IMPLEMENTAÇÃO DE UMA LINGUAGEM SEMI-NATURAL ESTRUTURADA PARA GERAÇÃO DE INTERFACES GRÁFICAS FUNCIONAIS EM JAVA

Por

LUCAS LOPES FRAGA

Trabalho de Conclusão de Curso

Instituto Federal de Educação, Ciência e Tecnologia da Bahia
coord.bsi.fsa@ifba.edu.br
portal.ifba.edu.br/feira-de-santana

FEIRA DE SANTANA/2021

Lucas Lopes Fraga

IMPLEMENTAÇÃO DE UMA LINGUAGEM SEMI-NATURAL ESTRUTURADA PARA GERAÇÃO DE INTERFACES GRÁFICAS FUNCIONAIS EM JAVA

Trabalho apresentado a Coordenação da Graduação em Sistemas de Informação do Bacharelado em Sistemas de Informação do Instituto Federal de Educação, Ciência e Tecnologia da Bahia como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: José Dihego da Silva Oliveira

FEIRA DE SANTANA
2021

Lucas Lopes Fraga

Implementação de uma Linguagem Semi-natural Estruturada para Geração de Interfaces Gráficas Funcionais em Java/ Lucas Lopes Fraga. – FEIRA DE SANTANA, 2021-

42 p. : il. (algumas color.) ; 30 cm.

Orientador José Dihego da Silva Oliveira

Trabalho de Conclusão de Curso – Instituto Federal de Educação, Ciência e Tecnologia da Bahia , 2021.

1. Geração de código. 2. Linguagem semi-natural. I. José Dihego da Silva Oliveira. II. Instituto Federal de Educação, Ciência e Tecnologia da Bahia, Campus Feira de Santana. III. Faculdade de xxx. IV. Implementação de uma Linguagem Semi-natural Estruturadapara Geração de Interfaces Gráficas Funcionais em Java.

CDU 02:141:005.7

Trabalho de conclusão de curso apresentado por **Lucas Lopes Fraga** ao Bacharelado em Sistemas de Informação do Instituto Federal de Educação, Ciência e Tecnologia da Bahia, Campus Feira de Santana, sob o título **Implementação de uma Linguagem Semi-natural Estruturada para Geração de Interfaces Gráficas Funcionais em Java**, orientado pelo(a) **Prof(a). José Dihego da Silva Oliveira** e aprovado pela banca examinadora formada pelos professores(as):

Prof(a). 1
Bacharelado em Sistemas de Informação/IFBA

Prof(a). 2
Bacharelado em Sistemas de Informação/IFBA

Prof(a). 3
Bacharelado em Sistemas de Informação/IFBA

FEIRA DE SANTANA
2021

Dedico este trabalho a minha família, meus amigos, meus professores e todos os que ajudaram na criação e desenvolvimento deste trabalho.

Agradecimentos

Agradeço a Deus por me dar uma segunda chance na vida e permitir que eu chegasse até aqui.

Agradeço veementemente a minha família que me criou e me apoiou até hoje, dando tudo de si para que eu pudesse chegar onde cheguei.

Aos meus amigos por me apoiarem nas minhas decisões e me ajudarem durante esse período.

Ao meu orientador José Dihego por me ajudar com o desenvolvimento desse projeto e dar o suporte necessário, e a todos os outros professores do IFBA Campus Feira de Santana.

Resumo

Devido ao aumento da complexidade dos sistemas de informações nos dias atuais, torna-se cada vez mais necessário que haja ferramentas que reduzam o tempo de desenvolvimento de software e auxiliam os desenvolvedores na hora de programar, tanto para a aceleração do processo de desenvolvimento de software quanto para que se evite erros no código. Para cumprir tal função com eficácia, essa ferramenta também precisa ser de fácil aprendizado e utilização para que seu uso seja possível por desenvolvedores iniciantes e experientes e para que não se perca muito tempo aprendendo como utilizar a ferramenta ou ao utilizá-la, permitindo assim que o tempo gasto na sua utilização seja satisfatório. Neste trabalho propomos a criação de uma linguagem semi-natural para geração de código com a intenção de acelerar o desenvolvimento de software a partir da criação de interfaces gráficas funcionais. Para atingir tal fim, foi criada uma estrutura para essa linguagem e também foi desenvolvido um algoritmo que analisa e interpreta o código escrito nessa linguagem semi-natural, gerando uma interface gráfica funcional a partir dele que implementa as funções CRUD. Com o algoritmo de geração de código criado também foi feito um estudo de caso para demonstrar a viabilidade e a eficácia de nossa abordagem. A partir dos resultados do estudo de caso, foi observado que o algoritmo é capaz de gerar interfaces funcionais que formam um ponto de partida importante para implementação, validação e testes de sistemas.

Palavras-chave: geração de código. linguagem semi-natural. aceleração no desenvolvimento de software. interface gráfica. CRUD.

Abstract

Due to the increasing complexity of information systems nowadays, it is becoming more essential to have tools that reduce the time of software development and help developers when programming, both to speed up the software development process and to avoid errors in code. To fulfill this function effectively, this tool also needs to be easy to learn and use, so that its use is possible by both novice and experienced developers and so that time isn't wasted when learning how to use the tool or when utilizing it, allowing so that the time spent on its use is satisfactory. This work proposed the creation of a semi-natural language for code generation, to accelerate software development with the creation of functional graphical user interfaces. To achieve this end, a structure for this language was created and an algorithm was developed to analyze and interpret the code written in this semi-natural language, generating a functional graphical user interface based on the aforementioned code, which implements the CRUD functions. By using the created code generation algorithm, a case study was also carried out to demonstrate the feasibility and effectiveness of our approach. From the results gathered from the case study, it was observed that the algorithm is capable of generating functional graphical user interfaces that form an important starting point for system implementation, validation, and testing.

Keywords: code generation. semi-natural language. acceleration of software development. graphical user interfaces. CRUD.

Lista de Figuras

Lista de Acrônimos

CRUD	*Create, Read, Update and Delete*
DAO	*Data Access Object*
API	*Application Programming Interface*
LSGC	Linguagem Semi-natural para Geração de Código
UUID	*Universally Unique Identifier*
GUI	*Graphical User Interface*

Sumário

1

Introdução

Com o aumento da importância e complexidade dos sistemas de informação nos dias atuais, cada vez mais desenvolvedores e projetistas desejam acelerar e simplificar etapas do processo de planejamento, estruturação, prototipação e desenvolvimento de software. Para agilizar a etapa de descoberta de requisitos deste processo devemos mostrar interfaces gráficas do usuário[1] funcionais com diversos cenários através de protótipos para os usuários com o objetivo de validar o sistema e ajudá-los a compreender o sistema. Segundo (SOMMERVILLE, 2011, p.30):

> Protótipos do sistema permitem aos usuários ver quão bem o sistema dá suporte a seu trabalho. Eles podem obter novas ideias para requisitos e encontrar pontos fortes e fracos do software; podem, então, propor novos requisitos do sistema. Além disso, o desenvolvimento do protótipo pode revelar erros e omissões nos requisitos propostos. A função descrita em uma especificação pode parecer útil e bem definida. No entanto, quando essa função é combinada com outras, os usuários muitas vezes percebem que sua visão inicial foi incorreta ou incompleta.

Ao auxiliar em mostrar protótipos dos sistemas aos usuários, os desenvolvedores podem receber *feedback* e trabalhar melhor na especificação de requisitos de sistema, pois "a satisfação dos requisitos especificados pelos usuários é a precondição básica para o sucesso de um software" (TURINE; MASIERO et al., 1996, p.3).

Para que essa validação ocorra cedo o suficiente e para que esses protótipos sejam feitos de forma rápida é importante que haja ferramentas que permitam aos programadores criarem interfaces gráficas funcionais com baixo custo e esforço. HARRISON; BARTON; RAGHAVACHARI (2000), escreveram em sua obra que ferramentas de geração de código além de acelerar o processo de desenvolvimento de software, também auxiliam a manter uma consistência entre o modelo do sistema e a sua implementação.

Ferramentas deste tipo possibilitam uma validação veloz dos requisitos de qualidade de software com o usuário através da amostragem de diversos protótipos criados com a intenção de mostrar uma determinada função do sistema ou o visual de uma das telas presentes no

[1] Interfaces gráficas ou interfaces gráficas do usuário (vindo do inglês GUI, *Graphical User Interface*) são interfaces que permitem a interação entre o usuário e a máquina utilizando elementos gráficos.

software a ser desenvolvido, pois, é essencial verificar o sistema com o usuário para saber se esse determinado software atende as necessidades do usuário. Essas verificações segundo ADRION; BRANSTAD; CHERNIAVSKY (1982) devem ocorrer o mais cedo possível, pois caso contrário, a descoberta do problema e por conseguinte a refatoração do código terá um custo muito mais elevado.

Sistemas de informação na maior parte das vezes utilizam de métodos que criam, leem, atualizam e deletam dados, esses métodos são denominados CRUD (termo vindo do inglês: *Create, Read, Update and Delete*) (YODER et al., 1998). Sem a presença dessas quatro funções fundamentais é difícil armazenar ou coletar informações, fazendo com que o CRUD seja de vital importância em sistemas de informação e implementá-las consome um tempo considerável do processo de desenvolvimento.

Esse projeto pretende criar uma solução de software capaz de produzir, a partir de modelos abstratos, interfaces gráficas funcionais. Tais interfaces que conseguem gerar código com classes, métodos e interfaces gráficas, possibilitando desenvolvedores de software a criação de protótipos verticais funcionais com baixo esforço e que podem servir como ponto de partida para a fase de implementação. Vale destacar que, exigindo apenas modelos de representação de alto nível, diversos profissionais da cadeia produtiva de software podem se beneficiar deste trabalho, não apenas aqueles com perícia em desenvolvimento. Aumentando assim a produtividade destes que são da área de desenvolvimento, possibilitando uma rápida prototipagem de um sistema e também permitindo que profissionais sem formação na área de desenvolvimento de sistemas consigam criar e visualizar um sistema em execução.

Neste trabalho definiremos uma linguagem de representação de sistemas (estrutura e comportamento) de alto nível chamada LSGC, sigla para Linguagem Semi-natural para Geração de Código, com o intuito de ser de fácil aprendizado e que acelere o tempo de criação de software pela representação via especificação em alto nível do sistema a ser desenvolvido. Também será permitida uma fácil integração de sistemas de gerenciamento de banco de dados via customização das classes DAO.

Uma das principais contribuições deste trabalho é a criação de uma linguagem de natureza semi-natural, ou seja, uma linguagem similar às linguagens naturais (como as linguagens escritas), mas que exiba particularidades das linguagens formais (como as linguagens de programação). A segunda contribuição deste trabalho consiste na geração de interfaces gráficas plenamente funcionais com base em sistemas representados em nossa linguagem semi-natural.

Escolhemos Java (NAUGHTON; SCHILDT, 1996) como a linguagem destino da nossa ferramenta, linguagem de programação que foi originalmente desenvolvida pela Sun Microsystems, e utilizará do conjunto de ferramentas Java Swing para a criação dos elementos visuais presentes no trabalho. Tal escolha se deve ao fato de Java ser uma linguagem sólida e bem documentada com *layouts* simples que auxiliam na hora do desenvolvimento e de sua presença ainda firme no mercado.

1.1 Justificativa

A linguagem de programação Java, foi desenvolvida na década de 90 e ainda é uma das linguagens mais utilizadas no mundo[2], de acordo com (DEMASTER, 2000) a popularidade da linguagem Java ocorreu devido à sua independência de plataforma, orientação a objetos e natureza dinâmica.

Devido à complexidade dos sistemas atuais, foram criadas várias maneiras de se reduzir o tempo de desenvolvimento e tratamento de erros, com a programação orientada a objetos, a linguagem de programação Java reduziu muito esse tempo utilizando referências a objetos, a proposta da LSGC é reduzir ainda mais esse tempo com uma criação de código otimizada.

O tempo gasto no desenvolvimento das classes básicas, interfaces e métodos CRUD de um sistema de informação cresce de acordo com o aumento do escopo e complexidade do sistema, isso faz com que os desenvolvedores do sistema gastem mais tempo repetindo os procedimentos básicos de criação de classes ao invés de focar nas partes mais complexas do projeto, por isso é importante que haja ferramentas simples e práticas que auxiliem nessa etapa do processo de desenvolvimento de software, com a LSGC é possível acelerar esse processo dessa parte mais simples do código possibilitando estes desenvolvedores a focar diretamente na parte mais complexa do sistema.

Com base nessa premissa, esse trabalho se propõe a criar uma ferramenta para geração de interfaces gráficas funcionais. Com a API do Java Swing, essas interfaces terão as funções básicas de cadastro, leitura, atualização e remoção, com mínimo custo e esforço.

A linguagem de alto nível (LSGC) desenvolvida neste trabalho tem uma natureza semi-natural para facilitar o aprendizado da linguagem e permitir que programadores e desenvolvedores de diversos níveis de conhecimento sobre a área de desenvolvimento de sistemas possam usar essa ferramenta. O intuito desta linguagem é de permitir que os desenvolvedores foquem nas partes mais complexas do projeto ao invés de repetir os procedimentos mais básicos, com isso acelerando o processo de desenvolvimento de software e possibilitando uma prototipagem, desenvolvimento e validação mais rápidas.

Com o foco em acelerar o processo de desenvolvimento de software, a LSGC planejada permitirá que desenvolvedores possam gerar múltiplas classes com uma interface gráfica funcional utilizando somente algumas palavras-chave que serão analisadas por um algoritmo de geração de código versátil e leve que possa ser executado em múltiplos ambientes e que gere esse código descrito na LSGC em questão de poucos segundos permitindo assim que os processos de prototipação, validação de requisitos com usuários e de desenvolvimento de software obtenham uma redução de tempo considerável.

[2]De acordo com uma enquete organizada pelo Stack Overflow em 2018. Disponível em: https://insights.stackoverflow.com/survey/2018/#most-popular-technologies Acesso em: 29 de nov. de 2019.

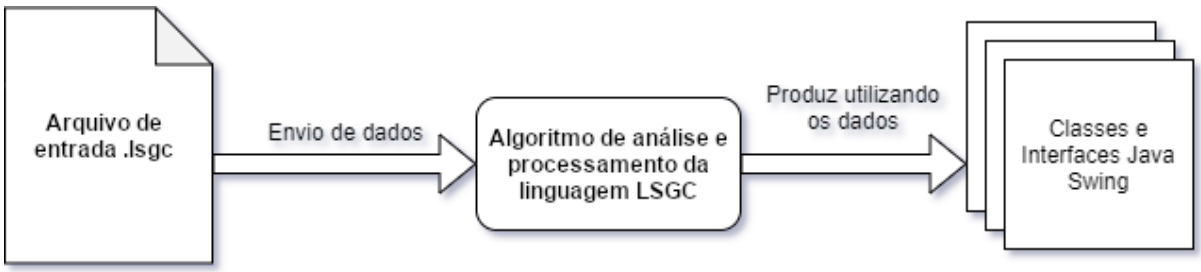
2

Objetivos

O objetivo geral deste trabalho é desenvolver uma solução de software para geração automática de interfaces gráficas plenamente funcionais a partir de uma representação abstrata de sistemas em uma linguagem semi-natural. Tais interfaces gráficas que possuem as operações CRUD através de um sistema desenvolvido em Java que receberá com entrada uma descrição do sistema em LSGC (Linguagem Semi-natural para Geração de Código) e transformá-los em interfaces gráficas funcionais em Java Swing, porém o escopo deste trabalho irá focar em somente três tipos de atributos e não irá trabalhar com os conceitos de herança de classes, que é encontrado em linguagens de orientação a objetos onde uma classe tem outra como atributo, e também não trabalhará com a implementação de banco de dados no algoritmo de geração de código.

Esse trabalho também tem o objetivo de fazer um estudo de caso utilizando a solução de software desenvolvida no trabalho para teorizar sobre a eficácia da LSGC no processo de aceleração de desenvolvimento de software através das métricas e dados obtidos no estudo de caso com a intenção de exemplificar o grau de complexidades das funcionalidades geradas de forma automática para um sistema de gestão de dados.

Figura 2.1: Processo generalizado de criação de classes.



Fonte: Criada pelo Autor

Objetivos Específicos

- Criar uma sintaxe e semântica para a linguagem semi-natural de representação que irá funcionar como entrada.
- Desenvolver um algoritmo capaz de fazer uma análise do código em LSGC.
- Projetar e desenvolver algoritmos capazes de criar classes, o DAO e o CRUD considerando os dados relevantes ao sistema descrito na linguagem semi-natural.
- Fazer um estudo de caso com a LSGC e o algoritmo desenvolvido.
- Analisar os dados obtidos pelo estudo de caso e verificar a eficácia da LSGC na aceleração do processo de desenvolvimento de código.

3

Metodologia

O estudo iniciou-se através de pesquisas bibliográficas e análises preliminares sobre o tema da pesquisa com o propósito de identificar métodos de desenvolvimento da linguagem de programação semi-natural estruturada.

A primeira biblioteca visitada para as pesquisas bibliográficas foi a biblioteca do IFBA - Campus Feira de Santana, onde aprofundei meus conhecimentos em Java para a futura criação do algoritmo. A pesquisa foi principalmente exploratória para que houvesse a familiarização com as ferramentas que serão necessárias para criação do sistema. Para o aprofundamento nos assuntos, foram também lidas várias obras disponíveis digitalmente.

O tipo de pesquisa deste trabalho foi exploratório em relação a LSGC e o algoritmo de geração de código que foram desenvolvidos.

A partir da pesquisa bibliográfica realizada na biblioteca do IFBA e da leitura de várias obras disponíveis digitalmente em bibliotecas virtuais, a sintaxe e a semântica da LSGC foram planejadas para fazer com que ela portasse as características de uma linguagem semi-natural com o intuito de facilitar o processo de aprendizado e uso da linguagem criada.

Para a realização deste trabalho foi desenvolvido um protótipo funcional para testar a eficiência do programa. Para o desenvolvimento desse protótipo, as seguintes etapas foram executadas:

- Pesquisa sobre a linguagem de programação Java e seus *frameworks* para escolher com maior eficiência o que vai ser utilizado no algoritmo.
- Desenvolvimento de um algoritmo que analisa e interpreta a LSGC planejada.
- Desenvolvimento de classes auxiliares para a geração do código baseado nas informações descritas na LSGC.
- Criação de um algoritmo que utiliza do código gerado para construir uma interface gráfica totalmente funcional.

Para construir as interfaces gráficas funcionais, os conceitos estudados de Java Swing e CRUD foram utilizados para dar funcionalidades às interfaces através do gerenciamento de objetos.

Este protótipo funcional foi usado para testar a eficiência do programa e também foi utilizado para um estudo de caso em uma máquina onde as seguintes métricas quali-quantitativas foram avaliadas com a intenção de mostrar o desempenho do algoritmo desenvolvido em uma máquina:

- Tamanho de armazenamento do algoritmo de geração de código.
- Tempo médio gasto para geração do código.
- Tempo médio gasto para montagem da interface gráfica.
- Quantidade de erros devido a falhas de sistema.

Também foi feito um pequeno estudo de caso para obter dados em relação à velocidade, desempenho e eficácia da LSGC e do algoritmo de geração de código para ter uma ideia básica do funcionamento e desempenho do algoritmo.

4

Fundamentação Teórica

O referencial teórico desta pesquisa é composto pelos conceitos de geração automática de código, linguagem natural, semi-natural e de alto nível, CRUD, DAO e da linguagem Java e um *widget toolkit*[1] de Java, o Java Swing. Como a LSGC é uma linguagem semi-natural para geração de códigos automática, estes conceitos são essenciais para haver o entendimento de como ela opera.

Os conceitos de linguagem natural e de alto nível servem para distingui-los entre o conceito de linguagem semi-natural, os conceitos de CRUD, DAO, Java e Java Swing se referem ao código e o conjunto de ferramentas a ser geradas pela LSGC e os conceitos de geração automática de código e compiladores auxiliaram a entender como esse procedimento ocorre.

O conceito principal de geração automática de código foi utilizado durante todo o processo e sua estruturação sendo essencial para o trabalho. Os conceitos de linguagem de alto nível, natural e semi-natural serviram para auxiliar na estrutura da LSGC com o intuito de fazer com ela seja simples e fácil de se entender e se assemelhe o máximo possível a uma linguagem natural mesmo tendo as restrições de uma linguagem de alto nível, fazendo com que ela seja uma linguagem semi-natural.

Como o algoritmo é escrito em Java, a fundamentação de Java é essencial para haver um entendimento de como esse algoritmo opera, e já que para a fabricação de interfaces gráficas foi-se utilizado o *widget toolkit* Java Swing, este também se dá como conhecimento fundamental. Na implementação do algoritmo, os conceitos de CRUD foram utilizados para dar funcionalidade às interfaces gráficas criadas e o DAO serve para segmentar as funções na parte do código e para uma possível ampliação deste trabalho através do uso de permanência de dados.

4.1 Geração Automática de Código

A Geração Automática de Código é uma área em crescimento que utiliza de ferramentas de geração de código, também conhecidas como geradores de código, para gerar código através de um determinado modelo, por exemplo, compiladores que transformam código escrito através de uma linguagem de programação para código de máquina. Segundo NETO (2011) as ferramentas

[1] Um *widget toolkit* é uma biblioteca ou uma coleção de bibliotecas que contém uma coleção de elementos interativos gráficos, como botões e barras, para construir a interface gráfica de programas.

de geração de código automática economiza tempo no desenvolvimento de soluções de software, um fator importante nas questões de produtividade e a existência de aplicações de software capazes de fazer a geração automática de código, ela também agiliza o desenvolvimento de novas aplicações, pois ela fornece de imediato o respectivo código.

Uma das formas mais conhecidas de criação automática de código são os compiladores. Compiladores são de extrema importância para que haja a tradução de uma linguagem para outra, um compilador é de forma resumida "[...] um programa que lê um programa escrito numa linguagem — a linguagem fonte — e o traduz num programa equivalente numa outra linguagem — a linguagem alvo." (AHO et al., 2007, p.1) Além disto o compilador tem como função apresentar mensagens de erro caso um problema ocorra durante a tradução.

Quando o código é gerado automaticamente e uniformemente pela ferramenta, ela também garante que todos os códigos criados seguirão um padrão e evitará que erros ocorram durante a construção dessas classes, pois as mesmas seguirão a mesma fórmula. O código gerado por esse trabalho está escrito em uma linguagem de alto nível, conceito que também foi utilizado para auxiliar na criação da LSGC.

4.2 Linguagem de programação de alto nível

Linguagens de programação de alto nível são caracterizadas por terem um nível de abstração relativamente elevado, mais próxima à linguagem humana, quando comparado as linguagens baixo nível, que por sua vez, estão relacionadas com a arquitetura do computador e utilizam instruções do processador, essa relação permite que uma linguagem possa ser escrita a partir de outra com um nível mais baixo (SANTOS et al., 2011).

4.3 Linguagem semi-natural

A Linguagem semi-natural, também conhecida como linguagem semi-formal, é um meio-termo entre as linguagens naturais, que de acordo com LYONS (1991) são as linguagens desenvolvidas naturalmente pelos seres humanos através do uso, da repetição e da ausência de planejamento consciente, e as linguagens formais como as linguagens de programação de alto nível.

Segundo (LOH, 1991, p.208) "[...] apresenta-se a linguagem como semi-natural por ser um subconjunto da Linguagem Natural e como semi-formal por ser mais precisa que a Linguagem Informal e por não apresentar o grau de formalismo exigido para as linguagens formais". Por não possuir o grau de formalidade exigido por uma linguagem formal, a linguagem semi-natural auxilia no entendimento do código escrito e possibilita que diversos profissionais das fábricas de software possam entender e utilizar este código com mais rapidez.

4.4 Java

Java é uma das linguagens de programação de alto nível mais utilizadas nos dias de hoje[2]. “A tecnologia Java foi criada como uma ferramenta de programação de um projeto da Sun Microsystems, chamado The Green Project, iniciado por Patrick Naughton, Mike Sheridan e James Gosling, em 1991.” (MENDES, 2009, p.16). O principal objetivo de Java não era de ser uma linguagem de programação, e sim de criar uma nova plataforma para computação interativa.

A popularidade de Java foi grande devido a uma série de vantagens que ela tinha comparada a muitas linguagens de programação que vieram antes, entre estas vantagens estão:

A simplicidade da linguagem, pois Java “permite o desenvolvimento de sistemas em diferentes sistemas operacionais e arquiteturas de hardware, sem que o programador tenha que se preocupar com detalhes de infra-estrutura.” (MENDES, 2009, p.17).

O seu paradigma, que diferentemente de muitos dos seus antecessores que utilizavam os paradigmas imperativos e estruturados, Java foca na Orientação à Objetos assim trazendo a vantagem da independência de módulos que de acordo com JUNGTHON; GOULART (2009), garante a maior chance daquele módulo ser reutilizado no futuro e a alteração dele não infere na alteração dos outros módulos.

Na utilização de Java, é possível utilizar diversos *widget toolkits* para ampliar ainda mais seu escopo e possibilitar a criação das mais diversas ferramentas, um desses *widget toolkits* é o Java Swing.

4.5 Java Swing

O Java Swing segundo LOY et al. (2002) é um conjunto de ferramentas para criação de interfaces gráficas criado pela Sun Microsystems para possibilitar que os programadores possam utilizar desta ferramenta para criar aplicações Java de grande escala com uma grande gama de componentes que podem ser facilmente estendidos e modificados para modificar a aparência e o comportamento deles.

O nome Swing não é um acrônimo, e foi uma escolha dos seus projetistas quando o projeto foi iniciado em 1996, O Java Swing começou a ser desenvolvido em 1997, e quando foi lançado em sua primeira versão, Swing 1.0 em 1998, sua biblioteca já contava com aproximadamente 250 classes e 80 interfaces (LOY et al., 2002).

Pelo fato do Java Swing ser mais robusto que sua competição e ter acesso a uma maior quantidade de classes e interfaces, ele foi selecionado para o projeto sendo utilizado junto ao *GroupLayout*, que é um tipo de disposição de elementos numa interface gráfica, permitindo assim que elementos sejam encaixados na tela de uma forma precisa e flexível, pois o *GroupLayout* posiciona os elementos tanto horizontalmente quanto verticalmente.

[2]De acordo com uma enquete organizada pelo Stack Overflow em 2018. Disponível em: https://insights.stackoverflow.com/survey/2018/#most-popular-technologies Acesso em: 29 de nov. de 2019.

Para adicionar funcionalidades às interfaces gráficas geradas em Java Swing, foram criadas tabelas, campos e botões para a implementação das operações básicas de persistência, ou seja, o CRUD.

4.6 CRUD

O CRUD, também conhecido como *Create, Read, Update and Delete* são operações básicas de persistência, que no contexto de computação segundo BALZER (2005) significa que um estado sobrevive mesmo após a morte do processo, o que é obtido ao adicionar a informação em um banco de dados. Essas operações básicas de persistência permitem que informações fiquem salvas até mesmo após o término do uso de um programa, conforme YODER et al. (1998) todos os objetos persistentes (objetos criados por via do paradigma de orientação a objetos que se enquadram no conceito de persistência da computação) precisam de operações para ler e escrever sobre eles mesmos no banco de dados, pois suas informações podem precisar serem atualizadas ou excluídas no futuro, tornando obrigatório que haja ao menos métodos para se poder criar, ler, atualizar e deletar esses objetos persistentes.

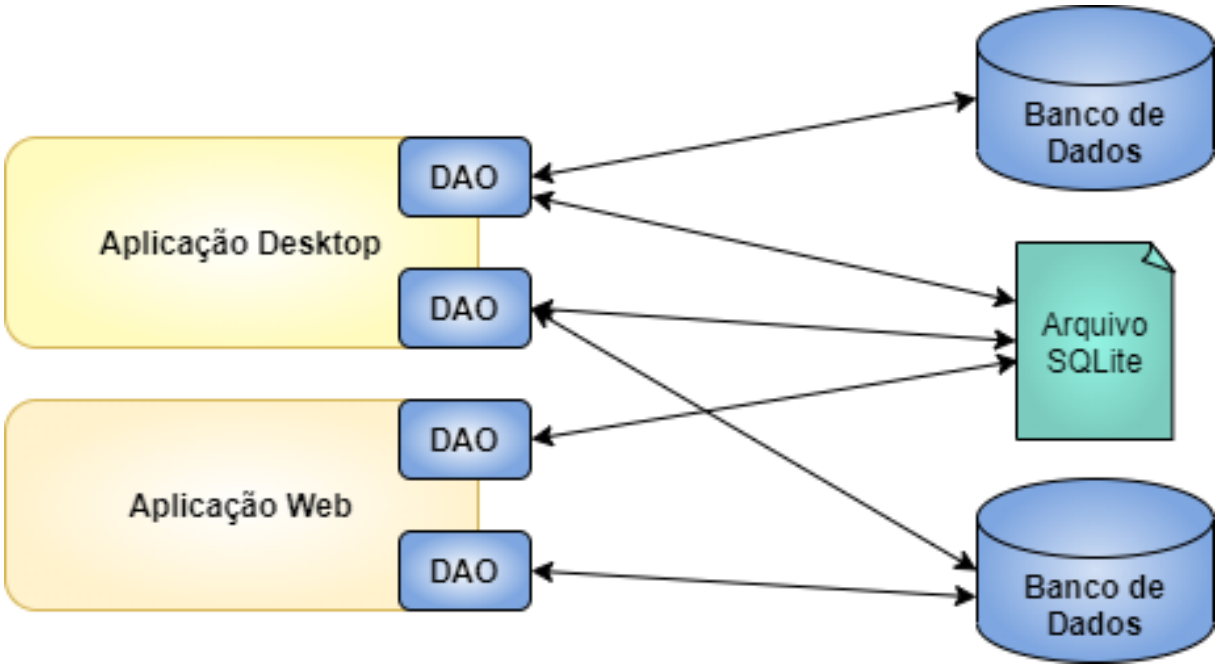
Para implementação do CRUD e segmentação adequada do código, foram-se aplicados os conceitos de DAO para acessar, ler e escrever os objetos.

4.7 DAO

O *Data Access Object*, que é usualmente abreviado para DAO, é um padrão que fornece uma abstração que cria uma interface entre o sistema e um banco de dados ou outro mecanismo de persistência. O DAO fornece operações para o programador sem expor detalhes do banco de dados que está sendo utilizado, aumentando assim a segurança do projeto e a facilidade na troca de banco de dados, pois a abstração será mantida e somente a conexão será alterada na maior parte dos casos.

Além de proteger os dados e conectar as aplicações com os mecanismos de persistência, o DAO também permite que uma aplicação utilize diferentes fontes de dados, como mostrado na figura a seguir.

Figura 4.1: DAO e fontes de dados.



Fonte: Criada pelo Autor

5

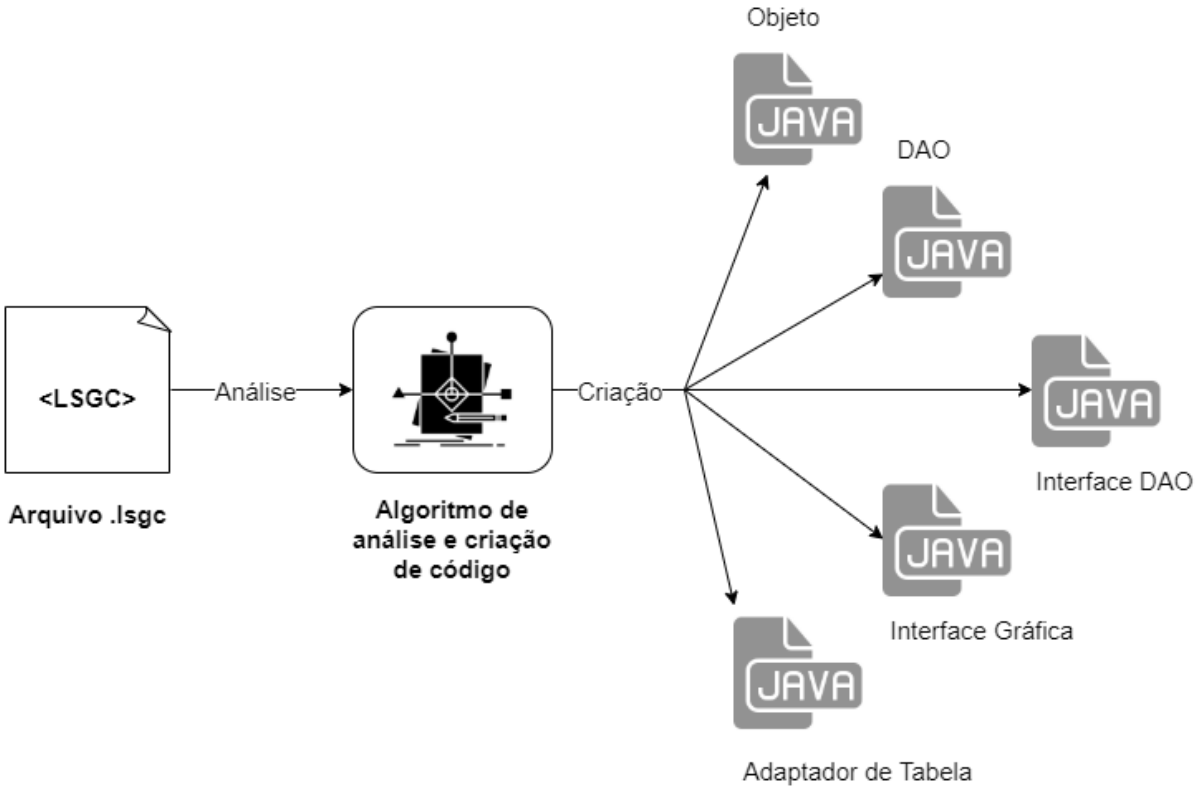
Desenvolvimento

Neste capítulo serão apresentadas as informações importantes relacionadas a construção semântica da LSGC que servirá como entrada para a geração do código e do desenvolvimento do algoritmo que irá analisar a LSGC criada e gerar código em Java considerando os dados escritos na LSGC.

5.1 Visão Geral

O processo elaborado por esse trabalho é dividido em três grandes partes: A LSGC e a sua semântica, o algoritmo de leitura da LSGC e as classes geradas por esse algoritmo.

Figura 5.1: Processo detalhado de criação de classes.



Fonte: Criada pelo Autor

A primeira parte do processo se encontra na escrita do arquivo LSGC onde se é definido o nome do pacote a ser criado e também a classe a ser construída, os seus atributos e os seus métodos. A LSGC criada neste trabalho dá suporte a duas línguas naturais, o inglês e o português, e ela utiliza somente vírgulas e pontos para separar expressões, considere o exemplo abaixo.

Figura 5.2: Exemplo em português da LSGC.

```
DEFINA DOMINIO Empresa.

DEFINA TIPO Empresa
COM cnpj COMO texto, nome_da_empresa COMO texto
OPERACOES crud.
```

Fonte: Criada pelo Autor

Figura 5.3: Exemplo em inglês da LSGC.

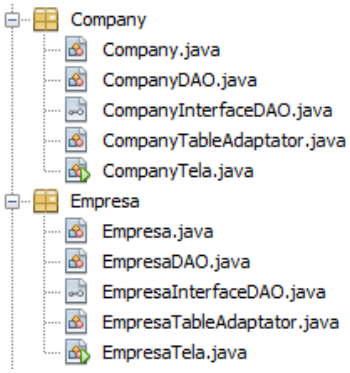
```
DEFINE NAMESPACE Company.

DEFINE TYPE Company
WITH cnpj AS text, company_name AS text
OPERATIONS crud.
```

Fonte: Criada pelo Autor

A segunda parte do processo se dá no algoritmo de análise e geração de código que lê os dados escritos na LSGC, os interpreta, e gera código Java funcional. Esse algoritmo escrito em Java é composto por sete classes, a classe principal que analisa o arquivo com a extensão LSGC, cinco classes que vão moldar o código final, e uma classe para a geração dos arquivos depois que todo o código foi escrito.

A terceira e última parte do processo é a criação dos arquivos e do pacote feita por uma das classes do algoritmo.

Figura 5.4: Classes geradas pelo algoritmo.



Fonte: Criada pelo Autor

5.2 Linguagem Semi-natural para Geração de Código (LSGC)

A LSGC é uma linguagem semi-natural de fácil aprendizado e que permite que profissionais que possuem ou não uma formação na área de desenvolvimento de sistemas possam entender o que está sendo escrito e aprender a utilizá-la rapidamente. Por ser uma linguagem semi-natural, ela só contém dois símbolos, a vírgula e o ponto final, isso é para prover que a escrita nessa linguagem ocorra da forma mais natural possível, também é possível escrever toda a expressão em somente uma linha, em estilo de frase, finalizando com um ponto final sem restrições de escrita em caixa alta ou em caixa baixa.

Para entender como a LSGC funciona é necessário entender a sua semântica, a sua sintaxe, como ela funciona e como o algoritmo irá analisar as suas informações. Os comandos da LSGC seguem uma ordem específica e são mostrados de um a um com uma explicação de suas funções.

5.2.1 Palavra-chave DEFINA

A palavra-chave DEFINA (ou em inglês *DEFINE*) é o primeiro comando de qualquer expressão e deve ser utilizado logo no começo ou após um ponto final para escrever duas expressões em um único arquivo.

Posterior ao comando *Define* vem o comando que irá dizer o que exatamente será definido naquela expressão que pode ser tanto um *Namespace* quanto um *Type*.

5.2.2 Palavra-chave DOMINIO

A palavra-chave DOMINIO (ou em inglês *NAMESPACE*) define em qual pacote Java as classes serão salvas e antecede a palavra a ser usada como nome do pacote, tudo que é gerado em um arquivo LSGC fica em um único domínio e se o domínio for declarado múltiplas vezes em um único arquivo, o último nome irá subscrever os outros.

Após inserir o nome do domínio a expressão deve ser encerrada com um ponto final.

Figura 5.5: Exemplo da palavra-chave *NAMESPACE* em ambos os idiomas.

```
DEFINA DOMINIO Casa.

DEFINE NAMESPACE House.
```

Fonte: Criada pelo Autor

Quando o domínio é definido múltiplas vezes no em um único arquivo, a última definição irá sobrescrever todas as chamadas anteriores.

Se um arquivo não possuir nenhuma expressão para alterar o nome do domínio, ele terá o nome padrão “Resultados”.

5.2.3 Palavra-chave TIPO

Já a palavra-chave TIPO (ou em inglês *TYPE*) define qual será a classe Java a ser criada antecedendo o nome da mesma. Diferentemente da palavra-chave *Namespace*, a expressão não pode ser finalizada após colocar o nome do tipo, pois as informações sobre o tipo são necessárias.

5.2.4 Palavras-chave COM e COMO

A conjunção COM (ou em inglês *WITH*) é a palavra-chave que divide o nome do tipo e seus atributos. Já a palavra-chave COMO (ou em inglês *AS*) é uma palavra-chave que auxilia na leitura dos atributos de um tipo. Ambas trabalham com o auxílio de vírgulas para adicionar atributos a um tipo como mostrado no exemplo a seguir:

Figura 5.6: Exemplo das palavras-chave *WITH* e *AS* em ambos os idiomas.

```
DEFINA TIPO Escola
COM nome_da_instituicao COMO texto, quantidade_estudantes COMO inteiro,
cep COMO texto, diretor COMO texto.

DEFINE TYPE School
WITH institution_name AS text, student_count AS integer,
zip_code AS text, headmaster AS text.
```

Fonte: Criada pelo Autor

Ao começar a declarar os atributos, é necessário se usar a palavra-chave *With* e os atributos precisam ser escritos seguindo a seguinte ordem: Nome do atributo, palavra-chave *With* e o tipo do atributo. Além disso, é preciso separar os atributos entre vírgulas e terminar com um ponto final ou com a chamada da palavra-chave *Operations*. Os tipos de atributos que a LSGC criada por esse trabalho suporta são três:

- Text/Texto: Atributo para qualquer tipo de texto, equivalente a uma *String* em Java.
- Integer/Inteiro: Atributo para números inteiros, equivalente a um *int* em Java.
- Float/Decimal: Atributo para números decimais, equivalente a um *float* em Java.

A expressão pode ser finalizada após os atributos serem declarados criando somente uma classe objeto, ou ela pode ser continuada para que funções sejam atribuídas a esse tipo.

5.2.5 Palavra-chave OPERACOES

Uma palavra-chave opcional, mas extremamente importante para a criação de tipos, chamada de *OPERATIONS* em inglês, esta palavra-chave antecede as funções que esse tipo possui.

A única função implementada atualmente é a função CRUD, que são quatro funções em uma, a função de criar uma instância, deletá-la, atualizá-la e removê-la.

Figura 5.7: Exemplo da palavra-chave *OPERATIONS* em ambos os idiomas.

```
DEFINA TIPO Escola
COM nome_da_instituicao COMO texto, quantidade_de_estudantes COMO inteiro,
cep COMO texto, diretor COMO texto
OPERACOES CRUD.

DEFINE TYPE School
WITH institution_name AS text, student_count AS integer
zip_code AS text, headmaster AS text
OPERATIONS CRUD.
```

Fonte: Criada pelo Autor

5.3 Algoritmo de Geração de Código

Para que a implementação da LSGC ocorra, foi criado um algoritmo que analisa um arquivo LSGC e gera classes Java considerando os dados descritos no arquivo.

Este algoritmo é composto por sete classes, uma classe principal que faz a análise da LSGC e gerência o processo principal enviando as informações relevantes para as outras classes, cinco classes protótipo que funcionam como um esqueleto para as classes que vão ser geradas armazenando o código que vai ser gerado, e uma classe que irá gerar os arquivos Java com base no código armazenado pelas classes protótipo.

5.3.1 Classe Principal

A classe principal do algoritmo que coleta as informações analisadas no arquivo LSGC e com base nelas, ela cria instâncias das classes protótipo e as preenchem com as informações necessárias para elas poderem ser moldadas no código final, que então é criado por instâncias das classes de geração de arquivos.

Ela também filtra os dados, interpreta a pontuação usada no LSGC com o intuito de não deixar pontos e vírgulas se misturarem com os atributos e armazena alguns dados. Informações como o nome do domínio, o nome dos atributos e os tipos dos atributos ficam salvos nela assim que são lidos e após serem postos em listas, eles são passados para as classes protótipo para que a geração de código seja feita.

5.3.2 Classes Protótipo

Um conjunto de cinco classes criadas para serem o esqueleto do código a ser gerado, existe uma classe protótipo para cada arquivo que pode ser criado pelo algoritmo. Esse algoritmo contém as seguintes classes protótipo: “PrototipoClasse”, “PrototipoDAO”, “PrototipoInterfaceDAO”, “PrototipoTela” e “PrototipoAdaptadorTabela”.

Todas as classes protótipos têm um *ArrayList* chamado “code” que armazena todas as linhas de código que vão ser escritas no arquivo final, essas classes contêm várias funções que

são chamadas pela classe principal e elas adicionam mais linhas de código neste *ArrayList*, entre essas funções, três delas são encontradas em todas as classes protótipo chamadas de *define*, *endLine* e *printCode*, uma que representa o começo do código, outra que representa o final do código e uma última que transforma o *ArrayList* gerado em uma única String para escrevê-la no arquivo Java.

```
void define(PrototipoClasse s, String classname, String packagename){
    code.add("package " + packagename + ";\n");
    code.add("import java.util.ArrayList;");
    code.add("import java.util.UUID;\n");
    code.add("public class " + classname + " {");
    code.add("String id;");
    this.title = classname;
}

void endLine(){
    code.add("}");
}

public String printCode(PrototipoClasse s){
    String writtenCode = "";
    for(String i : s.code){
        writtenCode = writtenCode + i + "\n";
    }
    return writtenCode;
}
```

Listagem 5.1: Exemplo das funções essenciais da classe “PrototipoClasse”.

Além dos métodos *define*, *endLine* e *printCode*, todas as classes protótipo possuem métodos únicos que auxiliam a inserir o código no local correto na ordem em que são chamados após certas informações são passadas pela classe principal.

Todas as classes protótipo também recebem um atributo identificador único universal (UUID), do inglês Universally Unique Identifier, para possibilitar a distinção de objetos e assegurar que não haverá identificadores repetidos nos objetos criados.

Como nos exemplos a seguir, a classe “PrototipoClasse” contém os métodos *addVariable*, *addConstructor*, *addGetters* e *addSetters* que criam partes do código que vão ser posicionadas entre a parte do *define* e do *endLine* seguindo a ordem na qual elas são chamadas pela classe principal.

```
void addVariable(String variableName, String variableType){
    code.add(variableType + " " + variableName + ";");
    variableTypes.add(variableType); variableNames.add(variableName);
}

void addConstructor(){
    ArrayList<String> variables = new ArrayList();
    for(int i=0; i != variableTypes.size(); i++){
        variables.add(variableTypes.get(i) + " " + variableNames.get(i));
    }
    String variableList = String.join(", ", variables);
    code.add("public " + this.title + "(" + variableList + ") {");
    for(int i=0; i != variableNames.size(); i++){
        code.add("this." + variableNames.get(i) + " = " + variableNames.get
            (i) + ";");
    }
    code.add("this.id = UUID.randomUUID().toString();");
    code.add("}\n");
}

void addGetters(){
    code.add("public String getId() {");
    code.add("return this.id;");
    code.add("}\n");

    for(int i=0; i != variableTypes.size(); i++){
        code.add("public " + variableTypes.get(i) + " get" + variableNames.
            get(i).substring(0, 1).toUpperCase() + variableNames.get(i).
            substring(1) + "() {");
        code.add("return this." + variableNames.get(i) + ";");
        code.add("}\n");
    }
}

void addSetters(){
    for(int i=0; i != variableTypes.size(); i++){
        code.add("public void set" + variableNames.get(i).substring(0, 1).
            toUpperCase() + variableNames.get(i).substring(1) + "(" +
            variableTypes.get(i) + " " + variableNames.get(i) + ") {");
        code.add("this." + variableNames.get(i) + " = " + variableNames.get
            (i)+ ";");
        code.add("}\n");
    }
}
```

Listagem 5.2: Exemplo de funções únicas da classe “PrototipoClasse”.

5.3.3 Classe Arquivo

A classe Arquivo[1] é tanto a classe que lê o arquivo LSGC e passa as suas informações para a classe principal, tanto quanto a classe que cria os arquivos Java com base nas classes protótipo.

Ela é responsável por analisar cada palavra presente no arquivo LSGC independentemente de ter quebra de linha ou não, ela coleta cada palavra, uma por uma, através de comandos chamados pela classe principal para que a classe principal possa depois analisar a palavra coletada para interpretá-la. Para ler as palavras-chave é utilizado o seguinte método:

```
public String readString() {
    String next = null;

    try {
        this.checkEOF();

        String line = this.buffer[this.nextTokenLin];
        for (int i = this.primLin; i < this.contLin; i++)
            this.buffer[i] = null;
        this.buffer[0] = line;
        this.nextTokenLin = this.primLin = 0;
        this.contLin = 1;

        int i, size = line.length();
        for (i = this.nextTokenCol; i < size; i++)
            if (line.charAt(i) == ' ')
                break;

        next = line.substring(this.nextTokenCol, i);
        this.nextChar = i;
        this.findNext();
    } catch (IOException e) {
        throw new RuntimeException(e.toString());
    }

    return next;
}
```

Listagem 5.3: Função de leitura de palavras-chave pela classe "Arquivo".

Uma outra função dessa classe é a de criar uma saída para o código contido, e para exercer tal função a classe "Arquivo" pega o resultado gerado pela função *printCode* de uma classe protótipo, a transforma em sua própria saída representada pelo atributo "out" e cria os arquivos através do método *createOutput* descrito a seguir.

[1] A classe arquivo utilizada no algoritmo é uma adaptação da classe "Arquivo" criada por Emannuel Macêdo da Universidade Federal de Pernambuco para a disciplina de Algoritmos e Estrutura de Dados.

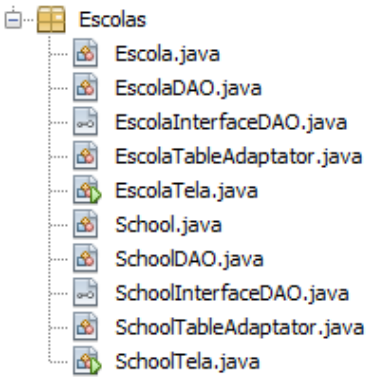
```
public void createOutput(String classname, String packagename) throws
    IOException{
     File f = new File("src/" + packagename);
     f.mkdir();
     try {
     this.out = new PrintWriter(new FileWriter("src/" + packagename + "/" +
         classname + ".java"), true);
     } catch (IOException e) {
          throw new RuntimeException(e.toString());
     }
}
```

Listagem 5.4: Função *createOutput* da classe "Arquivo".

5.4 Código Gerado e Interface Gráfica

O código gerado pela classe arquivo fica localizado em uma pasta com o nome do domínio localizada na fonte do projeto do algoritmo, a quantidade de classes varia conforme o que foi escrito na LSGC. Um tipo sem nenhuma operação resulta em somente uma classe Java com o nome do tipo que implementa os *getters* e *setters* de todos os atributos, já um tipo com as operações CRUD resulta em cinco classes, uma com o nome do tipo e quatro com o nome do tipo e os sufixos "-DAO", "-InterfaceDAO", "-Tela" e "-AdaptadorTabela".

Figura 5.8: Classes geradas pelo algoritmo da Figura 5.6.



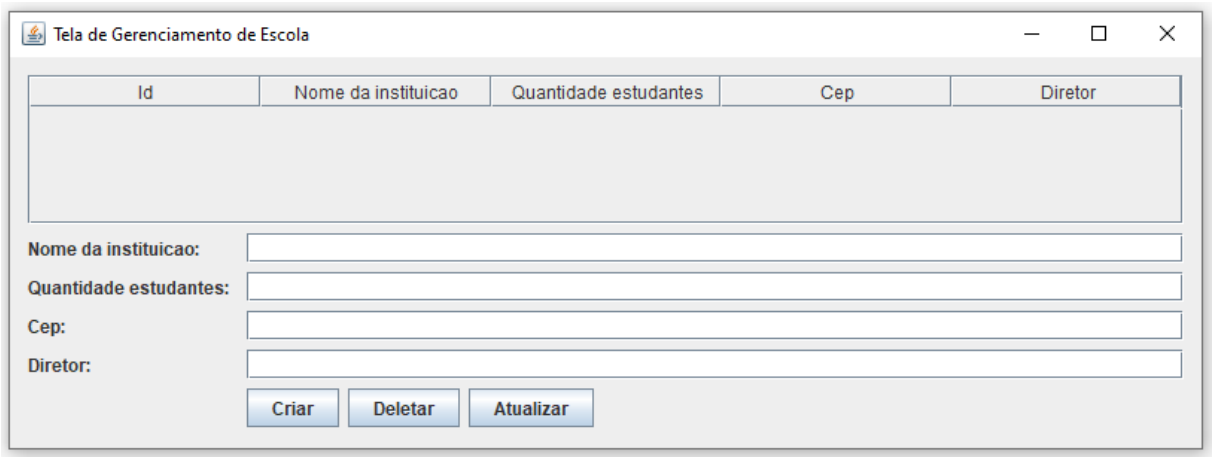
Fonte: Criada pelo Autor

```java
package Escolas;
import java.util.ArrayList;
import java.util.UUID;

public class Escola {
    String id;
    String nome_da_instituicao;
    String quantidade_estudantes;
    String cep;
    String diretor;

    public Escola(String nome_da_instituicao, String quantidade_estudantes, String cep, String
         diretor) {
        this.nome_da_instituicao = nome_da_instituicao;
        this.quantidade_estudantes = quantidade_estudantes;
        this.cep = cep;
        this.diretor = diretor;
        this.id = UUID.randomUUID().toString();
    }

    public String getId() {
        return this.id;
    }

    public String getNome_da_instituicao() {
        return this.nome_da_instituicao;
    }

    public String getQuantidade_estudantes() {
        return this.quantidade_estudantes;
    }

    public String getCep() {
        return this.cep;
    }

    public String getDiretor() {
        return this.diretor;
    }

    public void setNome_da_instituicao(String nome_da_instituicao) {
        this.nome_da_instituicao = nome_da_instituicao;
    }

    public void setQuantidade_estudantes(String quantidade_estudantes) {
        this.quantidade_estudantes = quantidade_estudantes;
    }

    public void setCep(String cep) {
        this.cep = cep;
    }

    public void setDiretor(String diretor) {
        this.diretor = diretor;
    }
}
```

Listagem 5.5: Classe “Escola” gerada pelo algoritmo.

A interface gráfica é gerada na classe com o sufixo “-Tela”, por isso ela somente está presente quando um tipo implementa uma operação, ao executar o método principal, uma tela como mostrada a seguir será construída.

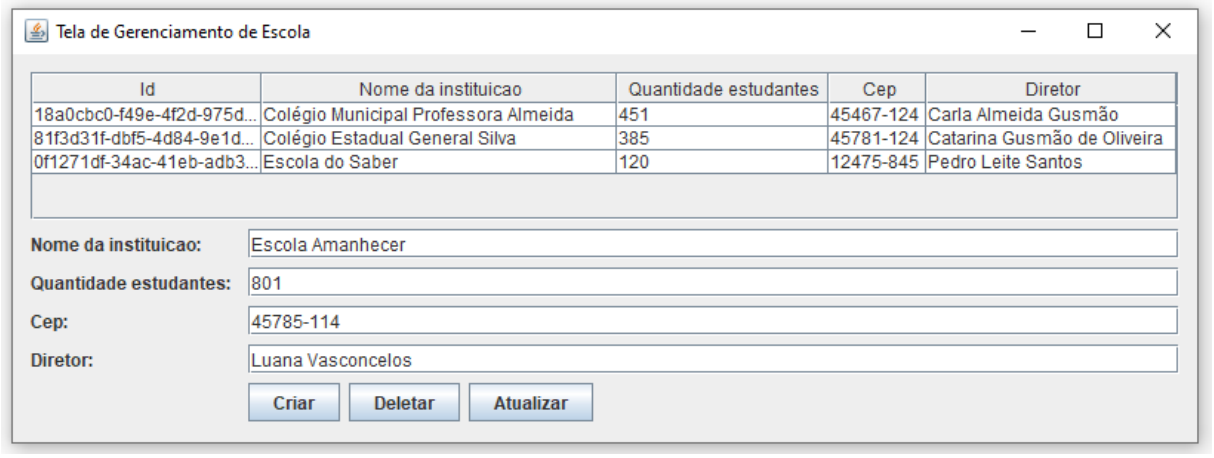
Figura 5.9: Tela gerada pelo sistema automaticamente.



Fonte: Criada pelo Autor

Nessa tela é possível executar às quatro funções CRUD através do uso da tabela no topo da tela, ao colocarmos informações nos campos de texto e clicarmos no botão criar, objetos daquela classe serão criados e exibidos na tabela como mostrado no exemplo a seguir:

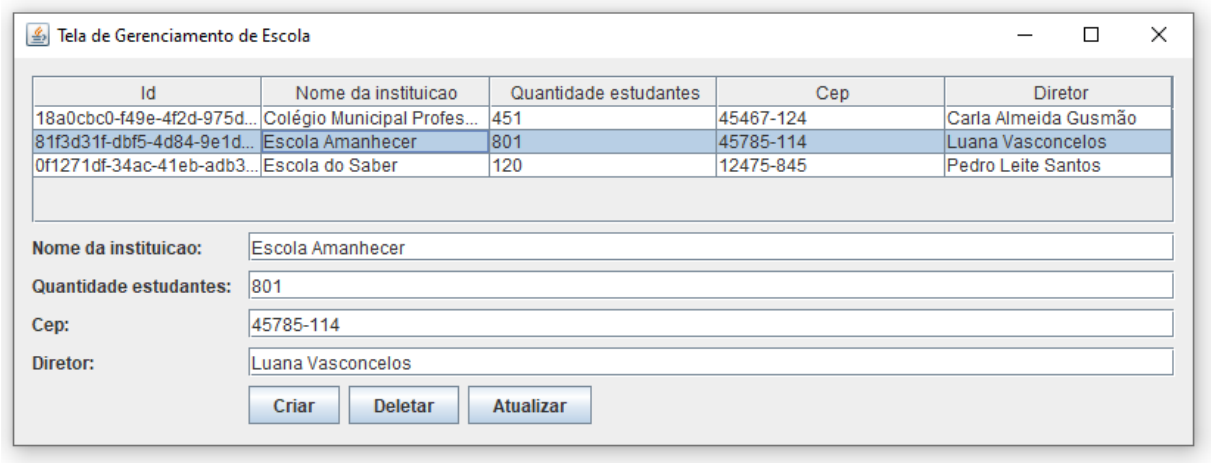
Figura 5.10: Tela gerada pelo sistema automaticamente com informações fictícias.



Fonte: Criada pelo Autor

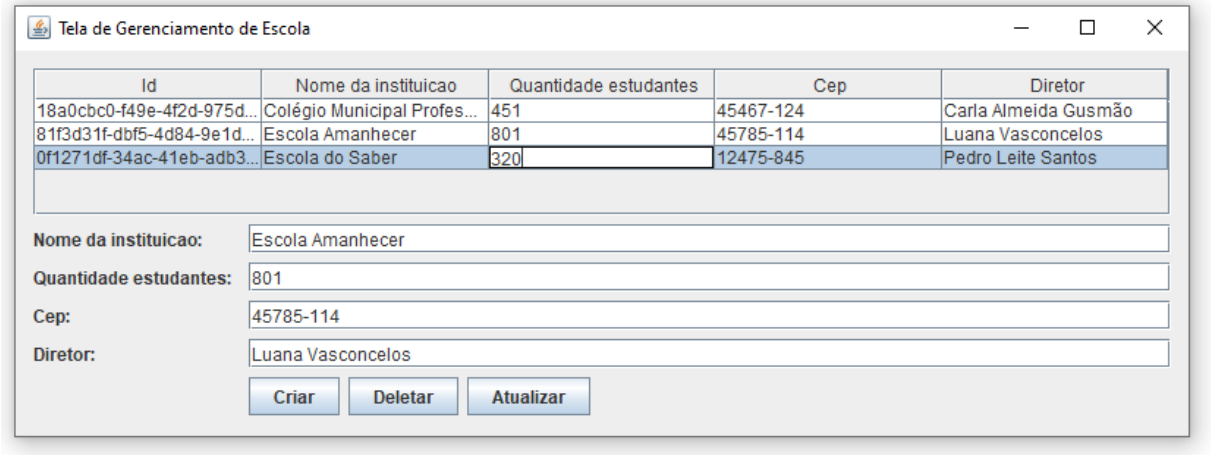
É possível também deletar e atualizar os elementos da tabela ao clicar na linha de um elemento e clicar no botão, ao deletar, a informação será excluída e ao atualizar, ela será sobrescrita pelos elementos no campo de texto mantendo seu ID único. A busca é feita em quase todas as operações executadas e ao atualizar a tabela, também é possível modificar os dados clicando num elemento da tabela e os editando diretamente, menos o ID, que é inalterável.

Figura 5.11: Modificando os elementos com funções da tela gerada pelo sistema automaticamente.



Fonte: Criada pelo Autor

Figura 5.12: Modificando os elementos diretamente da tela gerada pelo sistema automaticamente.



Fonte: Criada pelo Autor

Os nomes dos atributos aparecem na tela com a primeira letra sempre maiúscula e todos os caracteres "_" são interpretados como um clique da barra de espaço para possibilitar que o nome dos atributos possam conter espaços.

5.5 Estudo de Caso

O estudo de caso realizado teve a intenção de coletar dados sobre o desempenho e velocidade do algoritmo de geração de código levando as seguintes métricas quali-quantitativas a seguir:

- Tamanho de armazenamento do algoritmo de geração de código.
- Tempo médio gasto para geração do código.

- Tempo médio gasto para montagem da interface gráfica.
- Quantidade de erros devido a falhas do sistema.

Através dos dados obtidos por essas métricas podemos teorizar sobre a eficiência da utilização da linguagem LSGC criada para a aceleração no tempo do desenvolvimento de protótipos e sistemas através da criação de classes oferecida pela LSGC.

A máquina onde o estudo de caso foi realizado tem as seguintes configurações: Windows 10 Pro, Intel(R) Core(TM) i5-4590 e 12,0 GB RAM. A IDE de escolha para executar o algoritmo foi o Apache Netbeans.

O estudo de caso consistiu da criação de classes para um sistema de administração de despesas de uma empresa, os tipos necessários para a construção de sistemas foram estabelecidos com os seguintes nomes e atributos:

- Funcionário com nome, salário, função e idade.
- Equipamento com nome, custo e valor da manutenção.
- Imposto com nome e valor.

Como o sistema administra as despesas de uma empresa, o domínio será chamado de "Despesas", então temos o cenário a seguir. No domínio "Despesas" temos os seguintes tipos: Funcionário com os atributos nome como um texto, salário como um número decimal, função como um texto e idade como um número inteiro. Equipamento, com os atributos nome como um texto, custo como um valor decimal e o valor da manutenção como um valor decimal. E Imposto, com os atributos nome como um texto e valor como um número decimal. Como todos os tipos podem ser gerenciados, todos operam as operações CRUD.

Com esses tipos descritos acima em mente, o código a seguir foi escrito em LSGC:

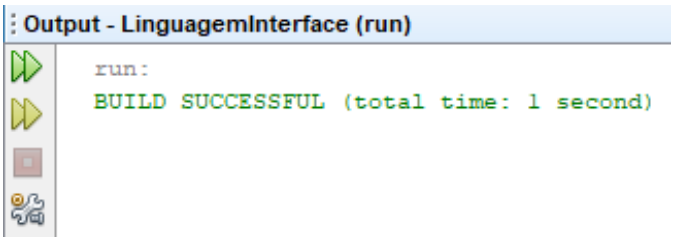
Figura 5.13: Código LSGC para o estudo de caso.

```
DEFINA DOMINIO Despesas.

DEFINA TIPO Funcionario
COM nome COMO texto, salario COMO decimal, funcao COMO texto, idade COMO inteiro
OPERAÇÕES CRUD.

DEFINA TIPO Equipamento
COM nome como texto, custo COMO decimal, valor_manutenção COMO decimal
OPERAÇÕES CRUD.

DEFINA TIPO Imposto
COM nome como texto, valor como decimal
OPERAÇÕES CRUD.
```

Fonte: Criada pelo Autor

Após executar o algoritmo para a leitura, análise e geração de código baseado no que estava escrito no arquivo LSGC, a IDE Apache Netbeans retornou o tempo gasto de aproximadamente 1 segundo.

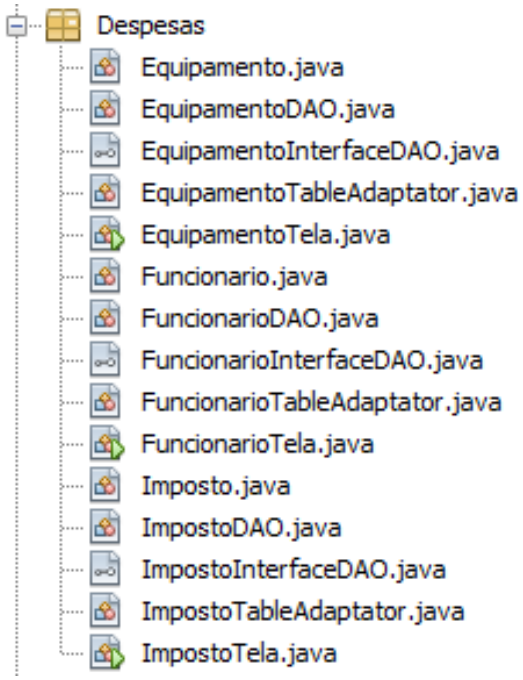
Figura 5.14: Tempo gasto pelo algoritmo retornado pela IDE do Apache NetBeans.



Fonte: Criada pelo Autor

Com o tempo aproximado de 1 segundo, o algoritmo foi capaz de criar as seguintes classes no pacote Java chamado “Despesas”, como descrito na LSGC onde a interface gráfica, o DAO e o CRUD já se encontram implementados no código:

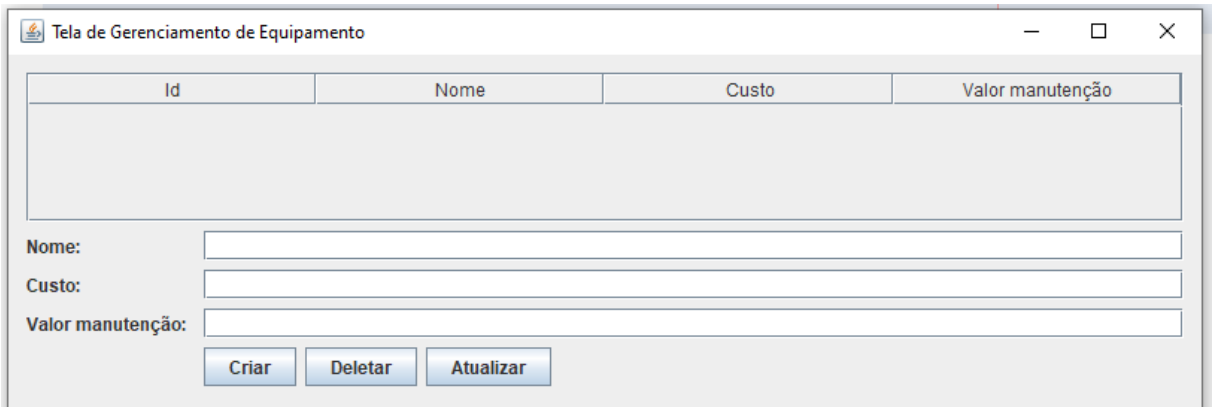
Figura 5.15: Classes criadas pelo algoritmo.



Fonte: Criada pelo Autor

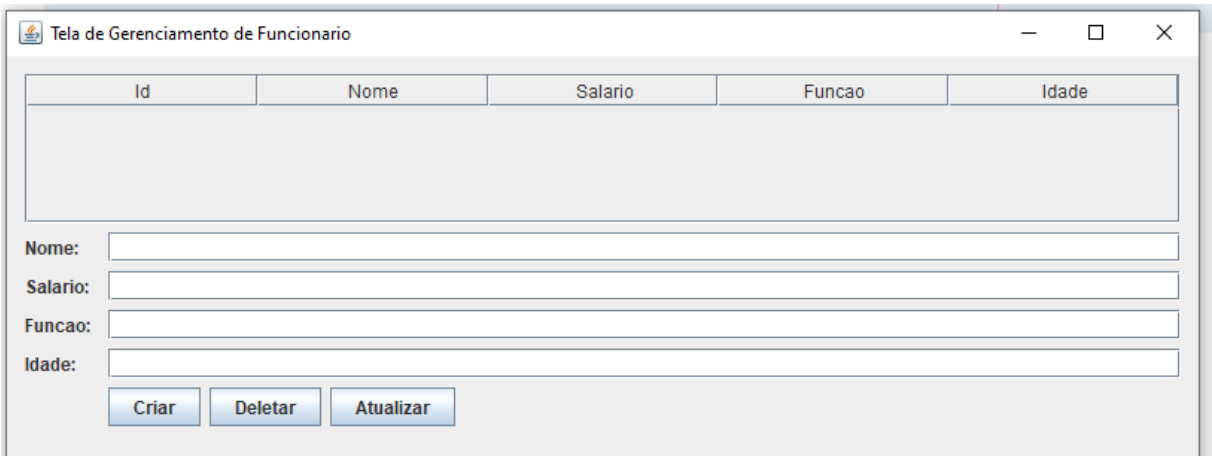
A quantidade de erros durante o estudo de caso foi de zero, pois a estrutura LSGC se encontrava correta e o código gerado não apresentou nenhum erro de compilação. O tempo de montagem das interfaces gráficas foi também de um segundo.

Figura 5.16: Tela de gerenciamento de equipamento gerada.



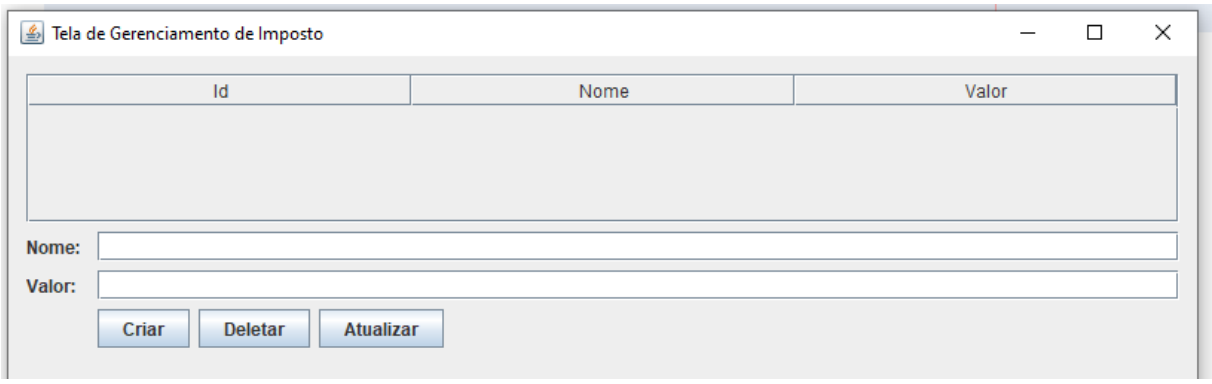
Fonte: Criada pelo Autor

Figura 5.17: Tela de gerenciamento de funcionário gerada.



Fonte: Criada pelo Autor

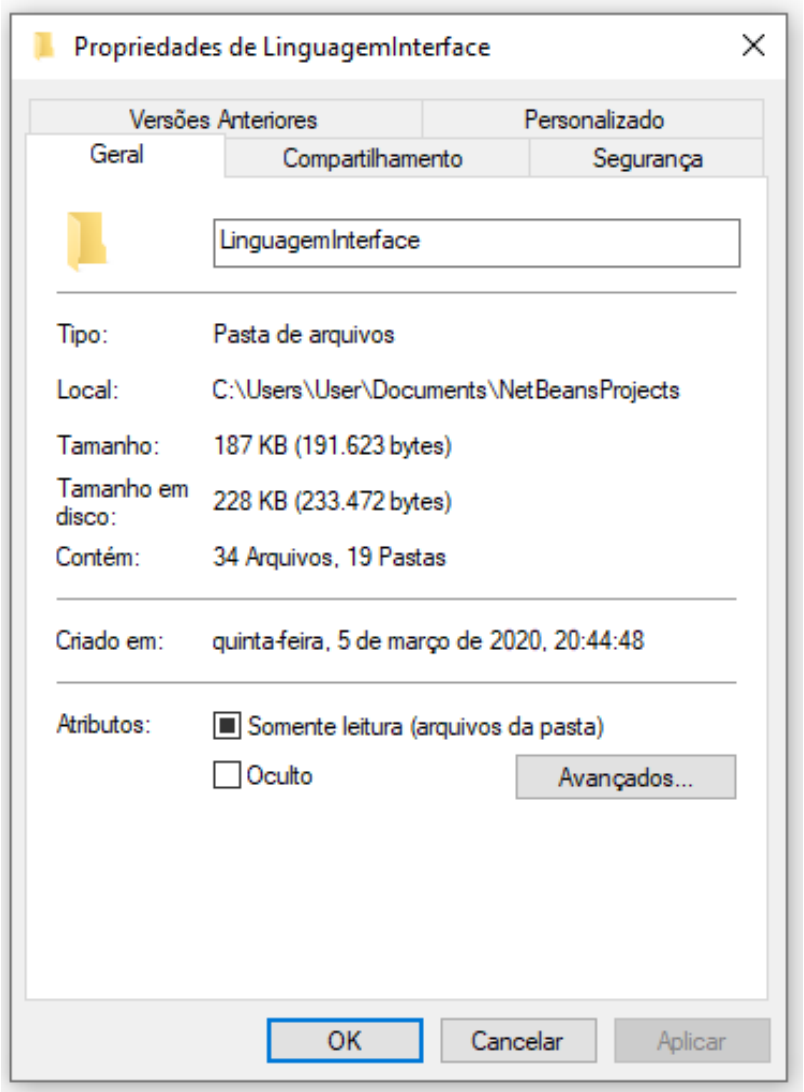
Figura 5.18: Tela de gerenciamento de imposto gerada.



Fonte: Criada pelo Autor

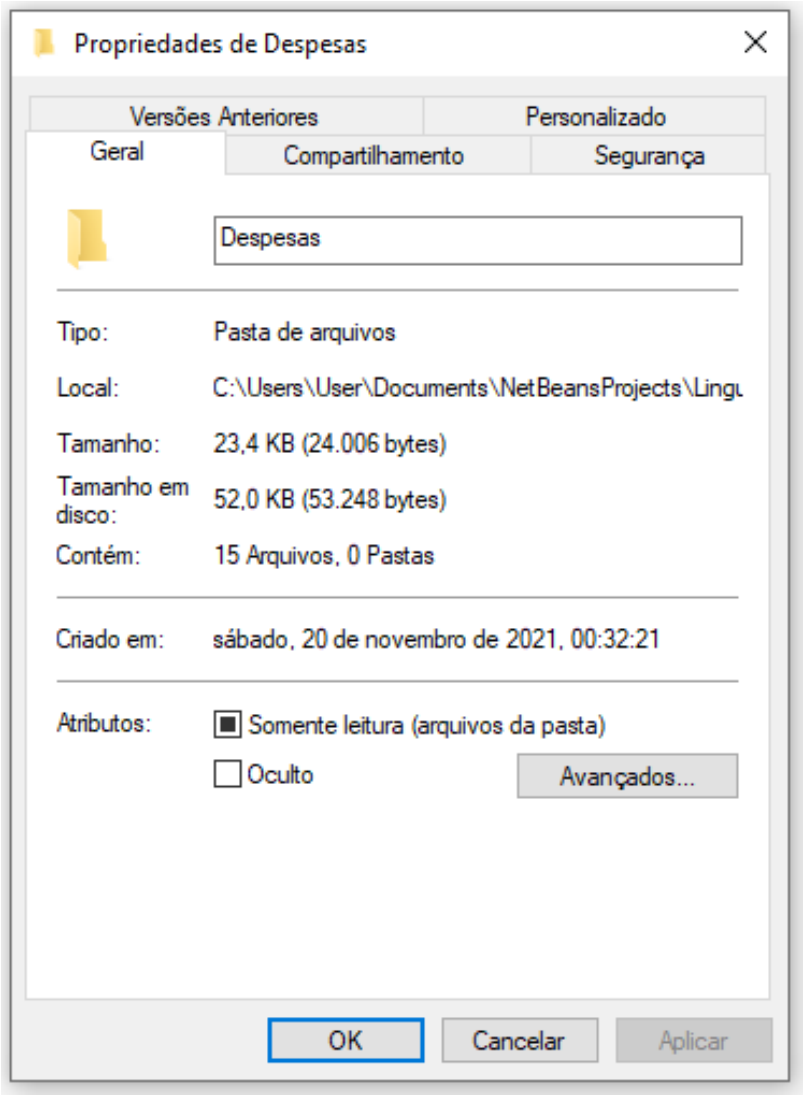
Em relação ao tamanho dos arquivos, o algoritmo de geração de código ocupa 187 KBs no total sem nenhuma compactação, já o pacote Java gerado ocupa 23.4 KBs de memória com todas as 15 classes geradas.

Figura 5.19: Tamanho do algoritmo de geração de código.



Fonte: Criada pelo Autor

Figura 5.20: Tamanho do pacote Java gerado.



Fonte: Criada pelo Autor

Ocupando pouco espaço em disco e sendo bastante veloz em sua execução, o algoritmo de geração de código mostra ótimos resultados de desempenho, e devido a versatilidade de programas codificados em Java, essa ferramenta pode ser executada em qualquer ambiente que possua os programas necessários para interpretar a linguagem Java.

6

Conclusão

Com o aumento no tamanho e complexidade dos sistemas criados nos dias atuais, se tornam cada vez mais necessárias as ferramentas para acelerar o processo de desenvolvimento de software. Com o intuito de acelerar esse processo com interfaces gráficas funcionais, a LSGC permite que os desenvolvedores foquem nas partes mais complexas do projeto ao gerar o código mais simples dos sistemas de forma eficaz, padronizada e veloz.

O objetivo geral foi desenvolver um algoritmo de geração de código que fabrique código capaz de produzir interfaces gráficas funcionais em Java, com a estruturação da LSGC durante o desenvolvimento do projeto, e de sua leitura e análise por um algoritmo que foi desenvolvido e serviu como protótipo para o estudo de caso presente neste trabalho, ambos os objetivos gerais e específicos foram alcançados ao longo do projeto.

A LSGC desenvolvida neste trabalho teve como prioridade ser fácil de se aprender e entender para que ela possa ser usada por desenvolvedores em diferentes níveis e áreas com facilidade e agilidade, gerando códigos capazes de gerar interfaces gráficas e assim acelerando o processo de prototipagem e desenvolvimento de software, com tal intuito o algoritmo criado para ler e interpretar a LSGC teve como prioridade ser veloz, leve e versátil para que ele possa agilizar o processo de desenvolvimento de software independentemente da máquina que ele se encontra.

O estudo de caso efetuado neste trabalho teve resultados positivos em relação à velocidade do algoritmo de interpretação da LSGC, levando somente 1 segundo para gerar um total de 15 classes com diversas funções e implementações que podem até mesmo gerar uma interface gráfica totalmente funcional que executa as quatro operações CRUD, já em relação à facilidade de aprendizado da LSGC, não foi possível de fazer uma pesquisa de campo com outros usuários devido ao contexto da pandemia gerada pelo COVID-19.

Devido à natureza semi-natural da linguagem e de uma menor quantidade de palavras para se gerar código, é possível deduzir que o tempo gasto na escrita em LSGC é muito menor comparado a escrever todo o código equivalente ao que é gerado pelo algoritmo, uma possibilidade de pesquisa futura é pesquisar com desenvolvedores e através da observação não-participante, coletar dados relacionados a dificuldades encontradas pelos usuários durante o processo.

Outras possibilidades de pesquisas futuras em relação a este projeto são de ampliar a quantidade de métodos e funcionalidades nas interfaces gráficas geradas pela LSGC e de estudar a sua eficiência em empresas, calculando o tempo economizado pelo seu uso ao longo prazo. Também de aprimorar as interfaces gráficas geradas com validadores e melhorar a usabilidade do programa.

Referências

ADRION, W. R.; BRANSTAD, M. A.; CHERNIAVSKY, J. C. Validation, Verification, and Testing of Computer Software. **ACM Comput. Surv.**, New York, NY, USA, v.14, n.2, p.159–192, June 1982.

AHO, A. V. et al. **Compiladores: princípios, técnicas e ferramentas**. 2st.ed. [S.l.: s.n.], 2007.

BALZER, S. Contracted persistent object programming. **Swiss Federal Institute of Technology Zürich, Tech. Rep**, [S.l.], 2005.

DEMASTER, J. D. **Java native interface code generator**. [S.l.]: U.S. Patent n. 6,066,181, 2000.

HARRISON, W.; BARTON, C.; RAGHAVACHARI, M. Mapping UML designs to Java. **Sigplan Notices - SIGPLAN**, [S.l.], v.35, p.178–187, 10 2000.

JUNGTHON, G.; GOULART, C. M. Paradigmas de Programação. **Monografia (Monografia)—Faculdade de Informática de Taquara, Rio Grande do Sul**, [S.l.], v.57, 2009.

LOH, S. O uso de uma linguagem semi-formal no processo de formalizacao de especificacoes de software. **In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 5., 1991, Ouro Preto. Anais... Belo Horizonte: UFMG**, [S.l.], 1991.

LOY, M. et al. **Java Swing, Second Edition**. 2.ed. [S.l.]: O'Reilly Media, Inc., 2002.

LYONS, J. **Natural Language and Universal Grammar**: essays in linguistic theory. [S.l.]: Cambridge University Press, 1991. v.1.

MENDES, D. R. **Programação Java com Ênfase em Orientação a Objetos**. [S.l.]: NOVATEC, 2009.

NAUGHTON, P.; SCHILDT, H. **Java: The Complete Reference**. 1st.ed. USA: McGraw-Hill, Inc., 1996.

NETO, J. E. V. S. M. **Geração automática de código para padrões de conceção**. 2011.

SANTOS, R. et al. PROGLIB: uma linguagem de programação baseada na escrita de libras. **Anais do Workshop de Informática na Escola**, [S.l.], v.1, n.1, p.1533–1542, 2011.

SOMMERVILLE, I. **Engenharia de software**. [S.l.]: Pearson Prentice Hall, 2011.

STACK OVERFLOW. **Developer Survey Results 2018**. Disponível em: https://insights.stackoverflow.com/survey/2018/most-popular-technologies. Acesso em: 29 de nov. de 2019.

TURINE, M. A. S.; MASIERO, P. C. et al. Especificação de requisitos: uma introdução. , [S.l.], 1996.

YODER, J. W. et al. **Connecting Business Objects to Relational Databases**. [S.l.]: Conference on the Pattern Languages of Programs, 1998.