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 e colegas pelo carinho, apoio e admiração, como também aos professores, por toda sua sabedoria, paciência, disposição, e pela enorme contribuição dada ao longo do curso.

.

RESUMO

Este projeto visa desenvolver um software de conhecimento computacional matemático de código aberto, cujo intuito principal é analisar, interpretar, simplificar e resolver de forma iterativa expressões algébricas com base em heurísticas simples, para que possa ser utilizado pelos os discentes, auxiliando-os em seus estudos nos mais variados temas das disciplinas de exatas (uma vez que a maioria das ferramentas disponíveis que tratam do tema são proprietárias). Para atingir este objetivo, as heurísticas tem como base conceitos adquiridos nas disciplinas de exatas do curso de Ciência da Computação, como Álgebra Linear, Geometria Analítica, Cálculo, Métodos Numéricos, entre outras, e para a implementação dessas heurísticas, são utilizados conceitos de teoria de compiladores para analisar as expressões, padrões de projetos para desenho e arquitetura de *software*, modelagem dirigida pelo domínio utilizando práticas e padrões com foco principal no domínio, desenvolvimento dirigido por testes para garantir o correto funcionamento das rotinas e diversos *frameworks* do mundo Java.

ABSTRACT

This project aims to develop an open source mathematical computational knowledge software whose main purpose is to analyze, interpret, simplify and solve iteratively algebraic expressions based on simple heuristics, so that it can be used by the students, helping them in their Studies in the most varied subjects of the exact disciplines (since most of the available tools that deal with the subject are proprietary). To achieve this goal, heuristics are based on concepts acquired in the mathematics subjects of the Computer Science course, such as Linear Algebra, Analytical Geometry, Calculus, Numerical Methods, among others, and for the implementation of these heuristics, compiler theory to analyze expressions, design patterns for software design and architecture, domain-driven design using domain-focused practices and standards, test driven development to ensure the correct operation of routines and various frameworks in the Java world.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

TDD – *Test Driven Development*

XP – *Extreme Programming*

DDD – *Domain Driven Design*

UML – *Unified Modeling Language*

IoC – *Invertion of Control*

DI – *Dependency Injection*

MVC – *Model View Controller*

JPA – *Java Persistence API*

ORM – *Object Relational Mapping*

IDE – *Integrated Development Environment*

SGBD – Sistema Gerenciador de Banco de Dados

HTTP – *Hypertext Transfer Protocol*

CRUD – *Create, read, update, delete*

XML – *eXtensible Markup Language*

AOP – *Aspect Oriented Programming*

CST – *Concrete Syntax Tree*

AST – *Abstract Syntax Tree*

JDK – *Java Development Kit*

HTML – *Hypertext Markup Language*

SUMÁRIO

Capítulo 1

Introdução

1.1. 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 acessível todo o conhecimento sistemático computável 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), e mesmo quando essas ferramentas dispõem de alguma forma de utilização sem custo, os recursos oferecidos são extremamente limitados. 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. Também, foi realizada uma consulta em artigos acadêmicos, e não foram encontradas referências que tratem do tema.

Portanto, 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.

1.2. Objetivo

Desenvolver um *software* de conhecimento computacional matemático de código aberto, 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, uma vez que as ferramentas disponíveis são proprietárias. O software deve fornecer uma forma simples de se estendê-lo utilizando-se linguagens embarcadas, fazendo com que não seja necessária a recompilação do código fonte do projeto.

Além do *software*, o projeto visa fornecer métodos para que seja possível a difusão de mais ferramentas e algoritmos acerca do tema.

1.3. Motivação

Há diversas ferramentas que tentam tornar acessível todo o conhecimento sistemático computável fornecendo respostas para consultas factuais, porém, a maioria dessas ferramentas além de serem pagas, não demonstram como esse objetivo é atingido (uma vez que não se tem acesso ao código fonte).

Este projeto tem como objetivo 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, bem como ser um ponto de partida para que novas ferramentas sobre o tema sejam desenvolvidas, ou simplesmente demonstrar uma abordagem sobre o tema.

1.4. 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.

Capítulo 2

Revisão Bibliográfica

2.1. Considerações Iniciais

Neste capítulo são apresentadas as referências bibliográficas utilizadas como base para a realização do projeto.

Na primeira parte, é dada uma visão geral sobre a teoria de compiladores, suas fases e as estruturas de dados utilizadas durante o processo. É dada nessa parte também uma explanação sobre geradores de analisadores léxicos e sintáticos, como o ANTLR.

Em seguida, é dada uma explicação sobre padrões de projetos, e é dada uma explicação introdutória sobre alguns padrões de desenho importantes, bem como alguns padrões de arquitetura de software.

A próxima seção descreve como o Domain-Driven Design auxilia os desenvolvedores e os analistas de domínio a produzir software mais coerente com o negócio.

Na sequência, é dada uma descrição completa sobre o que é Test Driven Development, e como essa técnica pode auxiliar a minimizar erros no desenvolvimento de software.

Por fim, é feita uma análise geral da tecnologia Java, e como outras linguagens de programação e *frameworks* podem ser usados nessa plataforma.

2.2. Compiladores

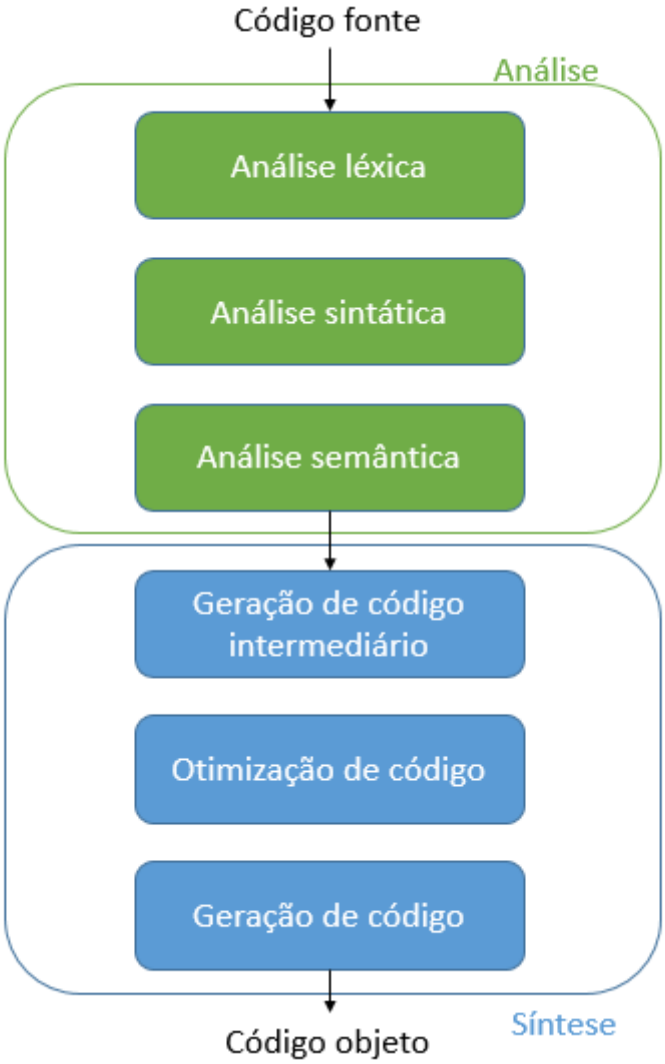
Segundo Louden (2004), compiladores são *softwares* que convertem um código fonte (ou linguagem origem), em um código objeto (ou linguagem-objeto). Normalmente, o código fonte é escrito em uma linguagem de programação de alto nível, com grande capacidade de abstração, e o código objeto é escrito em uma 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). A figura 1 demonstra o processo de tradução de um compilador.

Figura 1: Processo de tradução de um compilador

Fonte: Próprio autor

As etapas de compilação são complexas, e são compostas de dois processos: análise e síntese. O processo de análise é composto pela análise léxica, análise sintática e análise semântica, e tem como objetivo analisar o código fonte dado como entrada (esse processo tende a ser mais matemático), enquanto a síntese é composta pela geração de código intermediário, otimização de código e geração de código (requerendo técnicas mais especializadas). A figura 2 demonstra o processo de compilação.

Figura 2: Processo de compilação



Fonte: Próprio autor.

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.

2.2.1. 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.

2.2.2. 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.

2.2.3. 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), que é uma representação específica e simplificada de uma árvore que é gerada tendo como base uma *concrete syntax tree*.

2.2.4. 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 neles, 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 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 assumir diversos valores (linguagens de programação como Haskel tiram proveito desse mecanismo).

2.2.5. 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 (a figura 3 demonstra um exemplo de gramática g4 do ANTLR).

Figura 3: 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

2.3. Design Patterns

De acordo com Shvets (2001), um Design Pattern (padrão de projeto) descreve de forma geral um problema e como resolvê-lo, de tal maneira que se possa utilizá-lo indefinidas vezes, nunca fazendo da mesma maneira. Não é um projeto acabado que possa ser transformado diretamente em código, mas pode acelerar o processo de desenvolvimento fornecendo paradigmas de desenvolvimento testados e comprovados.

Muitas vezes, as pessoas só entendem como aplicar certas técnicas de *design* de *software* para determinados problemas. Estas técnicas são difíceis de aplicar a uma gama mais ampla de problemas. Os padrões de projeto fornecem soluções gerais, documentadas em um formato que não estão ligadas a um problema específico.

Também, os padrões permitem que os desenvolvedores se comuniquem usando nomes bem conhecidos e bem compreendidos para interações de *software*.

Segundo Gamma (2000), os padrões de projeto têm quatro elementos essenciais:

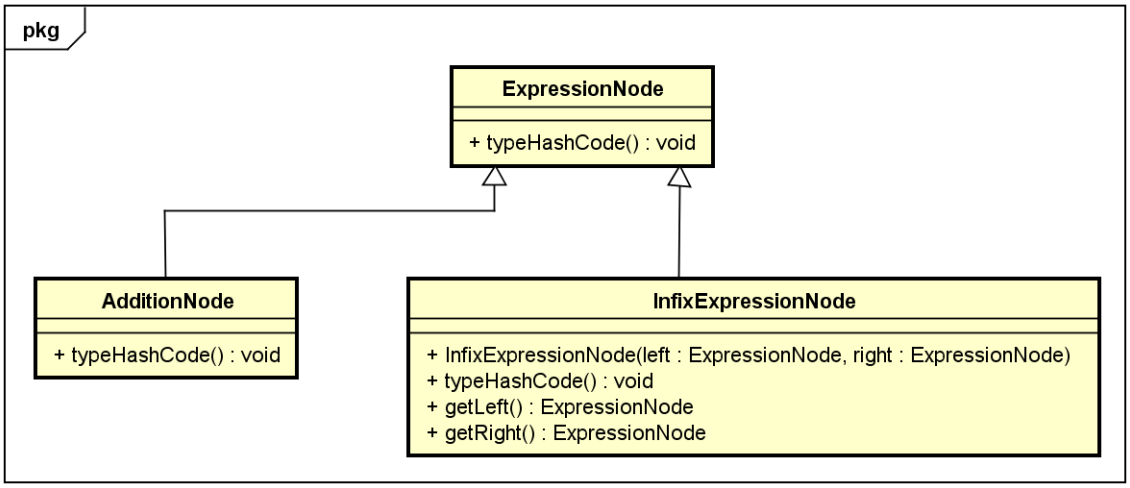
1. Nome do padrão: referência que pode ser usada para descrever um problema de projeto, suas soluções e consequências, elevando de forma imediata o vocabulário dos desenvolvedores em um projeto.
2. Problema: descreve em que situação o padrão pode ser aplicado, explicando o problema e seu contexto.
3. Solução: elementos que compõem o padrão de projeto, como seus relacionamentos, suas responsabilidades e colaborações, não descrevendo um projeto concreto ou uma implementação em particular.
4. Consequências: resultados e análises das vantagens e desvantagens ao se utilizar um padrão.

Ainda segundo o autor (Gamma, 2000), os padrões de projetos podem ser classificados como padrões criacionais (padrões de criação de classe e padrões de criação de objetos), estruturais (padrões sobre composição de classe e objeto utilizando herança) e comportamentais (tratam da comunicação de objetos da classe).

2.3.1. Composite

Segundo Gamma (2000), o padrão de projeto Composite (composição) é um padrão do tipo estrutural e 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. É usado onde é necessário tratar um grupo de objetos de maneira semelhante a um único objeto ou compondo objetos em termos de uma estrutura em árvore para representar parte dela ou sua hierarquia completa (a figura 4 demonstra uma estrutura de classes em árvore do padrão Composite).

Figura 4: Diagrama de classe em UML do padrão Composite



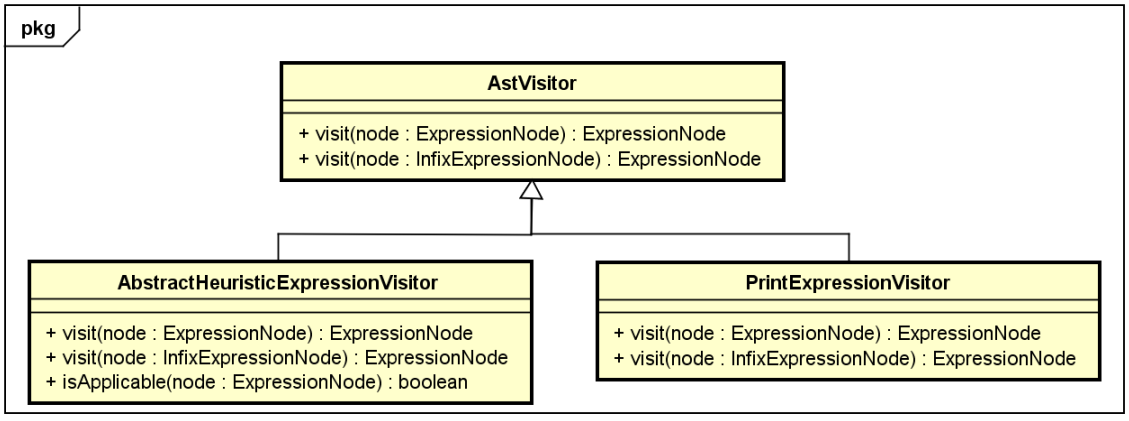
Fonte: Próprio autor

2.3.2. Visitor

De acordo com Gama (2000), o padrão de projeto Visitor (visitante) é um padrão comportamental que 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 uma 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 o padrão Composite e com estruturas de dados em árvore, fornecendo uma forma de navegar em sua estrutura (o diagrama de classes da figura 5 demonstra um exemplo do padrão Visitor).

Figura 5: Diagrama de classe em UML do padrão Visitor



Fonte: Próprio autor

2.3.3. Inversion of Control

Inversion of control (inversão de controle) é o padrão de projeto de *software* que consiste em mudar o 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, permitindo que os objetos podem ser adicionados e testados independentemente uns dos outros objetos.

2.3.4. 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).

2.4. 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 comunicação constante. Fornece práticas em nível tático e estratégico para criação de um modelo de domínio sólido, auxiliando na identificação de áreas 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.

2.4.1. 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.

2.4.2. 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.

2.4.3. 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.

2.4.4. 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*.

2.5. Test Driven Development

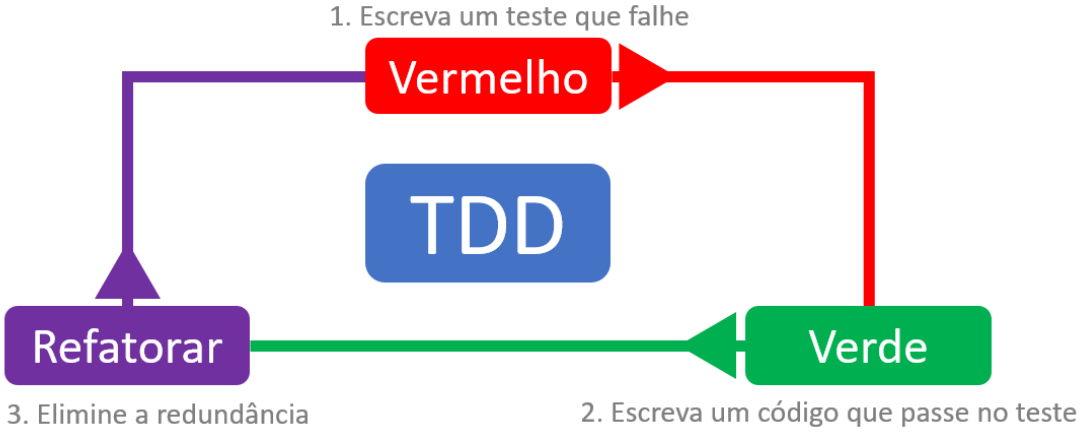
Test Driven Development (Desenvolvimento Dirigido por Testes), é uma técnica de desenvolvimento de software que tem como princípio a criação de testes automatizados antes de qualquer código de produção, com o objetivo de confirmar o funcionamento de uma implementação feita pelo programador. Essa técnica 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 6: 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 7 e a Figura 8 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 7: 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 8: 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

2.6. Tecnologia Java

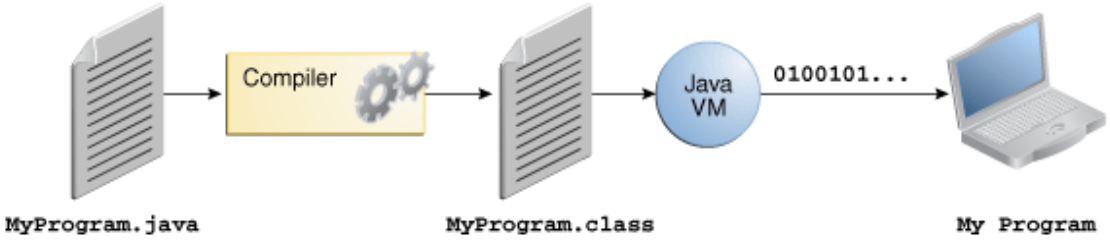
O Java é uma tecnologia composta por uma linguagem de programação e uma plataforma, e é um padrão global para desenvolvimento de aplicações embarcadas, móveis, *web* e softwares corporativos, contando com mais de 9 milhões de desenvolvedores ao redor do mundo.

Permite o desenvolvimento de aplicações portáteis de alto desempenho para uma ampla variedade de plataformas de computação, e é mantido por uma comunidade dedicada de desenvolvedores, arquitetos e entusiastas. Ao disponibilizar aplicações entre ambientes distintos, as empresas podem fornecer mais serviços, aumentando a produtividade, a comunicação e a colaboração com o usuário final.

A linguagem de programação Java é uma linguagem de alto nível que pode ser caracterizada por ser orientada a objetos, distribuída, *multithread*, dinâmica, portátil, robusta, segura e de alto desempenho.

Na linguagem de programação Java, todo o código fonte é escrito em arquivos de texto simples que posteriormente são compilados não contém código que é nativo para o processador, mas sim para *bytecode*, que pode ser executado pela ferramenta Java Launcher utilizando uma instância da JVM.

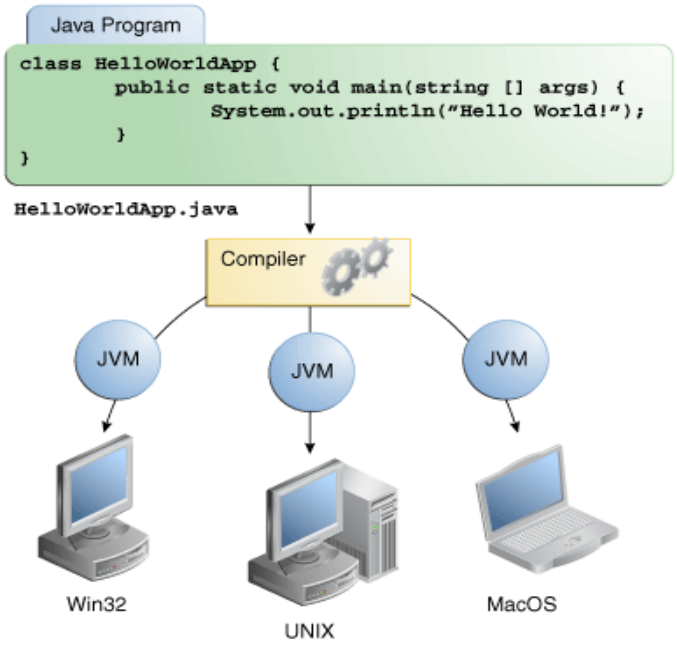
Figura 9: Processo de compilação de um arquivo fonte Java em *bytecode*



Fonte: The Java™ Tutorials

Como a JVM está disponível em vários sistemas operacionais, isso permite que os mesmos arquivos compiladores para *bytecode* sejam capazes de executar no Microsoft Windows, Solaris, Linux ou MacOS, sem a necessidade de recompilação do código fonte. Algumas máquinas virtuais podem executar etapas adicionais em tempo de execução para fornecer aumento de desempenho a uma aplicação, como identificar gargalos de desempenho e recompilar os *bytecodes* para código nativo.

Figura 10: A mesma aplicação Java executando em diferentes sistemas operacionais

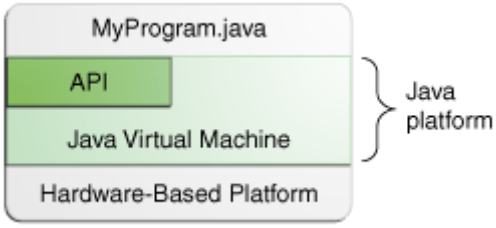


Fonte: *The Java™ Tutorials*

A plataforma Java difere da maioria das plataformas pois é somente um ambiente de software que é executado em cima de outras plataformas baseadas em hardware (uma plataforma é o ambiente de hardware ou software em que um programa é executado), e tem dois componentes principais: a JVM já descrita anteriormente e a *Java Application Programming Interface* (API).

A API é uma grande coleção de componentes de software que fornecem muitos recursos úteis. É agrupada em bibliotecas de classes e interfaces relacionadas, denominadas pacotes.

Figura 11: Como a API e a JVM isolam a aplicação



Fonte: *The Java™ Tutorials*

Como um ambiente independentemente de plataforma, o Java pode ser um pouco mais lento do que o código nativo, porém, os avanços no compilador nas tecnologias virtualização tem trazido desempenho próximo ao do código nativo sem ameaçar a portabilidade.

2.6.1. Apache Groovy

Apache Groovy é uma linguagem poderosa para a plataforma Java (opcionalmente tipada, dinâmica ou estática) que tem como objetivo melhorar a produtividade do desenvolvedor utilizando uma sintaxe concisa, familiar e simples de aprender.

Pode ser embarcada com qualquer programa Java, fornecendo assim recursos poderosos às aplicações, como *scripting*, criação de linguagens específicas de domínio, meta-programação em tempo de execução e programação funcional. A utilização do Apache Groovy com Java possibilita que os códigos sejam mesclados em tempo de execução, garantindo assim a extensibilidade.

Figura 12: Exemplo de como embarcar o Apache Groovy em um programa Java

```
GroovyShell shell = new GroovyShell();

shell.evaluate("println Olá Apache Groovy!'");
```

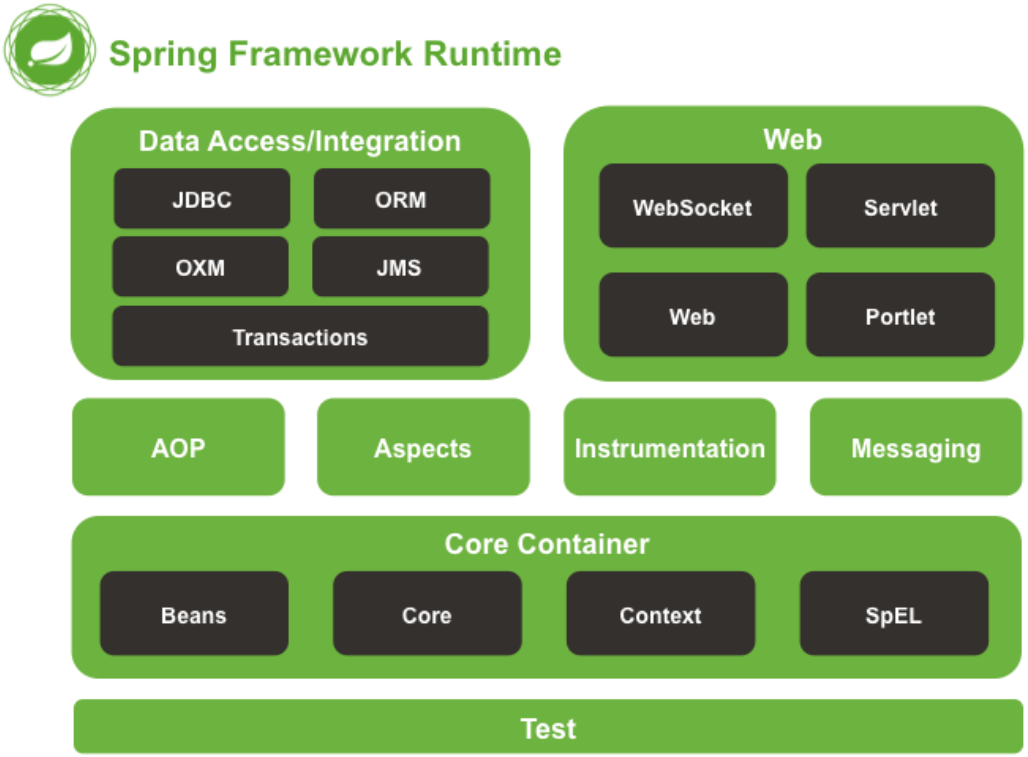
Fonte: Próprio autor

2.6.2. Spring Boot

O Spring Framework é uma solução leve desenvolvido para a plataforma Java que fornece suporte de infra-estrutura abrangente para o desenvolvimento de aplicações corporativas. Contudo, ele é modular, e permite utilizar os módulos somente seus módulos separadamente. Exemplo: pode-se utilizá-lo como contêiner de IoC (*inversion of control)*, com arquiteturas voltadas para a *Web*, integrando-o com o ORM (*object relational mapping*) Hibernate ou como uma camada de abstração do JDBC (*Java database connectivity*). Suporta o gerenciamento de transações declarativas, acesso remoto à lógica através de RMI (*remote method invocation*) ou serviços da *Web*, e várias opções para a persistência de dados. Oferece não só uma estrutura MVC (*model view controller*) com todas as funcionalidades, como também permite integrar o desenvolvimento AOP (*aspect oriented programming*) de forma transparente ao software.

O Spring é projetado para não ser intrusivo, o que significa que o código da lógica de domínio geralmente não tem dependências com o framework. Na camada de integração (como a camada de acesso a dados), existirão algumas dependências da tecnologia de acesso a dados e das bibliotecas Spring, no entanto, deve ser fácil isolar essas dependências do resto da base de código.

Figura 13: Módulos que compõem o Spring Framework



Fonte: Spring Framework (2017)

Seu núcleo é o contêiner de 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.

Fornece também uma implementação para programação orientada a aspectos compatível com AOP Alliance, permitindo definir, por exemplo, interceptores de métodos e pontos para desacoplar o código que implementa uma dada funcionalidada. Usando a funcionalidade de metadados a nível de código, é possível incorporar informações comportamentais ao código.

Com relação a acesso / integração de dados, há os módulos JDBC (que fornece uma camada de abstração JDBC, eliminando a necessidade de se fazer codificação JDBC tediosa e análise de códigos de erro específicos de um fornecedor de banco de dados), ORM (camadas de integração para APIs de mapeamento objeto-relacional populares, como por exemplo JPA, JDO e Hibernate), OXM, JMS e Transaction (gerenciamento de transações programáticas e declarativas para classes que implementam interfaces especiais).

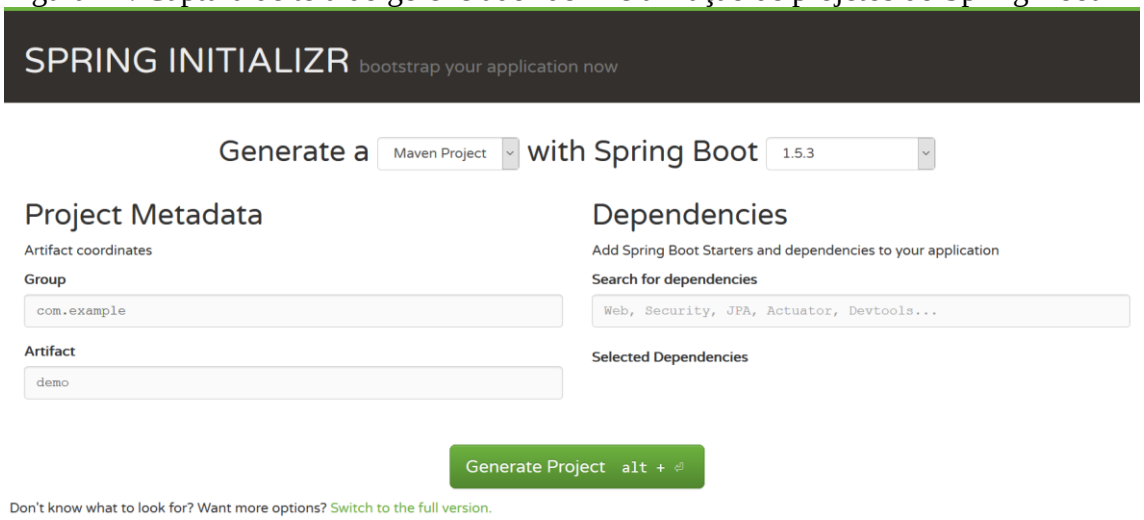
A camada da *web* é composta pelos módulos spring-web (fornece recursos básicos de integração dirigidos para a *web*, como a funcionalidade de *upload* de

arquivos *multipart*, inicialização do contêiner de IoC usando Servlet *listeners*), spring-webmvc (propõe a separação do código de modelo de domínio dos formulários da web), spring-websocket, and spring-webmvc-portlet.

Com tantos módulos, o Spring Framework fornece grande flexibilidade para configurar *beans* de várias formas utilizando XML e anotações, porém, com o aumento no número de recursos, a complexidade também aumentou e a configuração de aplicativos Spring tornou-se tediosa e propensa a erros. Para gerenciar a configuração de uma aplicação com Spring Framework, foi criado o Spring Boot, que é solução de "convenção sobre configuração" para a criação de aplicações autônomas.

Utilizando o spring-initializr, é possível criar um projeto estruturado selecionando uma ferramenta de automação de compilação (Maven, Gradle), a linguagem de programação (Java, Groovy), e o mais importate: quais módulos farão parte da aplicação.

Figura 14: Captura de tela do gerenciador de inicialização de projetos do Spring Boot



Fonte: Spring Framework (2017)

2.7. Considerações Finais

Neste capítulo foram apresentados os conhecimentos necessários e as tecnologias que serão utilizadas para a realização do projeto.

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.

3.2. Especificação de Usuários

Os usuários do *software* foram divididos entre usuários e desenvolvedores. Os usuários (que são representados por alunos e professores, como também desenvolvedores) podem acessar o *software* a fim de utilizá-lo na resolução de expressões algébricas, e visualizar de forma iterativa o resultado. Para os desenvolvedores, cabe a tarefa de melhorar as heurísticas previamente cadastradas, bem como estender o *software* com novas funcionalidades, sendo necessário possuir uma conta de usuário cadastrada, e efetuar o *login*.

3.3. Especificação de Requisitos

Os requisitos do software foram classificados como:

- Essencial: é o requisito sem o qual o sistema não entra em funcionamento. Requisitos essenciais são requisitos imprescindíveis, que têm que ser implementados impreterivelmente.
- Importante: é o requisito sem o qual o sistema entra em funcionamento, mas de forma não satisfatória. Requisitos importantes devem ser implementados, mas, se não forem, o sistema poderá ser implantado e usado mesmo assim.
- Desejável: é o requisito que não compromete as funcionalidades básicas do sistema, isto é, o sistema pode funcionar de forma satisfatória sem ele. Requisitos desejáveis podem ser deixados para versões posteriores do sistema, caso não haja tempo hábil para implementá-los na versão que está sendo especificada.

As técnicas de elicitação utilizadas para coleta de requisitos utilizada foram a entrevista, observação, ponto de vista e *brainstorm*.

3.3.1. Requisitos Funcionais

As tabelas 1 a 6 descrevem detalhes explicitamente funcionalidades e serviços que o software deve satisfazer:

Tabela 1: Requisito funcional da plataforma *web*

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 2: Requisito funcional do sistema web responsivo

Identificador	**Rq_02**
Nome	Sistema Web Responsivo
Descrição	Possibilidade de a aplicação rodar em qualquer plataforma utilizando um navegador web.
Técnicas de elicitação usadas	Entrevista
Classificação	Desejável

Fonte: Próprio autor

Tabela 3: Requisito funcional para resolução de expressão algébrica

Identificador	**Rq_03**
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 4: Requisito funcional para validação do usuário

Identificador	**Rq_04**
Nome	Validação de Usuário
Descrição	Será necessário o desenvolvedor validar o acesso para poder efetuar cadastros que venham a estender o *software*.
Técnicas de elicitação usadas	Entrevista
Classificação	Essencial

Fonte: Próprio autor

Tabela 5: Requisito funcional para gerenciamento de usuários

Identificador	**Rq_05**
Nome	Manter usuários
Descrição	Será necessário o desenvolvedor validar o acesso para poder efetuar inclusões, alterações e exclusões de usuários que estenderão o *software*.
Técnicas de elicitação usadas	Entrevista
Classificação	Importante

Fonte: Próprio autor

Tabela 6: Requisito funcional para gerenciamento de heurísticas

Identificador	**Rq_06**
Nome	Manter heurísticas
Descrição	Será necessário o desenvolvedor validar o acesso para poder efetuar inclusões, alterações e exclusões de heurísticas para resolução no *software*.
Técnicas de elicitação usadas	Entrevista
Classificação	Essencial

Fonte: Próprio autor

3.3.2. Requisitos Não Funcionais

A tabela 7 e a tabela 8 definem quais propriedades e restrições o software deve atender:

Tabela 7: Requisito não funcional de acesso à internet

Identificador	**Rq_09**
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

Tabela 8: Requisito não funcional necessidade de ao menos um desenvolvedor

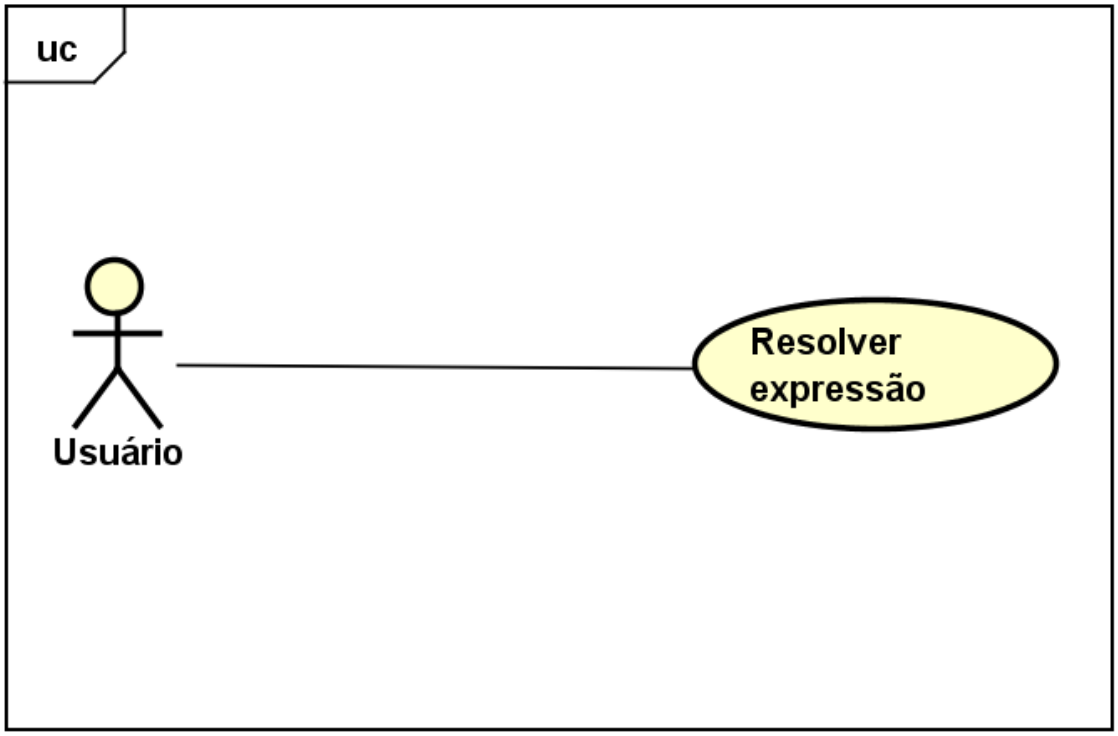
Identificador	**Rq_10**
Nome	É necessário ao menos um desenvolvedor no sistema
Descrição	Os desenvolvedores do sistema serão responsáveis pela manutenção das heurísticas e usuários do sistema.
Técnicas de elicitação usadas	Entrevista
Classificação	Essencial

Fonte: Próprio autor

3.3.3. Diagramas de Caso de Uso

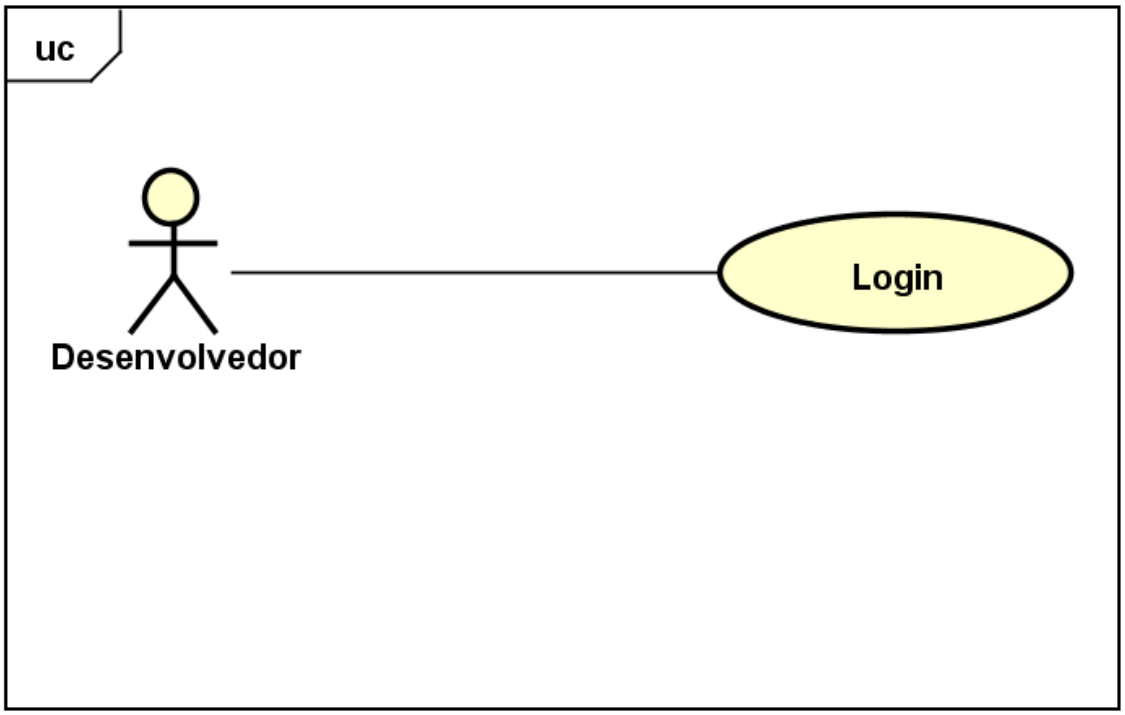
As figuras 15 a 18 contêm os diagramas de casos de uso que descrevem os cenários que mostram as funcionalidades do sistema do ponto de vista do usuário.

Figura 15: Diagrama de caso de uso para resolução de expressões algébricas



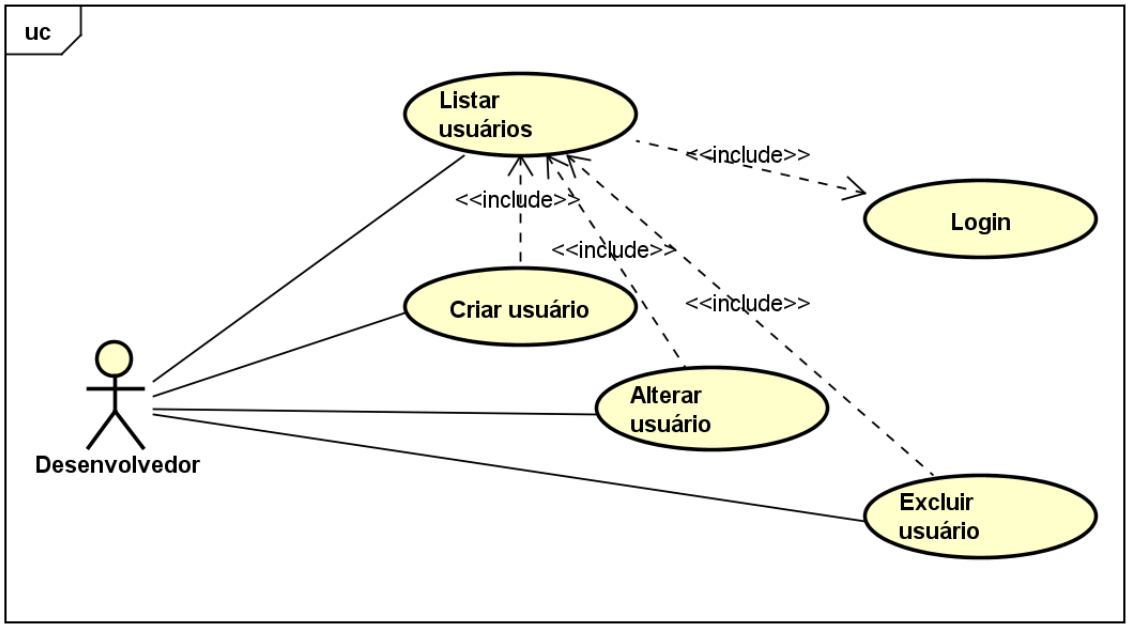
Fonte: Próprio autor

Figura 16: Diagrama de caso de uso de login



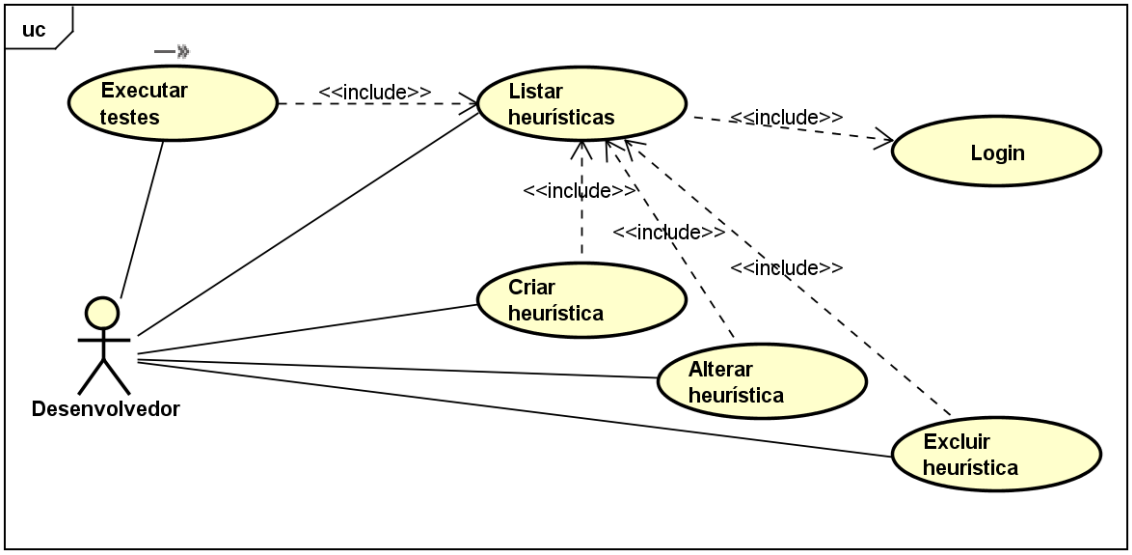
Fonte: Próprio autor

Figura 17: Diagrama de caso de uso para gerenciar usuários



Fonte: Próprio autor

Figura 18: Diagrama de caso de uso gerenciar heurísticas



Fonte: Próprio autor

A tabela 9 descreve quais atores, e quais suas respectivas responsabilidades quanto ao *software*.

Tabela 9: Atores e suas responsabilidades

Nome	**Descrição**	**Responsabilidades**
Usuário	Alunos, professores e desenvolvedores	Simplificar e resolver expressões.
Desenvolvedor	Desenvolvedores	Gerenciar usuários, heurísticas e testes de heurísticas.

Fonte: Próprio autor

As tabelas 10 a 19 contêm uma descrição completa dos casos de uso, suas relações com os atores, requisitos funcionais e não funcionais, bem como as condições em que se aplicam.

Tabela 10: Tabela de caso de uso para resolução de expressões

Identificador	UC_1
Nome	Resolver expressão
Atores	Usuário
Pré-condições	O usuário deve acessar o sistema
Pós-condições	As expressões são simplificadas e resolvidas de forma iterativa
Requisitos Funcionais	Rq_01, Rq_02, Rq_03
Requisitos Não-Funcionais	Rq_09

Fonte: Próprio autor

Tabela 11: Tabela de caso de uso para login

Identificador	UC_2
Nome	Login
Atores	Desenvolvedor
Pré-condições	O ator deve estar cadastrado no sistema

Pós-condições	O acesso do usuário é liberado ou rejeitado
Requisitos Funcionais	Rq_01, Rq_02, Rq_04
Requisitos Não-Funcionais	Rq_09, Rq_10

Fonte: Próprio autor

Tabela 12: Tabela de caso de uso para listagem de usuários

Identificador	UC_3
Nome	Listar usuários
Atores	Desenvolvedor
Pré-condições	O ator deve estar cadastrado no sistema
Pós-condições	É exibida a lista de usuários cadastrados no sistema
Requisitos Funcionais	Rq_01, Rq_02, Rq_05
Requisitos Não-Funcionais	Rq_09, Rq_10

Fonte: Próprio autor

Tabela 13: Tabela de caso de uso para inclusão de usuários

Identificador	UC_4
Nome	Criar usuário
Atores	Desenvolvedor
Pré-condições	O ator deve estar cadastrado no sistema, e acessar a lista de usuários cadastrados
Pós-condições	Um novo usuário é cadastrado no sistema caso não haja nenhum problema de validação de dados.
Requisitos Funcionais	Rq_01, Rq_02, Rq_05
Requisitos Não-Funcionais	Rq_09, Rq_10

Fonte: Próprio autor

Tabela 14: Tabela de caso de uso para alteração de usuários

Identificador	UC_5
Nome	Alterar usuário
Atores	Desenvolvedor
Pré-condições	O ator deve estar cadastrado no sistema, a lista de usuários deve ser acessada, e um usuário deve ser selecionado.
Pós-condições	Os dados do usuário são alterados caso não haja nenhum problema de validação de dados.
Requisitos Funcionais	Rq_01, Rq_02, Rq_05
Requisitos Não-Funcionais	Rq_09, Rq_10

Fonte: Próprio autor

Tabela 15: Tabela de caso de uso para exclusão de usuários

Identificador	UC_6
Nome	Excluir usuário
Atores	Desenvolvedor
Pré-condições	O ator deve estar cadastrado no sistema, a lista de usuários deve ser acessada, e um usuário deve ser selecionado.
Pós-condições	Haverá a exclusão do usuário
Requisitos Funcionais	Rq_01, Rq_02, Rq_05
Requisitos Não-Funcionais	Rq_09, Rq_10

Fonte: Próprio autor

Tabela 16: Tabela de caso de uso para listagem de heurísticas

Identificador	UC_7
Nome	Listar heurísticas
Atores	Desenvolvedor
Pré-condições	O ator deve estar cadastrado no sistema
Pós-condições	É exibida a lista de heurísticas cadastradas no sistema
Requisitos Funcionais	Rq_01, Rq_02, Rq_06
Requisitos Não-Funcionais	Rq_09, Rq_10

Fonte: Próprio autor

Tabela 17: Tabela de caso de uso para inclusão de heurísticas

Identificador	UC_8
Nome	Criar heurística
Atores	Desenvolvedor
Pré-condições	O ator deve estar cadastrado no sistema, e acessar a lista de heurísticas cadastradas
Pós-condições	Uma nova heurística é cadastrada no sistema caso não haja nenhum problema de validação de dados.
Requisitos Funcionais	Rq_01, Rq_02, Rq_06
Requisitos Não-Funcionais	Rq_09, Rq_10

Fonte: Próprio autor

Tabela 18: Tabela de caso de uso para alteração de heurísticas

Identificador	UC_9
Nome	Alterar heurística
Atores	Desenvolvedor
Pré-condições	O ator deve estar cadastrado no sistema, a lista de heurísticas deve ser acessada, e uma heurística deve ser selecionada.
Pós-condições	Os dados da heurística são alterados caso não haja nenhum problema de validação de dados.
Requisitos Funcionais	Rq_01, Rq_02, Rq_06
Requisitos Não-Funcionais	Rq_09, Rq_10

Fonte: Próprio autor

Tabela 19: Tabela de caso de uso para exclusão de heurísticas

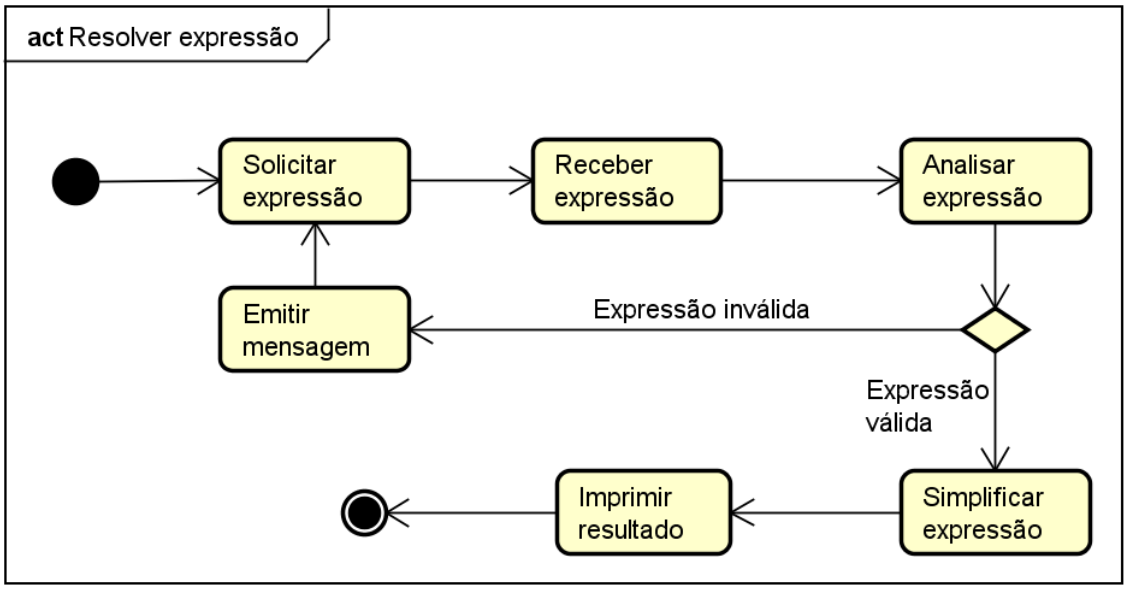
Identificador	UC_10
Nome	Excluir heurística
Atores	Desenvolvedor
Pré-condições	O ator deve estar cadastrado no sistema
Pós-condições	Haverá a exclusão da heurística
Requisitos Funcionais	Rq_01, Rq_02, Rq_06
Requisitos Não-Funcionais	Rq_09, Rq_10

Fonte: Próprio autor

3.3.4. Diagramas de Atividade

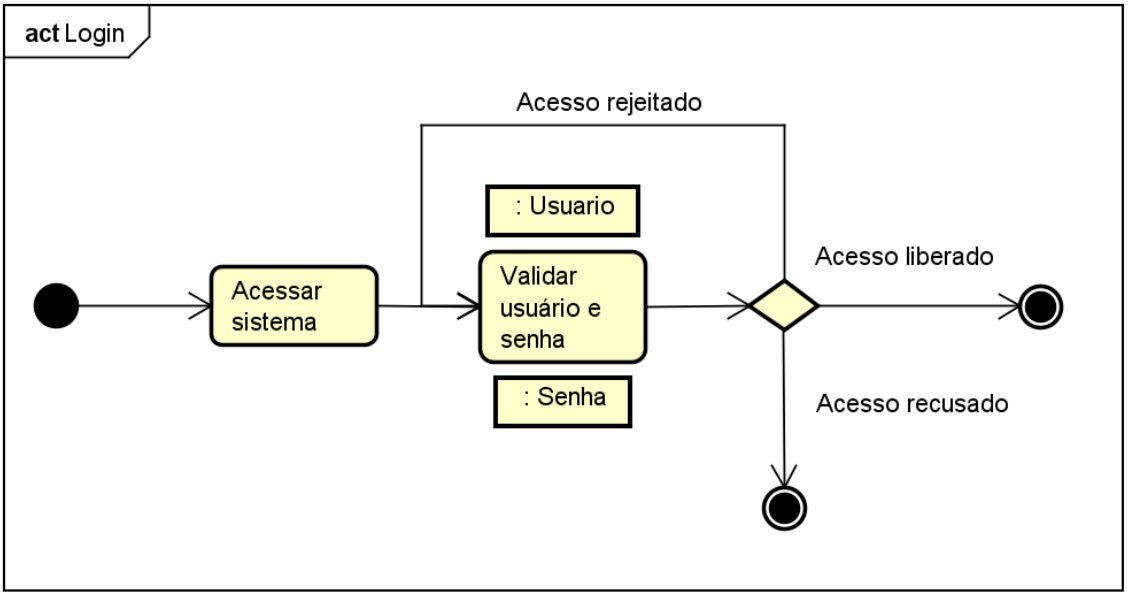
As figuras 19 a 28 descrevem os diagramas de atividade que representam os fluxos de controle de uma atividade para a outra tomando como base os casos de uso.

Figura 19: Diagrama de atividade para resolver expressões



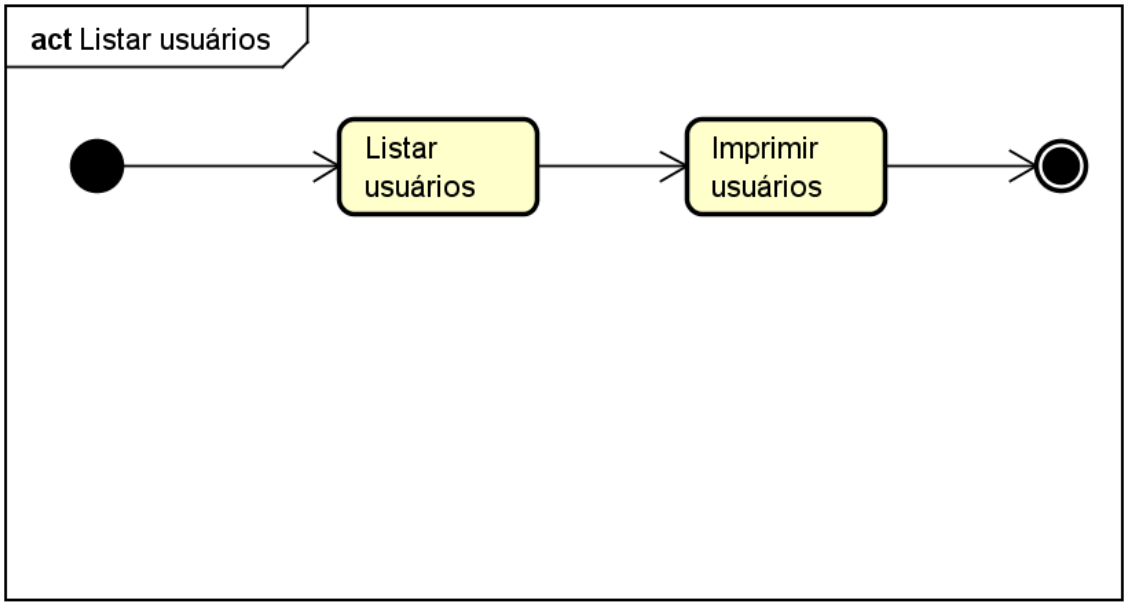
Fonte: Próprio autor

Figura 20: Diagrama de atividade de login



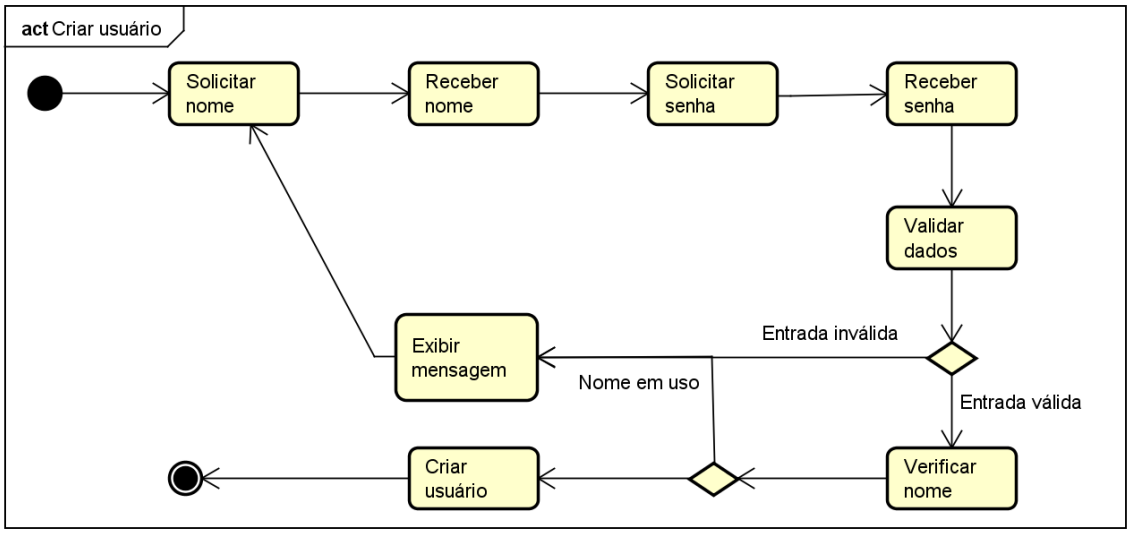
Fonte: Próprio autor

Figura 21: Diagrama de atividade para listagem de usuários



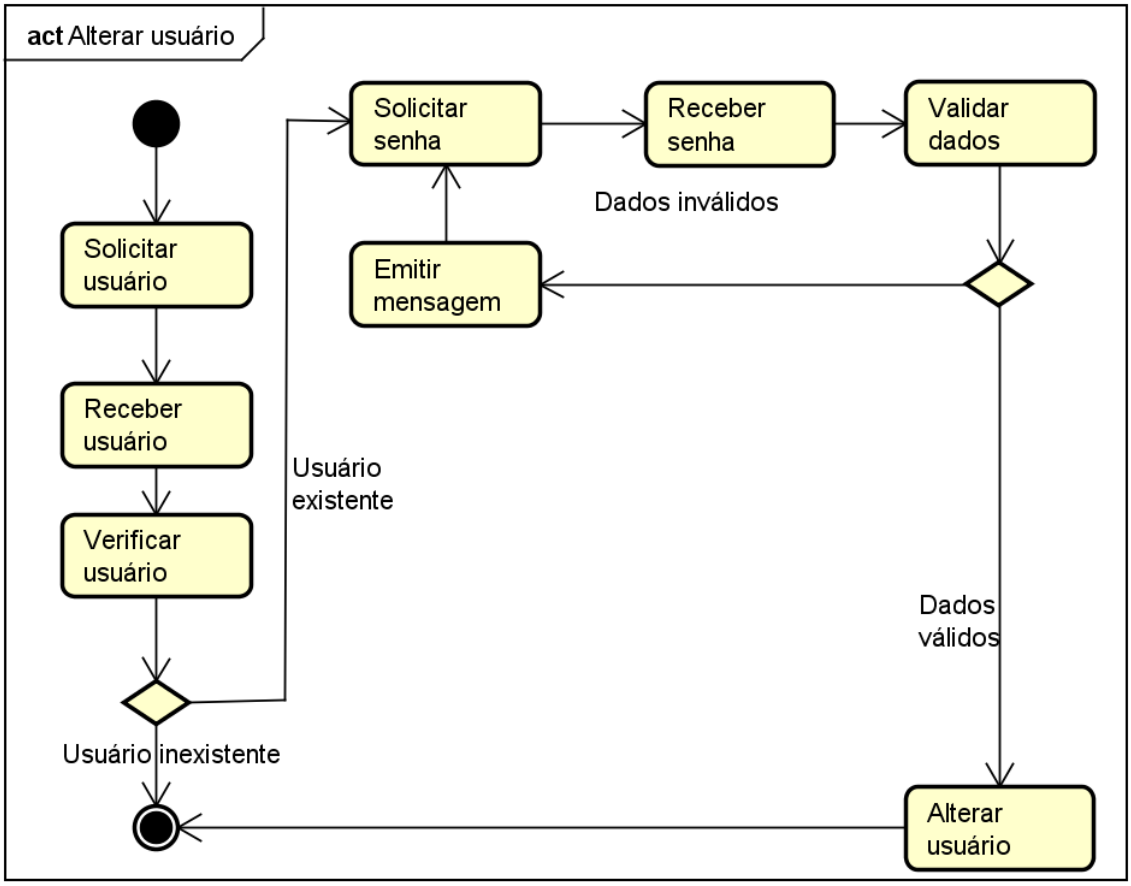
Fonte: Próprio autor

Figura 22: Digrama de atividade para inclusão de usuários



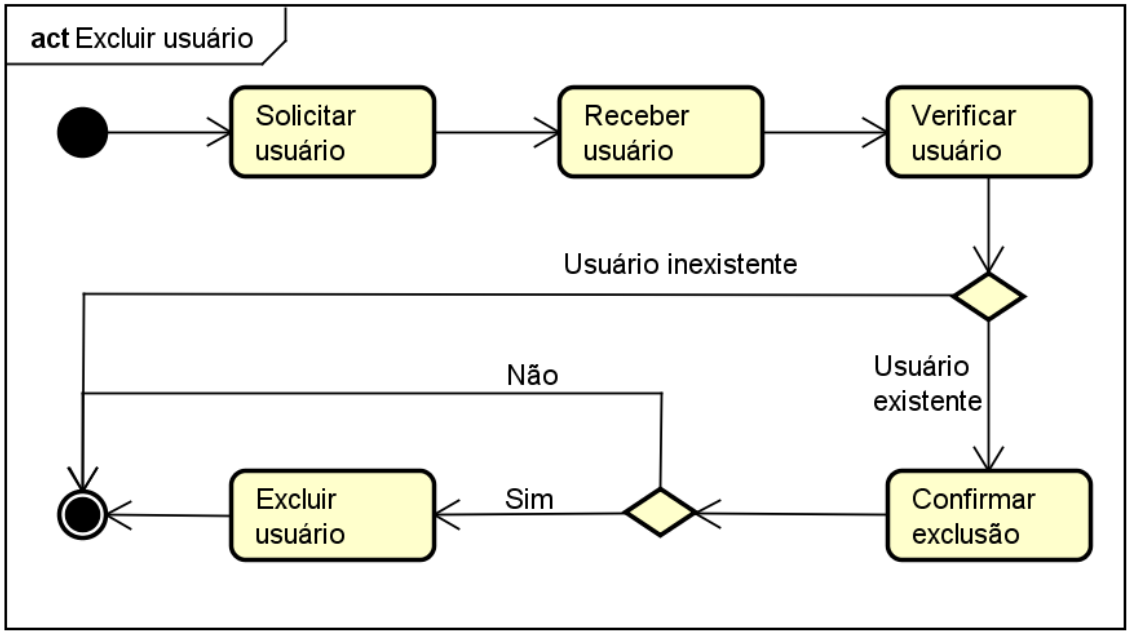
Fonte: Próprio autor

Figura 23: Digrama de atividade para alteração de usuários



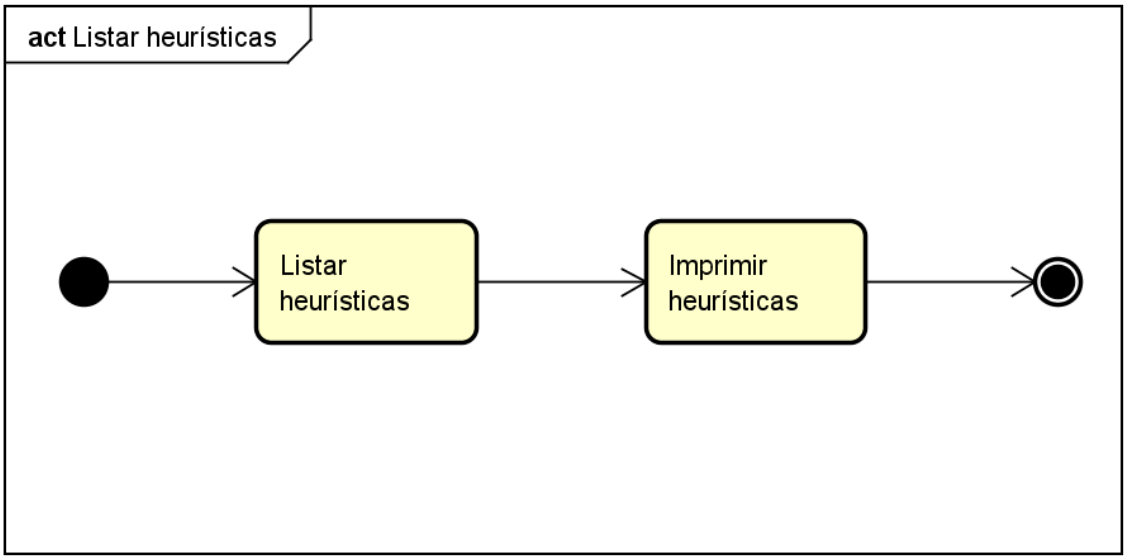
Fonte: Próprio autor

Figura 24: Digrama de atividade para exclusão de usuários



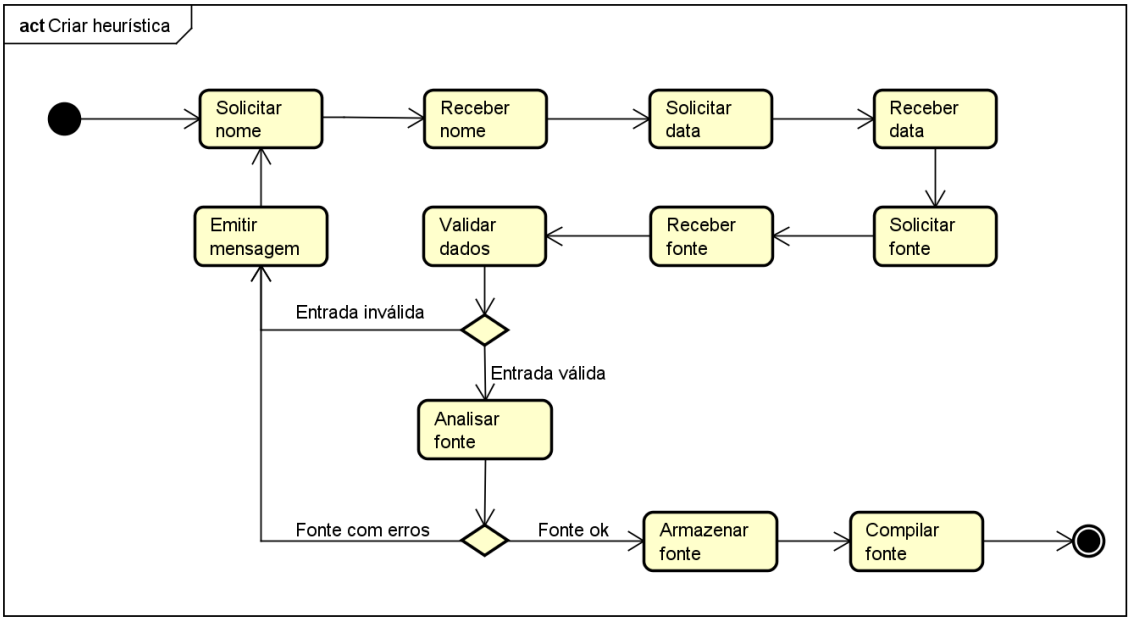
Fonte: Próprio autor

Figura 25: Digrama de atividade para listagem de heurísticas



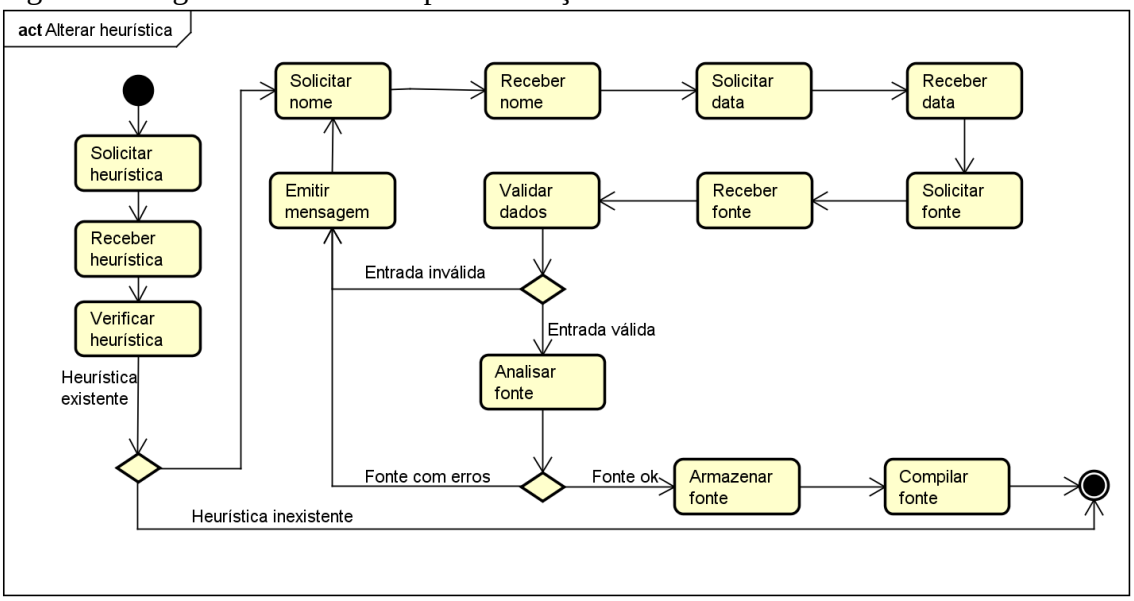
Fonte: Próprio autor

Figura 26: Digrama de atividade para inclusão de heurísticas



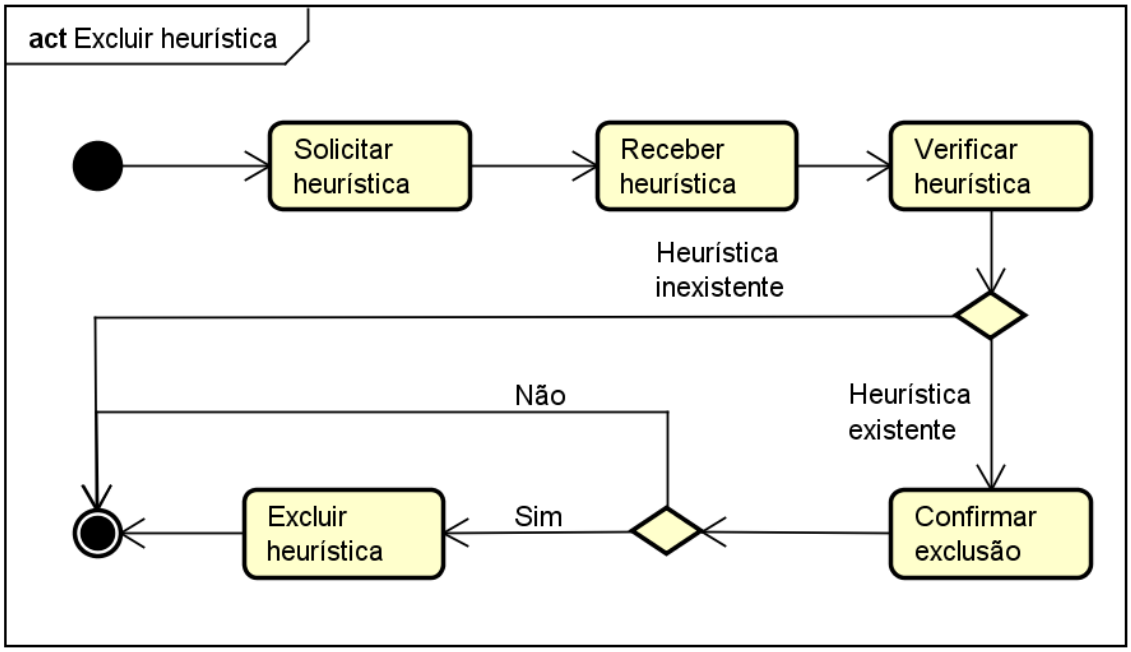
Fonte: Próprio autor

Figura 27: Digrama de atividade para alteração de heurísticas



Fonte: Próprio autor

Figura 28: Diagrama de atividade para exclusão de heurísticas



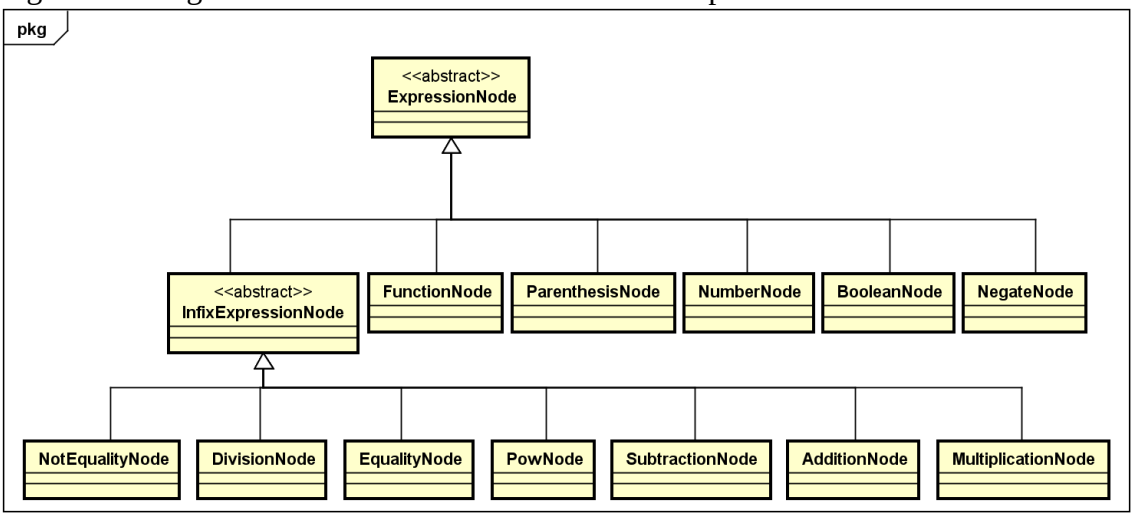
Fonte: Próprio autor

3.4. Artefatos de Análise

3.4.1. Diagramas de Classe Conceitual

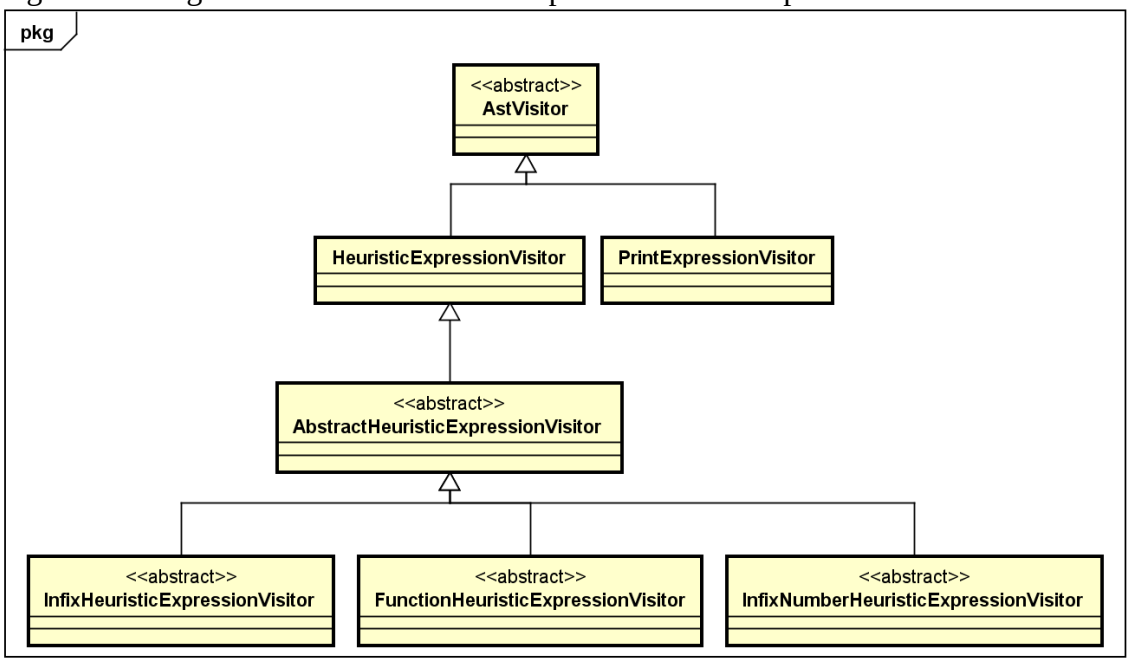
As figuras 29 a 32 demonstram uma representação conceitual dos diagramas de classe base.

Figura 29: Diagrama de classes conceitual de nós de expressão



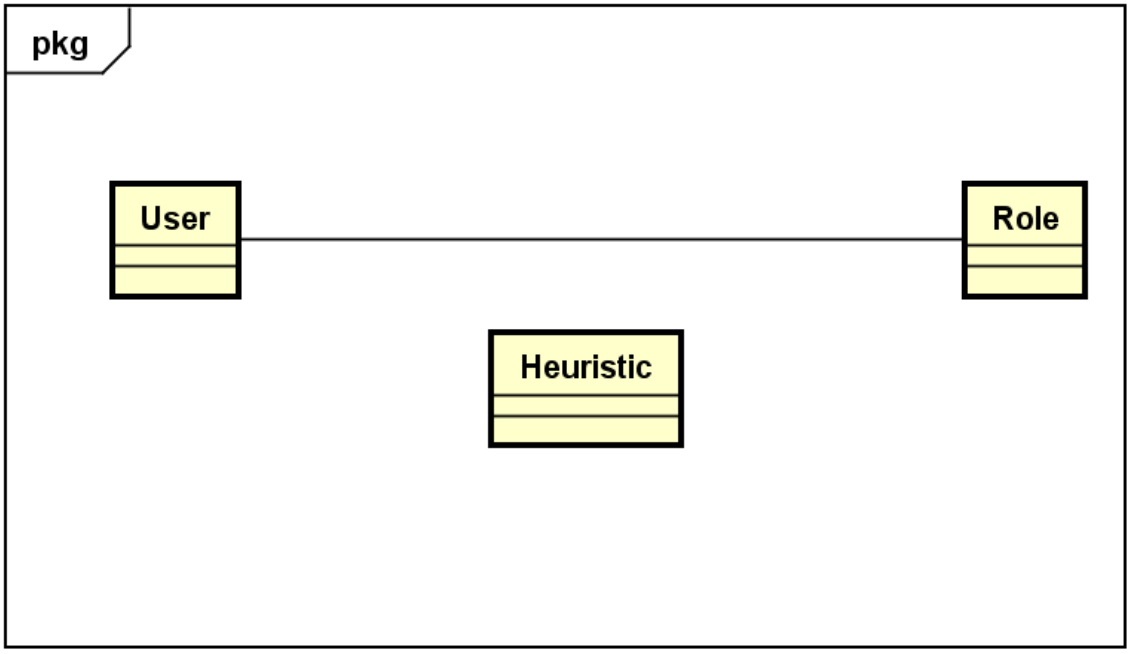
Fonte: Próprio autor

Figura 30: Diagrama de classes conceitual para *visitors* de expressões



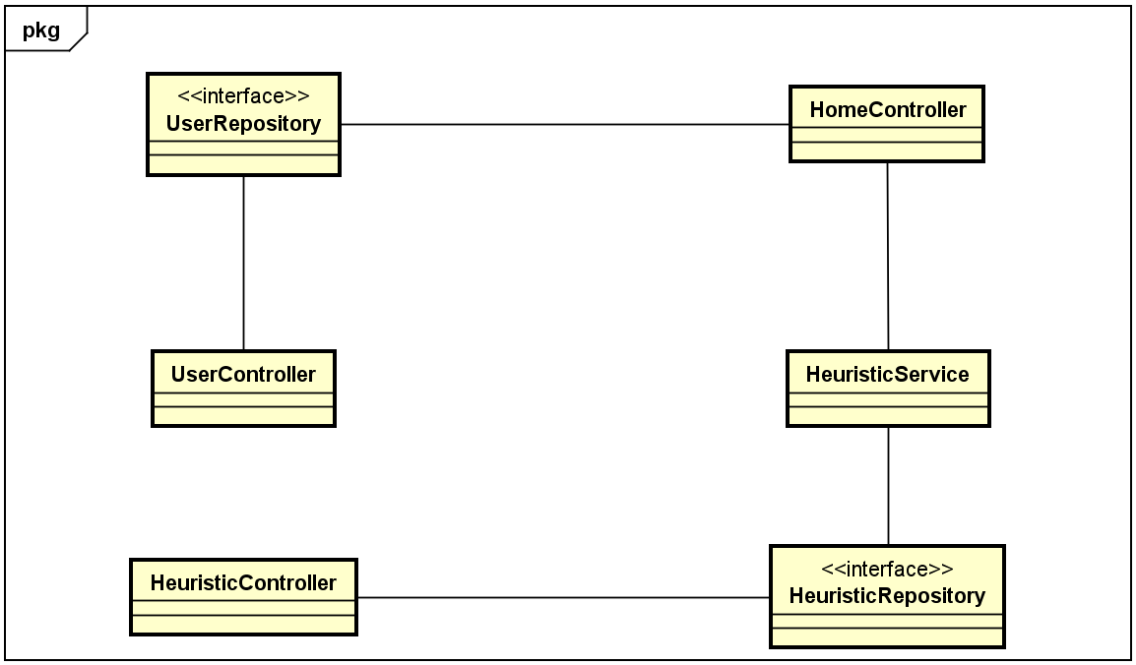
Fonte: Próprio autor

Figura 31: Diagrama de classes conceitual das entidades de domínio



Fonte: Próprio autor

Figura 32: Diagrama de classes conceitual dos controladores e repositórios

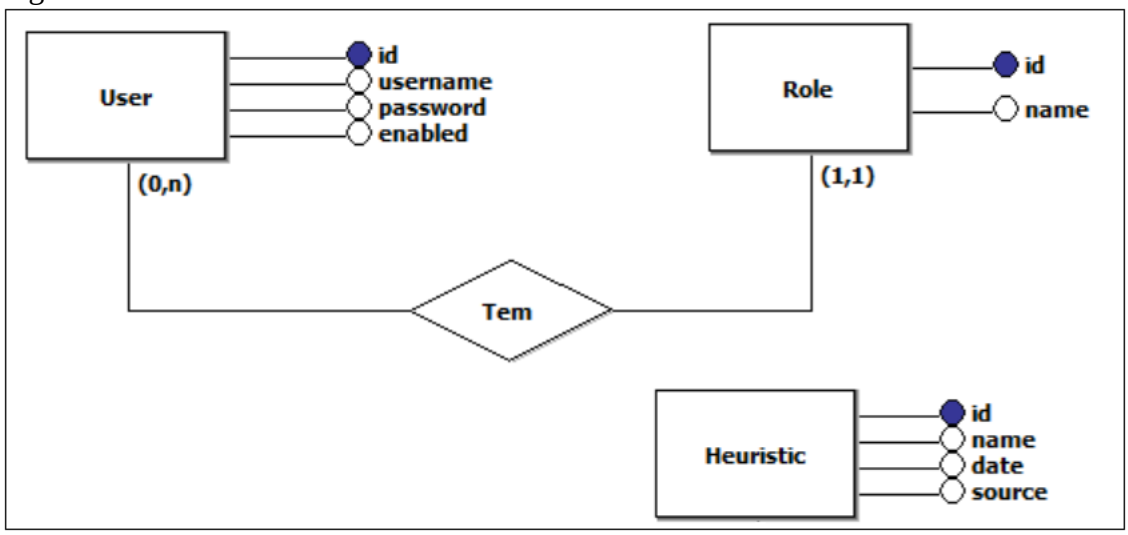


Fonte: Próprio autor

3.4.2. Modelo DER

A figura 33 demonstra o modelo DER (diagrama entidade-relacionamento) das entidades e relacionamentos que o *software* possui (o gerenciamento de usuários e funções não está diretamente vinculado com o gerenciamento de heurísticas e testes de heurística).

Figura 33: Modelo DER



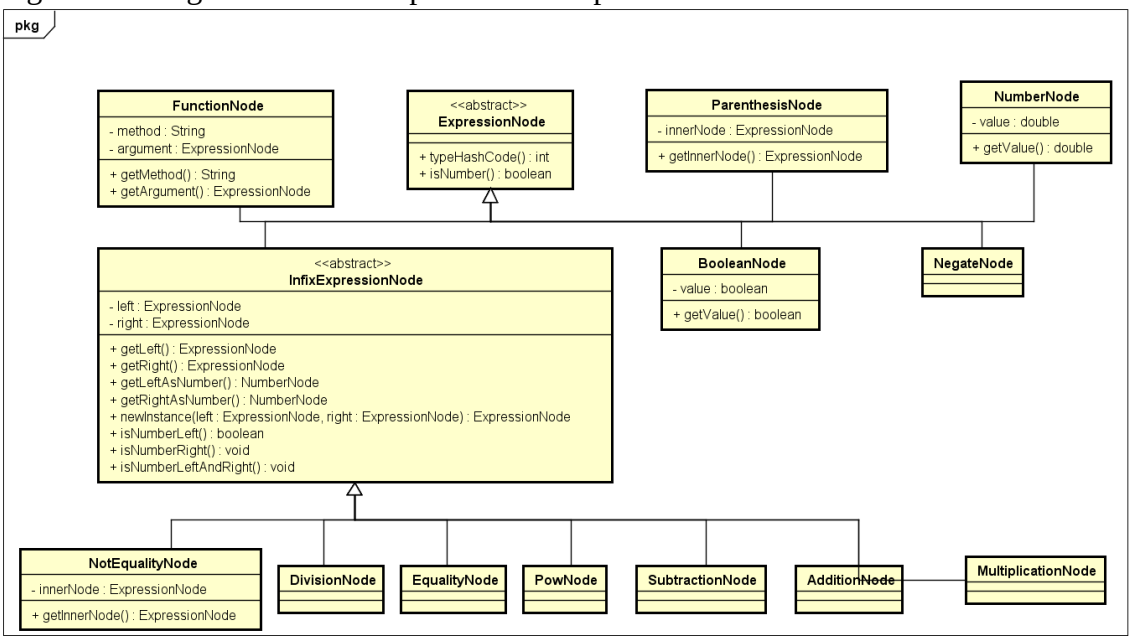
Fonte: Próprio autor

3.5. Artefatos de Projeto

3.5.1. Diagramas de classe

Os diagramas de classe da figura 34 representam a estrutura em árvore de nós de expressão.

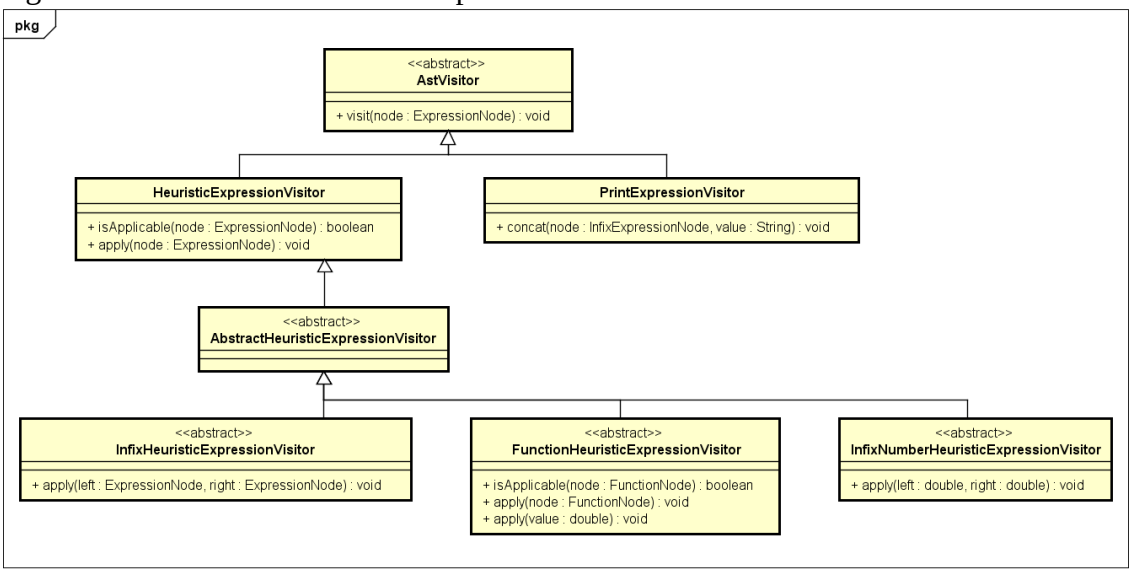
Figura 34: Diagrama de classes para nós de expressão



Fonte: Próprio autor

O diagrama de classe da figura 35 representa a estrutura das classes para análise da árvore de nós de expressão.

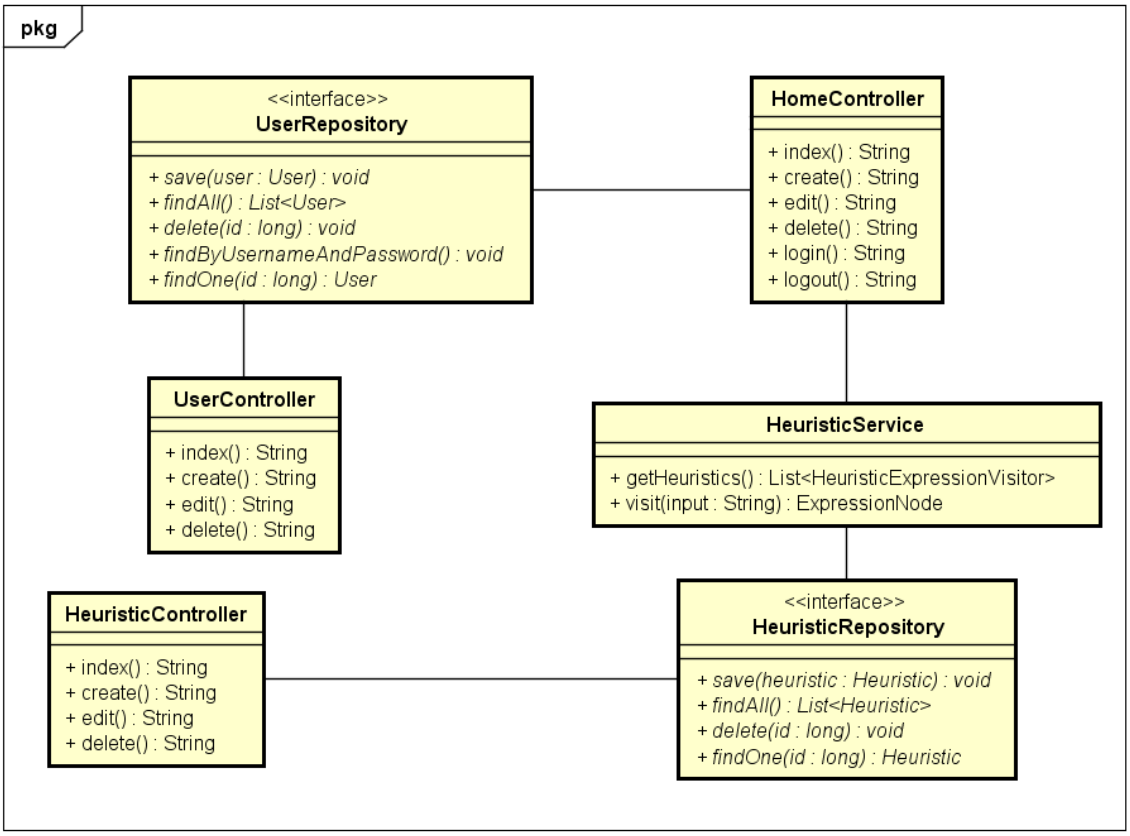
Figura 35: *Visitors* de árvores de expressão



Fonte: Próprio autor

O diagrama de classe da figura 36 representa a estrutura da aplicação com os controladores, repositórios e classe de serviço.

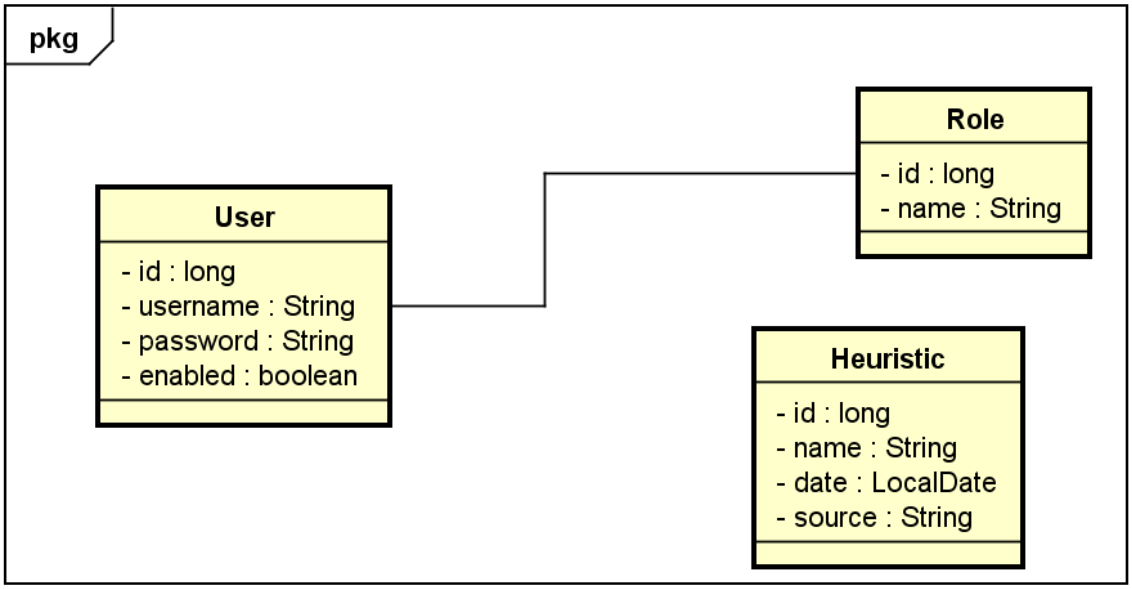
Figura 36: Diagrama de classes de controladores e repositórios



Fonte: Próprio autor

O diagrama da figura 37 descreve as entidades, quais seus relacionamentos e suas propriedades.

Figura 37: Diagrama de classes das entidades com seus atributos



Fonte: Próprio autor

3.5.2. Dicionário de entidades

As tabelas 20 a 22 descrevem de forma abrangente as entidades.

Tabela 20: 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 21: 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 22: 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

3.5.3. Dicionário de dados

As tabelas 23 a 33 contêm a definição dos elementos do dicionário de dados (que é um repositório de dados do *software*).

Tabela 23: Dicionários de dados do identificador da função

Identificador do Dado	**#Id**
Identificador de Entidade	#Role
Rótulo	id
Nome (s)	Identificador
Descrição	Identificador da função que o desenvolvedor representa
Requisito(s) Relacionado(s)	*(não se aplica)*

Tipo de Dado	Inteiro longo
Unidade de Medida	*(não se aplica)*
Precisão Numérica	*(não se aplica)*
Tamanho	*(não se aplica)*
Valor padrão	Nulo
Máscara de Edição	*(não se aplica)*

Fonte: Próprio autor

Tabela 24: Dicionários de dados do nome da função

Identificador do Dado	**#Name**
Identificador de Entidade	#Role
Rótulo	name
Nome (s)	Nome
Descrição	Nome da função que o desenvolvedor representa
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)*

Fonte: Próprio autor

Tabela 25: Dicionários de dados do identificador que relaciona com o usuário

Identificador do Dado	**#UserId**
Identificador de Entidade	#Role
Rótulo	User_id
Nome (s)	Identificador do usuário
Descrição	Identificador que relaciona a função com o usuário
Requisito(s) Relacionado(s)	*(não se aplica)*
Tipo de Dado	Inteiro longo
Unidade de Medida	*(não se aplica)*
Precisão Numérica	*(não se aplica)*
Tamanho	*(não se aplica)*
Valor padrão	Nulo
Máscara de Edição	*(não se aplica)*

Fonte: Próprio autor

Tabela 26: Dicionários de dados do identificador do usuário

Identificador do Dado	**#Id**
Identificador de Entidade	#User
Rótulo	id
Nome (s)	Identificador
Descrição	Identificador do usuário
Requisito(s) Relacionado(s)	*(não se aplica)*
Tipo de Dado	Inteiro longo
Unidade de Medida	*(não se aplica)*
Precisão Numérica	*(não se aplica)*
Tamanho	*(não se aplica)*
Valor padrão	Nulo
Máscara de Edição	*(não se aplica)*

Fonte: Próprio autor

Tabela 27: Dicionários de dados se o usuário está ou não habilitado

Identificador do Dado	**#Enabled**
Identificador de Entidade	#User
Rótulo	enabled
Nome (s)	Habilitado
Descrição	Se o usuário está ou não habilitado
Requisito(s) Relacionado(s)	*(não se aplica)*
Tipo de Dado	Booleano
Unidade de Medida	*(não se aplica)*
Precisão Numérica	*(não se aplica)*
Tamanho	*(não se aplica)*
Valor padrão	Nulo
Máscara de Edição	*(não se aplica)*

Fonte: Próprio autor

Tabela 28: Dicionários de dados da senha do usuário

Identificador do Dado	**#Password**
Identificador de Entidade	#User
Rótulo	password
Nome (s)	Senha
Descrição	Senha do usuário
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)*

Fonte: Próprio autor

Tabela 29: Dicionários de dados do nome de usuário

Identificador do Dado	**#Username**
Identificador de Entidade	#User
Rótulo	username
Nome (s)	Nome
Descrição	Nome do usuário
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)*

Fonte: Próprio autor

Tabela 30: Dicionários de dados do identificador da heurística

Identificador do Dado	**#Id**
Identificador de Entidade	#Heuristic
Rótulo	id
Nome (s)	Identificador
Descrição	Identificador da heurística
Requisito(s) Relacionado(s)	*(não se aplica)*
Tipo de Dado	Inteiro longo
Unidade de Medida	*(não se aplica)*
Precisão Numérica	*(não se aplica)*
Tamanho	*(não se aplica)*
Valor padrão	Nulo
Máscara de Edição	*(não se aplica)*

Fonte: Próprio autor

Tabela 31: Dicionários de dados da data da última alteração da heurística

Identificador do Dado	**#Date**
Identificador de Entidade	#Heuristic

Rótulo	date
Nome (s)	Data
Descrição	Data da última alteração da heurística
Requisito(s) Relacionado(s)	*(não se aplica)*
Tipo de Dado	Data
Unidade de Medida	*(não se aplica)*
Precisão Numérica	*(não se aplica)*
Tamanho	*(não se aplica)*
Valor padrão	Nulo
Máscara de Edição	DD/MM/AAAA

Fonte: Próprio autor

Tabela 32: Dicionários de dados do nome da heurística

Identificador do Dado	**#Name**
Identificador de Entidade	#Heuristic
Rótulo	name
Nome (s)	Nome
Descrição	Nome da heurística
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)*

Fonte: Próprio autor

Tabela 33: Dicionários de dados do código fonte da heurística

Identificador do Dado	**#Source**
Identificador de Entidade	#Heuristic
Rótulo	source
Nome (s)	Fonte
Descrição	Código fonte da heurística
Requisito(s) Relacionado(s)	*(não se aplica)*
Tipo de Dado	Texto longo
Unidade de Medida	*(não se aplica)*
Precisão Numérica	*(não se aplica)*
Tamanho	*(não se aplica)*

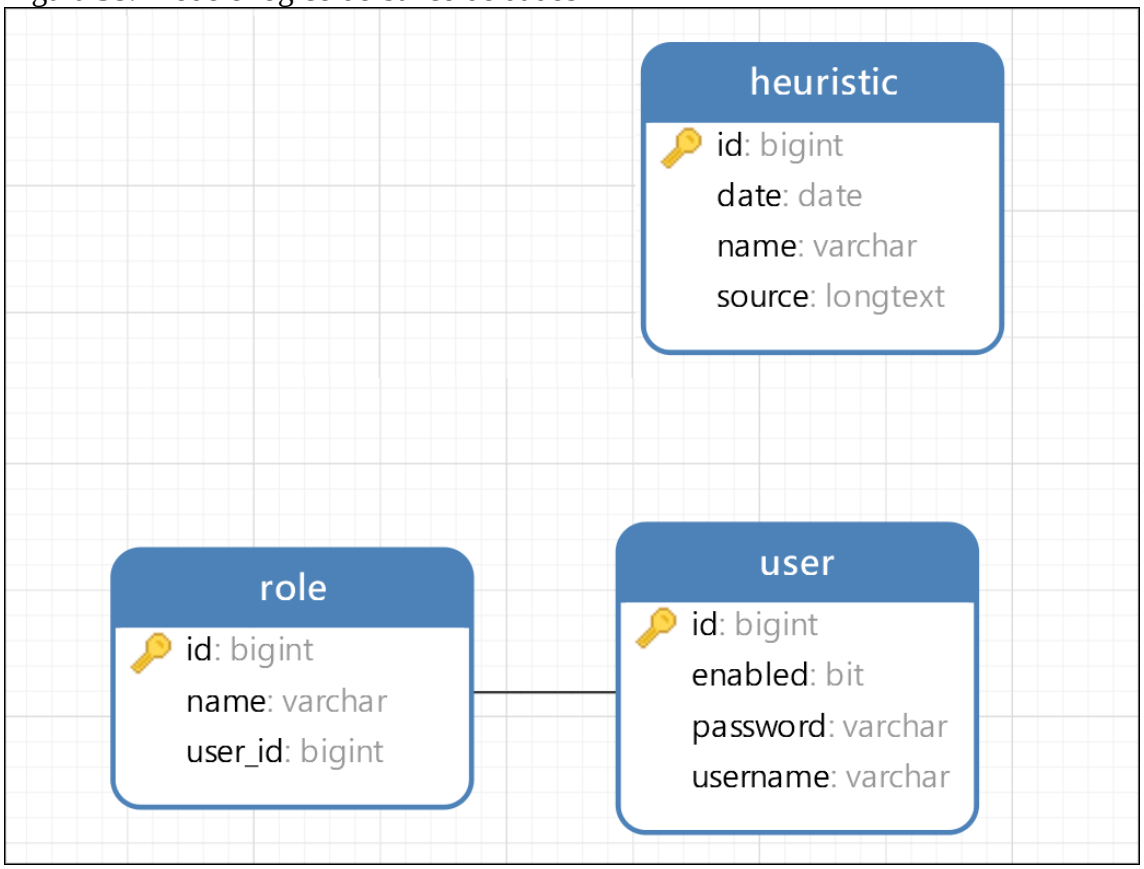
Valor padrão	Nulo
Máscara de Edição	*(não se aplica)*

Fonte: Próprio autor

3.5.1. Modelo Lógico do Banco de Dados

A figura 38 representa o modelo lógico do banco de dados utilizado no projeto.

Figura 38: Modelo lógico do banco de dados

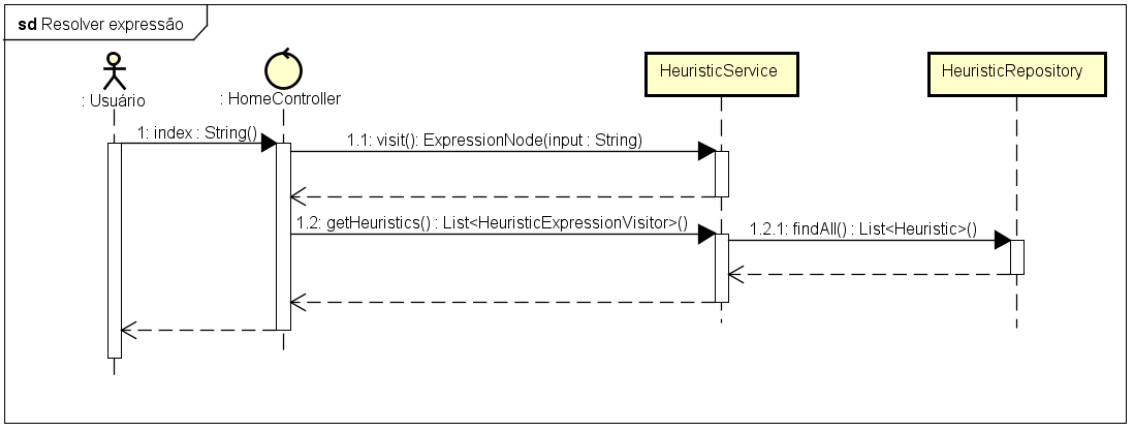


Fonte: Próprio autor

3.5.2. Diagramas de sequência

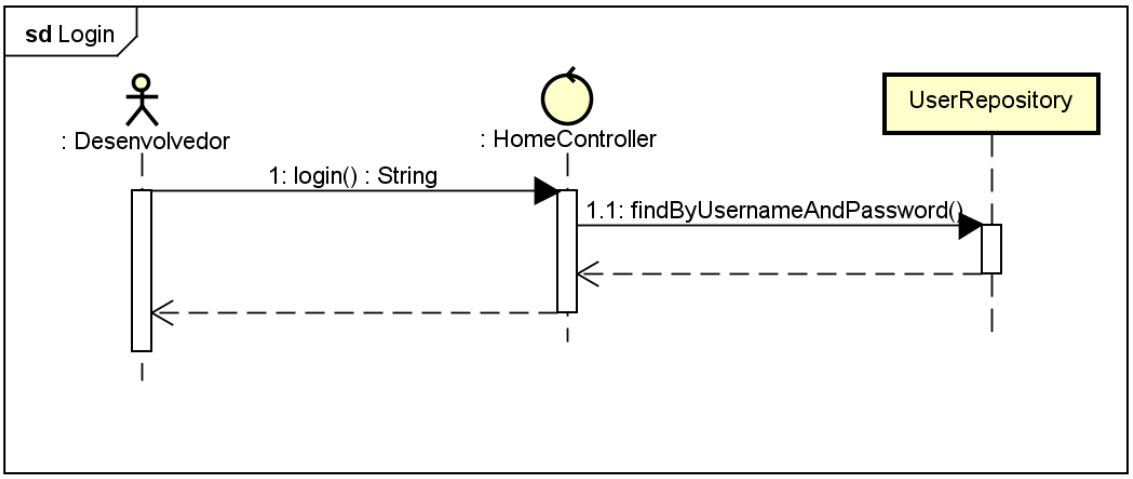
As figuras 39 a 48 descrevem os diagramas de sequência baseados nos casos de uso do *software*.

Figura 39: Diagrama de sequência para resolução de expressões



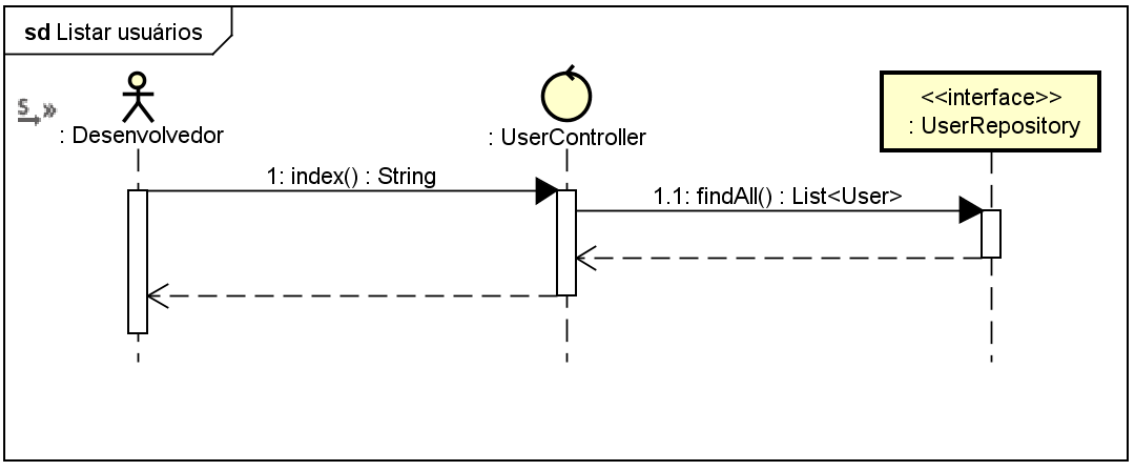
Fonte: Próprio autor

Figura 40: Diagrama de sequência de login



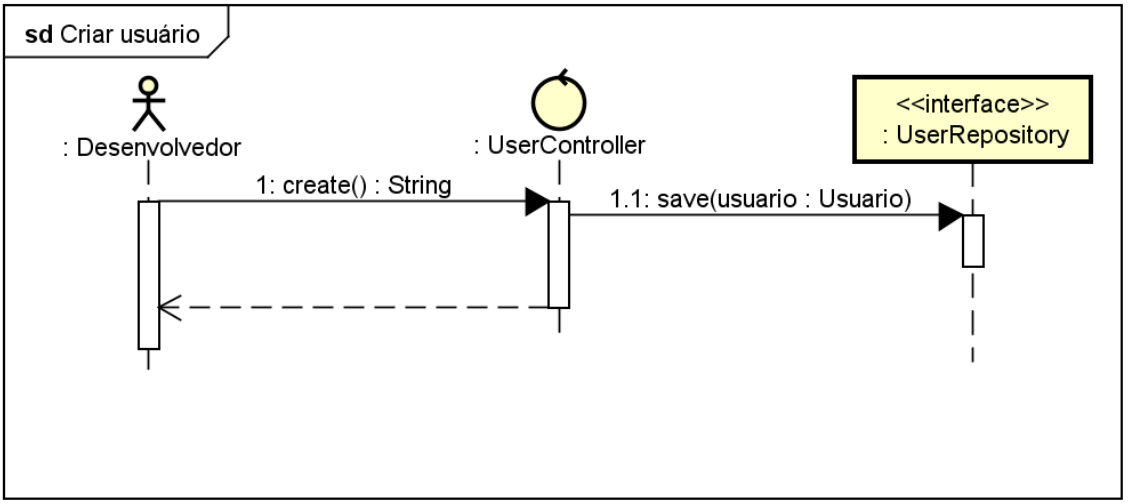
Fonte: Próprio autor

Figura 41: Diagrama de sequência de listagem de usuários



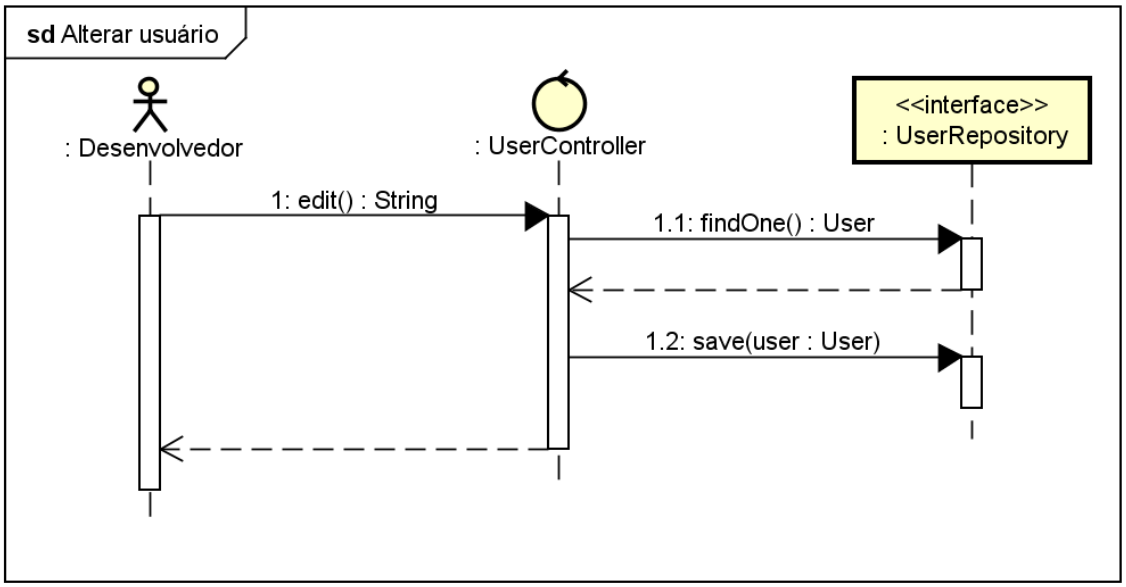
Fonte: Próprio autor

Figura 42: Diagrama de sequência de inclusão de usuários



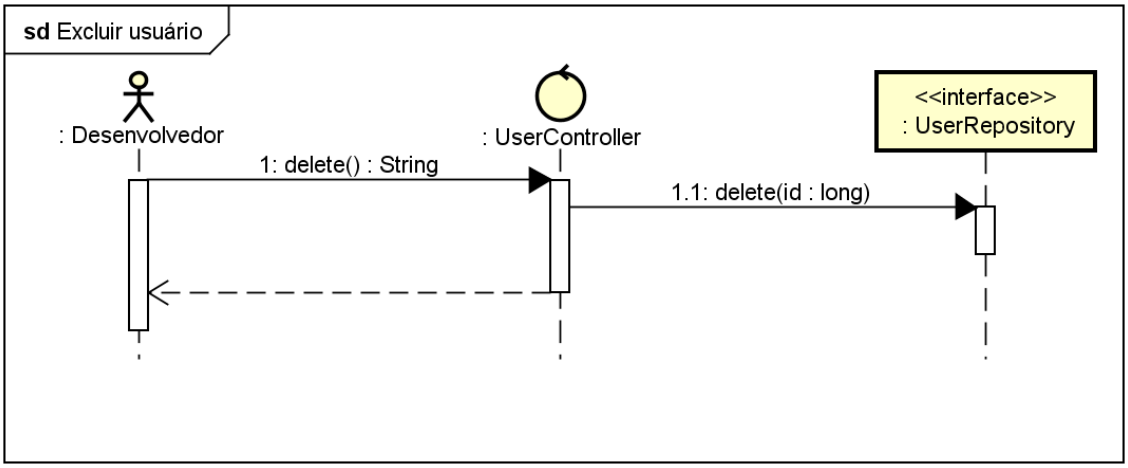
Fonte: Próprio autor

Figura 43: Diagrama de sequência de alteração de usuários



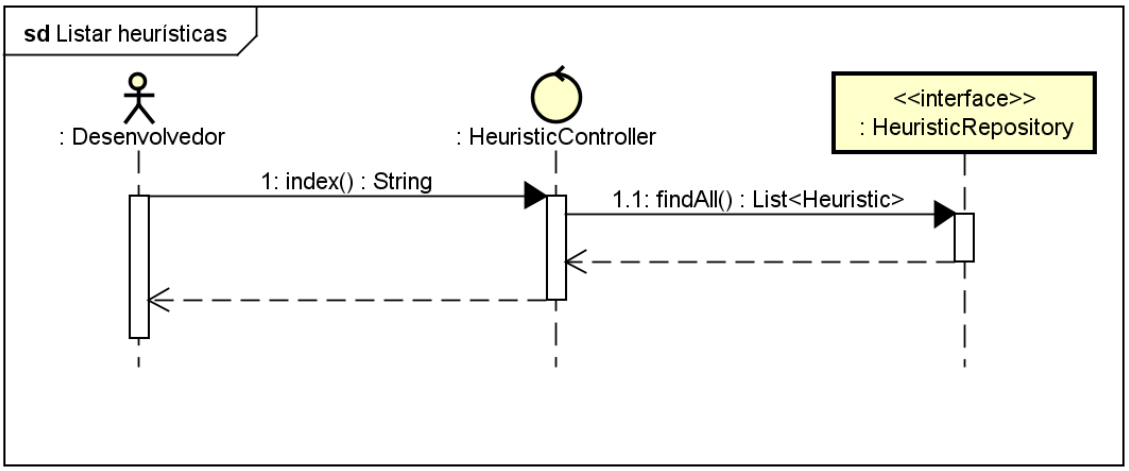
Fonte: Próprio autor

Figura 44: Diagrama de sequência de exclusão de usuários



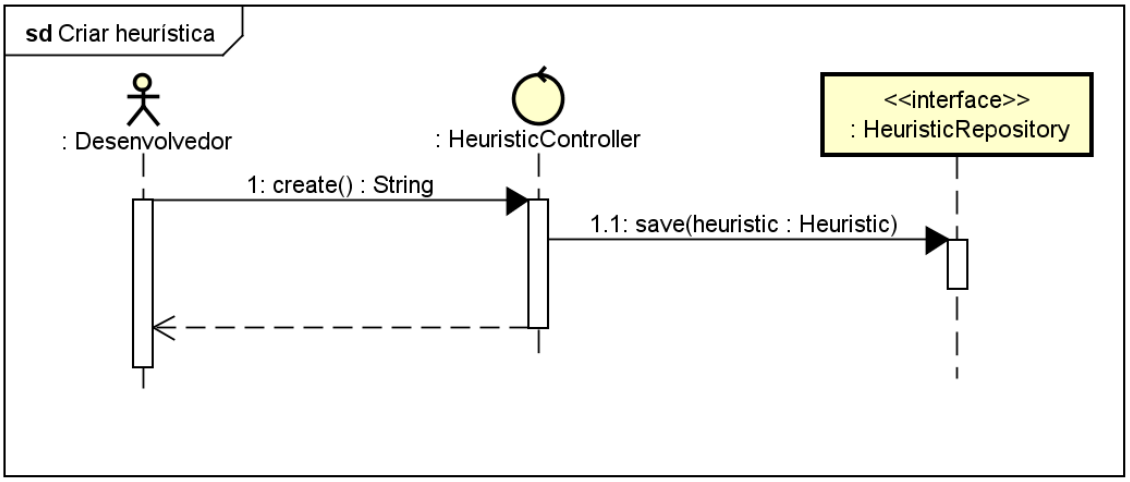
Fonte: Próprio autor

Figura 45: Diagrama de sequência de listagem de heurísticas



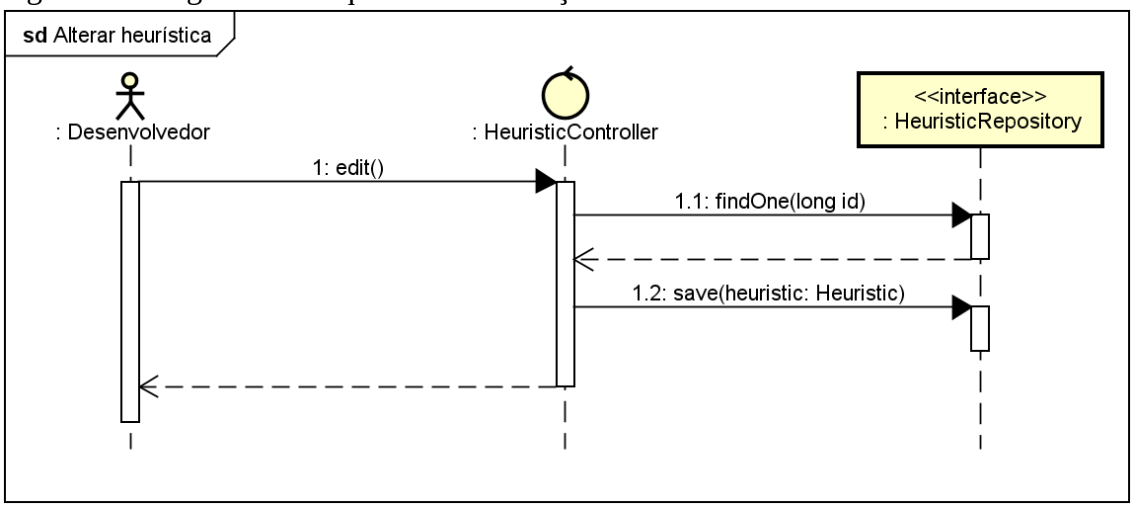
Fonte: Próprio autor

Figura 46: Diagrama de sequência de inclusão de heurísticas



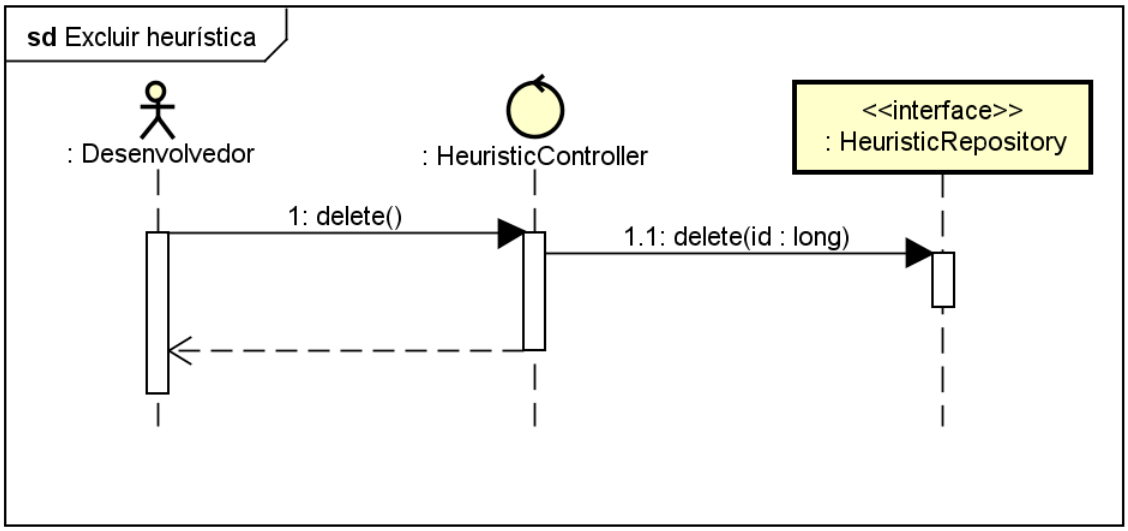
Fonte: Próprio autor

Figura 47: Diagrama de sequência de alteração de heurísticas



Fonte: Próprio autor

Figura 48: Diagrama de sequência de exclusão de heurísticas



Fonte: Próprio autor

3.6. Considerações Parciais

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

Capítulo 4

Desenvolvimento

4.1. Considerações Iniciais

Este capítulo descreve o desenvolvimento do software utilizando a linguagem Java, e como o projeto foi estruturado utilizando o design estratégico e tático do DDD, bem como o uso das ferramentas Gradle e Maven para gerenciamento de dependências e construção.

Realiza alguns comentários acerca da forma como o código-fonte foi implementado utilizando o ANTLR para a geração do analisador léxico/sintático para interpretação de expressões algébricas e geração da árvore sintática concreta. Descreve também como foi feita a estruturação da árvore sintática abstrata utilizando o padrão Composite, e descreve a criação das heurísticas utilizando o padrão Visitor.

Dispõe sobre especificidades dos módulos *core* e *web*, e como estes módulos interagem entre si, além de descrever a parte importante do *software* acerca da interoperabilidade entre as linguagens Java e Groovy.

Demonstra as telas principais do *software*, e finaliza tecendo comentários sobre a verificação e validação baseado nos testes realizados.

4.2. Ambiente de Desenvolvimento

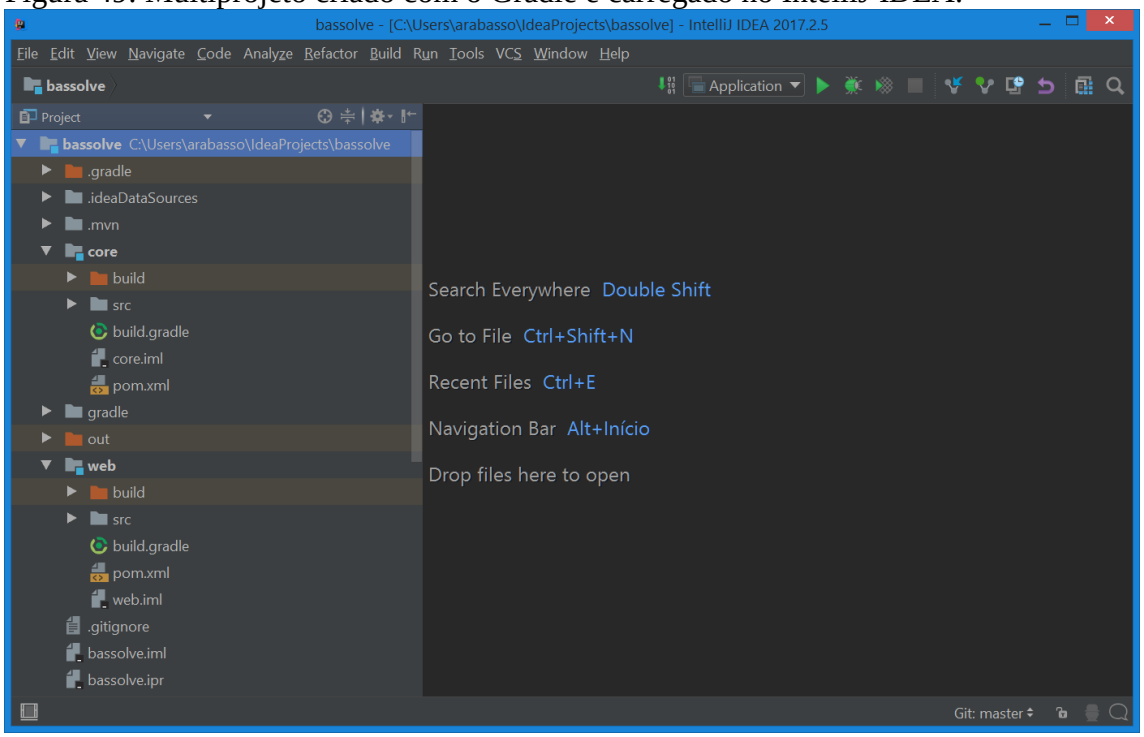
Para o desenvolvimento do software, foi utilizada a plataforma Java, cuja estrutura foi definida como um multiprojeto do Gradle e do Maven por conta do gerenciamento automatizado de dependências que ambas as ferramentas fornecem, execução de testes automatizados, e a construção da aplicação somente com uma instalação básica do JDK e o utilitário de linha de comando.

Tendo como base o design tático e estratégico do DDD, o projeto foi dividido em dois módulos:

1. *Core:* contêm o domínio da aplicação, onde foram definidas a gramática do ANTLR para análise de expressões e os pacotes com as classes que compõe o núcleo do software.
2. *Web:* implementação *web* da interface do usuário utilizando o SpringBoot para o gerenciamento das heurísticas que serão armazenadas em banco de dados, e que serão carregadas e interpretadas em tempo de execução por conta da interoperabilidade entre as linguagens Java e Groovy. Cabe ressaltar que este módulo foi criado utilizando o serviço de criação de aplicações do SpringBoot, onde foram selecionadas as dependências que compõe uma aplicação padrão com suporte a banco de dados (a figura 14 demonstra o serviço de criação de projetos do SpringBoot).

Para facilitar a escrita do código-fonte, foi utilizado o ambiente de desenvolvimento integrado IntelliJ IDEA versão 2017.2.5, cujo projeto foi criado com o auxílio da ferramenta Gradle. A figura 49 demonstra o multiprojeto carregado no IntelliJ IDEA.

Figura 49: Multiprojeto criado com o Gradle e carregado no IntelliJ IDEA.



Fonte: Próprio autor

4.3. Implementação

4.3.1. Módulo *core*

Tomando como base o design tático do DDD que prevê a separação do domínio em módulos, foi criada uma seção no módulo *core* para tratamento de questões acerca da análise de expressões algébricas, onde foi utilizada a ferramenta ANTLR para a geração do analisador léxico/sintático (a figura 50 demonstra um trecho da gramática utilizada).

Figura 50: Gramática para tratamento de expressões algébricas.

```
grammar Exp;

@header {
package sk.host.arabasso.bassolve.core.parser;
}

compileUnit
    : equation EOF
    ;

equation
    : left=expression op=(EQ|NEQ|LT|GT|LTE|GTE) right=expression    # equationExpression
    | expression                                                    # expressionNext
    ;

expression
    : op=(PLUS|MINUS) expression                                # unaryExpression
    | left=expression op=(PLUS|MINUS) right=multiplyingExpr     # plusMinusExpression
    | multiplyingExpr                                           # multiplyingExpressionNext
    ;

...
```

Fonte: Próprio autor.

Uma vez determinada a gramática e utilizando as ferramentas Gradle ou Maven, foi possível gerar o analisador léxico/sintático para análise de expressões algébricas e construção da árvore sintática. Entretanto, o analisador gerado pelo ANTLR retorna como resultado da análise uma CST (árvore sintática concreta) com grande densidade de informações que não necessárias para a implementação do software. Esta árvore sintática concreta fornecida pelo analisador gerado pelo ANTLR segue o padrão de projeto Composite, onde cada folha da árvore pode ser tratada utilizando o padrão de projeto Visitor.

Por questões de simplicidade, foi necessária a conversão da árvore sintática concreta em uma AST (árvore sintática abstrata), que conteria somente as informações relevantes para o domínio da aplicação. Assim como a CST, esta AST consiste de classes que implementam o padrão de projeto Composite, onde todas as classes têm como nó base a superclasse abstrata “ExpressionNode” (demonstrada na figura 51).

Figura 51: Superclasse de nós de expressão.

```
package sk.host.arabasso.bassolve.core.ast.node;

/**
 * Created by arabasso on 04/10/2016.
 *
 */
public abstract class ExpressionNode implements Cloneable {
    public int typeHashCode() {
        return getClass().hashCode();
    }
```

```
    public boolean isNumber() {
        return this instanceof NumberNode;
    }
}
```

Fonte: próprio autor.

Definidos os nós folhas, partiu-se para a implementação da classe que realizaria a conversão da CST em AST, sendo utilizado para tal o padrão de projeto Visitor, onde para cada método de análise de nós da CST, foi gerado um nó folha da AST. A figura 52 demonstra um trecho de código da referida classe para conversão de CST em uma AST.

Figura 52: Classe para conversão da CST em AST.

```
package sk.host.arabasso.bassolve.core.visitor;

import sk.host.arabasso.bassolve.core.ast.node.*;
import sk.host.arabasso.bassolve.core.parser.ExpBaseVisitor;
import sk.host.arabasso.bassolve.core.parser.ExpLexer;
import sk.host.arabasso.bassolve.core.parser.ExpParser;

/**
 * Created by arabasso on 04/10/2016.
 *
 */
public class BuildAstVisitor extends ExpBaseVisitor<ExpressionNode> {
    ...

    @Override
    public ExpressionNode visitPlusMinusExpression(ExpParser.PlusMinusExpressionContext ctx) {
        InfixExpressionNode node;

        ExpressionNode left = visit(ctx.left);
        ExpressionNode right = visit(ctx.right);

        switch (ctx.op.getType()) {
            case ExpLexer.PLUS:
                node = new AdditionNode(left, right);
                break;

            case ExpLexer.MINUS:
                node = new SubtractionNode(left, right);
                break;

            default:
                throw new UnsupportedOperationException();
        }

        return node;
    }

    @Override
    public ExpressionNode visitMultiplyingExpressionNext(ExpParser.MultiplyingExpressionNextContext ctx) {
        return visit(ctx.multiplyingExpr());
    }

    ...
}
```

Fonte: Próprio autor.

4.3.2. Módulo *web*

Definidas o núcleo base do módulo *core*, deu-se início o desenvolvimento do módulo *web*, cujo objetivo era implementar o design tático do DDD que consiste na camada de apresentação do *software*, repositórios para gerenciamento das entidades do domínio, classes de serviço, etc.

Sobre o tratamento de requisições e respostas, foi feita uma separação com a criação de um pacote de nome “controller” que conteria todos os controladores, onde para o desenvolvimento dos mesmos, utilizou-se o *framework* SpringMVC (parte integrante do SpringBoot), uma vez que este implementa o padrão MVC utilizando como princípio a separação dos controladores e ações que estes podem realizar, além de

tornar transparente a integração entre o modelo e a visualização. A figura 53 demonstra um trecho de código da implementação do controlador “Home”.

Figura 53: Trecho de código do controlador “Home”.

```
@Controller
public class HomeController {
    @GetMapping(value = {"/", "/index"})
    public String index(
            Model model) {

        model.addAttribute("form", new ExpressionForm("100 + 2 - 3 * 4 + 8 / 4 - 2 ^ 3"));

        return "index";
    }

    @Autowired
    HeuristicService heuristicService;

    ...
}
```

Fonte: Próprio autor.

A execução de um controlador consiste em receber uma requisição, encaminhá-la para a ação correta, processar e enviar os dados para uma visualização implementada em HTML (e que será interpretada pelo motor de visualização Thymeleaf), para finalmente, retornar o resultado para o usuário (utilizar um motor de visualização robusto permite por exemplo criar uma página principal que contêm a estrutura base de todas as páginas, onde para cada página é criada uma visualização simplificada para o tratamento de cada ação específica). A figura 54 demonstra a visualização “index”.

Figura 54: Visualização "index" utilizando o Thymeleaf.

```
<section class="content" layout:decorator="layout" layout:fragment="body"
         xmlns:th="http://www.thymeleaf.org"
         xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
    <!--/*@thymesVar id="form" type="sk.host.arabasso.bassolve.web.form.HeuristicForm"*/-->
    <form th:action="@{/index}" th:object="${form}" method="post">
        <ul th:if="${#fields.hasErrors('global')}">
            <li th:each="err : ${#fields.errors('global')}" th:text="${err}">Input is incorrect</li>
        </ul>

        <fieldset>
            <div class="grid">
                <div class="grid-1-5"><label th:for="*{value}"
th:text="#{expression.form.text}">Expression</label></div ><div class="grid-3-5"><input class="width-
1" type="text" th:field="*{value}" th:placeholder="#{expression.form.placeholder}"/></div><div
class="grid-1-5"><button class="width-1" type="submit"
th:text="#{expression.form.submit.text}">Solve</button></div>
            </div>
            <div th:if="${#fields.hasErrors('value')}" th:errors="*{value}"></div>
            <div id="expression" th:if="${form.hasValue()}">
                <!--/*@thymesVar id="exp" type="System.util.String"*/-->
                <p th:each="r : ${result}" th:text="|${r}|"></p>
            </div>
        </fieldset>
    </form>
</section>
```

Fonte: Próprio autor.

Quanto às entidades de domínio, foram criadas classes que foram mapeadas utilizando o *framework* para *Object Relacional Mapping* – ORM (mapeamento objeto relacional) Hibernate, possibilitando assim a integração com diversos bancos de dados (para o projeto, foi utilizado o banco de dados H2). A figura 55 demonstra como foi realizado o mapeamento da entidade “Heuristic”.

Figura 55: Mapeamento objeto relacional daclasse "Heuristic".

```
@Entity
@Table(name = "heuristic")
```

```
public class Heuristic implements Serializable {
    private static final long serialVersionUID = -6113123348770067062L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    private LocalDate date;
    @Lob
    public String source;
    ...
}
```

Fonte: Próprio autor.

A principal entidade é a classe “Heuristic”, onde é definida a estrutura de heurística com seu respectivo código-fonte escrito em linguagem Groovy (que é armazenado na propriedade “source”). Para a integração em tempo de execução entre o código-fonte armazenado em banco de dados com a máquina virtual Java, foi implementado um método “compile”, que retorna uma classe herdeira da superclasse “HeuristicExpressionVisitor” (utilizando para isso o *framework* de integração entre a linguagem Groovy e Java gerando *bytecode* Java em tempo de execução). Uma vez gerado o *bytecode* Java, é possível instanciar um objeto de heurística que será executado utilizando para tal, o método “getHeuristic”. A figura 56 demonstra como é feita a interoperabilidade entre Groovy e Java.

Figura 56: Interoperabilidade entre Groovy e Java.

```
public class Heuristic implements Serializable {
    ...

    public HeuristicExpressionVisitor getHeuristic() {
        Class clazz = compile();

        try {
            return (HeuristicExpressionVisitor) clazz.newInstance();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }

        return new HeuristicExpressionVisitor();
    }

    public Class compile() {
        GroovyClassLoader gcl = new GroovyClassLoader();

        return gcl.parseClass(source);
    }
}
```

Fonte: Próprio autor.

Para o gerenciamento em banco de dados das entidades, foi utilizado o padrão Repository do DDD, que tem por objetivo isolar os objetos ou entidades do domínio do código que acessa o banco de dados (permitindo dessa forma abstrair o armazenamento e consulta de uma ou mais entidades de domínio). A figura 57 contém um exemplo do padrão Repository para gerenciamento de usuários.

Figura 57: Repositório de usuários.

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
    @Query("SELECT CASE WHEN COUNT(u) > 0 THEN 'true' ELSE 'false' END FROM User u WHERE u.id <> ?1
and u.username = ?2")
    public Boolean existsByUsername(String username);
    @Query("SELECT CASE WHEN COUNT(u) > 0 THEN 'true' ELSE 'false' END FROM User u WHERE u.username =
?1")
    public Boolean existsByIdAndUsername(long id, String username);
```

```
}
```

Fonte: Próprio autor.

Esse gerenciamento baseado em repositórios foi fundamental para a criação do serviço de controle das heurísticas que foi implementado na classe “HeuristicService” do pacote “service”. Esta classe tem como função o carregamento de todas as heurísticas armazenadas no banco de dados utilizado o repositório de heurísticas, e integrá-las em tempo de execução usando o método “getHeuristic” (demonstrado na figura 58).

Figura 58: Serviço para carregamento de heurísticas.

```
@Component
@Scope("prototype")
public class HeuristicService {
    @Autowired
    public HeuristicRepository heuristicRepository;

    public List<HeuristicExpressionVisitor> getHeuristics(){
        List<Heuristic> list = heuristicRepository.findAll();

        return list.stream().map(Heuristic::getHeuristic).collect(Collectors.toList());
    }
}
```

Fonte: próprio autor.

Este serviço foi integrado ao controlador de requisições e repostas por meio da ação “index”, onde é feita a validação de erros na entrada, e realizada a simplificação dos termos da expressão algébrica por meio do método “visit”. Este método consiste em executar continuamente até que não haja mais como simplificar baseado no valor retornado pelo método “hashCode” (que calcula um valor que representa a árvore analisando todos os nós). Para cada iteração, é realizada a transformação da árvore em texto e o resultado é armazenado em uma lista para posterior exibição na visualização (o código da figura 59 descreve como este processo é realizado).

Figura 59: Processo iterativo de simplificação de expressões algébricas.

```
public List<String> visit(String value)
        throws NoSuchMethodException,
        IllegalAccessException,
        InvocationTargetException {
    PrintExpressionVisitor printExpressionVisitor = new PrintExpressionVisitor();

    List<String> list = new ArrayList<>();

    ExpressionNode ast = AstVisitor.visit(value);

    list.add(printExpressionVisitor.visit(ast));

    int previousHashCode = -1;
    int hashCode = ast.hashCode();

    List<HeuristicExpressionVisitor> visitors = heuristicService.getHeuristics();

    while(previousHashCode != hashCode) {
        previousHashCode = hashCode;

        for (AstVisitor<ExpressionNode> visitor : visitors) {
            ast = visitor.visit(ast);

            hashCode = ast.hashCode();


            if (previousHashCode != hashCode) {
                list.add(printExpressionVisitor.visit(ast));
```

```
                break;
            }
        }
    }

    return list;
}
```

Fonte: Próprio autor.

Com toda a base para cadastro e interpretação implementada, bastou definir as heurísticas com o software em execução para analisar os resultados. Tomando-se como exemplo a heurística que trata da simplificação de adições, foi criada uma classe que estende a superclasse genérica "InfixNumberHeuristicExpressionVisitor", onde a heurística seria aplicada se, e somente se, os dois termos fossem nós numéricos tendo como pai um nó o operador de adição. A figura 60 demonstra como foi implementada em linguagem Groovy a classe para simplificação de adições.

Figura 60: Heurística para simplificação de adições.

```
1  import sk.host.arabasso.bassolve.core.ast.node.*
2  import sk.host.arabasso.bassolve.core.visitor.*
3
4  /**
5   * Created by arabasso on 20/03/2017.
6   */
7  @groovy.transform.TypeChecked
8  class AdditionHeuristicVisitor
9  extends InfixNumberHeuristicExpressionVisitor<AdditionNode> {
10     public AdditionHeuristicVisitor() {
11         super(AdditionNode.class)
12     }
13
14     @Override
15     double apply(double left, double right) {
16         return left + right
17     }
18 }
```

Fonte: Próprio autor.

Por fim, foram criadas diversas heurísticas para simplificação de expressões algébricas, funções matemáticas, exponenciação, etc., e deixadas previamente instaladas no software utilizando o *script* de inicialização de banco de dados fornecido pelo SpringBoot. A figura 61 detalha uma parte do referido *script*.

Figura 61: Fragmento do *script* de inicialização de banco de dados.

```
...

INSERT INTO user (username, password, enabled) VALUES ('admin', 'admin', 1);
INSERT INTO user (username, password, enabled) VALUES ('arabasso', '123', 1);
INSERT INTO user (username, password, enabled) VALUES ('raphael', '456', 0);

INSERT INTO role (name, user_id) VALUES ('ROLE_ADMIN', 1);
INSERT INTO role (name, user_id) VALUES ('ROLE_USER', 2);
INSERT INTO role (name, user_id) VALUES ('ROLE_USER', 3);

INSERT INTO heuristic (name, date, source) VALUES ('Equality', CURRENT_DATE(), 'import
sk.host.arabasso.bassolve.core.ast.node.*
import sk.host.arabasso.bassolve.core.visitor.*
```

```
/**
 * Created by arabasso on 20/03/2017.
 */
@groovy.transform.TypeChecked
class EqualityHeuristicVisitor extends InfixHeuristicExpressionVisitor<EqualityNode> {
    public EqualityHeuristicVisitor() {
        super(EqualityNode.class)
    }

    @Override
    ExpressionNode apply(ExpressionNode left, ExpressionNode right) {
        NumberNode n1 = (NumberNode)left
        NumberNode n2 = (NumberNode)right

        return new BooleanNode(Double.compare(n1.getValue(), n2.getValue()) == 0);
    }
}');

...
```

Fonte: Próprio autor.

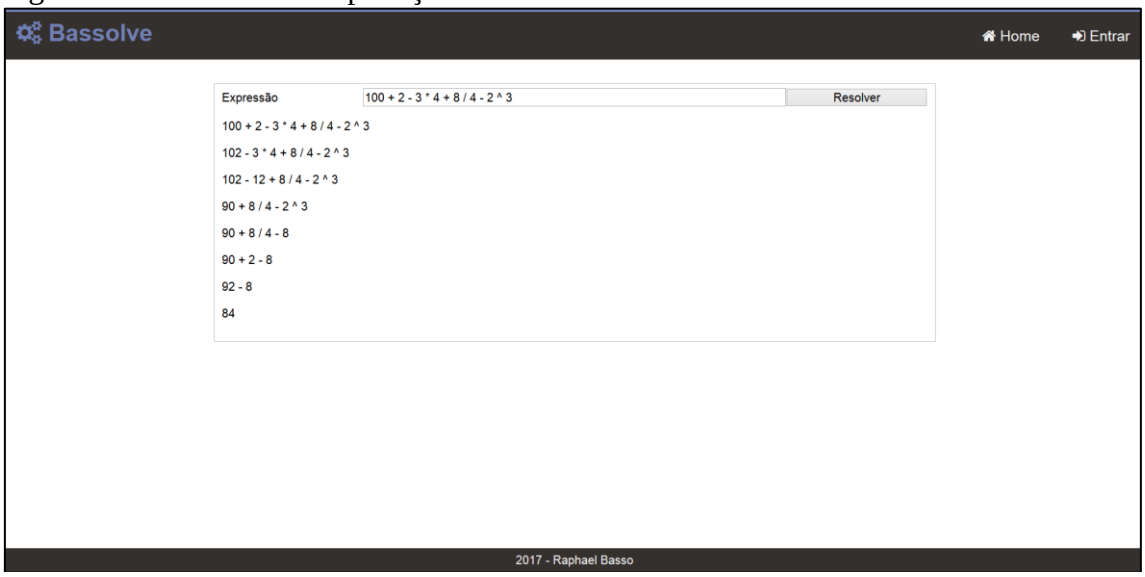
Importante ressaltar que todo o gerenciamento e controle de vida dos objetos como controladores, serviços, respositórios, entidades, sessões de banco de dados, entre outros, foi centralizado no *container* de injeção de dependências (Dependency Injection – DI) do próprio SpringFramework, favorecendo assim a inversão de controle (Inversion of Control – IoC) e tornando o código com alta coesão e baixo acoplamento.

4.4. Telas do Software

Com a conclusão do desenvolvimento e a execução dos devidos testes, foram feitas algumas capturas de tela para demonstrar como os atores interagem com o software.

Ao acessar a página inicial da aplicação, é exibida uma caixa de texto onde é possível informar uma expressão algébrica que pode ser resolvida de forma iterativa ao se clicar no botão “Resolver”. A figura 62 demonstra a referida tela.

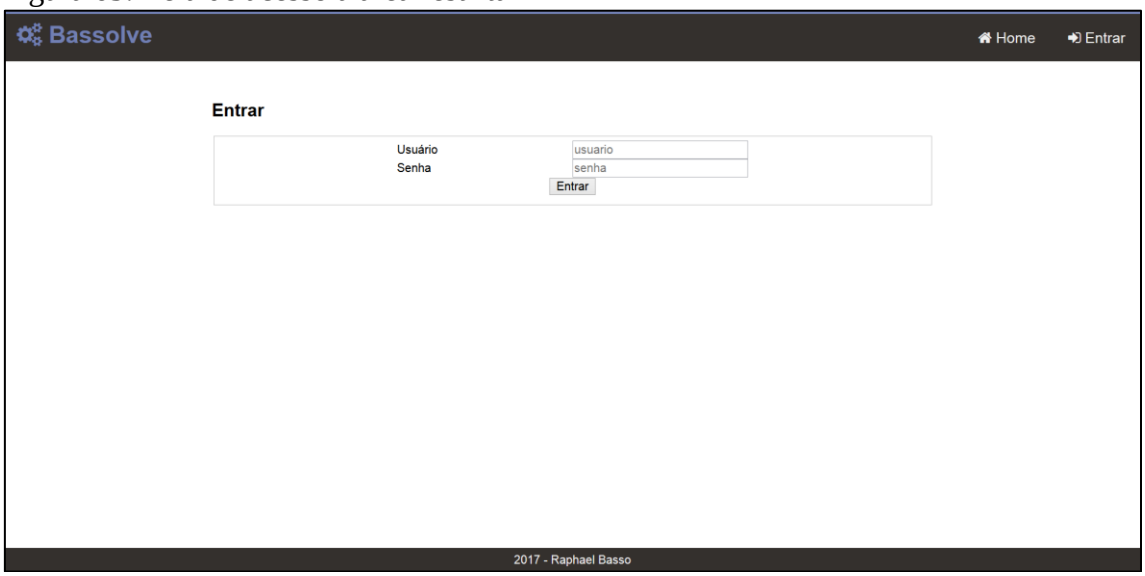
Figura 62: Tela inicial da aplicação.



Fonte: Próprio autor

Também na página inicial, é possível acessar o painel administrativo para gerenciamento das heurísticas de simplificação e resolução clicando no *link* “Entrar”, o que redireciona o usuário para efetuar a entrada na área restrita do software. A figura 63 demonstra a tela de acesso à área restrita.

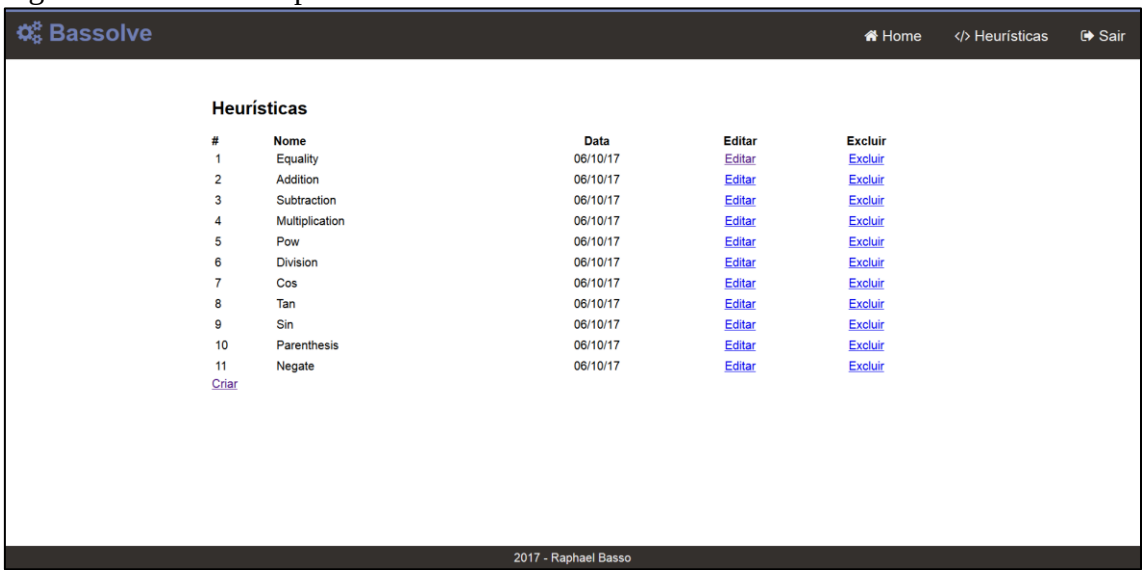
Figura 63: Tela de acesso à área restrita



Fonte: Próprio autor.

Uma vez efetuada a entrada na área restrita do software, é possível fazer o gerenciamento das heurísticas de simplificação e resolução de expressões algébricas selecionando o *link* "Heurísticas". Neste item, o usuário tem a possibilidade de incluir novas heurísticas de simplificação, bem como alterar e até excluir as previamente cadastradas. A figura 64 demonstra a lista de heurísticas previamente cadastradas pelo *script* de inicialização do SpringBoot.

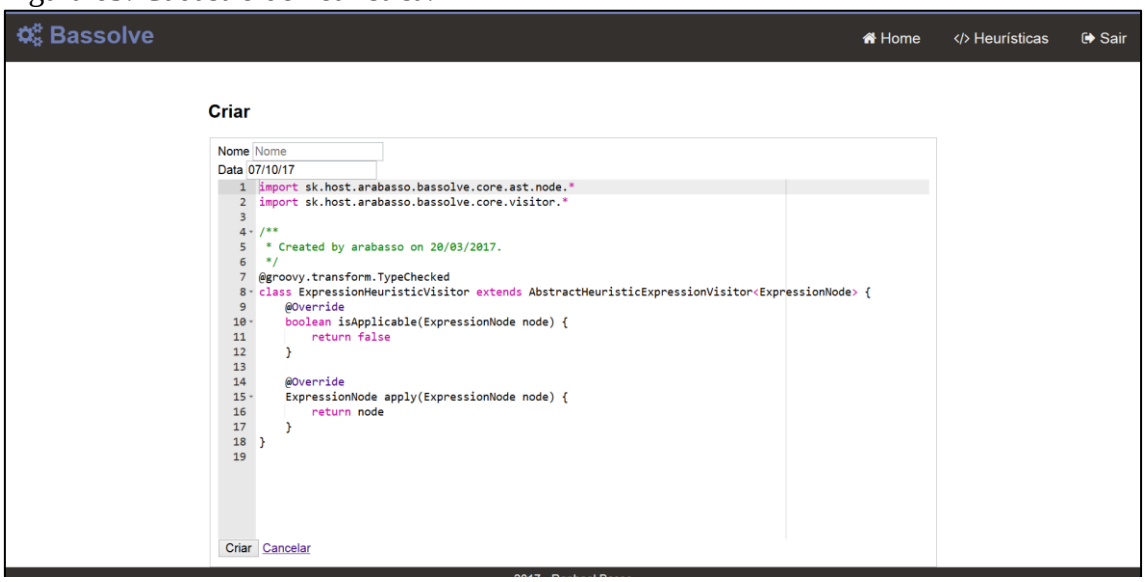
Figura 64: Heurísticas previamente cadastradas.



Fonte: Próprio Autor.

Clicando-se no *link* "Criar", se tem acesso à página de criação de heurísticas, onde é possível informar seu nome, data e o respectivo código-fonte para tratamento da simplificação. Inicialmente, é apresentada a data atual, bem como um código de exemplo para criação da heurística básica (que pode ser vista na figura 65).

Figura 65: Cadastro de heurística.



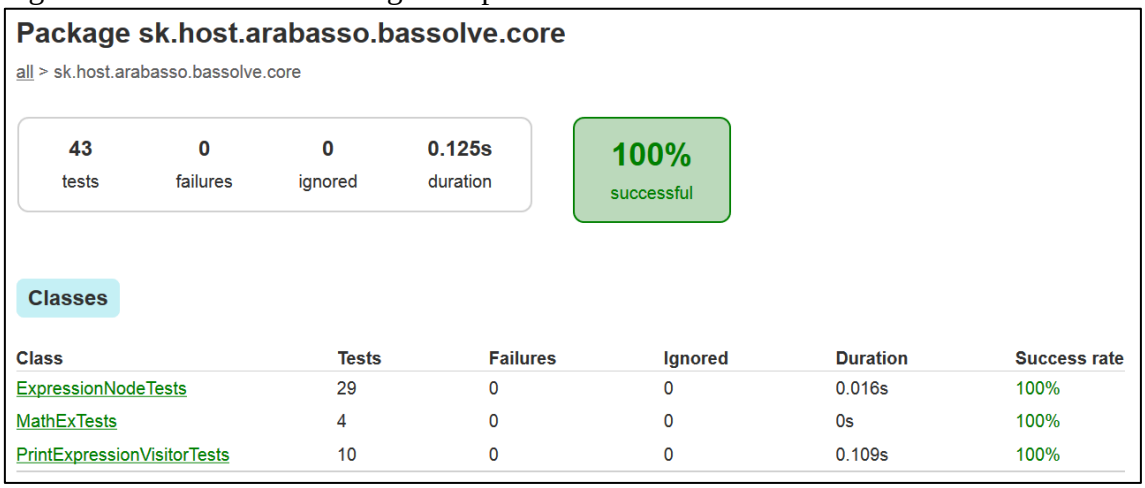
Fonte: Próprio autor.

4.5 Verificação e Validação

Como o princípio adotado para o desenvolvimento da aplicação foi o TDD, a maior parte dos problemas encontrados foram previamente resolvidos no decorrer do projeto utilizando-se testes automatizados de unidade, integração e sistema. Isto favoreceu com que o código ficasse limpo, coeso e com baixo acoplamento de classes.

Um ponto favorável ao uso de ferramentas para gerenciamento de projeto como o Gradle e o Maven é que para cada *build* da aplicação, todos os testes previamente criados são executados, e um relatório é gerado ao final do processo (além de atestar o correto funcionamento do código mostrando quais testes passaram ou não, o relatório informa também o tempo de execução das rotinas de teste). A figura 66 mostra o relatório de testes gerado em HTML para o módulo *core* quando se utiliza o Gradle para a construção da aplicação.

Figura 66: Relatório de testes gerado pelo Gradle.



Fonte: Próprio autor.

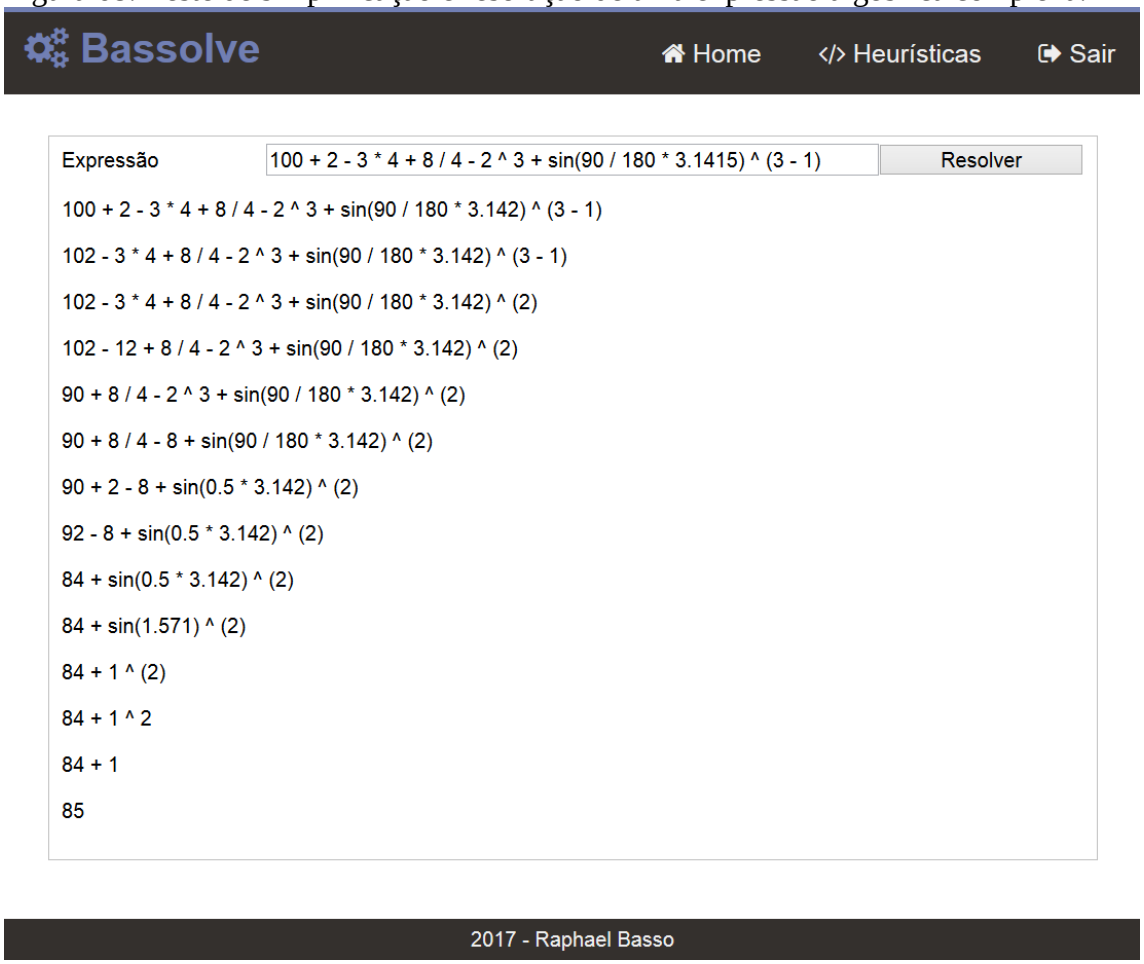
Além de testes automatizados, foram realizados testes para a validação da aplicação no cadastro e edição de heurísticas, bem como na simplificação e resolução de uma expressão algébrica extensa contendo diversos fatores. Para a validação do cadastro de heurísticas, foi dado como entrada um código em linguagem Groovy contendo um erro sintático acerca de uma superclasse inexistente, fazendo com que o software retornasse a linha, a coluna e a mensagem de exceção correspondente. O resultado deste teste por ser visto na figura 67.

Figura 67: Validação de compilação do código em linguagem Groovy.

Fonte: Próprio autor.

E para a validação de expressões algébricas complexas, foi utilizada a expressão “100 + 2 - 3 * 4 + 8 / 4 - 2 ^ 3 + sin(90 / 180 * 3.1415)”, pois esta contêm os tipos de simplificação baseados nas superclasses padrões. O resultado deste teste pode ser visto na figura 68.

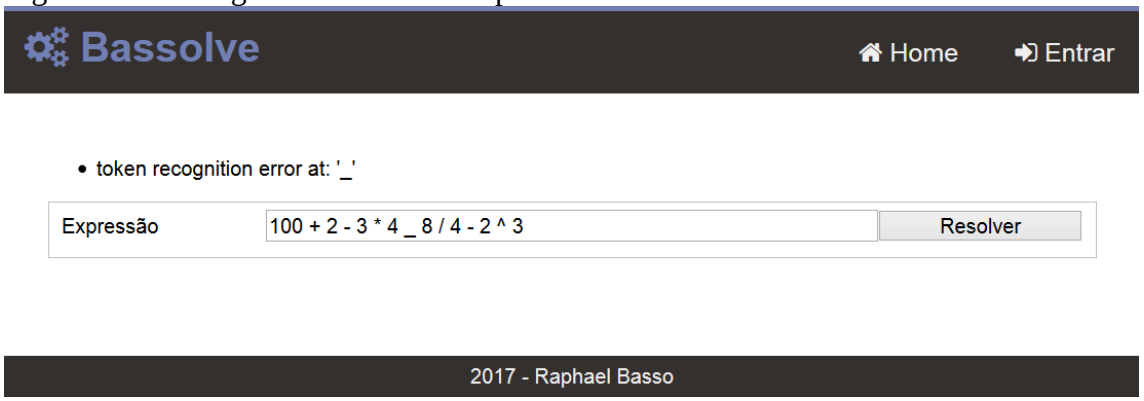
Figura 68: Teste de simplificação e resolução de uma expressão algébrica complexa.



Fonte: Próprio autor.

Para finalizar o processo de validação, foi realizado um teste utilizando como entrada a expressão “100 + 2 - 3 * 4 _ 8 / 4 - 2 ^ 3”, cujo erro se encontra no caractere sublinhado ao centro da expressão. Por conta deste erro, o software retornou a mensagem apresentada na figura 69.

Figura 69: Mensagem de erro emitida por causa de uma entrada inválida.



Fonte: Próprio autor.

4.6 Considerações Parciais

Neste capítulo foram apresentadas as ferramentas utilizadas para a estruturação do projeto, a implementação em código-fonte, as telas do *software* em funcionamento, e os testes realizados.

Pôde-se concluir com sucesso o desenvolvimento do projeto muito provavelmente pela forma como o projeto foi estruturado, pelos princípios de programação adotados, e também pelas ferramentas utilizadas.

Capítulo 5

Conclusão

5.1. Discussão Sobre os Resultados

Devido à complexidade do projeto, os resultados obtidos foram bastante satisfatórios, pois além de atender ao que foi proposto, permite um elevado grau de expansão com melhorias e adição de novos recursos.

O *software* como um todo é o resultado de diversos conhecimentos em informática, contendo por exemplo: teorias de compilação, princípios e práticas de desenvolvimento de *software* com o uso do DDD, TDD, padrões de projetos, *frameworks* e ferramentas diversas.

Quanto à estruturação do projeto, cabe ressaltar o excelente modelo de estrutura utilizado baseado em um multiprojeto, o que possibilitou implementar o design tático e estratégico do DDD que prevê a separação em *bounded contexts* (contextos delimitados) e módulos. Uma vez que foram utilizadas as ferramentas Gradle e Maven, foi possível adicionar gerenciamento de dependências ao projeto, bem como sua construção de forma automatizada e facilitando a compilação sem que houvesse a necessidade da instalação de ambientes de desenvolvimento integrado (é possível construir o projeto somente com o JDK instalado e um utilitário de linha de comando).

Sobre o analisador de expressões, o processo de criação foi bastante simplificado por conta do uso da ferramenta ANTLR, a qual permite a criação do código para análise léxica e sintática para diversas linguagens de programação baseado numa gramática do tipo g4 (no caso, foi utilizado para gerar código Java). É importante notar que o ANTLR gera o código para a linguagem Java basicamente utilizando dois padrões de projetos: o Composite (para a árvore sintática) e o Visitor (para navegação da árvore). O uso destes dois padrões de projeto forneceu ao software subsídios para que fosse possível a criação das heurísticas de simplificação e resolução de expressões algébricas.

Interessante ressaltar também a alta coesão e o baixo acoplamento de classes do software por conta do uso dos princípios e práticas do DDD, TDD e padrões de projetos. No caso do DDD, foram utilizados conceitos do design tático como entidades, serviços de domínio e repositórios para o gerenciamento centralizado das entidades em banco de dados. O TDD também foi de suma importância para favorecer essa coesão e baixo acoplamento de classes, uma vez que força o programador a criar testes automatizados e implementações simplificados, mas que atendam aos requisitos exigidos.

Outro ponto importante é com relação à integração entre a linguagens Java e Groovy, pois este recurso forneceu ao *software* a possibilidade de extensão sem que seja necessário a recompilação do código-fonte do projeto. Todas as heurísticas ficam

armazenadas em banco de dados, podendo ser alteradas, e caso necessário, podem ser criadas novas heurísticas permitindo assim estender as funcionalidades do software.

Cabe registrar também que a junção harmônica de todos os recursos descritos no projeto se deveu em grande parte pelo uso do ótimo *framework* para desenvolvimento de aplicações corporativas SpringBoot, uma vez que este já traz implementações padrão dos diversos conceitos empregados no projeto (como DDD, TDD, padrões de projetos, etc.).

5.2. Desafios Encontrados

Ocorreram diversos percalços durante a preparação do projeto, principalmente por conta da complexidade do software, além da escassez de referências bibliográficas em português sobre diversos tópicos requeridos para o desenvolvimento.

Outro desafio encontrado foi na documentação da seção de projeto, e mais uma vez por conta de o *software* ter uma elevada complexidade, foi necessário documentar tal seção de forma bastante abrangente para se evitar problemas futuros na seção de desenvolvimento.

No que concerne o desenvolvimento do *software*, houveram dificuldades diversas desde a estruturação do projeto utilizando as ferramentas de construção automatizada do Gradle e Maven (onde foi feita criado um multiprojeto com uma separação em dois módulos e diversos pacotes para atender alguns requisitos do DDD), passando pelo módulo *core* (que contêm o coração do software) até o módulo *web*. Neste último, foram realizados diversos ajustes para que fosse possível executar a aplicação sem a necessidade de um servidor de aplicação (recurso fornecido pelo SpringBoot), como também fornecer responsividade caso a aplicação fosse acessada de dispositivos móveis.

A maior dificuldade, no entanto, ficou por conta da possibilidade da extensão do software por meio de código armazenado em banco de dados, utilizando para tal a integração entre a linguagem Java e Groovy. Esta integração acabou requerendo um esforço considerável para que o código carregado do banco de dados H2 e interpretado em tempo de execução fosse integrado de forma correta.

5.3. Trabalhos Futuros

Como o software foi desenvolvido de forma bastante genérica, possibilitando extensibilidade, é possível realizar diversas melhorias, bem como adicionar novos recursos sem maiores complicações. Por exemplo, apesar do projeto ter tratado apenas de simplificação de expressões algébricas simples contendo apenas números, é possível adicionar suporte a simplificação de polinômios, de derivadas e de integrais, entre outros.

Com mais alguns ajustes no projeto, é possível também implementar uma forma de determinar a partir da entrada qual é o melhor conjunto de heurísticas aplicável ao contexto, permitindo assim agregar um vasto conhecimento computacional com rotinas de mineração de dados, pesquisa operacional, métodos numéricos, desenho de gráficos, etc.

Referências Bibliográficas

Colocar em ordem alfabética

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.

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

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

FOWLER, Martin. **Inversion of control containers and the dependency injection pattern.** Disponível em <https://martinfowler.com/articles/injection.html>. Acesso em 20 mai. 2017.

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

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

JAVA. **Obtenha informações sobre a tecnologia java.** Disponível em: <https://www.java.com/pt_BR/about/>. Acesso em: 22 mai. 2017.

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

SHVETS, Alexandre. **Design patterns explained simply.** [S.I.]: Sourcemaking.com, 2001.

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.

THE JAVA™ TUTORIALS. **About the java technology.** Disponível em: <https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html>. Acesso em: 22 mai. 2017.