

LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO DO RIO GRANDE DO
NORTE

AGOSTINHO CAMPOS NETO

DESENVOLVENDO DOMAIN SPECIFIC LANGUAGES NA PLATAFORMA JAVA

Natal/RN

2008

AGOSTINHO CAMPOS NETO

DESENVOLVENDO DOMAIN SPECIFIC LANGUAGES NA PLATAFORMA JAVA

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte como requisito obrigatório para obtenção do grau de Especialista em Sistemas Corporativos.

Orientador: David Ricardo do Vale Pereira, M. Sc.

Gleydson de Azevedo Ferreira Lima, M. Sc.

Membro Examinador

Natal/RN

2008

AGOSTINHO CAMPOS NETO

DESENVOLVENDO DOMAIN SPECIFIC LANGUAGES NA PLATAFORMA JAVA

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte, aprovada em ____ de ____ de _____, pela banca examinadora composta pelos seguintes membros:

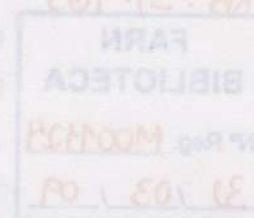
David Ricardo do Vale Pereira

David Ricardo do Vale Pereira, M. Sc.

Orientador

Gleydson de Azevedo Ferreira Lima, M. Sc.

Membro Examinador



RB: 31.963

FARN
BIBLIOTECA

N° Reg. M001124

31 / 03 / 09

004.42

C21d

2008

ex. 01

RESUMO

Frequentemente, desenvolvedores de software precisam criar sistemas que resolvam problemas em uma ampla variedade de domínios. Geralmente o desenvolvedor escolhe uma linguagem de programação genérica GPL (*General Purpose Language*), a partir de então, ele cria abstrações sobre o domínio no qual está trabalhando com a linguagem que é utilizada. Uma maneira de melhorar a forma como se trabalha com o domínio é utilizar linguagens específicas de domínio (*in English, Domain Specific Language - DSL*). Uma DSL é projetada para um dado domínio em particular de forma que o usuário da DSL possa utilizá-la de forma que resolva mais facilmente os problemas de um dado domínio. Apesar de qualquer linguagem de programação poder criar DSLs, quanto mais flexível e com suporte a meta-programação uma linguagem de programação for, mais simples é a criação de DSLs. As linguagens que atualmente são chamadas de linguagens de script ou linguagens dinâmicas são excelentes ferramentas para o projeto de DSLs. Neste trabalho serão apresentados vários pontos importantes sobre o projeto de DSLs e como as mesmas podem auxiliar o desenvolvedor, permitindo que o mesmo expresse em código sentenças mais naturais, fluentes e totalmente voltadas ao domínio que está sendo trabalhado do que uma linguagem genérica permitiria.

Palavras-chave: DSL, Linguagens dinâmicas, Linguagens de script, Java, Groovy.

*Uma linguagem que não afeta a maneira que
você pensa sobre programação, não vale a
pena ser aprendida.
(Alan Perlis).*

RESUMO

Frequentemente, desenvolvedores de software precisam criar sistemas que resolvam problemas nos mais variados domínios. Geralmente o desenvolvedor escolhe uma linguagem de programação genérica GPL (*General Purpose Language*), a partir de então, ele cria abstrações sobre o domínio no qual está trabalhando com a linguagem que é utilizada. Uma maneira de melhorar a forma como se trabalha com o domínio é utilizar linguagens específicas de domínio (em inglês, *Domain Specific Language* - DSL). Uma DSL é projetada para um dado domínio em particular de forma que o usuário da DSL possa utiliza-la de forma que resolva mais facilmente os problemas de um dado domínio. Apesar de qualquer linguagem de programação poder criar DSLs, quanto mais flexível e com suporte a meta-programação uma linguagem de programação for, mais simples é a criação de DSLs. As linguagens que atualmente são chamadas de linguagens de *script* ou linguagens dinâmicas são excelentes ferramentas para o projeto de DSLs. Nesse trabalho serão apresentados vários pontos importantes sobre o projeto de DSLs e como as mesmas podem auxiliar o desenvolvedor, permitindo que o mesmo expresse em código sentenças mais naturais, fluentes e totalmente voltadas ao domínio que está sendo trabalhado do que uma linguagem genérica permitiria.

Palavras-chave: DSL. Linguagens dinâmicas. Linguagens de script. Java. Groovy.

ABSTRACT

Often, software developers need to create systems that will solve problems in the most varied fields. Usually the developer chooses a programming language generic GPL (General Purpose Language), since then, it creates abstractions on the field in which it is working with the language we use. One way to improve the way it works in the field is using the domain specific languages (English, Domain Specific Language - DSL). A DSL is designed for a specific area in particular so that the user's DSL can use it in order to solve the problems more easily in a given area. Although any programming language to create DSLs, the more flexible and with support for meta-programming is a programming language, simpler is the creation of DSLs. The languages that are currently calls from the script language or languages dynamics are excellent tools for the design of DSLs. This work will be presented several important points about the design of DSLs and how they can help the developer, thus enabling it to express judgments in code more natural, fluent and fully geared to the area being worked as a generic language would allow.

Key Words: DSL. Dynamic languages. Script languages. Java. Groovy.

LISTA DE ILUSTRAÇÕES

Figura 01 – Linguagens e suas tipagens.....	15
Figura 02 - Utilizando <i>closures</i> em Groovy	16
Figura 03 - Outro exemplo de <i>closures</i> em Groovy.....	17
Figura 04 - Comportamento <i>Duck Typing</i>	17
Figura 05 – Plataforma Java.....	19
Figura 06 – Arquitetura básica de um compilador.....	21
Figura 07 – Arquitetura básica de um interpretador.....	22
Figura 08 - Captura de erro em tempo de execução.....	22
Figura 09 - Exemplo de DSL em <i>Ruby on Rails</i>	24
Figura 10 - Groovy é uma linguagem expressiva.....	26
Figura 11 - Exemplo de <i>Fluent Interface</i> em Java.....	27
Figura 12 – <i>Semantic Model</i> sendo populado a partir de uma DSL.....	28
Figura 13 - Implementação do padrão <i>Expression Builder</i>	29
Figura 14 - Implementação do padrão <i>Expression Builder</i> , continuação.....	30
Figura 15 - Utilizando a <i>Fluent Interface</i> criada anteriormente.....	31
Figura 16 - Listagem de código usando <i>Method Chaining</i>	31
Figura 17 - Maneira convencional de usar SQL em Java.....	33
Figura 18 - Usando uma DSL para trabalhar com SQL.....	33
Figura 19 - Interface com os métodos da API.....	34
Figura 20 - Classe que implementa a DSL.....	34
Figura 21 - Classe que implementa a DSL, continuação.....	35
Figura 22 - Usando <i>software</i> de terminal Curl.....	36
Figura 23 - Aplicação <i>client</i> acessando API Delicious através de uma DSL.....	37
Figura 24 - Implementação da DSL para acesso REST.....	37
Figura 25 - Implementação do acesso ao serviço REST.....	38
Figura 26 - Implementação do acesso ao serviço REST, continuação.....	39
Figura 27 - Implementação do acesso ao serviço REST, continuação.....	40
Figura 28 - Implementação do acesso ao serviço REST, continuação.....	41

Figura 29 – Quando um método é invocado em Groovy.....	42
Figura 30 - Método adicionado a classe Integer.....	42
Figura 31 - DSL para converter temperaturas.....	43
Figura 32 - Utilizando a DSL para verificar sintomas (Resfriado).....	44
Figura 33 - Utilizando a DSL para verificar sintomas (Sarampo).....	44
Figura 34 - Codificação da DSL para verificar sintomas.....	44
Figura 35 - Codificação da DSL para verificar sintomas, continuação.....	45

LISTA DE TABELAS

Tabela I - API REST do Delicious.....	36
1 ORGANIZAÇÃO DO TRABALHO.....	11
2 OBJETIVOS.....	12
2.1 GERAL.....	12
2.2 ESPECÍFICOS.....	12
3 REFERENCIAL TEÓRICO.....	13
3.1 PORQUE LINGUAGENS DINÂMICAS IMPORTAM.....	13
3.2 PRINCIPAIS CARACTERÍSTICAS DE LINGUAGENS DINÂMICAS.....	14
3.2.1 Tipagem Dinâmica.....	14
3.2.2 Closures.....	16
3.2.3 Duck Typing.....	17
4 MÁQUINAS VIRTUAIS E LINGUAGENS DINÂMICAS.....	18
5 COMPILADORES E INTERPRETADORES.....	20
6 INTRODUÇÃO A DSLs.....	23
6.1 DSLs EXTERNAS.....	23
6.2 DSLs INTERNAS.....	24
7 FLUENT INTERFACES.....	26
8 PADRÕES PARA DSLs.....	28
8.1 SEMANTIC MODEL.....	28
8.2 EXPRESSION BUILDER.....	29
8.3 METHOD CHAINING.....	31
9 DESENVOLVENDO DSLs.....	32
9.1 EXEMPLO 1: DSL PARA FACILITAR COMANDOS SQL.....	32
9.2 EXEMPLO 2: DSL PARA TRABALHAR COM SERVIÇOS RESTFUL.....	35
9.3 EXEMPLO 3: DSL EM GROOVY USANDO META-OBJECT PROTOCOL.....	41
9.4 EXEMPLO 4: DSL EM GROOVY PARA DETECTAR DOENÇAS.....	43
10 LIMITES NA UTILIZAÇÃO DE DSLs.....	47
11 CONCLUSÕES.....	48
REFERÊNCIAS.....	49

SUMÁRIO

O termo Domain Specific Language (DSL) vem se tornando cada vez mais popular na comunidade de desenvolvimento de software. DSLs nada mais são do que linguagens de programação que são projetadas para resolver problemas específicos de uma determinada área de conhecimento.

1 INTRODUÇÃO A DOMAIN SPECIFIC LANGUAGES.....	11
1.1 ORGANIZAÇÃO DO TRABALHO.....	11
2 OBJETIVOS.....	12
2.1 GERAL.....	12
2.2 ESPECÍFICOS.....	12
3 REFERENCIAL TEÓRICO.....	13
3.1 PORQUE LINGUAGENS DINÂMICAS IMPORTAM.....	13
3.2 PRINCIPAIS CARACTERÍSTICAS DE LINGUAGENS DINÂMICAS.....	14
3.2.1 Tipagem Dinâmica.....	14
3.2.2 Closures.....	16
3.2.3 Duck Typing.....	17
4 MÁQUINAS VIRTUAIS E LINGUAGENS DINÂMICAS.....	18
5 COMPILADORES E INTERPRETADORES.....	20
6 INTRODUÇÃO A DSLs.....	23
6.1 DSLs EXTERNAS.....	23
6.2 DSLs INTERNAS.....	24
7 FLUENT INTERFACES.....	26
8 PADRÕES PARA DSLs.....	28
8.1 SEMANTIC MODEL.....	28
8.2 EXPRESSION BUILDER.....	29
8.3 METHOD CHAINING.....	31
9 DESENVOLVENDO DSLs.....	32
9.1 EXEMPLO 1: DSL PARA FACILITAR COMANDOS SQL.....	32
9.2 EXEMPLO 2: DSL PARA TRABALHAR COM SERVIÇOS RESTFUL.....	35
9.3 EXEMPLO 3: DSL EM GROOVY USANDO META-OBJECT PROTOCOL.....	41
9.4 EXEMPLO 4: DSL EM GROOVY PARA DETECTAR DOENÇAS.....	43
10 LIMITES NA UTILIZAÇÃO DE DSLS	47
11 CONCLUSÕES.....	48
REFERÊNCIAS.....	49

1 INTRODUÇÃO A DOMAIN SPECIFIC LANGUAGES

O termo *Domain Specific Language* (DSL) vem se tornando cada vez mais popular na área de desenvolvimento de software nos últimos anos. Uma DSL nada mais é que uma linguagem projetada para resolver problemas de um domínio específico, ao invés de ser uma linguagem de uso geral, como Java ou C#.

O conceito de DSLs não é novo, linguagens específicas de domínio são utilizadas a bastante tempo. Hoje em dia existem DSLs de vários tipos, como por exemplo o CSS (*Cascading Style Sheet*) para formatação de folhas de estilo em páginas *web*, a SQL (*Structured Query Language*) para se trabalhar com banco de dados e a ferramenta *Ant* para automatização de tarefas na plataforma Java.

Esse trabalho tem como objetivo apresentar de forma sucinta como as DSLs estão inseridas no atual contexto do desenvolvimento de *software*, quais são suas vantagens, desvantagens, como utilizar alguns de seus padrões e como desenvolver DSLs com as linguagens Java e Groovy.

1.1 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado da seguinte maneira: no capítulo 2 serão apresentados o objetivo geral e os objetivos específicos do trabalho. Do capítulo 3 até o capítulo 4 é apresentado uma visão geral sobre linguagens dinâmicas e porque são importantes. Do capítulo 5 até o capítulo 11 é dedicado a DSLs, falando das vantagens, desvantagens e como é criar DSLs em Java e Groovy.

2 OBJETIVOS

O objetivo deste trabalho é explicar como DSLs podem ser desenvolvidas e incorporadas no dia-a-dia do desenvolvimento de software.

2.1 GERAL

Em várias ocasiões a utilização da ferramenta certa pode poupar o desenvolvedor de trabalho desnecessário. DSLs buscam resolver problemas de um domínio específico, são projetadas com um nível de abstração geralmente para simplificar tarefas. Nesse contexto, esse trabalho vai apresentar as vantagens do desenvolvimento de DSLs e como desenvolvê-las com exemplos na linguagem Java e Groovy.

2.2 ESPECÍFICOS

- Motivações para o uso de linguagens dinâmicas;
- Motivações para o uso de DSLs;
- Desenvolver DSLs em Java;
- Desenvolver DSLs em Groovy;

3 REFERENCIAL TEÓRICO

Para desenvolver este trabalho é necessário recorrer a alguns conceitos e tecnologias com o objetivo de esclarecer e contextualizar o desenvolvimento de software da perspectiva de DSLs. Neste capítulo serão apresentados alguns desses conceitos.

3.1 PORQUE LINGUAGENS DINÂMICAS IMPORTAM

Atualmente existem dezenas de linguagens dinâmicas, geralmente essas linguagens tem em comum uma sintaxe simplificada mas com alto nível de abstração, são interpretadas e possuem tipagem dinâmica. A maioria delas seguem o conceito “menos é mais”, ou seja, por ter um sintaxe compacta, com pouco código se é capaz de expressar muita coisa no *software*.

As linguagens dinâmicas vem se tornando cada vez mais populares por vários motivos, dentre eles podemos citar: existência de excelentes *frameworks*, capacidade de metaprogramação, sintaxe extremamente flexível e simplificada, que proporciona um considerável ganho de produtividade.

Na indústria de *software* o desenvolvimento geralmente é feito com as linguagens compiladas e de tipagem estática, como C# ou Java, porém nem sempre essas linguagens são as melhores ferramentas para determinado domínio. Isso ocorre porque o paradigma da linguagem influencia na sua sintaxe, que por sua vez afeta o modo como o desenvolvedor programa. Assim, algumas linguagens podem ser mais apropriadas do que outras para determinados trabalhos.

É fato que no ramo do desenvolvimento de *software* o custo de manutenção do mesmo é extremamente alto. Assim, se uma linguagem de programação permite você projetar *software* com menos código de forma legível e elegante (e a maioria das linguagens dinâmicas permitem isso), isso terá um impacto positivo e significativo na manutenção do mesmo.

Usando as capacidades de uma linguagem dinâmica é possível ser mais produtivo em vários aspectos. Por exemplo, criar um alto nível de abstração em pouco tempo e criar aplicativos funcionais de forma mais rápida (pois existe pouca burocracia por parte da

linguagem) e desse forma obter um rápido *feedback* pela equipe do cliente (BECK, 2000), por exemplo.

O código de uma linguagem dinâmica reduz a complexidade de seu código quando comparado a C++, Java ou C#. Percebe-se claramente uma sintaxe mais simples. Além da sintaxe propriamente dita, os *frameworks* das linguagens dinâmicas tendem a serem mais simples de usar do que os *frameworks* “convencionais”, por exemplo, o desenvolvimento de um aplicativo Java para Web, geralmente é necessário que o desenvolvedor crie toda uma infra-estrutura necessária em XML (do inglês, *Extensible Markup Language*) de forma que consiga juntar os “pedaços” do aplicativo como um todo (AKITA, 2006). No lado oposto, um *framework* como o Ruby on Rails seguem o padrão de *conversion over configuration* (OLSEN, 2007), tornando-se até certo ponto inteligente o suficiente para acelerar e simplificar a maior parte do trabalho repetitivo e propenso a erros do desenvolvedor.

A plataforma Java bem como .NET é muito utilizada hoje em dia na indústria de *software*, tornaram-se praticamente um padrão no mercado. O lado negativo desse atual estado na indústria é porque muitos desenvolvedores não percebem que nem sempre a linguagem principal que eles utilizam é a ferramenta certa para dado problema em particular (VENNERS, 2000). Atualmente é possível integrar linguagens e obter o melhor de cada uma, como será visto posteriormente nesse trabalho.

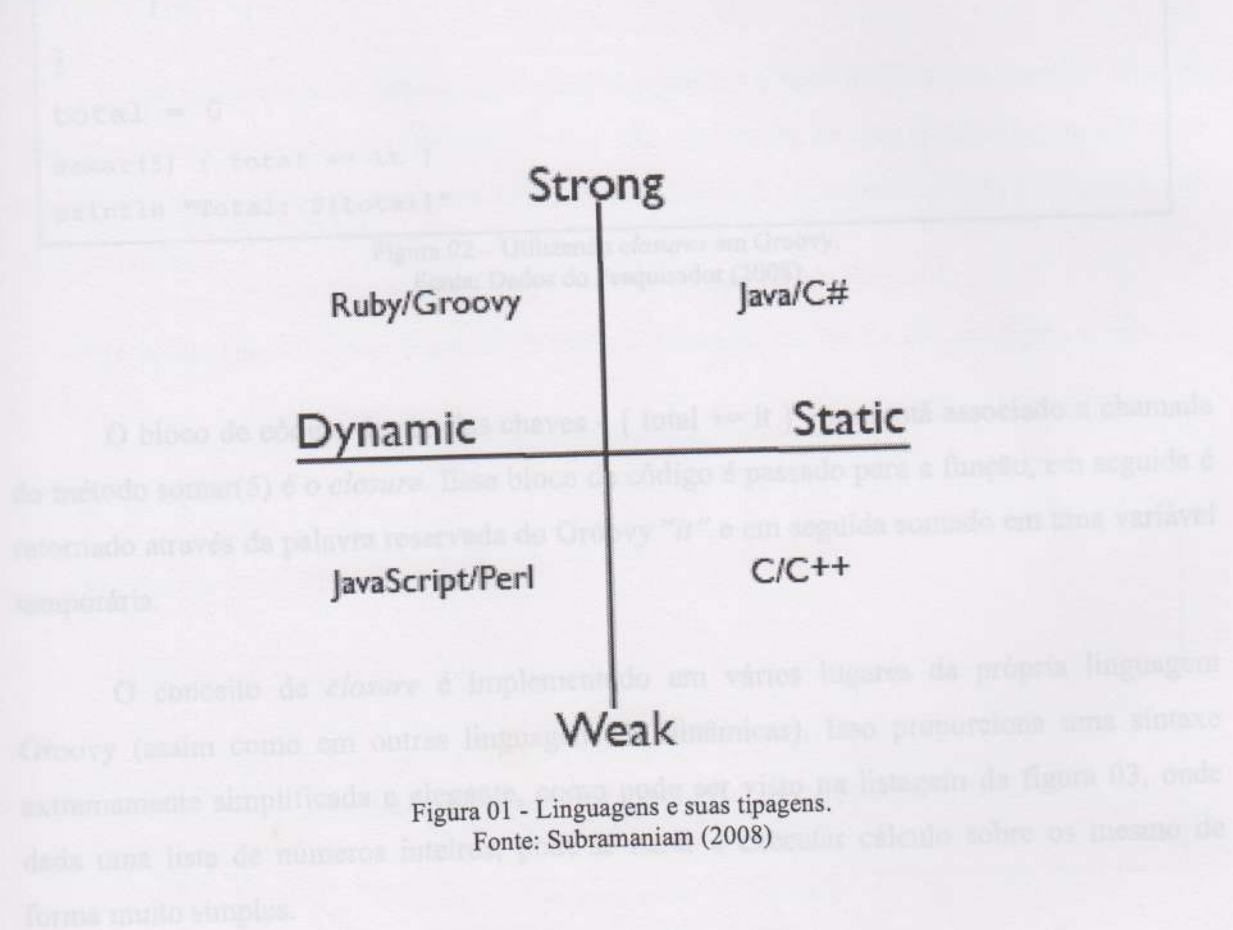
3.2 PRINCIPAIS CARACTERÍSTICAS DE LINGUAGENS DINÂMICAS

3.2.1 Tipagem Dinâmica

Comumente as linguagens de programação são divididas entre linguagens de tipagem estática e tipagem dinâmica. Linguagens como Java, C# ou C++ são linguagens que possuem tipagem estática, isso significa dizer que o compilador exige que o desenvolvedor especifique o tipo da variável antes dela poder ser usada, assegurando que em tempo de execução aquela variável vai aceitar apenas aquele tipo. A tipagem estática implica dizer que caso algum tipo de dado seja passado para uma variável em tempo de execução que não seja comportado pelo tipo que tenha sido especificado em tempo de compilação, o compilador irá detectar um erro.

Grande parte das linguagens dinâmicas implementam o modelo de tipagem dinâmica, isso permite se programar sem especificar um tipo de dado específico para as variáveis em tempo de desenvolvimento. Dessa forma, o valor associado a variável vai depender do contexto. Também existem linguagens dinâmicas que permitem o desenvolvedor escolher entre especificar ou não um tipo de dado quando estiver codificando, por exemplo, Groovy. Uma das vantagens da tipagem dinâmica é a produtividade trazida enquanto se desenvolve, por exemplo, é possível escrever métodos de forma que o desenvolvedor não precise se preocupar naquele momento em detalhes específicos de tipagem, pois em tempo de execução isso irá ser resolvido, automaticamente.

Algo importante a se notar é que tipagem dinâmica não significa dizer tipagem fraca, ou seja, linguagens de tipagem dinâmica e tipagem forte não permitem a tentativa de atribuição de valores inteiros a uma variável do tipo *String*, algo que seria permitido em uma linguagem que seguisse a filosofia de tipagem fraca. A figura 01 exemplifica o assunto.



3.2.2 Closures

Um dos recursos mais interessantes em linguagens dinâmicas são o conceito de *closure*. Essencialmente um *closure* é um bloco de código que pode ser passado como o argumento para outra função (FOWLER, 2004a).

Ou seja, basicamente uma *closure* é uma função (ou bloco de código) enviado como argumento para o corpo de outra função. A figura 02 demonstra esse conceito implementado na linguagem Groovy.

```
def somar(n, meuBloco) {
    for (int i = 1; i < n; i++) {
        meuBloco(i)
    }
}

total = 0
somar(5) { total += it }
println "Total: ${total}"
```

Figura 02 – Utilizando *closures* em Groovy.
Fonte: Dados do Pesquisador (2008)

O bloco de código dentro das chaves - { total += it } - que está associado a chamada do método somar(5) é o *closure*. Esse bloco de código é passado para a função, em seguida é retornado através da palavra reservada do Groovy “it” e em seguida somado em uma variável temporária.

O conceito de *closure* é implementado em vários lugares da própria linguagem Groovy (assim como em outras linguagens de dinâmicas). Isso proporciona uma sintaxe extremamente simplificada e elegante, como pode ser visto na listagem da figura 03, onde dada uma lista de números inteiros, pode-se iterar e executar cálculo sobre os mesmo de forma muito simples.

```
listaNumeros = [1, 3, 4, 1, 8]
total = 0
listaNumeros.each { total += it }
println "Total is ${total}"
```

Figura 03 – Outro exemplo de *closures* em Groovy.

Fonte: Dados do Pesquisador (2008)

3.2.3 Duck Typing

A não declaração de um tipo para uma variável é um dos recursos que mais chama a atenção numa linguagem dinâmica. Esse recurso é geralmente conhecido por “duck typing”, que é basicamente o princípio: “se parece um pato, então se comporta como um pato”. Em Ruby, por exemplo, isso é levado ao máximo e o desenvolvedor não precisa declarar o tipo para uma variável, enquanto em Groovy o desenvolvedor possui a liberdade de escolher se vai ou não declarar o tipo.

Existe um acalorado debate sobre qual abordagem é mais efetiva (declarar ou não um tipo), por exemplo, muitos dos que defendem a não declaração de tipo acreditam que são mais produtivos e que a detecção de erros em tempo compilação é capturada com a utilização de testes unitários.

O estilo *Duck Typing* é muito simples, a listagem da figura 04 apresenta como isso funciona.

```
variavelFoo1 = "algum texto...sou String"
variavelFoo2 = 150
variavelFoo3 = 35.8

puts variavelFoo1.class # comporta-se como tipo String
puts variavelFoo2.class # comporta-se como tipo Fixnum
puts variavelFoo3.class # comporta-se como tipo Float
```

Figura 04 – Comportamento *Duck Typing*.

Fonte: Dados do Pesquisador (2008)

4 MÁQUINAS VIRTUAIS E LINGUAGENS DINÂMICAS

As principais linguagens de programação moderna, como .NET e Java, utilizam o conceito de máquina virtual (VENNERS, 2000) como a peça fundamental das suas respectivas plataformas. Por exemplo, um código escrito em Java é compilado para um *byte-code* intermediário, que é interpretado pela *Java Virtual Machine* (JVM). Uma máquina virtual serve como uma camada de abstração entre o código escrito e o sistema operacional. Esse mesmo conceito é utilizado na plataforma .NET. A principal vantagem de uma plataforma que segue o modelo de máquina virtual é a portabilidade de código sem recompilação.

A linguagem Java é uma linguagem híbrida (compilada e interpretada), como pode ser visto na figura 05. Linguagens dinâmicas modernas, como Python por exemplo, seguem o mesmo conceito, no caso de Python, existe uma máquina virtual chamada *Python Virtual Machine* (PVM). Também existe o fato de várias linguagens dinâmicas terem sido portadas para a JVM, como JRuby, JPython, Scala e Groovy. Dessa forma, pode-se utilizar várias linguagens sobre uma mesma máquina virtual.

A comunidade de desenvolvedores acordou para o fato de que as linguagens dinâmicas podem ser produtivas no dia a dia e a integração com outras plataformas como Java ou .NET podem ser benéficas para o desenvolvimento como um todo.

Hoje em dia é possível observar que o mercado começa a buscar profissionais multidisciplinares, que saibam trabalhar com duas ou até três linguagens de programação, esse profissionais buscam seguir o princípio: “usar a ferramenta certa para o trabalho certo”, ou seja, procuram ser agnósticos em relação a tecnologia.

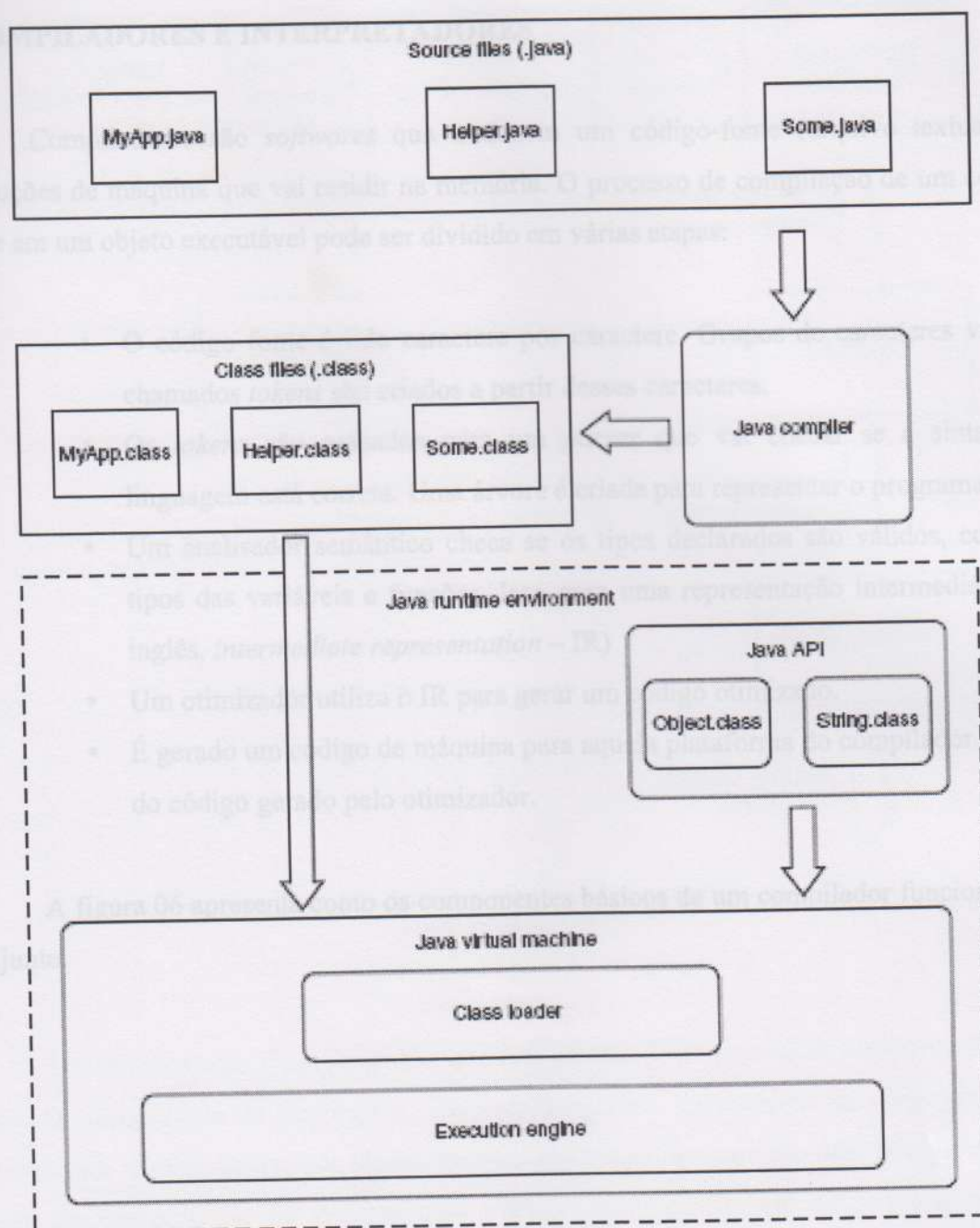


Figura 05 - Plataforma Java.
Fonte: Bosanac (2007)

5 COMPILADORES E INTERPRETADORES

Compiladores são *softwares* que traduzem um código-fonte (arquivo textual) em instruções de máquina que vai residir na memória. O processo de compilação de um código-fonte em um objeto executável pode ser dividido em várias etapas:

- O código fonte é lido caractere por caractere. Grupos de caracteres válidos, chamados *tokens* são criados a partir desses caracteres.
- Os *tokens* são passados para um *parser* que vai checar se a sintaxe da linguagem está correta. Uma árvore é criada para representar o programa.
- Um analisador semântico checa se os tipos declarados são válidos, como os tipos das variáveis e funções. Isso gera uma representação intermediária (do inglês, *intermediate representation* – IR)
- Um otimizador utiliza o IR para gerar um código otimizado.
- É gerado um código de máquina para aquela plataforma do compilador a partir do código gerado pelo otimizador.

A figura 06 apresenta como os componentes básicos de um compilador funcionam em conjunto.

Um interpretador é um programa que executa outros programas de alto nível, linha por linha. O código-fonte de linguagens interpretadas podem ser arquivos de script (como por exemplo um arquivo Ruby) que são executados em tempo de execução pelo interpretador. As etapas de um processo de interpretação é muito curto, geralmente os dados (scripts) são passados para o interpretador que avalia o código em tempo de execução. Esse processo pode ser visto na figura 07.

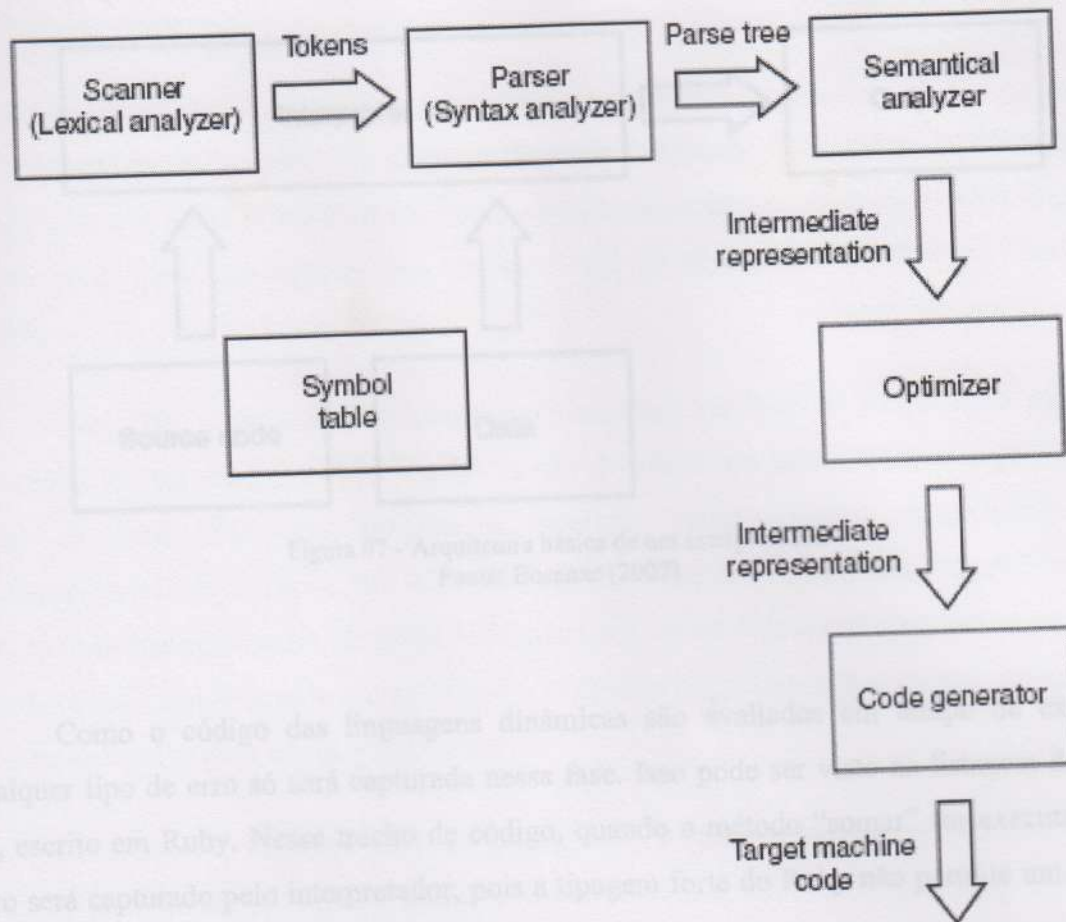


Figura 06 - Arquitetura básica de um compilador.
Fonte: Bosanac (2007)

Um interpretador é um programa que executa outros programas de alto nível, linha por linha. O código-fonte de linguagens interpretadas podem ser arquivos de *script* (como por exemplo um arquivo Ruby) que são executados em tempo de execução pelo interpretador. As etapas de um processo de interpretação é mais curto, geralmente os dados (*script*) são passados para o interpretador que avalia o código em tempo de execução. Esse processo pode ser visto na figura 07.

Os Testes Unitários por parte dos desenvolvedores em linguagens dinâmicas tende a ser muito forte pois permite que esse tipo de erro seja detectado pelo desenvolvedor ainda na fase de desenvolvimento do software.

Mesmo em linguagens de tipagem estática, também é uma boa prática utilizar Testes Unitários, pois sabe-se que só porque um código consegue ser compilado não significa dizer que o mesmo vai se comportar corretamente em tempo de execução.

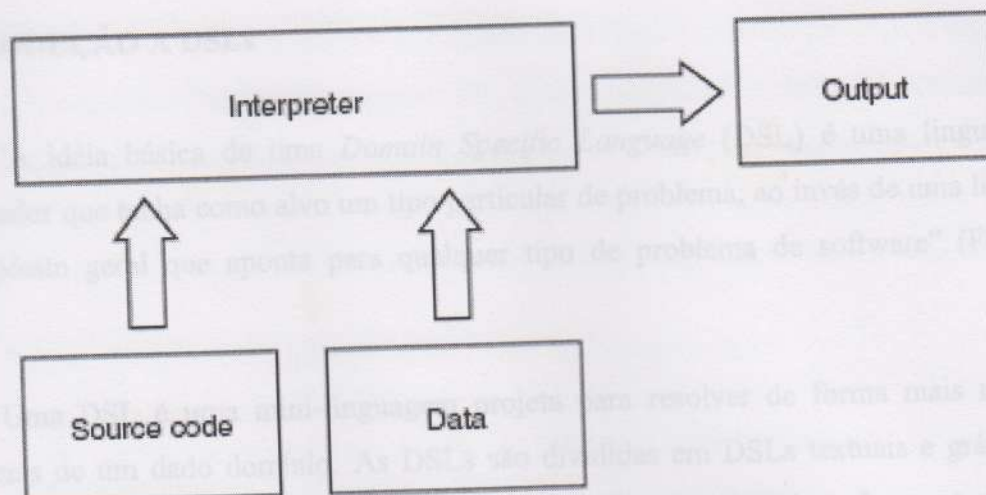


Figura 07 - Arquitetura básica de um interpretador.
Fonte: Bosanac (2007)

Como o código das linguagens dinâmicas são avaliados em tempo de execução, qualquer tipo de erro só será capturada nessa fase. Isso pode ser visto na listagem de figura 08, escrito em Ruby. Nesse trecho de código, quando o método “somar” for executado, um erro será capturado pelo interpretador, pois a tipagem forte do Ruby não permite um número ser somado com uma *String*.

```
def somar(a, b)
  a + b
end
somar(5, "dois")
```

Figura 08 - Captura de erro em tempo de execução.
Fonte: Dados do Pesquisador (2008)

A utilização de Testes Unitários por parte dos desenvolvedores em linguagens dinâmicas tende a ser muito forte pois permite que esse tipo de erro seja facilmente detectado pelo desenvolvedor ainda na fase de desenvolvimento do software.

Mesmo em linguagens de tipagem estática, também é uma boa prática utilizar Testes Unitários, pois sabe-se que só porque um código consegue ser compilado não significa dizer que o mesmo vai se comportar corretamente em tempo de execução.

6 INTRODUÇÃO A DSLs

“A idéia básica de uma *Domain Specific Language* (DSL) é uma linguagem de computador que tenha como alvo um tipo particular de problema, ao invés de uma linguagem de propósito geral que aponta para qualquer tipo de problema de software” (FOWLER, 2004b).

Uma DSL é uma mini-linguagem projetada para resolver de forma mais natural os problemas de um dado domínio. As DSLs são divididas em DSLs textuais e gráficas, que ainda se subdividem em internas e externas. São utilizadas frequentemente no desenvolvimento de *software*, por exemplo, a SQL (*Structured Query Language*) é uma DSL para se trabalhar com banco de dados relacionais, seu único foco é trabalhar com conjuntos de dados, ela não tem abstração e capacidade o suficiente para criar, por exemplo, um aplicativo *Web* ou aplicativo para um dispositivo móvel.

As DSLs são linguagens de escopo limitado, pequenas, expressivas e geralmente são simples de entender para quem conhece o domínio que está sendo abordado. Geralmente DSLs proporcionam uma interface altamente flexível e produtiva para o usuário da mesma, pois a linguagem é composta por palavras do vocabulário do próprio domínio.

Basicamente as DSLs são divididas em dois grupos, DSLs internas e externas, que serão abordadas em seguida.

6.1 DSLs EXTERNAS

Uma DSL externa é projetada para ser independente de qualquer linguagem de programação, como Java ou C#, o criador desse tipo de DSL será o responsável por especificar completamente sua sintaxe, compilador e *parser*. A principal vantagem dessa abordagem é que o criador tem total liberdade de criar sua própria sintaxe, a desvantagem é que criar um compilador e *parser* é um trabalho extra a ser realizado.

Uma DSL externa geralmente procura ser extremamente simples e intuitiva, para permitir que as pessoas que entendem do domínio possam ler e compreender de forma natural o que ali está expresso.

A ferramenta *open source* ANT utilizada para fazer *builds* de projetos é um exemplo de uma DSL externa. O domínio que essa DSL trata é o de *build* de sistemas escritos em Java. Então o usuário dessa DSL escreve em XML os comandos que serão interpretados pela linguagem, baseando-se no contexto que está sendo abordado, dessa forma, inúmeras tarefas relacionadas ao *build* de um sistema pode ser abstraído e simplificado para o usuário da DSL.

Para resumir, os principais pontos de uma DSL externa são:

- Desenvolvedor é responsável por criar uma nova linguagem;
- Necessário escrever os *parsers* para a nova linguagem;
- Desenvolvimento mais complexo;
- Proporciona mais liberdade.

6.2 DSLs INTERNAS

Segundo Martin Fowler (2004b) uma DSL interna é como uma *Application Programming Interface* (API), só que com um nome mais interessante.

Uma DSL interna é projetada dentro de outra linguagem, como Ruby ou Java, ou seja, o autor de uma DSL desse tipo irá utilizar a sintaxe fornecida por uma linguagem para criar outra linguagem. Por utilizar vários recursos fornecidos pela linguagem hospedeira, projetar e implementar DSLs internas é mais simples do que as externas mas tem a desvantagem de limitar o desenvolvedor a sintaxe da linguagem hospedeira.

Por exemplo, o *framework* de desenvolvimento *Ruby on Rails* utiliza uma DSL interna para validar os campos de entrada de um formulário de cadastro, como pode ser visto na figura 09, a DSL tornou esse processo muito intuitivo e sem burocracia para o desenvolvedor.

```
class CadastroUser
  validates_presence_of :name, :email
end
```

Figura 09 - Exemplo de DSL em *Ruby on Rails*
Fonte: Dados do Pesquisador (2008)

Assim como foi resumido os principais pontos de uma DSL externa, os principais pontos de uma DSL interna são:

- Utiliza-se a sintaxe de alguma linguagem para criar a DSL;
- Não precisa se criar *parsers*;
- Desenvolvimento mais simples;
- Liberdade do desenvolvedor imitada a sintaxe da linguagem usada.

```
// Exemplo em Java, contando de 1 a 10
for (int i = 0; i < 10; i++) {
    System.out.println("i = " + i);
}

// Exemplo em Groovy, contando de 1 a 10
0.until(10).each { i => println("i = " + i) }
```

Figura 10 - Groovy é uma linguagem expressiva.
Fonte: Dedos do Pesquisador (2008)

Por linguagens como Java e C# terem origens diferentes de Groovy, Python ou Ruby, fazem com que as mesmas sejam mais "burocráticas" sintaticamente e possuam um compilador rígido.

É possível utilizar o que Martin Fowler (2005) chamou de *Fluent Interface* em praticamente qualquer linguagem orientada a objetos. Usando *fluent interfaces* é possível criar seqüências encadeadas de atributos de mensagens a um determinado objeto (*method chaining*) de tal forma o envio dessas mensagens fiquem mais legíveis, parecendo mais com sentenças de uma frase. Geralmente o desenvolvedor tem que adaptar ou quebrar as convenções de sintaxe da linguagem em uso para atingir esse objetivo.

Na figura 11, o primeiro trecho de código segue a maneira convencional em Java definindo os atributos de um objeto enquanto o segundo trecho mostra como seria usado uma *fluent interface*.

7 FLUENT INTERFACES

Toda linguagem de programação segue um ou mais paradigmas e tem seu próprio estilo de sintaxe para construir *software*. Na construção de DSLs internas, quanto mais expressiva e fluente for a sintaxe da linguagem hospedeira, mais fácil será criar a DSL.

Atualmente, linguagens como Ruby e Groovy tem alto nível de abstração e simplicidade, isso faz parte da natureza dessas linguagens. A figura 10 compara um simples trecho de código escrito em Ruby e em Java.

```
// Exemplo em Java, contando de 1 a 10
for (int i = 0; i < 10; i++) {
    System.out.println(" " + i)
}

// Exemplo em Groovy, contando de 1 a 10
0.upto(10) { println it }
```

Figura 10 - Groovy é uma linguagem expressiva.

Fonte: Dados do Pesquisador (2008)

Por linguagens como Java e C# terem origens diferentes de Groovy, Python ou Ruby, fazem com que as mesmas sejam mais “burocráticas” sintaticamente e possuem um compilador rígido.

É possível utilizar o que Martin Fowler (2005) chamou de *Fluent Interface* em praticamente qualquer linguagem orientada a objetos. Usando *fluent interfaces* é possível criar seqüências encadeadas de envios de mensagens a um determinado objeto (*method chaining*) de tal forma o envio dessas mensagens fiquem mais legíveis, parecendo mais com sentenças de uma frase. Geralmente o desenvolvedor tem que adaptar ou quebrar as convenções de sintaxe da linguagem em uso para atingir esse objetivo.

Na figura 11, o primeiro trecho de código segue a maneira convencional em Java definindo os atributos de um objeto enquanto o segundo trecho mostra como seria usando uma *fluent interface*.

```
// usando método convencional, sem Fluent Interface
Cliente cliente = new Cliente();
cliente.setPraticaNatacao(true);
cliente.setPraticaVolei(false);
cliente.setOdade(25);
cliente.setPagamentoMensal(200);

// definir um objeto Cliente, usando Fluent Interface
ClienteAcademia cliente = new ClienteAcademiaImpl();
cliente.praticaNatacao(true).praticaVolei(false).idade(25).
pagamentoMensal(200.0);
```

Figura 11 - Exemplo de *Fluent Interface* em Java.

Fonte: Dados do Pesquisador (2008)

O que claramente pode ser visto na figura 11 é que o código fica parecendo mais natural (ou mais fluente) do que da maneira usual, que seria popular mecanicamente um objeto de acordo com a quantidade de atributos que existissem na entidade. Atualmente, inúmeros *frameworks* utilizam essa abordagem, como a API Criteria do Hibernate e o *framework* para testes JMock.

Existe uma linha fina que separa uma DSL interna de uma *fluent interface*, alguns autores, como Martin Fowler, sugerem que DSLs internas e *fluent interfaces*, são a mesma coisa, outros acham que uma *fluent interface* é apenas uma adaptação na linguagem para torna-la mais simples de entender.

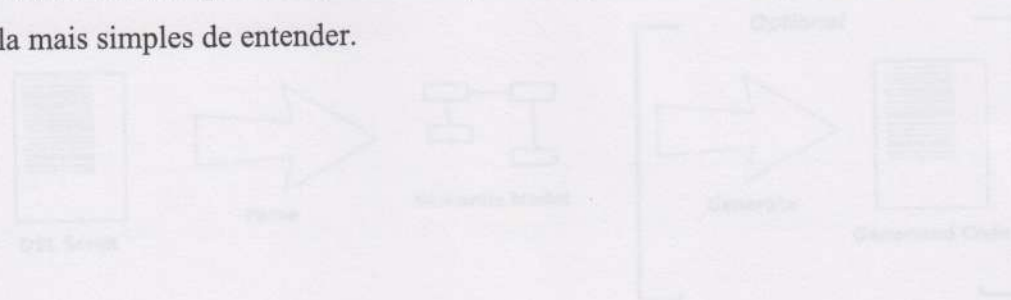


Figura 12 - *Semantic Model* sendo populado a partir de uma DSL.

Fonte: Fowler (2005)

8 PADRÕES PARA DSLs

Essa seção tem o objetivo de apresentar resumidamente alguns padrões “emergentes” utilizados no desenvolvimento de DSLs. Essa lista está longe de ser abrangente, mas é importante pois fornece uma visão macro da área.

8.1 SEMANTIC MODEL

O *Semantic Model* é o domínio que será populado pela DSL. *Semantic Model* é basicamente a mesma coisa que um *Domain Model*. Como o domínio é o centro do sistema, um *Semantic Model* é um subconjunto do mesmo, pois uma DSL tem como objetivo trabalhar com um foco específico do domínio.

Um *Semantic Model* se divide em duas interfaces distintas. Existe a interface operacional, que permite que o cliente use-a para popular o modelo que está trabalhando. Já a interface populacional é usada pela DSL para criar a instancia de um modelo.

Um *Semantic Model* pode ser populado a partir de uma DSL, como pode ser visto na figura 12.

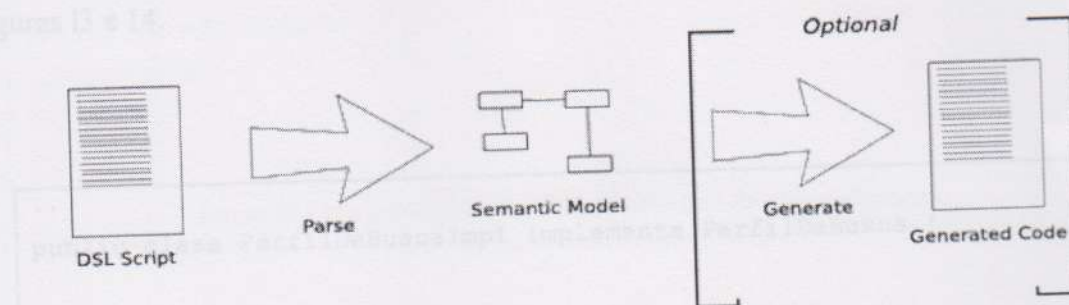


Figura 12 - *Semantic Model* sendo populado a partir de uma DSL.

Fonte: Fowler (2005)

8.2 EXPRESSION BUILDER

No desenvolvimento de uma DSL externa existe uma separação clara entre os *scripts*, o *parser* criado e o domínio que está sendo abordado. Basicamente, o fluxo em uma DSL externa se baseia no seguinte:

- Os *scripts* são escritos em uma linguagem própria;
- O *parser* criado pelo desenvolvedor lê o *script* da DSL;
- O domínio ou mais precisamente o *Semantic Model* é populado.

Já no desenvolvimento de uma DSL externa o processo não é tão bem definido. O padrão *Expression Builder* provê uma maneira de criar uma *fluent interface* como uma camada separada. Basicamente cria-se uma classe (que geralmente pode implementar alguma interface) que possui os métodos relacionados ao domínio que está sendo trabalhado. Cada método dessa classe retorna uma instância do próprio objeto (*this*), isso permite criar um “contexto”, onde cada chamada a um método permite que outro método seja chamado de forma encadeada (*method chaining*).

Uma exemplo da implementação desse padrão escrito em Java pode ser vista nas figuras 13 e 14.

```
...  
public class PerfilDeBuscaImpl implements PerfilDeBusca {  
  
    private Pessoa pessoa;  
  
    public static PerfilDeBusca buscarPerfis() {  
        return new PerfilDeBuscaImpl();  
    }  
}
```

Figura 13 - Implementação do padrão *Expression Builder*.
Fonte: Dados do Pesquisador (2008)

```

private PerfilDeBuscaImpl() {
    pessoa = new Pessoa();
}

public List<Pessoa> perfisLocalizados() {
    RepositorioPessoas repositório = new RepositorioPessoas();
    return repositório.findPorPessoa();
}

public PerfilDeBusca cidade(String c) {
    pessoa.setCidade(c);
    return this;
}

public PerfilDeBusca corOlhos(String o) {
    pessoa.setCorOlhos(o);
    return this;
}

public PerfilDeBusca idadeMaxima(int i) {
    pessoa.setIdade(i);
    return this;
}

public PerfilDeBusca sexo(char s) {
    pessoa.setSexo(s);
    return this;
}

public PerfilDeBusca altura(char a) {
    pessoa.setAltura(a);
    return this;
}

```

Figura 14 - Implementação do padrão *Expression Builder*, continuação.

Fonte: Dados do Pesquisador (2008)

```

LocalizarCarros(), doAno(2008), marca("Ford");

```

Figura 16 - Listagem de código usando *Method Chaining*

Fonte: Dados do Pesquisador (2008)

Como foi dito anteriormente, não temos o objetivo de apresentar uma lista abrangente com todos os padrões para desenvolvimento de DSLs. Para se ter uma ideia, entre alguns padrões que não foram abordados neste trabalho: *Function Sequence*, *Nested Function*, *Object Scoping*, *Annotation*, etc.

A utilização dessa *fluent interface* pode ser vista na figura 15.

```
...
public class PessoaMain {
    public static void main(String[] args) {

        List<Pessoa> pessoas = PerfilDeBuscaImpl.
            buscarPerfis()
                .cidade("Natal")
                .corOlhos("castanhos")
                .idadeMaxima(25)
                .sexo('F')
                .altura(1.70)

    }
}
```

Figura 15 - Utilizando a *Fluent Interface* criada anteriormente.

Fonte: Dados do Pesquisador (2008)

Posteriormente, será mostrado como a utilização do recurso de *import static* disponível a partir do Java 5 pode simplificar esse código, eliminando a necessidade de ter que declarar, nesse exemplo específico, a classe *PerfilDeBuscaImpl*.

8.3 METHOD CHAINING

A implementação do padrão *Method Chaining* é alterar o tipo de retorno do método para o mesmo tipo do objeto atual, permitindo dessa forma que uma nova chamada seja realizada em outro método desse objeto. Dessa forma, o *method chaining* serve para criar uma sequência encadeada de chamadas a vários métodos em um objeto. A figura 16 mostra a utilização desse padrão.

```
localizarCarros().doAno(2008).doFabricante("Ford");
```

Figura 16 - Listagem de código usando *Method Chaining*.

Fonte: Dados do Pesquisador (2008)

Como foi dito anteriormente, não temos o objetivo de apresentar uma lista abrangente com todos os padrões para desenvolvimento de DSLs. Para se ter uma idéia, entre alguns padrões que não foram abordados neste trabalho: *Function Sequence*, *Nested Function*, *Object Scoping*, *Annotation*, etc.

9 DESENVOLVENDO DSLs

Linguagens derivadas do C, como Java, possuem na sua sintaxe ponto-e-vírgulas para indicar o fim de uma linha de instrução para o compilador, bem com a utilização de parêntesis na invocação de métodos, por exemplo. No desenvolvimento de DSLs isso é chamado de ruído (no inglês, *noise*). É considerado um ruído porque adiciona informação (caracteres extras) a expressão da DSL que não tem nenhum relacionamento com o domínio da DSL, são caracteres que servem apenas para satisfazer o compilador, porém acabam atrapalhando a fluência da expressão. Por outro lado, linguagens como Ruby ou Groovy permitem um desenvolvimento mais fluente e flexível.

Nessa seção desse trabalho serão apresentadas algumas DSLs internas escritas em Java ou Groovy apresentando de forma prática como as DSLs são escritas e como podem facilitar a leitura de código.

9.1 EXEMPLO 1: DSL PARA FACILITAR COMANDOS SQL

Nesse primeiro exemplo, vamos abordar a criação de uma DSL para facilitar a utilização de código SQL dentro de uma aplicação Java.

Uma das maneiras mais usuais que os desenvolvedores utilizam é inserir código SQL dentro de uma camada separada do resto da aplicação. Essa camada é uma implementação do padrão DAO (*Data Access Object*). O código SQL é geralmente concatenado com alguns parâmetros, de forma que o código SQL seja montado em tempo de execução. A figura 17 mostra como isso pode ser implementado.

```
...
public class UsuarioDAO {
    public List<Usuario> findUsuariosByPrimarykey(int id) {
        StringBuilder sb = new StringBuilder();
        sb.append("SELECT nome, rua, telefone");
        sb.append("FROM tabela_usuario user");
        sb.append("INNER JOIN tabela_endereco end ON end.id = user.id");
        sb.append("WHERE user.id = " + id);
    }
}
```

Figura 17 - Maneira convencional de usar SQL em Java.
Fonte: Dados do Pesquisador (2008)

O problema dessa abordagem é a repetição de código em todos os métodos que precisam acessar o banco para realizar alguma operação de busca e a concatenação de *Strings* com as mesmas instruções, como *FROM*, *INNER JOIN*, etc.

Para utilizar a DSL, o usuário da mesma irá escrever algo equivalente a figura 18.

```
import static br.exemplo.dsl.sql.ConsultaSQLImpl.consultar;
...
String sql = consultar().select("nome,", "rua,",
    "telefone").from("tabela_usuario user")
    .inner("tabela_endereco end").on("end.id = user.id")
    .where("user.id = " + id).sql();
```

Figura 18 - Usando uma DSL para trabalhar com SQL
Fonte: Dados do Pesquisador (2008)

O primeiro método chamado da figura 18 serve para contextualizar as chamadas subsequentes, enquanto o último método chamado serve para retornar o código SQL montado pronto para ser executado pelo JDBC (*Java Database Connectivity*).

Comparando a listagem da figuras 17 com a da figura 18 é possível observar uma simplificação na maneira como a SQL vai ser escrita, lida e mantida. É utilizado o recurso de *import static* disponível no Java 5. Esse recurso permite eliminar a necessidade de ter que declarar uma classe para poder utilizar seus métodos estáticos, isso simplifica a escrita de código.

A figura 19 mostra a interface utilizada na construção dessa DSL. Outro recurso disponível a partir do Java 5, chamado *varargs* é utilizado para simplificar o código. Esse recurso permite que um numero arbitrário de parâmetros sejam passados para determinado método.

```
...
public interface ConsultaSQL {

    ConsultaSQL select(String ... campos);
    ConsultaSQL from(String tabela);
    ConsultaSQL inner(String tabela);
    ConsultaSQL on(String condicao);
    ConsultaSQL where(String condicao);
    String sql();
}
```

Figura 19 - Interface com os métodos da API.
Fonte: Dados do Pesquisador (2008)

As figuras 20 e 21 mostram a classe de implementação da DSL.

```
...
public class ConsultaSQLImpl implements ConsultaSQL {

    private StringBuilder builder = new StringBuilder();

    public static ConsultaSQLImpl consultar() {
        return new ConsultaSQLImpl();
    }

    public ConsultaSQL select(String ... campos) {

        String fields = "";
        for (String items : campos) {
            fields += items;
        }
        builder.append("SELECT " + fields + " ");
        return this;
    }

    public ConsultaSQL from(String tabela) {
        builder.append("FROM " + tabela + " ");
        return this;
    }
}
```

Figura 20 - Classe que implementa a DSL.
Fonte: Dados do Pesquisador (2008)

```

public ConsultaSQL inner(String tabela) {
    builder.append("INNER JOIN " + tabela + " ");
    return this;
}

public ConsultaSQL on(String condicao) {
    builder.append("ON " + condicao + " ");
    return this;
}

public ConsultaSQL where(String condicao) {
    builder.append("WHERE " + condicao);
    return this;
}

public String sql() {
    return builder.toString();
}
}

```

Figura 21 - Classe que implementa a DSL, continuação.
 Fonte: Dados do Pesquisador (2008)

9.2 EXEMPLO 2: DSL PARA TRABALHAR COM SERVIÇOS RESTFUL

Nesse exemplo será construída uma DSL que simplificará o acesso a API baseada na arquitetura REST (*Representational State Transfer*) do site de marcadores *on-line* Delicious.

Uma arquitetura REST se baseia no conceito de ROA (Arquitetura Orientada a Recursos), que basicamente é a publicação recursos (entidades) *on-line*, onde cada recurso permite uma série de operações sobre si, utilizando unicamente o protocolo HTTP.

Atualmente, inúmeros *sites* publicam os mais variados serviços através de uma arquitetura REST, como os serviços do Google, da Amazon e do Flickr. Os aplicativos *clients* são desacoplados dos recursos, assim esses *softwares* podem ser outros recursos REST ou mesmo um aplicativo executando em um dispositivo portátil.

Especificamente no caso do *site* Delicious, existem uma série de funcionalidades disponíveis, entre elas:

- Retornar uma lista de todos os *bookmarks*;
- Listar todos os *bookmarks* filtrando por *tags* específicas;
- Adicionar um novo *bookmark*;
- Remover um *bookmark*.

A tabela I exibe como é o formato de uma URL para alguns dos recursos disponíveis no Delicious, a lista completa encontra-se em no site do mesmo.

Tabela I - API REST do Delicious.

URL	FUNÇÃO
https://api.del.icio.us/v1/posts/all	Recupera todos os marcadores do usuário.
https://api.del.icio.us/v1/tags/get?tag=musica	Recupera todos os marcadores filtrando por uma <i>tag</i> (música).
https://api.del.icio.us/v1/tags/delete?tag=musica	Deleta uma <i>tag</i> . O parametro obrigatorio a ser passado é a <i>tag</i> a ser removida.

Para exemplificar de forma prática como a invocação a recursos baseados em REST pode ser realizada e como seria o formato da resposta gerada pelo *server* foi utilizado como *client* o *software* de linha de comando cURL para realizar a invocação ao recurso, como pode ser visto na figura 22.

```
> curl https://agostinhocn:password@api.del.icio.us/v1/posts/all

<?xml version="1.0" encoding="UTF-8"?>
<posts user="agostinhocn" update="2008-09-18T14:47:06Z" tag=""
total="2">

  <post href="http://www.infoq.com/"
hash="d445d8bf299c9b5c16058954f490be91" description="InfoQ: Tracking
change and innovation in the enterprise software ..."
tag="desenvolvimento arquitetura" time="2008-09-18T14:47:06Z"
extended=""/>

  <post href="http://www.myspace.com/cidadaoquem"
hash="1cd0de9f28d45668d4b57dbe949ad30d" description="MySpace.com -
Cidadão Quem - Porto Alegre, BR - Rock - www.myspace.com ..."
tag="musica" time="2008-09-08T18:10:41Z" extended=""/>
</posts>

<!-- fe04.api.del.ac4.yahoo.net uncompressed/chunked Thu Sep 18
07:48:34 PDT 2008 -->
```

Figura 22 - Usando *software* de terminal curl

Fonte: Dados do Pesquisador (2008)

Apresentamos na figura 23 como acessar o Delicious. Pode-se perceber como a invocação de métodos é extremamente simples e intuitiva.

```
import static br.exemplo.logarDelicious;
...
logarDelicious("agostinhocn", "password")
    .marcadoresDaTag("mac")
        .adicionarMarcador("www.uol.com.br", "UOL...")
```

Figura 23 - Aplicação *client* acessando API Delicious através de uma DSL
Fonte: Dados do Pesquisador (2008)

Mais uma vez é utilizado o recurso de *import static* disponível a partir do Java 5. Esse código lista todos os marcadores disponíveis em uma conta filtrando por uma *tag*, enquanto ao mesmo tempo ele insere um novo marcador.

A figura 24 apresenta a DSL desenvolvida para trabalhar com o Delicious, enquanto as figuras 25, 26, 27 e 28 são a implementação do acesso propriamente dito, que fica um nível abaixo da DSL.

```
//imports omitidos
public class DSLRestDelicious {

    private static HttpRESTConf httpConf;

    public static DSLRestDelicious logarDelicious(String usuario,
String senha) {
        httpConf = new HttpRESTConf(usuario, senha);
        return new DSLRestDelicious();
    }

    public DSLRestDelicious todosOsMarcadores() {
        httpConf.todosPostsRecentes();
        return this;
    }

    public DSLRestDelicious marcadoresDaTag(String tag) {
        httpConf.postsPorTag(tag);
        return this;
    }

    public DSLRestDelicious adicionarMarcador(String url, String
desc) {
        httpConf.adicionarMarcador(url, desc);
        return this;
    }
}
```

Figura 24 - Implementação da DSL para acesso REST
Fonte: Dados do Pesquisador (2008)

```

//imports omitidos
public class HttpRESTConf {

    private HttpClient client;
    private String usuario, senha, url;

    public HttpRESTConf(String usuario, String senha) {
        setUsuario(usuario);
        setSenha(senha);
    }

    private void autenticar() {
        Credentials creds = new
UsernamePasswordCredentials(getUsuario(), getSenha());
        client = new HttpClient();
        client.getState().setCredentials(AuthScope.ANY, creds);
    }

    public void abrirRecurso() throws HttpException, IOException,
ParserConfigurationException, SAXException, XPathExpressionException
{

        autenticar();
        GetMethod method = new GetMethod(url);

        client.executeMethod(method);
        InputStream responseBody =
method.getResponseBodyAsStream();

        parserResponseToXML(responseBody, method);
    }

    private void parserResponseToXML(InputStream responseBody,
GetMethod method) throws ParserConfigurationException, SAXException,
IOException, XPathExpressionException {
        DocumentBuilderFactory docBuilderFactory =
DocumentBuilderFactory.newInstance();
        DocumentBuilder docBuilder =
docBuilderFactory.newDocumentBuilder();
        Document doc = docBuilder.parse(responseBody);
        method.releaseConnection();

        exibir(doc);
    }
}

```

Figura 25 - Implementação do acesso ao serviço REST.
Fonte: Dados do Pesquisador (2008)

```

private void exibir(Document doc) throws XPathExpressionException {

    XPath xpath = XPathFactory.newInstance().newXPath();
    NodeList bookmarks =
(NodeList)xpath.evaluate("/posts/post", doc, XPathConstants.NODESET);

    for (int i = 0; i < bookmarks.getLength(); i++) {
        NamedNodeMap bookmark =
bookmarks.item(i).getAttributes();
        String description =
bookmark.getNamedItem("description").getNodeValue();
        String uri =
bookmark.getNamedItem("href").getNodeValue();
        System.out.println(description + ": " + uri);
    }
}

public void todosPostsRecentes() {

    url = "https://api.del.icio.us/v1/posts/all";
    try {

        abrirRecurso();

    } catch (HttpException e) {
        e.printStackTrace();
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ParserConfigurationException e) {
        e.printStackTrace();
    } catch (SAXException e) {
        e.printStackTrace();
    }
}
}

```

Figura 26 - Implementação do acesso ao serviço REST, continuação.
 Fonte: Dados do Pesquisador (2008)

```

public void postsPorTag(String tag) {
    url = "https://api.del.icio.us/v1/posts/get?tag=" + tag;
    try {
        abrirRecurso();
    } catch (HttpException e) {
        e.printStackTrace();
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ParserConfigurationException e) {
        e.printStackTrace();
    } catch (SAXException e) {
        e.printStackTrace();
    }
}

public void adicionarMarcador(String marcador, String desc) {
    url = "https://api.del.icio.us/v1/posts/add?url=" +
    marcador + "&description=" + desc;
    try {
        abrirRecurso();
    } catch (HttpException e) {
        e.printStackTrace();
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ParserConfigurationException e) {
        e.printStackTrace();
    } catch (SAXException e) {
        e.printStackTrace();
    }
}

```

Figura 27 - Implementação do acesso ao serviço REST, continuação.
 Fonte: Dados do Pesquisador (2008)

```

public String getUsuario{
    return usuario;
}

public void setUsuario(String usuario) {
    this.usuario = usuario;
}

public String getSenha() {
    return senha;
}

public void setSenha(String senha) {
    this.senha = senha;
}
}

```

Figura 28 - Implementação do acesso ao serviço REST, continuação.
Fonte: Dados do Pesquisador (2008)

9.3 EXEMPLO 3: DSL EM GROOVY USANDO META-OBJECT PROTOCOL

Um dos grandes recursos do Groovy é conhecido como Meta-Object Protocol (MOP). Esse recurso permite a utilização de meta-programação de uma maneira simplificada. O MOP permite que sejam adicionados novos métodos ou comportamentos a objetos Groovy em tempo de execução.

Para exemplificar como funciona o fluxo decorrente da invocação de um método em um objeto Groovy, é apresentado o fluxo na figura 29. Analisando a imagem, percebe-se que o Groovy pode tratar métodos que não existam em um objeto apropriadamente (entre outras coisas) permitindo, assim, que ações sejam personalizadas de acordo com a necessidade, proporcionando flexibilidade no desenvolvimento de uma DSL.

```

// Exemplo de uso do MOP
// Criando um objeto Integer
Integer i = new Integer(10);

// Invocando um método que não existe em Integer
// O Groovy trata isso como uma chamada de método dinâmico
// e cria um novo método no objeto i
i.métodoQueNãoExiste();

```

Figura 29 - Método adicionado a um objeto Integer.
Fonte: Dados do Pesquisador (2008)

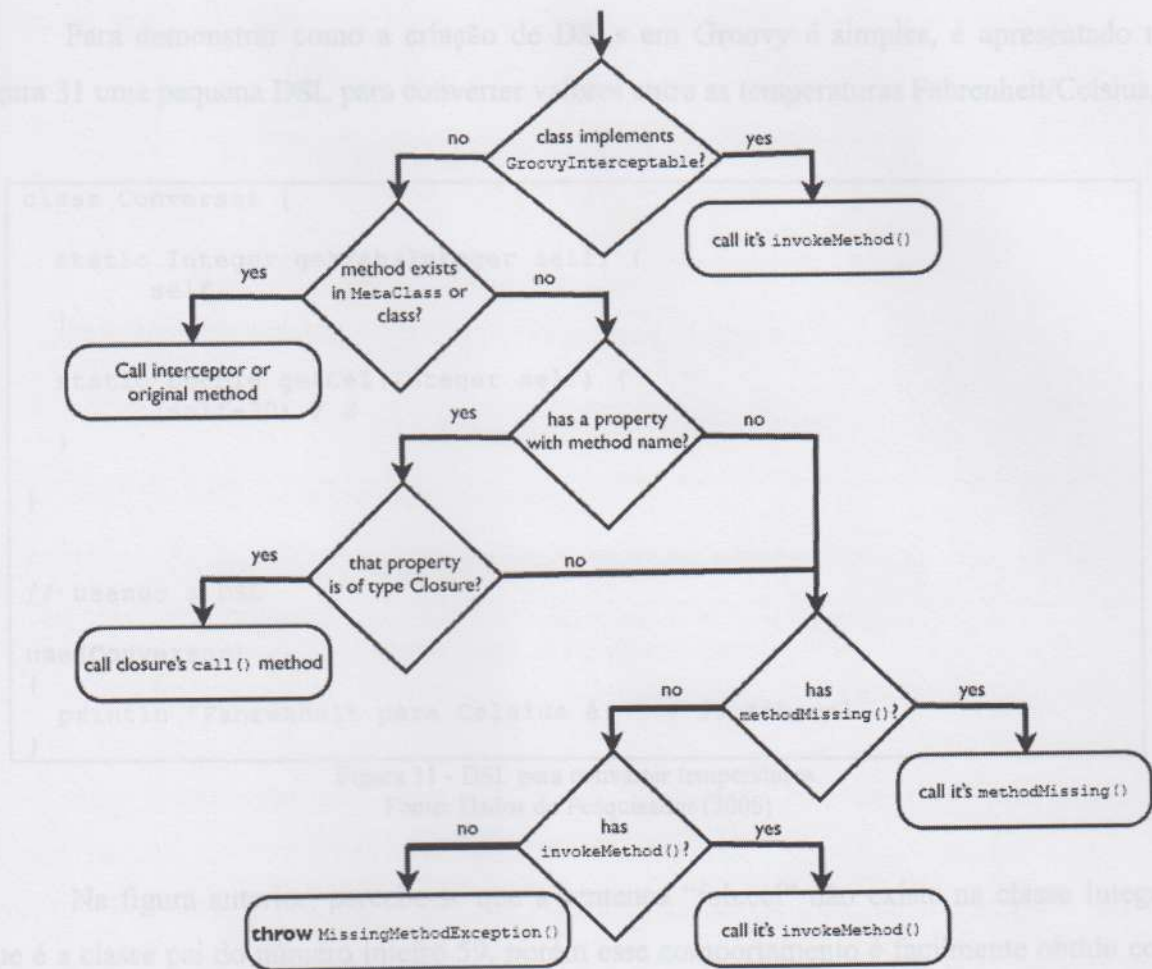


Figura 29 - Quando um método é invocado em Groovy.
Fonte: Subramaniam (2008)

As classes da linguagem Groovy são classes “abertas”, ou seja, o desenvolvedor pode incrementar essas classes de acordo com a necessidade. A figura 30 mostra uma das maneiras de como isso pode ser feito.

```

algumMetodo = { ->
    "vou ser adicionado a classe Integer"
}

Integer.metaClass.algumMetodo = algumMetodo

println "chamado do valor 10 " + 10.algumMetodo()
println "chamado do valor 150 " + 150.algumMetodo()

//saida no console
chamado do valor 10 vou ser adicionado a classe Integer
chamado do valor 150 vou ser adicionado a classe Integer
  
```

Figura 30 - Método adicionado a classe Integer.
Fonte: Dados do Pesquisador (2008)

Para demonstrar como a criação de DSLs em Groovy é simples, é apresentado na figura 31 uma pequena DSL para converter valores entre as temperaturas Fahrenheit/Celsius.

```
class Conversor {
    static Integer getFah(Integer self) {
        self
    }
    static Double getCel(Integer self) {
        (self-30) / 2
    }
}

// usando a DSL
use(Conversor)
{
    println "Fahrenheit para Celsius é: " + 59.fah.cel
}
```

Figura 31 - DSL para converter temperaturas.
Fonte: Dados do Pesquisador (2008)

Na figura anterior, percebe-se que a sentença “fah.cel” não existe na classe Integer, que é a classe pai do número inteiro 59, porém esse comportamento é facilmente obtido com um pouco de código. Uma vez isso feito, o usuário pode utilizar facilmente a funcionalidade.

9.4 EXEMPLO 4: DSL EM GROOVY PARA DETECTAR DOENÇAS

A DSL desenvolvida no próximo exemplo é uma simplificação de um sistema para analisar sintomas de doenças, nesse caso estamos verificando sintomas apenas para duas doenças: Sarampo ou para um simples resfriado.

Para utilizar a DSL desenvolvida, é possível escrever o código apresentado na figura 32. É importante observar como o código Groovy ficou extremamente próximo ao domínio que tentamos resolver e com o mínimo de ruído (o código não possui parênteses nem ponto-e-vírgula).

```
// verificando sintomas p/ resfriado

sintomas febre, dor_de_cabeca, enjojo
resultado
```

Figura 32 - Utilizando a DSL para verificar sintomas (Resfriado).
Fonte: Dados do Pesquisador (2008)

Para verificar o Sarampo, o código seria como na figura 33.

```
// verificando sintomas p/ sarampo

sintomas febre_alta, tosse_intensa, coriza, conjuntivite, mal_estar
Resultado
```

Figura 33 - Utilizando a DSL para verificar sintomas (Sarampo).
Fonte: Dados do Pesquisador (2008)

Após a execução do código apresentado na figura 32, a seguinte mensagem seria exibida para o usuário: “Provavelmente Resfriado, tratar: aspirina, bastante líquido, e repouso”. Enquanto a mensagem exibida para a execução da listagem escrita na figura 33 seria: “Provavelmente Sarampo, tratar: repouso, banhos mornos ou compressas frias para controlar a febre, bastante líquido”.

Nas figuras 34 e 35 é apresentada a codificação da DSL propriamente dita.

```
sintomas = [:]

def sintomas(String[] names) {
  names.each { sintomas[it] = 0 }
}

def getResultado() {
  def tratamento = ''
  def sintomasInformados = []
  def cont = 0;
  def resfriado = false, sarampo = false;

  sintomas.each { key, valor ->
    sintomasInformados[cont++] = key
  }

  for (i in sintomasInformados) {
```

Figura 34 - Codificação da DSL para verificar sintomas.
Fonte: Dados do Pesquisador (2008)

```

// sarampo
if ( i.equals("febre_alta"))
    febre_alta = true
if (i.equals("tosse_intensa"))
    tosse_intensa = true
if (i.equals("coriza") )
    coriza = true
if (i.equals("conjuntivite") )
    conjuntivite = true
if (i.equals("mal_estar") )
    mal_estar = true

// resfriado
if ( i.equals("febre"))
    febre = true
if (i.equals("dor_de_cabeca"))
    dor_de_cabeca = true
if (i.equals("enjoo") )
    enjoo = true

if ( febre == true && dor_de_cabeca == true && enjoo == true)
    resfriado = true

if ( febre_alta == true && tosse_intensa == true && coriza ==
true && conjuntivite == true && mal_estar == true)
    sarampo = true
}
if (sarampo)
    println "Provavelmente Sarampo, tratar: repouso, banhos mornos
ou compressas frias para controlar a febre, bastante líquido."

if (resfriado)
    println "Provavelmente Resfriado, tratar: aspirina, bastante
líquido, e repouso."
}

def propertyMissing(String name) {
    name
}

def methodMissing(String name, args) {
    if (sintomas.containsKey(name)) {
        sintomas[name] = args[0]
    }
    else {
        throw new MissingMethodException(name, this.class, args)
    }
}
}

```

Figura 35 - Codificação da DSL para verificar sintomas, continuação.
 Fonte: Dados do Pesquisador (2008)

16 **CP** Apesar de termos focado na linguagem Groovy, outras linguagens dinâmicas permitem o desenvolvimdo de DSLs internas com o mínimo de *overhead* possível e com o máximo de flexibilidade.

Em qualquer caso, ao desenvolvermos DSLs, existem algumas coisas que devem ser consideradas antes do desenvolvimento da mesma, pois isso sempre será a melhor maneira de seguir (FERNANDES, 2008, p. 364).

Primeira, geralmente as pessoas que podem ler e escrever realmente bem as DSLs são os próprios autores da mesma, dessa forma levando o conhecimento do resto da equipe de um projeto.

Segundo, se um desenvolvedor entrar em um projeto que possui DSLs, pode haver algumas dificuldades, provavelmente ele terá que passar uma curva de aprendizado, enquanto poderia estar produzindo código em uma linguagem de uso comum, como Java ou C#.

Terceira, uma programação é um recurso poderoso, porém pode complicar em alguns casos, um desenvolvedor pode adicionar muitos recursos próprios em alguns trechos de código dentro de uma DSL (alterando classes da própria linguagem) que talvez possa dificultar a manutenção do projeto, principalmente se essa manutenção for feita por outras pessoas.

Em resumo, apesar do desenvolvimento de DSLs ser útil em vários casos, é preciso que o desenvolvedor tenha bom senso ao aplicá-las.

10 LIMITES NA UTILIZAÇÃO DE DSLS

Apesar de existirem vantagens no desenvolvimento de DSLs, existem alguns pontos que devem ser considerados antes do desenvolvimento da mesma, pois nem sempre será o melhor caminho a seguir (FERNANDEZ, 2008, p. 304).

Primeiro, geralmente as pessoas que podem ler e escrever realmente bem as DSLs são os próprios autores da mesma, dessa forma limitando o conhecimento do resto da equipe de um projeto.

Segundo, se um desenvolvedor entrar em um projeto que possua DSLs para serem abordadas, provavelmente ele terá que passar uma curva de aprendizado, enquanto poderia está produzindo código em uma linguagem de uso comum, como Java ou C#.

Terceiro, meta-programação é um recurso poderoso, porém pode complicar em alguns casos, um desenvolvedor poder adicionar tantos recursos próprios em alguns trechos de código durante a criação de uma DSL (alterando classes da própria linguagem) que talvez possa dificultar a manutenção do projeto, principalmente se essa manutenção for feita por outra pessoa.

Em resumo, apesar do desenvolvimento de DSLs ser útil em vários casos, é preciso que o desenvolvedor tenha bom-senso ao aplicá-las.

11 CONCLUSÕES

Esse trabalho procurou abordar o desenvolvimento de DLSs internas, apresentando inúmeros conceitos essenciais de tal forma que permita ao leitor desse trabalho ter base o suficiente para ler uma obra mais completa posteriormente, de forma a aprofundar-se sobre o assunto.

Foi apresentada algumas características existentes nas linguagens dinâmicas e como essas linguagens podem simplificar o projeto de DLSs. Também foi abordado as vantagens que uma DSL (ou API) projetada de forma “fluente” pode trazer para o desenvolvedor, apresentando de forma prática (implementando código) como construir DLs em Groovy e em Java.

Por ser um tema que despertou recente interesse na comunidade de desenvolvimento, atualmente existe pouca literatura sobre o assunto, porém é visto que a medida que esse interesse cresce e se espalha, provavelmente não vai demorar muito para que obras de referência completa sejam publicadas, demonstrando muito mais vantagens (e também as armadilhas) ao desenvolver e utilizar DSLs.

FOWLER, Martin. *Fluent Interfaces*, 2003. Disponível em:
<http://www.martinfowler.com/bliki/FluentInterface.html>. Acesso em: 10 Jul. 2008.

GOSEN, Ross. *Design Patterns in Ruby*. Addison-Wesley, 2007.

SUBRAMANIAM, Venkat. *Programming Groovy: Dynamic Productivity for the Java Developer*. Pragmatic Bookshelf, 2008.

VENNERS, Bill. *Inside the Java Virtual Machine*. McGraw-Hill Companies, 2000.

REFERÊNCIAS

AKITA, Fábio. **Apenas Diga Não a XML**. Disponível em:
<<http://www.akitaonrails.com/2006/9/29/apenas-diga-nao-a-xml>>. Acesso em 6 jul. 2008.

BOSANAC, Dejan. **Scripting in Java: Languages, Frameworks, and Patterns**. Addison-Wesley, 2007.

FERNANDEZ, Obie. **The Rails Way**. Addison-Wesley, 2008.

FORD, Neal. **Building DSLs in Static and Dynamic Language**. Disponível em:
<<http://www.nealford.com/mypastconferences.htm>>. Acesso em: 30 jun. 2008.

FOWLER, Martin. **Closure**, 2004a. Disponível em:
<<http://martinfowler.com/bliki/Closure.html>>. Acesso em: 6 jul. 2008.

FOWLER, Martin. **DSL**, 2004b. Disponível em:
<<http://www.martinfowler.com/bliki/DomainSpecificLanguage.html>>. Acesso em: 6 jul. 2008.

FOWLER, Martin. **Fluent Interface**, 2005. Disponível em:
<http://www.martinfowler.com/bliki/FluentInterface.html>. Acesso em: 10 jul. 2008.

OLSEN, Russ. **Design Patterns in Ruby**. Addison-Wesley, 2007.

SUBRAMANIAM, Venkat. **Programming Groovy: Dynamic Productivity for the Java Developer**. Pragmatic Bookshelf, 2008.

VENNERS, Bill. **Inside the Java Virtual Machine**. McGraw-Hill Companies, 2000.