**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, pela sua divina misericórdia e por tudo de tão maravilhoso que nos tem dado.

À minha família, pai, mãe, irmã, sobrinho, tios e tias pelo carinho, apoio e admiração que vão além de nossos laços sanguíneos.

Ao Professor Odahyr Cavalini, por toda sua sabedoria, paciência, disposição, e pela incomensurável contribuição dada ao longo do curso.

À Escola de Engenharia de Piracicaba pelo apoio.

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: Processo de tradução de um compilador

Figura 2:

Figura 3: Diagrama de classe em UML do padrão Composite

Figura 4: Diagrama de classe em UML do padrão visitor

Figura 5: Clico de desenvolvimento do TDD (Vermelho, Verde, Refatorar)

Figura 6: Exemplo de classe “Usuario”

Figura 7: Teste unitário da classe “Usuario”

Figura 8: Diagrama de caso de uso

Figura 9: Diagrama de atividade de login

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*

IoC – *Invertion of Control*

DI – *Dependency Injection*

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. Álgebra
   3. Compiladores
      1. Análise Léxica
      2. Análise Sintática
      3. Árvore Sintática
      4. Análise Semântica
      5. ANTLR
   4. Design Patterns
      1. Composite
      2. Visitor
      3. Inversion of Control
      4. Dependency Injection
   5. Domain-Driven Design
      1. Ubiquitous Language
      2. Bounded Contexts
      3. Design Tático
      4. Design Estratégico
   6. Test Driven Development
   7. Agile Development
   8. Trabalhos Relacionados
      1. Tópico CCcc
   9. Trabalhos Relacionados
   10. Considerações Finais
3. PROJETO
   1. Considerações Iniciais
   2. Especificação de Usuários
   3. Especificação de Requisitos
   4. Artefatos de Análise
   5. Artefatos de Projeto
   6. Considerações Parciais
4. Desenvolvimento
   1. Considerações Iniciais
   2. Aaaaa
   3. Bbbbb
   4. Verificação e Validação
   5. Considerações Parciais
5. 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, que seja simples de estender fornecendo um mecanismo de acréscimo de funcionalidades sem a necessidade de recompilação da aplicação.

* 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**

Para que seja possível o desenvolvimento deste projeto, foram necessárias diversas ferramentas (em sua grande maioria de código aberto), princípios, padrões e práticas.

Para o versionamento do código fonte do projeto, foi utilizado o sistema Git, que permite o gerenciamento descentralizado de versões.

Como ferramenta para automação da compilação de código Java e gerenciamento de dependências, foram utilizados o Maven e o Gradle.

Sobre a geração do analisador léxico, analisador sintático e a construção da árvore sintática, foi utilizado a ferramenta ANTLR.

Todo o desenvolvimento de código fonte do projeto foi feito utilizando a IDE IntelliJ IDEA, pois possui integração com diversas ferramentas de automação e testes.

Testes unitários, testes de integração e testes de sistema foram feitos utilizando-se a ferramenta JUnit em conjunto com a biblioteca Hamcrest.

Para a montagem da arquitetura da aplicação, no qual foi feita a separação em mais de um projeto com várias camadas, dando foco principal na camada de domínio, foi utilizado o *framework* Spring Boot juntamente com o Spring MVC.

Quanto ao armazenamento dos complementos do projeto, foi utilizado o framework ORM Hibernate, e a persistência em banco de dados utilizando o SGBD H2.

Foi utiliza tal:

* Ferramenta tal (link)
  1. **Cronograma Realizado**

Acrescentar informações sobre o que foi feito antes do começo da monografia

Tendo como base alguns princípios de desenvolvimento ágil (principalmente *Extreme Programming*) e utilizando iterações curtas, cronograma foi dividido como segue:

Tabela 1: Cronograma

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Atividades | Mar | Abr | Mai | Jun | Jul | Ago | Set | Out | Nov |
| Estudo |  |  |  |  |  |  |  |  |  |
| Análise |  |  |  |  |  |  |  |  |  |
| Projeto |  |  |  |  |  |  |  |  |  |
| Desenvolvimento |  |  |  |  |  |  |  |  |  |

Fonte: Próprio autor

**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. Mínimo Múltiplo Comum**

Aaaa.

**2.2.2. Máximo Divisor Comum**

Aaaa.

**2.3. Compiladores**

Segundo Louden (2004), compiladores são *softwares* que convertem um código fonte (ou linguagem origem), em um código objeto (ou linguagem-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

As etapas de compilação são complexas, exigindo um esforço significativo, sendo que os primeiros compiladores eram escritos em *Assembly*. Hoje, tem-se um conjunto de ferramentas que facilitam a criação e manutenção de compiladores, muitas dessas ferramentas são escritas em linguagem como Java, C e C++ e já automatizam boa parte da construção de um compilador. Essas ferramentas geram códigos que podem ser incluídos no projeto do compilador. Um exemplo são os geradores de analisadores léxicos, que com base em expressões regulares, geram um algoritmo capaz de identificar os elementos léxicos de uma linguagem de programação. Outro exemplo são os geradores de analisadores sintáticos, que a partir de uma gramática, gerar um algoritmo para verificar a sintaxe de uma dada linguagem.

Outro conceito importante no estudo de compiladores é a otimização (ou técnicas de melhoria de código), que tem por objetivo produzir um compilador que gere código mais eficiente.

O tempo de compilação é também um fator muito importante que deve ser levado em consideração durante o desenvolvimento de um compilador.

**2.3.1. Análise Léxica**

Análise léxica (ou varredura) é a primeira fase de um compilador, que consiste em analisar a entrada de linhas de caracteres e produzir uma sequência de símbolos chamados *tokens* (sequência de caracteres com um significado coletivo). É nessa fase que são reconhecidos os identificadores, números, palavras reservadas, constantes (e demais itens pertencentes à linguagem de programação), além de executar outras tarefas como tratar espaços, remover comentários, contar do número de linhas para auxiliar as etapas subsequentes da compilação na emissão de mensagens de alerta e erros, etc. Um conceito muito importante no estudo de compiladores é a otimização, que se refere as atentivas de produzir um compilador que gere um código mais eficiente. Essa é uma etapa cada vez mais importante e complexa devido à grande variedade de arquiteturas de processadores. O tempo de compilação é outro fator muito importante que deve ser levado em consideração durante o desenvolvimento de um compilador. Usualmente, a análise léxica é invocada pelo analisador sintático cada vez que um novo *token* é encontrado.

**2.3.2. Análise Sintática**

A análise sintática (também conhecida como *parser*) é a segunda etapa do processo de compilação, e tem como tarefa principal determinar se o programa de entrada representado pelo fluxo de *tokens* produzido pelo analisador léxico possui as sentenças válidas para a linguagem de programação (normalmente representada utilizando gramáticas livres de contexto para especificar a sintaxe), ou seja, essa etapa do processo de compilação deve determinar se uma dada entrada é válida ou não. Além disso, deve se encarregar de capturar dados importantes para que as fases subsequentes do processo de compilação (como a análise semântica e geração de código).

Os métodos mais comumente usados nos compiladores são classificados como ascendentes (ou *top-down*) e descendentes (*bottom-up*). Os algoritmos ascendentes constroem árvores sintáticas da raiz para as folhas, enquanto que os algoritmos descendentes começam pelas folhas e trabalham árvore acima até a raiz (em ambos os casos, a entrada é varrida da esquerda para a direita, um símbolo de cada vez). Normalmente, os algoritmos ascendentes tendem a ser implementados utilizando geradores de analisadores sintáticos (como o ANTLR), enquanto os algoritmos ascendentes podem ser escritos manualmente.

**2.3.3. Árvore Sintática**

A árvore sintática é uma estrutura de dados em forma de árvore que representa sequência hierárquica da linguagem de programação, sendo gerada no processo de análise sintática conforme a gramática da linguagem é verificada. Essa estrutura representa a hierarquia do programa fonte, podendo utilizar poucas informações, ou pode ser uma representação da gramática da linguagem, incluindo os *tokens* e cadeias de caracteres lidos no processo de análise léxica (esse tipo de estrutura é denominada *concrete syntax tree*).

Pode-se também utilizar *abstract syntax tree* (árvore sintática abstrata) é uma representação específica e simplificada de uma árvore que é gerada tendo como base uma *concrete syntax tree*.

**2.3.4. Análise Semântica**

A análise semântica é a terceira etapa do processo de compilação verificar aspectos relacionados ao significado das instruções. As validações que não podem ser executadas pelas etapas anteriores devem ser executadas durante a análise semântica a fim de garantir que o programa fonte esteja coerente. Um exemplo que ilustra muito bem essa etapa de validação de tipos é a atribuição de objetos de tipos ou classes diferentes. Em alguns casos, o compilador realiza a conversão automática de um tipo para outro que seja adequado à aplicação.

Os tipos de dados são muito importantes nessa etapa da compilação, pois com base nos tipos, o analisador semântico pode definir quais valores podem ser manipulados (isso é conhecido com checagem de tipo). Os sistemas de tipos de dados podem ser divididos em dois grupos: sistemas dinâmicos e sistemas estáticos. Muitas das linguagens utilizam o sistema estático (esse sistema é predominante em linguagens compiladas), pois essa informação é utilizada durante a compilação e simplifica o trabalho do compilador. Outras linguagens utilizam um mecanismo muito interessante chamada inferência de tipos, que permite a uma variável assumir vários tipos durante o seu ciclo de vida, permitindo que a ela possa ter vários valores (linguagens de programação como Haskel tira proveito desse mecanismo).

**2.3.5. 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, o ANTLR gera analisadores que podem construir e percorrer árvores de sintáticas.

Figura x: Gramática g4 de uma calculadora simples

grammar calculator;

equation

: multiplyingExpression ((PLUS | MINUS) multiplyingExpression)\*

;

multiplyingExpression

: number ((TIMES | DIV) number)\*

;

number

: MINUS? DIGIT + (POINT DIGIT +)?

;

LPAREN

: '('

;

RPAREN

: ')'

;

PLUS

: '+'

;

MINUS

: '-'

;

TIMES

: '\*'

;

DIV

: '/'

;

POINT

: '.'

;

DIGIT

: ('0' .. '9')

;

WS

: [ \r\n\t] + -> channel (HIDDEN)

;

Fonte: Próprio autor

**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. Composite**

Segundo o autor (Gama, 2000), o padrão *composite* (composição) tem por objetivo compor objetos em estruturas de árvore para representar hierarquia partes-todo, permitindo aos clientes tratarem de maneira uniforme objetos individuais e composições de objetos.

Figura 2: Diagrama de classe em UML do padrão *Composite*



Fonte: Próprio autor

**2.4.2. Visitor**

Segundo Gama (2000), o padrão *visitor* (visitante) representa uma operação a ser executada nos elementos de uma estrutura de objetos, permitindo assim definir uma nova operação sem mudar as classes dos elementos sobre os quais opera. A sua ideia é separar as operações que serão executadas em determinada estrutura de sua representação. Assim, incluir ou remover operações não terá nenhum efeito sobre a interface da estrutura, permitindo que o resto do sistema funcione sem depender de operações específicas.

É comumente utilizado em conjunto com estruturas de dados em árvore.

Figura 3: Diagrama de classe em UML do padrão *visitor*



Fonte: Próprio autor

**2.4.3. Inversion of Control**

*Inversion of control* (Inversão de Controle) é o 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.

**2.4.4. 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 (serviço) para o dependente (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 feita pelo construtor da classe, método ou atributo (onde na maioria dos casos, o tipo do parâmetro é uma interface ou classe abstrata).

**2.5. Domain-Driven Design**

Segundo Evans (2010), Domain-Driven Design (modelagem dirigida pelo domínio) é um conjunto de práticas que tem por objetivo a construção de um software que expresse de forma bem clara um problema em questão, auxiliando os desenvolvedores e os analistas de domínio a produzir software mais coerente com o negócio utilizando para isso desenvolvimento iterativo e a comunicação constante. Fornece práticas em nível tático, na criação de um modelo de domínio sólido, e em nível estratégico, auxiliando na identificação das áreas mais importantes a serem atacadas e como essas áreas se comunicam. Um modelo de domínio é um conjunto de objetos interconectados, projetados para atender regras de negócio complexas, onde cada um deles tem um significado próprio dentro da área de negócio a ser atendida. Dentre as principais vantagens, pode-se destacar:

* Troca de conhecimento entre desenvolvedores e analistas de domínio contribuindo para reduzir as chances de que o conhecimento sobre o modelo de domínio fique nas mãos de poucas pessoas;
* Melhora experiência de usuário, uma vez que as telas do software passam a refletir uma operação de negócio;
* O código do software expressa melhor o negócio e a arquitetura da solução.

DDD não se trata de um padrão ou arquitetura para se desenvolver softwares, e pode ser utilizado com diversos conceitos, como a arquitetura em camadas, cebola, hexagonal, entre outros (o importante é que o modelo de domínio se mantenha isolado de detalhes técnicos). Além disso, fornece uma série de conceitos e padrões que auxiliam no design da solução, tanto em nível tático como em nível estratégico.

**2.5.1. Ubiquitous Language**

Os desenvolvedores e os analistas de domínio devem compartilhar uma linguagem comum, que deve ser compreendida por todos, não apresentar ambiguidades, e mais importante, essa linguagem deve definir a terminologia de negócios e não terminologia técnica. Evans (2010) denominou essa linguagem como *ubiquitous language* (linguagem onipresente, ou linguagem ubíqua), ou seja, é a linguagem criada pelo time de desenvolvimento em conjunto com os analistas de domínio que expressa o negócio em comunicação falada, em documentos, no próprio código, ou em um contexto específico. Toda vez que alguém perceber que um determinado conceito do domínio possui várias palavras que o represente, essa pessoa deve tentar readequar tanto a linguagem falada e escrita, quanto o código.

**2.5.2. Bounded Contexts**

*Bounded Contexts* (contextos delimitados) é uma fronteira conceitual onde reside o modelo de domínio e sua linguagem ubíqua. Buscam delimitar um domínio complexo em contextos baseados nas intenções do negócio, isto é, delimitando as intenções das entidades com base no contexto que elas pertencem e fornecendo aos membros das equipes de desenvolvimento um claro entendimento do que deve ser consistido e desenvolvido independentemente.

Dividir uma grande aplicação entre diferentes contextos delimitados adequadamente permitirá que a aplicação se torne mais modular, ajudando a separar preocupações diferentes e tornando a aplicação fácil de gerenciar e aprimorar. Cada um desses Contextos Limitados tem uma responsabilidade específica e pode operar de forma quase autônoma.

**2.5.3. Design Tático**

Em nível tático, há uma série de padrões que auxiliam na criação do modelo de domínio:

* Entidades: classes de objetos que necessitam de uma identidade;
* *Value objects* (objetos de valor): objetos que só carregam valores, mas que não possuem distinção de identidade;
* Serviços: classes que contém lógica de negócio que não pertence à nenhuma Entidade ou objetos de valor;
* Eventos: ações que devem ser executadas dependendo das circunstâncias;
* *Aggregates* (agregados): conjuntos de entidades ou objetos de valor que são encapsulados numa única classe;
* Módulos: abstrações que têm por objetivos agrupar classes por um determinado conceito do domínio;
* Repositórios: classes responsáveis por administrar o ciclo de vida dos outros objetos e de prover, alterar, e eliminar instâncias destes;
* *Factories* (fábricas): classes responsáveis pela criação de agregados ou objetos de valor.

**2.5.4. Design Estratégico**

Em nível estratégico, há inúmeros padrões úteis para se lidar com soluções muito complexas, compostas por vários sistemas, sejam eles internos ou externos. Além do próprio *bounded context*, apresenta-se os conceitos de *subdomain* (uma separação do domínio entre aquilo que é a motivação da aplicação e aquilo que é auxiliar), *context map*, *shared kernel* (que pode ser utilizado para interligar contextos ou simplesmente para reaproveitamento de entidade), *customer/supplier*, *conformist*, *anticorruption layer*, *separate ways*, *open host service* e *published language*.

**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 4: 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 5 e a Figura 6 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 5: 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 6: 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**

*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), tendo como objetivo minimizar os riscos utilizando períodos curtos de desenvolvimento (chamadas iterações), 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**

De acordo com Beck (2000), o 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 projeto 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**

Falar sobre XP diretamente

**2.8. Spring Framework**

Seu núcleo é o seu contêiner de inversão de controle (IoC), que fornece um meio consistente de configurar e gerenciar objetos utilizando reflexões em Java. O contêiner é responsável pela criação desses objetos, invocação de métodos de inicialização e configuração, e pelo gerenciamento de seu ciclo de vida (objetos criados pelo contêiner também são chamados de objetos gerenciados ou *beans*). A configuração do contêiner pode ser feita a partir de arquivos XML ou por anotações Java que são detectadas na inicialização dos objetos. Uma vez registrados no contêiner, os objetos podem ser obtidos por meio de pesquisa ou por injeção de dependência.

Em muitos casos, não é necessário usar o contêiner para se utilizar as outras bibliotecas que compõem o Spring Framework, embora seu o uso provavelmente tornará um aplicativo mais fácil de configurar e personalizar. O contêiner Spring fornece um mecanismo consistente para configurar aplicativos e se integra a quase todos os ambientes Java, desde aplicativos de pequena escala, até grandes aplicações empresariais.

**2.8.1. Spring MVC**

Aaaa.

**2.8.2. Spring JPA**

Aaaa.

**2.8.3. Spring Boot**

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**

Colocar a a descrição da classificação.

**3.3.1. Requisitos Funcionais**

Os requisitos devem ser citados (foram adotados os requisitos das tabelas 1 a 5).

Tabela 1: Requisito funcional 1

|  |  |
| --- | --- |
| **Identificador** | **Rq\_01** |
| **Nome** | Plataforma Web |
| **Descrição** | O sistema deve ser desenvolvido em plataforma web devido fácil acesso ao sistema |
| **Técnicas de Elicitação usadas** | Entrevista |
| **Classificação** | Essencial |

Fonte: Próprio autor

Tabela 1: Requisito funcional 2

|  |  |
| --- | --- |
| **Identificador** | **Rq\_02** |
| **Nome** | Resolver expressão algébrica de forma iterativa |
| **Descrição** | O sistema deve resolver de forma iterativa uma expressão algébrica recebida como entrada, e exibir passo a passo os resultados. |
| **Técnicas de Elicitação usadas** | Entrevista |
| **Classificação** | Essencial |

Fonte: Próprio autor

Tabela 2: Requisito funcional 3

|  |  |
| --- | --- |
| **Identificador** | **Rq\_03** |
| **Nome** | Validação de Usuário |
| **Descrição** | Será necessário o usuário validar o acesso para poder cadastrar novas heurísticas antes de entrar no sistema assim haverá um registro de suas realizações na aplicação web. |
| **Técnicas de Elicitação usadas** | Entrevista |
| **Classificação** | Essencial |

Fonte: Próprio autor

Tabela 4: Requisito funcional 4

|  |  |
| --- | --- |
| **Identificador** | **Rq\_04** |
| **Nome** | Sistema Web Responsivo |
| **Descrição** | Possibilidade de a aplicação rodar em qualquer plataforma, sendo ela mobile ou em navegadores. |
| **Técnicas de Elicitação usadas** | Entrevista |
| **Classificação** | Desejável |

Fonte: Próprio autor

Tabela 5: Requisito funcional 5

|  |  |
| --- | --- |
| **Identificador** | **Rq\_05** |
| **Nome** | E necessário ter internet para acesso ao Sistema |
| **Descrição** | A pessoa na qual ira acessar o sistema deve ter acesso a internet. |
| **Técnicas de Elicitação usadas** | Entrevista |
| **Classificação** | Essencial |

Fonte: Próprio autor

**3.3.1. Diagramas de caso de uso**

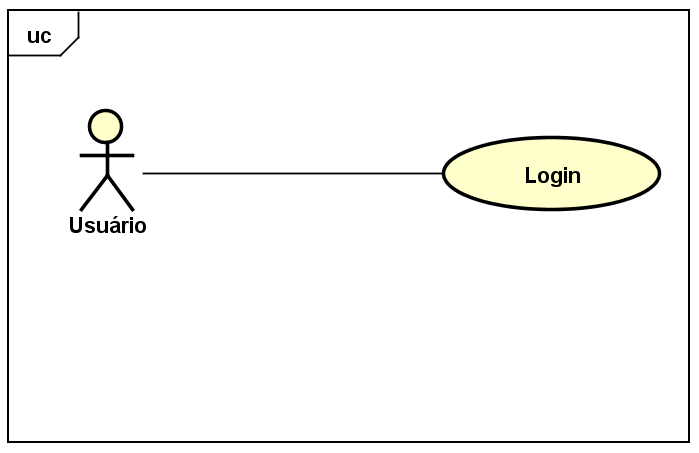
Fazer uma citação dos diagramas.

Figura 7: Diagrama de caso de uso para resolução de expressões algébricas



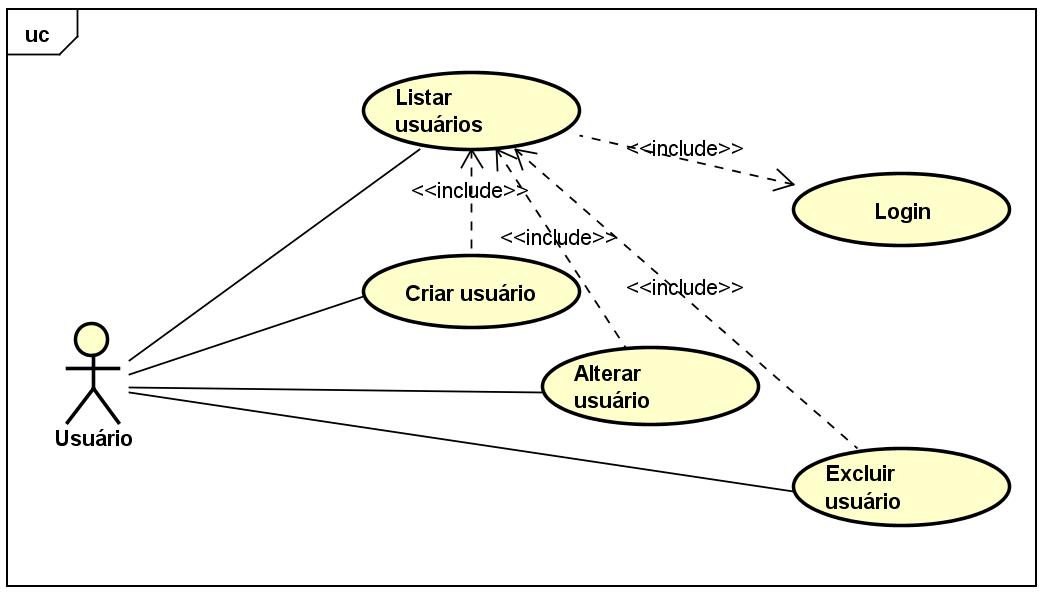
Fonte: Próprio autor

Figura 8: Diagrama de caso de uso de login



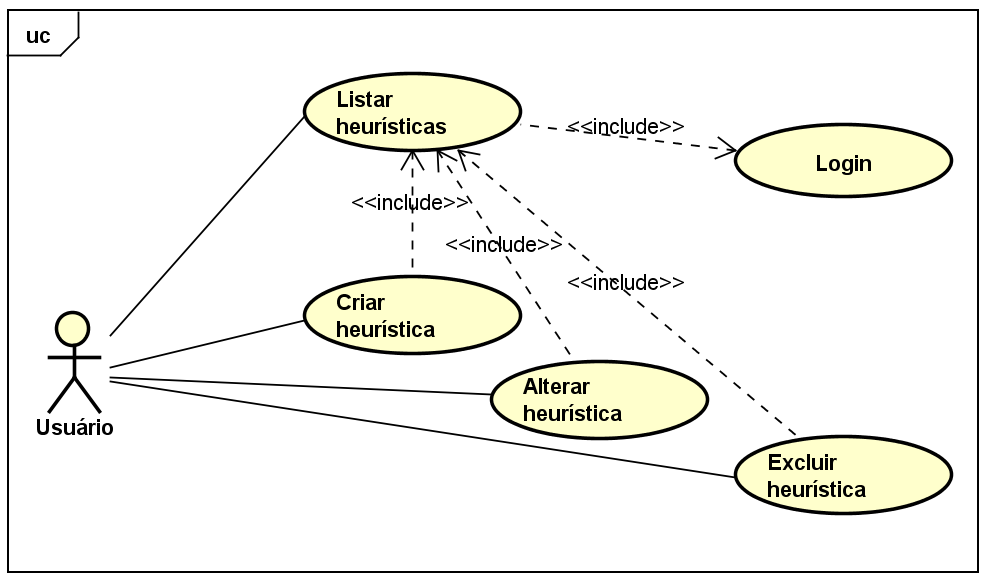
Fonte: Próprio autor

Figura 9: Diagrama de caso de uso para gerenciar usuários



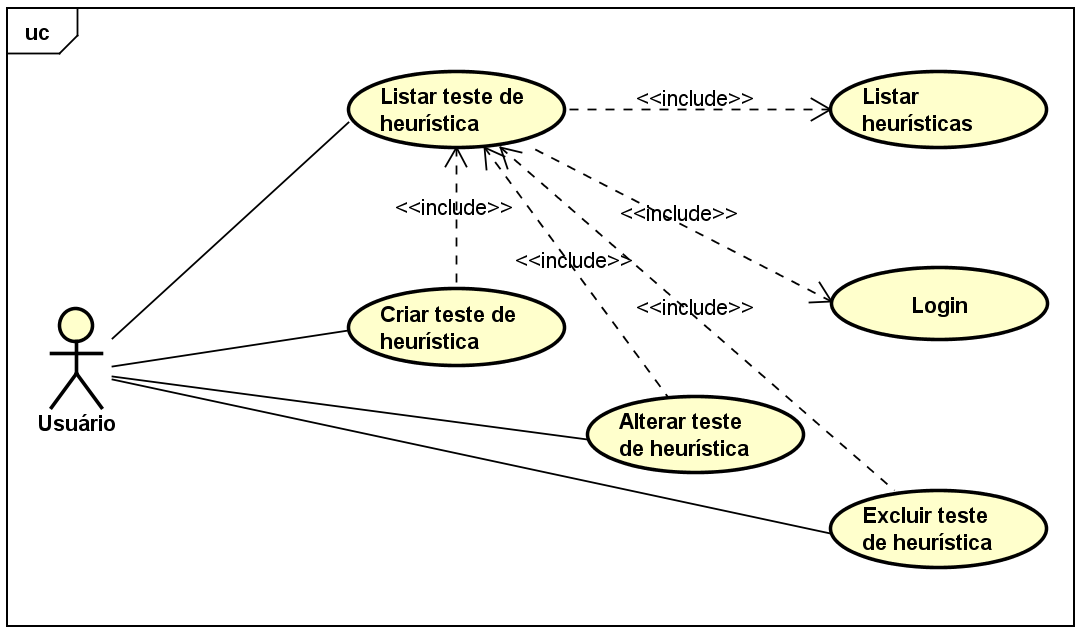
Fonte: Próprio autor

Figura 10: Diagrama de caso de uso gerenciar heurísticas



Fonte: Próprio autor

Figura 11: Diagrama de caso de uso para gerenciar testes de heurísticas



Fonte: Próprio autor

**3.3.2. Diagramas de atividade**

Figura 8: Diagrama de atividade para resolver expressões



Fonte: Próprio autor

Figura 9: Diagrama de atividade de login



Fonte: Próprio autor

Figura 10: Diagrama de atividade da listagem de usuários



Fonte: Próprio autor

Figura 11: Digrama de atividade de criação de usuário



Fonte: Próprio autor

Figura 12: Digrama de atividade de alteração de usuário



Fonte: Próprio autor

Figura 13: Digrama de atividade



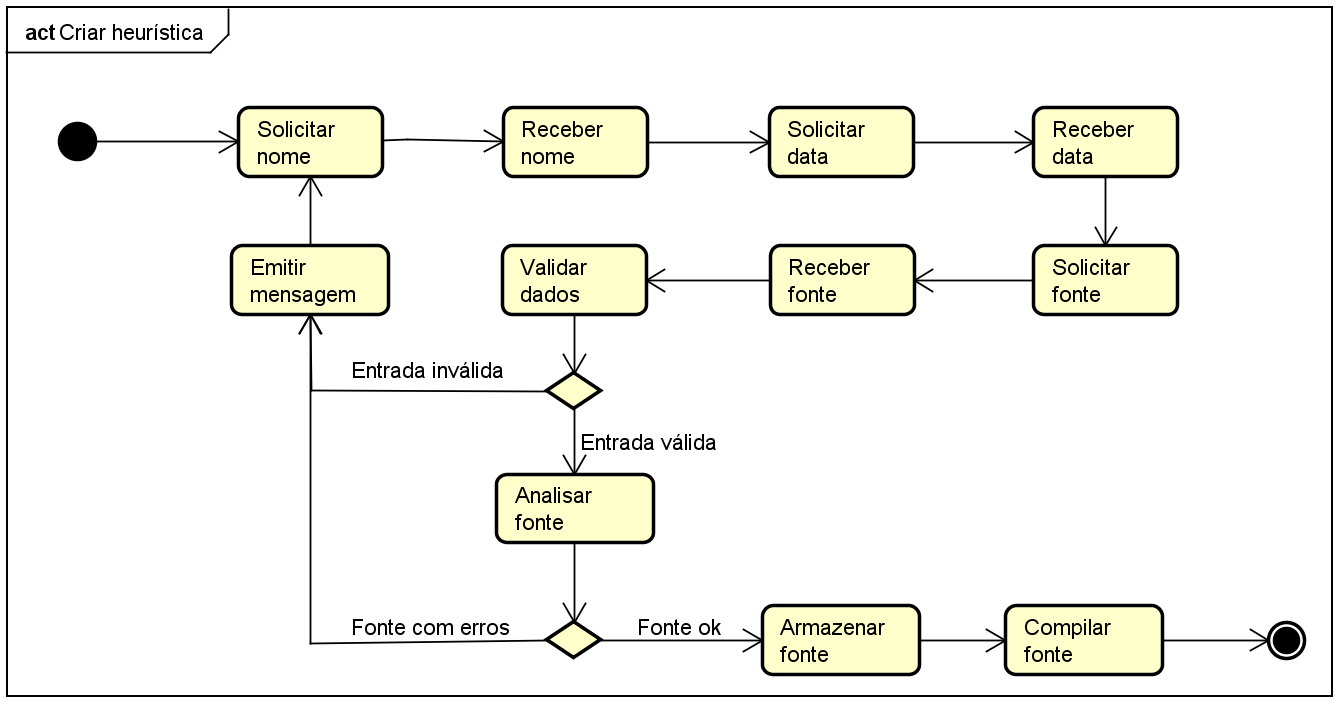
Fonte: Próprio autor

Figura 14: Digrama de atividade



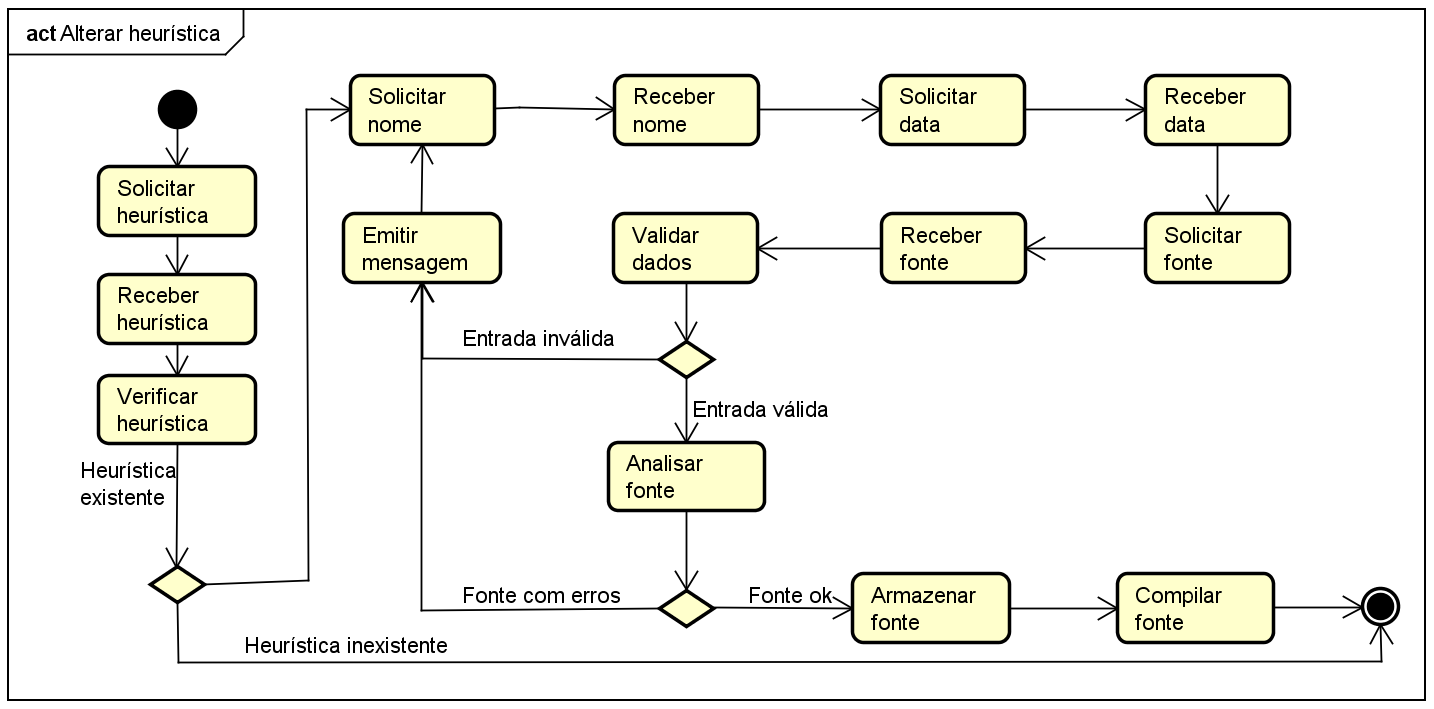
Fonte: Próprio autor

Figura 15: Digrama de atividade



Fonte: Próprio autor

Figura 16: Digrama de atividade



Fonte: Próprio autor

Figura 17: Diagrama de atividade



Fonte: Próprio autor

Figura 18: Diagrama de atividade

Fonte: Próprio autor

Figura 19: Diagrama de atividade

Fonte: Próprio autor

Figura 20: Diagrama de atividade

Fonte: Próprio autor

Figura 21: Diagrama de atividade

Fonte: Próprio autor

**3.4. Artefatos de Análise**

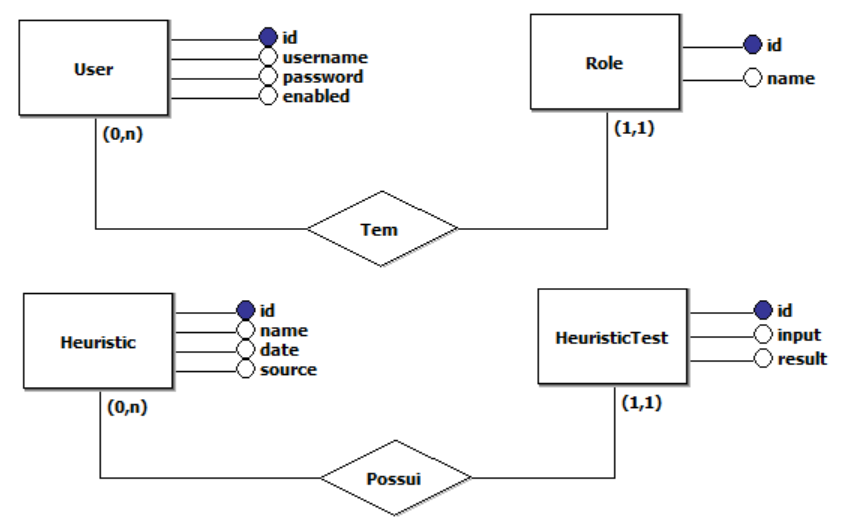
Aaaa.

**3.4.1. Diagramas de classe**

Diagramas sem tipos, em visibilidade.

**3.4.2. Modelo DER**

Figura x: Modelo DER



Fonte: Próprio autor

**3.5. Artefatos de Projeto**

**3.4.1. Diagrama de classe**

Diagrama completo.

Aaaa.

**3.5.1. Dicionário de tabelas**

Utilizar o dicionário de entidades.

Tabela 1: Dicionário da entidade role

|  |  |
| --- | --- |
| Identificador de Entidade | #Role |
| Descrição | Papel do usuário. |
| Motivo | Determinada quais papéis o usuário pode ter no sistema. |
| Requisito(s) Relacionado(s) | #User |
| Tabela | role |

Fonte: Próprio autor

Tabela 1: Dicionário da entidade user

|  |  |
| --- | --- |
| Identificador de Entidade | #User |
| Descrição | Usuário do sistema. |
| Motivo | Armazenar os usuários que podem gerenciar as heurísticas do sistema. |
| Requisito(s) Relacionado(s) | *(não se aplica)* |
| Tabela | user |

Fonte: Próprio autor

Tabela 1: Dicionário da entidade heuristic

|  |  |
| --- | --- |
| Identificador de Entidade | #Heuristic |
| Descrição | Heurística de simplificação de expressões. |
| Motivo | Armazenar os algoritmos de simplificação de expressões algébricas. |
| Requisito(s) Relacionado(s) | *(não se aplica)* |
| Tabela | heuristic |

Fonte: Próprio autor

Tabela 1: Dicionário da entidade heuristictest

|  |  |
| --- | --- |
| Identificador de Entidade | #HeuristicTest |
| Descrição | Testes de uma heurística de simplificação de expressões. |
| Motivo | Conter os testes para validação dos algoritmos de simplificação de expressões algébricas. |
| Requisito(s) Relacionado(s) | #Heuristic |
| Tabela | heuristic\_test |

Fonte: Próprio autor

Dicionário de Dados

|  |  |
| --- | --- |
| Identificador do Dado | #NomeArquivo |
| Identificador de Entidade | #Arquivo |
| Rótulo | nomeArquivo |
| Nome (s) | Nome do arquivo |
| Descrição | Nome do arquivo armazenado no banco de dados |
| Requisito(s) Relacionado(s) | *(não se aplica)* |
| Tipo de Dado | Texto |
| Unidade de Medida | *(não se aplica)* |
| Precisão Numérica | *(não se aplica)* |
| Tamanho | *30* |
| Valor padrão | Nulo |
| Máscara de Edição | *(não se aplica)* |

**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**

SILVA, Luiz Paulo Moreira. O que é álgebra?. Brasil Escola. Disponível em <http://brasilescola.uol.com.br/o-que-e/matematica/o-que-e-algebra.htm>. Acesso em 08 de maio de 2017.

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**: