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

ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»

JSON Tool

Виконав
студент групи ІА–22:
Щаблевський Д. Е.
Перевірив:
Мягкий Михайло Юрійович

Київ 2024

Зміст

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

Мета: ознайомитися з шаблонами проектування «Adapter», «Builder», «Command», «Chain Of Responsibility», «Prototype»., та набути практичних навичок їх застосування. Реалізувати частину функціоналу програми за допомогою одного з розглянутих шаблонів для досягнення конкретних функціональних можливостей та забезпечення ефективної взаємодії між класами.

Хід роботи

..28 JSON Tool (ENG) (strategy, command, observer, template method, flyweight)

Display JSON schema with syntax highlight. Validate JSON schema and display errors. Create user friendly table\list box\other for read and update JSON schema properties metadata (description, example, data type, format, etc.). Auto save\restore when edit, maybe history. Can check JSON value by schema (Put schema and JSON = valid\invalid, display errors). Export schema as markdown table. JSON to "flat" view.

Короткі теоретичні відомості

Шаблони проєктування представляють собою стандартизовані підходи до вирішення типових задач, які часто зустрічаються при створенні інформаційних систем. Вони включають в себе опис проблеми, ефективне рішення та рекомендації щодо застосування в різних контекстах. Завдяки загальновизнаним назвам, шаблони легко розпізнаються та багаторазово використовуються.

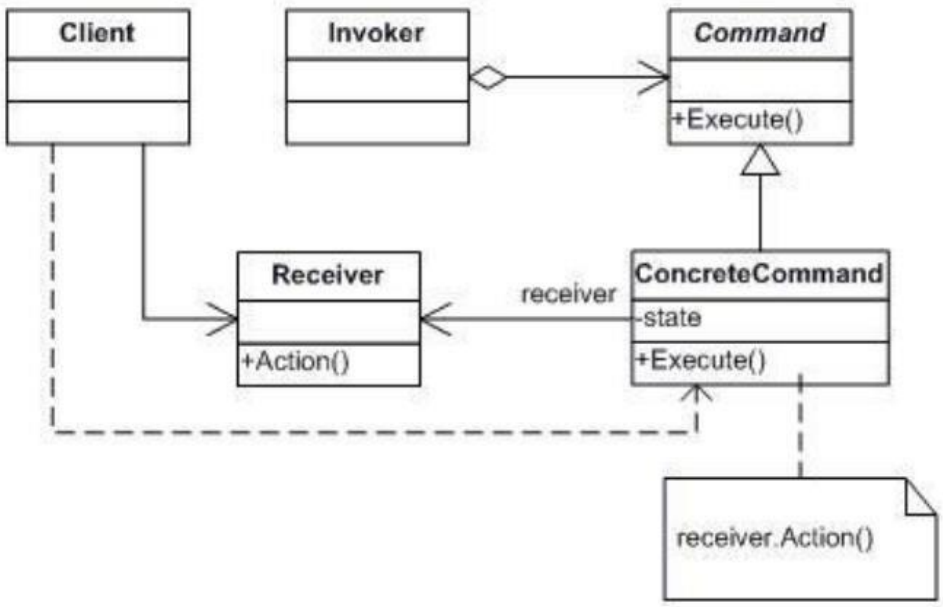
Ключовою особливістю роботи з шаблонами є ретельне моделювання предметної області, що дозволяє точно сформулювати проблему і підібрати найкращий шаблон. Використання таких рішень дає розробнику ряд переваг: вони допомагають зробити систему більш організованою, зрозумілою для вивчення та легкою у розширенні. Крім того, це підвищує гнучкість системи до змін і сприяє її простішій інтеграції з іншими рішеннями.

Таким чином, шаблони проєктування є перевіреними часом інструментами, які широко застосовуються в різних організаціях і проєктах для побудови ефективної архітектури в заданих умовах.

Навіщо використовувати шаблони ?

Шаблони проєктування сприяють зменшенню часу і зусиль на створення архітектури, надаючи системі важливі властивості, як-от гнучкість і адаптованість, а також спрощують її підтримку. Їх застосування полегшує комунікацію в команді, оскільки спільне розуміння шаблонів робить архітектуру зрозумілою для всіх учасників проєкту. Шаблони дозволяють уникнути "винаходу велосипеда", використовуючи перевірені практики, визнані спільнотою розробників, що економічно вигідно для бізнесу. Водночас їх використання повинно бути обґрунтованим, адже вони не є універсальним рішенням для всіх ситуацій.

Шаблон “Команда”



Призначення патерну Command

Шаблон «Команда» – це поведінковий шаблон проектування, який представляє запити у вигляді об'єктів, з можливістю передавати разом із запитом параметри, що визначатимуть поведінку команди.

Проблема

Шаблон «Команда» вирішує такі задачі:

- Необхідно передавати запити як об’єкти, щоб реалізувати відкладене виконання або динамічну зміну логіки.
- Забезпечити єдиний підхід до виконання різних дій, незалежно від того, який об'єкт їх викликає.

Рішення

Шаблон «Команда» інкапсулює кожну операцію у вигляді окремого класу, який реалізує спільний інтерфейс із методом execute()

Реалізувати не менше 3-х класів відповідно до обраної теми

Структура проекту з реалізованими класами зображена на рисунку 1.

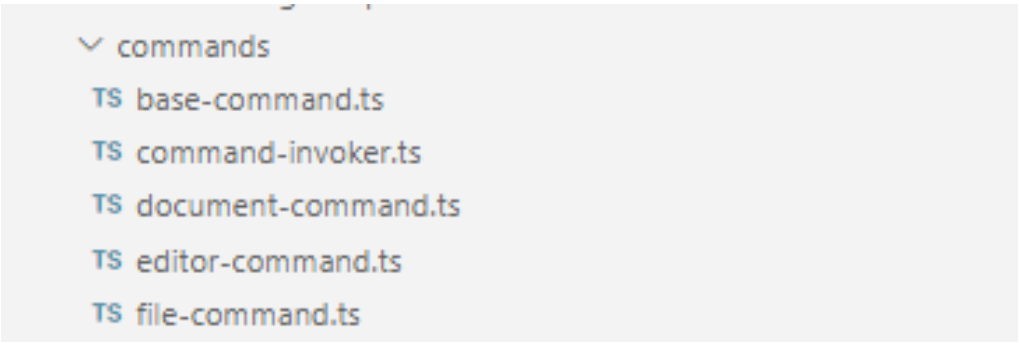


Рисунок 1 – структура проекту

Опис класів

В ході виконання лабораторної роботи були реалізовані наступні класи:

- **CommandInvoker** - виконує команди та відображає сповіщення про їх виконання, виступаючи центральною точкою для всіх операцій в застосунку.

```
@Injectable({
  providedIn: 'root'
})
export class CommandInvoker {
  private notification = inject(NotificationService);

  execute(command: BaseCommand): void {
    command.execute();

    this.notification.showNotification(command.getDescription());
  }
}
```

Рисунок 2 – клас CommandInvoker

- **FileImportCommand** - Відповідає за імпорт файлів у JSON-редактор, очищення завантажувача файлів і оновлення імені файлу.

```
export class FileImportCommand implements BaseCommand {

  constructor(
    private editor: JsonEditorComponent,
    private fileUpload: FileUpload,
    private event: FileSelectEvent,
  ) { }

  execute(): void {
    const file = this.event.currentFiles[0];
    const text = file?.text();

    this.fileUpload.clear();

    if (file) {
      const jsonTypeIndex = file.name.lastIndexOf('.json');
      const txtTypeIndex = file.name.lastIndexOf('.txt');
      const nameWithoutType = file.name.slice(0, Math.max(txtTypeIndex, jsonTypeIndex));

      new FileNameUpdateCommand(this.editor, nameWithoutType).execute();

      fromPromise(text).pipe(
        take(1),
        tap((content) => this.editor.control.setValue(content))
      ).subscribe();
    }
  }

  getDescription(): string {
    return `File "${this.event.currentFiles[0]?.name}" imported`;
  }
}
```

Рисунок 3 – клас FileImportCommand

- **SaveCommand** - Забезпечує збереження поточного стану JSON-редактора, делегуючи це відповідній службі.

```
export class SaveCommand implements BaseCommand {
  constructor(private editor: JsonEditorComponent) {
  }

  getDescription(): string {
    return 'Document content saved';
  }

  execute(): void {
    this.editor.saveService.saveState(this.editor.control.value);
  }
}
```

Рисунок 4 – клас SaveCommand

- **DocumentFormatCommand** – Виконує функцію форматування.

```
export class DocumentFormatCommand implements BaseCommand {

  constructor(
    private control: FormControl,
    private formatter: (value: string) => string
  ) {}

  execute(): void {
    this.control.setValue(this.formatter(this.control.value));
  }

  getDescription(): string {
    return 'Document formatted';
  }
}
```

Рисунок 5 – клас FormatCommand

- **LoadHistoryCommand** – відповідає за встановлення обраного користувачем значення з історії змін.

```
export class LoadHistoryCommand implements BaseCommand {
  constructor(private editor: JsonEditorComponent, private content: string) {
  }

  getDescription(): string {
    return 'Loaded content from history';
  }

  execute(): void {
    this.editor.control.setValue(this.content);
  }
}
```

Рисунок 6 – клас LoadHistoryCommand

Діаграма класів для паттерну «Команда»

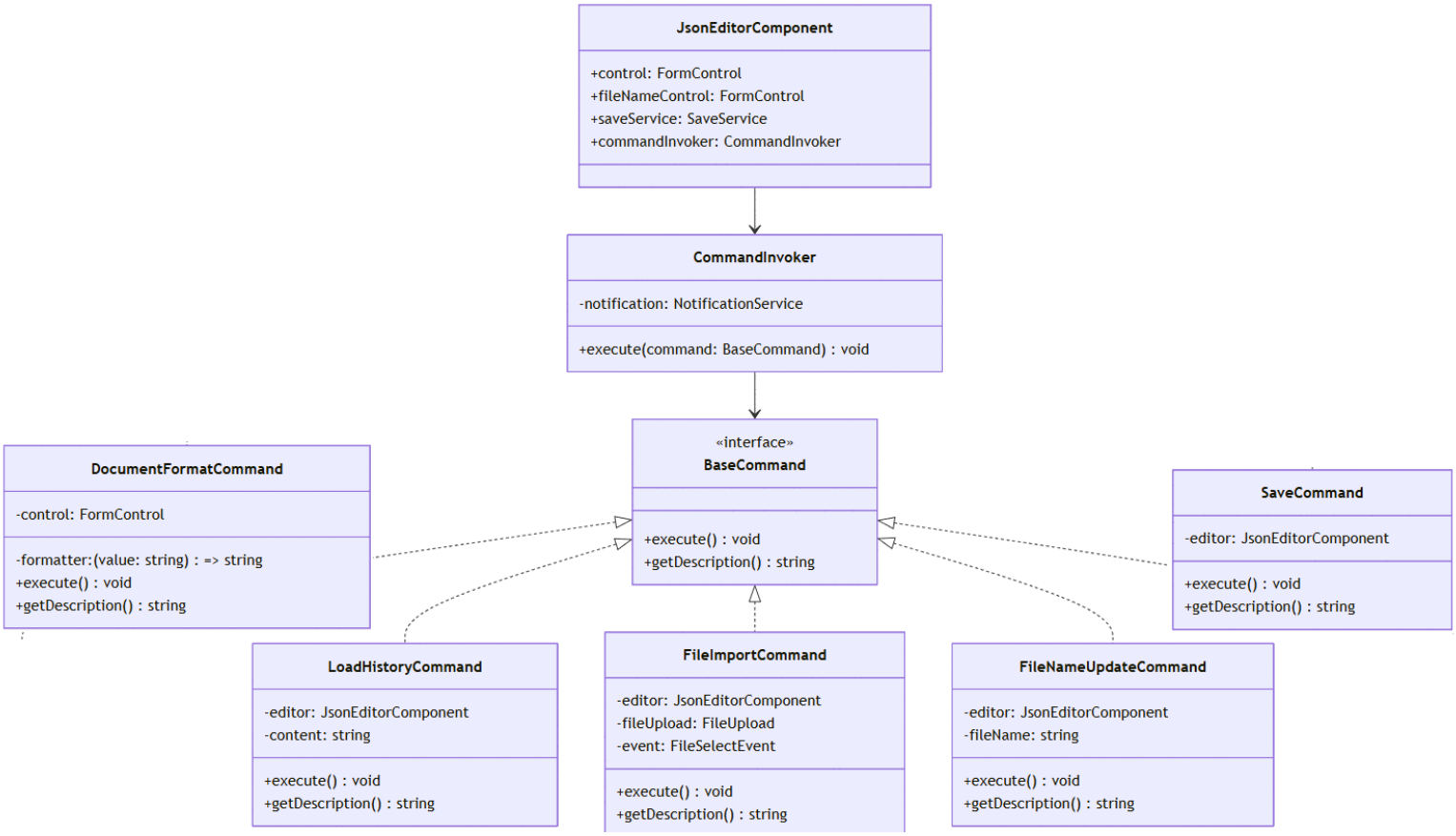


Рисунок 7 – діаграма реалізованих класів шаблону «Команда»

Переваги застосування патерну Command

1. **Інкапсуляція дій** - Кожна команда інкапсулює окрему дію, що полегшує управління логікою і тестування.
2. **Застосування принципу відкритості/закритості** - Нові команди можна додати без зміни існуючого коду, що знижує ризик виникнення помилок.
3. **Уникнення повторення коду**, та централізоване сповіщення користувача про виконані дії, за допомогою коду у класі **CommandInvoker**.

Посилання на репозиторій з кодом:

https://github.com/neodavis/json-tool

Висновок

Патерн «**Команда**» був реалізований для інкапсуляції запитів, таких як форматування, імпорт файлів та збереження даних. Це забезпечило модульність, легкість у підтримці та можливість динамічного виконання дій під час роботи програми. Шаблон дозволив уникнути жорсткої прив’язки логіки до компонентів, зробивши систему гнучкою та масштабованою.