МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Кафедра вычислительной математики и программирования

**Отчет**

**по курсу «Объектно-ориентированная разработка»**

Выполнила: Алексюнина Ю.В.

Группа: М8О-106М

Преподаватели: Аносова Н.П.

Москва, 2022

### Постановка задачи

Реализовать основные паттерны проектирования:

* MVC
* Abstract Factory
* Adapter
* Builder
* Chain of Responsibility
* Command
* Composite
* Facade
* Iterator
* Mediator
* Memento
* Observer
* Prototype
* Proxy
* Singleton
* State
* Strategy
* Visitor

Об одном из паттернов (Adapter) рассказать подробнее.

### Описание решения

Паттерн Адаптер относится к структурным паттернам проектирования и если используется в случаях, когда необходимо преобразовать интерфейс одного класса в интерфейс другого. Например, есть уже написанные классы с нужной для выполнения текущей задачи функциональностью, но у них несовместимые интерфейсы. Тогда в игру вступает Адаптер, превращая вызовы одного класса в вызовы, понятные другому.

Адаптер часто используется, если мы работаем со сторонней библиотекой или компонентом, доступ к изменениям методов которого у нас отсутствует. Или когда есть несколько сторонних систем, доступ к которым должен осуществляться единообразно через одинаковый интерфейс.

Для реализации паттерна Адаптер могут быть созданы следующие классы:

1. Целевой класс, который представляет объекты, использующиеся клиентом
2. Клиентский класс - описывает методы, через которые клиент может работать с другими классами
3. Адаптируемый класс, который мы хотели бы использовать у клиента вместо объектов целевого класса. Клиент не может использовать этот класс напрямую, так как он имеет непонятный клиенту интерфейс.
4. Класс-адаптер, который позволяет работать с объектами адаптируемого класса как с объектами целевого. Он реализует клиентский интерфейс и содержит ссылку на объект адаптируемого класса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит их в вызовы методов обёрнутого объекта в правильном формате.

Плюсы и минусы паттерна Адаптер:

**Плюсы:**

* Инкапсуляция реализации внешних классов (компонентов, библиотек), система становится независимой от интерфейса внешних классов;
* Переход на использование других внешних классов не требует переделки самой системы, достаточно реализовать только класс-адаптер;

**Минусы:**

* Усложняется при сильном отличии адаптируемого объекта;
* Затруднено замещение операций адаптируемого класса;
* Усложняет код программы из-за введения дополнительных классов;

Рассмотрим и затем реализуем пример применения паттерна Адаптер.

Представим, что у нас есть отель. Пусть в нашем отеле разработали новую систему входа в номера для обеспечения безопасности гостей отеля. Но злоумышленники хотят взломать номера в отеле, чтобы выкрасть все драгоценности. Для этого они взломали сейф в кабинете управляющего, и нашли подсказку в виде некоторого класса с соответствующим интерфейсом. Однако использовать этот класс непосредственно не удастся, так как им нужно понять сам шифр, и нужен адаптер.

Создадим три класса: абстрактный целевой – с которым может работать код клиента (в данном случае - бандиты), адаптируемый – нуждающийся в доработке(подсказка) и класс адаптера, который поможет разгадать тайну бандитам.

В целевом – создаем виртуальный метод запроса, который переопределим в адаптере и оставляем его. Адаптируемый класс содержит метод с подсказкой. А в классе адаптера, который мы наследуем от целевого класса, загружаем объект адаптируемого класса и переопределяем метод запроса, декодируя шифр.

#include <iostream>

#include<string>

using namespace std;

class Target {

public:

virtual string Request() = 0;

virtual ~Target() {};

};

class Adaptee {

public:

string SpecificRequest() const {

return "Lw lv wkh Zxhvdu zlskhu";

}

};

class Adapter : public Target {

private:

Adaptee\* adaptee\_;

public:

Adapter(Adaptee\* adaptee) : adaptee\_(adaptee) {}

string Request() override {

string decoded = this->adaptee\_->SpecificRequest();

for (int i = 0; i < decoded.size(); i++) {

if (decoded[i] == ' ')

continue;

else if (decoded[i] == 'x')

decoded[i] = 'a';

else if (decoded[i] == 'X')

decoded[i] = 'A';

else if (decoded[i] == 'Y')

decoded[i] = 'B';

else if (decoded[i] == 'y')

decoded[i] = 'b';

else if (decoded[i] == 'z')

decoded[i] = 'c';

else if (decoded[i] == 'Z')

decoded[i] = 'C';

else

decoded[i] -= 3;

}

return "Adapter: " + decoded;

}

};

void ClientCode(Target\* target) {

cout << target->Request();

}

int main() {

Adaptee\* adaptee = new Adaptee;

cout << "Bandit: I can't open the door, it's encrypted!" << endl;

cout << "Adaptee: " << adaptee->SpecificRequest();

cout << endl << endl;

cout << "Admin: But you can work with it via the Adapter" << endl;

Adapter\* adapter = new Adapter(adaptee);

ClientCode(adapter);

cout << endl << endl;

cout << "Bandit: Thanks, I won't kill you...now" << endl;

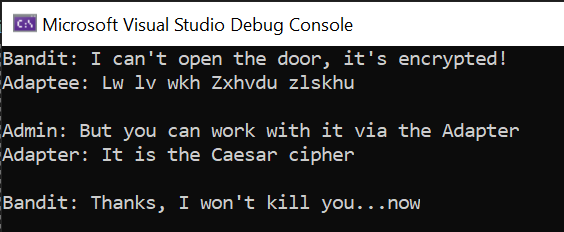
delete adaptee;

delete adapter;

return 0;

}

### Результаты



С исходным кодом других паттернов проектирования можно ознакомиться на Github: <https://github.com/xoxoginger/OOP_Patterns>

### Выводы

Паттерны проектирования — это один из инструментов разработчика, который помогает ему сэкономить время и сделать более качественное решение. Даже теоретические знания шаблонов проектирования помогают понять чужой код гораздо быстрее и, соответственно, они необходимы в работе над проектом даже если у вас нет коммерческого опыта работы.