|  |
| --- |
| Пермский филиал федерального государственного автономного образовательного учреждения высшего образования «Национальный исследовательский университет  «Высшая школа экономики»  Факультет экономики, менеджмента и бизнес-информатики |
| Образовательная программа бакалавриата «Программная инженерия» |
| **Лабораторная работа № 2**  **«Структурный паттерн проектирования Адаптер (Adapter)»**   |  |  | | --- | --- | |  | Выполнила:  студентка группы ПИ-15-1  Жигалова Светлана Андреевна | |  | Проверил:  преподаватель кафедры информационных технологий в бизнесе  Кычкин Алексей Владимирович | |

Пермь, 2018 год

Оглавление

[1. Назначение и цель применения паттерна 3](#_Toc506840629)

[2. UML-диаграмма паттерна и пояснение основных элементов 3](#_Toc506840630)

[Адаптер объектов 3](#_Toc506840631)

[Адаптер классов 4](#_Toc506840632)

[3. Области применения и особенности паттерна 4](#_Toc506840633)

[4. Реализация паттерна 5](#_Toc506840634)

[Шаги реализации 5](#_Toc506840635)

[Пример реализации 5](#_Toc506840636)

[Участники: 6](#_Toc506840637)

# Назначение и цель применения паттерна

Паттерн Адаптер (Adapter) предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.  
Когда надо использовать Адаптер?

- Когда необходимо использовать имеющийся класс, но его интерфейс не соответствует потребностям;  
- Когда надо использовать уже существующий класс совместно с другими классами, интерфейсы которых не совместимы.

# UML-диаграмма паттерна и пояснение основных элементов

## Адаптер объектов



***Клиент*** — это класс, который содержит существующую бизнес-логику программы.

***Клиентский интерфейс*** описывает протокол, через который клиент может работать с другими классами.

***Сервис*** – это какой-то полезный класс, обычно сторонний. Клиент не может использовать этот класс напрямую, так как сервис имеет непонятный ему интерфейс.

***Адаптер*** — это класс, который может одновременно работать и с клиентом, и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит их в вызовы методов обёрнутого объекта в правильном формате.

Работая с адаптером через интерфейс, клиент не привязываться к конкретному классу адаптера. Благодаря этому, вы можете добавлять в программу новые виды адаптеров, независимо от клиентского кода. Это может пригодиться, если интерфейс сервиса вдруг изменится, например, после выхода новой версии сторонней библиотеки.

## Адаптер классов

Эта реализация базируется на наследовании: адаптер наследует оба интерфейса одновременно. Такой подход возможен только в языках, поддерживающих множественное наследование, например C++.



***Адаптер классов*** не нуждается во вложенном объекте, так как он одновременно наследует и существующий и сервисный интерфейс.

# Области применения и особенности паттерна

1. Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.
2. Адаптер позволяет создать объект-прокладку, который будет превращать вызовы приложения в формат, понятный стороннему классу.
3. Когда вам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности. Причём расширять суперкласс вы не можете.
4. Вы могли бы создать ещё один уровень подклассов, и добавить в них недостающую функциональность. Но при этом придётся дублировать один и тот же код в обеих ветках подклассов.  
   Более элегантное решение — поместить недостающую функциональность в адаптер и приспособить его для работы с суперклассом. Такой адаптер сможет работать со всеми подклассами иерархии.

|  |  |
| --- | --- |
| **Преимущества** | **Недостатки** |
| Отделяет и скрывает от клиента подробности преобразования различных интерфейсов. | Усложняет код программы за счёт дополнительных классов. |

# Реализация паттерна

## Шаги реализации

1. Убедитесь, что у вас есть два класса с неудобными интерфейсами:

- полезный *сервис* — служебный класс, который вы не можете изменять (он либо сторонний, либо от него зависит другой код);

- один или несколько *клиентов* — классов приложения, несовместимых с сервисом из-за неудобного или несовпадающего интерфейса.

1. Опишите клиентский интерфейс, через который классы приложения смогли бы использовать сторонний класс.
2. Создайте класс адаптера, реализовав этот интерфейс.
3. Поместите в адаптер поле-ссылку на объект-сервис. В большинстве случаев, это поле заполняется объектом, переданным в конструктор адаптера. В случае простой адаптации этот объект можно передавать через параметры методов адаптера.
4. Реализуйте все методы клиентского интерфейса в адаптере. Адаптер должен делегировать основную работу сервису.
5. Приложение должно использовать адаптер только через клиентский интерфейс. Это позволит легко изменять и добавлять адаптеры в будущем.

## Пример реализации

Классическая реализация данного шаблона проектирования на C# выглядит следующим образом:

class Client

{

    public void Request(Target target)

    {

        target.Request();

    }

}

// класс, к которому надо адаптировать другой класс

class Target

{

    public virtual void Request()

    {}

}

// Адаптер

class Adapter : Target

{

    private Adaptee adaptee = new Adaptee();

    public override void Request()

    {

        adaptee.SpecificRequest();

    }

}

// Адаптируемый класс

class Adaptee

{

    public void SpecificRequest()

    {}

}

### Участники:

* Target: представляет объекты, которые используются клиентом
* Client: использует объекты Target для реализации своих задач
* Adaptee: представляет адаптируемый класс, который мы хотели бы использовать у клиента вместо объектов Target
* Adapter: собственно адаптер, который позволяет работать с объектами Adaptee как с объектами Target.

То есть клиент ничего не знает об Adaptee, он знает и использует только объекты Target. И благодаря адаптеру мы можем на клиенте использовать объекты Adaptee как Target.

Конкретный пример реализации.

Допустим, у нас есть путешественник, который путешествует на машине. Но в какой-то момент ему приходится передвигаться по пескам пустыни, где он не может ехать на машине. Зато он может использовать для передвижения верблюда. Однако в классе путешественника использование класса верблюда не предусмотрено, поэтому нам надо использовать адаптер:

class Program

{

    static void Main(string[] args)

    {

        // путешественник

        Driver driver = new Driver();

        // машина

        Auto auto = new Auto();

        // отправляемся в путешествие

        driver.Travel(auto);

        // встретились пески, надо использовать верблюда

        Camel camel = new Camel();

        // используем адаптер

        ITransport camelTransport = new CamelToTransportAdapter(camel);

        // продолжаем путь по пескам пустыни

        driver.Travel(camelTransport);

        Console.Read();

    }

}

interface ITransport

{

    void Drive();

}

// класс машины

class Auto : ITransport

{

    public void Drive()

    {

        Console.WriteLine("Машина едет по дороге");

    }

}

class Driver

{

    public void Travel(ITransport transport)

    {

        transport.Drive();

    }

}

// интерфейс животного

interface IAnimal

{

    void Move();

}

// класс верблюда

class Camel : IAnimal

{

    public void Move()

    {

        Console.WriteLine("Верблюд идет по пескам пустыни");

    }

}

// Адаптер от Camel к ITransport

class CamelToTransportAdapter : ITransport

{

    Camel camel;

    public CamelToTransportAdapter(Camel c)

    {

        camel = c;

    }

    public void Drive()

    {

        camel.Move();

    }

}

В данном случае в качестве клиента применяется класс Driver, который использует объект ITransport. Адаптируемым является класс верблюда Camel, который нужно использовать в качестве объекта ITransport. И адптером служит класс CamelToTransportAdapter.