PROJETO JAVA

Form

Rafael Rio, Ricardo Fortuna

Nº22100521, Nº22107919

ÍNDICE

INTRODUÇÃO

Este trabalho tem como objetivo criar um programa que incida sobre os conceitos de herança, polimorfismo, genéricos e coleções, estudados no âmbito do paradigma de programação orientada a objetos.

O programa vai simular um formulário de utilizador aplicado a Web, tendo em conta que existem várias generalizações ao longo do projeto uma delas o próprio Form que deve ser geral para uma possível aplicação noutros tipos de formulários.

São criadas 11 classes, das quais, 2 são dadas pelo enunciado (Cliente, UsernameForm) e uma é de origem da programação Java (HashMap)

.

DESCRIÇÃO

FUNCIONALIDADES DO PROGRAMA

Em termos de funcionalidades o programa:

- Estrutura de um formulário;
- Restrições de entrada nos Fields;

ESTRUTURA DO PROGRAMA

O programa não contém uma interface gráfica, porém tem como base estrutural os seguintes módulos:

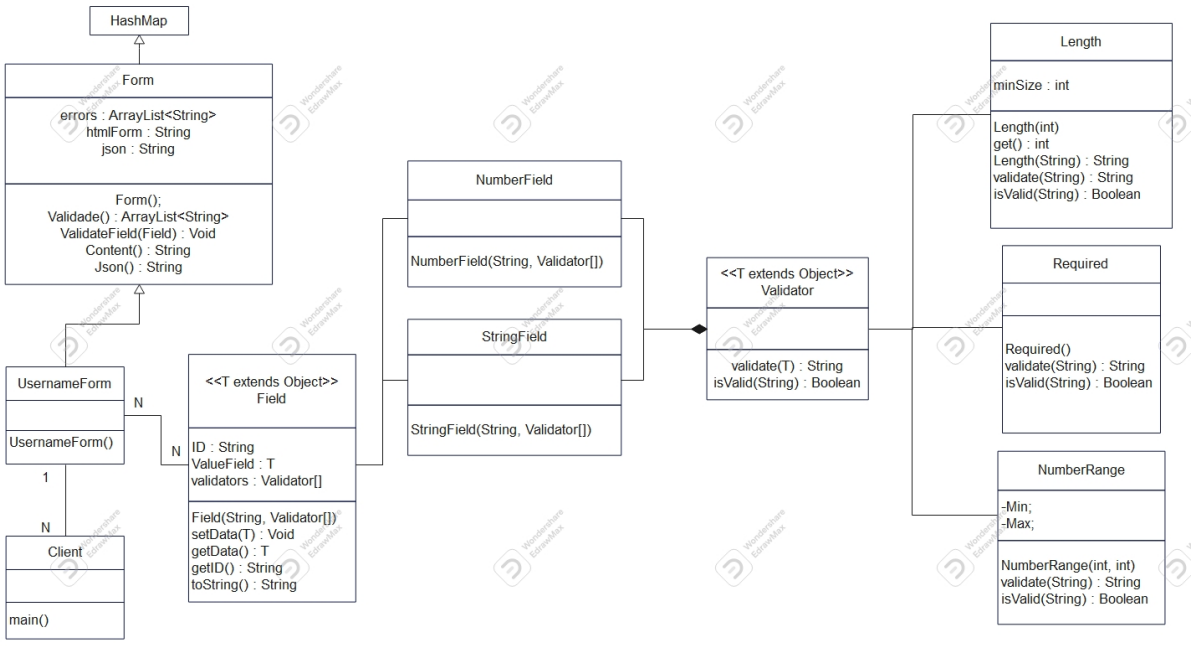
- Processamento dos dados – Define como os dados serão tratados de acordo com os objetivos do programa.
- Escrita dos dados processados – Possibilita ao utilizador a leitura dos dados processados na consola.

IMPLEMENTAÇÃO DO PROGRAMA

Iremos descrever a implementação do programa. Usando alguns trechos do código, assim como, também o diagrama de classes (UML).

DIAGRAMA DE CLASSES (UML)

Para entendermos melhor o problema, usamos o seguinte diagrama de classes:



No diagrama conseguimos observar as suas heranças, as relações e as agregações .

CLASSES

Este projecto foi desenvolvido em duas pastas, a pasta do código fonte e a dos testes unitários, para separar as funcionalidades de cada uma das pastas.

Têm ambas o mesmo package para as classes de teste conseguirem comunicar com as classes do código principal.

CLASSE CLIENT1

```
1  package Main;
2  public class Client1 {
3
4      public static void main(String[] args) {
5
6          UsernameForm form = new UsernameForm();
7
8          form.get("username").setData("tia");
9          form.get("email").setData("tia@gmail.com");
.0         form.get("age").setData(16);
.1         form.validate();
.2
.3
.4         for (String err : form.errors)
.5             System.out.println(err);
.6
.7         System.out.println(form.content());
.8         System.out.println(form.json());
.9     }
:0
:1 }
:2
```

Esta classe foi dada no enunciado do trabalho e contem um atributo do tipo UsernameForm instanciado.

A classe tem como objetivo simular a resposta de uma página Web, onde os dados são inseridos através de getters e setters. Caso algo dê erro, o programa vai acabar por imprimir no ecrã o erro seguido do formulário com as variáveis inseridas.

CLASSE CLIENT2

É igual à classe client1, mas induz erros propositados

```
1  package Main;
2
3  import org.junit.Ignore;
4
5  @Ignore
6  public class Client2 {
7
8      public static void main(String[] args) {
9
10         UsernameForm form = new UsernameForm();
11         form.get("username").setData("ti");
12         form.get("email").setData("");
13         form.get("age").setData(13);
14         form.validate();
15
16         for (String err : form.errors)
17             System.out.println(err);
18
19         System.out.println(form.content());
20         System.out.println(form.json());
21     }
22 }
23
```

CLASSE FORM

```
package Main;
import java.util.ArrayList;..

public class Form extends HashMap<String, Field> {

    ArrayList<String> errors = new ArrayList<String>();
    private String htmlForm;
    private String json;

    Form() {}

    public ArrayList<String> validate() {

        for (String key : keySet()) {
            var fieldToValidate = get(key);
            ValidateField(fieldToValidate);
        }
        return errors;
        }
```

```
    * ValidateField um Metodo privado que testa se deu erro

    private void ValidateField(Field fieldToValidate) {
        Validator[] fieldValidators = fieldToValidate.validators;

        if (fieldValidators != null) {
            for (Validator validator : fieldValidators) {
                String validationResult = validator.validate(fieldToValidate.getData());

                if (!validator.IsValid(validationResult)) {
                    errors.add(validationResult);
                }
            }
        }
    }

    * constroi o outPut como se fosse um htmlForm

    public String content() {// conteudo inserido

        htmlForm = "<form>\n";

        this.forEach((key, value) -> {
            htmlForm = htmlForm + "\t<label for='" + key.toString() + "'>" + key.toString() + ":</label>\n";
            htmlForm = htmlForm + "\t<label name='" + key.toString() + "' type='" + "' value='" + value.toString()
                    + "'/><br>\n";});
        htmlForm = htmlForm + "</form>";
        return htmlForm;
    }
```

```
*output dos parametros que nao foram respeitados ao inserir no UsernameField

public String json() {
    json = "{";
    this.forEach((key, value) -> {
        json = json + "'" + key.toString() + "':'";
        json = json + value.getData() + "',";
    });

    json = json + "}";

    return json;
} // show conteudo {'email':'','age':13,'username':'ti'}
```

Classe form é uma extensão da classe HashMap da linguagem Java.

Tem 3 atributos: um ArrayList, do tipo String, para guardar os erros; htlmForm do tipo String para dar um output do que é inserido nos fields e Json do tipo String para dar output final do processamento dos dados

Chegou-se a conclusão de que as funções desta classe são funções necessárias e genéricas para a criação de novos formulários para além do formulário Web.

CLASSE USERNAMEFORM

```
2  * interface UsernameForm
7
8  package Main;
9  class UsernameForm extends Form{
10
11     private static final long serialVersionUID = 1L;
12
13     UsernameForm(){
14         super();
15         this.put("username", new StringField("Username",new Validator[]{new Length(3)}));
16         this.put("email", new StringField("Email", new Validator[]{new Required()}));
17         this.put("age", new NumberField("Age", new Validator[]{new NumberRange(16,99)}));
18     }
19 }
```

Classe UsernameForm foi dada no enunciado, é uma extensão do Form, pois é aqui que podemos criar funções que achemos necessárias para a implementação mais específica do formulário.

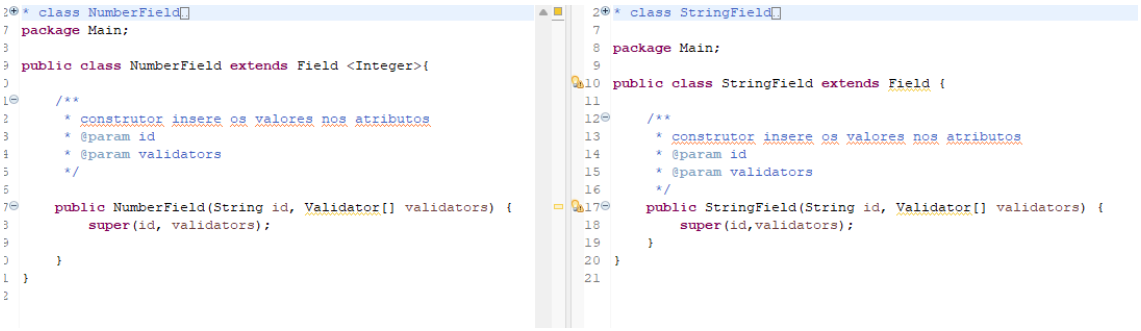
Decidimos deixá-lo simples, pois as funções usadas já satisfaziam o propósito do trabalho.

CLASSE FIELD

```
2⊕ * class Field..
7  package Main;
8
9  public class Field <T extends Object> {
10
11     public String id;                // Cliente Key Do HashMap
12     public T valueField;             // Cliente valor do hashMap
13     public Validator[] validators;
14
15⊖    public Field(String id, Validator[] validators) {
16         this.id = id;
17         this.validators = validators;
18     }
20⊕    * Inserir valor do tipo T (Generico)..
23⊖    public void setData(T x) {
24         this.valueField=x;
25     }
27⊕    * retorna a variável x do tipo T(Generico)..
30⊖    public T getData(){
31           return this.valueField;
32         }
34⊕    * retorna a variável x do tipo T(Generico)..
37⊖    public String getId(){
38           return this.id;
39         }
41⊕    * transforma um object em texto..
44⊖     public String toString(){
45           return String.valueOf(this.getData());
46        }
47 }
```

Decidimos construir a classe Field genérica, pois queríamos fields que pudessem receber inteiros ou strings(texto). Esta ferramenta implica que declaremos uma variável do tipo T que estende o tipo Objeto.

CLASSE NUMBERFIELD E STRINGFIELD



```
* class NumberField
package Main;

public class NumberField extends Field <Integer>{

    /**
     * construtor insere os valores nos atributos
     * @param id
     * @param validators
     */

    public NumberField(String id, Validator[] validators) {
        super(id, validators);

    }
}
```

```
* class StringField

package Main;

public class StringField extends Field {

    /**
     * construtor insere os valores nos atributos
     * @param id
     * @param validators
     */
    public StringField(String id, Validator[] validators) {
        super(id,validators);
    }
}
```

NumberField e StringField estendem da classe Field, é nestas duas classes que surge a especificação do tipo de Objeto da Variável T.

Para a classe NumberField vai assumir o tipo de Integer.

Para a classe StringField vai assumir o tipo de String.

INTERFACE VALIDATOR

```
* interface Validator

package Main;

interface Validator <T extends Object> {

    public String validate(T x);

    public Boolean IsValid(String validationResult);
}
```

Decidimos que esta classe seria uma interface para implementar nas classes legnth, NumberRange, Required.

Tem o método para validar se os requerimentos de field são respeitados o que devolve uma string de erro ou true.

O método isValid simplesmente converte do tipo string em boolean para facilitar o funcionamento.

CLASSE LENGTH

```
/* class Length..

package Main;

public class Length implements Validator <String> {

    private final int minSize;

    public Length(int x) {
        this.minSize = x;
    }

     * retorna o valor do minSize..

    public int get() {
        return this.minSize;
    }

     * metodo de validaçao retorna o Erro em string ou True em string..

    public String validate(String str) {

        if(str.length() < this.minSize) return "Error: less than min";
        else return "True";

    }
     * Converte o metodo anterior em string para Booleano..
    public Boolean IsValid(String validationResult) {
        return validationResult == "True";
    }

}
```

Classe Length é responsável pela validação de tamanho mínimo do Field, sendo assim, precisamos de um atributo do minSize do tipo inteiro para conseguirmos criar a condição.

CLASSE NUMBERRANGE

```
/* class NumberRange..

package Main;

public class NumberRange implements Validator <Integer> {

    final int min;
    final int max;

     * construtor insere os valores nos atributos..
    public NumberRange(int x, int y) {
        this.min = x;
        this.max = y;
    }

     * metodo de validaçao retorna o Erro em string ou True em string..

    public String validate(Integer x) {
        if(x < this.min || x > this.max) return "Error: value not in range";
        else return "True";
    }

     * Converte o metodo anterior em string para Booleano..
    public Boolean IsValid(String validationResult) {
        return validationResult == "True";
    }
}
```

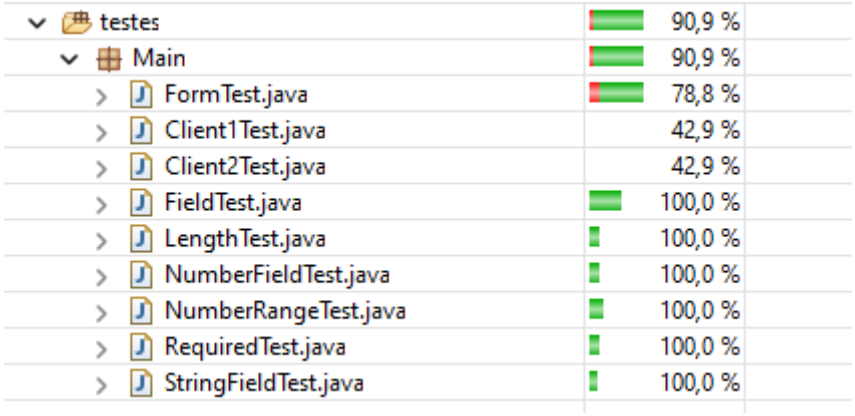
Classe NumberRange é responsável pela validação de intervalo mínimo e máximo do Field, sendo assim, precisamos de dois atributos, o min e o max, sendo eles do tipo inteiro, para conseguirmos criar a condição.

CLASSE REQUIRED

```
package Main;
public class Required implements Validator <String> {

    public Required() {

    }

     * metodo de validacao retorna o Erro em string ou True em string
    public String validate(String str) {

        if(str == null) return "Error: Value empty";
        else if(str.length() <= 0) return "Error: Value empty";
        else return "True";
    }


     * Converte o metodo anterior em string para Booleano
    public Boolean IsValid(String validationResult) {
        return validationResult == "True";
    }
}
```

Classe Required é responsável pela validação que obriga o Field a não estar vazio, o que faz com que tenhamos uma constante de “0”, sendo que não precisamos de criar uma variável para a controlar.

TESTES UNITÁRIOS



CLASSFIELDTEST

```
package Main;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class FieldTest  <T extends Object>{

    Field validar1 = new StringField("Username", new Validator[] { new Length(3) });
    Field validar2 = new StringField("Email", new Validator[] { new Required() });
    Field validar3 = new NumberField("Age", new Validator[] { new NumberRange(16, 99) });

    @Test
    public final void testField() {

        // setData
        validar1.setData("Tia");
        validar2.setData("ismat@gmail.com");
        validar3.setData(16);

        // GetID
        assertEquals(validar1.getId(), "Username");
        assertEquals(validar2.getId(), "Email");
        assertEquals(validar3.getId(), "Age");

        // GetData
        assertEquals(validar1.getData(), "Tia");
        assertEquals(validar2.getData(), "ismat@gmail.com");
        assertEquals(validar3.getData(), 16);

        // Teste de requesitos True
        assertEquals(validar1.validators[0].validate(validar1.getData()), "True");
        assertEquals(validar2.validators[0].validate(validar2.getData()), "True");
        assertEquals(validar3.validators[0].validate(validar3.getData()), "True");

        // Teste de requesitos False
        assertEquals(validar1.validators[0].validate("Ti"), "Error: less than min");
        assertEquals(validar2.validators[0].validate(""), "Error: Value empty");
        assertEquals(validar3.validators[0].validate(4), "Error: value not in range");
    }
```

```
    @Test
    public final void testToString() {
        validar1.setData("Tia");

        assertEquals(validar1.toString(), "Tia");
    }

}
```

CLASS FORMTESTE

```
package Main;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class FormTest extends HashMap<String, Field>{


    ArrayList<String> errors = new ArrayList<String>();
    UsernameForm form = new UsernameForm();

    private String htmlForm;
    private String json;

    @Test
    public final void testForm() {

        form.put("username", new StringField("Username",new Validator[]{new Length(3)}));
        form.put("email", new StringField("Email", new Validator[]{new Required()}));
        form.put("age", new NumberField("Age", new Validator[]{new NumberRange(16,99)}));

        form.get("username").setData("tia");
        form.get("email").setData("tia@gmail.com");
        form.get("age").setData(16);
    }

    @Test
    public final void testValidate() {
        for (String key : keySet()) {
            var fieldToValidate = get(key);
            ValidateField(fieldToValidate);
        }
    }

    @Test
    private void ValidateField(Field fieldToValidate) {
        Validator[] fieldValidators = fieldToValidate.validators;

        if (fieldValidators != null) {
            for (Validator validator : fieldValidators) {
                String validationResult = validator.validate(fieldToValidate.getData());

                if (!validator.IsValid(validationResult)) {
                    errors.add(validationResult);
                }
            }
        }
    }


    @Test
    public final void testContent() {


        htmlForm = "<form>\n";

        form.forEach((key, value) -> {
            htmlForm = htmlForm + "\t<label for='" + key.toString() + "'>" + key.toString() + ":</label>\n";
            htmlForm = htmlForm + "\t<label name='" + key.toString() + "' type='" + "' value='" + value.toString()
                    + "'/><br>\n";
        });
        htmlForm = htmlForm + "</form>";

        String contentIgual="<form>\n"
            +"\t<label for='email'>email:</label> \n"
            + "\t<label name='email' type='' value='tia@gmail.com'/><br>\n"
            + "\t<label for='age'>age:</label>\n"
            +"\t<label name='age' type='' value='16'/><br>\n"
            +"\t<label for='username'>username:</label>\n"
            +"\t<label name='username' type='' value='tia'/><br>\n"
        +"</form>";

        assertEquals(contentIgual, htmlForm);
    }
```

```
    @Test
    public final void testJson() {

        json = "{";
        form.forEach((key, value) -> {
            json = json + "'" + key.toString() + "':'";
            json = json + value.getData() + "',";
        });

        json = json + "}";

        String jsonIgual="{'email':'tia@gmail.com','age':'16','username':'tia',}";


        assertEquals(jsonIgual, json);

    }

}
```

CLASS STRINGFIELD

```
1  package Main;
2  import static org.junit.jupiter.api.Assertions.assertEquals;
10
11 public class StringFieldTest {
12
13     Field validar2 = new StringField("Email", new Validator[] { new Required() });
14
15     @Test
16     final void testStringField() {
17
18         validar2.setData("ismat@gmail.com");
19
20         assertEquals(validar2.getData(), "ismat@gmail.com");
21         assertEquals(validar2.validators[0].validate(validar2.getData()), "True");
22         assertEquals(validar2.validators[0].validate(""), "Error: Value empty");
23     }
24 }
```

CLASS NUMBERFIELD

```
package Main;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class NumberFieldTest {

    Field validar3 = new NumberField("Age", new Validator[] { new NumberRange(16, 99) });

    @Test
    public final void testNumberField() {

        validar3.setData(16);
        assertEquals(validar3.getData(), 16);
        assertEquals(validar3.validators[0].validate(validar3.getData()), "True");
        assertEquals(validar3.validators[0].validate(4), "Error: value not in range");
    }

}
```

CLASS LEGNTH

```
package Main;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class LengthTest {

    Length validar = new Length(3);

    @Test
    public void testget() {
        assertEquals(validar.get(),3);
    }

    @Test
    public void testValidateTrue() {
        String str = "Tia";
        assertEquals(validar.validate(str),"True");
    }

    @Test
    public void testValidateFalse() {
        String str1 = "Ti";
        assertEquals(validar.validate(str1),"Error: less than min");
    }

    @Test
    public void testIsValidTrue() {
        String str = "True";
        assertEquals(validar.IsValid(str),true);
    }

    @Test
    public void testIsValidFalse() {
        String str = "Error";
        assertEquals(validar.IsValid(str),false);
    }
```

CLASS NUMBERRANGE

```
package Main;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class NumberRangeTest {
    NumberRange validar = new NumberRange(2, 4);
    @Test
    final void testNumberRange() {
        assertEquals(validar.min, 2);
        assertEquals(validar.max, 4);
    }
    @Test
    final void testValidateTrue() {
        int idade=3;
        assertEquals(validar.validate(idade), "True");
    }

    @Test
    final void testValidateMenor() {
        int idade=0;
        assertEquals(validar.validate(idade), "Error: value not in range");
    }
    @Test
    final void testValidateMaior() {
        int idade=9;
        assertEquals(validar.validate(idade), "Error: value not in range");
    }
    @Test
    void testIsValidTrue() {
        String str = "True";
        assertEquals(validar.IsValid(str),true);
    }
    @Test
    void testIsValidFalse() {
        String str = "Error";
        assertEquals(validar.IsValid(str),false);
    }
}
```

CLASS REQUIRED

```
package Main;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class RequiredTest {

    Required teste = new Required();

    @Test
    final void testValidateTrue() {
        String validar = "True";
        assertEquals(teste.validate(validar),"True");
    }

    @Test
    final void testValidateFalse() {
        String validar ="";
        assertEquals(teste.validate(validar),"Error: Value empty");
    }

    @Test
    final void testValidateNull() {
        String validar = null;
        assertEquals(teste.validate(validar),"Error: Value empty");
    }

    @Test
    void testIsValidTrue() {
        String str = "True";
        assertEquals(teste.IsValid(str),true);
    }

    @Test
    void testIsValidFalse() {
        String str = "Error";
        assertEquals(teste.IsValid(str),false);
    }

}
```

CONCLUSÃO

O Projecto do trabalho prático 2, de formulário Web, foi bastante interessante pois tivemos de aplicar e intensificar o estudo, melhorando os conceitos como a herança, o polimorfismo, os genéricos e as coleções, estudados no âmbito do paradigma de programação orientada a objetos.

Concluímos assim que os testes unitários são de facto uma ferramenta bem forte no desenvolvimento do código e na prevenção de erros, o que é efetivamente útil para nos preparar melhor para o mercado de trabalho.