TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN
VALIDATION FRAMEWORK

Nhóm:	:	10
Lớp:	:	20_3
Năm:	:	2023 - 2024
Giảng viên hướng dẫn	:	Ths. Nguyễn Minh Huy Ths. Trần Duy Quang

Hồ Chí Minh, tháng 01 năm 2024

MỤC LỤC

Danh mục hình ảnh

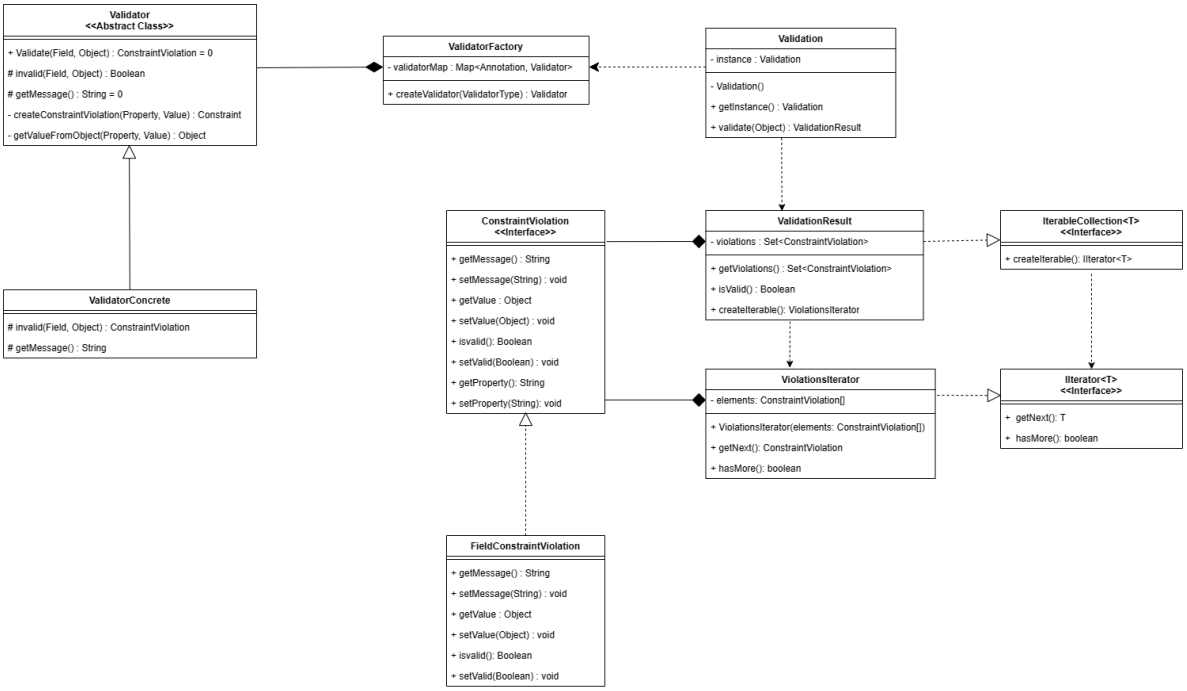
Thông tin nhóm

Mã số sinh viên	Họ tên
1712577	NGUYỄN PHI LONG
20120173	TRẦN NGUYỄN QUY
20120574	TRẦN DUY TÂN
20120580	DƯƠNG TẤN THÀNH

Công việc thực hiện

Sinh viên	Công việc
NGUYỄN PHI LONG	• Thiết kế lớp framework • Tổng hợp báo cáo
TRẦN NGUYỄN QUY	• Thiết kế lớp framework • Code giao diện • Code Iterator
TRẦN DUY TÂN	• Thiết kế lớp framework • Code giao diện • Code Iterator
DƯƠNG TẤN THÀNH	• Thiết kế lớp framework • Code Singleton, Factory method, Strategy

1. Sơ đồ lớp



Hình ảnh 1: Sơ đồ lớp

1.1. Ý nghĩa các lớp

1.1.1. Validator <<Abstract Class>>

- validate(Field, Object) : ConstraintViolation

-> Hàm này kiểm tra Field của Object và trả về đối tượng ConstraintViolation với các thông tin có hợp lệ hay không, và message

- getValueFromObject(Field field, Object object)

-> hàm này lấy giá trị từ 1 Field của 1 Object mục đích để kiểm tra ở hàm Invalid

- createConstraintViolation(String property, Object value) : ConstraintViolation

-> Hàm này tạo 1 đối tượng ConstraintViolation với property là tên field, value là giá trị của field.

- 2 Hàm ảo invalid(Field, Object) và getMessage(Field) là để kiểm tra hợp lệ và lấy message từ annotation

1.1.2. ValidatorConcrete

Là lớp triển khai cụ thể của Validator cho từng ràng buộc mong muốn ví dụ như NotNullValidator,...

Lớp này sẽ triển khai cụ thể cho 2 phương thức invalid(Field, Object) và getMessage(Field) với từng validator cụ thể

1.1.3. ValidatorFactory

Lớp này có nhiệm vụ tạo ra các validator cụ thể.

- Lớp dùng 1 Map<Annotation, Validator> để lưu các validator
- Phương thức createValidator(ValidatorType) sẽ nhận vào 1 Enum là ValidatorType và nếu trong map có validator tương ứng thì sẽ trả về nếu không sẽ tạo mới lưu vào Map và trả về.

1.1.4. Validation

Lớp này là 1 Singleton với nhiệm vụ sẽ dùng các Validator kiểm tra các Object và trả về kết quả

- Phương thức validate(Object) sẽ kiểm tra các ràng buộc của Object và trả về ValidationResult (là Set các ConstraintViolation)
- isValid(): kiểm tra xem kết quả có vi phạm ràng buộc nào không dựa vào Set ConstraintViolation có trống hay không

1.1.5. ConstraintViolation <<Interface>>

Interface này đại diện cho vi phạm ràng buộc.

1.1.6. FieldConstraintViolation

Lớp triển khai cụ thể cho interface ConstraintViolation. khi triển khai dựa trên interface sau này ta có thể dễ dàng thêm mới như là thêm ParamConstraintViolation (Vi phạm khi truyền parameter),...

1.1.7. ValidationResult

Lớp này đại diện cho kết quả kiểm tra gồm có thuộc tính violations là set các ràng buộc vi phạm.

1.1.8. Iterator <<Interface>>

Một interface tự cài đặt thay cho Iterator mặc định Java, khai báo các phương thức cần thiết để duyệt qua tất cả phần tử như getNext(), hasMore()...

1.1.9. IterableCollection <<Interface>>

Một interface tự cài đặt thay cho Iterable mặc định Java, khai báo phương thức createIterable() để những lớp muốn Iterable cần implement lại phương thức để tạo ra một IIterator ....

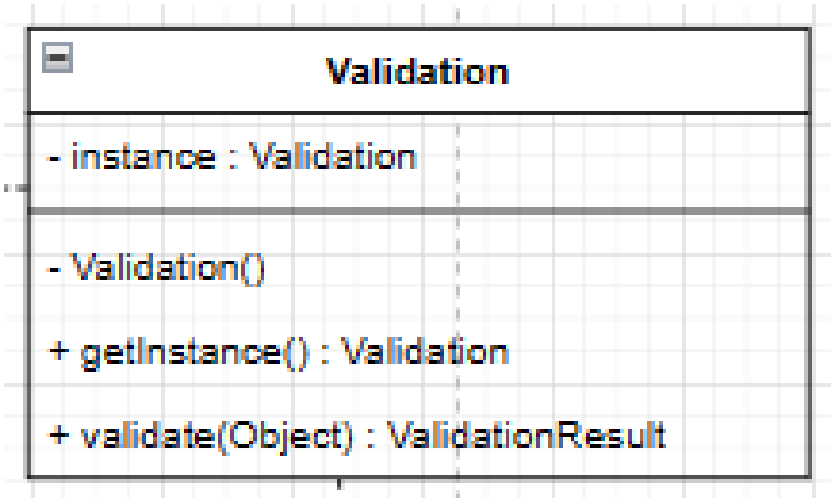
1.1.10. ViolationsIterator

Đây là lớp triển khai của Iterator, được tạo ra từ lớp Violation Result bằng phương thức createIterable(), sẽ có một danh sách các kết quả Violations Constraint được tạo ra và có có thể duyệt qua để xử lý logic hiển thị lỗi.

1.2. Các mẫu thiết kế hướng đối tượng sử dụng

1.2.1. Singleton

Sơ đồ lớp



Hình ảnh 2: Sơ đồ lớp Singleton

Đoạn code

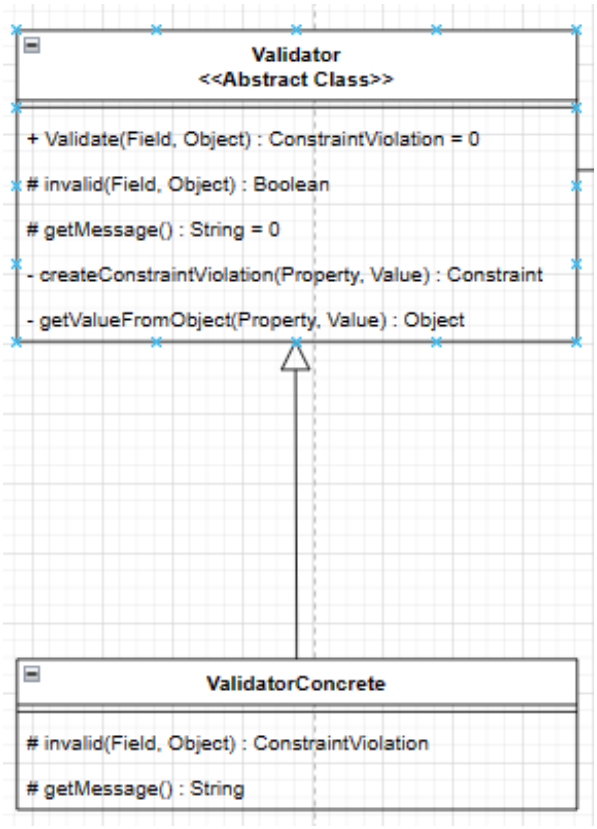
```
public class Validation {
    private static Validation instance = new Validation();
    private Validation() {

    }
    public static Validation getInstance() {
        return instance;
    }

    public ValidationResult validate(Object object) {
        // Triển khai hàm
        return validationResult;
    }
}
```

Ý nghĩa

- Nhiệm vụ validate chỉ cần 1 Instance là có thể đảm nhiệm nên Singleton giúp ta kiểm soát tài nguyên
- Đảm bảo lớp Validation chỉ có 1 Instance
- Cung cấp 1 điểm truy cập toàn cục tới Instance
- Vì lớp Validation tương đối đơn giản không yêu cầu nhiều tài nguyên nên sử dụng hướng tiếp cận Eager initialization giúp tránh các vấn đề về xử lý đồng thời.

1.2.2. Strategy

Sơ đồ lớp



Hình ảnh 3: Sơ đồ lớp Strategy

Đoạn code

```
public abstract class Validator {
    protected abstract boolean invalid(Field field, Object value);
    protected abstract String getMessage(Field field);
    public final ConstraintViolation validate(Field field, Object object) {

        Object value = getValueFromObject(field, object);

        ConstraintViolation constraint = createConstraintViolation(field.getName(), value);

            if (this.invalid(field, value)) {
                constraint.setMessage(this.getMessage(field));
                constraint.setValid(false);
            }

            return constraint;
    }

    private ConstraintViolation createConstraintViolation(String property, Object value){
        ConstraintViolation constraint = new FieldConstraintViolation();

        constraint.setProperty(property);
```

```
        constraint.setValue(value);

        return constraint;
    }

    private Object getValueFromObject(Field field, Object object) {
        try {
            Object value = field.get(object);
            return value;
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }

        return null;
    }


}
```

```
public class BlankValidator extends Validator {
    @Override
    protected boolean invalid(Field field, Object value) {
        if(value instanceof String) {
            return ((String) value).isEmpty();
        }
        return false;
    }

    @Override
    protected String getMessage(Field field) {
        Blank annotation = field.getDeclaredAnnotation(Blank.class);
        return annotation.message();
    }
}
```

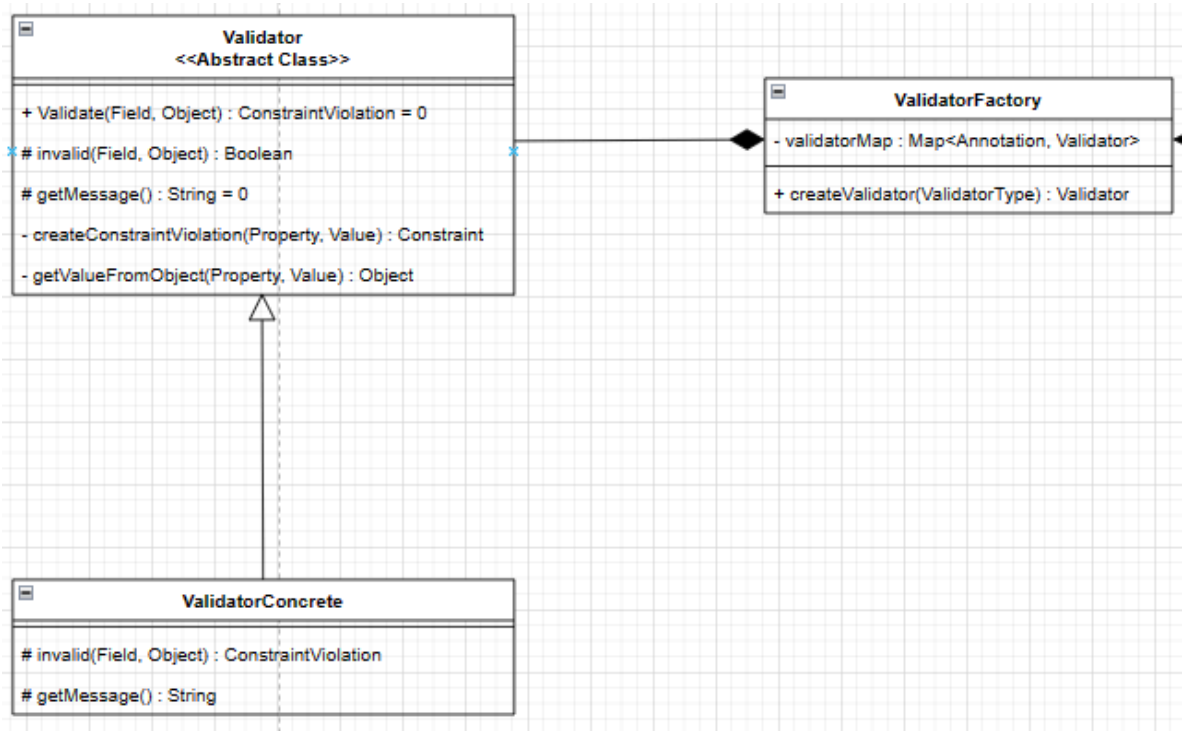
Ý nghĩa

- Dùng strategy ta có thể thay đổi linh hoạt các strategy (dựa vào tính năng liên kết động sẽ xác định cụ thể hàm nào sẽ được thực hiện dựa vào loại validator lúc runtime). Cụ thể là tùy vào loại validator thì hàm invalid và getMessage có hành vi khác nhau.
- Dễ Dàng Mở Rộng và Bảo Trì: ta có thể dễ dàng thêm mới các ConcreteValidator mà không gây ảnh hưởng tới hệ thống đang có.
- Tuân thủ nguyên tắc Open/Closed
- Các validator được tách biệt khỏi hệ thống giúp giảm sự phụ thuộc.

- Trong trường hợp này thì Validator có nhiều chiến lược khác nhau nên dùng mẫu strategy là hợp lý và hiệu quả

1.2.3. Factory method

Sơ đồ lớp



Hình ảnh 4: Sơ đồ lớp Factory method

Đoạn code

```
public enum ValidatorType {
    PHONE_NUMBER(PhoneNumber.class),
    REGEX(Regex.class),
    NOT_NULL(NotNull.class),
    MAX(Max.class),
    MIN(Min.class),
    NOT_BLANK(NotBlank.class),
    BLANK(Blank.class);
    private Class<? extends Annotation> value;
    private ValidatorType(Class<? extends Annotation> value) {
        this.value = value;
    }

    public Class<? extends Annotation> getValue() {
        return this.value;
    }
```

```
    public static ValidatorType getType(Class<? extends Annotation> annotationType) {
        for (ValidatorType validatorType : ValidatorType.values()) {
            if (annotationType == validatorType.getValue()) {
                return validatorType;
            }
        }
        return null;
    }
}
```

```
public class ValidatorFactory {
    private static Map<Class<? extends Annotation>, Validator> validatorMap = new HashMap<>();
    public static Validator createValidator(ValidatorType validatorType) {
        Validator validator = validatorMap.get(validatorType.getValue());

        if (validator == null){
            switch (validatorType){
                case NOT_NULL:
                    validator = new NotNullValidator();
                    break;

                case REGEX:
                    validator = new RegexValidator();
                    break;

                case PHONE_NUMBER:
                    validator = new PhoneNumberValidator();
                    break;

                case MAX:
                    validator = new MaxValidator();
                    break;

                case MIN:
                    validator = new MinValidator();
                    break;

                case NOT_BLANK:
                    validator = new NotBlankValidator();
                    break;

                case BLANK:
                    validator = new BlankValidator();
                    break;

                default:
                    return validator;
            }
            validatorMap.put(validatorType.getValue(), validator);
```

```
        }

        return validator;
    }
}
```

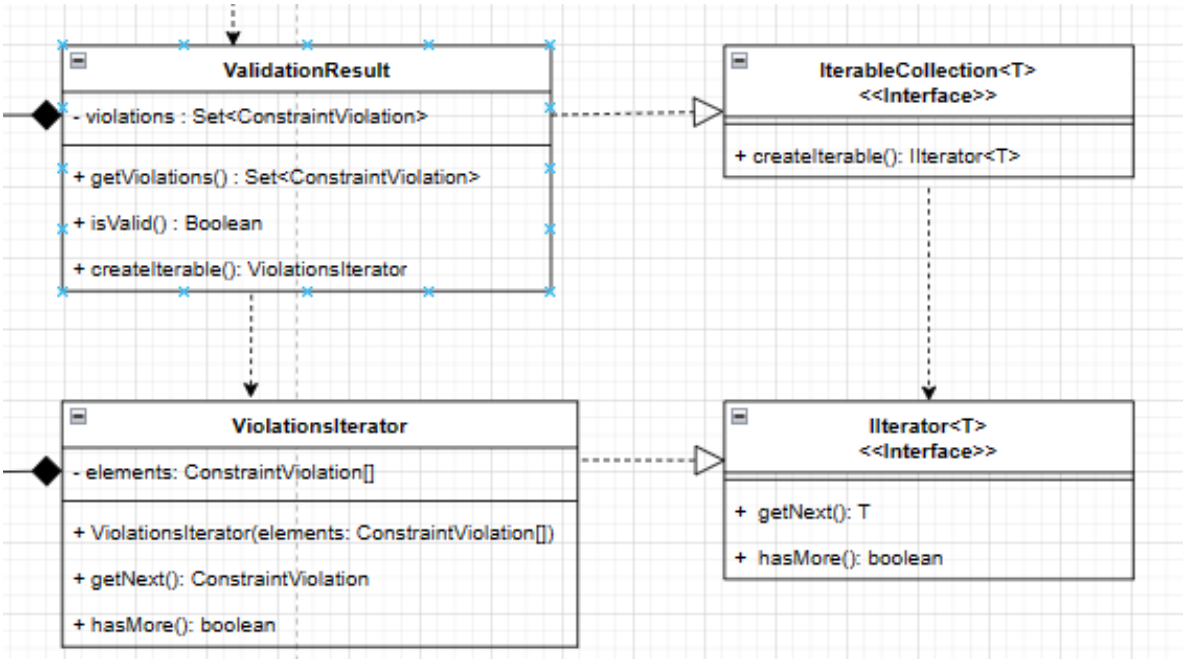
Ý nghĩa

- Tách biệt việc tạo đối tượng ra khỏi lớp và chỉ giao cho lớp ValidatorFactory. Tuân thủ tính Single Responsibility hướng đối tượng.
- Tuân thủ Open/Closed vì ta có thể thêm việc tạo ra validator mới chỉ cần sửa trong ValidatorFactory chứ không phá vỡ cấu trúc của phần code chính hiện tại.

Ngược lại thì áp dụng FactoryMethod sẽ làm code chúng ta phức tạp hơn vì phải tạo ra nhiều SubClass để triển khai.

1.2.4. Iterator

Sơ đồ lớp



Hình ảnh 5: Sơ đồ lớp Iterator

Đoạn code

```
public interface IIterator<T> {
    public T getNext();
    public boolean hasMore();
}
```

```
public interface IterableCollection<T> {
    public IIterator<T> createIterable();
}
```

```
public class ViolationsIterator implements IIterator<ConstraintViolation> {
    private ConstraintViolation[] elements;
    private int index;

    public ViolationsIterator(ConstraintViolation[] elements) {
        this.elements = elements;
        this.index = 0;
    }

    @Override
    public ConstraintViolation getNext() {
        if (hasMore()) {
            ConstraintViolation element = elements[index];
            index++;
            return element;
        } else {
            return null;
        }
    }

    @Override
    public boolean hasMore() {
        return index < elements.length;
    }
}
```

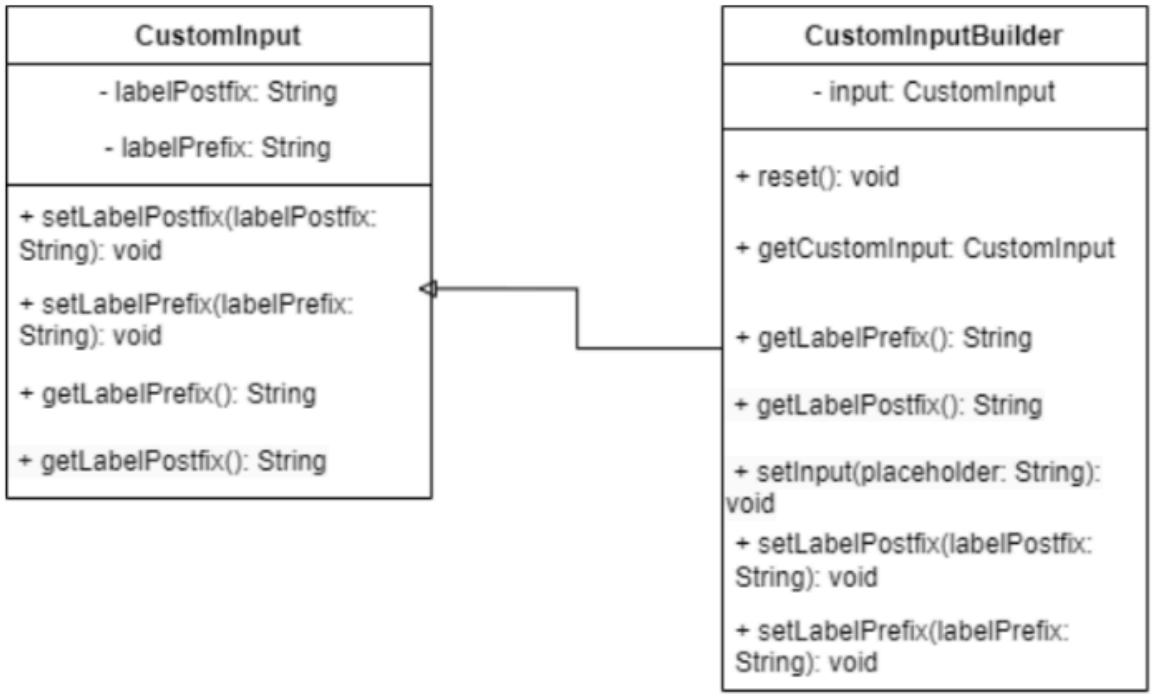
```
public class ValidationResult implements IterableCollection<ConstraintViolation> {
    private Set<ConstraintViolation> violations;
    public ValidationResult() {
        violations = new HashSet<ConstraintViolation>();
    }
    public ValidationResult(Set<ConstraintViolation> violations) {
        this.violations = violations;
    }

    public Set<ConstraintViolation> getViolations() {
        return violations;
    }
```

```
    public boolean isValid() {
        return violations.isEmpty();
    }

    public void addConstraintViolation(ConstraintViolation violation) {
        violations.add(violation);
    }

    @Override
    public ViolationsIterator createIterable() {
        return new ViolationsIterator(violations.toArray(new ConstraintViolation[0]));
    }
}
```

Ý nghĩa: Thay thế lặp mặc định của class Set, cung cấp khả năng tùy chỉnh cao hơn trong khi lặp, dễ dàng mở rộng logic sau này.

1.2.5. Builder

Sơ đồ lớp



Hình ảnh 6: Sơ đồ lớp Builder

Đoạn code

```
public class CustomInput extends TextField {
    private Label labelPrefix;
    private Label labelPostfix;

    public Label getLabelPostfix() {
        return labelPostfix;
    }

    public void setLabelPostfix(Label labelPostfix) {
        this.labelPostfix = labelPostfix;
    }

    public Label getLabelPrefix() {
        return labelPrefix;
    }

    public void setLabelPrefix(Label labelPrefix) {
        this.labelPrefix = labelPrefix;
    }
}
```

```
public class CustomInputBuilder {
    private CustomInput input;

    public CustomInputBuilder() {
        input = new CustomInput();
    }
    public void reset() {
        input = new CustomInput();

    }

    public CustomInput getCustomInput() {
        CustomInput returnedInput = input;
        this.reset();
        return returnedInput;
    }

    public void setPrefixLabel(String label) {
        Label constructedlabel = new Label();
        constructedlabel.setFont(Font.font("Verdana", FontWeight.NORMAL, 12));
        constructedlabel.setTextFill(Color.RED);
        constructedlabel.setText(label);

        this.input.setLabelPrefix(constructedlabel);
    }

    public void setPostfixLabel(String label) {
        Label constructedlabel = new Label();
        constructedlabel.setFont(Font.font("Verdana", FontWeight.NORMAL, 12));
```

```
        constructedlabel.setTextFill(Color.RED);
        constructedlabel.setText(label);

        this.input.setLabelPostfix(constructedlabel);
    }

    public void setInput(String placeholder) {
        this.input.setPromptText(placeholder);
        this.input.setPrefSize(400,30);
        this.input.setFont(Font.font("Verdana", FontWeight.NORMAL, 20));
    }
}
```

Ý nghĩa

- Đoạn code khởi tạo textfield, khởi tạo label và gán label vào textfield lặp đi lặp lại nên ta có thể sử dụng Builder để tối ưu
- Có thể thêm mới dễ dàng cho các loại Input khác (Date Input, Phone number Input)

Tài liệu tham khảo

[1] “Builder Design Pattern”. [Online]. Available: https://refactoring.guru/design-patterns/builder

[2] “Factory Method Design Pattern”. [Online]. Available: https://refactoring.guru/design-patterns/factory-method

[3] “Iterator Design Pattern”. [Online]. Available: https://refactoring.guru/design-patterns/iterator

[4] “Singleton Design Pattern”. [Online]. Available: https://refactoring.guru/design-patterns/singleton

[5] “Strategy Design Pattern”. [Online]. Available: https://refactoring.guru/design-patterns/strategy

[6] Baeldung, “Java Validation Tutorial”. [Online]. Available: https://www.baeldung.com/java-validation