Uma Linguagem Extensível para Geração de Linguagens de Domínio Específico Orientadas a Aspectos

Leonardo V. S. Reis[1]

[1]Departamento de Ciência da Computação – Universidade Fereral de Minas Gerais – UFMG
Av. Antônio Carlos, 6627 - Pampulha
31270-010 – Belo Horizonte – MG – Brasil

{leo,bigonha,mariza}@dcc.ufmg.br

Resumo. *Linguagens de programação de domínio específico (*DSL*, do inglês* Domain Specific Languages*) são construídas de forma a atender prioritariamente os requisitos de determinadas áreas de atuação. Exibem construções especialmente projetadas para facilitar a programação e serem mais produtivas em aplicações na área envolvida. O uso de DSLs no desenvolvimento de um sistema deixa o código menos complexo, mais fácil de manter, diminui o tempo de desenvolvimento e facitita o aprendizado.*

A Programação Orientada a Aspectos (POA) é um novo conceito de programação que apresenta recursos adequados para implementação de interesses transversais. Interesses transversais são aqueles que, em geral, atravessam o código de diversas classes de um sistema orientado a objetos. Esses interesses, na maioria das vezes, não são satisfatoriamente encapsulados em unidades de código quando se usa linguagens orientadas a objetos comuns. AspectJ*, definida como um superconjunto de Java com recursos para programação orientada a aspectos, é a mais conhecida linguagem de uso geral que emprega esses conceitos.*

Este trabalho propõe uma nova linguagem orientada a aspectos, definida como uma extensão de AspectJ, que permite que sua sintaxe concreta seja expandida. A semântica dos novos termos poderá ser definida dentro da própria linguagem, de maneira modularizada. Essa nova linguagem é uma ferramenta que poderá ter duas principais utilidades: criar linguagens de domínio específico orientadas a aspectos e constituir um ambiente com facilidade para testes de extensões para a própria linguagem AspectJ.

1. Introdução

Programadores estão habituados a utilizar linguagens de propósito geral, e.g. Java, C, C++, as quais têm um grande poder de expressão. Porém, certos domínios de aplicações são bastante trabalhosos para serem programados usando uma linguagem de propósito geral, diminuindo a produtividade e deixando mais difícil a manutenção do código. Isso ocorre porque as linguagens de propósito geral não são adequadas para expressar tais domínios, para os quais pode haver uma linguagem específica mais apropriada.

Linguagens de Domínio Específico (*DSL*, do inglês *Domain Specific Languages*) expressam o conhecimento específico de um domínio da aplicação, sendo melhores que as linguagens de próposito geral para o domínio em questão [Bravenboer et al. 2005,

Mernik et al. 2005]. DSLs têm a sintaxe bem próxima da notação do domínio da aplicação, o que facilita a interação de especialistas do domínio com os programadores, deixa o código mais fácil de entender e de manter. Em [Fowler 2005], são descritos dois tipos de DSLs, internas e externas. As DSLs internas são aquelas que são escritas dentro da linguagem principal da aplicação, ao contrário das DSLs externas. Existem na literatura [van Wijngaarden 2003, Riehl 2006, Bravenboer and Visser 2004, Batory et al. 1998] vários métodos para embutir DSLs externas na linguagem hospedeira. Porém, para cada DSL que é embutida na linguagem hospedeira, é necessário fazer um novo compilador, i.e., o processo de embutir DSLs não é automático e natural.

Programação Orientada a Linguagens (*LOP*, do inglês *Language Oriented Programming*) foi introduzida por [Ward 1994] como sendo uma nova maneira de se organizar o desenvolvimento de grandes sistemas de software. Fowler descreve LOP como sendo uma maneira de descrever um sistema por meio de DSLs [Fowler 2005]. Segundo Ward, os benefícios de LOP são aumento da produtividade e facilidade de manutenção do código.

Programação Orientada a Aspectos (*AOP*, do inglês *Aspect Oriented Programming*) é um novo paradigma de programação, introduzido em 1996 por Gregor Kiczales [Kiczales et al. 1996]. A principal motivação para a criação desse paradigma é apresentar uma solução adequada para a implementação de interesses transversais. Intereses transversais são funcionalidades que não podem ser adequadamente encapsuladas em classes e ficam espalhadas por todo o sistema.

Para solucionar famílias de problemas específicos, um subconjunto das funcionalidades de uma linguagem orientada a aspectos pode ser suficiente. O mesmo pode ser válido para interesses transversais específicos. A escolha desses subconjuntos pode dar origem a linguagens orientadas a aspectos de domínio específico. As restrições impostas por tais linguagens podem facilitar a utilização das funcionalidades da AOP dentro do domínio escolhido, evitando alguns dos problemas associados ao poder excessivo que é oferecido ao programador, e sem perder o benefício principal, que é a adequada implementação de interesses transversais.

Neste trabalho, pretende-se propor uma extensão para a linguagem AspectJ que a torne adequada para o uso em atividades de LOP. A nova linguagem possibilitará a extensão de sua sintaxe concreta de forma fácil, permitindo a criação de DSLs orientadas a aspectos e constituir um ambiente com facilidade para testes de extensões para a própria linguagem AspectJ. A semântica dos novos termos será dada de forma modularizada.

2. Motivação

O desenvolvimento dos recursos computacionais e o surgimento da indústria de software trouxeram a necessidade de produzir softwares rapidamente e mais complexos. Com o aumento da complexidade dos softwares, surgiram problemas de manutenção, pois a complexidade e o tamanho dos programas aumentaram, o que dificulta encontrar e corrigir erros. Neste contexto, surgiu a necessidade de desenvolver ferramentas que aumentem a produtividade e facilitem a manutenção dos softwares.

Os maiores ganhos em produtividade no desenvolvimento de software, historicamente, foram alcançados por meio da criação de linguagens de alto nível e de compiladores para essas linguagens. O uso de uma linguagem de alto nível, quando comparada

à linguagem de montagem (assembler), permite que os programas escritos sejam menores, simples, fáceis de entender e modificar. A metodologia de Programação Orientada a Linguagens sugere que um ganho adicional de produtividade pode ser alcançado pelo desenvolvimento de uma linguagem orientada ao problema a ser resolvido, que depois é usada na implementação do sistema [Dmitriev 2004].

Em [Ward 1994], são discutidos os problemas encontrados na produção de softwares em larga escala e os benefícios do uso de LOP. Ward advoga que o desenvolvimento de softwares orientados ao domínio da aplicação reduz o tempo de desenvolvimento, facilita a manutenção e o reúso. A abordagem se divide em três etapas:

1. Isolar um problema importante dentre as tarefas a serem realizadas pelo sistema. Definir, formalmente, uma nova linguagem de programação, especificamente orientada ao domínio do problema escolhido. Essa linguagem deve ter nível bem alto de abstração, e oferecer recursos para agilizar a especificação das soluções para esse problema.
2. Usando a linguagem construída, implementar as partes do sistema que estejam relacionadas ao problema escolhido.
3. Implementar um compilador ou interpretador para a nova linguagem.

Os passos descritos acima devem ser executados para quantos problemas se considerar adequados. O código final do sistema utilizará linguagens de programação convencionais e uma ou mais linguagens de domínio específico.

Na metodologia de programação orientada a linguagens, a etapa de construção das novas DSLs é considerada como um "nível médio" no processo de desenvolvimento. Isso possibilita atacar os problemas mais diretamente, e então trabalhar simultaneamente em direção a níveis mais baixos e níveis mais altos de abstração.

Programação Orientada a Objetos introduziu o conceito de classe, permitindo que um programa seja dividido em módulos, aumentado as possibilidades de reúso de código e facilidades de manutenção. No entanto, existem certas características em um programa que não podem ser encapsuladas em uma unidade, ou módulo, e ficam espalhadas por todo o código, denominadas interesses transversais. Exemplos importantes de interesses com essas características são o registro de operações, persistência de dados e controle de transações. AOP foi criada para ser uma solução adequada para implementar esses interesses transversais.

Porém, alguns problemas podem ser associados à utilização de AOP, como a falta de um modelo formalmente definido, a falta de ferramentas adequadas para o desenvolvimento e depuração de código, a captura não intencional de pontos de código por algum conjunto de junção, e efeitos colaterais indesejados quando se renomeia entidades de um programa como classes e métodos. Além disso, o poder oferecido por linguagens orientadas a aspectos permite que sejam violados importantes princípios da orientação a objetos, como a encapsulação. Outro problema associado a linguagens orientadas a aspecto é a complexidade da sintaxe da linguagem, que dificulta seu aprendizado.

Um subconjunto de funcionalidades de uma linguagem orientada a aspectos pode ser suficiente para uma aplicação específica, ou interesses tranversais específicos. Assim, DSLs orientadas a aspectos podem ser desenvolvidas utilizando um subconjunto de funcionalidades das linguagens orientadas a aspectos para atender as necessidades da

aplicação. As restrições impostas por tais DSLs podem facilitar a utilização das funcionalidades da AOP dentro do domínio escolhido, evitando alguns dos problemas associados ao poder excessivo que é oferecido ao programador, e sem perder o benefício principal, que é a adequada implementação de interesses transversais.

Resumindo, o principal objetivo desta seção é mostrar os benefícios do uso de LOP e AOP no desenvolvimento de softwares. Sugere-se que alguns problemas de AOP podem ser solucionados utilizando práticas de LOP, assim espera-se obter um maior ganho de produtividade com a utilização conjunta de LOP e AOP. Portanto, a principal motivação para este trabalho é o aumento da produtividade que pode ser conseguida com o uso dessas duas técnicas.

3. Revisão Bibliográfica

Kiczales [Kiczales et al. 1997] mostra que existem diversos problemas de implementação que não podem ser adequadamente implementados com os paradigmas de programação orientada a objetos ou procedural, de forma que capturem as decisões de projeto. Quando esses problemas são implementados em um desses paradigmas, as decisões de projeto ficam espalhados por todo o código, dificultando o desenvolvimento e a manutenção. Kiczales faz uma análise do porquê de algumas decisões de projeto serem difíceis de implementar usando o paradigma orientado a objeto, e propõe o paradigma orientado a aspectos, assim denominado por ele, para solucionar esses problemas. A linguagem AspectJ foi desenvolvida para incorporar os novos conceitos do paradigma orientado a aspectos [Kiczales et al. 2001a, Kiczales et al. 2001b, Laddad 2003].

Um estudo de caso mostrando como programação orientada a aspectos pode facilitar o desenvolvimento e solucionar problemas associados ao uso de programação orientada a objetos é apresentado por Mendhekar [Mendhekar et al. 1997]. RG é um sistema de processamento de imagem que permite operações sofisticadas via composição de filtros na imagem. Mendhekar afirma que o uso de programação orientado a objetos torna a implementação mais fácil, no entanto há perda de eficiência. Esse problema é contornado utilizando-se programação orientada a aspectos, tornando o desenvolvimento do sistema mais fácil sem perder em eficiência.

Na literatura, encontram-se diversos exemplos de DSLs que foram aplicadas com sucesso. Em [Deursen et al. 2000], encontram-se as principais referências sobre DSLs, discute-se os riscos e benefícios do uso de DSLs e as diversas estratégias de implementação.

Um exemplo de linguagem de domínio específico orientada a aspectos é a linguagem KALA [Fabry et al. 2008]. KALA é uma linguagem de domínio específico orientada a aspectos para fazer gerenciamento de transações. A linguagem KALA foi projetada a fim de separar as especificações das propriedades das transações dos métodos Java. KALA foi testada em diversas aplicações e os autores afirmaram que o uso de KALA nesses sistemas facilitou o desenvolvimento e a manutenção, pois separou da aplicação os detalhes do controle de transações.

Maya é uma extensão da linguagem Java descrita em [Baker and Hsieh 2002], que permite que sua sintaxe e semática seja estendida. Maya usa casamento de padrões para definir macros que alteram a árvore de sintaxe abstrata. As macros são definidas usando

uma nova linguagem denominda *mayan* que define regras de produções da gramática em termos de padrões.

Assim como Maya, XJ [Clark et al. 2008a] é uma extensão da linguagem Java que permite estender sua sintaxe e semântica. Em XJ, novos comandos têm o caractere de escape @ que permite que a semântica dos novos termos seja dada de forma modularizada e em termos de classes, diferente de Maya.

Este trabalho propõe uma extensão de uma linguagem orientada a aspectos que permita estender sua sintaxe concreta, de forma similar a Maya e XJ, com o objetivo de criar DSLs orientadas a aspectos. Existem na literatura diversas abordagens para criar e implementar linguagens de domínio específico orientadas a aspectos. As principais abordagens são citadas e discutidas a seguir:

1. Meta-AspectJ (MAJ) [Huang and Smaragdakis 2006] - MAJ é uma extensão de Java que permite criar programas que geram código em AspectJ, portanto MAJ pode ser usada para implementar linguagens de domínio específico orientadas a aspectos, gerando código das DSLs orientadas a aspectos na linguagem AspectJ. Uma devantagem do uso de MAJ é que as DSLs criadas têm o seu poder de expressão limitado ao poder do AspectJ. O desenvolvimento de novas DSLs orientadas a apectos não é automático, sendo necessário fazer um novo compilador toda vez que uma nova DSL é criada.
2. Usando programas para transformação de código [Bagge and Kalleberg 2006] - Normalmente, a maioria das implementações das DSLs usuais são feitas usando bibliotecas, ou tranformação para chamada de funções de bibliotecas, porém com DSLs orientadas a aspectos o uso de bibliotecas é mais difícil por causa da natureza desse paradigma. Aspectos são semelhantes às linguagens de transformação de código, e desta forma, em [Bagge and Kalleberg 2006] a ferramenta Stratego/XT [Visser 2001a] é usada para fazer uma biblioteca de transformação de código e as DSLs orientadas a aspectos são implementadas utilizando chamadas de funções para essa biblioteca, semelhante ao método MetaBorg usado na implementação de DSLs usuais [Bravenboer et al. 2005, Riehl 2006]. O poder das DSLs orientadas a aspectos que podem ser criadas utilizando esse método estará limitada ao poder de expressão da biblioteca de transformação, que pode ser muito menor do poder que linguagens de propósito geral oferecem, i.e. AspectJ.
3. XAspects [Shonle et al. 2003] - XAspects é um sistema para desenvolver DSLs orientadas a aspectos. Este sistema é baseado em uma arquitetura de plug-in. As DSLs são transformadas em um plug-in pelo compilador XAspects, depois código AspectJ é gerado usando o compilador do AspectJ, e os plug-in fazem análise e geram código AspectJ que depois é costurado pelo compilador do AspectJ.
4. Join Point Selectors [Breuel and Reverbel 2007] - Join Point Selector é um método que permite criar novos pontos de junção mais expressivos, assim podem ser criados pontos de junção que tenham a semântica desejada para as novas DSLs orientadas a aspectos, e o código para costura será feito com o adendo adequado. Porém, esse método restringe a criação de DSLs que podem ser expressadas somente com pontos de junção, as que necessitam de mais características não poderiam ser criadas.
5. Josh [Chiba and Nakagawa 2004] - Josh é similar a Join Point Selectors, sendo uma linguagem orientada a aspectos que permite criar novos pontos de junção.

Josh é baseada na linguagem AspectJ e tem semântica e sintaxe similar, porém não tem todas as características de AspectJ, tem apenas um subconjunto da linguagem. A principal diferença de Josh para Join Point Selector é que Josh permite que pontos de junção sejam criados de forma genérica e permite a declaração de novos "inter type".

6. Trabalhar no compilador - Uma maneira para estender a linguagem AspectJ é trabalhar diretamente em um compilador da linguaguem. Isso não é uma maneira de criar novas DSLs orientadas a aspectos, mas sim de estender a linguagem base. As duas principais abordagens são:
   (a) Compilador abc [Avgustinov et al. 2005] - AspectBench Compiler (abc) é um compilador para a linguagem AspectJ que foi projetado para facilitar a criação e testar novas características na linguagens. Uma extensão da linguagem AspectJ que permite criar pontos de junção para capturar laços é apresentada em [Harbulot and Gurd 2006]. Para estender a linguagem é usado o compilador abc, sendo um bom exemplo para ilustrar essa técnica.
   (b) A Declarative Syntax Definition for AspectJ [Bravenboer et al. 2006b] - Utilizar o compilador abc para estender a linguagem AspectJ tem o inconveniente de necessitar aprender todo o código do compilador, e este é mal documentado, o que dificulta o trabalho. Bravenboer apresenta uma proposta para o front-end do compilador AspectJ [Bravenboer et al. 2006b] que pode ser usada como uma alternativa ao compilador abc para estender a linguagem AspectJ. O analisador sintático é feito usando SGLR [Visser 1997] e gramática definida com o SDF [Heering et al. 1989], e por ser dividida em módulos facilita a extensão da linguagem.

4. Abordagem Proposta

A linguagem que este trabalho pretende definir deve possibilitar a extensão da sintaxe concreta de AspectJ, juntamente com definição da semântica dos novos comandos, de maneira simples e modular. A linguagem proposta deve ter características semelhantes a linguagem XJ [Clark et al. 2008a], porém a principal diferença é que na linguagem XJ não é possível criar DSLs orientadas a aspectos.

Um exemplo esquemático do funcionamento do sistema é apresentado na Figura 1. O sistema recebe, como entradas, a gramática e semântica de uma nova DSL orientada a aspectos *L*, e um programa escrito em AspectJ estendido com *L*. Gera, como saída, um código objeto em AspectJ. Em vez de especificar uma nova DSL, as construções sintáticas de L podem constituir a proposta de uma extensão para a própria linguagem AspectJ, que pode ser facilmente implementada e testada com a ferramenta. É importante ressaltar que tanto a grámatica e semântica de L serão dadas de forma modular na linguagem especificada nesta proposta.

Uma importante vantagem dessa abordagem em relação às outras para criar DSLs orientadas a aspectos e extensões para a linguagem AspectJ é que, na linguagem proposta, o "engenheiro de linguagem", que irá criar as extensões, não precisará utilizar diferentes métodos e ferramentas, o que poderia gerar problemas futuros, caso as tecnologias usadas não fossem estáveis. Tudo poderá ser feito dentro da própria linguagem, aumentado a portabilidade das extensões propostas. A seguir, é apresentado um exemplo para ilustrar como esses objetivos podem ser alcançados.

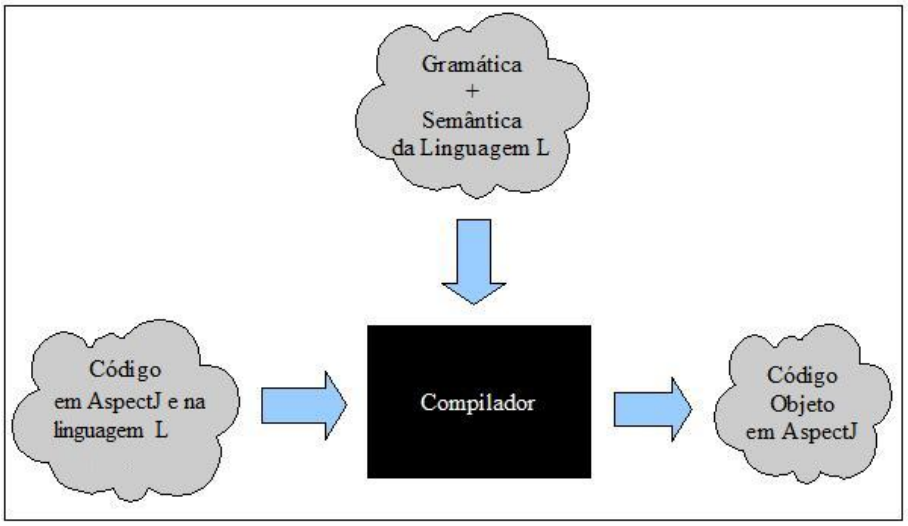

Figura 1. Esquema Mostrando o Funcionamento do Sistema.

Suponha que, em uma determinada aplicação, deseja-se implementar o padrão de projeto *visitor* utilizando a abordagem orientada a aspectos. Neste padrão, todas as classes da estrutura a ser visitada devem receber um método adicional com o formatado apresentado na Figura 2. O método parece o mesmo em todas as classes, mas, na realidade, as chamadas a "visit" são diferentes em cada classe, uma vez que o tipo de "this" é diferente.

```
1 void accept(Visitor v) {
2   v.visit(this);
3 }
```

Figura 2. Método accept exigido pelo padrão visitor.

Suponha que *Base* seja a classe base de toda a estrutura visitada, que contém classes *C1*, *C2*, ... Uma solução, em AspectJ, poderia ser a definição de uma série de *introductions*, como mostra a Figura 3. Observa-se que existe um padrão, que porém não pode ser aproveitado em AspectJ, portanto, para cada classe a ser visitada, deve ser feito um código semelhante.

A linguagem Josh [Chiba and Nakagawa 2004] oferece um mecanismo de *introduction* mais poderoso. Pode-se especificar um tipo de introdução a ser adicionada em todas as subclasses de um determinado tipo, por exemplo. Uma solução para o problema acima, em Josh, teria sintaxe com mostrada na Figura 4. Essa solução é mais sucinta que a de AspectJ, e ainda traz outras vantagens. Por exemplo, se a hierarquia de classes for modificada, acrescentado ou eliminando classes, o código do aspecto em Josh não precisaria ser modificado.

A linguagem proposta neste trabalho deve permitir que um "engenheiro de linguagens" especifique, de maneira simples, clara e eficiente, uma extensão para AspectJ que ofereça os mesmos benefícios que Josh, para o caso de introduções simultâneas em múltiplas classes.

O esboço de solução apresentado a seguir segue a abordagem adotada por XJ

```
aspect VisitorAspectJ {
    void C1.accept(Visitor v) {
        v.visit(this);
    }

    void C2.accept(Visitor v) {
        v.visit(this);
    }

    void C3.accept(Visitor v) {
        v.visit(this);
    }
    ...
}
```

Figura 3. Implementação do Padrão Visitor Usando AspectJ.

```
aspect VisitorAspectJ {
    void Base+.accept(Visitor v) {
        v.visit(this);
    }

}
```

Figura 4. Implementação do Padrão Visitor Usando Josh.

[Clark et al. 2008a], para ilustrar como esse problema poderia ser implementado utilizando a nova linguagem. O novo elemento da linguagem (introduções simultâneas em múltiplas classes) pode ser definido por um tipo especial de classe, que não existe em Java ou AspectJ - uma classe que define uma extensão para a sintaxe concreta da linguagem. Essa classe definiria a sintaxe e a semântica do novo elemento, como mostra a Figura 5. Reunindo toda a definição do novo elemento em uma mesma unidade, pretende-se obter um ganho em modularidade.

Cada ocorrência do novo elemento sintático em um programa dá origem a uma nova instância da classe *MultiIntro*, que identifica o tipo de retorno, o nome da classe base usada e o resto da declaração a ser inserida. Um exemplo de uso da nova construção é apresentado na Figura 6. O tipo de retorno é "void", "Base" é o nome da classe, e "accept(..." forma o resto da declaração (assinatura e bloco de comandos).

No método desugar, todo código fornecido é executado em **tempo de compilação**. O código a ser gerado é a AST construída. O sistema deve oferecer funcionalidades para analisar a estrutura do programa (em forma de AST) e para construir novos elementos da AST. Usando essa abordagem, junto com técnicas de programação gerativa, pode-se ter uma geração de código eficiente e poderosa.

Esse exemplo apresenta uma idéia das facilidades oferecidas pela linguagem a ser definida. A implementação da linguagem deverá se feita por um compilador com funcionalidades incomuns, e será uma importante parte do trabalho a ser desenvolvido, como é descrito no cronograma apresentado neste documento.

Para implementar a linguagem, existem duas possibilidades. A primeira é fazer um novo compilador ou adicionar novas características a um compilador existente para reconhecer a linguagem. A segunda abordagem é usar alguma linguagem que já oferece re-

cursos para modificação da própria sintaxe concreta, e.g. XMF [Clark et al. 2008b], uma linguagem que permite modificar sua sintaxe concreta. Existe uma experiência semelhante usando a linguagem Java, que pode ser encontrada em [Clark et al. 2008a]. Porém, para fazer DSLs orientadas a aspectos é mais complicado do que a abordagem usada para estender a linguagem Java, pois no caso de linguagens orientadas a aspectos existe o processo de costura (weaver) que torna a implementação da proposta mais desafiadora.

Para validar a solução proposta, a nova linguagem será utilizada na implementação de diversas DSLs, como por exemplo, a linguagem orientada a aspectos para sincronização proposta em [Di Iorio et al. 2008]. Espera-se que a implementação da linguagem seja mais fácil e rápido do que usando as ferramentas para transformação de código usadas na implementação da linguagem [Visser 2001b, Visser 2004].

5. Contribuições

Assim como XJ e Maya, a linguagem proposta permite que sua sintaxe concreta seja estendida, possibilitando criar novas DLSs. Porém, diferentemente de XJ e Maya, a linguagem proposta permite criar novas DSLs orientadas a aspectos. A definição da sintaxe e semântica das novas DSLs criadas em XJ é dada de forma modular, sendo uma importante vantagem em relação a Maya, portanto a mesma abordagem será adotada na implementação da linguagem proposta.

A definição da sintaxe e semântica das novas construções serão dadas na própria linguagem proposta, o que pode ser considerada uma vantagem em relação à outras abordagens para criar DSLs. O "projetista de linguagens" não necessita utilizar diferentes métodos e ferramentas, o que poderia gerar problemas futuros, caso as tecnologias usadas não fossem estáveis e aumenta a portabilidade do código.

A linguagem propota tem como principal objetivo permitir que novas DSLs orientadas a aspectos sejam definidas e utilizadas de forma fácil, possibilitando que os usuários tirem proveito das vantagens proporcionadas pelo uso de DSLs no sistema. O poder de expressão da linguagem proposta permitirá ainda que novas extensões para a linguagem AspectJ sejam criadas e testadas facilmente.

- Uso de DSLs e Orientação a Aspectos
  - Aumento da Produtividade
  - Facilidade na Manutenção do Sistema
- Rápida implementação e desenvolvimento de DSLs
- Sintaxe e Semântica das novas construções modularizada
  - Portabilidade do Código
- Sem dependência de várias ferramentas para implementar DSLs
- Permite o uso de um conjunto limitado de características das linguagens orientadas a aspectos suficiente para uma determinada aplicação
- Clareza na sintaxe com DSLs orientadas a aspectos, facilitando o uso e o aprendizado
- Possibilita testes de novas características a linguagens orientadas a aspectos

6. Cronograma

A Tabela 1 mostra o cronograma do projeto. As linhas são as atividades que serão desenvolvidas no decorrer deste trabalho e as colunas representam os meses, 1 para outubro de 2008 até 18, para março de 2010.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	X																	
2		X	X															
3a				X														
3b				X														
3c				X	X													
3c					X													
4						X												
5a							X											
5b							X											
5c								X	X	X	X							
6a												X	X					
6b													X	X				
7															X			
8																X	X	
9																X	X	
10																		X

Tabela 1. *Cronograma do Projeto*

1. Estudo de trabalhos relacionados
2. Proposta inicial da linguagem
3. Testes com a proposta inicial
   a.Seleção de um conjunto significativo de DSALs
   b.Especificação das DSALs selecionadas usando a linguagem (sem teste real, por ausência ainda da implementação da linguagem)
   c.Avaliação
   d.Possíveis alterações a serem conduzidas na proposta da linguagem
4. Elaboração e submissão de artigo com a proposta da linguagem. Possibilidades: SBLP (abril 2009); LA-WASP (julho 2009)
5. Implementação da linguagem
   a.Seleção de ferramentas para o front-end
   b.Seleção de ferramentas para o back-end
   c.Codificação da implementação
6. Testes com a implementação
   a.Testes com o conjunto de DSALs selecionadas, agora com a linguagem implementada
   b.Possíveis alterações a serem conduzidas na proposta da linguagem
7. Elaboração e submissão de artigo com a implementação da linguagem. Possibilidades: workshop DSAL, no congresso ACM-AOSD (janeiro 2010)
8. Elaboração e submissão de artigo com resultado geral do trabalho, incluindo as possíveis alterações. Possibilidade: selecionar revista adequada
9. Escrita do texto da dissertação
10. Defesa

7. Sumário da dissertação

O sumário da dissertação é apresentado abaixo:

1. Introdução
    1.1 Descrição do Problema
    1.2 Solução Proposta
    1.3 Contribuições
    1.4 Estrutura do Documento
2. Revisão Bibliográfica
    2.1 Linguagens de Domínio Específico
    2.2 Programação Orientada a Linguagens
    2.3 Programação Orientada a Aspectos
    2.4 Abordagens para extensão de linguagens orientadas a aspectos
3. Proposta de uma Linguagem Orientada a Aspectos Extensível
4. Implementação da Linguagem Proposta
5. Validação da Proposta
6. Conclusão

8. Conclusão

Programação Orientada a Linguagens e Programação Orientada a Aspectos são áreas de pesquisa recentes e que podem trazer grandes benefícios no desenvolvimento de softwares, aumentando a produtividade e facilitando a manutenção dos sistemas. Uma proposta de uma linguagem extensível que permite a criação de DSLs orientadas a aspectos foi apresentada.

As principais contribuições deste trabalho são o ganho de produtividade e manutenção com o uso de uma linguagem que permite que DSLs sejam criadas e usadas no sistema de forma fácil e rápida. Consegue-se também solucionar alguns problemas do uso de linguagens orientadas a aspectos, visto que elas permitem ao programador violar alguns princípios básicos de orientação a objetos e têm sintaxe complicada. Assim, o uso de DSLs orientadas a aspectos para solucionar problemas específicos da aplicação pode restrigir o poder das linguagens orientada a aspectos, prover uma sintaxe mais clara e próxima do problema e o programador fica mais centrado na aplicação.

Muitas pesquisas estão sendo feitas para se adicionar novas características a linguagens orientadas a aspectos, em especial na linguagem AspectJ, porém os compiladores existentes não são bem documentados e dificultam a implementação e testes de novas características [Bravenboer et al. 2006a]. Portanto a linguagem extensível proposta neste trabalho pode ser usada para testar novas características em linguagens orientadas a aspectos, simulando novas construções propostas.

9. Bibliografia a estudar

Serão estudadas várias DSLs orientadas a aspectos, que depois serão implementadas para validar o sistema desenvolvido. Algumas alternativas e outros trabalhos que estendem linguagens também serão estudadas. Todos os trabalhos identificados para estudos até o presente momento estão listados abaixo:

1. [Jo and Monteiro 2008] - Apresenta uma linguagem de domínio específico para computação paralela
2. [Dinkelaker and Mezini 2008] - Uma extensão de uma linguagem orientada a aspectos que permitir embutir linguagens de domínio específico na sintaxe de adendos.

3. [Timbermont et al. 2008] - Uma linguagem de domínio específico para layout de objetos em máquina virtual
4. [Boix et al. 2008] - Linguagem de domínio específico para implementar ”contrato” em redes móveis Ad hoc
5. [Fabry et al. 2007] - Implementação da linguagem KALA utilizando Reflex

Referências

Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA. ACM.

Bagge, A. H. and Kalleberg, K. T. (2006). Dsal = library+notation: Program transformation for domain-specific aspect languages. In *Proceedings of the Domain-Specific Aspect Languages Workshop*.

Baker, J. and Hsieh, W. C. (2002). Maya: multiple-dispatch syntax extension in java. *SIGPLAN Not.*, 37(5):270–281.

Batory, D., Lofaso, B., and Smaragdakis, Y. (1998). Jts: Tools for implementing domain-specific languages. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 143, Washington, DC, USA. IEEE Computer Society.

Boix, E. G., Cleenewerk, T., Dedecker, J., and Meuter, W. D. (2008). Towards a domain-specific aspect language for leasing in mobile ad hoc networks. In *DSAL '08: Proceedings of the 2008 AOSD workshop on Domain-specific aspect languages*, pages 1–5, New York, NY, USA. ACM.

Bravenboer, M., Groot, R. D., and Visser, E. (2005). Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt. In *In Participants Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering*. Springer Verlag.

Bravenboer, M., Tanter, E., and Visser, E. (2006a). Declarative, formal, and extensible syntax definition for aspectj. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 209–228, New York, NY, USA. ACM.

Bravenboer, M., Tanter, E., and Visser, E. (2006b). Declarative, formal, and extensible syntax definition for AspectJ. A case for scannerless generalized-lr parsing. In Cook, W. R., editor, *Proceedings of the 21th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2006)*, pages 209–228, Portland, Oregon, USA. ACM Press.

Bravenboer, M. and Visser, E. (2004). Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In Vlissides, J. M. and Schmidt, D. C., editors, *OOPSLA*, pages 365–383. ACM.

Breuel, C. and Reverbel, F. (2007). Join point selectors. In *SPLAT '07: Proceedings of the 5th workshop on Software engineering properties of languages and aspect technologies*, page 3, New York, NY, USA. ACM.

Chiba, S. and Nakagawa, K. (2004). Josh: an open aspectj-like language. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 102–111, New York, NY, USA. ACM.

Clark, T., Sammut, P., and Willans, J. (2008a). Beyond Annotations: A Proposal for Extensible Java (XJ). (http://www.ceteva.com/docs/XJ.pdf).

Clark, T., Sammut, P., and Willans, J., editors (2008b). *Superlanguages: Developing Languages and Applications with XMF (First Edition)*. Ceteva.

Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35:26–36.

Di Iorio, V. O., Goulart, C. C., Reis, L. V. S., and Oikawa, M. (2008). An application-specific language for synchronization using aspect-oriented programming concepts. In *Proceedings of the II Latin American Workshop on Aspect-Oriented Software Development*, pages 70–79. Institute of Computing, UNICAMP.

Dinkelaker, T. and Mezini, M. (2008). Dynamically linked domain-specific extensions for advice languages. In *DSAL '08: Proceedings of the 2008 AOSD workshop on Domain-specific aspect languages*, pages 1–7, New York, NY, USA. ACM.

Dmitriev, S. (2004). Language Oriented Programming: The Next Programming Paradigm. *onBoard Online Magazine*, 1.

Fabry, J., Éric Tanter, and D'Hondt, T. (2007). Relax: implementing kala over the reflex aop kernel. In *DSAL '07: Proceedings of the 2nd workshop on Domain specific aspect languages*, page 3, New York, NY, USA. ACM.

Fabry, J., Éric Tanter, and DHondt, T. (2008). Kala: Kernel aspect language for advanced transactions. *Sci. Comput. Program.*, 71(3):165–180.

Fowler, M. (2005). Language Workbenches: The Killer-App for Domain Specific Languages? Last update in June 2005 at http://martinfowler.com/articles/languageWorkbench.html.

Harbulot, B. and Gurd, J. R. (2006). A join point for loops in aspectj. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA. ACM.

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism sdf—reference manual—. *SIGPLAN Not.*, 24(11):43–75.

Huang, S. S. and Smaragdakis, Y. (2006). Easy language extension with meta-aspectj. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 865–868, New York, NY, USA. ACM.

Jo a. L. S. and Monteiro, M. P. (2008). A domain-specific language for parallel and grid computing. In *DSAL '08: Proceedings of the 2008 AOSD workshop on Domain-specific aspect languages*, pages 1–4, New York, NY, USA. ACM.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001a). Getting started with aspectj. *Commun. ACM*, 44(10):59–65.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001b). An overview of AspectJ. *Lecture Notes in Computer Science*, 2072.

Kiczales, G., Irwin, J., Lamping, J., Lopes, C. V., and Maeda, C. (1996). Aspect-oriented programming.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. (1997). Aspect-oriented programming. *ECOOP'97–object-oriented Programming: 11th European Conference, Jyväskylä, Finland, June 9-13, 1997: Proceedings*.

Laddad, R. (2003). *AspectJ in Action*. Manning Publications Company.

Mendhekar, A., Kiczales, G., and Lamping, J. (1997). Rg: A case-study for aspect-oriented programming. Technical report.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344.

Riehl, J. (2006). Assimilating metaborg:: embedding language tools in languages. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 21–28, New York, NY, USA. ACM.

Shonle, M., Lieberherr, K., and Shah, A. (2003). Xaspects: an extensible system for domain-specific aspect languages. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 28–37, New York, NY, USA. ACM.

Timbermont, S., Adams, B., and Haupt, M. (2008). Towards a dsal for object layout in virtual machines. In *DSAL '08: Proceedings of the 2008 AOSD workshop on Domain-specific aspect languages*, pages 1–4, New York, NY, USA. ACM.

van Wijngaarden, J. (2003). Code generation from a domain specific language. designing and implementing complex program transformations. Master's thesis, Utrecht University, Utrecht, The Netherlands. INF/SCR-03-29.

Visser, E. (1997). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam.

Visser, E. (2001a). Stratego: A language for program transformation based on rewriting strategies. system description of stratego 0.5. In *Rewriting Techniques and Applications (RTA 01), volume 2051 of Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag.

Visser, E. (2001b). Stratego: A language for program transformation based on rewriting strategies. system description of stratego 0.5. In *Rewriting Techniques and Applications (RTA 01), volume 2051 of Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag.

Visser, E. (2004). Program transformation with stratego/xt. rules, strategies, tools, and systems in stratego/xt 0.9. Technical Report UU-CS-2004-011, Department of Information and Computing Sciences, Utrecht University.

Ward, M. P. (1994). Language oriented programming. *Software|Concepts and Tools*, 15:147–161.

```
class MultiIntro extends Sugar { // se estende Sugar, é um novo elemento

    // Type, Identifier, Signature, Block e Introduction são classes da AST,
    // definidas em biblioteca padrão da linguagem

    private Type t; // tipo de retorno da introdução
    private Identifier c; // representa o nome da classe base para inserir introduções
    private Signature s; // representa o cabeçalho da declaração a ser inserida
    private Block b; // bloco de comando a ser inserido

    public MultiIntro(Type t, Identifier c, Signature s, Block b) {
      this.t = t; this.c = c; this.s = s; this.b = b;
    }

    @Grammar extends Introduction {
    // esta seção define a sintaxe do novo elemento
    // estende Introduction porque pode aparecer nesse ponto, na gramática da linguagem
      MultiIntro ::=
        t = Type
        c = Identifier
        "+"
        "."
        s = Signature
        b = Block
        { return new MultiIntro(t,c,s,b); }
    }
    // ou seja, a sintax é algo como: Type Identifier "+" "." Signature Block

    // Este método fornece a semântica, na forma de uma estrutura sintática
    // que é o resultado da "compilação" do novo elemento.
    // Recebe o contexto (tabela de símbolos) onde o elemento é utilizado
    public AST de sugar(Context ctx) {
      Class class = "classe com o nome c";
      Class sub[] = "suclasses de class no sistema";
      ListIntro list = new ListIntro();
      for (Class x : sub) {
        Introduction i = t + " " + x + "." + s + b;
        list.add(i);
      }
      return list;
    }

}
```

Figura 5. Definição da Sintaxe e Semântica do Novo Comando.

```
void Base+. accept(Visitor v) {
  v.visit(this);
}
```

Figura 6. Exemplo de Uso do Novo Comando Definido