PROGRAMAÇÃO ORIENTADA A ASPECTOS

Uma Visão Geral

Alexandre Henrique Vieira Soares
Anderson de Rezende Rocha
Flávio Luis Alves
Júlio César Alves

{ahvsoares, undersun, alves, jcalves}@comp.ufla.br

Departamento de Ciência da Computação
Universidade Federal de Lavras

Resumo

Este artigo propõe uma visão geral sobre o paradigma de programação orientada a aspectos, através da definição de aspectos genéricos.

Este paradigma pode ser aplicado a linguagens orientadas a objeto através da implementação das abstrações para a representação de aspectos e de uma ou mais estratégias de implementação do modelo.

Apresenta-se também, nos anexos, uma breve descrição da ferramenta Aspect-J, especializada em programação orientada a aspectos.

Abstract

This paper intends a overview about the aspect oriented programming paradigm, through the definition of generic aspects.

This paradigm can be applied to object-oriented languages by the implementation of the abstractions for aspects and one or more strategies defined in the model.

It presents, as a append, a brief description of Aspect-J tool, that is proper to aspect oriented programming.

Introdução

A engenharia de software e as linguagens de programação coexistem em um relacionamento de suporte mútuo.

A maioria dos processos de desenvolvimento de software da atualidade considera um sistema com unidades cada vez menores de desenvolvimento.

Diz-se que a engenharia de software provê as bases teóricas para o bom desenvolvimento de software, por outro lado as linguagens de programação permitem a abstração de unidades e composição destas de inúmeras formas possíveis de forma que, agrupando-as consiga-se um todo bem definido ou ainda mais que este (sinergia).

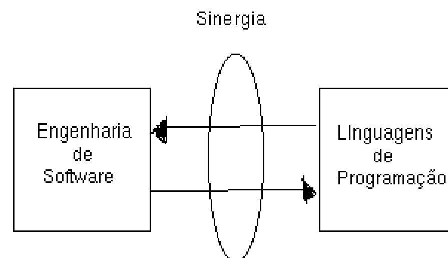

Figura 1

Sabe-se que no desenvolvimento de software existem propriedades que não se enquadram em componentes da decomposição funcional, tais como: tratamento de exceções, restrições de tempo real, distribuição e controle de concorrência. Elas normalmente estão espalhadas em diversos componentes do sistema afetando a performance ou a semântica da aplicação.

Embora elas possam ser visualizadas e analisadas relativamente em separado, sua implementação utilizando linguagens orientadas a objeto ou estruturadas torna-se confusa e seu código encontra-se espalhado através do código da aplicação, dificultando a separação da funcionalidade básica do sistema dessas propriedades.

A programação orientada a aspectos é uma abordagem que permite a separação dessas propriedades ortogonais dos componentes funcionais de uma forma natural e concisa,

utilizando-se de mecanismos de abstração e de composição para a produção de código executável.

Atualmente conhece-se vários problemas de programação em que as técnicas de programação orientadas a objetos ou de programação estruturada não são suficientes para separar claramente todas as decisões de projeto que o programa deve implementar.

Isto se deve ao fato de que as abordagens mais utilizadas concentram-se em encontrar e compor unidades funcionais da decomposição do sistema, enquanto que outras questões importantes não são bem localizadas no projeto funcional.
Exemplos disto podem ser propriedades que envolvem várias unidades funcionais, tais como: sincronização, restrições de tempo, concorrência, distribuição de objetos, persistência, etc.

Quando duas propriedades sendo programadas devem ser compostas de maneira diferente e ainda coordenarem-se é dito que elas são ortogonais entre si.

Aspectos *versus* Componentes

Uma propriedade de um sistema que deve ser implementada pode ser vista como um componente ou como um aspecto:

- A propriedade pode ser vista como um componente se puder ser encapsulada em um procedimento generalizado (objeto, método, procedimento, API). Os componentes tendem a ser unidades da decomposição funcional do sistema. Como exemplos, podem ser citados: contas bancárias, usuários ou mensagens.

- Aspectos não são normalmente unidades da decomposição funcional do sistema, mas sim propriedades que envolvem diversas unidades de um sistema, afetando a semântica dos componentes funcionais sistematicamente.

Pode-se citar vários exemplos de aspectos, tais como: controle de concorrência em operações em uma mesma conta bancária, registro das transações de uma determinada conta, política de segurança de acesso aos usuários de um sistema, restrições de tempo real associadas à entrega de mensagens.

Dada a definição de componentes e aspectos, é possível colocar o objetivo da programação orientada a aspectos: *oferecer suporte para o programador na tarefa de separar claramente os componentes dos aspectos, os componentes entre si e os aspectos entre si, utilizando-se de mecanismos que permitam a abstração e composição destas, produzindo o sistema desejado*.

A programação orientada a aspectos estende outras técnicas de programação (orientada a objetos, estruturada, etc.) que oferecem suporte apenas para separar componentes entre si abstraindo e compondo-os para a produção do sistema desejado.

A implementação de aspectos

Uma implementação baseada no paradigma de programação orientada a aspectos é composta normalmente de:

- Uma linguagem de componentes para a programação de componentes;
- Uma ou mais linguagens de aspectos para a programação de aspectos;
- Um combinador de aspectos (aspect weaver) para a combinação das linguagens;
- Um programa escrito na linguagem de componentes;
- Um ou mais programas escritos na linguagem de aspectos.

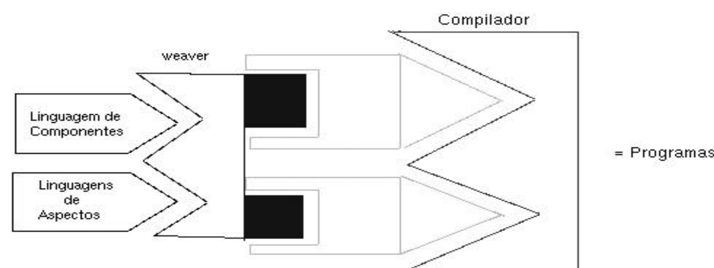

Figura 2

Componentes

Os componentes no contexto da programação orientada a aspectos são abstrações providas pela linguagem que permitem a implementação da funcionalidade do sistema. Logo

procedimentos, funções, classes, objetos, procedimentos são denominados de componentes em programação orientada a aspectos.

Eles originam-se da decomposição funcional de um sistema, independentes de a linguagem ser orientada a objetos ou estruturada. A programação orientada a aspectos não é restrita ou delimitada pela orientação a objetos, ela *estende* o modelo.

Aspectos

Propriedades de um sistema envolvendo diversos componentes funcionais não podem ser expressas utilizando as notações e linguagens atuais de uma maneira bem localizada (tais como: sincronização, interação entre componentes, distribuição, persistência).

Estas propriedades são expressas através de fragmentos de código espalhados por diversos componentes do sistema.

Algumas propriedades que normalmente são vistas como aspectos em relação à funcionalidade básica da aplicação: sincronização de objetos concorrentes, distribuição, tratamento de exceções, coordenação de múltiplos objetos, persistência, serialização, atomicidade, replicação, segurança, visualização, logging, tracing, tolerância à falhas, obtenção de métricas, dentre outras.

Um simples exemplo de aspecto

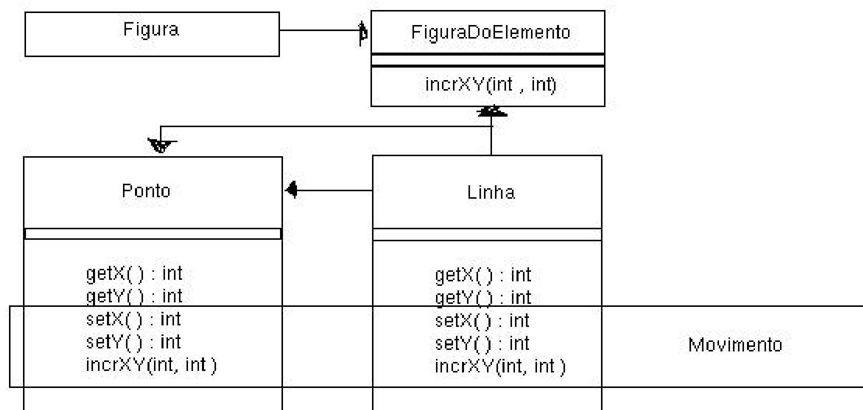

Figura 3

Tem-se na figura acima a definição de dois tipos de figura, ponto e linha. A posição do ponto e da linha é definida através de duas coordenadas (x e y). Deste modo ambas permitem acesso aos valores de x e y (sua posição) através das funções getX() e getY().

Além disso, pode-se modificar a posição de um ponto ou linha, modificando-se a posição na coorenada x (setX), na coordenada y (setY) ou as duas ao mesmo tempo (incrXY). Nota-se então que movimentar um ponto ou uma linha é feito da mesma maneira nos dois tipos de figura.

Com isso, percebe-se que existe um aspecto *movimento*, definido tanto para linha quanto para o ponto. Ou seja, deve-se encapsular o aspecto movimento em um só lugar.

Na verdade, o movimento está definido para uma figura e redefinido para a outra. Assim um tratamento de exceção que diz, por exemplo, que não se pode ter x ou y negativos é definido duas vezes. Da mesma forma para os valores máximos de x e y.

É importante notar também as vantagens desta definiçao do aspecto movimento em relação à reutilização de software. Por exemplo:

> *Suponha que sem a utilização de aspecto, x e y máximos foram definidos como 1000 e foi implementado todo um tratamento de exceção quanto a isso em cada classe.*
> *Suponha agora que alguém pretende reaproveitar este código em outro software. Mas neste as coordenadas máximas serão (1500, 1500). Como a implementação anterior não utilizou a abordagem de aspectos, todo o tratamento de exceção anteriormente previsto para as coordenadas (1000, 1000) deverá ser redefinido para cada classe.*
> *Entretanto se a abordagem de aspecto houvesse sido considerada para a definição de movimento, bastaria redefinir apenas em único local, na definição do aspecto.*

Deste modo percebe-se que a orientação a aspectos propõe sobre a orientação a objetos algo como o que a orientação a objetos propôs sobre a programação estruturada, ou seja, elementos comuns devem ser encapsulados em um só local para uma melhor definição e manutenibilidade.

Linguagem de aspectos

A linguagem de aspectos deve suportar a implementação das propriedades desejadas de forma clara e concisa, fornecendo construções necessárias para que o programador crie estruturas que descrevam o comportamento dos aspectos e definam em que situações eles ocorrem.

Requisitos que devem ser observados na especificação de uma linguagem de aspectos:

- Sua sintaxe deve ser fortemente relacionada com a da linguagem de componentes, de maneira a facilitar o aprendizado e obter maior aceitação por parte dos desenvolvedores;

- A linguagem deve ser projetada para especificar o aspecto de maneira concisa e compacta. Normalmente as linguagens de aspecto são de mais alto nível de abstração que as linguagens de programação de uso geral;

- Sua gramática deve possuir elementos que permitam ao combinador compor os programas escritos usando as linguagens de aspectos e componentes.

Combinador de aspectos

O processo de combinação realizado pelo combinador de aspectos é o produto do cruzamento de um ou mais aspectos com os componentes descritos na linguagem de componentes.

A função do combinador de aspectos é processar a linguagem de aspectos e a de componentes, compondo essas linguagens corretamente a fim de produzir a operação geral desejada.

O projeto de um sistema orientado a aspectos requer o entendimento sobre o que deve ser descrito na linguagem de componentes e de aspectos, bem como as características que são compartilhadas entre elas.

Mecanismos de Composição

Herança, parametrização e chamadas de funções são exemplos de mecanismos de composição. Os mecanismos de composição de aspectos, segundo devem permitir (idealmente) os seguintes requisitos:

- Fraco acoplamento: Os aspectos devem possuir o mínimo possível de ligações com os componentes;

- Adição não invasiva de aspectos ao código existente: é a capacidade de adaptar um componente ou um aspecto sem modificá-lo manualmente.

Tecnologias de implementação para programação orientada a aspectos:

Prover suporte para aspectos envolve duas etapas: implementar abstrações para expressar aspectos e implementar um combinador para compor o código de aspectos com o de componentes.

Para a implementação de abstrações que permitam expressar aspectos de maneira concisa e simples,existem duas abordagens principais:

- A primeira delas consiste em codificar o suporte a aspectos como uma biblioteca convencional.

- A segunda é projetar uma linguagem específica para o aspecto, implementada através de um pré-processador, compilador ou interpretador.

Existem diversas abordagens que se concentram em encapsular propriedades que são ortogonais às unidades funcionais do sistema. Estas abordagens estendem o modelo de programação orientado a objeto, fornecendo mecanismos para a descrição de propriedades que são ortogonais a funcionalidade básica do sistema.

Filtros de composição (*Composition Filters*)

Filtros de composição é uma técnica de programação orientada a aspectos onde diferentes aspectos são expressos em filtros, de maneira declarativa, juntamente com especificações para a transformação de mensagens.

Esta abordagem permite ao desenvolvedor expressar aspectos de uma forma geral através do uso de filtros e possibilita sua composição sem a necessidade da construção de geradores específicos. Os filtros de composição podem ser anexados a objetos de diferentes linguagens orientadas a objeto.

Nesta abordagem as mensagens que chegam a um objeto são avaliadas e, se necessário, manipuladas pelos filtros que atuam sobre aquele objeto. Estes filtros podem ser anexados a linguagens orientadas a objeto, sem a necessidade de modificá-las.

A linguagem de componentes utilizada nesta abordagem é uma linguagem orientada a objetos e a linguagem de aspectos é o mecanismo de filtros de composição. A maioria do processo de combinação ocorre em tempo de execução. Nesta abordagem os pontos de combinação são o envio e recebimento de mensagens por parte de um objeto.

São exemplos de ferramentas/linguagens de suporte a programação orientada a aspectos: HyperJ, QIDL, AOP/ST, linguagem de sincronização de processos, linguagem detracing, AspectJ, D, COOL, RIDL, IL, D2AL, JST, AspectIX.

Abstrações para a representação de aspectos

De forma geral, como o paradigma de orientação a aspectos está ainda em sua *infância*, a abstração de aspectos está atrelada à orientação a objetos. Assim se constrói um conjunto de classes inter-relacionadas que fornecem mecanismos para representar aspectos em uma aplicação.

Objetos destas classes são referenciados e utilizados da mesma forma que outros objetos no sistema, com a particularidade de que não são feitas chamadas para métodos ou propriedades destes pelos componentes da aplicação.

Um aspecto pode estar presente em diversas classes, podendo atuar em diversos eventos do sistema. Ao ser inserido na aplicação, é necessária sua associação com as classes com os quais ele interage.

Cada aspecto possui diversos pontos de atuação (pointcuts) que determinam um conjunto de situações (pontos de combinação) e um conjunto de ações que devem ser tomadas no decorrer de cada situação.

A seguir têm-se algumas definições no escopo da programação orientada a aspectos. Mais sobre estas definições pode ser encontrada nos anexos.

Pointcuts* e *Joinpoints

Para entender o que é um pointcut, é necessário saber primeiro o que é um *joinpoint. Joinpoints* representam pontos bem definidos na execução de um programa. *Joinpoints* típicos podem ser, por exemplo, chamadas a métodos, acessos a membros. *Joinpoints* podem conter outros *joinpoints*.

Pointcut é uma contrução de linguagem que junta um cojunto de *joinpoints* baseando-se em um critério bem definido.

Advice

Agora que o aspecto definiu os pontos a serem linkados é usado o *advice* para completar a implementação. Advice é o trecho de código que é executado antes, depois e simultaneamnete a um *joinpoint.*

É algo como “rode tal código antes de todos os métodos que eu escrever um log”.

Por que usar o paradigma de orientação a aspectos?

Uma vantagem na utilização da orientação a aspectos está na diminuição do tamanho do código dos componentes (visto que uma parte do código fica na definição dos aspectos), diminuindo sua complexidade.

Por estar centralizado em uma única unidade, alterações são muito mais simples, não é preciso reescrever inúmeras classes. É claro que um código mais conciso facilita sua manutenibilidade e reusabilidade.

Por que "ninguém" usa este paradigma?

Apesar de apresentar uma proposta inovadora no desenvolvimento de software pode-se dizer que a Orientação a Aspectos, apesar de sua constante evolução, ainda deixa a desejar em alguns pontos tais como:

- Não há um meio claro para se definir o que deve e o que não deve ser um aspecto em um projeto de software.

- A falta de metodologias ainda é um fator limitante deste paradigma.

Conclusões

A programação orientada a aspectos a cada dia que passa deixa de ser uma grande promessa para fazer parte do dia-a-dia dos desenvolvedores, ainda que existam detalhes que precisam ser melhor estudados, como a dita falta de metodologia .

Pode-se dizer, ainda mais uma vez, que a Orientação a Aspectos propõe uma inovação na abstração do desenvolvimento de softwares como a Orientação a Objetos propôs em relação à programação estruturada.

Referências

ASPECT-J, *Guia Do Programador* . *in* http://aspectj.org/doc/dist/progguide.pdf.

ASPECT-J, *Tutorial*. In http://aspectj.org/doc/dist/tutorial.pdf.

ASPECT-J, in www.aspectj.org.

DEITEL & DEITEL, *Java, How to Program*.

GROSSO, Willian. *Aspect-Oriented Programming & AspectJ*. *In* DrDobbs.com (DDJ.com).

LAUREANO, Eduardo. *Persistence implementation with Aspect-J.* Dissertaçao de mestrado, UFPE.

PRESMANN, *Software Engeneering.*

SOARES, Sérgio. Programação orientada a aspectos em Java, UFPE.

Anexos

O AspectJ

Nesta seção, apresenta-se a linguagem AspectJ, uma extensão orientada a aspectos, de propósito geral, da linguagem Java.

Anatomia de um aspecto

A principal construção em AspectJ é o aspecto. Cada aspecto define uma função específica que pode afetar várias partes de um sistema, como, por exemplo, distribuição. Um aspecto, como uma classe Java, pode definir membros (atributos e métodos) e uma hierarquia de aspectos, através da definição de aspectos especializados.

Join Points

Um *“join point“* é um ponto bem definido no fluxo de execução de um programa. Além de afetar a estrutura estática, um aspecto também pode afetar a estrutura dinâmica de um programa. Isto é possível através da interceptação de pontos no fluxo de execução, chamados "*join points*", e da adição de comportamento antes ou depois dos mesmos, ou ainda através da obtenção do total controle sobre o ponto de execução.

Exemplos de “*join points*” são: invocação e execução de métodos, inicialização de objetos, execução de construtores, tratamento de exceções, acesso e atribuição a atributos, entre outros.

Ainda é possível definir um *"join point"* como resultado da composição de vários "*join points*". Normalmente um aspecto define "*pointcuts*", os quais selecionam "*join points*" e valores nestes *join points* e *advices* que definem o comportamento a ser tomado ao alcançar os *join points* definidos pelo *pointcut*.

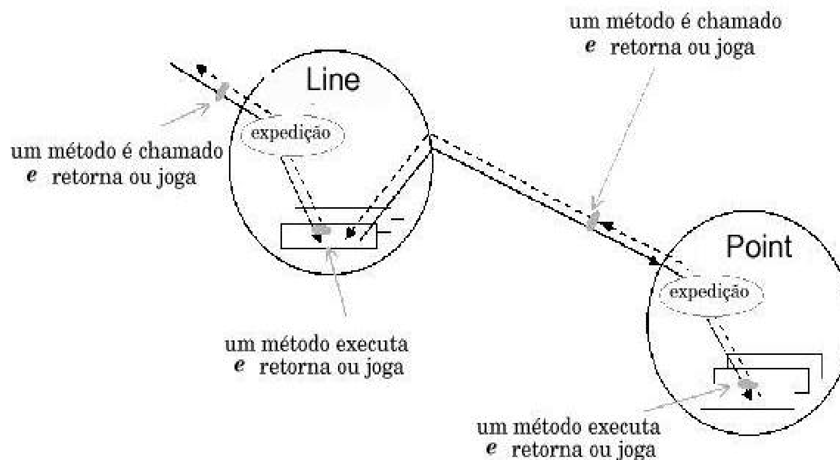

Figura 6: ***Join points*** **de um fluxo de execução**

O primeiro *join point* é a invocação de um método do objeto A, o qual pode retornar com sucesso, ou levantar uma exceção. O próximo "" é a execução deste método, que por sua vez também pode retornar com sucesso ou levantar uma exceção. Durante a execução do método do objeto A é invocado um método do objeto B. A invocação e execução deste método são "*join points*", e da mesma forma que os do objeto A podem retornar com sucesso ou levantar uma exceção.

"*Pointcut*":

"*Pointcuts*" são formados pela composição de "*join points*", através dos operadores && (e), || (ou), e ! (não). Utilizando "*pointcuts*" podemos obter valores de argumentos de métodos, objetos em execução, atributos e excecões dos "*join points*".

Para definir "*pointcuts*", identificando os "*join points*" a serem afetados, utiliza-se 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 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, representandoqualquer 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 em [1].

"Advices":

"*Advices*" são construções que definem código adicional que deverá executar nos "*join points*".

Os *advices* de AspectJ são apresentados a seguir.

Before()	Executa quando o *"join point"* é alcançado, mas imediatamente antes da sua computação
After() returnig()	Executa após a computação com sucesso do *"join point"*
After() throwing()	Executa após a computação sem sucesso do *"join point"*
After()	Executa após a computação do *"join point"*, em qualquer situação
Around()	Executa quando o *"join point"* é alcançado e tem total controle sobre a sua computação

"Static crosscutting":

Como mencionou-se anteriormente, a linguagem AspectJ permite alterar a estrutura estática de um programa através da adição de membros de classe, da alteração a hierarquia de classes ou da substituição de exceções checadas por não checadas.

"Introduction"

O mecanismo que adiciona membros a uma classe é chamado "introduction". Em AspectJ pode-se introduzir métodos concretos ou abstratos, construtores e atributos em uma classe.

A seguir apresenta-se as construções do tipo "introduction" de AspectJ.

Modificadores Tipo PadrãoTipo.Id(Formais){Corpo}	Define um método nos tipos em PadrãoTipo
Abstract Modificadores Tipo PadrãoTipo.Id(Formais);	Define um método abstrato nos tipos em PadrãoTipo
Modificadores PadrãoTipo.new(Formais){Corpo}	Define um construtor nos tipos em PadrãoTipo

A seguir apresenta-se as outras construções em AspectJ que alteram a estrutura estática de um programa.

Declare parents: PadrãoTipo extends ListaTipos;	Declara que os tipos em PadrãoTipo herdam dos tipos em ListaTipos
Declare parents: PadrãoTipo implements ListaTipos;	Declara que os tipos em PadrãoTipo implementam os tipos em ListaTipos
Declare soft: PadrãoTipo:Pointcut;	Declara que qualquer exceção de um tipo em PadrãoTipo que for lançada em qualquer *"join point"* identificado por "Pointcut" será encapsulada em uma exceção não checada

Maiores informações sobre "crosscutting concerns" podem ser encontradas em [1].

Aspectos reusáveis:

AspectJ permite a definição de aspectos abstratos, os quais devem ser estendidos provendo a implementação do componente abstrato. Os componentes abstratos podem ser métodos, como em uma classe Java, e "*pointcuts*", os quais devem ser definidos em um aspecto concerto, permitindo o reuso do comportamento dos aspectos abstratos.

Considerações sobre AspectJ:

A linguagem AspectJ é uma extensão orientada a aspectos da linguagem Java, permitindo assim a programação orientada a aspectos em Java. Entre as desvantagens da linguagem estão a necessidade de se familiarizar com as novas construções de AspectJ e a pouca maturidade do ambiente de desenvolvimento, resultando em um ambiente ainda não muito estável. Entretanto, o ambiente teve uma evolução considerável, estando hoje bem mais estável que nas versões anteriores.

Pontos a melhorar no ambiente de desenvolvimento são o tempo de compilação (weaving) e o tamanho do bytecode gerado.

Outra fraqueza de AspectJ é a sua política de tratamento de exceções. O "advice around" é o único tipo de "advice" que suporta a declaração de uma cláusula “throws”. Nos demais não é possível lançar uma exceção checada que já não seja tratada pelos métodos afetados. A solução dada é utilizar exceções "soft", ou seja, encapsular exceções checadas em uma exceção não checada (exceções do tipo "runtime"). Como este tipo de exceção não obriga o programador a prover o tratamento para a mesma, o programa pode gerar erros inesperados, caso se esqueça de prover o tratamento adequado.

AspectJ provê construtores muito poderosos, que devem ser utilizados com precaução, uma vez que o uso de "wildcards" pode afetar várias partes de um programa, inclusive partes indesejadas.

A definição de um "pointcut" em AspectJ requer a identificação de pontos específicos de um programa. Como estes pontos são identificados através de nomes e tipos de métodos, parâmetros e etc., os aspectos ficam dependentes do sistema, ou da nomenclatura utilizada por ele, dificultando as chances de reuso. Por exemplo, para identificar todas as chamadas de métodos que inserem dados no sistema poderia-se identificar as invocações a métodos com nome "insert", o que obriga a definição de métodos que inserem dados sempre com este nome, e não "register" ou "add". Isto mostra a necessidade de uso de um padrão de nomenclatura, o que também beneficia a legibilidade do sistema.

Outro suporte ao desenvolvimento por parte de AspectJ são extensões para IDEs (ferramentas CASE de programação) bem disseminadas, como Borland JBuilder. Com isto é possível utilizar estes ambientes de programação para desenvolver sistemas com AspectJ.

Estas extensões também permitem visualizar que partes do código são afetadas pelos aspectos. Atualmente estão sendo desenvolvidas extensões para outras IDEs.

A maior vantagem de AspectJ é a possibilidade de implementar funcionalidades em separado da parte funcional do sistema, e automaticamente inserir ou remover tais aspectos do mesmo.

Para remover um aspecto do sistema basta gerar uma nova versão do sistema sem o aspecto que se quer remover. Além disso, para alterar um aspecto, como mudar o protocolo de distribuição de um sistema, basta implementar outro aspecto de distribuição, e passar a usá-lo no processo de recomposição. Com a separação, a legibilidade do código funcional é favorecida, uma vez que não há códigos com diferentes propósitos entrelaçados entre si e com código funcional. Isto também permite a validação precoce dos requisitos funcionais, antes mesmo da implementação de aspectos como persistência, distribuição, e controle de concorrência, tendo assim um desenvolvimento progressivo do sistema.

Alguns exemplos de programas implementados usando AspectJ:

```
class Line {
	private Point p1, p2;

	Point getP1() { return p1; }
	Point getP2() { return p2; }

	void setP1(Point p1) {
		this.p1 = p1;
	}

	void setP2(Point p2) {
		this.p2 = p2;
	}
}

class Point {
	private int x = 0, y = 0;
	int getX() { return x; }
	int getY() { return y; }
	void setX(int x) {
		this.x = x;
	}
	void setY(int y) {
		this.y = y;
	}
}
```

O aspecto SubjectObserverProtocol contém todas as partes genéricas de um protocolo

```
abstract aspect SubjectObserverProtocol {

       abstract pointcut stateChanges(Subject s);

              after(Subject s): stateChanges(s) {
                     for (int i = 0; i < s.getObservers().size(); i++) {

((Observer)s.getObservers().elementAt(i)).update();
                     }
              }

       private Vector Subject.observers = new Vector();
       public void Subject.addObserver(Observer obs) {
              observers.addElement(obs);
              obs.setSubject(this);
       }
       public void Subject.removeObserver(Observer obs) {
              observers.removeElement(obs);
              obs.setSubject(null);
       } public Vector Subject.getObservers() { return observers; }

       private Subject Observer.subject = null;
       public void Observer.setSubject(Subject s) { subject = s; }
       public Subject Observer.getSubject() { return subject; } }
```

Note que este aspecto faz três coisas. Ele define um "*pointcut*" abstrato. Ele define "*advices*" que deveriam ser executados depois do "*pointcut*". E ele introduz estados e comportamentos sobre o Assunto e as interfaces do Observador