Reengenharia do Projeto do Servidor *Web* JAWS utilizando Programação Orientada a Aspectos[1]

Uirá Kulesza *Dilma Menezes da Silva*

Departamento de Ciência da Computação
Universidade de São Paulo
E-mail: {uira, dilma}@ime.usp.br

Resumo

Programação orientada a aspectos foi proposta originalmente como uma técnica de programação que busca facilitar a separação de código relativo a requisitos funcionais do referente a requisitos não-funcionais. Este artigo descreve a reengenharia do projeto do servidor *Web* adaptativo JAWS usando programação orientada a aspectos. Os objetivos do estudo foram analisar e avaliar a aplicabilidade desta abordagem no projeto de sistemas com requisitos de adaptação estática e dinâmica. O estudo demonstrou que a aplicação dos preceitos de programação orientada a aspectos em um sistema com características de adaptação semelhantes ao JAWS pode levar a várias vantagens: separação de interesses e facilidades na reutilização, na evolução estática e na configuração dinâmica do sistema.
Palavras-chaves: programação orientada a aspectos, projeto orientado a objetos, sistemas adaptáveis.

Abstract

Aspect-Oriented Programming (AOP) has been proposed to facilitate the separation of code related to functional and non-functional system requirements. This paper describes the reengineering of the JAWS Web server design using AOP. The goals of this study were the analysis and assessment of applying AOP in the design of systems with static and dynamic adaptation requirements. The study demonstrated that the use of aspect-oriented programming can bring about several benefits in the design of adaptable systems, such as: separation of concerns, facility in reutilization, static evolution, and dynamic configuration.
Key-words: aspect-oriented programming, object-oriented design, adaptable systems.

1. Introdução

Apesar do amadurecimento das pesquisas em engenharia de *software*, manutenção de *software* permanece como um problema central à área. O processo de manutenção envolve não apenas a correção de erros, mas sobretudo a adequação do sistema para a integração de novas tecnologias e novos requisitos. O alto custo associado a estas adequações [1, 2] estimula o desenvolvimento de sistemas mais flexíveis a mudanças futuras. Outros fatores que motivam o desenvolvimento de sistemas mais flexíveis são:

- a dificuldade na especificação e entendimento dos requisitos de determinados domínios de aplicação (ex: sistemas de tempo real ou tolerante a falhas);
- a possibilidade de funcionamento por tempo prolongado [3];

[1] Este trabalho teve o apoio financeiro da FAPESP (processo 98/10186-2) e CNPq/NSF (processo 680037//99-3).

- a tendência a constantes e rápidas mudanças nos requisitos de classes específicas de sistemas, como por exemplo, aplicações financeiras;
- a possibilidade de reutilização da estrutura do sistema para a construção de similares;
- a crescente demanda por aplicações que executem sobre o ambiente de execução heterogêneo, constantemente modificado e altamente dinâmico que é a *Internet*.

Uma das abordagens propostas para tratar questões de evolução e mudança em *software* é construí-lo com base em uma arquitetura e projeto concebidos de forma a serem adaptáveis. Um sistema adaptável pode ser definido "como um sistema que pode ser facilmente modificado" [4], ou "um sistema que permite adaptar facilmente sua estrutura completa ou partes específicas devido a mudanças nos requisitos" [5]. Em [6], são apresentadas quatro características que um sistema adaptável deveria ter, sendo elas: flexibilidade, extensibilidade, facilidade de conserto e ajuste de desempenho. Neste mesmo trabalho, os autores destacam o paradigma orientado a objetos como sendo promissor para o desenvolvimento de sistemas adaptáveis.

O paradigma de orientação a objetos (OO) se consolidou ao longo da década de 90, e estudos [7, 8] têm demonstrado as facilidades que ele pode trazer para o processo de manutenção de *software*, tais como, facilidade de reuso através de relações de herança e composição de classes, e encapsulamento de decisões de implementação. Entretanto, diversos problemas ainda são encontrados durante a evolução estática e dinâmica destes sistemas, entre eles: entrelaçamento de código relacionado a diferentes interesses [9]; dificuldade no entendimento de colaborações entre classes [10]; aumento da complexidade do sistema com a construção de extensas hierarquias de classes e uso de polimorfismo/ligação dinâmica [8, 11]. Com o objetivo de favorecer requisitos de manutenibilidade de sistemas OO, a comunidade de engenharia de *software* vem propondo nos últimos anos diversas técnicas de projeto de *software*, tais como: padrões [12, 13], programação orientada a domínio [14], programação adaptativa [15] e programação orientada a aspectos [9]. Alguns estudos têm sido conduzidos recentemente com o intuito de avaliar a usabilidade e utilidade de tais técnicas [16, 17, 18, 19, 20, 21].

Paralelamente, as comunidades de sistemas configuráveis distribuídos e sistemas operacionais vêm estudando e desenvolvendo há bastante tempo diversas implementações de sistemas que incorporam aspectos de adaptação [22, 23]. Este trabalho investiga o impacto da técnica de programação orientada a aspectos (POA) na reconstrução do projeto de um destes sistemas. O objetivo foi avaliar a técnica de POA em relação a sua utilidade e vantagens no projeto de sistemas com requisitos de adaptação estática e dinâmica.

Este artigo descreve um estudo de reengenharia do projeto do sistema JAWS, um servidor *Web* adaptativo e de alto desempenho [24], utilizando a técnica de POA. O JAWS foi escolhido para participação no estudo por ser um sistema orientado a objetos, desenvolvido com a utilização de diversos padrões catalogados pela comunidade e, principalmente, por apresentar explicitamente diversas características de adaptação.

O artigo apresenta a seguinte organização: na seção 2 são feitas considerações a respeito da técnica de programação orientada a aspectos, com ênfase especial para a abordagem que está sendo desenvolvida no Xerox PARC, denominada AspectJ. Na seção 3 são descritas as características e estrutura do JAWS, assim como são apresentados entrelaçamentos de interesses identificados em seu código. Metodologia e modelos resultantes do estudo são apresentados na seção 4. A seção 5 traz os resultados da comparação entre o projeto original do JAWS e o seu novo projeto usando a técnica de POA. Na seção 6 o estudo é confrontado com trabalhos relacionados. Conclusões finais acerca do trabalho estão presentes na seção 7.

2. Programação Orientada a Aspectos

Programação Orientada a Aspectos (POA) [9] é uma técnica que propõe um novo paradigma para o desenvolvimento de sistemas. Segundo pesquisadores da Xerox em Palo Alto, as linguagens de programação atuais permitem apenas a construção de estruturas (procedimentos, funções, objetos) associadas diretamente à funcionalidade da aplicação. Outras propriedades gerais de interesse do sistema (concorrência, distribuição, manipulação de falhas, alocação de memória) decorrentes de decisões de projeto ficam entrelaçadas e difusas ao longo do código destas estruturas dificultando seu desenvolvimento, localização e futura manutenção. Também as metodologias de projeto procuram quebrar o sistema em unidades funcionais que juntas colaboram para o atendimento dos requisitos do sistema. Assim, tanto as metodologias quanto as linguagens de programação atuais focam sua atenção principal na decomposição funcional do sistema.

O objetivo de POA é promover a especificação separada dos interesses/requisitos funcionais e não-funcionais de sistemas, com a promessa de que esta separação traga facilidades para o processo de entendimento, evolução e manutenção do *software* [25]. Um sistema em POA é composto de:

- um conjunto de componentes – unidades oriundas da decomposição funcional do sistema e que podem ser expressas nas linguagens de programação atuais. Exemplos: entidades do domínio da aplicação, interface gráfica da aplicação, bibliotecas de funções utilitárias;
- um conjunto de aspectos – propriedades gerais do sistema relacionadas a seus interesses não-funcionais, e que afetam o seu desempenho ou a semântica dos componentes de uma forma sistemática. Para especificar e compor este conjunto de aspectos que afetam os componentes do sistema, propõe-se a utilização de linguagens específicas. Exemplos de tais propriedades gerais são sincronização de objetos concorrentes, manuseio de falhas e exceções, estratégias de comunicação, otimização de alocação de memória, persistência.

Em POA, um sistema é construído através de uma clara separação entre componentes e aspectos, e em tempo de compilação ou execução estes elementos são combinados e compostos, através de um compilador (*aspect weaver*), para produção do sistema completo.

2.1 AspectJ™

AspectJ [26] é uma extensão orientada a aspectos para Java™. Ela permite especificar "aspectos" que afetam classes e objetos de um programa escrito em Java.

Assim como uma classe Java, um aspecto em AspectJ possui um nome e pode declarar atributos e métodos com diferentes visibilidades. Cada aspecto pode ainda definir duas construções adicionais: (a) `introduction` – que permite inserir novos atributos e métodos em classes do sistema; e (b) `advice` – que permite inserir código antes (`before`) ou depois (`after`) de métodos de classes ou objetos do sistema.

Considere a existência de uma classe `Point`, contendo dois atributos inteiros `x` e `y`, e dois métodos públicos, `getx()` e `gety()`, para acesso a tais atributos. Na figura 1 é definido o aspecto `ShowAccesses` que afeta esta classe `Point` da seguinte forma: (1) são adicionados um novo atributo (`extraAtribute`) e um novo método (`extraMethod`) em tal classe; e (2) os métodos `getx()` e `gety()` da classe `Point` são afetados por uma construção `advice` que indica a entrada e saída da execução de tais métodos.

As inserções de atributos e métodos de construções `introduction` de AspectJ são sempre realizadas sobre classes e em tempo de compilação. Assim, todas as instâncias de uma classe afetada por um aspecto incorporam as extensões de construções `introduction`. Já a construção `advice` pode afetar uma classe (conseqüentemente todas suas instâncias serão estendidas) ou apenas um conjunto específico de objetos, através da definição de construções

advice não estáticas. Neste último caso, tem-se o conceito de aspecto dinâmico que deve ser associado explicitamente aos objetos a serem estendidos.

```
public aspect ShowAccesses {
        static int nsets=0;
        introduction Point {
                int extraAtributte;
                public void extraMethod(){ ... }
        }
        crosscut getters():
                Point & (int getx() | int gety());
        static advice getters(){
            before {
                    System.out.println(" Antes de Ler ");
            }
            after {
                  System.out.println(" Depois de Ler ");
                  ++nsets;
            }
        }
}
```

crosscut:
Indica pontos específicos de classes/objetos que terão sua implementação afetada por um determinado aspecto.

Figura 1: Aspecto `ShowAccesses`

Na figura 2 o aspecto `ShowAccesses` é redefinido para que a construção `advice getters()` passe a ser não estática e assim possa afetar apenas instâncias específicas. O código de associação de objetos `Point` a uma instância do aspecto `DynamicShowAccesses` é também apresentado na figura 2. Apenas após esta associação é que os objetos passam a incorporar as extensões da construção `getters()`.

```
public aspect DynamicShowAccesses {
       crosscut getters():
                Point & (int getx() | int gety());
        advice getters(){
           before {
                System.out.println(" Antes de Ler ");
           }
           after {
                System.out.println(" Depois de Ler ");
           }
        }
}
...
Point p1, p2;
DynamicShowAccesses dynShowAccesses = new DynamicShowAccesses();
p1 = new Point();
p2 = new Point();
// Associação dos objetos p1 e p2 ao aspecto dynShowAccesses;
dynShowAccesses.addObject(p1);
dynShowAccesses.addObject(p2);
...
```

Figura 2: Aspecto `DynamicShowAccesses`

3. JAWS - Servidor *Web* Adaptativo

JAWS [24] é um *framework* orientado a objetos que permite a construção de servidores *Web* configuráveis. Diversas estratégias de configuração podem ser combinadas no JAWS, permitindo sua adaptação tanto estática quanto dinâmica a mudanças ocorrendo no seu ambiente de execução. O próprio JAWS é um servidor *Web* adaptativo e de alto desempenho que implementa o protocolo HTTP. Além disso, a estrutura definida pelo *framework* do JAWS possibilita que ele seja utilizado para a construção de outros tipos de servidores de comunicação.

O JAWS é formado por um conjunto de componentes, que foram estruturados utilizando o *framework* para *software* de comunicação ACE [27] e diversos padrões de projetos [28, 12, 13] catalogados pela comunidade.

São definidas no JAWS quatro estratégias que podem ser configuradas em tempo de compilação ou execução:

(1) **Estratégia de Concorrência** – determina uma política de concorrência para execução do protocolo definido para o JAWS. Em sua versão atual estão implementados duas diferentes estratégias: (a) *Thread-per-Request* – manipula cada pedido de cliente em uma *thread* separada; e (b) *Thread-Pool* – durante inicialização do JAWS são ativadas um conjunto pré-definido de *threads*, as quais obtêm pedidos de clientes de uma fila e os processam;

(2) **Estratégia de I/O** – determina uma política para entrega e recuperação de dados. Duas estratégias de I/O estão presentes na versão atual do JAWS: (a) I/O síncrona – neste modelo o *kernel* não retorna a *thread* de controle para o servidor antes que o pedido da operação de I/O tenha sido completado ou tenha falhado; e (b) I/O assíncrona – neste modelo o *kernel* executa cada pedido de operação de I/O de forma assíncrona, enquanto o servidor processa outros pedidos;

(3) **Estratégia de "Pipeline" de Protocolo** – determina mecanismos para reconfiguração do protocolo HTTP definido pelo JAWS. Um protocolo é definido no JAWS como um conjunto de tarefas organizadas em seqüência, seguindo o padrão *Pipes and Filters* [13]. Este protocolo pode ser modificado estática ou dinamicamente (através da adição, remoção ou substituição de tarefas);

(4) **Estratégia de Cache de Arquivos** – determina uma política para armazenamento de arquivos em cache, com o intuito de melhorar o desempenho do servidor *Web*, através da diminuição de acessos ao servidor de arquivos remoto. Tal estratégia encontra-se ainda em desenvolvimento, e portanto, não foi abordada neste estudo.

Nas subseções seguintes, são apresentados: um diagrama de classes UML [29] que descreve parcialmente a estrutura do JAWS, e entrelaçamentos de diferentes interesses identificados no código do JAWS.

3.1 Estrutura do JAWS

As principais classes presentes no *framework* do JAWS são as seguintes:

(a) `JAWS_Server` – responsável por agregar a política de funcionamento do servidor, pela configuração inicial das estratégias do JAWS e pela ativação de algumas destas estratégias;

(b) `JAWS_Default_Dispatch_Policy` – define a política do servidor e mantém as estratégias do JAWS;

(c) `JAWS_Concurrency_Base` – define a estratégia de concorrência. Esta classe implementa mecanismos para executar de forma concorrente o protocolo do JAWS. Suas subclasses implementam estratégias de concorrência concretas;

(d) `JAWS_IO` – implementa a estratégia de I/O. Suas subclasses implementam estratégias de I/O concretas;

(e) `JAWS_IO_Acceptor` – define a estratégia de estabelecimento de conexões com clientes. Possui também subclasses que implementam estratégias concretas;

(f) `JAWS_Pipeline_Handler` – define uma interface básica para as tarefas do protocolo do JAWS. Subclasses implementam cada uma da tarefas do protocolo;

(g) `JAWS_Data_Block` – define a unidade de comunicação entre as tarefas do protocolo do JAWS.

A Figura 3 apresenta um diagrama de classes parcial do JAWS.

Figura 3: Diagrama de Classes parcial do JAWS

3.2 Entrelaçamento de Interesses

A análise da estrutura e comportamento do projeto do JAWS demonstrou o entrelaçamento de código relacionado aos seguintes interesses:

- configuração e ativação das estratégias - as diferentes estratégias presentes no JAWS são configuradas e ativadas inicialmente nos métodos `init()` e `open()` da classe `JAWS_Server`. Embora o código relacionado a tal interesse esteja bem localizado dentro destes métodos, há um claro entrelaçamento do código de configuração e ativação das diferentes estratégias. Tal entrelaçamento pode dificultar o entendimento e posteriores adaptações no comportamento destes métodos a medida que novas estratégias são criadas ou estratégias existentes são modificadas ou removidas do sistema. Além disso, a reconfiguração dinâmica e reativação de uma determinada estratégia poderia ser feita em diferentes pontos ao longo do código do JAWS, o que torna o código relacionado ao atendimento de tal interesse difuso por entre diversos métodos de classes do sistema, possivelmente aumentando o entrelaçamento de código com outros interesses;
- definição do protocolo e processamento concorrente - estes interesses são ambos atendidos pela classe `JAWS_Concurrency_Base`. Isto dificulta o entendimento da "porção" do projeto do sistema relacionado a cada um dos interesses, assim como impede a possível reutilização do protocolo com um mecanismo de suporte a processamento concorrente diferente do fornecido pelo *framework* ACE, e que é o utilizado pelo JAWS;

- tratamento de I/O assincrono - o tratamento deste interesse no JAWS envolve não apenas a configuração de uma estratégia de I/O assíncrona (classe `JAWS_Asynch_IO`), mas também a invocação do método `wait_for_completion()` de um objeto do tipo `JAWS_Waiter`, cuja responsabilidade é lidar com eventos de I/O assíncronos que estão sendo tratados em um dado instante. Como a invocação deste método no JAWS é feita dentro do método `svc_hook()` da instância que define a sua estratégia de concorrência, isto acaba trazendo entrelaçamento de código relacionado aos interesses de tratamento de I/O assíncrono, processamento concorrente e de definição do protocolo do JAWS. Outros aspectos de configuração do objeto `JAWS_Waiter` estão também emaranhados ao longo do código de diversos métodos da classe `JAWS_Concurrency_Base` e suas subclasses.

A identificação destes entrelaçamentos de código relacionados a diferentes interesses de projeto do JAWS foi de fundamental importância para o processo de reconstrução do seu projeto utilizando a técnica de POA, indicando partes específicas de classes do sistema que poderiam ser projetadas utilizando a abstração de aspectos. Na figura 3 apresentada anteriormente são explicitados os pontos de entrelaçamento na estrutura de classes do JAWS.

4. Reengenharia do Projeto do JAWS com Programação Orientada a Aspectos[2]

O estudo de reengenharia do projeto do JAWS com POA consistiu dos seguintes passos: (I) leitura e entendimento do código-fonte do JAWS com a possível identificação de entrelaçamento de interesses (apresentados na seção 3); (II) engenharia reversa[3] do JAWS com a geração de modelos UML que descrevessem a sua estrutura e comportamento; (III) processo de reconstrução do JAWS, no nível de estruturas de projeto e implementação, utilizando a técnica de POA (apresentado nesta seção 4); e (IV) comparação do projeto original com o novo projeto do JAWS com POA (descrita na seção 5).

Nas subseções seguintes são descritos o método empregado e modelos resultantes da reconstrução do JAWS utilizando a técnica de POA.

4.1 Componentes Básicos do JAWS

Na fase inicial de modelagem do JAWS com Programação Orientada a Aspectos (POA), o foco principal foi determinar a funcionalidade básica que um servidor de comunicação, tal como o JAWS, deve implementar. Este tipo de raciocínio foi útil para buscar a separação dos componentes funcionais básicos de um servidor de comunicação dos aspectos que os afetam.

As seguintes atividades funcionais básicas por parte de um servidor de comunicação foram identificadas: (1) atividade de espera por conexões de clientes em uma porta específica da rede; (2) atividade de recepção de pedidos de clientes e encaminhamento dos mesmos para processamento; (3) atividade de processamento do pedido do cliente, que pode eventualmente ser customizada de acordo com as características do pedido; e (4) atividade de retorno de um arquivo ou erro para o cliente do pedido processado.

Cada uma destas atividades funcionais básicas possui no JAWS um componente/classe responsável por seu processamento. Os componentes funcionais do JAWS que definem estas atividades funcionais são as tarefas do seu protocolo (subclasses de `JAWS_Pipeline_Handler`)

[2] JAWS está codificado em C++, e AspectJ é utilizada para construir programas orientados a aspectos em Java. O uso destas duas diferentes tecnologias não trouxe problemas para o estudo porque houve interesse em investigar apenas questões relacionadas ao projeto do JAWS, e Java e C++ são linguagens orientadas a objetos muito similares neste aspecto. Assim, foi assumido a disponibilidade de uma versão C++ de AspectJ.

[3] A engenharia reversa do JAWS foi realizada manualmente através da análise do seu código-fonte.

em colaboração com a estratégia de I/O. Estes componentes foram mantidos no novo projeto do JAWS com POA. Houve, entretanto, a necessidade de criar um elemento que agregasse tais componentes e realizasse o gerenciamento da execução destas atividades funcionais. Foi então criada a classe `JAWS_RequestManager`, cuja função principal é definir o serviço a ser oferecido pelo JAWS, integrando os componentes responsáveis pelo processamento do seu protocolo. No projeto original do JAWS, a função desempenhada pelo `JAWS_RequestManager`, é atendida pelo método `JAWS_ConcurrencyBase::svc_hook()`. Na figura 4 é apresentado um diagrama de classes com o `JAWS_RequestManager` e os componentes funcionais que ele agrega.

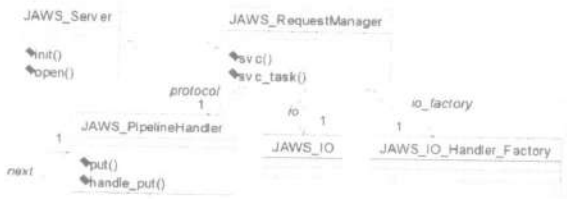

Figura 4: Diagrama de Classe do `JAWS_RequestManager`

4.2 Aspectos no JAWS

O passo seguinte da reconstrução do JAWS com POA foi identificar possíveis aspectos que poderiam afetar o componente `JAWS_RequestManager` e os demais componentes funcionais que ele agrega, apresentados na seção anterior. O entendimento do funcionamento e estrutura do JAWS, e a compreensão de aspectos capturados em outros sistemas [18, 19, 30, 31] foi bastante útil para concluir este passo.

Os seguintes aspectos foram identificados:

(a) **estratégia de concorrência** – os pedidos de clientes poderiam ser processados de forma concorrente por diferentes *threads* ou processos, objetivando melhorar o desempenho e tempo de resposta do próprio servidor;

(b) **estratégia de aceitação de conexões** - responsável por definir um componente *Acceptor* [32] para receber pedidos de clientes em uma determinada porta da rede;

(c) **configuração da estratégia de I/O** – para manipulação de dados que chegam ao servidor ou que são enviados por ele é importante definir uma configuração para a estratégia de tratamento de I/O;

(d) **configuração do protocolo** – para determinados tipos de pedidos para um servidor de comunicação há a necessidade de se construir protocolos customizados;

(e) **monitoramento do ambiente de execução e carga do sistema** – é fundamental definir elementos no JAWS que sejam responsáveis pela supervisão do seu ambiente de execução e carga, a fim de que esta atividade permita a adaptação dinâmica de suas diferentes estratégias;

(f) **tratamento de falhas e exceções** – poderiam ser tratados separadamente em aspectos para facilitar a manutenção de tais códigos e permitir que sejam reutilizados na construção de novos servidores de comunicação;

(g) **comunicação remota com outros servidores** (de banco de dados ou de arquivos) – um aspecto poderia se responsabilizar por garantir a transparência nesta comunicação;

(h) **serviço de *logging*** – servidores *Web*, em geral, realizam o registro de suas atividades para fins de administração do sistema;

(i) ***tracing* da execução do servidor** – pode ser útil para fins de depuração e administração do sistema, podendo ser utilizado em conjunto com o serviço de *logging*.

Embora todos os aspectos acima sejam relevantes, foi enfocada a modelagem dos aspectos de (a) a (d), que já estavam expressos no projeto do JAWS, e também do aspecto (e), que tem uma relação direta com seus objetivos de adaptação dinâmica. Os aspectos de (f) a (i) vêm sendo abordados em outros trabalhos [18, 19, 30, 31], e portanto, não são tratados aqui. Nas subseções seguintes são apresentadas as modelagens dos aspectos de (a) a (e). Código-fonte em AspectJ relacionado ao novo projeto do JAWS pode ser encontrado em [33].

4.2.1 Estratégia de Concorrência

A estratégia de concorrência do JAWS é implementada pela classe `JAWS_Concurrency_Base` e suas subclasses `JAWS_Thread_Pool` e `JAWS_Thread_Per_Task`. Estas classes definem não apenas uma estratégia para gerência (criação, destruição e monitoramento) de *threads*, mas também se encarregam de definir o serviço a ser fornecido pelo JAWS, ou seja, de executar seqüencialmente as tarefas de seu protocolo.

No projeto com POA, a estratégia de concorrência foi separada em um aspecto que afeta diretamente o componente `JAWS_RequestManager`. Desta forma garante-se a separação de interesses, deixando a responsabilidade de implementação do serviço do protocolo do JAWS sob encargo do `JAWS_RequestManager`. Na figura 5 é apresentado um diagrama de classes e aspectos[4] do novo projeto com POA da estratégia de concorrência. São indicados as classes e/ou aspectos responsáveis por determinados interesses.

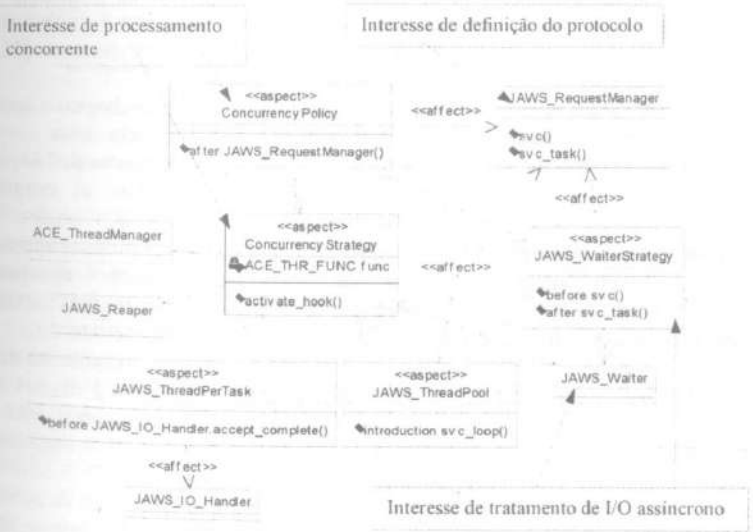

Figura 5: Diagrama de Classes e Aspectos da Estratégia de Concorrência

[4] Aspectos são representados nos diagramas de classes UML deste artigo, através do uso do estereótipo `<<aspect>>`. Relacões de dependência entre um aspecto A e uma classe C (com o estereótipo `<<affect>>`) indicam que o aspecto A afeta a implementação da classe C, e também que mudanças em C afetarão A. Cada aspecto A define ainda seu conjunto próprio de construções `introduction` e `advice`.

O aspecto `ConcurrencyPolicy` define qual (ou quais) componente/classe será afetado pela estratégia de concorrência, qual estratégia será instanciada e quando ela será ativada. No caso específico do JAWS, este aspecto afetará o componente `JAWS_RequestManager`, inicializando e ativando em seu construtor a estratégia de concorrência escolhida.

A estratégia de concorrência propriamente dita é implementada pelo aspecto `ConcurrencyStrategy`, cuja finalidade é definir uma interface padrão para uma estratégia concreta e manter atributos comuns para alguma estratégia de concorrência. No caso específico do JAWS, este aspecto agrega: (1) um objeto do tipo `ACE_Thread_Manager` – responsável por fornecer os mecanismos para execução concorrente do protocolo HTTP do JAWS e gerenciamento de *threads*; (2) um objeto do tipo `JAWS_Reaper` – responsável por realizar operações após a finalização de todas as *threads*; e (3) um objeto do tipo `JAWS_Waiter` - responsável pelo tratamento de eventos assíncronos. Além disso, ele ainda declara o atributo `func` que representa o método que será executado concorrentemente pelas *threads*, e define a assinatura do método de ativação da estratégia de concorrência. Os sub-aspectos de `ConcurrencyStrategy` são responsáveis pela implementação da interface definida pelo seu super-aspecto e podem definir inclusões ou extensões de métodos em algumas classes.

4.2.2 Estratégias de Aceitação de Conexões e de I/O

A estratégia de aceitação de conexões de um servidor de comunicação define que tipo de componente será utilizado para estabelecer conexões com seus clientes e quando este componente será criado, ativado e encerrado. No JAWS, este componente é implementado pela classe `JAWS_IO_Acceptor` e suas subclassses que encapsulam objetos *Acceptor* [32] do *framework* ACE.

A estratégia de I/O define uma política para tratamento dos dados que chegam e que são enviados por um servidor de comunicação. Ela é implementada no JAWS pela classe `JAWS_IO` – que define uma interface abstrata a ser implementada por estratégias concretas de I/O; e por suas subclasses `JAWS_Synch_IO` e `JAWS_Asynch_IO` – que representam as estratégias concretas de I/O síncrona e assíncrona, respectivamente.

No novo projeto do JAWS com POA, as configurações das estratégias de aceitação de conexões e I/O foram integradas dentro de um único aspecto, denominado `IO_Acceptor_Policy_Configuration`. Tal decisão de integração foi tomada devido a constatação da forte ligação entre as estratégias, evidente no código do método `init()` da classe `JAWS_Server`. Neste método, observa-se que o tipo de *Acceptor* e componente de I/O configurados para o JAWS dependem ambos do valor da variável de instância `dispatch` da classe `JAWS_Server`. Em um novo projeto estas estratégias poderiam, entretanto, ser configuradas separadamente em aspectos distintos. Na figura 6 é apresentado o diagrama de aspectos e classes relacionado ao interesse de configuração destas estratégias.

O aspecto `IO_Acceptor_Policy_Configuration` possui, para as estratégias de aceitação de conexões e I/O, basicamente a mesma função que o `ConcurrencyPolicy` possui para a estratégia de concorrência, ou seja, ele define: (1) qual componente funcional será afetado pela estratégia de aceitação e de que forma; (2) que tipo de *Acceptor* será instanciado e quando ele será ativado; e (3) uma configuração inicial para a estratégia de I/O. O aspecto `IO_Acceptor_Policy_Configuration` definido acima afeta a implementação do método construtor do `JAWS_RequestManager` criando e ativando a estratégia de aceitação de conexões e definindo uma configuração para a estratégia de I/O do JAWS no final (`after`) deste construtor.

A estratégia de aceitação de conexões é representada pelo aspecto `AcceptorStrategy`, que define uma interface padrão para estratégias concretas de aceitação e mantém um atributo do tipo `ACE_Acceptor`. As funções dos sub-aspectos de `AcceptorStrategy` são definir o tipo de objeto *Acceptor* do *framework* ACE que será utilizado; os atributos adicionais necessários para determinadas estratégias de aceitação; e inserções de métodos `accept()` (cláusulas `introduction`) que estendem subclasses de `JAWS_IO` com métodos para a aceitação de conexões baseado na estratégia escolhida.

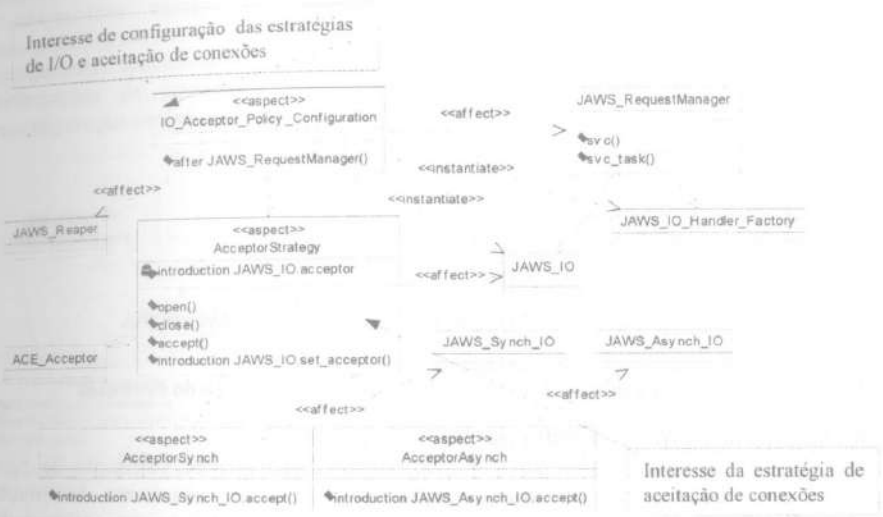

Figura 6: Diagrama de Classes e Aspectos das Estratégias de Aceitação de Conexões e I/O

4.2.3 Configuração e Extensão do Protocolo

Um protocolo pode ser visto como um conjunto de tarefas organizadas seqüencialmente, onde o resultado do processamento de uma delas serve como entrada para o processamento da subseqüente. O padrão *Pipes and Filters* [13] explicita bem este tipo de comportamento.

Para o protocolo do JAWS foram identificadas e descritas, na subseção 4.1, quatro tarefas básicas para a sua composição. Embora este conjunto de tarefas possa ser suficiente para processar a maioria dos pedidos HTTP de clientes *Web*, podem existir situações [24] onde a customização dinâmica do protocolo (através da inclusão, remoção, substituição e extensão de tarefas) torna-se necessária. Assim, no projeto do protocolo HTTP de um servidor *Web*, tal como o JAWS, é fundamental oferecer mecanismos que facilitem a sua configuração estática e dinâmica.

O protocolo do JAWS com POA foi projetado considerando dois tipos de componentes na sua estrutura, sendo eles: (1) tarefas - que definem as atividades funcionais do protocolo e que já estavam definidas no projeto original do JAWS como subclasses de `JAWS_Pipeline_Handler`; e (2) filtros - que definem extensões para as tarefas e que são definidos no novo projeto como aspectos que podem ser anexados dinamicamente às tarefas do protocolo.

Foi então criado o aspecto `ProtocolConfiguration`, responsável pelas configurações: **estática** – que consiste na inicialização e composição das tarefas e filtros – e **dinâmica** – que

consiste na definição de métodos e cláusulas `advice`, responsáveis pela inclusão, remoção, substituição e refinamento de tarefas e filtros. Na figura 7 é apresentado o diagrama de classes e aspectos relacionado ao interesse de configuração estática e dinâmica do protocolo HTTP do JAWS.

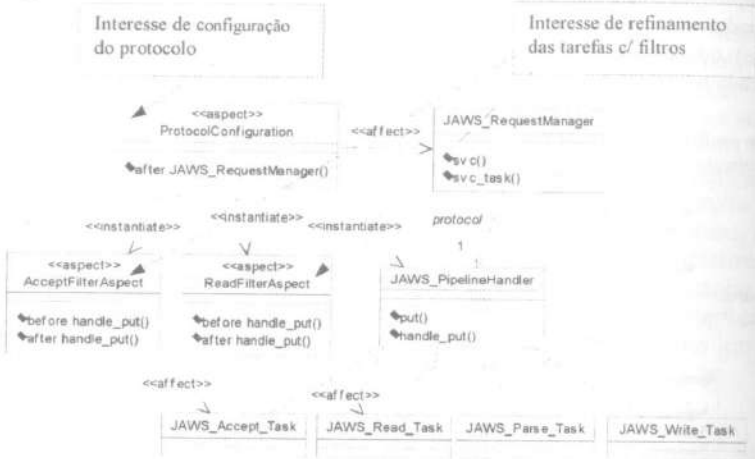

Figura 7: Diagrama de Classes e Aspectos da Configuração e Extensão do Protocolo

4.2.4 Monitoramento e Reconfiguração do Sistema

O JAWS é um servidor *Web* para alto desempenho que permite adaptação de suas estratégias em relação a fatores estáticos (ambiente de *hardware*, plataforma de sistema operacional) e fatores dinâmicos (carga na máquina em que o servidor executa, número de pedidos simultâneos ao servidor, carga de trabalho do servidor, uso de memória dinâmica). Seu projeto atual suporta a configuração estática e dinâmica de suas estratégias de I/O, concorrência e aceitação de conexões, e de seu protocolo. Entretanto, não existem ainda em seu projeto, componentes relacionados ao monitoramento do estado do seu ambiente de execução e carga no sistema, de forma a permitir que as adaptações de suas estratégias possam ser feitas dinamicamente de acordo com variações neste estado.

Na reconstrução do JAWS com POA, foi criado o aspecto `Monitoring`, que afeta o componente `JAWS_IO` e trata do interesse de monitoramento do ambiente de execução e carga no sistema. Diagrama contendo tal aspecto é apresentado na figura 8.

Para realizar o monitoramento, o aspecto `Monitoring` mantém:

- o estado do ambiente de execução no qual o JAWS opera – com informações sobre a quantidade de processadores presentes, a presença ou não de suporte a *threads* e I/O assíncrono no sistema operacional. Tais informações devem ser configuradas na inicialização do servidor, de forma automática (através da leitura de variáveis/parâmetros do sistema operacional) ou manual (por um administrador do sistema), e são úteis para determinar as possíveis estratégias que o JAWS poderá instanciar, assim como permitem definir sua configuração inicial;
- o estado da carga no servidor – com informações sobre o número de conexões estabelecidas com clientes, a quantidade de pedidos atendidos em um determinado intervalo de tempo, e *throughput* (bytes transmitidos por segundo). Estas informações são atualizadas em pontos específicos do código do JAWS, os quais são definidos por cláusulas `advice` do

aspecto `Monitoring`. A partir dos valores que estas informações da carga no servidor assumem elas podem ocasionar a reconfiguração dinâmica de alguma(s) das estratégias.

Os estados do ambiente de execução e da carga no servidor mantidos pelo aspecto `Monitoring` são repassados para o aspecto `Configurator` durante a inicialização e execução do JAWS. O aspecto `Configurator` é responsável pela definição de uma política para configurações estáticas e dinâmicas nas estratégias do JAWS, baseado nas variações dos estados mantidos pelo aspecto `Monitoring`. Para exercer sua função, o aspecto `Configurator` agrega os aspectos dinâmicos `IO_Acceptor_Policy_Configuration`, `ConcurrencyPolicy` e `ProtocolConfiguration`, e os associa dinamicamente aos respectivos componentes funcionais que eles afetam, durante a inicialização do sistema. Baseado nas informações do estado da carga no servidor, o `Configurator` pode direcionar a reconfiguração dinâmica de uma ou mais estratégias do JAWS.

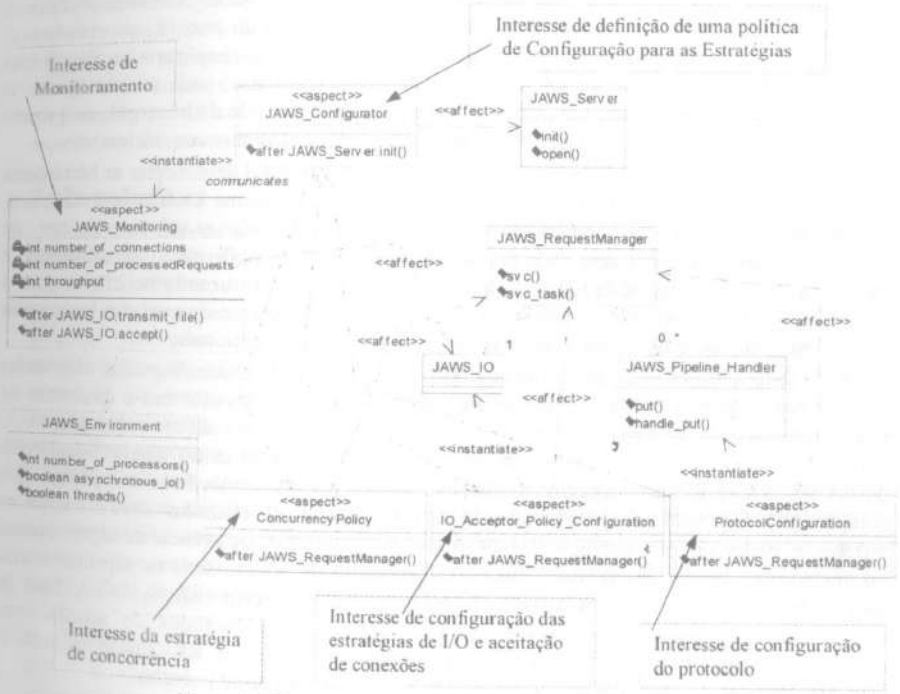

Figura 8: Diagrama de classes e aspectos dos Interesses de Monitoramento e Configuração do Sistema

5. Avaliação do Projeto do JAWS com Programação Orientada a Aspectos

Uma análise da nova estrutura do projeto do JAWS com POA revela diversas vantagens obtidas em relação ao seu projeto original, entre elas:

- **separação de interesses**: um dos principais objetivos da técnica de POA é tentar expressar sintaticamente e de forma separada os diversos interesses que estão envolvidos no projeto de um sistema [25]. No novo projeto do JAWS com POA, este objetivo de separação

de interesses foi alcançado através da expressão em hierarquias separadas de classes e aspectos, das estratégias de concorrência e aceitação de conexões, das configurações das estratégias de concorrência, I/O, aceitação de conexões e do protocolo, e também do interesse de monitoramento do ambiente de execução e carga no sistema. Os entrelaçamentos de código de diversos interesses descritos na subseção 3.2 foram evitados, com cada um deles sendo expresso por uma hierarquia distinta de aspectos/classes, o que conseqüentemente leva a um melhor entendimento de como tais interesses são alcançados, e contribui portanto, para facilidade de manutenção no código do JAWS;

- **reutilização**: a especificação separada de cada um dos interesses do JAWS permite também reutilizá-los em novos contextos ou em novos servidores de comunicação. Esta reutilização implica apenas na redefinição de algumas cláusulas presentes em aspectos da hierarquia de cada interesse que explicitam os pontos específicos do código do JAWS que cada aspecto afetará. Na estratégia de concorrência, por exemplo, o aspecto `ConcurrencyPolicy` define a configuração inicial e ativação da estratégia no construtor do `JAWS_RequestManager`. A reutilização da estratégia de concorrência em um novo contexto implicaria na redefinição desta cláusula, com a especificação do método de uma dada classe, onde a estratégia seria então configurada e possivelmente ativada. A estrutura como um todo da hierarquia do projeto de cada um dos interesses poderia assim ser facilmente reutilizada em novos contextos;
- **evolução estática**: com o novo projeto do JAWS é possível desconectar as hierarquias de aspectos/classes que afetam o `JAWS_RequestManager` de forma estática ou dinâmica. Assim, permite-se a fácil adaptação deste componente para novas hierarquias de aspectos/classes, que enderecem cada um de seus interesses de configuração de estratégias e monitoramento. Esta facilidade de "desplugar" os aspectos acaba trazendo benefícios para a evolução estática do sistema, permitindo reconstruir completamente cada uma das hierarquias de interesses que afetam o comportamento de seus componentes funcionais;
- **configurações estáticas e dinâmicas**: as hierarquias de classes/aspectos elaboradas para o novo projeto do JAWS permitem expressar as configurações estáticas e dinâmicas de suas estratégias. Os aspectos `Concurrency_Policy`, `IO_Acceptor_Policy_Configuration`, e `Protocol_Configuration` definem as configurações iniciais para as estratégias adaptativas do JAWS. Estes mesmos aspectos poderiam definir "pontos" específicos no código dos componentes funcionais do JAWS, através da especificação de cláusulas `advice`, onde as estratégias poderiam ser reconfiguradas dinamicamente a partir da ocorrência de algum evento ou mudança no estado do sistema. Uma outra alternativa de projeto seria os aspectos acima apenas definirem métodos para a troca de estratégias e o aspecto `Configurator` (que já agrega tais aspectos) se responsabilizar pela invocação destes métodos de acordo com mudanças ocorrendo nos estados mantidos pelo aspecto `Monitoring`. Esta última alternativa foi a escolhida na reengenharia do projeto do JAWS apresentada neste trabalho.

6. Trabalhos Relacionados

Em [18], Kersten & Murphy apresentam um estudo de casos de construção de um *software* de ensino baseado na *Web* usando AspectJ. O objetivo do estudo foi relatar a experiência do emprego da técnica de POA no desenvolvimento de um sistema que privilegia sua qualidade de manutenibilidade. O estudo destes autores apresentou diversas contribuições para a realização deste trabalho. Foram utilizadas na reengenharia do projeto do JAWS as chamadas políticas de reassociação (que consistem na alteração de atributos de objetos por um aspecto) para configurar as estratégias do protocolo e de I/O. Algumas diretrizes para projeto orientado a aspectos, apresentadas em [18], têm sido usadas com êxito no estudo do JAWS.

De forma similar ao trabalho destes autores, foi inicialmente mantido um modelo de objetos independente que permitiu pensar apenas na funcionalidade básica do sistema, e em seguida, foram identificados e projetados os diferentes aspectos que afetavam tal funcionalidade. Foram utilizadas também relações entre aspectos e classes, nas quais apenas os primeiros fazem suposições sobre o funcionamento do outro (*class-directional*). Este tipo de relação garante a relativa independência entre o modelo de objetos do sistema e as hierarquias de aspectos e classes projetadas para o JAWS. Finalmente, o trabalho de Kersten & Murphy ilustra o projeto de hierarquias de aspectos que foram úteis na compreensão da aplicabilidade e utilização da abordagem AspectJ de POA na fase inicial do estudo descrito neste artigo, além de mostrar exemplos concretos do uso de aspectos dinâmicos.

O estudo desenvolvido por Lippert & Lopes [19] possui similaridades e diferenças em relação ao estudo do JAWS. O trabalho destes autores, assim como o descrito neste artigo, realizou uma reengenharia parcial de um sistema existente usando AspectJ. O foco principal do trabalho de Lippert & Lopes, entretanto, foi analisar a aplicabilidade de AspectJ na especificação separada de interesses de detecção e manipulação de exceções em sistemas. O estudo apresentou evidências qualitativas das facilidades que AspectJ pode trazer para reutilização, tolerância a mudanças, suporte a diferentes configurações e desenvolvimento incremental do interesse de tratamento de exceções em sistemas.

7. Conclusões

Este trabalho apresentou um estudo de reengenharia do projeto do servidor *Web* adaptativo de alto desempenho JAWS, utilizando a técnica de programação orientada a aspectos. O objetivo foi avaliar a técnica de POA em relação a sua utilidade e vantagens no projeto de sistemas com requisitos de adaptação estática e dinâmica. O estudo demonstrou diversas vantagens que podem ser alcançadas quando projetando um sistema com características de adaptação semelhantes ao JAWS com a técnica de POA, entre elas: separação de interesses; e facilidades de reutilização, evolução estática e configuração dinâmica do sistema. De forma geral, o estudo constatou os benefícios que a técnica de POA pode trazer para requisitos de manutenibilidade em sistemas.

Foram encontradas diferentes alternativas para a construção de hierarquias de aspectos e classes visando atender os diversos interesses de adaptação/configuração do sistema. A presença de diversos padrões no projeto original do JAWS não trouxe complicações para a sua reconstrução usando POA, sendo a maioria deles mantido. Assim, o estudo conclui a possibilidade de utilização conjunta destas duas técnicas de projeto.

8. Referências

[1] D. Coleman, D. Ahs, B. Lowther, P. Oman. "*Using Metrics to Evaluate Software System Maintainability*", IEEE Computer 24, vol. 27, no. 8, pp. 44-49, August-1994.

[2] R. Graddy. "*Successfully Applying Software Metrics*", IEEE Computer, vol. 27, no.9, pp.18-25, September-1994.

[3] C. Guezzi, M. Jazayeri. D. Mandrioli. *Fundamentals of Software Engineering*, Prentice Hall, 1991.

[4] K. Lieberherr, C. Lopes. "*Adaptable and Adaptive Software Workshop Report*", OOPSLA'95.

[5] B. Tekinorgan, M. Askit. "*Adaptability in Object-Oriented Software Development*", ECOOP'96 Workshop Report.

[6] M. Fayad, M. Cline. "*Aspects of Software Adaptability*", Communications of the ACM, Vol. 39, No. 10, pp. 58-59, October-1996.

[7] S. Henry, M. Humphrey. "*Object-Oriented versus Procedural Programming Languages: Effectiveness in Program Maintenance*", JOOP, vol. 6, no. 3, pp.41-49, June 1993.

[8] N. Wilde, P. Matthews, R. Huitt. "*Maintaining Object-Oriented Software*", IEEE Software, Vol. 10, No. 1, pp. 75-80, January-1993.
[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. "*Aspect-Oriented Programming*". In Proceedings of the ECOOP'97, Finland, June-1997.
[10] M. Mezini, K. Lieberherr. "*Adaptive Plug-and-Play Components for Evolutionary Software Development*", in Proceedings of ACM OOPSLA'98, October-1998.
[11] R. Schach. *Classical and Object-Oriented Software Engineering With UML and Java*, McGraw-Hill, 1999.
[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing, 1995.
[13] F. Busschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture*, John Wiley & Sons, 1996.
[14] *Homepage of Subject-Oriented Programming Project*, IBM J. T. Watson Research Center, Yorktown Heights, New York, [http://www.research.ibm.com/sop], July 2000.
[15] K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, 1996.
[16] G. Murphy, R. Walker, E. Baniassad. "*Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming*", IEEE Transactions on Software Engineering, Vol. 25, No. 4, pp. 438-455, July/August-1999.
[17] R. Walker, E. Baniassad, G. Murphy. "*An Initial Assessment of Aspect-Oriented Programming*", in Proceedings of the ICSE '99, Los Angeles, May 1999.
[18] M. Kersten, G. Murphy. "*Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-Oriented Programming*", in Proceedings of ACM OOPSLA'99, Denver, Nov-1999.
[19] M. Lippert, C. Lopes. "*A Study on Exception Detection and Handling Using Aspect-Oriented Programming*", in Proceedings of the ICSE '2000, Limerick, Ireland, June 2000.
[20] E. Quadros, C. Rubira. "*Construção de um Framework para Sistemas Controladores de Trens utilizando Padrões de Projeto e Metapadrões*", Anais do XI SBES, Fortaleza, Out-1997.
[21] D. Schmidt. "*Using Design Patterns to Develop Reusable Object-Oriented Communication Software*", Communications of the ACM, Vol. 38, No. 10, pp. 65-74, October-1995.
[22] N. Islam. "*Customizing System Software Using OO Frameworks*", IEEE Computer, Vol. 30, No. 2, February-1997.
[23] J. Kramer, J. Magee. "*Dynamic Configuration for Distributed Systems*", IEEE Transactions on Software Engineering, Vol. 11, No. 4, April-1985.
[24] J. Hu, D. Schmidt. "*JAWS: A Framework for High-performance Web Servers*" in Domain-Specific Application Frameworks: Frameworks Experience by Industry, John-Wiley, 1999.
[25] P. Tarr, H. Ossher, W. Harrison, S. Sutton. "*N Degrees of Separation: Multi-Dimensional Separation of Concerns*", in Proceedings of the ICSE '99, Los Angeles, May 1999.
[26] AspectJ™ – [http://www.aspectj.org/], July 2000
[27] D. Schmidt, T. Suda. "*An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems*", IEEE/BCS Distributed Systems Engineering Journal, vol. 2., Dec.94.
[28] D. Schmidt. "*Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software*", in Handbook of Programming Languages (P. Salus, ed.), MacMillan Computer Publishing, 1997.
[29] G. Booch, J. Rumbaugh, I. Jacobson. *Unified Modeling Language – User Guide*, Addison- Wesley, 1999.
[30] C. Lopes. "*D: A Language Framework for Distributed Programming*", Ph.D. thesis, Northeastern University, Nov-1997.
[31] G. Kiczales, C. Lopes. *Aspect-Oriented Programming with AspectJ*, Tutorial at OOPSLA'99, Nov. 1999.
[32] D. Schmidt. "*Acceptor and Connector: Design Patterns for Initializing Communication Services*", in Pattern Languages of Program Design (editors: R. Martin, F. Buschmann), Addison-Wesley, 1997.
[33] U. Kulesza, D. Silva. "*Reengenharia do Projeto de Sistemas Adaptáveis usando Técnicas de Orientação a Objetos*", Relatório Técnico, DCC/IME/USP, Agosto-2000.