

Министерство науки и высшего образования Российской Федерации

Севастопольский государственный университет

Высшая технологическая школа «Севастопольский приборостроительный
институт»

Факультет ИТ

ОТЧЁТ

по лабораторной работе №5

ИССЛЕДОВАНИЕ СПОСОБОВ ПРИМЕНЕНИЯ СТРУКТУРНЫХ ПАТТЕРНОВ
ПРОЕКТИРОВАНИЯ ПРИ РЕФАКТОРИНГЕ ПО

Выполнил:

ст. гр. ИС/б-22-2-о

Мовенко К. М.

Проверил:

Севастополь

2025

1. ЦЕЛЬ РАБОТЫ

Исследовать возможность использования структурных паттернов проектирования. Получить практические навыки применения структурных паттернов при объектно-ориентированном проектировании и рефакторинге ПО.

2. ЗАДАНИЕ

- 2.1. Ознакомиться с основными преимуществами объектно-ориентированного проектирования на основе паттернов, изучить порядок проектирования с использованием паттернов. Изучить назначение и структуру паттерна Адаптер;
- 2.2. Применительно к программному продукту, выбранному для рефакторинга, проанализировать возможность использования паттерна Адаптер. Для этого построить диаграмму классов, на диаграмме классов найти класс-клиент и адаптируемый класс, функциональностью которого должен воспользоваться клиент;
- 2.3. Выполнить перепроектирование системы, использовав паттерн Адаптер, изменения отобразить на диаграмме классов;
- 2.4. Сравнить полученные диаграммы классов, сделать выводы по целесообразности использования паттернов проектирования для данной системы;
- 2.5. На основе полученной UML-диаграммы модифицировать программный код, скомпилировать программу, выполнить её тестирование и продемонстрировать её работоспособность;

3. ХОД РАБОТЫ

Первым делом было проведено ознакомление с применением **структурных паттернов** в объектно-ориентированном программировании. Особое внимание было уделено паттерну **Адаптер**.

Далее был рассмотрен фрагмент кода для рефакторинга (листинг 3.1). Представлены два класса – `MediaPlayer` и `SoundDevice`. Оба позволяют «воспроизводить» задаваемую композицию.

Листинг 3.1 – Фрагмент кода для рефакторинга

```
#include <iostream>
#include <string>

using namespace std;

// Объект музыкального трека
class MusicTrack {
private:
    string author;           // автор
    string title;            // название
public:
    MusicTrack(const string& title, const string& author)
        : title(title), author(author) {}

    string getTitle() { return title; }
    string getAuthor() { return author; }
};

// Интерфейс устройства воспроизведения музыки
class AudioPlayer {
public:
    virtual void play(MusicTrack track) = 0;
    virtual ~AudioPlayer() = default;
};

// Проигрыватель музыки
class MediaPlayer : public AudioPlayer {
public:
    void play(MusicTrack track) {
        cout << "MediaPlayer plays: ";
        cout << track.getAuthor() << " - " << track.getTitle() << "\n\n";
    }
};

// Старый плеер с несовместимым интерфейсом
class SoundDevice {
public:
    void startPlayback(const string& trackName) {
        cout << "~~~ " << trackName << " ~~~" << endl;
        cout << "~~~ playback finished ~~~" << "\n\n";
    }
};
```

```

int main() {
    MusicTrack track("Despacito", "Luis Fonsi");

    AudioPlayer* player = new MediaPlayer();
    player->play(track);

    SoundDevice oldPlayer;
    oldPlayer.startPlayback(track.getAuthor() + " - " + track.getTitle());

    delete player;
    return 0;
}

```

Объект `MusicTrack` по очереди воспроизводится через каждое устройство. Плеер `SoundDevice` считаем «устаревшим», т.к. он не может напрямую работать с `MusicTrack` и требует дополнительных преобразований в вызове.

Предложение для рефакторинга – приспособить класс `SoundDevice` под более «современный» интерфейс `AudioPlayer`.

В качестве паттерна рефакторинга предлагается **Адаптер**. Он отлично подходит для ситуаций, где необходимо класс с одним интерфейсом приспособить для использования указателем на другой.

Для лучшего понимания структуры программы была построена **диаграмма классов** (рисунок 3.1).

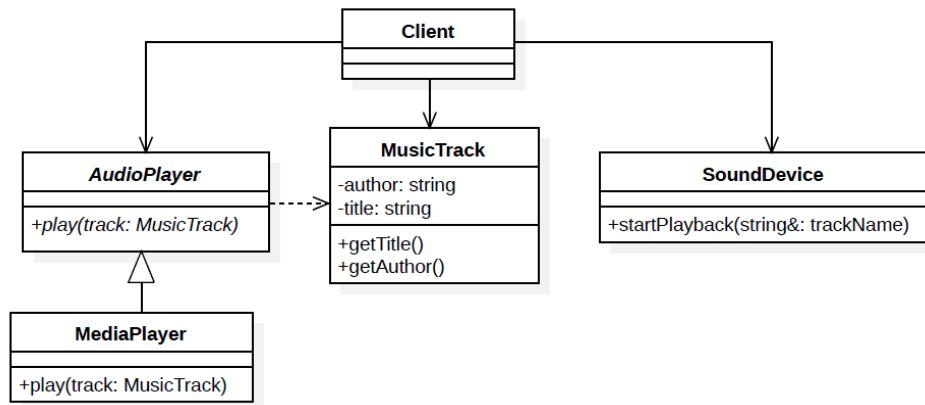


Рисунок 3.1 – Структура классов до рефакторинга

По диаграмме можно определить участников предстоящего рефакторинга. Клиент (функция `main`) обращается к целевому интерфейсу `AudioPlayer`. Цель рефакторинга – адаптировать интерфейс `SoundDevice`.

С учётом этого структура классов была перестроена таким образом, чтобы клиент взаимодействовал с `SoundDevice` через адаптер (рисунок 3.2).

В диаграмму был добавлен адаптер класса – `LegacyPlayer`. Он наследует интерфейс от `AudioPlayer` и реализацию от `SoundDevice`.

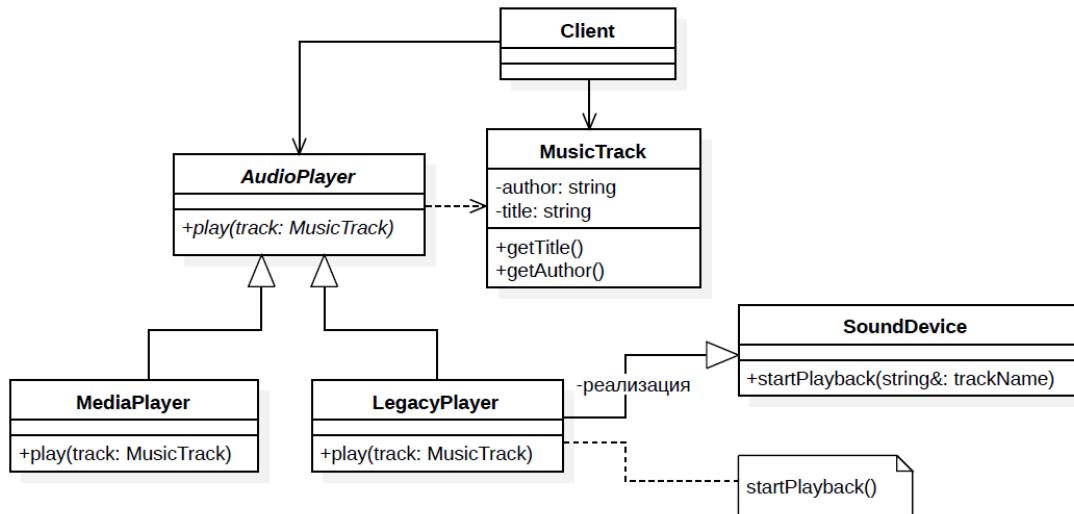


Рисунок 3.2 – Структура адаптера класса

Использование адаптера позволяет проигрывать трек на различных плеерах через единообразный интерфейс. Благодаря этому у клиента нет нужды вникать в специфику реализации плеера.

Код программы был модифицирован с учётом проведённого рефакторинга (листиング 3.2).

Листинг 3.2 – Код с применением Адаптера

```

#include <iostream>
#include <string>

using namespace std;

// Объект музыкального трека
class MusicTrack {
private:
    string author;           // автор
    string title;            // название
public:
    MusicTrack(const string& title, const string& author)
        : title(title), author(author) {}

    string getTitle() { return title; }
    string getAuthor() { return author; }
}
  
```

```

};

// Интерфейс устройства воспроизведения музыки
class AudioPlayer {
public:
    virtual void play(MusicTrack track) = 0;
    virtual ~AudioPlayer() = default;
};

// Проигрыватель музыки
class MediaPlayer : public AudioPlayer {
public:
    void play(MusicTrack track) {
        cout << "MediaPlayer plays: ";
        cout << track.getAuthor() << " - " << track.getTitle() << "\n\n";
    }
};

// Старый плеер с несовместимым интерфейсом
class SoundDevice {
public:
    void startPlayback(const string& trackName) {
        cout << "~~~ " << trackName << " ~~~" << endl;
        cout << "~~~ playback finished ~~~" << "\n\n";
    }
};

// Адаптер: подгоняет SoundDevice под AudioPlayer
class LegacyPlayer : public AudioPlayer, private SoundDevice {
public:
    void play(MusicTrack track) {
        string trackname = track.getAuthor() + " - " + track.getTitle();
        startPlayback(trackname);
    }
};

int main() {
    MusicTrack track("Despacito", "Luis Fonsi");

    AudioPlayer* player = new MediaPlayer();
    player->play(track);

    AudioPlayer* oldPlayer = new LegacyPlayer();
    oldPlayer->play(track);

    delete player;
    delete oldPlayer;

    return 0;
}

```

При запуске программа показала свою производительность (рисунок 3.3).

```
C:\Users\kosta\Documents\Лычко\Рабочий стол\7 ёхъхё\Личные\MediaPlayer plays: Luis Fonsi - Despacito
MediaPlayer plays: Luis Fonsi - Despacito
~~~ Luis Fonsi - Despacito ~~~
~~~ playback finished ~~~

-----
Process exited after 0.1266 seconds with return value 0
Для продолжения нажмите любую клавишу . . .
```

Рисунок 3.3 – Результат работы программы

4. ВЫВОД

В ходе работы было проведено ознакомление с принципами применения паттернов при проектировании структуры ПО. Были рассмотрены структурные паттерны проектирования уровня класса.

Отдельно был изучен паттерн Адаптер, его классовая и объектная вариации. В практических целях был проведён структурный анализ кода и его модификация с использованием адаптера уровня класса.

Полученные знания позволяют создавать гибкие деревья классов, расширять функциональность классов за счёт адаптации к новым интерфейсам.