD_PROCESS;
        }
        return resp;
    }

    pointcut adminExecution(SetupTableHandler sth, String tableName):
                                               target(sth) &&
                                               args(tableName) &&
                                               call(* removeTableInfos(String));

    pointcut abstractProcessExecution(AbstractProcessorThread apt):
                                               target(apt) &&
                                               call(* process());

    pointcut abstractProcessTable(AbstractProcessorThread apt,String tableName):
                                               target(apt) &&
                                               args(tableName) &&
                                               call(* processTable(String));

    pointcut inputProcessExecution(InputProcessorThread ipt,
                                   String[] tableNames):
                                               target(ipt) &&
                                               args(tableNames, *) &&
                                               call(* process(String[], *));
```

```
    pointcut inputProcessTable(InputProcessorThread ipt, String tableName):
                                                target(ipt) &&
                                                args(tableName, *) &&
                                                call(* process(String, *));

    String[] around(String[] tableNames):
                    set(String[] AbstractProcessorThread.tableNames) &&
                    args(tableNames) {
        String[] resp = manager.sort(tableNames);
        proceed(resp);
        return tableNames;
    }

    void around(SetupTableHandler sth, String tableName)
                                                :adminExecution(sth,tableName) {
        manager.beginExecution(ConcurrencyManager.ADMIN_PROCESS,
                               new String[] {tableName},
                               sth.getUser());
        manager.getNextTableLock(ConcurrencyManager.ADMIN_PROCESS,
                                 sth.getUser());
        proceed(sth,tableName);
        manager.releaseTable(ConcurrencyManager.ADMIN_PROCESS,
                             tableName,sth.getUser());
        manager.endExecution(ConcurrencyManager.ADMIN_PROCESS, sth.getUser());
    }

    void around(AbstractProcessorThread apt): abstractProcessExecution(apt){
        manager.beginExecution(getProcessID(apt),
                               apt.getTableNames(), apt.getUser());
        proceed(apt);
        manager.endExecution(getProcessID(apt), apt.getUser());
    }

    boolean around(AbstractProcessorThread apt, String tableName)
                                          :abstractProcessTable(apt,tableName){
        manager.getNextTableLock(getProcessID(apt),apt.getUser());
        boolean ret = proceed(apt,tableName);
        manager.releaseTable(getProcessID(apt),tableName,apt.getUser());
        return ret;
    }

    void around(InputProcessorThread ipt, String[] tableNames)
                                        :inputProcessExecution(ipt,tableNames) {
        tableNames = manager.sort(tableNames);
        manager.beginExecution(ConcurrencyManager.INPUT_PROCESS,
                               tableNames, ipt.getUser());
        proceed(ipt, tableNames);
        manager.endExecution(ConcurrencyManager.INPUT_PROCESS, ipt.getUser());
    }

    void around(InputProcessorThread ipt, String tableName)
                                          :inputProcessTable(ipt, tableName) {
        manager.getNextTableLock(ConcurrencyManager.INPUT_PROCESS,
                                 ipt.getUser());
        proceed(ipt,tableName);
        manager.releaseTable(ConcurrencyManager.INPUT_PROCESS,
                             tableName,ipt.getUser());
    }
}
```

A.2 RepositoryManagerAspect

```
package br.com.mobile.ms.concurrency;

import java.util.List;

import br.com.mobile.ms.business.User;
import br.com.mobile.ms.business.Medium;
import br.com.mobile.ms.repository.RepositoryFacade;
import br.com.mobile.ms.repository.RepositoryException;
import br.com.mobile.ms.repository.Table;
import br.com.mobile.ms.repository.Register;
import br.com.mobile.ms.repository.PrimaryKey;
import br.com.mobile.ms.repository.aspects.AbstractStateSynchronizationAspect;
import br.com.mobile.ms.util.SystemProperties;
import br.com.mobile.ms.data.out.OutputProcessorThread;
import br.com.mobile.ms.data.out.ReloadProcessorThread;
import br.com.mobile.ms.data.in.InputProcessorThread;
import br.com.mobile.ms.gui.web.handler.SetupTableHandler;

public aspect RepositoryManagerAspect {
    private RepositoryManager repository;

    declare parents: AbstractProcessorThread implements UserProcess;
    declare parents: InputProcessorThread implements UserProcess;
    declare parents: SetupTableHandler implements UserProcess;
    declare parents: AbstractStateSynchronizationAspect implements UserProcess;
    declare soft: TableLockedException: exceptionHandler();
    declare soft: TableNotLockedException: exceptionHandler();

    public RepositoryManagerAspect() {
        repository = ConcurrencyManagerAspect.getConcurrencyManager()
                                             .getRepository();
    }

    public interface UserProcess {
        public User getUser();
    }

    pointcut exceptionHandler(): call(* RepositoryManager.*(..));

    pointcut repositoryCall(UserProcess o): call(* RepositoryFacade.*(..)) &&
                                        this(o) &&
                                       !withincode (* RepositoryManager+.*(..));

    pointcut createTable(UserProcess o, Table table):
                               repositoryCall(o) &&
                               call(void RepositoryFacade.createTable(Table)) &&
                               args(table);

    pointcut dropTable(UserProcess o, String table):
                               repositoryCall(o) &&
                               call(void RepositoryFacade.dropTable(String)) &&
                               args(table);

    pointcut truncateTable(UserProcess o, String table) :
                             repositoryCall(o) &&
                             call(void RepositoryFacade.truncateTable(String)) &&
                             args(table);


    pointcut getTable(UserProcess o, String table):
                                repositoryCall(o) &&
                                call(List RepositoryFacade.getTable(String)) &&
                                args(table);


    pointcut getTableUser(UserProcess o, String table, User user):
                           repositoryCall(o) &&
                           call(List RepositoryFacade.getTable(String, User)) &&
                           args(table, user);
```

```
pointcut getTableMedium(UserProcess o, String table, Medium medium):
                    repositoryCall(o) &&
                    call(Table RepositoryFacade.getTable(String, Medium)) &&
                    args(table, medium);


pointcut getTableMediumUser(UserProcess o,String table,
                            Medium medium, User user):
               repositoryCall(o) &&
               call(Table RepositoryFacade.getTable(String, Medium, User)) &&
               args(table, medium, user);

pointcut updateRegister(UserProcess o,String table, Medium medium,
                        Register register, PrimaryKey pk):
                    repositoryCall(o) &&
                    call(void RepositoryFacade.updateRegister(String,
                                                              Medium,
                                                              Register,
                                                              PrimaryKey)) &&
                    args(table, medium, register, pk);


pointcut insertRegister(UserProcess o,String table,
                        Medium medium, Register register):
                      repositoryCall(o) &&
                      call(void RepositoryFacade.insertRegister(String,
                                                                Medium,
                                                                Register)) &&
                      args(table, medium, register);

pointcut removeRegister(UserProcess o,String table,
                        Medium medium, Register register):
                      repositoryCall(o) &&
                      call(void RepositoryFacade.removeRegister(String,
                                                                Medium,
                                                                Register)) &&
                      args(table, medium, register);

pointcut resetAllFlags(UserProcess o,String table, Medium medium):
                        repositoryCall(o) &&
                        call(void RepositoryFacade.resetAllFlags(String,
                                                                 Medium)) &&
                        args(table, medium);

pointcut removeAllRegisters(UserProcess o,String table, Medium medium):
                    repositoryCall(o) &&
                    call(void RepositoryFacade.removeAllRegisters(String,
                                                                  Medium)) &&
                    args(table, medium);

pointcut optimize(UserProcess o) : repositoryCall(o) &&
                                   call(void RepositoryFacade.optimize());


void around(UserProcess o, Table table)throws RepositoryException
                                      :createTable(o,table) {
    repository.createTable(table);
}

void around(UserProcess o, String table)throws RepositoryException
                                       :dropTable(o,table) {
    repository.dropTable(table,o.getUser());
}

void around(UserProcess o, String table)throws RepositoryException
                                       :truncateTable(o,table) {
    repository.truncateTable(table,o.getUser());
}
```

```
    List around(UserProcess o, String table)throws RepositoryException
                                            :getTable(o,table) {
        return repository.getTable(table,o.getUser());
    }

    List around(UserProcess o, String table, User user)
                        throws RepositoryException
                        :getTableUser(o,table, user) {
        return repository.getTable(table,user,o.getUser());
    }

    Table around(UserProcess o, String table, Medium medium)
                        throws RepositoryException
                        :getTableMedium(o,table,medium) {
        return repository.getTable(table,medium,o.getUser());
    }

    Table around(UserProcess o, String table, Medium medium, User user)
                        throws RepositoryException
                        :getTableMediumUser(o,table,medium,user) {
        return repository.getTable(table,medium,user,o.getUser());
    }

    void around(UserProcess o, String table, Medium medium,
                Register register, PrimaryKey pk)
                        throws RepositoryException
                        :updateRegister(o,table,medium, register,pk) {
        repository.updateRegister(table,medium,register,pk,o.getUser());
    }

    void around(UserProcess o, String table, Medium medium, Register register)
                        throws RepositoryException
                        :insertRegister(o,table,medium,register) {
        repository.insertRegister(table,medium,register,o.getUser());
    }

    void around(UserProcess o, String table, Medium medium, Register register)
                        throws RepositoryException
                        :removeRegister(o,table,medium, register) {
        repository.removeRegister(table,medium,register,o.getUser());
    }

    void around(UserProcess o, String table, Medium medium)
                        throws RepositoryException
                        :resetAllFlags(o,table,medium) {
        repository.resetAllFlags(table,medium,o.getUser());
    }

    void around(UserProcess o, String table, Medium medium)
                        throws RepositoryException
                        :removeAllRegisters(o,table,medium) {
        repository.removeAllRegisters(table,medium,o.getUser());
    }

    void around(UserProcess o) throws RepositoryException: optimize(o) {
        repository.optimize();
    }

    after() throwing(TableLockedException ex) throws RepositoryException
                                              :exceptionHandler() {
        throw new RepositoryException(ex);
    }

    after() throwing(TableNotLockedException ex) throws RepositoryException
                                                 :exceptionHandler() {
        throw new RepositoryException(ex);
    }
}
```

A.3 RepositoryAspect

```
package br.com.mobile.ms.repository.aspects;

import br.com.mobile.ms.repository.RepositoryFacade;
import br.com.mobile.ms.repository.Repository;
import br.com.mobile.ms.util.SystemProperties;
import br.com.mobile.ms.util.log.Log;

public aspect RepositoryAspect {
    private Repository repositoryImplementation;

    public RepositoryAspect() {
        try {
            String className = SystemProperties.getSystemParameters()
                                               .getParam("REPOSITORY_CLASS_NAME");
            repositoryImplementation = (Repository)Class.forName(className)
                                               .newInstance();
        }catch (Exception ex) {
            Log.getInstance().logException("ERRO: Na criacao do "+
                                           "REPOSITORY_CLASS_NAME",ex);
        }
    }

    after() returning (RepositoryFacade fac):call(RepositoryFacade.new(..)) {
        fac.setRepository(repositoryImplementation);
    }
}
```

A.4 AbstractStateSynchronizationAspect

```
package br.com.mobile.ms.repository.aspects;

import br.com.mobile.ms.business.User;

abstract aspect AbstractStateSynchronizationAspect {
    private User user;

    public AbstractStateSynchronizationAspect() {
    }

    public User getUser() {
        return user;
    }

    public void setUser(User u) {
        user = u;
    }
}
```

A.5 OutputStateSynchronizationAspect

```
package br.com.mobile.ms.repository.aspects;

import java.util.Iterator;

import br.com.mobile.ms.data.in.ReplicaUpdater;
import br.com.mobile.ms.data.in.InputProcessorThread;
import br.com.mobile.ms.data.out.OutputProcessorThread;
import br.com.mobile.ms.merge.Merger;
import br.com.mobile.ms.merge.MergeException;
import br.com.mobile.ms.repository.Table;
import br.com.mobile.ms.repository.Register;
import br.com.mobile.ms.repository.RepositoryException;
import br.com.mobile.ms.util.SystemProperties;
import br.com.mobile.ms.business.User;
import br.com.mobile.ms.util.log.Log;

public aspect OutputStateSynchronizationAspect extends
                        AbstractStateSynchronizationAspect
                        pertarget (execution (* OutputProcessorThread.*(..)) ||
                                   execution(OutputProcessorThread.new(..))){

    public OutputStateSynchronizationAspect() {

    }

    private void updateRepository(Table[] differences)
                                                        throws RepositoryException {
        RepositoryFacade fac = SystemProperties.getRepositoryFacade();
        for (int i = 0; i < differences.length; i++) {
            Iterator iter = differences[i].iterator();
            while (iter.hasNext()) {
                Register item = (Register)iter.next();
                char status = item.getField("STATUS").getValue().getChar();
                if (status==Register.STATUS_DELETED) {
                    fac.removeRegister(
                      differences[i].getName(),differences[i].getMedium(),item);
                } else if (status==Register.STATUS_INSERTED) {
                    fac.insertRegister(
                      differences[i].getName(),differences[i].getMedium(),item);
                } else {
                    fac.updateRegister(
                       differences[i].getName(),differences[i].getMedium(),item,
                       differences[i].getPrimaryKeyLayout());
                }
            }
            fac.removeAllRegisters(
                 differences[i].getName(),differences[i].getMedium(),getUser());
            fac.resetAllFlags(
                 differences[i].getName(),differences[i].getMedium(),getUser());
        }
    }

    pointcut outputExecution():
                        cflow(execution(void OutputProcessorThread.process()));

    pointcut updateMerge():
                      outputExecution() &&
                      call(Table[] OutputProcessorThread.processTableMerge(..));

    pointcut updateGeneralTable(String name, Table[] t): outputExecution() &&
                            call(void processGeneralTable(String,Table[])) &&
                            args(name,t);

    pointcut updateSingleWriterTable(String name, Table[] t):
                            outputExecution() &&
                            call(void processSingleWriter(String,Table[])) &&
                            args(name,t);
```

```
    after() returning (Table[] differences) throws RepositoryException
                                            :updateMerge() {
        updateRepository(differences);
    }

    after(String name, Table[] t) throws RepositoryException
                                  :updateGeneralTable(name,t) {
        RepositoryFacade fac = SystemProperties.getRepositoryFacade();
        fac.removeAllRegisters(name,t[0].getMedium());
        fac.resetAllFlags(name,t[0].getMedium());
    }

    after(String name, Table[] t) throws RepositoryException
                                  :updateSingleWriterTable(name,t) {
        RepositoryFacade fac = SystemProperties.getRepositoryFacade();
        fac.removeAllRegisters(name,t[0].getMedium(),getUser());
        fac.resetAllFlags(name,t[0].getMedium(),getUser());
    }
}
```

A.6 InputSynchronizationAspect

```
package br.com.mobile.ms.repository.aspects;

import java.util.Iterator;
import br.com.mobile.ms.data.in.ReplicaUpdater;
import br.com.mobile.ms.data.in.InputProcessorThread;
import br.com.mobile.ms.repository.Table;
import br.com.mobile.ms.repository.Register;
import br.com.mobile.ms.repository.RepositoryException;
import br.com.mobile.ms.util.SystemProperties;
import br.com.mobile.ms.business.User;
import br.com.mobile.ms.util.log.Log;

privileged aspect InputStateSynchronizationAspect extends
                        AbstractStateSynchronizationAspect
                        pertarget (execution (* InputProcessorThread.*(..)) ||
                                   execution(InputProcessorThread.new(..))){

    public InputStateSynchronizationAspect() {
    }

    private void updateRepository(ReplicaUpdater rep)throws RepositoryException{
        RepositoryFacade fac = SystemProperties.getRepositoryFacade();
        Iterator iter = rep.difference.iterator();
        while (iter.hasNext()) {
            Register item = (Register)iter.next();
            String op = (String)rep.registerOperations.get(item);
            if (op.equals(ReplicaUpdater.DELETE)) {
                fac.removeRegister(
                      rep.difference.getName(),rep.difference.getMedium(),item);
            } else if (op.equals(ReplicaUpdater .INSERT)) {
                fac.insertRegister(
                      rep.difference.getName(),rep.difference.getMedium(),item);
            } else {
                fac.updateRegister(
                      rep.difference.getName(),rep.difference.getMedium(),item,
                      rep.difference.getPrimaryKeyLayout());
            }
        }
    }


    pointcut inputExecution():cflow(execution(void InputProcessorThread.run()));

    pointcut updateReplica(ReplicaUpdater rep): inputExecution() &&
              (call(boolean ReplicaUpdater.processCompleteData(Table,Table)) ||
               call(boolean ReplicaUpdater.processPartialData(Table,Table))) &&
               target(rep);

    boolean around(ReplicaUpdater rep): updateReplica(rep) {
        boolean ok = proceed(rep);
        if(ok) {
            try {
                updateRepository(rep);
            } catch (RepositoryException ex) {
                ok = false;
                Log.getInstance().logException("ERRO: Processando retorno "+
                                   "para o usuário '"+getUser().getLogin()+
                                   "'",ex);
            }
        }
        return ok;
    }
}
```

A.7 DataSynchronizationAspect

```
package br.com.mobile.ms.repository.aspects;

import br.com.mobile.ms.data.out.OutputProcessorThread;
import br.com.mobile.ms.data.in.InputProcessorThread;

public aspect DataSynchronizationAspect {

    public DataSynchronizationAspect() {
    }

    after() returning (OutputProcessorThread process)
                                        :call(OutputProcessorThread.new(..)) {
       OutputStateSynchronizationAspect.aspectOf(process)
                                      .setUser(process.getUser());
    }

    after() returning (InputProcessorThread process)
                                         :call(InputProcessorThread.new(..)) {
       InputStateSynchronizationAspect.aspectOf(process)
                                     .setUser(process.getUser());
    }
}
```

Referências

[1] Vander Alves and Paulo Borba. Distributed adapters pattern:A design pattern for object-oriented distributed applications. In *First Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.

[2] Scott Ambler. *Building Object Applications that Work*. Cambridge University Press and Sigs Books, 1998.

[3] Grady Booch. *Object–Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern–Oriented Software Architecture*. John Wiley & Sons, 1996.

[5] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect–Oriented Programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison–Wesley, second edition, 2000.

[8] Demeter Research Group. Connections between Demeter/Adaptive Programming and Aspect-Oriented Programming (AOP). At http://www.ccs.neu.edu/home/lieber/-connection-to-aop.html.

[9] Demeter Research Group. Subject-Oriented Programming (SOP) and adaptive programming (AP). At http://www.ccs.neu.edu/research/demeter/SOP/.

[10] Lieberherr K. J., Silva-Lepe I., and et al. Adaptive Object–Oriented Programming Using Graph–Based Customization. *Communications of the ACM*, 37(5):94–101, 1994.

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[12] Ramnivas Laddad. I want my aop!, part 1. *JavaWorld*, January 2002. Avaliable at http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html.

[13] Eduardo Laureano. Persistence Implementation with AspectJ. Master's thesis, Informatics Center (CIn) — Federal University of Pernambuco (UFPE) — Brazil, January 2002.

[14] Bertrand Meyer. *Object–Oriented Software Construction*. Prentice–Hall, second edition, 1997.

[15] Gail C. Murphy, Robert J. Walker, Elisa L.A. Baniassad, Martin P. Robillard, Albert Lai, and Milk A. Kersten. Does aspect–oriented programming work? *Communications of the ACM*, 44(10):75–77, October 2001.

[16] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying suject–oriented composition. *TAPOS*, 2(3):179–202, 1996. Special Issue on Subjectivity in OO Systems.

[17] Harold Ossher and Peri Tarr. Using subject–oriented programming to overcome common problems in object–oriented software development/evolution. In *International Conference on Software Engineering, ICSE'99*, pages 698–688. ACM, 1999.

[18] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the OOPSLA '02 conference on Object Oriented Programming Systems Languages and Applications*. ACM Press, November 2002. To appear.

[19] AspectJ Team. The AspectJ Programming Guide. At http://aspectj.org, 2002.