Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №5

Теорія Розробки Програмного Забезпечення

Шаблони «Adapter», «Builder», «Command», «Chain Of Responsibility», «Prototype»
Варіант 23

Виконав
студент групи ІА-14:
Фіалківський І.О.

Перевірив:
Мягкий М.Ю.

Київ 2023

Тема: Шаблони «Adapter», «Builder», «Command», «Chain Of Responsibility», «Prototype»

Завдання:

- Ознайомитися з короткими теоретичними відомостями.
- Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
- Застосування одного з розглянутих шаблонів при реалізації програми.

Хід роботи:

Project Management software (proxy, chain of responsibility, abstract factory, bridge, flyweight, client-server)

Основна суть Chain Of Responsibility шаблону: зв'язування об'єктів-одержувачів у ланцюжок та передача запиту по ньому.

Яку проблему вирішує даний шаблон: при звичайному підході відправник запиту буде тісно пов'язаний з його одержувачем і це ускладнить зміну поведінки системи, оскільки зміни в одержувачі, ймовірно, вимагатимуть змін у відправнику.

Шаблон Chain Of Responsibility допомагає уникнути прив'язки відправника запиту до його одержувача, що дозволяє обробити даний запит кількома об'єктами. Це досягається шляхом створення ланцюжка об'єктів, кожен з яких може обробляти запит, або передавати запит ланцюжком до тих пір, поки він не буде оброблений. З цим шаблоном відправнику нічого не потрібно знати про одержувача.

У контексті нашого коду, ланцюг валідації представляє собою послідовність валідаторів, кожен з яких відповідає за виконання певної перевірки. Запит на валідацію об'єкта проходить через цей ланцюг, і кожен валідатор вирішує, чи може він обробити цей запит чи повинен передати його наступному валідатору у ланцюзі.

Такий підхід дозволяє гнучко налаштовувати та розширювати логіку валідації, додаючи чи змінюючи окремі етапи без зміни загальної структури системи. Крім того, він спрощує код відправника, оскільки він не пов'язаний напряму з конкретними валідаторами, і може обробляти об'єкти універсальним чином.

За допомогою цього інтерфейсу створюємо чергу з валідаторів.

```
package com.example.util.validation;

import com.example.entity.Task;
import org.springframework.validation.Errors;

public interface ValidationChain<T> {
    void validate(T toValidate, Errors errors);
}
Цей код визначає інтерфейс ValidationChain<T>, який слугує основою для ланцюга валідації. Ланцюг валідації - це спосіб об'єднати різні правила валідації для об'єкта типу T, які можуть бути застосовані послідовно. В цьому випадку, його використовують для валідації об'єктів типу Task, оскільки T визначено як Task
```

Реалізація нашого інтерфейсу.

```
package com.example.util.validation;

import com.example.entity.Task;
import com.example.util.validation.validator.AbstractValidationStep;
import org.springframework.validation.Errors;

public class TaskValidationChain implements ValidationChain<Task>{
    private final AbstractValidationStep<Task> chainHead;

    public TaskValidationChain(AbstractValidationStep<Task> chainHead) {
        this.chainHead = chainHead;
    }

    @Override
    public void validate(Task toValidate, Errors errors) {
        chainHead.validate(toValidate, errors);
    }
}
```

Цей підхід забезпечує гнучкість та легкість розширення валідації об'єктів типу Task, оскільки ви можете легко додавати нові етапи ланцюга без зміни основного коду.

Був створений загальний інтерфейс для різних chain element.

В ньому реалізується метод `buildChain(List<T> elements)`

```java
package com.example.util.validation.validator;

import java.util.List;

public interface ChainElement<T> {
    void setNext(T step);

    static <T extends ChainElement<T>> T buildChain(List<T> elements) {
        if (elements.isEmpty()) {
            throw new IllegalArgumentException("Chain elements can't be
empty");
        }
        for (int i = 0; i < elements.size(); i++) {
            var current = elements.get(i);
            var next = i < elements.size() - 1 ? elements.get(i + 1) : null;
            current.setNext(next);
        }
        return elements.get(0);
    }
}
```

Цей інтерфейс може слугувати основою для будь-якого елемента ланцюга валідації, незалежно від його конкретної реалізації. Використання статичного методу buildChain забезпечує зручний спосіб конфігурації ланцюга валідації, особливо коли маєте багато елементів, і ви хочете визначити їх порядок та взаємодію один з одним.

Також був створений абстрактний клас `AbstractValidationStep` для реалізації загальної логіки.

```
package com.example.util.validation.validator;

import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

public abstract class AbstractValidationStep<T> implements
ChainElement<AbstractValidationStep<T>>, Validator {

    private AbstractValidationStep<T> next;

    @Override
    public final void setNext(AbstractValidationStep<T> step) {
        this.next = step;
    }

    protected void nextStep(Object target, Errors errors) {
        if(next == null) return;

        next.validate(target, errors);
    }
}
```

Цей клас служить основою для реалізацій конкретних етапів валідації у вашому ланцюгу. Кожен конкретний етап розширює цей клас та реалізує конкретні правила валідації в методі validate. Також цей клас забезпечує механізм для переходу до наступного етапу у ланцюгу валідації.

Приклад конкретного валідатору. Перевіряє на унікальність назви таски певного проджекту. Стандартними методами цього зробити неможливо, тож довелося реалізовувти власний валідатор.

```
package com.example.util.validation.validator;

import com.example.entity.Task;
import com.example.services.impl.TaskService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;

@Component
@Order(0)
public class UniqueTaskValidator extends AbstractValidationStep<Task> {

    private final TaskService taskService;

    @Autowired
    public UniqueTaskValidator(TaskService taskService) {
        this.taskService = taskService;
    }

    @Override
    public boolean supports(Class<?> clazz) {
        return Task.class.equals(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        Task task = (Task) target;

        if(taskService.findByUniquePrams(task).isPresent()) {
            errors.rejectValue("title", "400", "This project already have task
with this name");
        }
        nextStep(target, errors);
    }
}
```

```
Цей клас додає конкретне правило валідації для об'єктів типу Task, перевіряючи унікальність задачі за певними параметрами. Це може бути корисним у випадках, коли потрібно перевірити унікальність об'єкта перед його збереженням чи оновленням у базі даних.
```

Переглянути весь код можна за посиланням:
https://github.com/vergovters/trpz-spring/tree/master

Конкретні частини реалізаці:

https://github.com/vergovters/trpz-spring/blob/master/trpz/demo/src/main/java/com/example/util/validation/ValidationChain.java

https://github.com/vergovters/trpz-spring/blob/master/trpz/demo/src/main/java/com/example/util/validation/TaskValidationChain.java

https://github.com/vergovters/trpz-spring/blob/master/trpz/demo/src/main/java/com/example/util/validation/validator/ChainElement.java

https://github.com/vergovters/trpz-spring/blob/master/trpz/demo/src/main/java/com/example/util/validation/validator/AbstractValidationStep.java

https://github.com/vergovters/trpz-spring/blob/master/trpz/demo/src/main/java/com/example/util/validation/validator/UniqueFileValidator.java

Висновок:

Використання шаблону "ланцюжок відповідальності" дозволяє покращити модульність та розширюваність системи, особливо у випадках валідації об'єктів. Цей шаблон дозволяє створити ланцюжок валідаторів, які можуть обробляти запити послідовно без прив'язки відправника до конкретного одержувача.

Створені інтерфейси та класи, такі як ValidationChain, AbstractValidationStep, та інші, реалізують загальну логіку для побудови та виконання ланцюжка валідаторів. Кожен конкретний валідатор, такий як UniqueTaskValidator чи FutureDeadLineTaskValidation, реалізує свою

конкретну логіку перевірки та може бути доданий до ланцюжка.

Такий підхід дозволяє легко розширювати функціональність системи за допомогою нових валідаторів без необхідності змінювати відправника запиту. Кожен валідатор може обробляти свої власні види перевірок, що робить систему більш гнучкою та підтримує принцип відокремлення відповідальностей.