Expressividade da linguagem no ato de programar

Jônatas Davi Paganini

novembro de 2009

Abstract

Este trabalho busca mostrar uma linguagem de programação mais expressiva. Através de exemplos de programas de computador, será mostrado como a linguagem de programação pode ser simples e de fácil compreensão, como a linguagem Ruby pode ser caracterizada como uma linguagem de computador mais humana.

This paper shows how many expressive language is Ruby programming language. Through code examples, the language can be simple and easy to understand. This article shows how the computer's language can be like human's language.

Palavras-chave: linguagem, expressividade, DSL, Ruby, código expressivo.

1 *Introdução*

A construção de programas de computador retraí as pessoas, elas pensam que programas são complexos e difíceis de aprender. Com muitas regras e limites, a resistência ao aprendizado aumenta ainda mais. Em sua maioria, as linguagens oferecem suporte apenas em Língua Inglesa, tornando a interpretação da linguagem computacional ainda mais desafiante para o profissional.

Os programas crescem e tornam-se complexos. Através de técnicas de desenvolvimento e o uso de uma linguagem bem específica, é possível mudar esta paisagem. Em uma análise das dificuldades que o programador encara, muitas vezes parte deste trabalho é culpa da linguagem, sua dificuldade de aprendizado e expressão.

Os tempos evoluiram e será abordada uma linguagem simples, com foco nas pessoas e não nos computadores. Com o objetivo de chegar mais próximo da realidade externa a um programa de computador, é possível escrever mini-linguagens, bibliotecas bem definidas com uma sintaxe única e expressiva para realizar o seu trabalho. Ferramentas específicas realizam tarefas específicas. Fazendo a escolha certa, é possível obter um código simples, que informe exatamente a sua funcionalidade, deixando a complexidade para camadas mais abstratas e específicas.

Uma maneira inteligente de entender coisas complexas, é quebrá-las em partes menores. Quando um elemento demonstra-se complexo, pode ser dividido em outros conceitos e cada um deles terá sua responsabilidade mais específica e mais simples. Existem diversas maneiras de fragmentar um problema complexo, usando uma solução criativa, é possível diminuir a dificuldade de compreensão; estabelecendo terminologias, é possível desmassificar a complexidade.(10)

Aplicando soluções simples, é possível fragmentar e unir um software complexo.

2 A linguagem

Linguagem não é somente as línguas, mas é uma série de outros sistemas de comunicação, notação ou cálculo, criados artificialmente pelo homem com objetivos específicos. Apesar de ser um código com regras determinadas, natural ou artificialmente, as linguagens podem ser consideradas, sob um ponto de vista comportamental, além de representarem ideias, comunicação, significados e pensamentos. (9). Embora os animais também se comuniquem, a linguagem propriamente dita pertence apenas ao Homem(1).

Portanto, pode se dizer que as linguagens possuem flexibilidade e versatilidade, sendo usadas através do interesse e da vontade humana. Isso se deve à produtividade dos sistemas de linguagem, que possibilitam a construção e interpretação de novos sinais. Todos os sistemas possibilitam a seus usuários construir e compreender um número indefinido de enunciados, que jamais viram antes. Isso, por outro lado, se dá dentro dos limites estabelecidos pelas regras.

As linguagens de programação estão sob estes mesmos trilhos, conseguem ser flexíveis e estabelecer a definição de novos sinais e expressões. Este artigo abordará exemplos práticos de expressividade na programação, trará ao conhecimento do leitor uma forma produtiva de programar, analisando um domínio específico da linguagem (*DSL - Domain Specified Language*), e as regras terminológicas que desenham as mini-linguagens que tornam o contexto expressivo.

As linguagens de computador têm o objetivo específico de traduzir o código para linguagem de máquina. Arquitetadas para facilitar o trabalho do homem, muitas vezes se tornam complexas e de difícil aprendizado. Devido a esta deficiência, são criadas ferramentas específicas para cada área do desenvolvimento.

Ferramentas nascem com objetivo de minimizar o esforço humano. Para codificação, existem inúmeros *Frameworks* que seguem este objetivo. Através da abstração de problemas comuns e adoção de soluções padronizadas, são desenhadas soluções que tornem o desenvolvimento de software produtivo.

3 *Evolução da linguagem de programação*

A evolução dos computadores trouxe dispositivos menores e mais potentes. Na década de 80, usavam-se grandes computadores para realizar pequenos processos, trinta anos mais tarde estes dispositivos ganharam velocidade, *design* e consomem pouca energia. Dispositivos que cabem na palma da mão, com apenas alguns toques ou cliques, tornam acessível a informação desejada.

Da mesma forma, os computadores ganharam potência, as linguagens de programação se tornaram expressivas e humanas. Tal dinamismo não tem o objetivo de trazer conforto à máquina, mas sim ao seu manipulador, o homem.

A linguagem de computador, inicialmente era grosseira e de difícil compreensão, com o passar do tempo, as técnicas foram evoluindo, ganhando forma e expressão. Houve uma percepção de mudança, que tornaria a linguagem de programação uma auxiliadora do programador e não uma interpretadora.

Uma linguagem tem os seus limites, na maioria dos casos são formais e burocráticos. A linguagem Ruby quebra esta formalidade, torna o processo de codificação simples e livre. Em outras palavras, ela não bloqueia o trabalho do programador.

3.1 Exemplo 'Hello world'

O programa Hello world é o programa mais conhecido no mundo inteiro e é o exemplo mais básico de uma linguagem de programação, com o objetivo de imprimir a mensagem "Hello, world!" e guiar o iniciante em sua primeira compilação/execução de um programa de computador. Abaixo seguem dois exemplos "Hello, world" em duas linguagens de programação distintas: asssembly e ruby.

3.1.1 Exemplo usando a linguagem de programação *assembly*

Assembly ou *Assembler* é uma notação escrita por humanos que permite escrever instrução de máquina de forma mais fácil. *Assembly* paga o preço da complexidade pelo reduzido número de comandos e opções.

Listing 3.1: Exemplo em *assembly*

```
1  variable:
2      .message    db    "Hello world!\$"
3  code:
4      mov   ah,9
5      mov   dx,offset .message
6      int   0x21
7      ret
```

Sem dificuldades, a listagem acima 3.1 demonstra-se complexa. Apenas para imprimir uma frase foram necessárias 6 linhas. Essa declaração não tão amigável é usada para realizar o mesmo objetivo do comanddo abaixo **??**, apenas estão escritas de forma diferente. O fato é que os compiladores exigem uma quantidade mínima de declarações para que possa compilar e executar.

3.1.2 Exemplo usando a linguagem de programação *ruby*

Listing 3.2: Exemplo em *ruby*

```
1  print "Hello, world!"
```

Ruby demonstra-se mais simples nestas exigências, é possível compilar apenas uma expressão, não precisando de declarações e nem padrões repetitivos. O código possui apenas o comando de impressão e a frase que deseja ser impressa.

3.2 A simplicidade da linguagem

Foi necessário apenas uma linha de código para representar o mesmo exemplo na linguagem *ruby*. No exemplo mais simples, o objetivo é apenas imprimir "Hello, world!" e é exatamente isso que está escrito. Diferente de *assembly*, *ruby* não é uma linguagem compilada e, sim, dinâmica. Assim, tudo acontece em tempo real, enquanto está sendo executado.

Listing 3.3: Tradução do programa *ruby*

```
1  imprima "Ola mundo!"
```

Apenas com uma linha de código é possível fazer exatamente o que está sendo proposto. Este programa de computador foi escrito de forma simples e humanamente legível, diferente da primeira instrução de máquina escrita em *assembly*. Com poucas palavras, cumpriu exatamente o objetivo do software no domínio em questão.

Exemplos como este, mostram o poder do homem em categorizar e generalizar as informações. Desta forma, a percepção mudou de 'programar para o computador' para 'programar para as outras pessoas'. Anteriormente, com uma programação rígida e a escassez de processamento, a codificação de um software realmente fazia parte de um processo árduo e lento, em que não era possível tornar agradável a leitura de uma instrução de computador.

3.3 A burocracia da linguagem

Declarar e inicializar variáveis, importar pacotes, classes finais e fechadas são fatores que interferem no trivial desenvolvimento de software. As linguagens estáticas empurram o programador a escrever mais código, forçam a declarar estruturas sem sentido, por não saber lidar com o problema em 'alto nível'.

Listing 3.4: Programa Hello World em Java

```
1  public class Hello {
2      public static void main(String[] args) {
3          System.out.println("Hello, World!");
4      }
5  }
```

Para ser possível executar um código em java, este deve estar em uma classe Java, em arquivo na extensão .java. A classe deve ser pública e deverá conter um método estático, sem retorno (**void**) com o nome **main**. Este método receberá como parâmetro um vetor de *strings* que representam os parâmetros da linha de comando. Para compilar o arquivo java, será necessário usar o compilador **javac** que então criará um arquivo com o mesmo nome do código fonte, com a extensão .class. Esse arquivo pode ser executado usando o comando **java**.

Todos estes passos devem ser seguidos sem falhas para que o exemplo "Hello, world!" seja compilado e executado usando a linguagem java. Cada detalhe deste tem um motivo e

é importante em algum aspecto na linguagem. Mas, em muitos momentos, estes recursos se tornam desumanos e complexos, ofuscando a expressividade da linguagem.

4 A linguagem ruby

Ruby é uma linguagem dinâmica, *open source*, com foco na simplicidade e produtividade. Tem uma sintaxe elegante, de leitura natural e fácil escrita. Com um grande número de bibliotecas disponíveis gratuitamente (gems), tem atraído muitos desenvolvedores.

Para usá-lo no *MacOSX* ou *Linux*, é necessário abrir o **Terminal** ou **Console** e, no Windows, pode-se usar o **Command-DOS** e, após, digitar **ruby.exe** para invocar o compilador, **irb** para iniciar uma interação.

4.1 O compilador *ruby*

Ruby é uma linguagem de *script*, dinâmica e compila em tempo de execução. O mesmo compilador que compila o código também o executa. O compilador pode ser invocado usando a linha de comando e receber apenas um fragmento de código por parâmetro para compilar.

Listing 4.1: Usando o compilador na linha de comando

```
1 jonatas@xonatax-mac:~$ ruby -e "puts 1+2"
2 3
```

Também é possível escrever um arquivo ou vários

4.2 *Ruby* Interativo - IRB

Existe um programa chamado IRB, que vem instalado com compilador de *ruby* e que serve para interagir com a linguagem na forma de console. As linhas do console *ruby* digitadas são iniciadas por » e as linhas que trazem o resultado da expressão são iniciadas por **=>**. Estes símbolos, quando exibidos no início da linha, não fazem parte do código de programação, apenas identificam a situação do terminal interativo.

Listing 4.2: Usando o compilador na linha de comando

```
1 jonatas@xonatax-mac:~$ irb
2 >> 1.class
3 => Fixnum
4 >> 1 + 2
5 => 3
```

4.3 Tipos de dados

Tudo em *ruby* é objeto, todas as classes são abertas e podem ser alteradas ou extendidas. A seguir, serão abordados os tipos básicos de dados para trabalhar com *ruby*.

4.3.1 *Strings*

As *strings* podem ser limitadas por vários delimitadores, em geral, são usadas aspas simples e duplas, mas existem outras formas. Uma *string* é um dado que contém uma informação em formato texto. Em *ruby* é encarado como uma cadeia de caracteres, e pode ser também fácilmente manuseado como um vetor de caracteres.

Listing 4.3: Exemplos de uso de *string*

```
1 >> "␣outra␣string" + '␣mais␣uma' + %[ outra ]
2 => "␣outra␣string␣mais␣uma␣outra␣"
```

4.3.2 Números

Os números são objetos, como qualquer outro, e os operadores são simples métodos.

Listing 4.4: Exemplos de uso de números

```
1 >> 1.class
2 => Fixnum
3 >> 1.class.ancestors
4 => [Fixnum, Integer, Precision, Numeric, Comparable, Object, Kernel]
5 >> 1.0.class
6 => Float
```

4.3.3 *Hashes*

Este tipo de elemento amarzena valores como uma matriz, mas é possível acessar os valores através de uma chave. O valor do elemento é indicado a partir da expressão:

Listing 4.5: Syntaxe do hash

```
1 { :chave => valor }
```

Tanto chave quanto valor podem ser objetos de qualquer tipo. Lembrando que a chave é a forma de acessar um valor, logo não poderá haver chaves iguais na mesma estrutura.

Listing 4.6: Exemplos de uso de hashes

```
1 sexo = { "M" => "Masculino", "F" => "Feminino"}
2 sexo["M"] # => "Masculino"
3
4 pessoa = { :nome => "jonatas", :idade => 22, :cpf => "047..." }
5 pessoa[:rg] # => nil
```

Quando invocada uma chave que não existe, a expressão retorna **nil** ou **nulo**. **Nil** também é um objeto em *Ruby* e é um objeto falso, o que auxilia no tratamento de excessões simples como requerir o documento de **cpf** ou **rg** da variável **pessoa** declarada acima4.6 :

Listing 4.7: Exemplos de uso de hashes com operador ou

```
1 documento = pessoa[:rg] || pessoa[:cpf] # => "047..."
2 print "sem␣documento!" if documento.nil?
```

No caso acima, quando não houver **rg**, irá buscar por **cpf** e atribuir a **documento**. Se não encontrar nenhum dos dois, então irá imprimir a mensagem avisando a falta do documento.

4.3.4 Entrada de dados

No exemplo abaixo, será feita uma pergunta para o usuário e, quando ele confirmar, a resposta irá saudar o usuário.

Listing 4.8: Exemplo de entrada de dados

```
1  print "qual␣seu␣nome?"
2  nome = gets
3  print "oi␣#{nome}"
```

Na primeira linha, é impressa a pergunta. Na próxima linha, a variável **nome** recebe o valor digitado pelo usuário, que é captado através do comando **gets**.

Em seguinda, é impressa a saudação juntamente com a variável **nome**. O uso da interpolação #{} embutido na string concatena o resultado da expressão interna invocando o método to_s(transforma em *string*).

4.3.5 Métodos

Os métodos podem ser usados a partir de uma classe ou estarão definidos na classe **Object** que é a classe maior. Métodos são formas de interagir com o ambiente atual e encapsulam a função com parâmetros. Eles podem pertencer às classes ou às instâncias.

Listing 4.9: Exemplo de método

```
1 def ligar_para(telefone)
2   puts "ligando para #{telefone}"
3 end
```

4.3.6 Classes e Módulos

Classe é a declaração que permite definir uma classe marcado pela palavra *class*, seguida do nome de elementos. Após isso, vem o corpo com as definições do corpo. O fim do corpo é marcado pela palavra 'end'.

Módulos permitem dividir comportamentos de classe e de instância, permitindo assim a múltipla extensão de classes. Ele pode ser usado em blocos e sub-blocos e todo o código gerado fica encapsulado na declaração. Esse recurso também pode ser usado para agrupar classes do mesmo gênero.

Listing 4.10: Exemplo de módulo

```
1   module Mamifero
2     def mamar
3       print "mamando..."
4     end
5   end
6   class Gato
7     include Mamifero
8     def miar
```

```
9           print "meauuuu"
10        end
11      end
```

Os módulos também são uma boa possibilidade para criar espaços de nomes. Com este recurso, é possível trabalhar com nomes específicos e sem repetições.

No exemplo acima, é definido um módulo **Mamifero**, cujas características internas estão sendo declaradas. Um módulo não pode ser instanciado, ele é apenas usado como um espaço de nomes. O objetivo dele é organizar pequenos comportamentos que podem ser utilizados em outros momentos.

Listing 4.11: Exemplo de módulo como espaço

```
1 module Condominio
2   include PortaoEletronico
3   class Apartamento
4    # ... define o apartamento internamente
5   end
6 end
```

4.3.7 Variáveis

Quando iniciados com apenas um arroba (@), indicam ser uma variável de instância. Diferente de Java e muitas outras linguagens, não é necessário declarar a variável antes de usá-la. Elas, simplesmente, criam a referência logo na primeira vez em que são usadas.

As variáveis locais são iniciadas por letras ou *underscore*.

A atribuição de um valor a uma variável é feita através do sinal de igualdade (=).

Listing 4.12: Exemplo de variável local

```
1 meu_nome = "Jonatas"
```

A leitura da expressão acima seria: meu nome (a variável) é igual a Jonatas.

Em uma classe, as váriaveis são usadas como atributos internos.

Listing 4.13: Exemplo de variável de instância em uma classe

```
1 class Pessoa
```

```
2     def initialize(nome)
3       @nome = nome
4     end
5     def saudar
6       puts "ola␣#{@nome}"
7     end
8   end
```

O código anterior declarou a classe, que pode ser utilizada da seguinte forma:

Listing 4.14: Exemplo de utilização da classe descrita acima

```
1 jonatas = Pessoa.new "Jonatas"
2 jonatas.saudar
```

4.3.8 Blocos de código

(2) Segundo Dave Thomas, todo mundo já quiz implementar o seu próprio recheio de método na sua própria estrutura. Um bloco pode ficar entre chaves ou entre as palavras *do* e *end* .

Listing 4.15: Exemplo de bloco de código

```
1     5.times { print "hello␣word" }
```

Traduzindo:

Listing 4.16: Exemplo de bloco de código

```
1     5.vezes { imprima "ola␣mundo" }
```

O bloco de código é extensamente utilizado em *frameworks* para apenas criar a estrutura e tornar óbvio o objetivo da linguagem.

5 *Shoes*

5.1 O que é Shoes?

Shoes é um *framework* para a construção de interfaces rápidas. *Shoes* nasceu pra ser fácil. Realmente, feito para iniciantes absolutos. Com esta ferramenta, é realmente fácil de fazer interfaces e artes gráficas.

Esta ferramenta permite criar interfaces *desktop* para todas as plataformas e muitos divertidas de aprender.

5.2 Primeiro exemplo

Este é um dos exemplos mais simples do *shoes*.

Listing 5.1: Primeiro exemplo do *framework Shoes*

```
1 Shoes.app {
2   button("Click␣me!") {
3       alert("Good␣job.")
4   }
5 }
```

Este programa foi escrito em uma linguagem chamada *Ruby*. Consiste em uma janela com um botão. Quando o botão for clicado, ela deverá responder por: "*Good job.*" ou seja "Bom trabalho."

Shoes roda na maioria das plataformas operacionais. Isto é ótimo pois é possível escrever apenas uma vez e usar no *Linux, MacOSX, Windows* e muitos outros.

5.2.1 Exemplo de expressividade - um bloco de notas com *Shoes*

Como próximo exemplo, será progamado um bloco de notas que possa apagar e inserir novas notas em uma janela. O programa será composto por:

- uma janela com um título "Minhas Notas", largura de 300 pixels. E dentro desta janela terá
  - uma linha de edição para escrever a nota
  - um botão para adicionar a anotação que quando for clicado deve
    - adicionar o que foi escrito na linha de edição para as notas abaixo listadas
    - limpar o texto da linha de edição da nota
- cada nota adicionada deve conter um link para remover a nota ...

5.2.2 Funcionamento do *framework*

Como descrito na primeira linha de código do exemplo anterior, o *framework* é declarado, contendo uma janela principal. Esta janela, recebe um título e um tamanho inicial.

A declaração:

Listing 5.2: Um bloco de anotações com Shoes

```
1  Shoes.app :title => "Minhas␣Tarefas", :width => 300 {
2    @anotacao = edit_line
3    button "adicionar" {
4      @notas.append {
5        para(@anotacao.text + "␣" +
6             link("apagar") {|nota| nota.parent.remove })
7      }
8      @anotacao.text = ""
9    }
10   @notas = stack :margin => 20, :width => 300
11 }
```

Shoes.app inicia um aplicativo do *framework* e, dentro do corpo deste método, é possível empilhar e enfileirar objetos. Com suas definições específicas de janela, também pode receber parâmetros. Este objeto pode encaixar muitos outros componentes internamente. Também é

possível manipular elementos de fora para dentro e de dentro para fora, fazendo com que um bloco interfira no outro, sem dificuldades.

Como o exemplo acima mostra, o *framework Shoes* basicamente é uma pilha de componentes, podendo adicionar, empilhar e remover elementos com facilidade e clareza. Estes elementos, quando empilhados, podem interagir apenas empilhados, ou empilhados e aninhados a outras pilhas de elementos.

Listing 5.3: Primeira explicação do *framework Shoes*

```
1 Shoes.app :title => "Minhas Tarefas", :width => 300 {
```

Estes parâmetros como **título *(title)*** e **tamanho *(width)*** são pertencentes à janela principal do aplicativo e podem ser acompanhados de outros, como bloqueio de redimensionamento de janela.

Os parâmetros no formato **chave** e **valor** facilitam o uso e a extensão das possibilidades para cada elemento. Infinitamente, podem trabalhar com novas funcionalidades e com os valores padrão.

Na segunda linha do código existe a declaração do elemento da interface edit_line que é responsável pela linha de edição que permite a entrada de dados.

Listing 5.4: Entendendo a linha de edição

```
2   @anotacao = edit_line
```

Este método edit_line retorna um componente do *Shoes* do tipo 'caixa de entrada de texto', e é possível acessar o valor do texto digitado através do atributo **text**.

Listing 5.5: Entendendo o botão e sua ação

```
3   button "adicionar" {
4     @notas.append {
5       para(@anotacao.text + " " +
6             link("apagar") {|nota| nota.parent.remove })
7     }
8     @anotacao.text = ""
9   }
```

No fragmento acima, é declarado o botão que faz. Muito semelhante à especificação do programa *Shoes* proposto, este fragmento cumpre a seguinte parte:

- adicionar o que foi escrito na linha de edição para as notas (linhas 5 e 6)
- cada nota adicionada deve conter um link para apagar a nota (linha 6)
- limpar o texto da linha de edição da nota (linha 8) ...

Traduzindo, literalmente, a declaração do botão:

Listing 5.6: Botão adicionar - código traduzido

```
1  botao "adicionar" {
2    @notas.adicionar {
3      paragrafo(@anotacao.texto + "␣" +
4          link("apagar") {|nota| nota.pai.remover })
5    }
6    @anotacao.texto = ""
7  }
```

Um botão adicionar **(botao "adicionar")**, que faz com que o texto da anotação **(@anotacao.texto)** seja adicionado sob as notas **(@notas.adicionar)** já existentes. Quando clicado no link de apagar **(link("apagar"))** o elemento **(nota)**, este referencia-se ao seu 'pai' que o remove como 'filho' **(nota.pai.remover)**, desaparecendo da pilha de anotações **(@notas)**.

Listing 5.7: Entendendo a pilha de notas

```
10  @notas = stack :margin => 20, :width => 300
```

Traduzind, literalmente, a linha 10:

Listing 5.8: Entendendo a pilha de componentes

```
1  @notas = pilha :margem => 20, :largura => 300
```

A variável **@notas** é igual a uma pilha de componentes *shoes* com margem de 20 pixels, e largura de 300 pixels. Essa pilha não exibe nada na tela até que não seja adicionado nenhum elemento nela. Quando o botão adicionar for pressionado, então ela é usada para adicionar o texto escrito na **nota**. Observe que foi usada a variável em um bloco de código, antes mesmo de ela existir. Se o botão for clicado antes da pilha de notas ser criada, então o sistema lançará uma merecida excessão.

Shoes trabalha com pilhas e fluxos (`stacks and flows`), esse conceito permite empilhar elementos e criar fluxos com outros. Eles não se misturam, mas podem interagir. A pilha

criada foi atribuida à váriavel **@notas** e foi definido uma margem e uma largura para esta pilha de elementos. O processo pode funcionar recursivamente dentro desta pilha, permitindo criar o mesmo fluxo interno e concatenar elementos com liberdade.

6 *Linguagens de domínio específico*

No domínio específico de construir telas, o *framework* tem características que ressaltam a sua expressividade. A simplicidade de fazer um botão ter uma ação é expressiva, no domínio de construir um botão com uma mensagem de alerta. É muito semelhante ao exemplo a seguir6.1. Se é um botão, consequentemente, terá uma mensagem e uma ação. Em uma linha de edição (**edit_line**), terá exatamente as propriedades de uma entrada de texto.

Listing 6.1: Simplicidade do botão

```
1 botao("mensagem") {  alerta("clicou␣no␣botao" }
```

Para desenhar uma linguagem de domínio específico, deve-se olhar para a forma mais fácil de expressar-se naquele contexto.

Cada elemento, em seu domínio específico, tem a sua forma de expressão. Por exemplo, existem várias opções para construir uma janela, esses elementos podem ser descritos através dos parâmetros do comando. Uma janela pode ter um título, um tamanho inicial, ser redimensionável, etc.

Listing 6.2: Expressividade da construção de uma janela

```
1 janela :largura => 300, :altura => 500,
2        :titulo => "Janela␣de␣exemplo" {
3
4   botao("exemplo")
5 }
```

Cada problema, no seu contexto, pode ser expressivo quando usando uma terminologia adequada para o domínio específico. A maioria das linguagens de programação dificulta a criação desta sintaxe dentro da própria linguagem. *Ruby* trabalha com estas linguagens de forma mais transparente. Leandro Heuert(3), abordou o exemplo da criação de uma receita de bolo no seu domínio específico da seguinte forma:

Listing 6.3: Expressividade de uma receita no seu domínio específico(3)

```
1 receita {
2   nome "Bala de Pinga"
3   ingrediente 1, 'kg', 'acucar cristal'
4   ingrediente 1, 'copo', 'pinga'
5   ingrediente 2, 'copos', 'agua'
6   ingrediente 2, 'envelopes', 'gelatina incolor'
7   ingrediente 1, 'envelopes', 'gelatina vermelha'
8 }
```

Os domínios específicos tornam o código estético e preciso. Yukihiro Matz Matsumoto, o criador da linguagem *Ruby*, pergunta, em uma entrevista a Dave Thomas, se as linguagens influenciam o ser humano. Ele discute o que é uma boa linguagem de programação, que maneira muda sua forma de pensar e como o faz um programador melhor.

Como japônes, logo pensa em japônes, mas tem que traduzir para inglês a sua forma de pensar. Ele argumenta que não existe uma maneira de programar em japônes, a linguagem simplesmente não permite, essa não seria uma boa linguagem para explorar como uma linguagem de programação. As linguagens devem fazer você pensar melhor e faze-lo um melhor programador.(2)

Em outra entrevista (5) com Matz, Matsumoto comenta sobre a importância de aprender vários tipos de linguagem de programação. Para entender a produtividade que a linguagem pode oferecer, é necessário conhecer os bons aspectos das linguagens de *script*, funcionais, lógicas, etc. Aprender linguagens é uma maneira de aprender a programar. Ler códigos fontes é uma maneira de obter conhecimento e informação. Também não deve-se dar foco às ferramentas, pois elas mudam. O importante é conhecer a base fundamental dos algorítmos.

Ser preguiçoso é uma fonte de inspiração para programar em *Ruby*, as máquinas servem aos homens e a elas pertence o esforço. Falando pouco e se expressando bem, é possível comandar computadores com produtividade.

Heuert(3) conclui que, quanto mais conhecido é o domínio específico, melhor será a solução. Uma linguagem pode ser desenhada com sucesso para um determinado escopo, pode ser comunicativa para as pessoas e para os computadores e expressiva como os exemplos aqui apresentados.

A atividade de trabalhar com a linguagem, uma forma de se comunicar com a máquina, deve ser divertida e, em primeiro lugar, de homem para homem, depois, de homem para máquina e,

por fim, de máquina para máquina(5). Em outras palavras, deve iniciar por uma definição compreensiva ao homem, depois deve ser comunicado para a máquina de forma fácil e, finalmente, a máquina pode compreender e se preocupar com suas tarefas de máquina.

Com uma linguagem bem desenhada, como Heuert citou "pode-se facilmente que pessoas que tenham um mínimo necessário de conhecimento em linguagens de programação consigam entender o quê está acontecendo por trás de um software"(3). Trabalhando com elementos conhecidos na contexto do programador, é possível que, mesmo sendo um iniciante, codifique sem dificuldades, já que conhece o domínio específico.

Existe um framework muito utilizado pelos programadores *ruby*, chamado *ActiveRecord*. Este *framework* tem o objetivo de encapsular o mapeamento objeto-relacional que permite a persistência de objetos em banco de dados e conversão de um registro de uma tabela para um objeto. O exemplo a seguir6.4 faz uma busca por um usuário ativo do sistema que possua o *login* igual a 'jonatas'.

Listing 6.4: Exemplo de busca com o framework ActiveRecord

```
1 User.active.find_by_login("jonatas")
```

Traduzindo na literal tem-se:

Listing 6.5: Exemplo traduzido de busca com o framework ActiveRecord

```
1 Usuario.ativo.busca_por_login("jonatas")
```

A expressão acima6.4 irá executar uma consulta no banco de dados que fará o filtro necessário pelo login e por usuários ativos e trará o objeto instanciado se algum registro for encontrado. Com este *framework* também é possível mapear relacionamentos entre outros objetos agregados.

Listing 6.6: Exemplo de classe extendida do ActiveRecord

```
1 class NotaFiscal < ActiveRecord::Base
2   validates_uniqueness_of :numero
3   validates_presence_of :cliente
4
5   belongs_to :cliente
6   has_many :itens
7 end
```

Quando uma classe é estendida do *framework ActiveRecord*, abre-se uma gama de métodos específicos para indicar os relacionamentos existentes, validar, filtrar, inserir, alterar atributos ou excluir. Também é possível criar observadores para diversos pontos e estados do objeto. Traduzindo a listagem anterior, tem-se a seguinte declaração:

Listing 6.7: Exemplo de classe do ActiveRecord traduzida

```
1 class NotaFiscal < ActiveRecord::Base
2   validar_unicidade_do :numero
3   validar_presenca_do :cliente
4
5   tem_muitos :produtos
6   pertence_a :vendedor
7 end
```

Em um *software* determinado, dá-se uma nota fiscal, que terá um número único, não permitirá ser lançada sem um cliente, terá muitos produtos e pertence a um vendedor. O exemplo acima apresenta dois tipos de métodos mapeadores das associações e os validadores.

A metaprogramação é o poder de estender a linguagem a partir da própria linguagem. Dave Thomas(2) ressalta que uma das coisas que mais chama atenção em *Ruby* é a facilidade de instrospecção à linguagem. Com este recurso, é possível criar objetos dinamicamente, é possível trabalhar com o inesperado criando métodos e classes de forma dinâmica, através da própria linguagem. Também é este o recurso que mais auxilia na definição do domínio específico da linguagem a ser desenhada.

7 *Considerações finais*

A programação de computadores é um trabalho árduo e de difícil compreensão. Mas, quando localizado em um domínio específico, pode-se apresentar simples e com fácil manutenção. A programação torna-se fácil, quando existe clareza nos objetivos e expressividade na linguagem escolhida. O conhecimento das regras e recursos da linguagem é crucial para obter resultados realmente expressivos.

A escolha das premissas e da terminologia no domínio específico não dependem apenas da escolha do programador, que irá arquitetar a solução, mas também dos recursos que a linguagem oferece. *Ruby* destaca-se na simplicidade de programar, na facilidade de impôr limites e desenhar linguagens internamente. Com os recursos aqui exemplificados, é possível escolher boas ferramentas, codificar com expressividade, escolher nomes convenientes e criar códigos de máquina mais humanos.

Referências

1 Wikipédia, Acesso em setembro de 2009.

http://pt.wikipedia.org/wiki/Linguagem

2 Thomas, D. Programming Ruby: The Pragmatic programmers Guide. Segunda edição. Dallas, Texas: The Pragmatic Bookshelf, 2006.

3 HEUERT, L. Linguagens Específicas de Domínio: um exemplo prático em Ruby. Francisco Beltrão: UNIPAR, 2008.

4 Thomas, Dave; Hansson, David Heinemeier. Agile Web Development with Rails Second Edition. Dallas, Texas: The Pragmatic Bookshelf, 2007.

5 Ruby Creator Yukihiro Matsumoto. Interview by Stephen Ibaraki, I.S.P. http://www.stephenibaraki.com/rubycreater122101.htm Acesso em novembro de 2009.

6 Matsumoto, Y. Ruby Design Principles Technometria with Phil Windley, Acesso em novembro de 2009. http://itc.conversationsnetwork.org/shows/detail1638.html

7 Gageot, David. We cant write expressive code, Acesso em novembro de 2009. http://blog.javabien.net/2009/01/12/can-we-really-write-expressive-code

8 Venners, B. The Philosophy of Ruby A Conversation with Yukihiro Matsumoto http://www.artima.com/intv/ruby.html Acesso em novembro de 2009.

9 Lyons, John. Linguagem e Linguística, Inglaterra: Universidade de Sussex, 1981.

10 Nazemi, Kate. Expressive Code Massachusetts College of Art and Design, 2005.

http://www.dynamicmediainstitute.org/projects/expressive-code