UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

UMA DSL INTERNA EM RUBY PARA A ESCRITA DE DOCUMENTOS LaTeX

TRABALHO DE GRADUAÇÃO

Breno Simonetti Portella

Santa Maria, RS, Brasil

2013

UMA DSL INTERNA EM RUBY PARA A ESCRITA DE DOCUMENTOS LaTeX

Breno Simonetti Portella

Trabalho de Graduação apresentado ao Bacharelado em Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Orientador: Prof. Dr. Eduardo Kessler Piveta

Trabalho de Graduação N. 347
Santa Maria, RS, Brasil

2013

Universidade Federal de Santa Maria
Centro de Tecnologia
Bacharelado em Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

UMA DSL INTERNA EM RUBY PARA A ESCRITA DE DOCUMENTOS LaTeX

elaborado por
Breno Simonetti Portella

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Eduardo Kessler Piveta, Dr.
(Presidente/Orientador)

Márcia Pasin, Drª. (UFSM)

Iara Augustin, Drª. (UFSM)

Santa Maria, 20 de Fevereiro de 2013.

AGRADECIMENTOS

Primeiramente quero agradecer à minha família por ter sempre me dado apoio a todas as minha decições. À minha namorada Anelise que está sempre do meu lado até nos meus momentos de insuportável *nerdice*. Gostaria de agradecer ao professor Eduardo Piveta pela orientação do trabalho. Não poderia deixar de agradecer a todos que conheci durante estes longos anos de curso e que de certa forma colaboraram para a minha formação.

"Always Look on the Bright Side of Life"
— Do filme Monty Python's Life of Brian

"Don't Panic"
— Do livro O Guia do Mochileiro das Galáxias

// RESUMO

Trabalho de Graduação
Bacharelado em Ciência da Computação
Universidade Federal de Santa Maria

UMA DSL INTERNA EM RUBY PARA A ESCRITA DE DOCUMENTOS LATEX
AUTOR: BRENO SIMONETTI PORTELLA
ORIENTADOR: EDUARDO KESSLER PIVETA
Local da Defesa e Data: Santa Maria, 20 de Fevereiro de 2013.

Linguagens Específicas de Domínio (DSLs) são projetadas para resolver problemas específicos de um determinado domínio. O desenvolvimento de DSLs tem ganhado espaço na comunidade Ruby dada a grande flexibilidade de sua sintaxe e de seus poderosos recursos para programação dinâmica.

LATEX é uma linguagem de marcação para a criação de documentos de alta qualidade tipográfica. Seu principal objetivo é separar o texto da apresentação, deixando o autor focado no conteúdo.

Em razão do dinamismo da linguagem Ruby e da importância do LATEX, este trabalho tem como objetivo desenvolver uma linguagem específica de domínio interna para a criação de documentos em LATEX. Ao final do desenvolvimento obteve-se uma linguagem que é capaz de criar documentos de baixa complexidade e que exige do usuário conhecimentos básicos de linguagens de programação.

Palavras-chave: DSL. Ruby. LATEX.

ABSTRACT

Undergraduate Final Work
Undergraduate Program in Computer Science
Federal University of Santa Maria

AN INTERNAL DSL IN RUBY TO WRITE LaTeX DOCUMENTS
AUTHOR: BRENO SIMONETTI PORTELLA
ADVISOR: EDUARDO KESSLER PIVETA
Defense Place and Date: Santa Maria, February 20st, 2013.

Domain Specific Languages are designed to solve specific problems of an specific domain. The development of DSLs has gained ground in the Ruby community given its great syntax flexibility and your powerfull tools for dynamic programming.

LaTeX is a document markup language to create hight quality typographic documents. Its main goal is to isolate the structure of a document from its layout, leaving the author focused in the content.

Due to the dynamism of Ruby language and the importance of LaTeX, this work has the objective of developing an internal domain specific language for writing LaTeX documents. At the end of development obtained a language that is capable of creating low complexity documents and that requires from users basic knowledge of programming languages.

Keywords: DSL. Ruby. LaTeX.

FIGURAS

2.1 Variáveis de instância ... 19
2.2 Classes são objetos ... 20
2.3 Métodos de instância de *Class* ... 20
2.4 Hierarquia de classes ... 21
2.5 Cadeia de busca ... 21
2.6 *Dynamic Dispach* ... 22
2.7 *Dynamic Method* ... 22
2.8 *Ghost Methods* ... 23
2.9 *Blocks* ... 23
2.10 *Bindings* ... 24
2.11 Comandos básicos em LaTeX ... 24
2.12 Comandos básicos em LaTeX ... 25
3.1 Exemplo de um documento simples na DSL ... 29
3.2 Método *Create* da classe *MyLatex* ... 29
3.3 Código LaTeX gerado a partir do exempo básico 3.1 ... 29
3.4 Método *command* ... 31
3.5 Inserindo código *inline* ... 31
3.6 Inserindo pacotes ... 32
3.7 Inserindo cabeçalhos ... 33
3.8 Código LaTeX gerado do cabeçalho do exemplo 3.7 ... 33
3.9 Adicionando padrões de cabeçalhos ... 34
3.10 Incorporando arquivos ... 34
3.11 Adicionado Imagens ... 35
3.12 Código LaTeX gerado da imagem do exemplo 3.11 ... 35
3.13 Tabela na DSL ... 37
3.14 Código LaTeX do exemplo 3.13 ... 38
3.15 Bibliografia com a DSL ... 39
3.16 Código BibTex gerado do exemplo 3.15 ... 40
3.17 Criando comandos com a DSL ... 41
3.18 Código LaTeX do exemplo 3.17 ... 41
3.19 Adicionando código aos textos ... 42
3.20 Código LaTeX gerado do exemplo 3.19 ... 42
4.1 Preâmbulo do estudo de caso ... 44
4.2 Cabeçalho do estudo de caso ... 45
4.3 Textos do estudo de caso ... 46
4.4 comandos com parênteses ... 46
4.5 Bibliografia do caso de uso ... 47
6.1 MyLatex ... 51
6.2 Bibtex ... 61
6.3 RakeLatex ... 63

LISTA DE TABELAS

Tabela 3.1 – Parâmetros para imagens 36
Tabela 3.2 – Especificações para tabelas 36
Tabela 3.3 – Tabela gerada do exemplo 3.13 38
Tabela 6.1 – Lista de métodos da classe MyLatex 51

LISTA DE ABREVIATURAS E SIGLAS

DSL	*Domain-Specific Language*
LPG	Linguagens de Propósito Geral
CDF	*Cognitive Dimensions of Notations Framework*
WYSIWYG	*What-You-See-Is-What-You-Get*

SUMÁRIO

1 INTRODUÇÃO 13
1.1 Objetivos e contribuição do trabalho 13
1.2 Estrutura do Texto 14
2 REVISÃO BIBLIOGRÁFICA 15
2.1 Linguagens Específicas de Domínio 15
2.1.1 Definindo uma DSL 15
2.1.2 DSL externa, interna e *workbench* 17
2.1.3 Vantagens de usar uma DSL 17
2.1.4 Problemas com DSL's 18
2.1.5 Implementando uma DSL interna 18
2.2 Meta-programação em Ruby 19
2.2.1 *Object Model* 19
2.2.1.1 Variáveis de instância 19
2.2.1.2 Métodos 20
2.2.1.3 Classes 20
2.2.1.4 *Method Lookup* 21
2.2.2 Métodos Dinâmicos 21
2.2.2.1 *Dynamic Dispach* 22
2.2.2.2 *Dynamic Method* 22
2.2.2.3 *Ghost Methods* 22
2.2.2.4 *Blocks* 23
2.2.3 *Closures* 23
2.3 LaTeX 24
2.3.1 Noções Básicas 24
2.3.1.1 Comandos 24
2.3.1.2 Estrutura de um documento 25
2.3.2 Vantagens e Desvantagens do LaTeX 25
3 DSL EM RUBY PARA LaTeX 27
3.1 Visão geral 27
3.2 Comandos genéricos 30
3.3 Texto puro 31
3.4 Gerenciamento de pacotes 32
3.5 Cabeçalhos 32
3.6 Múltiplos arquivos 34
3.7 Imagens 34
3.8 Tabelas 35
3.9 Bibliografia com BibTex 38
3.10 Criação de comandos 40
3.11 Adição de código de linguagens de programação 41
3.12 Limitações 42
4 ESTUDO DE CASO 44
4.1 Preâmbulo 44
4.2 Cabeçalho 45
4.3 Textos e comandos 45
4.4 Bibliografia 46

5 CONCLUSÃO 48
REFERÊNCIAS 49
6 ANEXOS 51
6.1 Lista de Métodos 51
6.2 MyLatex 51
6.3 Bibtex 61
6.4 RakeLatex 63

1 INTRODUÇÃO

Uma linguagem específica de domínio, ou do inglês *domain-specific language* (DSL), é uma linguagem de programação projetada para resolver problemas específicos de um determinado domínio e, geralmente,é criada para funcionar apenas dentro desse domínio (DEURSEN; KLINT; VISSER, 2000).

O uso de DSLs não é algo novo. Os exemplos mais comuns incluem HTML, criada para especificar páginas WEB; SQL, para fazer consultas a bancos de dados; e expressões regulares projetadas para encontrar e manipular padrões específicos de texto.

As DSLs têm como vantagem reduzir o tamanho dos problemas, por exemplo. Sem a linguagem, seria mais trabalhosa a tarefa de consultar e manipular grandes listas de dados. Elas permitem às pessoas, que têm conhecimento do problema específico daquele domínio, possam aprender mais facilmente a linguagem.

De acordo com (LATEX, 2012), o LaTeX é um sistema de marcação para a editoração de documentos de alta qualidade tipográfica. A ideia principal do LaTeX é separar, o máximo possível, o texto do documento da apresentação, deixando o autor focado no conteúdo.

Um dos problemas do LaTeX é que muitos usuários não estão familiarizados com a linguagem e seus recursos. Em grandes grupos colaborativos nos quais os membros pertencem a diferentes organizações, ou em grandes grupos de pesquisa nos quais os membros possuem diferentes conhecimentos, fica difícil fazer que todos aprendam a linguagem. Outro problema é a dificuldade em depurar um documento LaTeX, pois, mesmo para os problemas mais simples, as mensagens de erro podem não ser intuitivas.

Para a implementação da DSL, foi escolhida a linguagem Ruby devido à sua sintaxe flexível e por ser uma linguagem dinâmica. Na linguagem Ruby existe uma forte cultura de desenvolvimento de DSLs: muitas bibliotecas em Ruby são desenvolvidas no formato de DSLs. Em particular, um dos frameworks mais conhecidos em Ruby, Ruby on Rails (RoR), trabalha usando uma coleção de DSLs.

1.1 Objetivos e contribuição do trabalho

Devido à flexibilidade ao dinamismo da linguagem Ruby e da importância do LaTeX para a criação de documentos de alta qualidade, este trabalho tem como objetivo desenvolver uma linguagem específica de domínio interna usando como suporte a linguagem Ruby para a criação

de documentos em LaTeX. Para atingir esse objetivo, é necessário o estudo de técnicas de meta-programação na linguagem Ruby, que são a base da implementação deste trabalho, bem como estudar implementações já existentes de DSLs em Ruby. É também necessário que a DSL não exija do usuário um grande conhecimento em linguagens de programação (embora algum seja necessário).

Dada a escassez de soluções de DSLs pra LaTeX, este trabalho tem como maior contribuição o desenvolvimento de uma ferramenta que auxilie na criação de documentos acadêmicos em LaTeX através de uma linguagem relativamente simples.

1.2 Estrutura do Texto

Este trabalho está estruturado da seguinte forma: No capítulo 2 é realizada uma revisão bibliográfica envolvendo técnicas de desenvolvimento de DSLs, meta-programação em Ruby e uma introdução ao LaTeX. No capítulo 3 é explicado como foi implementada a DSL e quais suas funcionalidades. No capítulo 4 é utilizado com artigo como estudo de caso. E no capítulo 5 são apresentadas as considerações finais.

2 REVISÃO BIBLIOGRÁFICA

2.1 Linguagens Específicas de Domínio

Neste capítulo é feita uma análise do que são DSLs: conceitos, características, tipos de DSLs, exemplos e tópicos gerais relacionados a implementação de DSLs internas.

2.1.1 Definindo uma DSL

DSL é um termo e um conceito que é assunto de debate. Determinadas linguagens são claramente classificadas como DSL, porém outras podem ser classificadas de diferentes pontos de vista. Entretanto, (DEURSEN; KLINT; VISSER, 2000) define uma terminologia:

> Uma linguagem especifica de domínio (DSL) é uma linguagem de programação ou de especificação que oferece, através de abstração e notação apropriada, um poder de expressão focado, e geralmente restrito, a um determinado domínio.

(FOWLER, 2011) define quatro elementos-chave para esta definição:

- **Linguagem de programação:** Uma DSL é usada por humanos para instruir um computador a realizar determinada tarefa. E assim como qualquer linguagem moderna, é preciso que ela tenha uma estrutura de fácil entendimento para humanos, mas que ao mesmo tempo seja executável por um computador.
- **Natureza da linguagem:** Uma DSL é uma linguagem de programação, e por isso deve possuir uma certa fluência onde a expressividade não vem somente para expressões individuais mas também da forma que elas podem ser organizadas como um todo.
- **Expressividade limitada:** Uma linguagem de propósito geral fornece diversos recursos: suporte a vários tipos de dados, controle e abstração de estruturas. Todos esses recursos são úteis, porém tornam a linguagem difícil de aprender e usar. Uma DSL dá suporte a um mínimo de recursos necessários para resolver os problemas de seu domínio. Não é possível construir sistema completo apenas com uma DSL. Porém é possível usar uma DSL para um aspecto em particular de um sistema.
- **Foco no domínio:** Uma linguagem limitada é útil somente se tem um claro foco em um pequeno problema. Esse foco é o que torna uma linguagem limitada ser útil.

As características das Linguagens de propósito geral (LPG) são amplamente conhecidas entre os desenvolvedores a ponto de serem consideradas como algo natural. Contudo, no

contexto das DSLs, o cenário é diferente. Por isso existe a necessidade de elencar suas características para defender o seu uso ao invés de uma LPG. Essa necessidade é justificada pelas pesquisas e estudos que ocorreram nos últimos anos (MERNIK; HEERING; SLOANE, 2005; KOSAR et al., 2008) que mostram a importância que essas linguagens têm representado para o desenvolvimento de software.

Cada DSL possui características próprias, entretanto é possível discriminar um conjunto de aspectos que estão presentes em quase todas e, a partir desse contexto, explicar cada uma delas. (PEREIRA et al., 2008) utiliza o *Cognitive Dimensions of Notations Framework* (CDF) para fazer essa análise das características, porém aqui a intenção é descrever e apontar razões para as características das DSLs sem levar em consideração o CDF.

Geralmente, DSLs são pequenas. Quando essas linguagens são projetadas para lidar com problemas específicos de um domínio, os projetistas podem escolher as características e conceitos daquele domínio, e assim, criar uma notação pequena e restrita. Essa notação permite a especificação de soluções para resolver os problemas do domínio ao invés de programá-los, o que ocorre quando se trabalha com LPGs. Logo, DSLs são mais declarativas ou descritivas do que linguagens imperativas (DEURSEN; KLINT; VISSER, 2000).

DSLs são abstratas (HUDAK, 1996) e expressivas (MERNIK; HEERING; SLOANE, 2005). Quando se analisa e projeta uma DSL, é preciso estar atento ao domínio onde a linguagem será aplicada. A semântica do domínio deve estar implícita na notação da linguagem (HUDAK, 1998), ou seja, a abstração deve ser levada para a notação da linguagem. Isso significa que detalhes de como algo deve ser feito precisam ser encapsulados em uma notação de alto nível permitindo assim os usuários criarem um mapeamento entre a sintaxe da linguagem e os objetos do domínio do problema.

O foco em definir uma notação que expresse conceitos de um domínio em particular traz a possibilidade de moldar a linguagem de forma a torná-la mais eficiente de diversas formas. Uma forma é torna-la fácil de ser lida e entendida por pessoas que possuam conhecimento do domínio (CONSEL et al., 2005), porém essas pessoas geralmente não possuem muito conhecimento em linguagens de programação. Porém, dada a abstração e expressividade das DSLs, elas podem facilmente ler programas e aprender a linguagem. Outra forma para aumentar a eficiência é encontrada nas ferramentas que são suporte para a linguagem. os processadores da linguagem, por exemplo, podem ser modificados pra obter melhores resultados já que o domínio é restrito e o conhecimento é centralizado.

2.1.2 DSL externa, interna e *workbench*

(FOWLER, 2011) divide as DSLs em três categorias principais: DSLs externas, internas e Language Workbenches:

- **DSL externa:** é uma linguagem separada da linguagem principal da aplicação no qual ela trabalha. Geralmente, uma DSL externa possui uma sintaxe própria, porém usar sintaxes de outras linguagens é algo comum (XML é uma escolha frequente). Um script em uma DSL externa geralmente é analisado por um código na linguagem hospedeira utilizando técnicas de *parsing* de texto. Exemplos de DSLs externas incluem expressões regulares, SQL, CSS, HTML, Awk e arquivos de configuração em XML como Struts e Hibernate;

- **DSL interna:** é um modo particular de usar uma LPG de forma que esta fique mais legível para um determinado fim. Um script em uma DSL interna é um código válido na LPG, porém utiliza somente um subconjunto de ferramentas da linguagem de uma forma particular para lidar com um pequeno aspecto do sistema. Como resultado, obtém-se uma linguagem que possui estilo próprio. Um exemplo clássico desse estilo é Lisp; programadores Lisp geralmente falam em desenvolvimento dentro de um contexto de criação e uso de DSLs. Ruby também possui uma forte cultura de desenvolvimento de DSLs: muitas bibliotecas em Ruby são desenvolvidas no formato de DSLs. Em particular, um dos frameworks mais conhecidos em Ruby, Ruby on Rails, trabalha usando uma coleção de DSLs (RAILS, 2012);

- ***Language Workbenches*:** são forma especializada de IDE (*Integrated Development Environment*) para definir e construir DSLs. Ela é usada não somente para determinar a estrutura de uma DSL, mas também como um ambiente de edição customizável para aqueles que forem escrever scripts em DSL;

2.1.3 Vantagens de usar uma DSL

(HUDAK, 1997) fornece cinco razões para se usar uma DSL:

- Aumento de produtividade durante o desenvolvimento;

- Código mais conciso;

- Facilidade de manutenção;

- São mais fáceis de se trabalhar;
- Pode ser escrita por não-programadores;

(HUDAK, 1997) destaca o último item, pois, não-programadores que possuem grande conhecimento do domínio, ajudam a diminuir a distância que separa desenvolvedores e usuários, potencialmente um dos grandes custos implícitos que envolvem o desenvolvimento de software.

2.1.4 Problemas com DSL's

(DEURSEN; KLINT; VISSER, 2000) cita algumas desvantagens no uso de DSL's:

- Custo para projetar, implementar e manter uma DSL;
- Custo para treinar os usuários da DSL;
- Dificuldade em encontrar um escopo adequado para a DSL;
- Dificuldade em balancear entre DSL e LPG;
- Potencial perda de desempenho quando comparado com um sistema desenvolvido inteiramente com uma LPG;

2.1.5 Implementando uma DSL interna

(FOWLER, 2011) apresenta diversas técnicas que podem ser usadas na implementação de uma DSL interna de forma a tornar a linguagem mais fluente e fácil de ser lida. Neste tópico, apresento três técnicas que foram utilizadas para o desenvolvimento deste trabalho. Elas são vistas em detalhes no tópico 2.2.

- *Object Scoping*: essa técnica faz uso das propriedades de um objeto onde é possível definir um escopo restrito para funções e dados. Herança permite usar esse escopo separado de onde ele é definido. Uma DSL pode usar essa facilidade para definir funções da DSL em uma classe principal e depois permitir que programadores possam escrever a DSL em subclasses.

  Essa técnica é usada de uma forma um pouco diferente na linguagem Ruby que utiliza a *instance evaluation* para executar código dentro do contexto de um objeto em particular.

- *Closure*: *closures* são basicamente blocos de código que podem ser representados como um objeto (ou estruturas de primeira classe) e são executados dentro de seu contexto.

- *Closures* aninhadas: Essa técnica usa as *closures* de forma aninhada combinando a estrutura hierárquica da funções aninhadas com a habilidade de controlar quando os argumentos serão avaliados.

2.2 Meta-programação em Ruby

Ruby foi originalmente planejado e desenvolvido no Japão por Yukihiro Matsumoto em 1995 (STEWART, 2012). É uma linguagem de programação dinâmica com uma complexa gramática e possui uma biblioteca de classes com uma extensa API. Ruby é influenciado por linguagens como Lisp, Smalltalk e Perl, porém utiliza uma gramática que é fácil para programadores C e Java aprenderem. Ruby é orientado a objetos, entretando também pode ser utilizado para programação funcional e imperativa (FLANAGAN; MATSUMOTO, 2008). Ruby ainda possui capacidades para meta-programação, capacidade que é discutida neste capítulo.

2.2.1 *Object Model*

Ruby é uma linguagem orientada a objetos, na qual mesmo tipos nativos como String, Float e Integer são objetos. Logo, para entender metaprogramação é preciso saber o básico da *Object Model* do Ruby, ou seja, como ela trabalha com objetos.

2.2.1.1 Variáveis de instância

Diferentemente da maioria das linguagens estáticas, em Ruby não existe conexão entre a classe do objeto e suas variáveis de instância. Essas variáveis somente tornam a existir quando é atribuído um valor a elas, então é possível ter objetos da mesma classe que carregam diferentes conjuntos de variáveis de instância. Por exemplo: se no código 2.1 não tivesse sido chamado *obj.meu_ metodo* então *obj* não possuiria variáveis de instância.

```
1 Class MinhaClasse
2   def meu_metodo
3     @v = 1
4   end
5 end
6 obj = MinhaClasse.new
7 obj.class #saida: MinhaClasse
```

```
8 obj.meu_metodo
9 obj.instance_variables #saida: [:@v]
```

Figura 2.1 – Variáveis de instância

2.2.1.2 Métodos

Além de variáveis de instância, objetos também possuem métodos. A maioria dos objetos em Ruby herdam um conjunto de métodos da classe *Object*. Um *obj* possui uma lista de métodos. Internamente, um objeto contem somente variáveis de instância e uma referência para a sua classe.

Objetos que compartilham a mesma classe também compartilham os mesmos métodos, logo métodos são armazenados na classe e não no objeto. É importante esclarecer que *meu_metodo* na linha 2 da Listagem 2.1 é uma instância de método de *MinhaClasse* o que significa que é definido dentro de *MinhaClasse* e que é preciso uma instância dessa classe para chamá-lo. É o mesmo método, porém, quando se refere a classe, é chamado de instância de método e quando for um objeto é chamado apenas de método.

2.2.1.3 Classes

Classes em Ruby são objetos e, consequentemente, tudo que se aplica a objetos também é aplicado para classes. Classes, assim como qualquer objeto, possuem sua própria classe chamada *Class* (2.2).

```
1 ola_mundo.class #saida: String
2 String.class    #saida: Class
```

Figura 2.2 – Classes são objetos

Os métodos de um objeto são também métodos de instância de sua classe, isso significa que métodos de uma classe são os métodos de instância da classe *Class* (2.3).

```
1 heranca = false
2 Class.instance_methods(heranca) #saida: [:superclass, :allocate
    , :new]
```

Figura 2.3 – Métodos de instância de *Class*

Os métodos *new* e *allocate* trabalham na criação de objetos, já *superclass* retorna a classe hierarquicamente superior.

Todas as classes acabam herdando da *Object* e essa herda da classe *BasicObject*, que está no topo da hierarquia. Também é possível verificar na linha 4 da Listagem 2.4 a superclasse de

Class.

```
1 String.superclass      #saida: Object
2 Object.superclass      #saida: BasicObject
3 BasicObject.superclass #saida: nil
4 Class.superclass       #saida: Module
5 Module.superclass      #saida: Object
```

Figura 2.4 – Hierarquia de classes

Após essa análise da *Object Model*, obtem-se o seguinte diagrama:

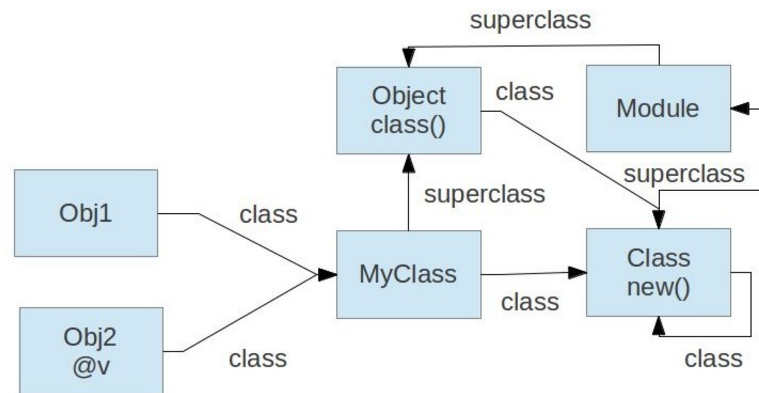

Figura 2.1 – *Object Model*

2.2.1.4 *Method Lookup*

Sempre que um método é executado, primeiramente ele é procurado na classe ao qual o objeto pertence, caso não seja encontrado, a busca é estendida para as superclasses. Essa estratégia é chamada de *one step to the right, then up* (PERROTA, 2010). Para visualizar esse caminho percorrido, pode-se chamar o método *ancestors*, como demonstrado na Listagem 2.5.

```
1 MinhaClasse.ancestors #saida: [:MinhaClasse, :Object, :Kernel,
     :BasicObject]
```

Figura 2.5 – Cadeia de busca

Observa-se que o módulo *Kernel* está incluído na cadeia da Listagem 2.5. Isso ocorre quando é incluído um módulo em uma classe, então o Ruby cria uma classe anônima para este módulo e o insere na cadeia.

2.2.2 Métodos Dinâmicos

Ruby fornece três técnicas para trabalhar com métodos dinâmicos:

2.2.2.1 *Dynamic Dispach*

Sempre que um método é invocado, uma mensagem é enviada a um objeto (PERROTA, 2010). Esse conceito fica claro com o método *send* na linha 8 na Listagem 2.6.

```
1 Class MinhaClasse
2   def meu_metodo(arg)
3     @v = arg
4   end
5 end
6 obj = MinhaClasse.new
7 obj.meu_metodo(3)         #saida: 3
8 obj.send(:meu_metodo, 3) #saida: 3
```

Figura 2.6 – *Dynamic Dispach*

Usar *meu_ metodo* ou *send* produz o mesmo efeito e nesse caso é mais prático usar *meu_ metodo*, entretanto *send* é útil quando precisa-se escolher qual método a ser chamado durante a execução do código.

2.2.2.2 *Dynamic Method*

Em Ruby é possível criar métodos durante a execução do código. Uma forma de fazer isso é utilizando o método *define_ method*.

```
1 Class MinhaClasse
2   define_method :meu_metodo do |arg|
3     arg += 1
4   end
5 end
6 obj = MinhaClasse.new
7 obj.meu_metodo(3)      #saida: 4
```

Figura 2.7 – *Dynamic Method*

Esse método recebe como argumentos o nome do método a ser criado e um bloco de código como por ser visto na linha 2 da Listagem 2.7.

2.2.2.3 *Ghost Methods*

Como explicado no tópico 2.2.1.4, sempre que um método é invocado, é feita uma busca primeiramente na classe do objeto em que ocorreu a invocação, caso o método não seja localizado, a busca se estende para as classes superiores. Se o método não é encontrado, é invocado o *method_ missing* localizado no módulo *Kernel*.

```
1  Class MinhaClasse
2    def method_missing(method, *args)
3      puts "Metodo chamado: #{method} - argumentos: #{args.join("
          ,")}"
4      puts "Um bloco foi chamado" if block_given?
5    end
6  end
7  obj = MinhaClasse.new
8  obj.ola_mundo(1,2,3) do
9    #um bloco de codigo
10 end
11 #saida: Metodo chamado ola_mundo - argumentos: 1,2,3
12 #Um bloco foi chamado
```

Figura 2.8 – *Ghost Methods*

É possível interceptar mensagens sobrescrevendo o método *method_ missing* da linha 2 na Listagem 2.8. Esse método recebe como argumentos o nome do método invocado, uma lista de argumentos e um bloco de código associado com a invocação. Essa técnica de chamar métodos que não existem foi amplamente usada no desenvolvimento deste trabalho.

2.2.2.4 *Blocks*

Um bloco é caracterizado por chaves ou pelas palavras-chave *do...end* e é definido somente quando um método é chamado. O bloco é passado direto através do método e, então, este pode chamar o bloco através da palavra-chave *yield*. Como pode ser visto na Listagem 2.9, um bloco pode receber argumentos (x e y). Assim como num método, um bloco pode fornecer valores para os seus argumentos e retornar o valor da última linha de código a ser executada.

```
1 def meu_metodo(a, b)
2   a + yield(a,b)
3 end
4 meu_metodo(1,2){|x,y| (x+y)*3} #saida: 10
```

Figura 2.9 – *Blocks*

2.2.3 *Closures*

Blocos não podem ser considerados apenas blocos de código sendo executados fora de contexto. Quando eles são executados, é preciso ter um ambiente com variáveis, variáveis de instância, etc. Esse conjunto de elementos no ambiente é chamado de vinculações (*bindings*) (PERROTA, 2010). A principal característica de um bloco é que ele vem pronto para ser executado e possuir esse conjunto de *bindings*.

Quando um bloco é definido, ele captura os bindings que foram definidos naquele contexto e os leva junto quando o bloco é passado para um método.

Observando a Listagem 2.10, quando um bloco é criado, ele captura as *bindings* locais (variável x) daquele contexto. e as carrega junto quando o bloco é passado para o método (que possui o seu próprio conjunto de *bindings*).

```
1 def meu_metodo
2   x = "ola"
3   yield("mundo")
4 end
5 x = "tchau"
6 meu_metodo{|y| "#{x} #{y}"} #saida: tchau mundo
```

Figura 2.10 – *Bindings*

2.3 LaTeX

LaTeX (LAMPORT, 1994) é um pacote de macros que permite aos autores processar e imprimir textos utilizado para criar documentos científicos e matemáticos de alta qualidade tipográfica. Entretanto, o sistema não fica limitado somente a documentos científicos. Ele pode ser usado também para produzir de simples cartas até livros. LaTeX utiliza como mecanismo de formatação o TeX (KNUTH, 1984) como seu mecanismo de processamento. TeX é um programa de computador criado por Donald E. Knuth (KNUTH, 1984). É utilizado para processamento de textos e fórmulas matemáticas. O TeX é reconhecido por ser extremamente estável, por funcionar em diferentes computadores e por ser virtualmente livre de erros (LAMPORT, 1994).

2.3.1 Noções Básicas

2.3.1.1 Comandos

Comandos em LaTeX iniciam por uma barra invertida e são formados por um nome construído inteiramente por letras. O nome dos comandos são terminados por um espaço, número ou qualquer outro caractere que não seja letra.

Determinados comandos necessitam de parâmetros adicionais que devem ser passados por chaves após nome do comando. Existem comandos que aceitam parâmetros extras que são passados em colchetes como o comando *documentclass* na linha 3 do exemplo 2.11.

```
1 \newline
2 \texsl{ola mundo}
```

```
3 \documentclass[11pt,twoside,a4paper]{article}
```

Figura 2.11 – Comandos básicos em LaTeX

2.3.1.2 Estrutura de um documento

Para o LaTeX processar um arquivo, é preciso seguir uma determinada estrutura. O arquivo de entrada, que geralmente possui a extensão *.tex*, possui um preâmbulo e um corpo. No preâmbulo estão comandos que definem parâmetros globais para processar o texto tais como o tipo de documento, formato do papel, altura e largura do texto, etc. É no preâmbulo que está o comando *documentclass[opções]{classe}* como pode ser visto na primeria linha do exemplo 2.12. Este comando vai determinar o tipo de documento a ser gerado. O parâmetro classe define o tipo de documento a ser gerado (*article*, *report*, *book* ou *slides*) e *opções* cria um padrão para o comportamento das classes do documento.

No preâmbulo também pode-se incluir novos pacotes através do comando *usepackage{...}* que adiciona novas funcionalidades ao sistema.

Após concluir a configuração do documento, inicia-se o corpo do texto através do comando *begin{document}*. Assim que o documento estiver finalizado, utiliza-se o comando *end{document}* para avisar ao LaTeX que termine o processamento. Dessa forma o LaTeX vai ignorar qualquer comando que tenha sido adicionado após a finalização do documento.

```
1  \documentclass[10pt,a4paper]{article}
2  \author{S. ~Breno}
3  \title{Um Exemplo}
4  \begin{document}
5     \maketitle
6     \tableofcontents
7     \section{Inicio}
8     Inicio do Exemplo
9     \section{Fim}
10    Fim do Exemplo
11 \end{document}
```

Figura 2.12 – Comandos básicos em LaTeX

2.3.2 Vantagens e Desvantagens do LaTeX

(OETIKER et al., 2002) e (LOVE, 2006) apresenta algumas vantagens e desvantagens do LaTeX em comparação com processadores de texto gráfico (*WYSIWYG*).

Vantagens:

- Disponibilidade de layouts criados profissionalmente;
- Suporte para fórmulas matemáticas de uma maneira conveniente;
- Separação da aparência do texto do seu conteúdo;
- O LaTeX encoraja a criação de textos bem estruturados;
- Funciona em qualquer plataforma de hardware que possua um compilador;

Desvantagens:

- O desenvolvimento de um layout inteiramente novo demanda muito tempo;
- Como LaTeX incentiva a separação da estrutura do texto de seu conteúdo, isso pode representar um problema para muitos não-programadores, pois não estão acostumados a trabalhar dessa forma;
- Por não ter uma interface gráfica, às vezes pode ser difícil de descobrir como realizar determinada tarefa;

3 DSL EM RUBY PARA LaTeX

Este capítulo descreve a DSL interna criada neste trabalho. No primeiro tópico é dada uma explicação geral de como está organizado o código para na sequência detalhar cada uma das funcionalidades da linguagem.

3.1 Visão geral

A DSL proposta neste trabalho é formada basicamente pela classe *MyLatex*, onde são implementadas todas as funcionalidades da linguagem, com exceção do suporte à bibliografia, que será discutido em detalhes no tópico 3.9. Na figura 3.1 tem-se uma visão geral da DSL: no item 1 está a classe principal *MyLatex* responsável por gerar os arquivos *.tex* e a *Bibtex* pelas referências bibliográficas. No item 2 temos um exemplo simplificado da DSL e da bibliografia. E em 3 o código resultante. A classe *RakeLaTex* (item 4) é um conjunto de tarefas *Rake*, que utiliza a ferramenta *rake4latex* (LICKERT, 2013), responsáveis por compilar o projeto em um determinado tipo de saída e também por calcular o número de compilações necessárias quando se trabalha com bibliografias.

LaTeX faz uso de uma grande quantidade de comandos onde cada comando pode receber diferentes tipos de argumentos e parâmetros, além disso, os mais diversos pacotes (CTAN, 2012) podem ser incluídos dando suporte a novas funcionalidades. Dado esse cenário, implementar separadamente cada uma das funcionalidades torna-se uma tarefa árdua, se não impossível. Para contornar esse problema, adotou-se a metodologia de métodos dinâmicos, mais precisamente a técnica de Ghost Methods (tópico 2.2.2.3).

A técnica de Ghost Methods foi utilizada com a finalidade de englobar o maior número possível de comandos LaTeX e é implementada pelo método *command*, que é visto em detalhes no tópico 3.2. Entretanto, nem todos os comandos podem ser tratados por essa técnica e para isso foram criados métodos específicos para tratar a criação de tabelas, imagens, cabeçalhos, criação de comandos, todos esses vistos em detalhes no tópicos subsequentes.

No exemplo 3.1 é dado um exemplo básico de como funciona a DSL. Antes de começar a adicionar comandos ao documento, é preciso chamar o método *create* passando um bloco de código que corresponde ao conjunto de comandos do documento. *Create* cria uma instância de *MyLatex* na linha 2 do exemplo 3.2 passando como argumento o formato do arquivo de saída e um bloco de código que será executado dentro do contexto do objeto *textLatex* através do

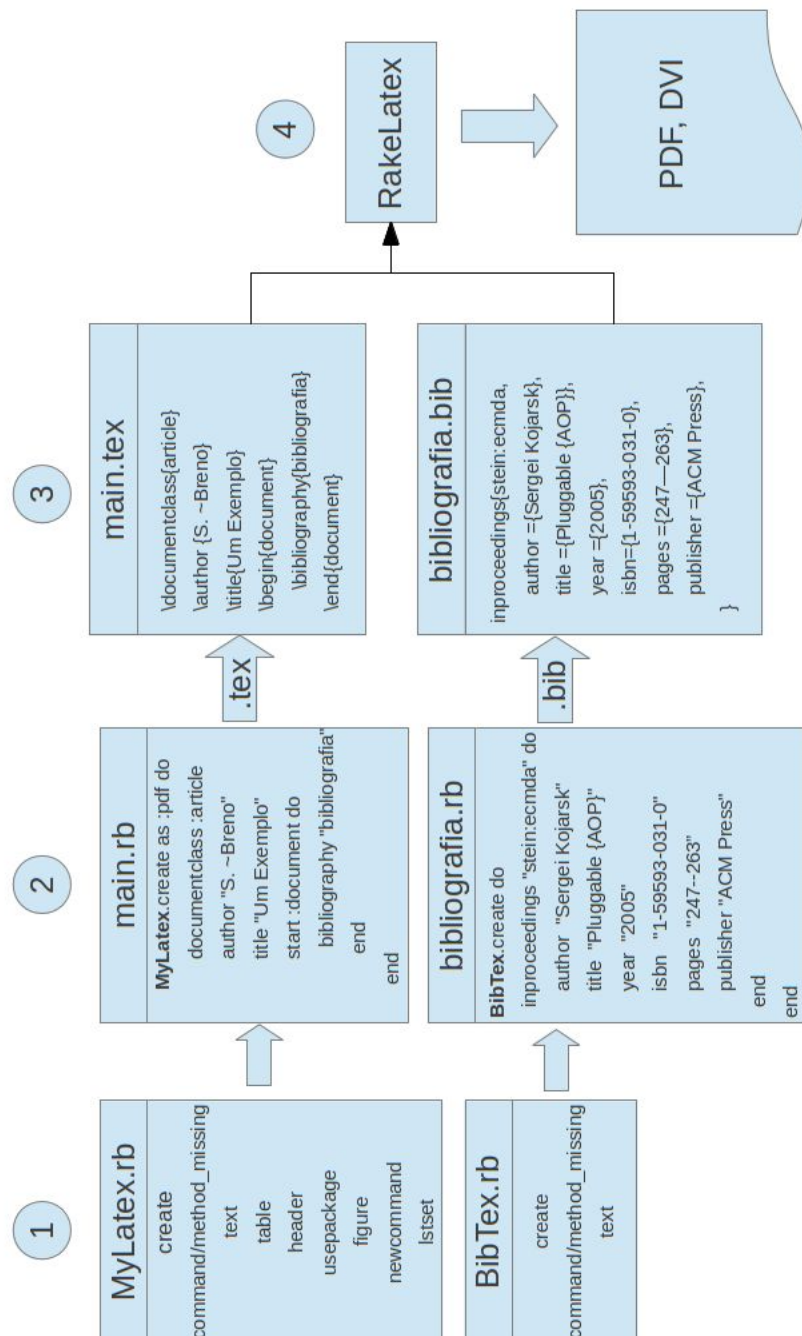

Figura 3.1 – Visão geral do projeto

método *instance_ eval*. Essa técnica é chamada de *Object Scoping* e foi apresentada no tópico 2.2.3. Todo comando (*author*, *start*, *title* etc) passado pelo bloco de código é encarado como uma chamada de método do objeto *textLatex*.

```
1  MyLatex.create as :pdf do
2    documentclass :article, opts: ["10pt", "a4paper"]
3    author "S. ~Breno"
4    title "Um Exemplo"
5    start :document do
6      maketitle
7      tableofcontents
8      section "Inicio"
9      text "Inicio do Exemplo"
10     section "Fim"
11     text "Fim do Exemplo"
12   end
13 end
```

Figura 3.1 – Exemplo de um documento simples na DSL

```
1   def self.create(output = {as: :pdf}, &block)
2      textLatex =  self.new
3      name = textLatex.get_file_name
4      textLatex.instance_eval(&block)
5      textLatex.dsl_to_latex(name)
6      RakeLatex.new(main_file: name, bibfile: textLatex.
          get_bib_file, type: output[:as]) if textLatex.
          main_document?
7      return textLatex
8    end
```

Figura 3.2 – Método *Create* da classe *MyLatex*

Sempre que um comando é chamado, a DSL armazena em um vetor o comando já convertido para a linguagem LaTeX e após todos os comandos serem executados, o método *dsl_ to_ latex* 3.2 fica responsável por gerar um documento *.tex* a partir dos comandos armazenados no vetor. Em 3.3 está o código LaTeX gerado a partir do exemplo básico 3.1.

```
1   \documentclass[10pt,a4paper]{article}
2  \author{S. ~Breno}
3  \title{Um Exemplo}
4  \begin{document}
5     \maketitle
6     \tableofcontents
7     \section{Inicio}
8     Inicio do Exemplo
9     \section{Fim}
10    Fim do Exemplo
11 \end{document}
```

Figura 3.3 – Código LaTeX gerado a partir do exempo básico 3.1

3.2 Comandos genéricos

Para a DSL oferecer suporte a um maior número de comandos LaTeX implementou-se a técnica de *Ghost Methods* (tópico 2.2.2.3) através do método *command*, que é na verdade um *alias* para o método *method_ missing* (tópico 2.2.2.3).

Sempre que quando um comando da DSL é executado é feita uma busca pelo método na classe *MyLatex* e, caso ele não seja encontrado, extende-se a busca para as classes hierarquicamente superiores (tópico 2.2.1.4). Se a busca não encontrar o método, é então invocado *command*. *Command* recebe como argumentos o nome do comando, uma lista de argumentos e um bloco de código (primeira linha da Listagem 3.4). A lista de argumentos é passada na forma de um vetor de dados genérico e é opcional assim como o bloco de código.

Para melhor entendimento usaremos o exemplo básico do item 3.1: No comando *documentclass* é passado como argumento um *Symbol* (*:article*) e uma *Hash* tendo como única chave *opts*. No lugar de *Symbol* poderia ser usada uma *String*, pois ambas compartilham muitas características, entretanto, ao contrário de *Strings*, *Symbols* não podem ser alterados (FLANAGAN; MATSUMOTO, 2008). Sempre que um comando precisa de parâmetros opcionais usa-se a *Hash* opts que recebe um vetor de parâmetros. Dentro do método *command*, os parâmetros adicionais são removidos da lista de argumentos através do método *get_ options* e armazenados na variável *options*. Neste caso não é passado um bloco juntamente com o comando, então ele é apenas convertido para um comando LaTeX válido em *else*. O método *check_ stack* verifica se um comando foi chamado de dentro de outro comando através de seus argumentos, essa verificação é necessária para que não haja duplicação de comandos no código LaTeX gerado. Depois dessa verificação, o comando em LaTeX é adicionado a uma pilha através do método *add_ to_ stack*..

Quando um comando possui um bloco de código, como o *start :document*, a palavra-chave *yield* é responsável por chamar os métodos que estão no bloco. Se os métodos do bloco chamado não tiverem sido definidos na classe, *command* será invocado novamente de forma recursiva. As variáveis *@ident, @n, @spaces* e *n* são apenas para gerar um código LaTeX com indentação.

```
1  def command(method, *args, &block)
2     options = get_options args
3     @ident = get_ident @n
4     if block_given?
5       add_to_stack(@ident + "\\begin{#{args.first}}#{options}")
6       n = @n
7       @n += @spaces
8       yield
9       @n = n
10      add_to_stack(get_ident(n) + "\\end{#{args.first}}")
11    else
12      old_command, method = clear_command method#tira os "_"
13      command = "\\#{method}"
14      params = options
15      args.each do |arg|
16         params << "{#{arg.join(",")}}" if arg.is_a? Array
17         params << "{#{arg}}" unless arg.is_a? Array
18      end
19      check_stack(params)
20      line = command + params
21      add_to_stack(@ident + command + params)
22      line
23    end
24  end
25  alias method_missing command
```

Figura 3.4 – Método *command*

3.3 Texto puro

Em muitos casos é preciso adicionar texto ao documento LaTeX sem precisar de algum comando que o anteceda. Para essas situações foi implementado o método *text*, pois a linguagem Ruby ignora qualquer literal que não seja atribuído a uma variável. Como exemplo, é possível observar os comandos *text* nas linhas 8 e 10 no exemplo 3.1. Esse método é também uma alternativa para os casos em que é preciso usar algum comando LaTeX que a DSL não suporta, dessa forma é possível adicionar código LaTeX *inline*. No exemplo 3.5 temos um código LaTeX *inline*.

```
1  text "\\comando{arg1}{arg2}"
```

Figura 3.5 – Inserindo código *inline*

3.4 Gerenciamento de pacotes

Para a inclusão de pacotes foi criado um método específico chamado *usepackage*. Este método armazena os pacotes em uma *Hash* onde cada chave corresponde a um pacote e o conteúdo da chave é um vetor de parâmetros. No exemplo 3.6 é mostrado um comparativo da DSL com o código gerado em LaTeX: Os pacotes podem ser adicionados na forma de um vetor de *Symbols* ou *Strings* (linha 1) ou um a um (linhas 2 e 3) e os parâmetros adicionais são passados através da Hash *opts*.

O método verifica se um pacote foi adicionado múltiplas vezes e, caso isso ocorra, é gerado o comando com apenas um pacote. Essa verificação é necessária, pois existem métodos como *table* e *figure* que adicionam pacotes automaticamente quando invocados e é preciso evitar duplicação caso eles sejam invocados múltiplas vezes.

```
1 usepackage [:latex8, :times, :balance]
2 usepackage :graphicx, opts: [:pdftex, :dvips]
3 usepackage :inputenc, opts: :latin1
```

Figura 3.6 – Inserindo pacotes

3.5 Cabeçalhos

Instituições e eventos científicos exigem que os artigos e teses enviados estejam de acordo com determinados padrões de formatação e estrutura do texto. Um estrutura em particular que muda de acordo com o formato é o cabeçalho do documento, onde são exibidos os nomes dos autores, contatos de email, título do documento e instituições as quais os autores fazem parte.

Para automatizar a formatação do cabeçalho foram implementados os métodos *header*, *institute*, *authors*, *location* e *emails*. Para melhor entendimento, observe o exemplo 3.7: o método *header* recebe como argumento o título do documento e o tipo de formato que será gerado. Para este exemplo a única formatação suportada é a do evento (SEKE, 2012). Para cada instituto é passado um bloco de código onde são especificados os autores, local da instituição e emails dos autores.

```
1  header "Avoiding Bad Smells in Aspect-Oriented Software",
      format: :SEKE do
2    institute do
3      authors "Eduardo K. Piveta", "Marcelo Hecht", "Marcelo S.
          Pimenta"
4      location "Inst. Informatica, Univ. Federal do Rio Grande
          do Sul - Porto Alegre, Brazil"
5      emails "epiveta@inf.ufrgs.br", "mvhecht@inf.ufrgs.br", "
          mpimenta@inf.ufrgs.br"
6    end
7    institute do
8      authors "Ana Moreira", "Joao Araujo", "Pedro Guerreiro"
9      location "Dept. Informatica, FCT, Universidade Nova de
          Lisboa - Caparica, Portugal"
10     emails "amm@di.fct.unl.pt", "ja@di.fct.unl.pt", "pg@di.fct
          .unl.pt"
11   end
12 end
```

Figura 3.7 – Inserindo cabeçalhos

```
1 \begin{document}
2 \title{Avoiding Bad Smells in Aspect-Oriented Software}
3 \author{
4 Eduardo K. Piveta\(^1\), Marcelo Hecht\(^1\), Marcelo S.
     Pimenta\(^1\), Ana Moreira\(^2\), \\
5 Joao Araujo\(^2\), Pedro Guerreiro\(^2\) \\
6 $^1$Inst. Informatica, Univ. Federal do Rio Grande do Sul -
     Porto Alegre, Brazil\\
7 $^2$Dept. Informatica, FCT, Universidade Nova de Lisboa -
     2829-516 Caparica, Portugal\\
8 \{epiveta, mvhecht, mpimenta\}@inf.ufrgs.br, \{amm, ja, pg\}@di
     .fct.unl.pt\\}
9 \maketitle
```

Figura 3.8 – Código LaTeX gerado do cabeçalho do exemplo 3.7

Para os desenvolvedores criarem mais formatos cria-se uma classe herdada de *MyLatex* e sobrescreve-se o método *setup_ header*, como no exemplo 3.9. Os dados do cabeçalho são obtidos do atributo @*header* e os comandos adicionados ao documento através do método *add_ to_ stack*.

```
1  class MyLatex2 < MyLatex
2    def setup_header format
3      super
4      case format
5        when :UFSM
6            format_ufsm
7        end
8    end
9
10   def format_ufsm
11     @header
12     add_to_stack("formato da ufsm")
13   end
14 end
```

Figura 3.9 – Adicionando padrões de cabeçalhos

3.6 Múltiplos arquivos

Quando se trabalha com documentos muito grandes é comum dividi-los em arquivos menos afim de melhorar a organização do texto. Para isso foi implementado o método *input* que recebe como argumento o arquivo Ruby contendo código DSL pra ser incorporado ao documento. No exemplo 3.10 o método *input* vai procurar por um arquivo Ruby *relatedWork.rb* e, se encontrado, vai executá-lo para gerar o arquivo *relatedWork.tex*.

```
1  input :relatedWork
```

Figura 3.10 – Incorporando arquivos

3.7 Imagens

Para trabalhar com imagens é preciso primeiramente definir o local onde elas estão armazenadas, isso é feito com o método *imagespath* (exemplo 3.11) que pode receber múltiplos caminhos através de um vetor de *Strings*. As imagens são incluídas pelo método *figure* sendo o titulo da imagem único parâmetro obrigatório, os parâmetros opcionais foram realçados em azul no exemplo 3.11 para melhor visualização. Veja a tabela 3.1 para uma descrição detalhada destes parâmetros. O método automaticamente adiciona os pacotes *graphicx* e *float*. Passando um bloco de código, a imagem será tratada como uma figura e então *label* e *caption* poderão ser adicionados, caso contrário ela será tratada apenas como uma imagem.

```
1 imagespath "./imagens/", "./imagens2/"
2 figure "monster", placement: :h, reflected: true, scale: 0.3,
3    trim: "10mm 80mm 20mm 5mm", style: {border: "1pt", padding:
        "5pt"}, clip: true do
4    centering
5    caption "The Spaghetti Monster"
6    label "fig:monster"
7 end
```

Figura 3.11 – Adicionado Imagens

```
1 \graphicspath{{./imagens/}{./imagens2/}}
2 \begin{figure}[h]
3    \reflectbox{\setlength\fboxsep{5pt}
4    \setlength\fboxrule{1pt}
5    \fbox{\includegraphics{monster}}}
6    \centering
7    \caption{Imagem resultante do exemplo \ref{ex22}}
8    \label{fig:monster}
9 \end{figure}
```

Figura 3.12 – Código LaTeX gerado da imagem do exemplo 3.11

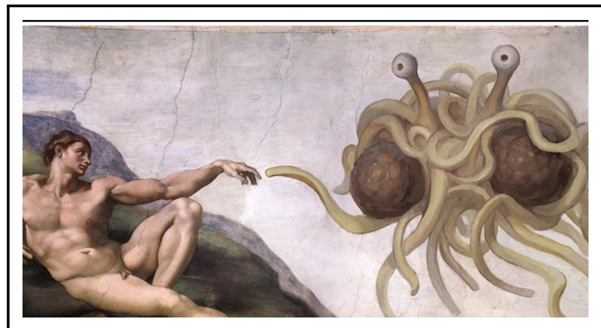

Figura 3.2 – Imagem resultante do exemplo 3.11

3.8 Tabelas

Tabelas em LaTeX podem ser relativamente fáceis de construir ou extremamente complexas dependendo dos recursos empregados. Devido a essa grande flexibilidade optou-se por desenvolver um método que fosse útil para a construção de tabelas simples, porém úteis. Primeiramente é recomendável criar um ambiente para tabelas através do método *start :table*, pois assim é possível adicionar label, caption e usar posicionamento flutuante. Os parâmetros adicionais passados pela *Hash opts* são os mesmos usados pelo parâmetro *placement* em imagens.

Parâmetro	Descrição
width	Especifica o comprimento da imagem.
height	Especifica a altura da imagem.
keepaspectratio	Pode ser definido como *true* ou *false*, Quando *true*, irá escalar a imagem de acordo com *width* e *height*, mas sem distorcer a imagem.
scale	Escala a imagem de acordo com um fator. Ex: 0.5 reduz a imagem pela metade.
angle	Rotaciona a imagem em x graus (anti-horário).
trim	Corta a imagem de acordo com as medidas passadas.
page	Se a imagem é um arquivo .pdf com várias páginas, esse parametro permite usar uma página que não seja a primeira.
placement	Define onde a figura ficará posicionada. *h* fica exatamente onde a imagem é incluída, *t* no topo da página, *b* no fim da página, *p* coloca em uma página especial somente para itens que flutuam, *!* sobrescreve parâmetros internos do LaTeX usados para determinar bom posicionamentos.
reflected	Espelha a imagem caso seja definido como *true*.
style	Define a espessura da borda (*border*) e a distância (*padding*) dela com a imagem.

Tabela 3.1 – Parâmetros para imagens

Para criar a tabela propriamente dita, invoca-se o método *table* passado como argumento uma *String* com especificações para o alinhamentos de cada coluna e o tipo de linha vertical a ser inserido (ver tabela 3.2 para descrição das especificações). Essas especificações seguem a mesma sintaxe das definidas em LaTeX.

Parâmetro	Descrição
r	Coluna Justificada à esquerda.
l	Coluna Justificada à direita.
c	Coluna centralizada.
p{comprimento}	Texto verticalmente alinhado no topo.
m{comprimento}	Texto verticalmente alinhado no meio.
b{comprimento}	Texto verticalmente alinhado na base.
|	Linha vertical simples.
||	Linha vertical dupla.

Tabela 3.2 – Especificações para tabelas

Linhas horizontais são traçadas com o método *hline*, para criar linhas dentro de um intervalor de colunas usa-se o método *cline x..y* passando como parâmetros a coluna inicial (x) e a final (y). Linhas são adicionadas com o comando line, que recebe um vetor como argumento, onde cada elemento do vetor corresponde a uma coluna, caso o número de elementos não corresponda ao de colunas, definido nas especificações da tabela, um erro é gerado. Para criar

linhas que se estendem para mais de uma coluna usa-se um vetor como na linha 5 do exemplo 3.13: no primeiro elemento desse vetor especifica-se o número de linhas pelo qual essa coluna se estenderá, no segundo elementos é adicionado o texto dessa célula (neste caso um comando para gerar um texto em negrito) e o último elemento é a especificação para o alinhamento dessa coluna assim como o tipo de linha vertical. O método *multirow* na linha 7 cria colunas que se abrangem múltiplas linhas. O método recebe como argumento o texto da célula e o número de linhas abrangidas. No bloco de código são passadas as linhas que são abrangidas.

```
1  start :table, opts: [:b] do
2    centering
3    table "|l | c | p{2cm} | b{2cm} | m{2cm} |" do
4      hline
5      line 1, [4,textbf("Multiplas colunas"), "c|"]
6      hline
7      multirow "Multiplas linhas", 2 do
8        line 1, 2, 3, 4
9        line 1, 2, 3, 4
10     end
11     cline 1..5
12     line 6, 7, 8, 9, 10
13     hline
14     line 11, 12, 13, 14, 15
15     hline
16   end
17   caption "Tabela Exemplo 1"
18   label "tab:exemplo1"
19 end
```

Figura 3.13 – Tabela na DSL

```
\begin{table}[b]
   \centering
   \begin{tabular}{|l | c | p{2cm} | b{2cm} | m{2cm} |}
   \hline
   1 &\multicolumn{4}{c|}{\textbf{Multiplas colunas}} \\
   \hline
   \multirow{2}{*}{Multiplas linhas}
   &1 &2 &3 &4 \\
   &1 &2 &3 &4 \\
   \cline{1-5}
   \cline{1-5}6 &7 &8 &9 &10 \\
   \hline
   11 &12 &13 &14 &15 \\
   \hline
   \end{tabular}
   \caption{Tabela Exemplo 1}
   \label{tab:exemplo}
\end{table}
```

Figura 3.14 – Código LaTeX do exemplo 3.13

1	**Multiplas colunas**			
Multiplas linhas	1	2	3	4
	1	2	3	4
6	7	8	9	10
11	12	13	14	15

Tabela 3.3 – Tabela gerada do exemplo 3.13

3.9 Bibliografia com BibTex

A DSL dá suporte a bibliografias através da ferramenta BibTeX que é usada para descrever e processar listas de referências (BIBTEX, 2012). Esse suporte é dado pela classe *BibTex* que implementa a mesma técnica de *Ghost Method* da classe *MyLatex*. Uma bibliografia é criada criando uma instância de BibTex (linha 1 do exemplo 3.15) através do método *create* que recebe um bloco de código que corresponde às entradas de dados e seus respectivos campos. Cada entrada de dados (*string*, *inproceedings*, *article* etc) e campos (*author*, *year*, *title*, *pages* etc) correspondem a métodos serão executados dentro do contexto da instância criada por *BibTex*. Ao final da execução é criado um arquivo *.bib* com o mesmo nome do arquivo Ruby usado. Assim como em arquivos LaTeX, BibTex possui um preâmbulo que fornece informações extras que serão usadas na lista de entradas de dados. No preâmbulo do exemplo 3.15 é usado

o comando *string* que serve para substituir trechos de texto pelo conteúdo de sua variável. Note que o bloco de código foi passado através de chaves, pois o bloco possui apenas uma linha de código. Caso o bloco tivesse mais de uma linha, seria obrigatório o uso de *do...end*. Cada entrada de dado começa com um tipo de entrada (*inproceedings*, na linha 6, por exemplo) seguida de um campo ou uma chave usada para citações (*stein:ecmda-abmbsocss05*, na linha 6). Os campos recebem apenas uma *String* como argumento, alguns desses campos são obrigatórios enquanto outros, opcionais. Para uma lista completa de quais são obrigatórios ou opcionais, veja (HEINZ; HEINZ, 2004) seção 13.2.1.

```
1  BibTex.create do
2    string { icse  "Int'l Conf. Software Engineering"}
3    string { ieee_software "IEEE Software"}
4    string{tapos "Theory and Practice of Object Systems"}
5
6    inproceedings "stein:ecmda-abmbsocss05" do
7      author    "Sergei Kojarski and David H. Lorenz"
8      title     "Pluggable {AOP}: {Designing} aspect mechanisms
          for third-party composition"
9      booktitle "OOPSLA '05: #{tapos}"
10     year      "2005"
11     isbn      "1-59593-031-0"
12     pages     "247--263"
13     location  "San Diego, CA, USA"
14     doi       "http://doi.acm.org/10.1145/1094811.1094831"
15     publisher "ACM Press"
16   end
17 end
18
19 bibliography "aosd-bibliography"
```

Figura 3.15 – Bibliografia com a DSL

```
1  @string{,
2     icse = {Int’l Conf. Software Engineering},
3  }
4  @string{,
5     ieee_software = {IEEE Software},
6  }
7  @string{,
8     tapos = {Theory and Practice of Object Systems},
9  }
10 @inproceedings{stein:ecmda-abmbsocss05,
11    author = {Sergei Kojarski and David H. Lorenz},
12    title = {Pluggable {AOP}: {Designing} aspect mechanisms for
         third-party composition},
13    booktitle = {OOPSLA ’05: Proceedings of the 20th annual ACM
         SIGPLAN conference},
14    year = {2005},
15    isbn = {1-59593-031-0},
16    pages = {247--263},
17    location = {San Diego, CA, USA},
18    doi = {http://doi.acm.org/10.1145/1094811.1094831},
19    publisher = {ACM Press},
20 }
```

Figura 3.16 – Código BibTex gerado do exemplo 3.15

3.10 Criação de comandos

Criar comandos é relativamente simples, o método *newcommand* recebe como primeiro argumento o nome do novo comando que pode ser um *Symbol* ou *String*. Caso precise passar argumentos para o novo comando, é preciso indicar através do segundo argumento o número de argumentos (ver exemplo 3.17, linha 5). Se o comando não recebe argumentos, apenas o nome é necessário. O conteúdo do novo comando é criado no bloco de código onde qualquer comando da DSL pode ser usado.

Como consequência de uma regra da linguagem Ruby não é possível criar comandos que comecem por letra maiúscula e não recebem argumentos, pois neste caso Ruby vai considerar o comando como sendo uma constante.

```
1 newcommand :sw do
2    text "software"
3 end
4 newcommand(:ssw){"Software"}
5 newcommand :tres_argumentos, 3 do
6    text "arg1:#1 arg2:#2 arg3:#3"
7 end
```

Figura 3.17 – Criando comandos com a DSL

```
1 \newcommand{\sw}[0]{
2 software
3 }
4 \newcommand{\ssw}[0]{
5 Software
6 }
7 \newcommand{\tres_argumentos}[3]{
8 arg1:#1 arg2:#2 arg3:#3
9 }
```

Figura 3.18 – Código LaTeX do exemplo 3.17

3.11 Adição de código de linguagens de programação

O pacote *listings* permite adicionar código fonte de qualquer linguagem de programação dentro do documento. Na DSL, essa funcionalidade é implementada dentro do ambiente *lstlisting* (ver exemplo 3.19, linha 4).

Diversos parâmetros podem ser modificados para alterar a forma com o código é apresentado. Isso é feito com o método *lstset*, com mostrado na primeira linha do exemplo 3.19. Esse método pode ser chamado em qualquer lugar do documento, até mesmo no preâmbulo. Para um lista completa dos parâmetros ver (MITTELBACH; GOOSSEN, 2004, p. 24).

```
1  lstset language: "[AspectJ]Java", basicstyle: scriptsize,
2  numbers: :left, firstnumber: 1, aboveskip: "10pt", captionpos:
      :t,breaklines: true
3
4  start :lstlisting, opts: [label: :screenObserver] do
5     text "public aspect ScreenObserver extends ObserverProtocol{
6           declare parents: Screen implements Subject;
7           declare parents: Screen implements Observer;
8           pointcut subjectChange(Subject sub): call(void Screen.
               display(String)) && target(sub);
9           void updateObserver(Subject sub, Observer obs) {
10            ((Screen)obs).display('Updated');
11          }
12        }"
13 end
```

Figura 3.19 – Adicionando código aos textos

```
1  \lstset{emph={ObserverProtocol,declare,parents,subjectChange,
      updateObserver},emphstyle=\color{blue}}
2     \begin{lstlisting}[label = screenObserver]
3     public aspect ScreenObserver extends ObserverProtocol{
4              declare parents: Screen implements Subject;
5              declare parents: Screen implements Observer;
6              pointcut subjectChange(Subject sub): call(void
                  Screen.display(String)) && target(sub);
7              void updateObserver(Subject sub, Observer obs) {
8                ((Screen)obs).display('Updated');
9              }
10           }
11    \ end{lstlisting}
```

Figura 3.20 – Código LaTeX gerado do exemplo 3.19

3.12 Limitações

Ainda que Ruby seja uma linguagem flexível, com recursos poderosos como as técnicas de métodos dinâmicos, grande parte das funcionalidades da linguagem LaTeX não puderam ser contempladas pelo fato de serem muitas e possuírem uma sintaxe própria. Entre os recursos que não foram implementados, destacam-se:

- Desenhos gráficos;
- Glossários;

- Apresentação de Slides;
- Tabelas avançadas;
- Textos técnicos: teoremas, fórmulas matemáticas, algoritmos e pseudocódigo e fórmulas químicas.

4 ESTUDO DE CASO

Como estudo de caso foi escolhido o artigo "*Avoiding Bad Smells in Aspect-Oriented Software*" (PIVETA et al., 2007), pois é um artigo curto (6 páginas) e que ao mesmo tempo utiliza boa parte das funcionalidades da linguagem. O objetivo deste capítulo não é estudar o artigo por completo, pois boa parte de sua estrutura se repete e muitos aspectos da linguagem já foram abordados no capítulo 2, mas fazer observações a determinados pontos da estrutura do artigo.

4.1 Preâmbulo

No código 4.1, linha 1 foi chamado primeiramente o método *create* passando como argumento o formato do arquivo de saída. Por padrão a linguagem gera documentos em *pdf*, mas é possível também gerar arquivos *dvi*. Essa funcionalidade elimina a necessidade do usuário ter de adicionar pacotes específicos para a geração de *pdf* e também de ter que compilar manualmente o projeto.

Além de configurar o tipo de documento e adicionar pacotes, temos a configuração dos exemplos em código Java que serão usados no artigo. Essa configuração é feita na linha 8 pelo método *lstset*.

Nas linhas 10 e 11 são incluídos arquivos que contem novos comandos, que serão utilizados ao longo do artigo.

```
1  MyLatex.create as: :pdf do
2    documentclass :article, opts: ["10pt", :twocolumn]
3    usepackage [:latex8, :times, :balance, :url, :graphix, :
        listings, :color]
4    pagestyle :empty
5    usepackage :graphicx
6    usepackage :inputenc, opts: :latin1
7    usepackage :babel, opts: :english
8    lstset language: "[AspectJ]Java", basicstyle: scriptsize,
        numbers: :left,
9    firstnumber: 1, aboveskip: "10pt", captionpos: :t,breaklines:
        true
10   input :comandos
11   input :relatedWork
```

Figura 4.1 – Preâmbulo do estudo de caso

4.2 Cabeçalho

Para este estudo já foi implementado a formatação exigida pelo evento (SEKE, 2012). O cabeçalho deste estudo de caso possui 2 instituições com seus respectivos atributos: autores, correio eletrônico e local das instituições.

```
12 start :document do
13     header "Avoiding Bad Smells in Aspect-Oriented Software",
          format: :SEKE do
14       institute do
15         authors "Eduardo K. Piveta", "Marcelo Hecht", "Marcelo
              S. Pimenta", "R. Tom Price"
16         location "Inst. Informatica, Univ. Federal do Rio
              Grande do Sul - Porto Alegre, Brazil"
17         emails "epiveta@inf.ufrgs.br", "mvhecht@inf.ufrgs.br",
              "mpimenta@inf.ufrgs.br", "tomprice@terra.com.br"
18       end
19       institute do
20         authors "Ana Moreira", "Joao Araujo", "Pedro Guerreiro"
21         location "Dept. Informatica, FCT, Universidade Nova de
              Lisboa - 2829-516 Caparica, Portugal"
22         emails "amm@di.fct.unl.pt", "ja@di.fct.unl.pt", "pg@di.
              fct.unl.pt"
23       end
24     end
```

Figura 4.2 – Cabeçalho do estudo de caso

A ordem dos atributos de cada instituição não influência na geração do cabeçalho, pois isso será determinado pelo formato do evento, entretanto a ordem fornecida para autores e endereços eletrônicos é refletida no documento gerado

4.3 Textos e comandos

O resumo do artigo é dado pelo ambiente *abstract* (ver Listagem 4.3) compreendido entre as linhas 25 e 31. Usa-se o o comando *text* na criação dos parágrafos do resumo, pois a intenção aqui é apenas inserir um texto sem que exista um comando o precedendo. Quando for preciso adicionar um comando dentro do texto usa-se o símbolo sustenido seguido de colchetes como nos comandos das linhas 27, 28 e 29. Existe uma limitação em relação a comandos que recebem como argumento outros comandos, como exemplo temos o comando text da linha 27 que recebe, além da String, diversos outros comandos. Neste caso não há problema, pois nenhum dos comandos internos (rf, aos, bss etc) recebe como argumento outro comando.

```
25 start :abstract do
26       text "Different kinds of shortcomings can appear during
            software development. Some of them can be detected
            using static analysis tools and be
27       removed using #{rf} techniques. A different strategy is
            to avoid the introduction of those shortcomings before
             they appear in the software.
28       Therefore, based on a collection of #{bss} that can occur
             in #{aos}, we proposed a set of guidelines to #{aosd
            }.
29       The use of these guidelines aims to reduce the occurrence
             of #{bss} in #{aos}.
30       A running example was adopted to exemplify the use of
            these guidelines, showing the benefits of using the
            recommendations hereby described."
31     end
32     Section :Introduction
33     label "sec:introduction"
```

Figura 4.3 – Textos do estudo de caso

Umas das desvantagens de trabalhar com DSL internas são as limitações impostas pela linguagem base. Como exemplo temos o comando *Section* da linha 32 do código 4.3: a linguagem Ruby reconhece como tipo constante qualquer variável que comece por letra maiúscula, entretanto, como o comando *Section* recebe um argumento, a linguagem reconhece como sendo uma chamada de método. Caso ela não recebesse, um erro de compilação seria gerado. Ruby exige que, comandos que são passados como parâmetros e que recebem argumentos, utilizem parênteses para englobar os argumentos, por exemplo o código 4.4, linha 2, o parâmetro *emphstyle* recebe como valor o comando *color* que por sua vez tem como argumento o *Symbol* "*blue*".

```
1 lstset emph: [:ObserverProtocol, :declare, :parents, :
     subjectChange, :updateObserver],
2 emphstyle: color(:blue)
```

Figura 4.4 – comandos com parênteses

4.4 Bibliografia

No caso de uso, a bibliografia utiliza um arquivo que contém padrões de formatação tanto para documentos em geral como para bibliografias, neste caso, o arquivo é *latex8* e é incluído pelo comando *bibliographystyle*. A bibliografia é incluída no documento través do comando *bibliography* (4.5, linha 2), passando como argumento o nome do arquivo .bib que

contem as referências. Nota-se que foi usada uma *String* para indicar o nome do arquivo ao invés de um Symbol, isso foi feito pois Ruby não aceita que *Symbols* possuam hífem.

```
1 bibliographystyle :latex8
2 bibliography "aosd-bibliography"
```

Figura 4.5 – Bibliografia do caso de uso

5 CONCLUSÃO

Neste trabalho foram abordados conceitos para o desenvolvimento de DSLs, a implementação de uma linguagem para a geração de documentos LaTeX e a sua real utilização através da análise de um estudo de caso.

Este trabalho produziu uma linguagem que, ainda que não tenham sido implementadas todas as funcionalidades do LaTeX e que possua limitações intrínsecas à linguagem Ruby, atende satisfatoriamente o desenvolvimento de documentos de razoável complexidade. O maior problema na criação da DSL foi encontrar uma forma de adaptar para uma linguagem de programação a grande flexibilidade da sintaxe do LaTeX. Esse problema foi solucionado utilizando as técnicas de programação dinâmica suportadas pela linguagem Ruby. Algumas dificuldades surgiram durante a implementação, boa parte relacionadas à programação dinâmica, pois este paradigma não é visto durante o curso de Ciência da Computação.

Ainda que a linguagem seja capaz de produzir documentos usados em ambientes acadêmicos, ela carece de praticidade, pois exige do usuário um determinado nível de conhecimento tanto de linguagens programação como do próprio LaTeX. Contudo, o projeto é ponto de partida para diversas abordagens que podem ser utilizadas em projetos futuros: criação de uma interface gráfica que utilize como base a DSL para gerar documentos no estilo WYSIWYG, aumentar o nível de abstração da linguagem para torná-la mais fácil de ser usada por pessoas com pouco conhecimento em programação e externalizar a linguagem através do desenvolvimento de uma DSL externa.

REFERÊNCIAS

BIBTEX. **Your BibTeX resource**. Acessado em 18 de Dezembro de 2012, http://www.bibtex.org.

CONSEL, C. et al. A generative programming approach to developing DSL compilers. In: GPCE'05 PROCEEDINGS OF THE 4TH INTERNATIONAL CONFERENCE ON GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING. **Anais...** Springer-Verlag Berlin: Heidelberg, 2005. p.29–46.

CTAN. **Comprehensive TeX Archive Network (CTAN)**. Acessado em 11 de Dezembro de 2012, http://www.ctan.org/.

DEURSEN, A. V.; KLINT, P.; VISSER, J. Domain-specific languages: an annotated bibliography. **ACM SIGPLAN Notices**, Holanda, Amsterdã, p.26–36, Junho 2000.

LOUKIDES, M. (Ed.). **The Ruby Programming Language**. 1005 Gravenstein Highway North, Sebastopol, CA 95472., EUA: O'Reilly, 2008.

FOWLER, M. **Domain-Specifc Languages**. [S.l.]: Pearson Educarion, 2011.

HEINZ, C.; HEINZ, C. **The Listings Package**. 1.4.ed. [S.l.: s.n.], 2004.

HUDAK, P. Building domain-specific embedded languages. **ACM Computing Surveys (CSUR)**, New York, NY, EUA, v.28, p.196–202, dezembro 1996.

HUDAK, P. Domain-specific languages. **Handbook of Programming Languages**, [S.l.], v.3, p.39–60, dezembro 1997.

HUDAK, P. Modular Domain Specific Languages and Tools. In: ICSR '98 PROCEEDINGS OF THE 5TH INTERNATIONAL CONFERENCE ON SOFTWARE REUSE. **Anais...** IEEE Computer Society Washington: DC: USA, 1998.

KNUTH, D. E. **The TeXbook**. [S.l.]: Addison-Wesley, 1984.

KOSAR, T. et al. A preliminary study on various implementation approaches of domain-specific language. **Information and Software Technology**, Butterworth-Heinemann Newton, MA, EUA, p.390–405, Abril 2008.

LAMPORT, L. **LaTeX - A Document Preparation System**. [S.l.]: Addison-Wesley Publishing Company, 1994.

LATEX. **LaTeX – A document preparation system**. Acessado em 20 de Junho de 2012, http://www.latex-project.org.

LICKERT, K. **Define rake processes to build LaTeX projects**. Acessado em 8 de Janeiro de 2013, http://gems.rubypla.net/rake4latex/.

LOVE, T. **Why LaTeX?** Acessado em 13 de Novembro de 2012, http://www-h.eng.cam.ac.uk/help/tpl/textprocessing/latex_advocacy.html.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. **ACM Computing Surveys (CSUR)**, Nova York, NY, EUA, p.316–344, Dezembro 2005.

MITTELBACH, F.; GOOSSEN, M. **The LATEX Companion**. Segunda.ed. [S.l.]: Addison-Wesley, 2004.

OETIKER, T. et al. **Introdução ao LaTeX2**. [S.l.]: LShort, 2002.

PEREIRA, M. J. V. et al. Program Comprehension for Domain-Specific Languages. **Computer Science and Information Systems**, [S.l.], v.5, Dezembro 2008.

STEINBERG, J. (Ed.). **Metaprogramming Ruby**. Dallas, Texas, EUA: The Pragmatic Bookshelf, 2010.

PIVETA, E. K. et al. Avoiding Bad Smells in Aspect-Oriented Software. In: NINETEENTH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING (SEKE'2007). **Anais...** [S.l.: s.n.], 2007.

RAILS, R. O. **Web development that doesn't hurt**. Acessado em 5 de Novembro de 2012, http://rubyonrails.org.

SEKE. **Software Engineering and Knowledge Engineering**. Acessado em 15 de Dezembro de 2012, http://www.ksi.edu/seke/seke13.html.

STEWART, B. **An Interview with the Creator of Ruby**. Acessado em 7 de Novembro de 2012, http://linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html.

6 ANEXOS

6.1 Lista de Métodos

Método	Descrição
create	Cria o arquivo
text	Adição de código LaTeX *inline*.
bibliography	Adiciona a bibliografia.
input	Inclui arquivos.
usepackage	Adição de pacotes.
lstset	Adicionar linguagens de programação.
figure	Adicionar figuras.
imagespath	Indicar os locais onde estão as imagens.
newcommand	Cria comandos.
header	Adiciona cabeçalhos.
table	Adicionar tabelas.
line	Adiciona linha em tabelas.
cline	Adcionar linhas tracejadas em tabelas.
multirow	Cria colunas que abrangem múltiplas linhas.
command	Recebe todos os comandos que não são contemplados pelos outros métodos.

Tabela 6.1 – Lista de métodos da classe MyLatex

6.2 MyLatex

```
1  require '../rakelatex'
2
3  class MyLatex
4
5    def initialize
6      @packages = {}
7
8      @bibfile = nil
9
10     #relativo a identacao
11     @ident = ""
12     @n = 0
13     @spaces = 3
14
15     #criacao do cabecalho
16     @header = {}
17     @authors_per_line = 3
18
19     @env = {context: nil}
```

```
20
21      #relativo a tabelas
22      @n_columns = 0
23
24      @stack = []
25
26    end
27
28    private
29
30    def add_to_stack cmd
31      @stack.push(cmd)
32    end
33
34    def add_to_packages pack, extra_pack
35      @packages[pack.to_sym] = extra_pack
36    end
37
38    def get_packages
39      return @packages
40    end
41
42    def check_stack params
43      commands = get_cmds_from_string(params)
44      if !get_stack.empty?
45        cp = get_stack.last
46        first_cmd = get_cmds_from_string(cp).first
47        while first_cmd.eql?(commands.pop) && !first_cmd.nil?
48          get_stack.pop
49          cp = get_stack.last
50          first_cmd = get_cmds_from_string(cp).first if !cp.nil?
51        end
52      end
53    end
54
55    def get_cmds_from_string cmd
56      cmd.scan(/\\\w+/)
57    end
58
59    def get_ident n
60      ident =""
61      n.times{ident << " "}
62      ident
63    end
64
65    def get_options args
66      options ={}
67      args.delete_if do |item|
```

```
68        if item.is_a? Hash
69          options = item
70          true
71        else
72          false
73        end
74      end
75      opts = ""
76      if options.has_key?(:opts)
77        opts << "["
78        options[:opts].each do |op|
79          op.each{|k,v| opts << "#{k} = #{v},"} if op.is_a? Hash
80          opts << "#{op}," unless op.is_a? Hash
81        end
82        opts = opts[0...-1]
83        opts << "]"
84      end
85      opts
86    end
87
88    def clear_command method
89      old_cmd = method
90      method = method.to_s.delete("_")
91      [old_cmd, method]
92    end
93
94    public
95
96    def get_stack
97      return @stack
98    end
99
100   def get_packages
101     return @packages
102   end
103
104   def get_bib_file
105     return @bibfile
106   end
107
108   def main_document?
109     self.get_stack.first.include?("documentclass")
110   end
111
112   def dsl_to_latex(file_name)
113     out, idx = "", 0
114     latex, packages = Array.new(self.get_stack), self.
           get_packages
```

```
115       if main_document?
116         out = latex.delete_at(0)+"\n"
117       end
118       packages.each do |pk, params|
119         out << "\\usepackage[#{params.join(",")}]{#{pk}}\n" if
              params.is_a? Array
120         out << "\\usepackage[#{params}]{#{pk}}\n" unless params.
              is_a? Array
121       end
122       latex.each do |command|
123         out << command << "\n"
124       end
125       outFile = File.new(file_name + ".tex", "w")
126       outFile.write(out)
127       outFile.close
128     end
129
130     def get_file_name
131       path = caller(2).first
132       name = File.basename(path).partition(".").first
133       return name
134     end
135
136     def self.create(output = {as: :pdf}, &block)
137       textLatex =  self.new
138       name = textLatex.get_file_name
139       textLatex.instance_eval(&block)
140       textLatex.dsl_to_latex(name)
141       RakeLatex.new(main_file: name, bibfile: textLatex.
            get_bib_file, type: output[:as]) if textLatex.
            main_document?
142       return textLatex
143     end
144
145     def command(method, *args, &block)
146       options = get_options args
147       @ident = get_ident @n
148       if block_given?
149         add_to_stack(@ident + "\\begin{#{args.first}}#{options}")
150         n = @n
151         @n += @spaces
152         yield
153         @n = n
154         add_to_stack(get_ident(n) + "\\end{#{args.first}}")
155       else
156         old_command, method = clear_command method#tira os "_"
157         command = "\\#{method}"
158         params = options
```

```
159         args.each do |arg|
160            params << "{#{arg.join(",")}}" if arg.is_a? Array
161            params << "{#{arg}}" unless arg.is_a? Array
162         end
163         check_stack(params)
164         line = command + params
165         add_to_stack(@ident + command + params)
166         line
167       end
168     end
169
170     alias method_missing command
171
172     def text txt
173       check_stack(txt)
174       add_to_stack(@ident + txt)
175     end
176
177     def bibliography bib
178       @bibfile = bib
179       cmd = @ident + "\\bibliography{#{bib}}"
180       add_to_stack(cmd)
181     end
182
183     def input(arq, options = {})
184       file = arq.to_s + ".rb"
185       if File.exists?(file)
186         %x(ruby #{file})
187         out = "\\input #{arq.to_s + ".tex"}"
188         add_to_stack(out)
189       else
190         raise "There is no such file called \"#{file}\""
191       end
192       out
193     end
194
195     def usepackage(*args)
196       t_args = Array.new(args).flatten
197       opts = {opts: []}
198       t_args.each_with_index do |arg, idx|
199         if arg.is_a?(Hash)
200           opts = arg
201           t_args.delete_at(idx)
202         end
203       end
204       opts[:opts] = [opts[:opts]] if !opts[:opts].is_a?(Array)
205       t_args.each do |pack|
206         if get_packages.include?(pack.to_sym)
```

```
207         extra_pk = (opts[:opts] + get_packages[pack.to_sym]).
               uniq
208         add_to_packages(pack, extra_pk)
209       else
210          add_to_packages(pack, opts[:opts])
211       end
212     end
213   end
214
215   ################### Adicao de codigo
         #####################
216
217   def lstset(params ={})
218     cmd, param = "\\lstset{", ""
219     params.each do |key, value|
220       param << key.to_s << "="
221       param << "{#{value.join(",")}}" if value.is_a? Array
222       param << value.to_s unless value.is_a? Array
223       param << ","
224     end
225     param = param[0...-1]#remove ultima virgula
226     param << "}\n"
227     check_stack(param)
228     add_to_stack(cmd + param)
229   end
230
231   #################### adicao de imagens
         ######################
232
233   def figure(img_name, opts = {}, &block)
234     usepackage [:graphicx, :float]
235     raise "You must give an image name" unless img_name.is_a?
           String
236     placement = opts[:placement]; opts.delete(:placement)
237     reflected = opts[:reflected]; opts.delete(:reflected)
238     styles = opts[:style]; opts.delete(:style)
239     params = "[#{opts.collect{|key, value| "#{key}=#{value}"}.
           join(",")}]"
240     check_stack(params)
241     out_figure = "\\includegraphics#{params}{#{img_name}}"
242     if styles
243       out_styles =""
244       out_styles << "\\setlength\\fboxsep{#{styles[:padding]}}\
             n" if styles.has_key?(:padding)
245       out_styles << "\\setlength\\fboxrule{#{styles[:border]}}\
             n" if styles.has_key?(:border)
246       out_styles << "\\fbox{#{out_figure}}"
247       out_figure = out_styles
```

```
248      end
249      if reflected
250        out_figure = "\\reflectbox{#{out_figure}}\n"
251      end
252      if block_given?
253        add_to_stack "\\begin{figure}[#{placement}]"
254        add_to_stack out_figure
255        yield
256        add_to_stack "\\end{figure}\n"
257      else
258        add_to_stack out_figure
259      end
260    end
261
262    def imagespath *paths
263      ps = ""
264      paths.each do |p|
265        ps << "{#{p}}"
266      end
267      add_to_stack "\\graphicspath{#{ps}}"
268    end
269
270    ####################Criacao de Comandos
          ################################
271
272    def newcommand(name, num_args = 0, &block)
273      add_to_stack(@ident + "\\#{__method__}{\\#{name}}[#{
            num_args}]{")
274      yield
275      add_to_stack("}")
276    end
277
278    ######################### Criacao do Cabecalho
          #######################
279
280    def header(title, config = {}, &block)
281      @header = {title: title}
282      if block_given?
283        yield
284      else
285        raise "You must set the headers's body"
286      end
287      setup_header config[:format]
288    end
289
290    def institute &block
291      @header[:institutes] =[] unless @header.has_key?(:
            institutes)
```

```
292      @header[:institutes].push ({emails: [], authors: [],
            location: nil})
293      yield
294    end
295
296    def location place
297      @header[:institutes].last[:location] = place
298    end
299
300    def emails *adress
301      @header[:institutes].last[:emails] = adress
302    end
303
304    def authors *aut
305      @header[:institutes].last[:authors] = aut
306    end
307
308    def setup_header format
309      case format
310        when :SEKE
311          format_seke
312      end
313    end
314
315    def format_seke
316      @out_header = ""
317      @out_header << @ident << "\\title{#{@header[:title]}}\n"
318      @out_header << @ident << "\\author{\n"
319      temp_authors, temp_inst, temp_address = [], [], []
320      temp_emails ={}
321      @header[:institutes].each_with_index do |inst, idx|
322        temp_authors += inst[:authors].collect{|aut| aut + "
              \\(^#{idx+1}\\), "}
323        temp_inst.push("$^#{idx+1}$" + inst[:location] + "\\\\n"
              )
324        inst[:emails].each do |email|
325          e_data = email.partition("@") #separa o login do
                dominio
326          temp_emails[e_data.last.to_sym] =[] unless temp_emails.
                has_key?(e_data.last.to_sym)
327          temp_emails[e_data.last.to_sym].push(e_data.first)
328        end
329      end
330      temp_authors[temp_authors.length-1] = temp_authors.last.
            delete(",")
331      n_lines = temp_authors.size/(@authors_per_line + 1)
332      n_lines += 1 if (temp_authors.size%(@authors_per_line + 1)
            > 0)
```

```
333     n_lines = 1 unless n_lines > 0
334     init, len = 0, @authors_per_line
335     n_lines.times do
336       @out_header << @ident << temp_authors[init..len].join
337       @out_header << "\\\\"
338       init += @authors_per_line + 1
339       len  += @authors_per_line + 1
340     end
341     @out_header << "\n" + temp_inst.join
342     temp_emails.each do |domain, logins|
343       temp_logins = logins.join(", ")
344       if logins.size > 1
345          temp_address.push("\\{#{temp_logins}\\}@#{domain}")
346       else
347         temp_address.push("#{temp_logins}@#{domain}")
348       end
349     end
350     @out_header << @ident << temp_address.join(", ") + "\\\\}\n
          "
351     @out_header << @ident << "\\maketitle\n"
352     add_to_stack(@out_header)
353   end
354 ################################ Tabelas
        ##########################
355
356 def table *params
357   usepackage :xcolor, opts: :table
358   add_to_stack(@ident + "\\begin{tabular}{#{params.first}}")
359   @n_columns = get_columns(params.first)
360   if (params.last.is_a? Hash)
361     c = params.last[:columns]
362     l = params.last[:lines]
363     l.times do |nl|
364       temp_out = ""
365       c.times do |nc|
366         temp_out << " #{nc} &"
367       end
368       temp_out = temp_out[0...-1]#remove o ultimo &
369       add_to_stack(@ident + temp_out << "\\\\")
370     end
371   end
372   yield if block_given?
373   add_to_stack(@ident + "\\end{tabular}")
374 end
375
376 def get_columns(exp)
377   exp.gsub(/@{\W*}|{\w*}|\s+/,'').delete("|").size #remove
          coisas do tipo @{...} m{...}, | e espacos em branco
```

```
378   end
379
380   def line *args
381     mrow, temp_l, total_columns = "", "", 0
382     if @env[:context] == :multirow
383       temp_l << "&"
384       total_columns += 1
385     end
386     args.each do |item|
387       if item.is_a? Array
388         nc, text, style = item.first, item[1], ""
389         total_columns += nc
390         style = item[2] if item.length == 3
391         check_stack(text)
392         temp_l << "\\multicolumn{#{nc}}{#{style}}{#{text}} &"
393       else
394         temp_l << item.to_s << " &"
395         check_stack(item.to_s)
396         total_columns += 1
397       end
398     end
399     if (total_columns != @n_columns)
400       raise "Wrong number of columns, #{total_columns} instead
             of #{@n_columns}"
401     end
402     temp_l = temp_l[0...-1]
403     temp_l << "\\\\\n"
404     add_to_stack(@ident + temp_l)
405   end
406
407   def cline range
408     b, e = range.begin, range.last
409     if ((1..@n_columns).include?(b) && (1..@n_columns).include?(
           e))
410       add_to_stack(@ident << "\\#{__method__}{#{b}-#{e}}")
411     else
412       raise "#{b} or #{e} are out of range"
413     end
414   end
415
416   def multirow title, num, width = "*", &block
417     usepackage :multirow
418     @env[:context] = __method__
419     if block_given?
420       check_stack(title)
421       add_to_stack(@ident + "\\multirow{#{num}}{#{width}}{#{
             title}}")
422       yield
```

```
423       else
424         raise "#{__method__} needs lines"
425       end
426       @env[:context] = nil
427     end
428
429
430     ##
          ##################################################################

431
432     def latex
433       add_to_stack(@ident + "\\LaTeX")
434     end
435
436     def tex
437       add_to_stack(@ident + "\\TeX")
438     end
439
440     def latexe
441       add_to_stack(@ident + "\\LaTeXe")
442     end
443   end
```

Figura 6.1 – MyLatex

6.3 Bibtex

```
 1  class BibTex
 2    def initialize
 3      @out = ""
 4      @required_types = {article:       [:author, :title, :journal
          , :year],
 5                         book:          [:author, :title, :
                              publisher, :year],
 6                         booklet:       [:title],
 7                         conference:    [:author, :title, :
                              booktitle,:publisher, :year],
 8                         inbook:        [:author, :title, :
                              publisher, :year, :editor],
 9                         incollection:  [:author, :title, :
                              publisher, :year, :booktitle],
10                         inproceedings: [:author, :title, :
                              booktitle, :year],
11                         manual:        [:title],
12                         mastersthesis: [:author, :title, :school,
                                :year],
13                         misc:          [],
14                         phdthesis:     [:author, :title, :school,
```

```
                         :year],
15                   proceedings:   [:title, :year],
16                   techreport:    [:author, :title, :
                        institution, :year],
17                   unpublished:   [:author, :title, :note],
18                   string:        []
19                 }
20  @temp_req =[]
21  end
22
23  def self.create(*args, &block)
24    textBibi =  BibTex.new
25    name = textBibi.get_file_name
26    bib = textBibi.instance_eval(&block)
27    outFile = File.new(name + ".bib", "w")
28    outFile.write(bib)
29    outFile.close
30    return textBibi
31  end
32
33  def get_file_name
34    path = caller(2).first
35    name = File.basename(path).partition(".").first
36    return name
37  end
38
39  def method_missing(type, *args, &block)
40    if block_given?
41      if type_exists?(type)
42        @out << "@#{type}{#{args.first},\n"
43        yield
44        missing = (@required_types[type] - @temp_req)
45        raise "For type '#{type}' these tags are mandatory: #{
             missing.join(" ,")}" if !missing.empty?
46        @temp_req.clear
47        @out << "}\n"
48      else
49        raise "Type #{type} does not exist"
50      end
51    else
52      @temp_req.push type
53      @out << "#{type} = {#{args.first}},\n"
54    end
55  end
56
57  def type_exists?(type)
58    @required_types.has_key?(type) ? true : false
59  end
```

```
60 end
```

Figura 6.2 – Bibtex

6.4 RakeLatex

```
1  require 'rake4latex'
2  include Rake::DSL
3
4  class RakeLatex
5    def initialize(settings)
6      task :basefile => "#{settings[:main_file]}.tex"
7
8      file "#{settings[:main_file]}.#{settings[:type]}" => "#{
          settings[:main_file]}.tex"
9      file "#{settings[:main_file]}.#{settings[:type]}" => "#{
          settings[:bibfile]}.bib"
10     file "#{settings[:main_file]}.bbl" => "#{settings[:bibfile
          ]}.bib"
11
12     task :touch => "#{settings[:main_file]}.tex"
13
14     task :default => :touch
15     task :default => "#{settings[:main_file]}.#{settings[:type
          ]}"
16     app = Rake.application
17     app[:default].invoke
18   end
19 end
```

Figura 6.3 – RakeLatex