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

із дисципліни «**Технології розроблення програмного забезпечення**»

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

Варіант №5. Аудіоредактор

Виконав
студент групи ІА–24
Криворучек В. С.

Перевірив
викладач
Мягкий М. Ю.

Київ 2024

Зміст

Мета: Вивчення основних принципів застосування шаблонів «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE» при створенні проєкту за індивідуальним варіантом

Тема (Варіант №5)

5 Аудіо редактор (singleton, adapter, observer, mediator, composite, client-server) Аудіо редактор повинен володіти наступним функціоналом: представлення аудіо даних будь-якого формату в WAVE-формі, вибір і подальші операції копіювання / вставки / вирізання / деформації по сегменту аудіозапису, можливість роботи з декількома звуковими доріжками, кодування в найбільш поширених форматах (ogg, flac, mp3).

Хід роботи

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

Теоретичні відомості

Анти-шаблони проєктування (anti-patterns) — це повторювані рішення проблем у розробці програмного забезпечення, які виявляються неефективними або шкідливими в довгостроковій перспективі. Вони виникають через недоліки у розумінні принципів проектування, тиск дедлайнів або недостатній досвід розробників. Анти-шаблони можуть призводити до зниження продуктивності, складності коду, проблем із масштабованістю та підтримкою.

Приклади анти-шаблонів:

1. Spaghetti Code — код без структури, важкий для розуміння.
2. God Object — об’єкт, який має занадто багато відповідальностей.
3. Golden Hammer — використання одного підходу для вирішення всіх проблем.
4. Magic Numbers — використання числових значень у коді без пояснення їхнього значення.

Анти-патерни в управлінні розробкою ПЗ:

- Дим і дзеркала: демонстрація незавершених функцій.
- Роздування ПЗ: збільшення вимог до ресурсів без виправданих змін.
- Функції для галочки: додавання непотрібних функцій для вигляду.

Анти-патерни в розробці ПЗ:

- Великий клубок бруду: складні і непідтримувані системи.
- Інверсія абстракції: приховування функціоналу без необхідності.
- Роздування інтерфейсу: занадто великі інтерфейси.

Анти-патерни в ООП:

- Божественний об'єкт: концентрація надмірної логіки в одному класі.

- Самотність (Singletonitis): надмірне використання патерну "Одинак".

Анти-патерни в програмуванні:

- Непотрібна складність: введення зайвої складності.
- Жорстке кодування: використання фіксованих значень в коді замість гнучких рішень.
- Спагеті-код: заплутаний і важкий для підтримки код.

Методологічні анти-патерни:

- Копіювання-вставка: копіювання коду замість створення спільних рішень.
- Передчасна оптимізація: оптимізація без достатньої інформації.

Adapter (Адаптер)

Призначення: дозволяє об'єктам із несумісними інтерфейсами працювати разом, забезпечуючи проміжний інтерфейс.

Коли використовувати:

- Коли потрібно узгодити інтерфейс існуючого класу з інтерфейсом, який очікує клієнт.
- Для роботи із застарілими класами без їх зміни.

Реалізація: створюється клас-адаптер, який трансформує інтерфейс одного класу в інтерфейс іншого.

Переваги: зменшує залежність від зовнішніх бібліотек, сприяє повторному використанню коду.

Недоліки: ускладнює структуру, якщо занадто багато адаптерів.

Builder (Будівельник)

Призначення: розділяє створення складного об’єкта на етапи, дозволяючи поетапно будувати різні його представлення.

Коли використовувати:

- Коли потрібно створювати складні об'єкти з великою кількістю параметрів.
- Для спрощення читабельності коду при конфігуруванні об’єктів.

Реалізація: використовується клас-будівельник, який крок за кроком створює об'єкт. Завершений об'єкт повертається через метод build().

Переваги: забезпечує гнучкість у створенні об'єктів, робить код зрозумілішим.

Недоліки: збільшує кількість класів.

Command (Команда)

Призначення: інкапсулює запит у вигляді об'єкта, дозволяючи відкладати виконання запиту, реєструвати його або підтримувати скасування операцій.

Коли використовувати:

- Для реалізації системи скасування та повтору дій.
- Для створення черги команд або логування.

Реалізація: створюється інтерфейс Command із методом execute(). Конкретні команди реалізують цей інтерфейс і виконують певну дію.

Переваги: знижує залежність між відправником і отримувачем запиту, дозволяє створювати складні макрокоманди.

Недоліки: ускладнює архітектуру через додаткові класи.

Chain of Responsibility (Ланцюг відповідальності)

Призначення: дозволяє передавати запити вздовж ланцюга обробників, поки один із них не виконає запит.

Коли використовувати:

- Коли потрібно уникнути тісного зв’язку між відправником і отримувачем запиту.

- Для реалізації динамічного оброблення запитів.

Реалізація: кожен обробник містить посилання на наступний обробник у ланцюгу. Запит передається від одного обробника до іншого.

Переваги: зменшує кількість залежностей між об’єктами, полегшує додавання нових обробників.

Недоліки: може бути важко відстежити, який обробник виконав запит.

Prototype (Прототип)

Призначення: дозволяє створювати нові об'єкти, копіюючи вже існуючі, замість створення об'єктів із нуля.

Коли використовувати:

- Коли створення об'єкта є дорогим або складним.
- Коли потрібно уникнути створення об’єктів через конструктор.

Реалізація: клас має метод clone(), який повертає копію об'єкта.

Переваги: швидке створення об'єктів, дозволяє уникати складної ініціалізації.

Недоліки: складність у копіюванні об'єктів із вкладеними залежностями.

Шаблон Singleton

Структура та обгрунтування вибору.

Шаблон "Адаптер" дозволяє об'єднувати дві системи з несумісними інтерфейсами, створюючи "перехідник" між ними. Це забезпечує можливість спільної роботи класів без необхідності змінювати їхній вихідний код.

Шаблон Adapter застосований тут для забезпечення сумісності між класами ієрархії Audiotrack та іншими компонентами програми або зовнішніми бібліотеками. Основна мета адаптера полягає в тому, щоб приховати специфічну реалізацію аудіотреків (Mp3, Flac, Ogg) і надати уніфікований інтерфейс для доступу до потрібних даних, таких як атрибути аудіо або файл аудіотреку.

Клас AudioAdapter виконує роль "перекладача", який дозволяє іншим компонентам працювати з об’єктами Audiotrack, не знаючи їхньої конкретної реалізації. Наприклад, метод adaptAttributes() повертає атрибути аудіо у форматі, очікуваному зовнішньою бібліотекою, а метод adaptFile() повертає об'єкт файлу.

Описану структуру показано на Рисунку 1.

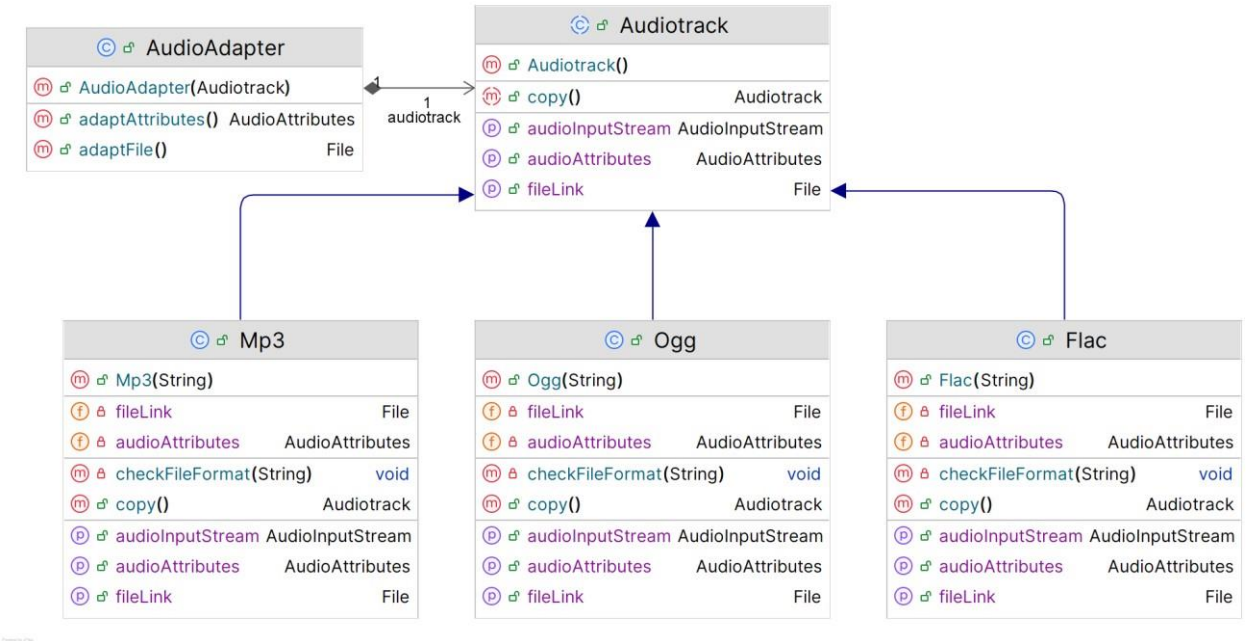

Рисунок 1. Структура класів реалізованого шаблону Adapter

Audiotrack — абстрактний клас, який визначає загальний інтерфейс для аудіоформатів, з методами для отримання атрибутів аудіо, файлів і копії треку. Класи Mp3, Ogg, і Flac наслідують його та реалізують ці методи для конкретних форматів.

AudioAdapter — адаптер, який містить об'єкт Audiotrack і надає зручний інтерфейс для отримання атрибутів і файлів з конкретних реалізацій Audiotrack, таких як Mp3, Ogg, або Flac. Він перетворює різні формати в єдиний, стандартизований інтерфейс, що дозволяє працювати з ними однотипно.

Реалізація функціоналу у коді з використанням шаблону

abstract class Audiotrack

```
public abstract class
Audiotrack {
    public abstract AudioAttributes getAudioAttributes();
public abstract File getFileLink();
    public abstract AudioInputStream getAudioInputStream();
public abstract Audiotrack copy();
}
```

class AudioAdapter

```
package org.example.audiotrack;

import
it.sauronsoftware.jave.AudioAttributes;
import java.io.File;

public class AudioAdapter{
    private final Audiotrack audiotrack;

    public AudioAdapter(Audiotrack audiotrack)
{          this.audiotrack = audiotrack;
    }       public AudioAttributes
adaptAttributes() {          return
audiotrack.getAudioAttributes();

}
    public File adaptFile() {
        return audiotrack.getFileLink();
} }
```

class Flac

```
package org.example.audiotrack;

import it.sauronsoftware.jave.AudioAttributes;

import
javax.sound.sampled.AudioInputStream;
import java.io.File; import
java.io.IOException; import
java.nio.file.Files; import
java.nio.file.Path;
  public class Flac extends Audiotrack
{     private AudioAttributes
audioAttributes;     private File
fileLink;
     public Flac(String
filePath) {
checkFileFormat(filePath);
        File audioFile = new File(filePath);

        fileLink = audioFile;
        audioAttributes = new AudioAttributes();
audioAttributes.setCodec("flac");
audioAttributes.setBitRate(128000);
audioAttributes.setChannels(2);
audioAttributes.setSamplingRate(44100);
    }
```

```
      private void checkFileFormat(String
path) {          try {
            if (!Files.probeContentType(Path.of(path)).equals("audio/x-
flac")) {                  throw new IllegalArgumentException("Wrong audio
format");              }
        } catch (IOException e) {
throw new RuntimeException(e);
        }

}
    public AudioAttributes getAudioAttributes() {
return audioAttributes;
    }
     public File
getFileLink() {
return fileLink;
    }

    @Override
    public AudioInputStream getAudioInputStream()
{          return null;
    }

    @Override
    public Audiotrack copy() {
return null;
    } }
```

class MP3

```
package org.example.audiotrack;

import it.sauronsoftware.jave.AudioAttributes;

import
javax.sound.sampled.AudioInputStream;
import java.io.File; import
java.io.IOException; import
java.nio.file.Files; import
java.nio.file.Path;
 public class Mp3 extends Audiotrack {
private AudioAttributes
audioAttributes;     private File
fileLink;
     public Mp3(String
filePath) {
checkFileFormat(filePath);
        File audioFile = new File(filePath);
fileLink = audioFile;
        audioAttributes = new AudioAttributes();
audioAttributes.setCodec("libmp3lame");
audioAttributes.setBitRate(128000);
audioAttributes.setChannels(2);
audioAttributes.setSamplingRate(44100);
    }

    private void checkFileFormat(String path)
{          try {
             if
(!Files.probeContentType(Path.of(path)).equals("audio/mpeg")) {
throw new IllegalArgumentException("Wrong audio format");              }
```

```
        } catch (IOException e) {
throw new RuntimeException(e);
        }

}
    public AudioAttributes getAudioAttributes() {
return audioAttributes;

}
    public File getFileLink() {
return fileLink;
    }

    @Override
    public AudioInputStream getAudioInputStream()
{          return null;
    }

    @Override
    public Audiotrack copy() {
return null;

}
}
```

class Ogg

```
package org.example.audiotrack;

import it.sauronsoftware.jave.AudioAttributes;

import
javax.sound.sampled.AudioInputStream;
import java.io.File; import
java.io.IOException; import
java.nio.file.Files; import
java.nio.file.Path;
 public class Ogg extends Audiotrack {
private AudioAttributes
audioAttributes;     private File
fileLink;
     public Ogg(String
filePath) {
checkFileFormat(filePath);
        File audioFile = new File(filePath);
fileLink = audioFile;
        audioAttributes = new AudioAttributes();
audioAttributes.setCodec("vorbis");
audioAttributes.setBitRate(128000);
audioAttributes.setChannels(2);
audioAttributes.setSamplingRate(44100);
    }

    private void checkFileFormat(String path)
{          try {
            if (!Files.probeContentType(Path.of(path)).equals("audio/ogg"))
{                 throw new IllegalArgumentException("Wrong audio format");
             }
        } catch (IOException e) {
throw new RuntimeException(e);
```

```
        }
    }
      public AudioAttributes
getAudioAttributes() {         return
audioAttributes;
    }

    public File getFileLink() {
return fileLink;
    }

    @Override
    public AudioInputStream getAudioInputStream()
{          return null;
    }

    @Override
    public Audiotrack copy() {
return null;

}
}
```

Висновок

У ході виконання лабораторної роботи було детально розглянуто п'ять важливих шаблонів проектування: Adapter, Builder, Command, Chain of Responsibility та Prototype. Кожен із цих шаблонів має свої особливості, переваги та сферу застосування, які дозволяють спрощувати, покращувати і робити більш гнучкою архітектуру програмного забезпечення.

Кожен із шаблонів вирішує специфічну задачу:

- Adapter забезпечує сумісність між несумісними інтерфейсами.
- Builder полегшує створення складних об'єктів, розділяючи процес побудови та його кінцеву структуру.
- Command інкапсулює запити як об'єкти, дозволяючи гнучко управляти виконанням дій.
- Chain of Responsibility передає запити через ланцюжок обробників, що дозволяє створити гнучку систему обробки.
- Prototype дозволяє створювати нові об'єкти на основі існуючих, копіюючи їх стан

Особливу увагу приділено шаблону Adapter, який було практично реалізовано у нашому проєкті Аудіоредактор. За допомогою цього шаблону вдалося забезпечити інтеграцію сторонньої бібліотеки для обробки аудіофайлів із власним інтерфейсом аудіоредактора. Адаптер дозволив уникнути змін у коді сторонньої бібліотеки, зменшив залежність від її деталей реалізації та забезпечив гнучкість у роботі з різними аудіо форматами.

Використання шаблонів проектування дозволяє вирішувати типові задачі проєктування ефективно та уніфіковано. Практичне застосування шаблону Adapter у нашому проєкті продемонструвало його важливість для інтеграції сторонніх рішень і підвищення модульності системи. Цей підхід допоможе зменшити складність підтримки та розширення проєкту в майбутньому.