**Bacharelado em Ciência da Computação**

Escola de Engenharia de Piracicaba

Fundação Municipal de Ensino de Piracicaba

Software de conhecimento computacional para simplificação e resolução de expressões algébricas

*Antonio Raphael de Arruda Basso*



***Ano: 2017***

**Bacharelado em Ciência da Computação**

Escola de Engenharia de Piracicaba

Fundação Municipal de Ensino de Piracicaba

Software de conhecimento computacional para simplificação e resolução de expressões algébricas

*Monografia de Conclusão de Curso de Graduação*

*apresentada à Escola de Engenharia de Piracicaba*

*como um requisito para a conclusão do Curso de   
Bacharelado em Ciência da Computação*

Discente: Antonio Raphael de Arruda Basso

Docente Orientador: Odahyr Cavalini Junior

Folha de Aprovação

**Data de Defesa:**

**Banca Examinadora**

|  |  |
| --- | --- |
| **Prof. ...............................**  **Assinatura:** |  |
| **Prof. .................................**  **Assinatura:** |  |
| **Prof. ................................**  **Assinatura:** |  |

Agradecimentos

Primeiramente à Deus

Sadasds asd asd asdasd asd ad asda sdasd asd asd asd asd sadasdas dasd asd asdasdasdasda sa dasd asd asda sda sdasd asda sda sda dasd ad asd asd

Asda sdas dasdasd asd asd asd asdasd asda sdasd asd asd asd a

resumo

Desenvolver um software de conhecimento computacional matemático, cujo objetivo principal é analisar, interpretar, simplificar e resolver de forma iterativa expressões algébricas com base em heurísticas simples, utilizando para isso conceitos das disciplinas de exatas do curso de Ciência da Computação. É software de código aberto, e poderá ser utilizado com o intuito de ajudar os discentes nos estudos dos mais variados temas relacionados às disciplinas de exatas, uma vez que a maioria das ferramentas disponíveis são proprietárias. O software pretende concorrer com softwares proprietários, tampouco, implementar um conjunto rico de recursos para simplificação e resolução de expressões algébricas.

Abstract

Resumo em inglês

lista de ilustrações

Figura 1 – csadasdasdasdasd

Figura 2 – sadasd asdasd asd asd

Figura 3 – sasfdasdas dasdasda dasd ad as

Figura 4 – asdasdsada sdasd asd adasdas d

lista de Tabelas

Tabela 1 – csadasdasdasdasd

Tabela 2 – sadasd asdasd asd asd

Tabela 3 – sasfdasdas dasdasda dasd ad as

Tabela 4 – asdasdsada sdasd asd adasdas d

lista de ABreviaturas e siglas

TDD – *Test Driven Development*

XP – *Extreme Programming*

DDD – *Domain Driven Design*

GOF – *Gang of Four*

UML – *Unified Modeling Language*

HTML – *HyperText Markup Language*

CSS – *Cascading Style Sheet*

MVC – *Model View Controller*

JPA – *Java Persistence API*

IDE – *Integrated Development Environment*

SGBD – Sistema Gerenciador de Banco de Dados

HTTP – *Hypertext Transfer Protocol*

SO – Sistema Operacional

AJAX – *Asynchronous Javascript And XML*

CRUD – *Create, read, update, delete*

XML – *eXtensible Markup Language*

AOP – *Aspect Oriented Programming*

CST – *Concrete Syntax Tree*

AST – *Abstract Syntax Tree*

Sumário

RESUMO iii

ABSTRACT iv

lISTA DE Ilustrações v

lISTA DE Tabelas vi

lISTA DE abreviaturas e siglas vii

1. Introdução
   1. [Contextualização](#_toc360) 01
   2. [Objetivo](#_toc365) 02
   3. [Motivação](#_toc367) 03
   4. [Materiais e Métodos](#_toc370) 04
   5. [Cronograma Realizado](#_toc372) 05
2. Revisão Bibliográfica
   1. Considerações Iniciais 06
   2. Tópico AA

2.2.1. Tópico AAaa

* 1. Tópico BB

2.3.1. Tópico BBbb

* 1. Trabalhos Relacionados
     1. Tópico CCcc
  2. Trabalhos Relacionados
  3. Considerações Finais

1. PROJETO
   1. Considerações Iniciais
   2. Especificação de Usuários
   3. Especificação de Requisitos
   4. Artefatos de Análise
   5. Artefatos de Projeto
   6. Considerações Parciais
2. Desenvolvimento
   1. Considerações Iniciais
   2. Aaaaa
   3. Bbbbb
   4. Verificação e Validação
   5. Considerações Parciais
3. Conclusão
   1. [Discussão sobre os Resultados](#_toc632)
   2. [Desafios Encontrados](#_toc636)
   3. [Trabalhos Futuros](#_toc648)

REFERÊNCIAS BibliogrÁFICAS

ApêndiceS

A – Nome do Apêndice

B – Nome do Apêndice

ANEXOS

A – Nome do Anexo

B – Nome do Anexo

**Parecer do Orientador**

**Capítulo 1**

**Introdução**

* 1. **Contextualização**

Há uma série de ferramentas que tem como propósito tornar o conhecimento sistemático computável. Alguns *softwares* são mais voltados para conhecimento algébrico (como o *Derive* ou *TI-Nspire* da *Texas Instruments)*, outros vão além, tentando tornar todo o conhecimento sistemático computáveis acessível fornecendo respostas para consultas factuais (como o *Wolframalpha*).

Por conta do caráter complexo dessas ferramentas, muitas são pagas e não estão disponíveis a todo tipo de público (principalmente para os discentes de cursos de graduação, uma vez que licenciar ferramentas do gênero tem elevado custo). Outro problema dessas ferramentas fica por conta do suporte, seja por questões de internacionalização ou pela descontinuidade da ferramenta. Também, os algoritmos (tampouco o código-fonte) por trás dessas ferramentas não estão acessíveis ao público.

Este projeto tem como propósito desenvolver um software de código aberto que possa ser utilizado como mecanismo de conhecimento computacional matemático, cujo objetivo principal é analisar, interpretar, simplificar e resolver de forma iterativa expressões algébricas.

* 1. **Objetivo**

Desenvolver um software de conhecimento computacional matemático, cujo objetivo principal é analisar, interpretar, simplificar e resolver de forma iterativa expressões algébricas com base em heurísticas simples, utilizando para isso conceitos das disciplinas de exatas do curso de Ciência da Computação.

* 1. **Motivação**

Desenvolver um software de código aberto que possa ser utilizado como mecanismo de conhecimento computacional matemático com o intuito de ajudar os discentes nos estudos dos mais variados temas relacionados às disciplinas de exatas, uma vez que a maioria das ferramentas disponíveis são proprietárias.

* 1. **Materiais e Métodos**
  2. **Cronograma Realizado**

**Capítulo 2**

**Revisão Bibliográfica**

**2.1. Considerações Iniciais**

Neste capítulo são apresentados os recursos utilizados para realizar o desenvolvimento da aplicação proposta, que inclui especificação de usuários e requisitos, artefatos de análise e projeto e custo do projeto.

**2.2. Álgebra**

aaaa.

**2.2.1. Algoritmo Euclidiano**

aaaa.

**2.3. Compiladores**

Segundo Louden (2004), compiladores são *softwares* que traduzem o código fonte de uma linguagem para outra (recebendo como entrada o código fonte de um programa transformando-o em código objeto). Normalmente, o código fonte é escrito em uma linguagem de programação de alto nível, com grande capacidade de abstração, e o código objeto é escrito em uma linguagem de baixo nível, como uma sequência de instruções a ser executada pelo processador (pode ocorrer do processo de tradução ser feito de uma linguagem de alto nível para outra).

Figura 1: Processo de tradução de um compilador



Fonte: Próprio autor

O processo de tradução de uma linguagem por um compilador

**2.3.1. Análise Léxica**

Aaaa.

**2.3.2. Análise Sintática**

Aaaa.

**2.3.3. Análise Semântica**

Aaaa.

**2.3.4. ANTLR**

ANTLR (*ANother Tool for Language Recognition*) é um poderoso gerador de analisadores para leitura, processamento, execução, ou tradução estruturada de textos ou arquivos binários. É largamente usada na construção de linguagens, ferramentas e *frameworks*. A partir de uma gramática, ANTLR gera analisadores que podem construir e percorrer árvores de sintáticas.

**2.4. Design Patterns**

*Design patterns* (Padrões de Projeto) de *software* descrevem um problema e como resolvê-lo, de tal forma que se possa utilizar esse padrão indefinidas vezes, nunca fazendo da mesma maneira. Segundo Gamma (2000), os padrões de projeto têm quatro elementos essenciais:

1. Nome:
2. Problema:
3. Solução:
4. Consequências:

**2.4.1. Inversion of Control**

*Inversion of control* (Inversão de Controle) é o nome do padrão de projeto de *software* que consiste em mudar no fluxo de execução de um programa, ou seja, ao invés do programador determinar quando um procedimento será executado, ele apenas determina qual é esse procedimento. Tem como objetivo reduzir o acoplamento, aumentar a coesão, facilitar o reuso e os testes no projeto de software.

**2.4.1. Dependency Injection**

*Dependency injection* (Injeção de Dependência) é uma das formas de se aplicar a inversão de controle.

A técnica consiste em passar a dependência (o serviço) para o dependente (o cliente). Isso é a chamada injeção. O importante é entender que o serviço é injetado no cliente ao invés do próprio cliente procurar e construir o serviço que irá utilizar, permitindo que estados e comportamentos sejam determinados através de passagem de parâmetros. Em programação orientada a objetos, essa passagem de parâmetros pode ser pelo construtor da classe, método ou atributo, onde na maioria dos casos, o tipo do parâmetro é uma interface ou classe abstrata.

**2.4.1. Visitor**

Aaaa.

**2.5. Domain Driven Design**

aaaa.

**2.6. Test Driven Development**

*Test Driven Development* (Desenvolvimento Dirigido por Testes), é uma técnica de desenvolvimento de software que tem como princípio a criação de testes automatizados antes de qualquer código de produção, com o objetivo de confirmar o funcionamento de uma implementação feita pelo programador. Essa técnica foi desenvolvida por Kent Beck (BECK, 1999), e é um dos pilares do *Extreme Programming* (Programação Extrema). Segundo o autor, há uma série de benefícios nesse estilo de programação que acabam melhorando o produto final e principalmente, a filosofia de trabalho do programador, como:

* Código limpo que funciona, que acaba ajudando na simplicidade, clareza, eliminação de duplicação em várias partes da aplicação, *design* coeso onde cada parte do código tem apenas uma responsabilidade, baixo acoplamento das partes do código reduzindo a alta dependência entre os módulos;
* O número de surpresas desagradáveis e a densidade de defeitos pode ser suficientemente reduzida, contribuindo para ter *software* pronto e com novas funcionalidades a cada dia.
* Os testes anteriormente criados servem como atestados de que o código funciona, e também como documentação de como esse código deve ser utilizado.
* É uma forma de administrar o medo do programador pois tem-se maior segurança ao realizar alterações num código que tem um teste automatizado que comprova seu correto funcionamento.

De acordo com os autores Beck (2010), Freeman e Pryce (2012), entre outros, a técnica utiliza um ciclo de desenvolvimento que consiste em identificar uma funcionalidade a se desenvolver, acrescentando uma tarefa à lista de tarefas, e então:

1. Criar um teste automatizado que verifique uma pequena porção dessa funcionalidade. Uma vez definido o teste, implementa-se o código de produção que atenda à necessidade (o código deve ser simples o bastante para compilar). Roda-se o teste, e o mesmo deve falhar (alguns ambientes de desenvolvimento integrado que possuem integração com ferramentas de testes automatizados emitem uma barra vermelha como indicativo de falha).
2. Identificado que o teste falhou, deve-se partir para a implementação mais simples possível do código de produção que faça esse teste passar (obtendo assim, uma barra verde).
3. Por fim, deve-se refatorar o código, eliminando redundâncias, porém, sem alterar seu comportamento (há diversas técnicas de refatoração como: eliminar duplicação, deixar clara a intenção com nomes mais sugestivos para identificadores, extrair classes, interfaces, métodos, etc.). Feita a refatoração, roda-se o teste novamente para garantir que que o código ainda funciona, caso contrário, recomeçamos o processo até fazer o teste passar novamente.

Figura 1: Clico de desenvolvimento do TDD (Vermelho, Verde, Refatorar)



Fonte: Próprio autor

Há diversos tipos de testes automatizados passíveis de implementação com o TDD, como o Unit Test (teste unitário), que tem como objetivo testar pequenas porções de código independentes, o *Integration test* (teste de integração), que compreende vários módulos que possuem algum tipo de dependência, o *System Test* (teste de sistema) e o *Acceptance Test* (teste de aceitação), que tem como função testar as camadas mais externas ao *software*, entre outros. A Figura 2 e a Figura 3 demonstram um exemplo de *Unit Test* (teste unitário) utilizando a linguagem de programação Java em conjunto com as bibliotecas JUnit (facilita a criação de código para automatização de testes, e possui integração com vários ambientes de desenvolvimento integrado) e Hamcrest (melhora a legibilidade dos testes):

Figura 2: Exemplo de classe “Usuario”

**public class** Usuario{

**private** String nome;

**private** String login;

**private** String email;

**public** Usuario(String nome, String login, String email){

this.nome = nome;

this.login = login;

this.email = email;

}

**public** String descricao(){

**return** nome + " <" + this.email + ">";

}

}

Fonte: Próprio autor

Figura 3: Teste unitário da classe “Usuario”

**public class** UsuarioTest{

// O nome do método deve descrever o que o teste faz

*@Test*

**public void** descricaoDeveTerNomeEmail(){

Usuario u = **new** Usuario("Raphael Basso", "arabasso",

"arabasso@yahoo.com.br");

String descrição = "Raphael Basso <arabasso@yahoo.com.br>";

// Aqui é feita a validação

assertThat(u.descricao(), is(equalTo(descricao)));

}

}

Fonte: Próprio autor

**2.7. Agile Development**

Desenvolvimento ágil foi uma iniciativa de um grupo de especialistas de engenharia de software em 2001 como uma reação contra métodos que possuíam regulamentação excessiva (como o modelo em cascata), que tem como objetivo minimizar os riscos utilizando períodos curtos de desenvolvimento, permitindo respostas mais rápidas a mudanças no software. Cada iteração é um projeto à parte e pode durar de uma semana a um mês.

**2.7.1. Valores**

Segundo Beck (2000), o desenvolvimento ágil de software possui os seguintes valores:

* Os indivíduos e as interações são mais importantes do que os processos e as ferramentas;
* O software funcionando é mais importante do que uma documentação completa;
* A colaboração com e dos clientes acima de apenas negociações de contratos e;
* Respostas a mudanças acima de seguir um plano.

**2.7.2. Princípios**

Segundo Beck (2000), os desenvolvimento ágil de software possui os seguintes princípios:

* Garantir a satisfação do consumidor entregando rapidamente e continuamente software funcionais;
* Até mesmo mudanças tardias de escopo no projecto são bem-vindas para garantir a vantagem competitiva do cliente;
* Software funcionais são entregues frequentemente (semanas, ao invés de meses);
* Cooperação diária entre pessoas que entendem do 'negócio' e desenvolvedores;
* Projetos surgem através de indivíduos motivados, entre os quais existe relação de confiança.
* A maneira mais eficiente e efetiva de transmitir informações é conversas cara a cara;
* Software funcionais são a principal medida de progresso do projeto;
* Processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes para manter um ritmo constante indefinidamente.
* Design do software deve prezar pela excelência técnica;
* Simplicidade é essencial;
* As melhores arquiteturas, requisitos e projetos emergem de equipes auto organizadas;
* Em intervalos regulares, a equipe reflete sobre como para tornar-se mais eficaz, então sintoniza e ajusta seu comportamento apropriadamente.

**2.7.3. Extreme Programming**

Aaaa.

**Capítulo 3**

**Projeto**

**3.1. Considerações Iniciais**

Neste capítulo são apresentados os recursos utilizados para realizar o desenvolvimento da aplicação proposta, que inclui especificação de usuários e requisitos, artefatos de análise e projeto e custo do projeto.

**3.2. Especificação de Usuários**

O público alvo do Sistema de Gerenciamento de Câmaras Frigoríficas são as Empresas do setor alimentício que visam um melhor monitoramento e administração de seus produtos e que necessitam de uma temperatura bem abaixo das temperaturas consideradas ambientes, ou seja, uma temperatura ambiente em média é de temperaturas abaixo de 5ºC.

O sistema auxiliará no gerenciamento das câmaras, pois todo o acompanhamento é feito em tempo real onde o usuário acessará uma página web para monitorar as temperaturas.

Os utilizadores deste projeto serão os moradores de uma residência. O papel do Administrador pode ser representado pelo responsável pela residência como, por exemplo, o pai ou a mãe. O papel do Usuário será representado pelos outros moradores.

**3.3. Especificação de Requisitos**

Aaaa.

**3.3.1. Diagramas de caso de uso**

Aaaa.

**3.3.2. Diagramas de atividade**

Aaaa.

**3.4. Artefatos de Análise**

Aaaa.

**3.4.1. Diagramas de classe**

Aaaa.

**3.4.2. Modelo DER**

Aaaa.

**3.5. Artefatos de Projeto**

Aaaa.

**3.5.1. Dicionário de tabelas**

Aaaa.

**3.5.2. Diagramas de sequência**

Aaaa.

**3.6. Considerações Parciais**

Neste capítulo são apresentados os recursos utilizados para realizar o desenvolvimento da aplicação proposta, que inclui especificação de usuários e requisitos, artefatos de análise e projeto e custo do projeto.

**Referências  
Bibliográficas**

ANTLR. ANTLR 4 Documentation. Disponível em: <https://github.com/antlr/antlr4/blob/4.6/doc/index.md>. Acesso em: 26 mar. 2017.

APACHE GROOVY. Integrating Groovy into applications. Disponível em: <http://groovy-lang.org/integrating.html>. Acesso em: 20 mar. 2017.

SPRING. Spring Boot. Disponível em: <https://projects.spring.io/spring-boot/>. Acesso em: 21 mar. 2017.

SPRING. Spring Initializr. Disponível em: <http://start.spring.io/>. Acesso em: 21 mar. 2017.

LOUDEN, Kenneth C. **Compiladores: princípios e práticas**. São Paulo: Pioneira Thomson Learning, 2004.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Padrões de projetos: soluções reutilizáveis de software orientado a objetos**. Porto Alegre: Bookman, 2000.

FREEMAN, Eric; FREEMAN, Elisabeth. **Use a cabeça: padrões de projetos. Rio de Janeiro: Alta Books**, 2009.

EVANS, Eric. **Domain-driven design: atacando as complexidades no coração do software**. Rio de Janeiro: Alta Books, 2010.

BECK, Kent. **TDD: desenvolvimento guiado por testes**. Porto Alegre: Bookman, 2010.

FREEMAN, Steve; PRYCE, Nat. **Desenvolvimento de software orientado a objetos, guiado por testes**. Rio de Janeiro: Alta Books, 2012.

ANTLR. ANTLR 4 Documentation. Disponível em: <https://github.com/antlr/antlr4/blob/4.6/doc/index.md>. Acesso em: 26 mar. 2017.

BECK, Kent. et al. Manifesto para o desenvolvimento ágil de software. Disponível em: < http://www.manifestoagil.com.br/index.html>. Acesso em: 07 mai. 2017.

MARTIN, Robert C.; MARTIN, Micah. **Princípios, padrões e práticas ágeis em C#**. Porto Alegre: Bookman, 2011.

**Apêndices**

**Anexos**

**Parecer do Orientador**

Texto asdasdasdasd asd asdasdasd asd asd asdasd asd asd asdasd asd

As dasdasdas dasdasd asd adasd asd asd

**Assinatura:**

**Data**: