СТРУКТУРНЫЕ ПАТТЕРНЫ(АДАПТЕР)

АДАПТЕР

Адаптер – это структурный паттерн проектирования, который позволяет объектам ***с несовместимыми интерфейсами работать вместе.***

Адаптер - объект который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого.

1. ***Адаптер*** имеет ***интерфейс***, который совместим с одним из объектов.

2. Поэтому этот объект может свободно вызывать методы адаптера.

3. Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.

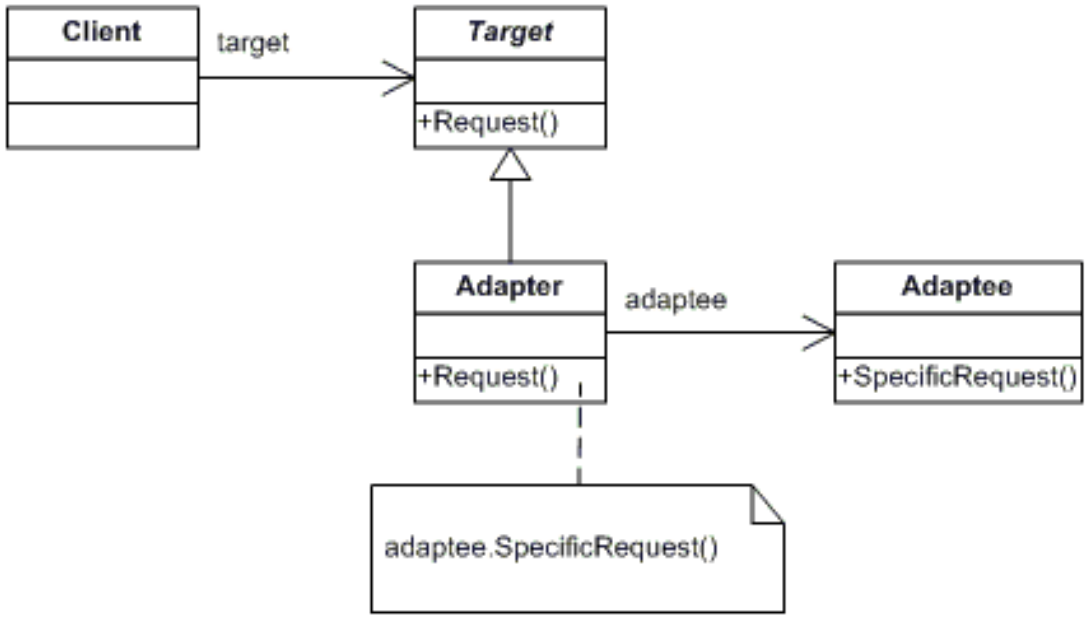
Иногда возможно создать даже *двухсторонний адаптер*, скоторый работал бы в обе стороны.

АДАПТЕР – UML ДИАГРАММА

Tagert: это интерфейс, с которым взаимодействует ***клиент***.

Adaptee: это ***интерфейс***, с которым клиент хочет взаимодействовать, но не может взаимодействовать без помощи адаптера.

Adapter: наследуется от **Target** и содержит объект Adaptee.



АДАПТЕР – ПРИМЕР

В качестве примера, создадим небольшой клиент, который будет использовать две фиктивные библиотеки для выполнения некоторых операций.

Эти две библиотеки предоставляют разные интерфейсы, поэтому, для того чтобы использовать их напишем небольшой адаптер.

Рассмотрим интерфейс, с которым клиент хочет взаимодействовать, но не может взаимодействовать без помощи адаптера(**Adaptee**).

```
class LibraryA
{public:
   void SomeActionFromLibraryA(){
            std::cout<< "Using Library A to perform the action\n";
        }
};

class LibraryB
{public:
    std::string SomeActionFromLibraryB() {
                    return "Using Library B to perform the action\n";
        }
};
```

АДАПТЕР – ПРИМЕР

Далее, следуя схеме UML, напишем интерфейс адаптера для наших конкретных адаптеров(**Target**).

```
class IAdapter
{public:
 virtual void DoAction() = 0;
};
```

Теперь напишем конкретные классы Adapter для использования двух библиотек.

```
class AdapterLibA : public IAdapter
{public:
        void DoAction() {
                LibraryA one;
                one.SomeActionFromLibraryA();}};
```

```
class AdapterLibB : public IAdapter
{public:
        void DoAction() {
                LibraryB two;
                std::cout << two.SomeActionFromLibraryB();
        }};
```

АДАПТЕР – ПРИМЕР

Теперь у нас есть единый интерфейс, который может использовать Клиент. Соответствующие адаптеры позаботятся о выполнении вызовов соответствующих базовых объектов. Рассмотрим как Клиент может использовать этот адаптер.

```
int main()
{
IAdapter *adapter = 0;
cout <<  "Укажите, какую библиотеку вы хотите использовать для выполнения
операции{ 1,2 }";
int x;
cin >> x;
if (x == 1){
  adapter = new AdapterLibA();}
else if (x == 2) {
adapter = new AdapterLibB();
}
//Выполним операцию
adapter->DoAction();
delete adapter;
return 0;
}
```

АДАПТЕР – ПРИМЕР

Теперь клиент может использовать один и тот же интерфейс для выполнения операций с использованием обоих базовых объектов.

Паттерн Adapter особенно полезен, когда имеется два класса, которые выполняют сходные функции, но имеют разные интерфейсы.

Рассмотрим диаграмму классов, предложенного примера и сопоставим с UML диаграммой Адаптера.