CONGRESSO CATARINENSE DE CIÊNCIA DA COMPUTAÇÃO
2019. Ano 6, Vol. 6. ISSN: 2319-0655

AutoFX – Framework geradora de interface gráfica JavaFx

William Gabriel Pereira[1], Rodrigo Curvêllo[2]
[1]Aluno da ciência da computação no IFC
[2]Professor e mestre de ciência da computação
will.gabriel.pereira@gmail.com, rodrigo.curvello@ifc.edu.br

Abstract. *Thinking in the difficulty, out time and generalization to build an GUI for management to objects, was constructed a framework called AutoFX, where it is possible to send any class meets the requirements from framework and it manufactures a interface totally intuitive, with options to save, change and delete any object already saved within the list, also generating a JSON file containing the objects listed by the framework. With it you can avoid wasting time by building a system that needs to manage simple objects.*
Key-words: *Framework; JavaFX; Persistence, Generator.*

Resumo. *Pensando na dificuldade, falta de tempo e generalização para construir uma interface de gerenciamento de objetos, foi construída a framework AutoFX, onde é possível enviar com qualquer classe atenda aos requisitos da framework e ela fabrica uma interface totalmente intuitiva, com opções de salvar, alterar e excluir qualquer objeto já salvo dentro da lista, gerando também um arquivo JSON contendo os objetos listados pela framework. Com ela é possível evitar desperdício de tempo ao construir um sistema que necessite gerenciar objetos simples.*
Palavras-chave: *Framework; JavaFX; Persistencia, Gerador.*

1. Introdução

Frameworks então a cada dia mais presentes na vida de programadores, podendo elas, resolver vários problemas e servindo de soluções para muitos sistemas, evitando o retrabalho de programadores. Uma framework pode servir um simples sistema de persistência em "*.csv",* até um sistema completo de inteligência artificial.

JavaFX é uma biblioteca gráfica, ou seja, é usada para fornecer sistemas com GUI (Gaphical User Interface, interface gráfica de usuário). JavaFX é a mais nova biblioteca gráfica oficial para o java, com a inovação de não ser necessário compilar seu sistema em cada JVM para que funcione.

A ideia da framework AutoFX é facilitar na criação de sistemas gerenciadores de classes simples (DTOs), ou seja, deve ser uma framework que gera uma interface gráfica onde se possa gerenciar qualquer classe que o usuário (programador) passe para ela, podendo também fazer a persistência de seus dados, ou seja, gravar e buscar dados salvos utilizando o sistema de persistência *JSON*.

2. Metodologia

Foram usados neste trabalho algumas ferramentas, linguagens e conceitos. Framework baseada na linguagem de programação Java, utilizando recursos Java Reflection, JavaFX e Gson, este último disponibilizado pela Google. Alguns dos conceitos abordados na framework são os ponteiros, final class, Singleton e Generic wild card.

2.1. Java

Java é uma linguagem de programação muito utilizada, por sua arquitetura e paradigma de programação utilizada, que no caso é a orientação a objetos, também muito utilizada hoje em dia. Além do paradigma, o java possui uma VM, ou seja, uma "máquina virtual" por assim dizer, onde existe uma JVM para cada tipo de sistema operacional (ou até mesmo não operacional), como, por exemplo, para o Windows, Linux e até mesmo aplicações que devam funcionar em BIOS, essa flexibilidade torna o Java muito requisitado e amado por programadores.

Java é uma linguagem compilada, ou seja, torna o código fonte inacessível, fazendo assim também, um sistema mais rápido quando comparado com algumas linguagens interpretadas, pois não é necessário um interpretador no caminho, porém por ser uma linguagem que funciona em VM, ela acaba sendo um pouco lenta para alguns casos mais extremos e pouco utilizada quando falamos de tratar muitos dados.

Segundo Bastos(2010), a compilação é o processo que reúne o código fonte e o transforma em algo que a máquina consiga entender. Ele também afirma que do ponto de vista do código fonte, qualquer linguagem é compilada, tendo em vista que a programação nunca é escrita de forma "*natural*" para os humanos.

Foi utilizada esta linguagem por ser a mais conhecida pelo autor do trabalho e pela facilidade de implementar alguns conceitos da programação orientada a objetos, também foi utilizada Java pois torna-se impossível criar uma ferramenta que gera interface Java em outra linguagem.

2.2 Orientação a Objetos

Orientação a objetos (POO), é um paradigma da programação onde se busca descrever o objeto a ser usado em forma de programação, tendo ele seus atributos e funções próprias.

Segundo Welson(2015), a definição de POO foi baseada em três pilares básicos, são eles: encapsulamento que dita como está disponível tal atributo ou método, herança onde um objeto pode herdar os mesmos atributos e funções que sua classe "pai" e polimorfismo, que consiste em poder tratar como um objeto mais genérico um objeto mais específico.

Já Noronha(2018), diz que temos mais um pilar, a abstração, que pode ser confundido com o modificador "*Abstract*", mas na verdade ele fala sobre saber aquilo que é necessário para o sistema, como, por exemplo, não é necessário saber o modelo do seu computador em um sistema hospitalar.

2.3 JavaFX

JavaFX é uma biblioteca gráfica, é usada para a criação de aplicações RIA (*Rich Internet Application*) que se comportam de maneira consistente em todas as plataformas segundo Menezes(2015). Ou seja, JavaFX é usado para construção de interfaces gráficas, onde, não é necessário compilar, muito menos construir, o sistema separadamente para cada SO, amenizando o tempo de produção de um sistema.

Utilizando a linguagem de marcação baseada no XML, o FXML é usado para construir e customizar a interface, porém é possível utilizar um criador gráfico, chamado SceneBuilder, um aplicativo que gera o FXML e seus controladores usando a intuitividade de um sistema gráfico, possibilitando que o programado veja na hora as modificações.

Neste projeto, foi construído um arquivo FXML padrão que gera a Stage de fundo, apenas pedindo ao programador a Stage a qual ele deseja plotar a interface. Pelo fato de ser obrigatória a inicialização do Stage por uma aplicação JavaFX, foi impossível deixar que o programador apenas enviasse a classe que deseja, forçando-o a criar uma aplicação JavaFX onde, dentro poderá utilizar da framework.

2.4 Java Reflection

"O Reflection, em poucas palavras, serve para determinar métodos e atributos que serão utilizados de determinada classe (que você nem conhece) em tempo de execução." (LANHELLAS, 2013).

Usado para descobrir atributos e métodos, o Reflection, é usado para que se torne possível utilizar uma classe da qual você nada sabe, nem mesmo seu nome. Apesar de se saber todos os seus atributos e funções, torna-se difícil utilizá-las quando uma classe é muito complexa e não se sabe o que especificamente tal método faz, ainda mais quando nem se sabe o seu próprio nome.

Mesmo com nomes e funcionalidades não conhecidas, algumas funções, quando uma classe é implementada de forma correta, algumas funções padrões vem com nomes muito parecidos, independente de quem fez a classe ou onde, como, por exemplo, os *getters* e *setters,* sabendo disso, é possível fazer o gerenciamento do estado de um objeto sem saber como ele é, apenas sabendo que ele vai possuir os *getters* e *setters.*

3. Resuldados e Discussões

Exceto a funcionalidade de buscar arquivo de objetos salvos, todos os objetivos foram alcançados. Utilização de classes simples, permitir gerenciamento pelo usuário final e salvar em arquivo JSON.

Para a descoberta de atributos e invocação de funções, existe uma classe que utiliza das funcionalidades do Java Reflection.

```
1  private Class<?> classe;
2  private Object objeto;
```

```
3  private String nomeClasse;
4  private Field[] atributos;
```

Acima estão os atributos utilizados pela classe, sendo eles, da linha 1, a classe que será passada pelo usuário da framework, os outros atributos são todos calculados a partir da classe recebida.

```
1  private Reflection(Class<?> classe) {
2        this.classe = classe;
3        String nome[] = classe.getName().replace('.', ' ').split(" ");
4        this.nomeClasse = nome[nome.length - 1];
5        try {
6           objeto = classe.newInstance();
7        } catch (InstantiationException | IllegalAccessException e) {
8           e.printStackTrace();
9        }
10       this.setAtributos();
11    }
```

Acima temos o construtor, percebe-se que ele é privado, pois esta classe está usando o padrão Singleton.

As linhas 3 e 4 estão apagando o nome do package em que está a classe, pois quando usada a função *Class.getName(),* junto com o nome, é capturada sua localização dentro do projeto.

Para a invocação do atributo, foi feita uma função que pudesse chamar qualquer método get ou set da classe, sabendo que essas funções apenas utilizam o prefixo (set ou get) e então o nome do atributo com capitulação, foi feito uma espécie de gerador de nome de método, criando o nome como, por exemplo, usar uma função “set”, para o atributo “nome”, gera-se o seguinte nome de função: “setNome”.

```
1  String pre = "";
2        switch (setGet) {
3           case 0:
4              pre = "set";
5              break;
6           case 1:
```

```
7               pre = "get";
8               break;
9           default:
10              break;
11      }
12      String palavra = atributo.getName();
13      palavra = pre + palavra.substring(0, 1).
14              toUpperCase().concat(palavra.substring(1));
15      if (setGet == 0) {
16          switch (atributo.getType().getName()) {
17              case "int":
18                  break;
19          }
20      }
21
```

Foi utilizado para esse gerador de nome, um switch, que verifica qual constante foi recebida e então é salvo em uma variável o seu prefixo, então é feita a capitalização do atributo e concatenado com o prefixo.

```
1   Method metodo;
2   if (parametro == null) {
3           metodo = classe.getMethod(palavra, null);
4           return metodo.invoke(objeto, null);
5   }else{
6           metodo = classe.getMethod(palavra, parametro.getClass());
7           return metodo.invoke(objeto, parametro);
8   }
```

Após ter o nome da função pronta, é possível fazer a chamada de tal função, encontrando tal método enviando o nome e o tipo de retorno, logo depois é possível fazer a invocação do método, passando o objeto de qual será invocada a função e os seus parâmetros.

Tendo a classe que faz a invocação de métodos e guarda o objeto, já é possível criar então uma interface a partir do que conseguimos descobrir com essa classe. Para a criação de interfaces, temos uma classe que faz este trabalho, a classe *CriadorInterface*, utilizando também Singleton para a sua instância.

Seu construtor é bem simples, fazendo apenas a chamada do FXML pré-pronto. Para a criação das funcionalidades que permitem a gerência do objeto, é necessário saber seus atributos, sabendo a quantidade e nome de atributos, basicamente são criados os *Labels* e *TextFields* de acordo com a classe e plotados dentro da interface

```
1   for (Field f : atributos) {
2           // CRIA LABEL
3           Label lb = new Label();
4           lb.setPrefHeight(19);
5           lb.setPrefWidth(232);
6           lb.setLayoutX(layoutXLabel);
7           lb.setLayoutY(layoutY);
8           lb.setFont(fonte);
9           lb.setText(f.getName());
10          // CRIA TEXTFIELD
11          TextField tf = new TextField();
12          tf.setPrefHeight(25);
13          tf.setPrefWidth(249);
14          tf.setLayoutX(layoutXText);
15          tf.setLayoutY(layoutY);
16          tf.setPromptText(f.getName());
17          // ADICIONA TF COMO CAPTURADO PELO LABEL
18          lb.setLabelFor(tf);
19
20          // AUMENTA DISTANCIA VERTICAL
21          layoutY += 33;
22
23          // ADICIONA LB E TF NO PANE DO SCROLL
24          painelCad.getChildren().add(lb);
25          painelCad.getChildren().add(tf);
26
27          fields.add(tf);
28  }
```

Para pegar os dados no entanto, torna-se um pouco mais complicado, pois a tipologia de um *TextField* é sempre *String*, mas nem sempre queremos um *String*, então

se faz necessário um cast, mas como fazer um cast sem saber qual tipo de dado queremos? Usamos os parsers para cada *TextField* que temos. Segue o código:

```
1   switch (atributos[i].getType().getName()) {
2           default: reflection.invokeParaAtributo(atributos[i],
3                                   Reflection.SET,fields.get(i).getText());
4                   break;
5           case "int":
6                   int t = Integer.parseInt(fields.get(i).getText());
7                   reflection.invokeParaAtributo(atributos[i], Reflection.SET, t);
8           break;
9           case "float":
10                  float f = Float.parseFloat(fields.get(i).getText());
11                  reflection.invokeParaAtributo(atributos[i], Reflection.SET, f);
12          break;
13          case "double":
14                  double d = Double.parseDouble(fields.get(i).getText());
15                  reflection.invokeParaAtributo(atributos[i], Reflection.SET, d);
16          break;
17          case "boolean":
18                  boolean b = Boolean.parseBoolean(fields.get(i).getText());
19                  reflection.invokeParaAtributo(atributos[i], Reflection.SET, b);
20          break;
21          case "char":
22                  char c = fields.get(i).getText().charAt(0);
23                  reflection.invokeParaAtributo(atributos[i], Reflection.SET, c);
24          break;
25          case "byte":
26                  byte by = Byte.parseByte(fields.get(i).getText());
27                  reflection.invokeParaAtributo(atributos[i], Reflection.SET, by);
28          break;
29          case "short":
30                  short s = Short.parseShort(fields.get(i).getText());
31                  reflection.invokeParaAtributo(atributos[i], Reflection.SET, s);
32          break;
33          case "long":
```

```
34                  long l = Long.parseLong(fields.get(i).getText());
35                  reflection.invokeParaAtributo(atributos[i], Reflection.SET, l);
36              break;
37      }
```

Na nossa classe “principal” da framework, ou seja, aquela que o usuário deverá utilizar para fazer o uso das funcionalidades, temos também outro Singleton. Todas as classes Singleton foram pensadas nesse jeito para que não houvesse a instância de duas telas ao mesmo tempo, ou duas classes por exemplo.

Enviando ao construtor a classe e a Stage que se deseja fazer a plotagem, o sistema se encarrega do resto, apenas sendo necessário ao usuário que chame a função “*apresentarTela”*, onde será carregada toda a tela e as entradas de dados. Pode-se também utilizar da função “*setSalvarAoFechar*” para definir se deverá surgir uma tela de escolha de local para salvar ao fechar o sistema, por padrão vem como *true.* É possível também, esconder ou mostrar a tela, uma vez que o contrário é real, ou seja, se está escondida é possível mostrá-la.

Função de salvar está disponível nesta classe também, porém a função “*salvaJson*” não apresenta tela de escolha.

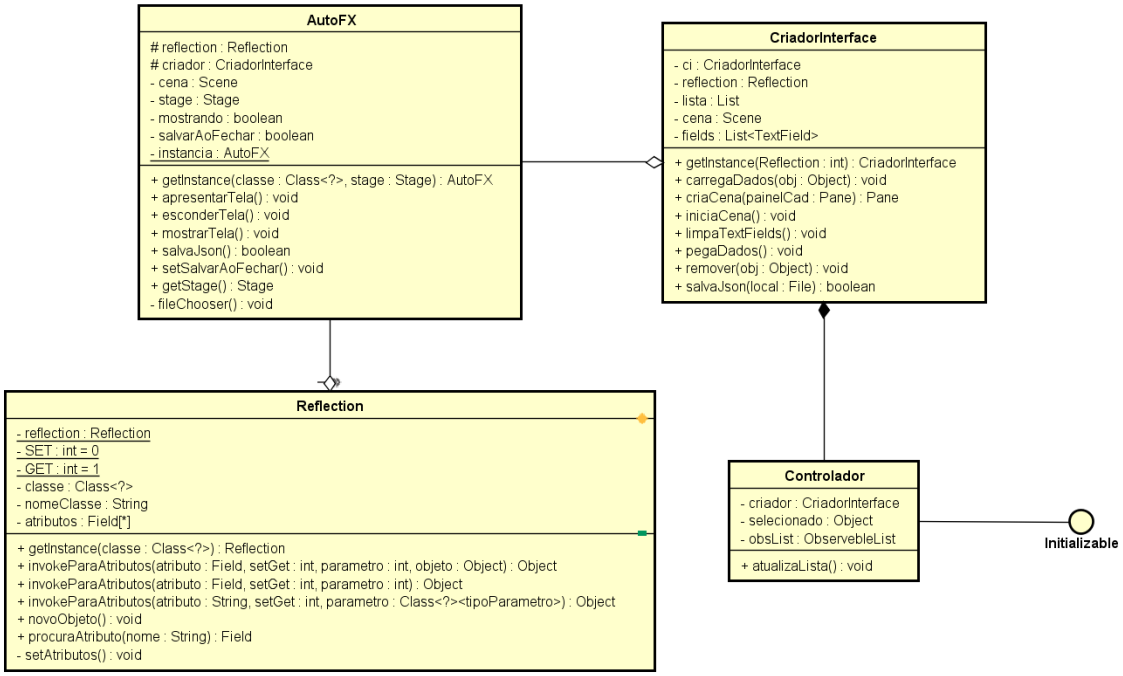

Figura 1: Diagrama de classes UML

Referências

BASTOS, Henrique. **Diferenças entre linguagem compilada e linguagem interpretada.** 2010. Disponível em: <https://www.oficinadanet.com.br/artigo/programacao/diferencas_entre_linguagem_compilada_e_linguagem_interpretada>. Acesso em: 04 jul. 2019.

Guia Completo de Java. Reunião de guias. Disponível em: <https://www.devmedia.com.br/guia/linguagem-java/38169>. Acesso em: 04 jul. 2019.

LANHELLAS, Ronaldo. **Conhecendo Java Reflection.** 2013. Disponível em: <https://www.devmedia.com.br/conhecendo-java-reflection/29148>. Acesso em: 04 jul. 2019.

MENEZES, Daniel Assunção Faria de. **JavaFX 8: Uma introdução à arquitetura e às novidades da API.** 2015. Disponível em: <https://www.devmedia.com.br/javafx-8-uma-introducao-a-arquitetura-e-as-novidades-da-api/33702>. Acesso em: 04 jul. 2019.

NORONHA, Caio. **Programação Orientada a Objetos(POO).** 2018. Disponível em: <https://medium.com/@caio.cnoronha/programa%C3%A7%C3%A3o-orienta%C3%A7%C3%A3o-a-objetos-poo-759d96dda910>. Acesso em: 04 jul. 2019.

Welson. **Vantagens e Desvantagens da POO.** 2015. Disponível em: <https://www.devmedia.com.br/vantagens-e-desvantagens-da-poo/32655>. Acesso em: 04 jul. 2019.