




Тема: Базовые принципы проектирования (SOLID)

- 
- Что такое хороший дизайн?
 - По каким критериям его оценивать и каких правил придерживаться при разработке?
 - Как обеспечить достаточный уровень гибкости, связанности, управляемости, стабильности и понятности кода?

Рассмотрим универсальные принципы проектирования.

Принципы S.O.L.I.D

Рассмотрим пять принципов проектирования, которые известны как **SOLID**.

Эти принципы были впервые изложены **Робертом Мартином** в книге *Agile Software Development, Principles, Patterns, and Practices*.

Термин **SOLID** — это аббревиатура, за каждой буквой которой стоит отдельный принцип проектирования.

Главная цель этих принципов — повысить

гибкость вашей архитектуры,

уменьшить связанность между её компонентами и облегчить повторное использование кода.

Но, соблюдение этих принципов имеет свою цену, которая выражается в ***усложнении кода программы.***

В реальной жизни, нет таких решений, в которых бы соблюдались все эти принципы сразу.

Принцип единственной ответственности

Single Responsibility Principle

Правило:

У класса должен быть только один мотив для изменения

Нужно стремиться к тому, чтобы каждый класс отвечал только за одну часть функциональности программы, причём она должна быть полностью инкапсулирована в этот класс (*скрыта внутри класса*).

Принцип единственной ответственности предназначен **для борьбы со сложностью**.

Если класс делает слишком **много вещей сразу**, то приходится изменять его каждый раз, когда одна из этих вещей изменяется.

При этом есть риск разрушить остальные части класса.

Хорошо иметь возможность сосредоточиться на сложных аспектах системы по отдельности.

Но если по отдельности не получается, то лучше применить принцип единственной ответственности, разделяя классы на части.

Что такое ответственность ?!

Ответственность может быть определена как причина изменения.

Всякий раз, когда мы думаем, что некоторая часть нашего кода потенциально является ответственностью, мы должны рассмотреть возможность отделения его от класса.

Если у нас есть **две причины** для изменения класса, нам нужно разделить функциональность на **два класса**.

Каждый класс будет обрабатывать только **одну ответственность**, и в будущем, если мы хотим сделать одно изменение, мы собираемся сделать это в классе, который его обрабатывает.

Когда нам нужно внести изменение в класс, имеющий больше обязанностей, это изменение может повлиять на другие функциональные возможности классов

Принцип единственной ответственности

Single Responsibility Principle

Пример:

Рассмотрим класс `Book`

```
class Book {  
private:  
    string name;  
    string author;  
    string text;  
public:  
    //constructor, getters and setters  
    // methods that directly relate to the book properties  
    string replaceWordInText(string word) {  
        return text.replace(text.begin(), text.end(), word);  
    }  
    bool isWordInText(string word) {  
        return text.find(word);  
    }  
};
```


Принцип единственной ответственности

Single Responsibility Principle

Пример:

Рассмотрим класс `Book`

```
class Book {  
private:  
    string name;  
    string author;  
    string text;  
public:  
  
    ...  
  
    void printTextToConsole() {  
        // our code for formatting and printing the text  
    }  
  
};
```

Принцип единственной ответственности

Single Responsibility Principle

Пример:

Рассмотрим класс `Book`

```
class BookPrinter {  
  
    // methods for outputting text  
    void printTextToConsole(string text) {  
        //our code for formatting and printing the text  
    }  
  
    void printTextToAnotherMedium(string text) {  
        // code for writing to any other location..  
    }  
};
```

Таким образом, разработан класс, который освобождает `Book` от своих обязанностей по печати, но также можно использовать наш класс `BookPrinter` для отправки нашего текста на другие носители.

Принцип единственной ответственности

Single Responsibility Principle

Пример:

Рассмотрим класс `Email`, `EmailSender`

```
class Email
{public:
    //constructors, getter, setter
private:
    string Theme;
    string From;
    string To;
};
class EmailSender
{public:
    void Send(Email email)
    {
        // ... sending...
        cout<<"Email from '" + email.getFrom + "' to '" + email.getTo + "'
was send");
    }
};
```

Принцип единственной ответственности

Single Responsibility Principle

Пример:

Класс `EmailSender`, кроме того, что при помощи метода `Send`, он отправляет сообщения, он еще и решает как будет вестись лог.

В данном примере лог ведется через консоль.

Если случится так, что придется менять способ логирования, то придется вносить правки в класс `EmailSender`.

Хотя, казалось бы, эти правки не касаются отправки сообщений.

Очевидно, `EmailSender` выполняет несколько обязанностей и, чтобы класс не был привязан только к одному способу вести лог, нужно вынести выбор лога из этого класса.

Принцип единственной ответственности

Single Responsibility Principle

Пример:

```
class ILog
{
    virtual void Write(string str)= 0;
};

class ConsoleLog : ILog
{
public:
    void Write(string str)
    {
        cout<<str<<endl;
    }
};
```

Принцип единственной ответственности

Single Responsibility Principle

Пример:

```
class EmailSender
{
public:
    EmailSender(ILog* log)
    {
        _log = log;
    }
    void Send(Email email)
    {
        // ... sending...
        _log->Write("Email from '" + email.From + "' to '" + email.To
+ "' was send");
    }
private:
    ILog* _log;
};
```


Принцип единственной ответственности

Single Responsibility Principle

Пример:

Рассмотрим класс `Employee`(сотрудник)

```
class Employee
{
    int employee_Id;
    std::string employeeName;
public:
    // Этот метод вставляет работника в таблицу некоторой БД

    bool InsertIntoEmployeeTable(Employee &em);
    // Этот метод генерирует отчет относительно заданного работника
    void GenerateReport(Employee &em);
};
```

Принцип единственной ответственности

Single Responsibility Principle(SRP)

Класс «Сотрудник» берет на себя 2 обязанности:

1. Работа с базой данных сотрудников;
2. создание отчета о сотрудниках.

Класс «Сотрудник» *не должен брать на себя ответственность за генерацию отчета.*

Почему?

Предположим, через некоторое время потребовалось, например, предоставить средство для генерации отчета в Excel или любом другом формате отчетности, следовательно этот класс нужно будет изменить, и это не очень хорошо.

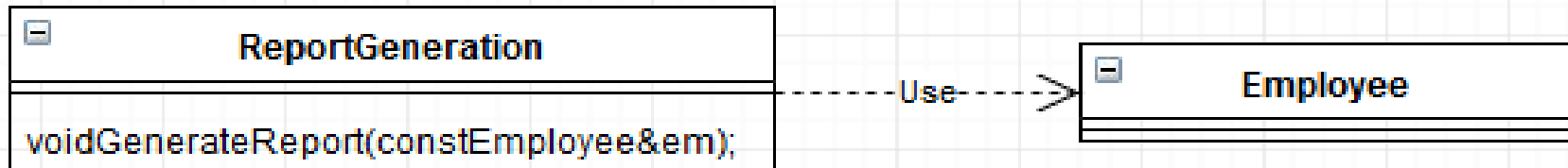
Принцип единственной ответственности

Single Responsibility Principle(SRP)

Таким образом, согласно SRP, один класс должен нести одну ответственность, поэтому необходимо описать другой класс для генерации отчетов, чтобы любые изменения в *генерации отчетов* не влияли на класс «Сотрудник».

Например.

```
class ReportGeneration
{ // Метод генерации отчета
public:
void GenerateReport(const Employee &em);
};
```



Принцип открытости/закрытости

Open/closed Principle

Правило.

Расширяйте классы, но не изменяйте их первоначальный код.

Нужно стремиться к тому, чтобы классы были открыты для расширения, но закрыты для изменения.

Главная идея этого принципа в том, чтобы не разрушать существующий код при внесении изменений в программу.

Класс можно назвать **открытым**, если он доступен для расширения.

Класс можно назвать **закрытым** (можно сказать законченным), если он готов для использования другими классами. Это означает, что интерфейс класса уже окончательно определён и не будет изменяться в будущем.

Принцип открытости/закрытости

Open/closed Principle

Если класс уже был написан, одобрен, протестирован, возможно, внесён в библиотеку и включён в проект, после этого пытаться модифицировать его содержимое нежелательно.

Вместо этого вы можете создать подкласс и расширить в нём базовое поведение, не изменяя код родительского класса напрямую.

Выгода от использования такой практики очевидна. Не нужно пересматривать уже существующий код, не нужно менять уже готовые для него тесты.

Если нужно ввести какую-то дополнительную функциональность, то это не должно коснуться уже существующих классов или как-либо иначе повредить уже существующую функциональность.

Принцип открытости/закрытости

Open/closed Principle

В качестве пример рассмотрим следующий класс **ReportGeneration**

В чем может быть здесь проблема?

```
class ReportGeneration
{
    std::string ReportType;
    // Метод генерации отчета
public:
    void GenerateReport(const Employee &em)
    {
        if (ReportType == "CRS")
        {
            // Генерация отчетов с данными о сотрудниках в формате CSV.
        }
        if (ReportType == "PDF")
        {
            // Генерация отчетов с данными о сотрудниках в формате PDF.
        }
    }
};
```

Принцип открытости/закрытости

Open/closed Principle(OCP)

Основная проблема здесь, это слишком много условий «if».

То есть, если потребуется ввести другой новый тип отчета, например «Excel», то нужно будет написать еще одно условие.

По принципу OCP этот класс должен быть **открыт** для расширения, но **закрит** для модификации.

Но как это сделать !!!

Принцип открытости/закрытости

Open/closed Principle(OCP)

```
class IReportGeneration
{ // Метод генерации отчета
public:
    virtual void GenerateReport(const Employee &em)=0;
};

class CSVReportGeneraion : public IReportGeneration
{ public:
    //Переопределяем метод, в котром генерируется отчет в формате CSV
    void GenerateReport(const Employee &em);
};

class PDFReportGeneraion : public IReportGeneration
{ public:
    //Переопределяем метод, в котром генерируется отчет в формате PDF
    void GenerateReport(const Employee &em);
};
```

В предложенном решении, если потребуется ввести новый тип отчета, то достаточно будет наследовать его от IReportGeneration. Таким образом, IReportGeneration открыт для расширения, но закрыт для модификации.

Принцип открытости/закрытости

Open/closed Principle(OCP)

```
class Report
{
    public:
        unsigned Total;
        string Name;
        time_t CreateDate;

        Report(unsigned Total, string Name, time_t CreateDate);

        string ToString()
        {
            //build formatted string with values  Total,Name, CreateDate
        }
};
```

Принцип открытости/закрытости

Open/closed Principle(OCP)

```
class ReportProcessor
{
public:
    ReportProcessor()
    {
        _reports.push_back(Report(120, "Sum", time(0)));
        _reports.push_back(Report(120, "Sum", time(0)));
        _reports.push_back(Report(120, "Sum", time(0)));
        _reports.push_back(Report(170, "Sum", time(0)));
        _reports.push_back(Report(180, "Average", time(0)));
        _reports.push_back(Report(180, "Average", time(0)));
        _reports.push_back(Report(180, "Average", time(0)));
    }
    list<Report> GetByName(string name)
    {
        //Gets list of reports by name
    }
private:
    list<Report> _reports;
};
```


Принцип открытости/закрытости

Open/closed Principle(OCP)

Есть какой-то набор отчетов, которые класс ReportProcessor может предоставить.

В данном случае программа выведет на экран отчеты с названием Sum.

Как нужно будет поступить, если потребуется выбирать отчеты еще и по итоговому полю?

Наверняка первая мысль, это добавить такой следующий метод в класс ReportProcessor

```
list<Report> GetByTotal(unsigned total)
{
    //Gets list of reports by "Total"
}
```

Принцип открытости/закрытости

Open/closed Principle(OCP)

Так будет происходить при каждом добавлении новых требований?

Если требований по фильтрации будет очень много, то нам придется добавить очень много новых методов в этот класс.

Кажется, что все-таки ничего плохого в этом все равно нет.

Но добавление новых методов в класс это всегда потенциальная опасность.

Ведь если класс уже реализует какую-то функциональность, то его исправление может нанести вред уже достигнутому.

Принцип открытости/закрытости

Open/closed Principle(OCP)

```
class ISpecification
{ public:
    bool IsSatisfied(Report r);
};

class NameSpecification : public ISpecification
{
    private:
        string _name;
    public:
        NameSpecification(string name)
        {
            _name = name;
        }
        bool IsSatisfied(Report r)
        {
            return r.Name == _name;
        }
};
```

Принцип открытости/закрытости

Open/closed Principle(OCP)

//Псевдокод

```
class ReportProcessor
```

```
{
```

```
    /// ...
```

```
public:
```

```
list<Report> GetReports(ISpecification *spec)
```

```
{
```

```
list<Report> *rs = new list<Report>();
```

```
foreach(Report r : _reports)
```

```
if (spec->IsSatisfied(r))
```

```
rs.push_back(r);
```

```
return rs;
```

```
}
```

```
}
```

Принцип подстановки Лисков

Liskov Substitution Principle

Правило.

Формулировка №1: если для каждого объекта o_1 типа S существует объект o_2 типа T , который для всех программ P определен в терминах T , то поведение P не изменится, если o_2 заменить на o_1 при условии, что S является подтипом T .

Формулировка №2: Функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты производных классов, **не зная об этом**.

Другими словами: если нужно добавить какое-то **ограничение** в переопределенный метод, и этого ограничения не существует в базовой реализации, то, нарушается принцип подстановки Liskov.

Принцип подстановки Лисков

Пример №1: Реализуем свой список с интерфейсом **IList**. Его особенностью будет то, что все записи в нем дублируются.

```
template<class T> class IList
{
public:
    void Add(T & ) = 0;
};

template<class T> class List: public IList<T>
{private:
    IList<T> *innerList;
public:
    List(){
        innerList = new IList<T>();
    }
    void Add(T& item){
        innerList.Add(item);
    }
};
```


Принцип подстановки Лисков

Пример №1: Реализуем свой список с интерфейсом `IList`. Его особенностью будет то, что все записи в нем дублируются.

```
template<class T>
class DoubleList: public IList<T>
{
private:
    IList<T> *innerList;
public:
    DoubleList(){
        innerList = new IList<T>();
    }
    void Add(T& item)
    {
        //2 times
        innerList.Add(item);
        innerList.Add(item);
    }
};
```

Принцип подстановки Лисков

```
void SomeMethod(IList<string> *list)
{
    List<string> *urls = urlService.GetUrls();

    foreach(string url : urls)
    {
        if (SomeBoolLogic(url))
            list->Add(url);
    }

    if (urls.Count > list.Count)
        throw Exception();
}
```

- Результаты сравнения свойств **Count** будут отличаться в зависимости от того, какой из наследников интерфейса **IList** будет передан в метод **SomeMethod**.
 - Результат этой функции будет зависеть от конкретной реализации **IList**.
-

Принцип подстановки Лисков

```
void SomeMethod(IList<string> list)
{
    List<string> urls = urlService.GetUrls();
    foreach(string url in urls)    {
        if (SomeBoolLogic(url))
            list.Add(url);
    }
    // start HardCode

    int realCount;
    if (list is DoubleList)
        realCount = list.Count / 2;
    else
        realCount = list.Count;
    // end HardCode

    if (urls.Count > realCount)
        throw Exception();
}
```

Принцип подстановки Лисков

Решение

Правильным решением будет использовать свой собственный интерфейс, например, **IDoubleList**.

Этот интерфейс будет объявлять для пользователей поведение, при котором добавляемые элементы удваиваются.

Принцип подстановки Лисков

Проектирование по контракту

Есть формальный способ понять, что наследование является **ошибочным**. Это можно сделать с помощью *проектирование по контракту*.

Бертран Мейер, его автор, сформулировал следующий принцип:

Примерно звучит так:

Наследуемый объект может заменить родительское пред-условие на такое же или более слабое и родительское пост-условие на такое же или более сильное.

Принцип подстановки Лисков

Сильные и слабые условия

Понятие “сильнее” и “слабее” пришли из логики.

Говорят, что условие $P1$ сильнее, чем $P2$, а $P2$ слабее, чем $P1$, если выполнение условия $P1$ влечет за собой выполнение условия $P2$, но они не эквивалентны.

Например

Условие $x > 5$ ($P1$), сильнее условия $x > 0$ ($P2$), поскольку при выполнении условия $P1$ выполняется и условие $P2$

(если x больше 5, то, естественно, что x больше 0), при этом эти условия не эквивалентны).

Принцип подстановки Лисков

Проектирование по контракту

Рассмотрим пред- и пост-условия для интерфейса `IList`. Для функции `Add`:

пред-условие: `item != null`

пост-условие: `count = oldCount + 1`

Для нашего `DoubleList` и его функции `Add`:

пред-условие: `item != null`

пост-условие: `count = oldCount + 2`

Здесь видно, что по контракту пост-условие базового класса не выполняется.

Другими словами, когда мы используем интерфейс `IList`, то как пользователи этого базового класса знаем только его пред- и пост-условия.

Нарушая принцип проектирования по контракту мы меняем поведение унаследованного объекта.

Принцип подстановки Лисков

```
/// Пред-условие : input > 1 && input < 20
/// Пост-условие result = input - 10 stronger
/// Пост-условие : ShowResultOnDisplay = true
class BaseCalculator
{
protected:
    int result;
    bool ShowResultOnDisplay;
    bool ShowResultOnPrintOut;

    virtual void DoSomeCalculation(int input)
    { if (input > 0 && input < 20){
      result = input - 10;
      ShowResultOnDisplay = true;
    }
    else{
      throw (new ArgumentException("Preconditions are not met"));
    }
    }
};
```


Принцип подстановки Лисков

```
/// <summary>
/// Weaker preconditions
/// Stronger postconditions
/// Precondition input > 0
/// Postcondition result = input - 10
/// Postcondition : ShowResultOnDisplay = true
/// Postcondition : ShowResultOnPrintOut = true
```

```
class LCalculator : public BaseCalculator
{ public:
void DoSomeCalculation(int input) {
    if (input > 0) {
        result = input - 10;
        ShowResultOnDisplay = true;
        ShowResultOnPrintOut = true;
    } else {
        throw (new ArgumentException("Preconditions are not met"));
    }
}
};
```

Принцип подстановки Лисков

```
/// Stronger preconditions  
/// Weaker postconditions  
/// Precondition input > 0 && input < 5  
/// Postcondition result = input - 10
```

```
class NLCalculator : public BaseCalculator  
{  
public:  
    void DoSomeCalculation(int input)  
    {  
        if (input > 1 && input < 5)  
        {  
            result = input - 10;  
        }  
        else  
        {  
            throw (new ArgumentException("Preconditions are not met"));  
        }  
    }  
};
```


Принцип разделения интерфейса

Interface Segregation Principle

Правило:

Клиенты не должны зависеть от методов, которые они не используют.

Стремитесь к тому, чтобы интерфейсы были достаточно **узкими**, чтобы классам не приходилось реализовывать избыточное поведение.

Принцип разделения интерфейсов говорит о том, что слишком **«толстые» интерфейсы** необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе.

В итоге при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

Принцип разделения интерфейса

Interface Segregation Principle

Наследование позволяет классу иметь только один суперкласс, но не ограничивает количество интерфейсов, которые он может реализовать.

Большинство объектных языков программирования позволяют классам реализовывать сразу несколько интерфейсов, поэтому нет нужды снабжать в ваш интерфейс большим количеством различных поведений, чем он того требует.

Всегда можно присвоить классу сразу несколько интерфейсов поменьше.

Принцип разделения интерфейса

Interface Segregation Principle

Пример:

Представьте библиотеку для работы с облачными провайдерами.

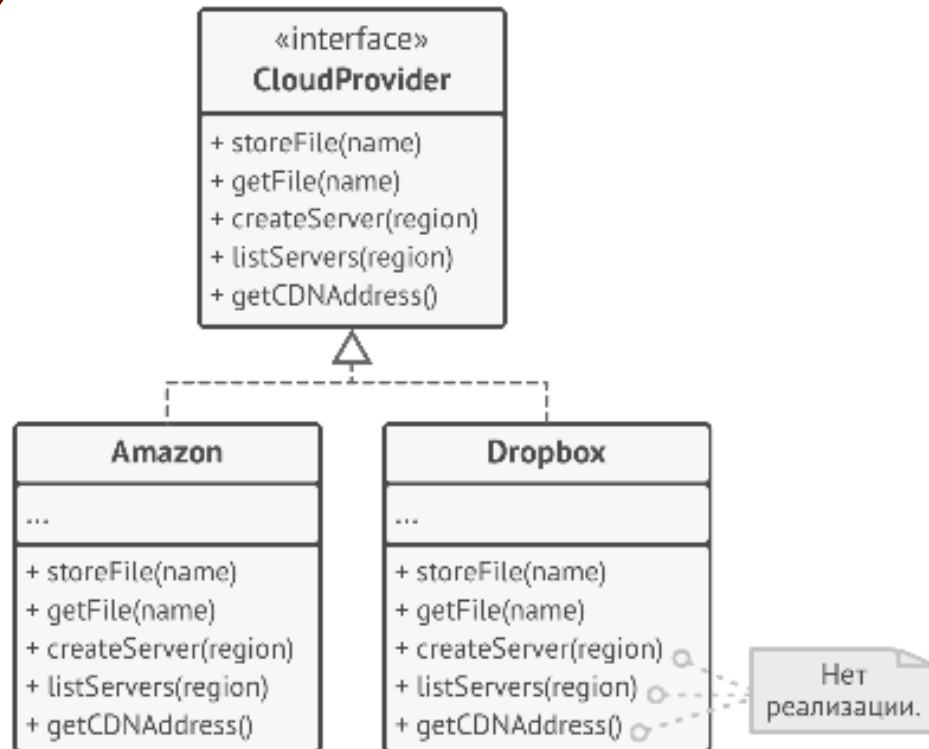
В первой версии она поддерживала только **Amazon**, имеющий полный набор облачных услуг. Исходя из них и проектировался интерфейс будущих классов.

Но позже стало ясно, что получившийся интерфейс облачного провайдера слишком широк, так как есть другие провайдеры, реализующие только часть из всех возможных сервисов.

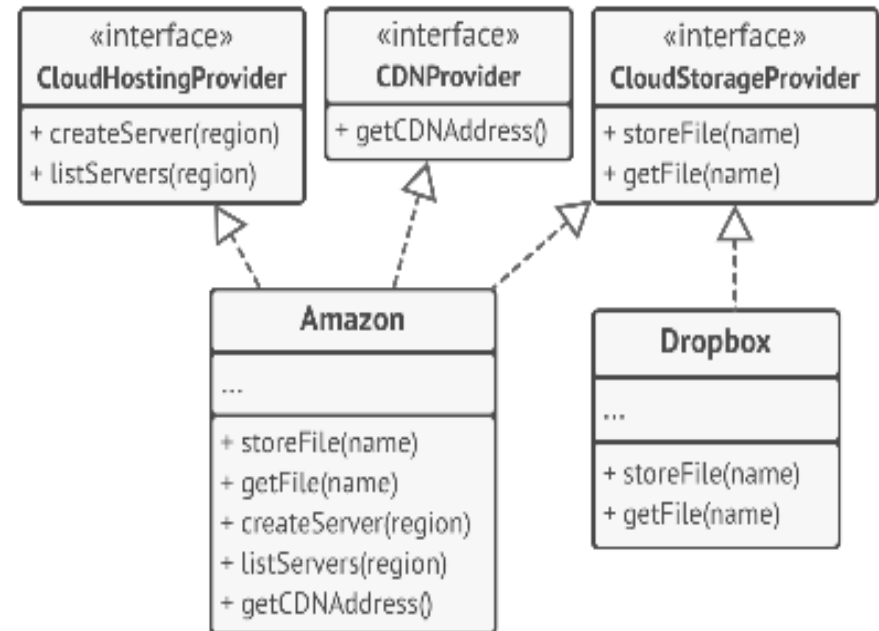
Чтобы не плодить классы с пустой реализацией, раздутый интерфейс можно разбить на части. Классы, которые были способны реализовать все операции старого интерфейса, могут реализовать сразу несколько новых частичных интерфейсов.

Принцип разделения интерфейса

Interface Segregation Principle



ДО: не все клиенты могут реализовать операции интерфейса.



ПОСЛЕ: раздутый интерфейс разбит на части.

Dependency Inversion Principle

Правило:

Классы **верхних уровней** не должны зависеть от **классов нижних уровней**. Оба должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Обычно при проектировании программ можно выделить два уровня классов.

Классы нижнего уровня реализуют базовые операции вроде работы с диском, передачи данных по сети, подключения к базе данных и прочее.

Классы высокого уровня содержат сложную бизнес-логику программы, которая опирается на классы низкого уровня для осуществления более простых операций.

Принцип Инверсий зависимостей

Dependency Inversion Principle

Рассмотрим случай когда, сначала, проектируются классы нижнего уровня, а только потом проектируются классы верхнего уровня.

При таком подходе классы бизнес-логики становятся зависимыми от более примитивных низкоуровневых классов.

Каждое изменение в низкоуровневом классе может затронуть классы бизнес-логики, которые его используют.

Принцип инверсии зависимостей предлагает изменить направление, в котором происходит проектирование.

Принцип Инверсий зависимостей

Dependency Inversion Principle

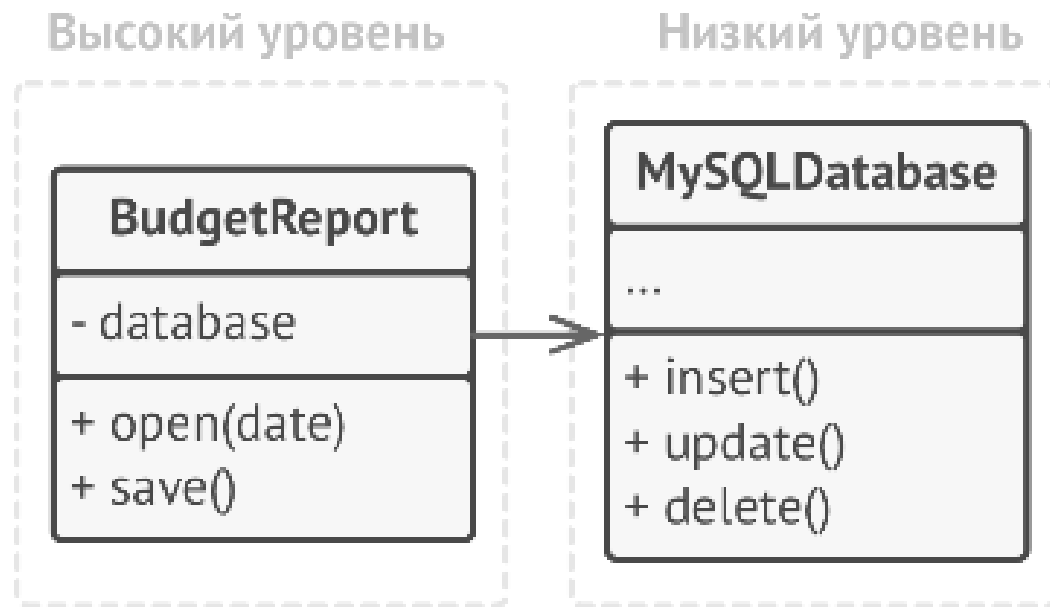
1. Для начала вам нужно описать интерфейс низкоуровневых операций, которые нужны классу бизнес-логики.
 2. Это позволит вам убрать зависимость класса бизнес-логики от конкретного низкоуровневого класса, заменив её «мягкой» зависимостью от интерфейса.
 3. Низкоуровневый класс, в свою очередь, станет зависимым от интерфейса, определённого бизнес-логикой.
-

Принцип Инверсий зависимостей

Dependency Inversion Principle

Пример:

Рассмотрим высокоуровневый класс формирования бюджетных отчётов который напрямую использует класс базы данных для загрузки и сохранения своей информации.

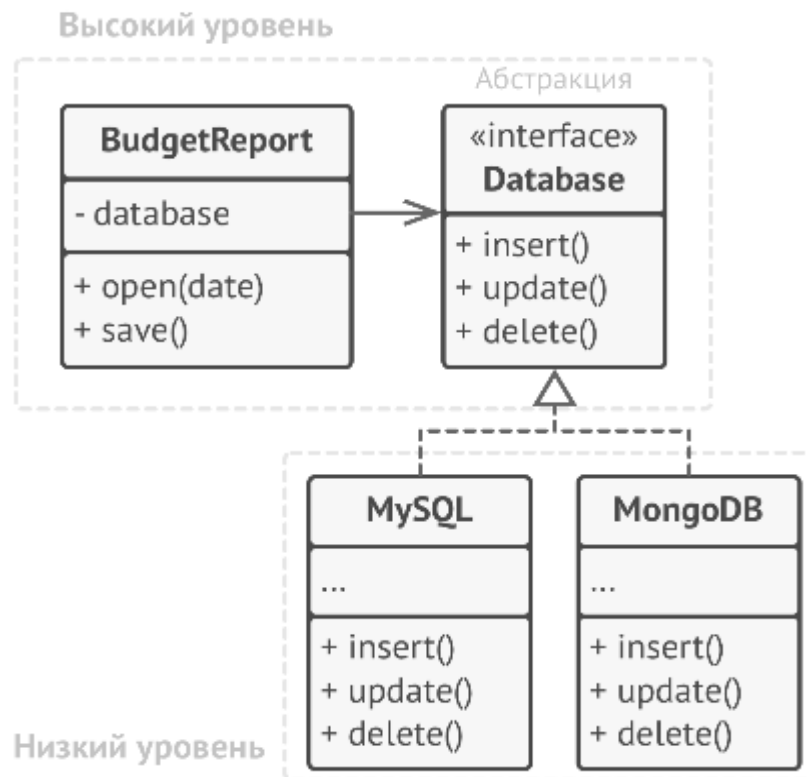


ДО: высокоуровневый класс зависит от низкоуровневого.

Принцип Инверсий зависимостей

Dependency Inversion Principle

Можно исправить проблему, создав высокоуровневый интерфейс для загрузки/сохранения данных и привязав к нему класс отчётов. Низкоуровневые классы должны реализовать этот интерфейс, чтобы их объекты можно было использовать внутри объекта отчётов.



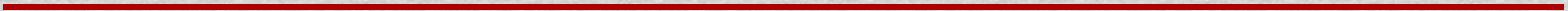
ПОСЛЕ: низкоуровневые классы зависят от высокоуровневой абстракции.

Принцип Инверсий зависимостей

Dependency Inversion Principle

Таким образом, меняется направление зависимости.

Если раньше высокий уровень зависел от низкого, то сейчас всё наоборот — низкоуровневые классы зависят от высокоуровневого интерфейса.



ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ



Паттерн проектирования — это часто встречающееся решение определённой **проблемы** при проектировании архитектуры программ.

Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды разрабатываемой программы.

(паттерн — инженерный чертёж, на котором нарисовано решение, но не конкретные шаги его реализации)

Составляющие паттерна

Паттерны, как правило описываются формально и чаще всего состоят из пунктов: Описания паттернов обычно очень формальны и чаще всего состоят из таких пунктов:

1. проблема, которую решает паттерн;
2. структуры классов, составляющих решение;
3. особенностей реализации в различных контекстах;
4. связей с другими паттернами.

Классификация паттернов

Паттерны отличаются по уровню сложности, детализации и охвата проектируемой системы.

Низкоуровневые и простые паттерны — идиомы. Они **не универсальны**, поскольку применимы только в рамках одного языка программирования.

***Идиома программирования** — устойчивый способ выражения некоторой составной конструкции в одном или нескольких языках программирования. Идиома является шаблоном решения задачи, записи алгоритма или структуры данных путём комбинирования встроенных элементов языка. Идиому можно считать самым низкоуровневым шаблоном проектирования.*

Пример простой идиомы:

Инкремент

```
i += 1; /* i = i + 1; */
```

```
++i; /* тот же результат */
```

```
i++; /* тот же результат */
```

Самые **универсальные** — архитектурные паттерны, которые можно реализовать практически на любом языке.

Существует три основные группы паттернов:

- **Порождающие паттерны** (*для создания объектов без внесения в программу лишних зависимостей*).
- **Структурные паттерны** (*показывают различные способы построения связей между объектами*).
- **Поведенческие паттерны** (*эффективная коммуникации между объектами*).

Зачем знать паттерны?

- 1. Проверенные решения.** Тратится меньше времени при использовании готовых решений.
- 2. Стандартизация кода.** Делается меньше просчётов при проектировании, используя типовые унифицированные решения, так как все скрытые проблемы в них уже давно найдены.
- 3. Общий программистский словарь.** В предлагаемом решении достаточно указать название паттерна, что бы всем остальным стало понятно о чем идет речь.

ПОРОЖДАЮЩИЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ



Порождающие паттерны - отвечают за *удобное* и *безопасное* создание новых объектов или даже целых семейств объектов.

1. **Одиночка**
2. **Фабричный метод**
3. **Абстрактная фабрика**
4. **Строитель**
5. **Прототип**

Паттерн Одиночка (Singleton)

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только **один экземпляр**, и предоставляет к нему **глобальную точку доступа**.

Применяется когда

1. В программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам, например, общий доступ к базе данных из разных частей программы.

Одиночка скрывает от клиентов все способы создания нового объекта, кроме **специального метода**. Этот метод либо **создаёт объект**, либо **отдаёт существующий объект**, если он уже был создан.

2. Необходимо иметь больше контроля над глобальными переменными.

В отличие от глобальных переменных, паттерн Одиночка гарантирует, что никакой другой код не заменит созданный экземпляр класса, поэтому вы всегда уверены в наличии лишь одного объекта-одиночки.

Паттерн Одиночка (Singleton)

Шаги реализации

1. Добавьте в класс приватное статическое поле, которое будет содержать одиночный объект.
2. Объявите статический создающий метод, который будет использоваться для получения одиночки.
3. Добавьте «ленивую инициализацию» (создание объекта при первом вызове метода) в создающий метод одиночки.
4. Сделайте конструктор класса приватным.

Паттерн Одиночка (Singleton) - Пример

```
class Singleton
{
public:
static Singleton& Instance()
{
// согласно стандарту, этот код ленивый и потокобезопасный
static Singleton s;
return s;
}
private:
Singleton() { } // конструктор недоступен
~Singleton() {

} // и деструктор
// необходимо также запретить копирование
Singleton(Singleton const&); // реализация не нужна
Singleton& operator= (Singleton const&); // и тут
};

int main(int argc, char** argv) {
//new Singleton(); // Won't work
Singleton& instance = Singleton::Instance();
return 0;
}
```


Паттерн Одиночка (Singleton)

```
class Singleton
{public:
    static Singleton& Instance()
    { static Singleton s;
      return s;
    }
    void SomeInformation()
    {
        for (int i = 0; i < 10; i++) std::cout << mas[i] << ", ";
        std::cout <<std::endl;
    }
private:
    Singleton() {
        for (int i = 0; i < 10; i++){mas[i] = 111; }
    };
    ~Singleton() { std::cout << "Private Destructor is called"<<std::endl;
    };
    Singleton(Singleton const&); // реализация не нужна
    Singleton& operator= (Singleton const&);
protected:
    int mas[10];};
```

Паттерн Одиночка (Singleton)

Microsoft Visual Studio Debug Console

111, 111, 111, 111, 111, 111, 111, 111, 111, 111,
Private Destructor is called

```
int main()
{
    //new Singleton(); // Won't work
    Singleton& instance = Singleton::Instance();
    Singleton::Instance().SomeInformation();
}
```


Фабричный метод

Фабричный метод (также известен как Виртуальный конструктор, Factory Method) – это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

- Для того, чтобы система оставалась независимой от различных типов объектов, паттерн Factory Method использует *механизм полиморфизма* т.е. классы всех конечных типов наследуют от одного абстрактного базового класса, предназначенного для полиморфного использования.
- В этом базовом классе определяется *единый интерфейс*, через который пользователь будет оперировать объектами конечных типов.
- Для обеспечения относительно простого добавления в систему новых типов паттерн Factory Method локализует создание объектов конкретных типов в специальном классе-фабрике.
- Методы этого класса, посредством которых создаются объекты конкретных классов, называются фабричными.

Существуют две разновидности паттерна Factory Method:

- **Обобщенный конструктор;**
- **Классическая реализация;**

Фабричный метод (Factory Method) - это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах.

То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

Паттерн применяется когда

1. Когда заранее неизвестно, объекты каких типов необходимо создавать;
2. Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать;
3. Когда создание новых объектов необходимо делегировать из базового класса классам наследникам;

Существуют две разновидности паттерна Factory Method:

- **Обобщенный конструктор;**
- **Классическая реализация;**

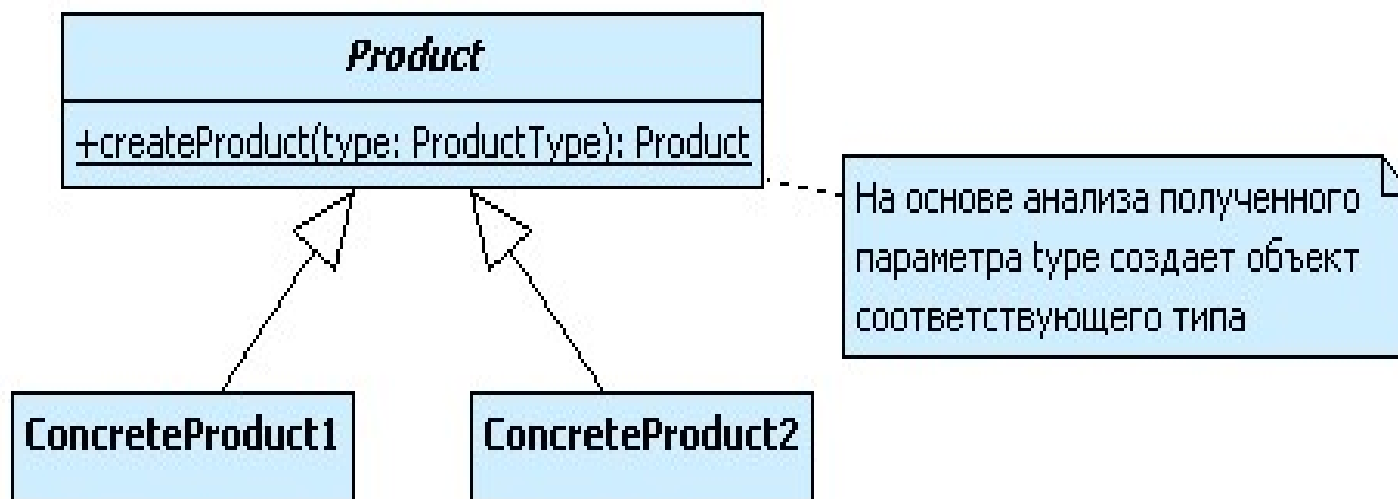
Фабричный метод (Factory Method) - это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах.

То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

Паттерн применяется когда

1. Когда заранее неизвестно, объекты каких типов необходимо создавать;
2. Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать;
3. Когда создание новых объектов необходимо делегировать из базового класса классам наследникам;

Реализация паттерна Factory Method: *обобщенный конструктор*



Реализация паттерна Factory Method: *обобщенный конструктор*

```
enum TypesObjects {ObjectX_ID = 0, ObjectY_ID, ObjectZ_ID };

class MainObject
{
    public:
    virtual void info() = 0;
    virtual ~MainObject() {}
    // Параметризированный статический фабричный метод
    static MainObject* createObject( TypesObjects id );
};
```

Реализация паттерна Factory Method: *обобщенный конструктор*

```
class ObjectX:public MainObject
{
    public:
    void info(){
        cout << "This is ObjectX" << endl;
    }
};

class ObjectY:public MainObject
{
    public:
    void info(){
        cout << "This is ObjectY" << endl;
    }
};

class ObjectZ: public MainObject .....
```

Реализация паттерна Factory Method: *обобщенный конструктор*

```
MainObject* MainObject::createObject(TypesObjects id)
{
    MainObject * p = nullptr;
    switch (id)
    {
        case ObjectX_ID:
            p = new ObjectX();
            break;
        case ObjectY_ID:
            p = new ObjectY();
            break;
        case ObjectZ_ID:
            p = new ObjectZ();
            break;
    }
    return p;
}
```


Реализация паттерна Factory Method: *обобщенный конструктор*

```
int main(int argc, char** argv) {  
  
    vector<MainObject*> vectorObj;  
    vectorObj.push_back(MainObject::createObject(ObjectX_ID));  
    vectorObj.push_back(MainObject::createObject(ObjectY_ID));  
    vectorObj.push_back(MainObject::createObject(ObjectZ_ID));  
  
    for (int i = 0; i < vectorObj.size(); i++)  
        vectorObj[i]->info();  
    return 0;  
}
```

Представленный вариант паттерна Factory Method пользуется популярностью благодаря своей простоте.

В нем статический фабричный метод `createObject()` определен непосредственно в полиморфном базовом классе `MainObject`.

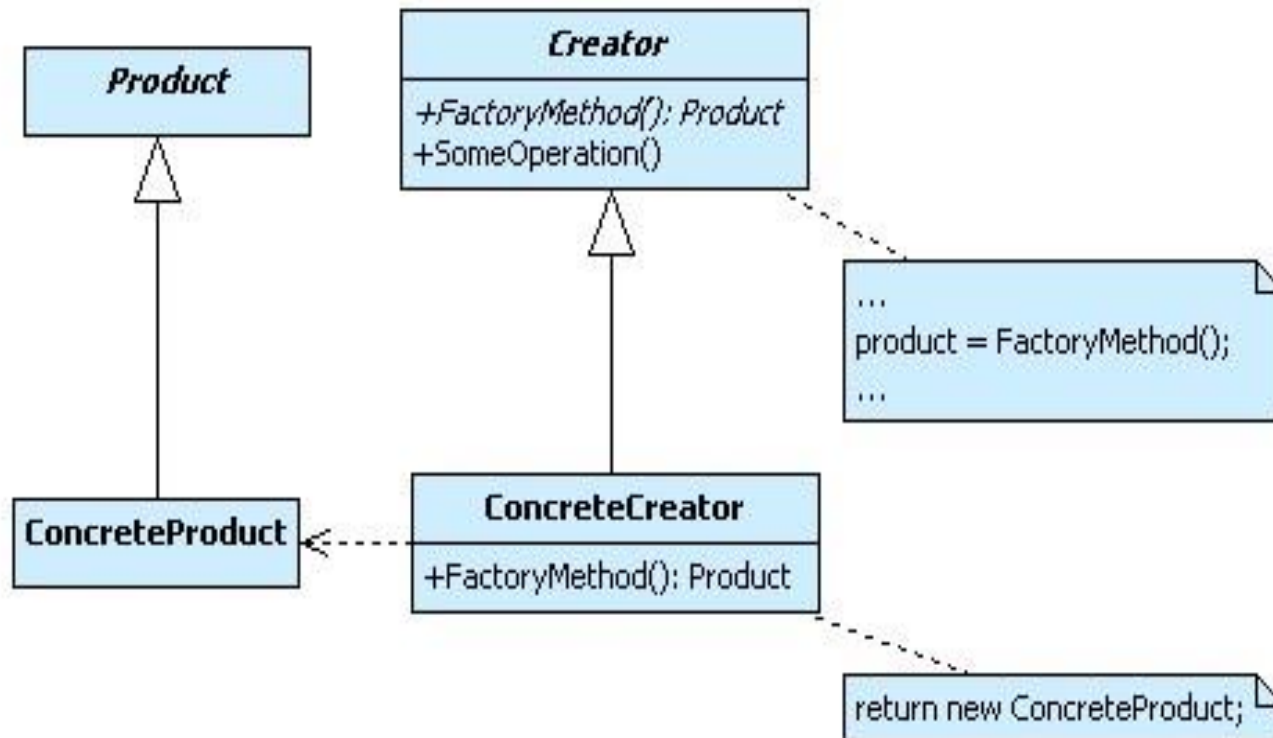
Этот фабричный метод является параметризованным, то есть для создания объекта некоторого типа в `createObject()` передается соответствующий идентификатор типа.

С точки зрения "чистоты" объектно-ориентированного кода у этого варианта есть следующие недостатки:

- Так как код по созданию объектов всех возможных типов сосредоточен в статическом фабричном методе класса `createObject()`, то базовый класс `MainObject` обладает знанием обо всех производных от него классах, что является нетипичным для объектно-ориентированного подхода.
- Подобное использование оператора `switch` (как в коде фабричного метода `createObject()`) в объектно-ориентированном программировании также не приветствуется.

Указанные недостатки отсутствуют в классической реализации паттерна Factory Method.

Классическая Реализация паттерна Factory Method




```
#include<assert.h>
#include<iostream>
#include<vector>

class MainObject
{
    public:
    virtual void info() = 0;
    virtual ~MainObject() {}
};

class ObjectX:public MainObject
{
    public:
    void info(){
        cout << "This is ObjectX" << endl;
    }
};

class ObjectY:public MainObject {};
class ObjectZ:public MainObject {};
```

```
#include<assert.h>
#include<iostream>
#include<vector>

class FactoryObj
{
public:
    virtual MainObject* createObject() = 0;
    virtual ~FactoryObj(){}
};

class FactoryObjX: public FactoryObj
{
public:
    MainObject*createObject(){

        return new ObjectX();
    }

}

class FactoryObjY: public FactoryObj{};
class FactoryObjZ: public FactoryObj{};
```

```
#include<assert.h>
#include<iostream>
#include<vector>

int main(int argc, char** argv) {

    FactoryObjX ObjXCreator;
    FactoryObjY ObjYCreator;
    FactoryObjZ ObjZCreator;

    vector<MainObject*> vectorObj;

    vectorObj.push_back( ObjXCreator.createObject());
    vectorObj.push_back( ObjYCreator.createObject());
    vectorObj.push_back( ObjZCreator.createObject());

    for(int i=0; i<vectorObj.size(); i++){
        vectorObj[i]->info();
        //...Do Something
        delete vectorObj[i];
    }
    return 0;
}
```


Классический вариант паттерна Factory Method использует идею полиморфной фабрики.

Специально выделенный для создания объектов полиморфный базовый класс *FactoryObj* объявляет интерфейс фабричного метода **createObject()**, а производные классы его реализуют.

Представленный вариант паттерна Factory Method является наиболее распространенным, но не единственным. Возможны следующие вариации:

Класс *FactoryObj* имеет реализацию фабричного метода **createObj()** по умолчанию.

Фабричный метод **createObj()** класса *FactoryObj* параметризован типом создаваемого объекта (как и у представленного ранее, простого варианта Factory Method) и имеет реализацию по умолчанию.

В этом случае, производные от *FactoryObj* классы необходимы лишь для того, чтобы определить нестандартное поведение **createObj()**.

Достоинства паттерна Factory Method

- Создает объекты разных типов, позволяя системе оставаться независимой как от самого процесса создания, так и от типов создаваемых объектов.

Недостатки паттерна Factory Method

- В случае классического варианта паттерна даже для порождения единственного объекта необходимо создавать соответствующую фабрику

Паттерн Abstract Factory(абстрактная фабрика)

Паттерн используется тогда, когда

1. Система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов.
2. Необходимо создавать **группы** или **семейства взаимосвязанных объектов**, исключая возможность одновременного использования объектов из разных семейств в одном контексте.

Паттерн Abstract Factory реализуется на основе **фабричных методов**. Любое семейство или группа взаимосвязанных объектов характеризуется несколькими общими типами создаваемых продуктов, при этом сами продукты таких типов будут различными для разных семейств.

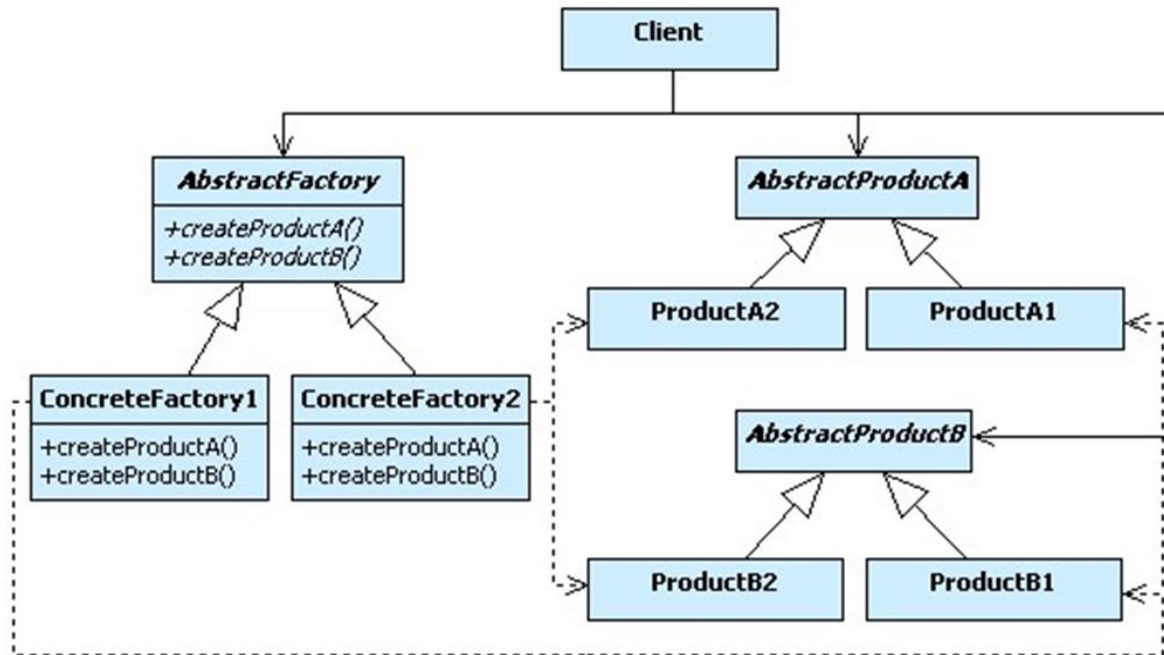
Для того чтобы **система** оставалась независимой от специфики того или иного семейства продуктов необходимо использовать **общие интерфейсы для всех основных типов продуктов**.

Для решения задачи по созданию **семейств взаимосвязанных объектов** паттерн **Abstract Factory** вводит понятие **абстрактной фабрики**.

Паттерн Abstract Factory(абстрактная фабрика)

Абстрактная фабрика представляет собой некоторый полиморфный базовый класс(***AbstractFactory***), назначением которого является *объявление интерфейсов фабричных методов*(***CreateProductA***, ***CreateProductB***), служащих для создания продуктов всех основных типов (один фабричный метод на каждый тип продукта).

Производные от него классы(***ConcreteFactory1***, ***ConcreteFactory2***), реализующие эти интерфейсы, предназначены для создания продуктов **всех типов** внутри *семейства* или *группы*. (Т.е. по схеме, ***ConcreteFactory1*** создает объекты для всех типов порождённых от(***AbstractProductA***, ***AbstractProductB***) и объединённых в одно семейство (***ProductA1***, ***ProductA2***), вторая фабрика ***ConcreteFactory2***, также, создаёт все типы объектов, но для второго семейства (***ProductB1***, ***ProductB2***).



Паттерн Abstract Factory(абстрактная фабрика)

Например, формально можно описать так.

Рассмотрим множество групп

Groups = {GroupA, GroupB, GroupC }

Рассмотрим множество объектов

SetObj = {ObjectX, ObjectY, ObjectZ }

Каждая группа состоит из объектов, для которых определены свойства в SetObj, но в каждой группе эти свойства определяются по своему.

GroupA = SetObj^A = {ObjectX^A, ObjectY^A, ObjectZ^A}

GroupB = SetObj^B = {ObjectX^B, ObjectY^B, ObjectZ^B}

GroupC = SetObj^C = {ObjectX^C, ObjectY^C, ObjectZ^C}

Представим реализацию абстрактной фабрики на C++

Сначала выполняется описание абстрактных классов для объектов из **SetObj**.

```
#include <iostream>
#include <vector>
class ObjectX
{
    public:
        virtual void info() = 0;
        virtual ~ObjectX() {}
};
class ObjectY
{
    public:
        virtual void info() = 0;
        virtual ~ObjectY() {}
};
class ObjectZ
{
    public:
        virtual void info() = 0;
        virtual ~ObjectZ() {}
};
```


Создание конкретных объектов для заданных групп **Groups**

```
class GroupAObjX: public ObjectX
{
public:
    void info(){
        cout << "GroupAObjX" << endl;
    }
};

class GroupAObjY: public ObjectY
{
public:
    void info(){
        cout << "GroupAObjY" << endl;
    }
};

class GroupAObjZ: public ObjectZ
{
public:
    void info(){
        cout << "GroupAObjZ" << endl;
    }
};
```

Создание конкретных объектов для заданных групп **Groups**

```
class GroupBObjX: public ObjectX
{
    public:
    void info(){
        cout << "GroupBObjX" << endl;
    }
};
class GroupBObjY: public ObjectY
{
    public:
    void info(){
        cout << "GroupBObjY" << endl;
    }

};

class GroupBObjZ: public ObjectZ
{
    public:
    void info(){
        cout << "GroupBObjZ" << endl;
    }

};
```

Создание конкретных объектов для заданных групп **Groups**

Тоже самое для класса **GroupBObjX**

```
class GroupCObjX: public ObjectX
{
    public:
    void info(){
        cout << "GroupCObjX" << endl;
    }
};

class GroupCObjY: public ObjectY
{
    public:
    void info(){
        cout << "GroupCObjY" << endl;
    }
};

class GroupBObjZ: public ObjectZ
{
    public:
    void info(){
        cout << "GroupBObjZ" << endl;
    }
};
```


Создание абстрактной фабрики, конкретной фабрики для каждой группы

```
class GroupFactory
{
public:
    virtual ObjectX * createObjectX() = 0;
    virtual ObjectY * createObjectY() = 0;
    virtual ObjectZ * createObjectZ() = 0;
    virtual ~GroupFactory(){};
};
```

```
class GroupAFactory: public GroupFactory
{
public:
    ObjectX * createObjectX(){
        return new GroupAObjX();
    }
    ObjectY * createObjectY(){
        return new GroupAObjY();
    }
    ObjectZ * createObjectZ(){
        return new GroupAObjZ();
    }
};
```

```
class GroupBFactory:public GroupFactory
{
public:
    ObjectX * createObjectX(){
        return new GroupBObjX();
    }
    ObjectY * createObjectY(){
        return new GroupBObjY();
    }
    ObjectZ * createObjectZ(){
        return new GroupBObjZ();
    }
};
```

//Класс группа содержащий ту или иную группу

```
class Group
{
    public:
    ~Group() {
        int i;
        for(i=0; i< x.size(); ++i) delete x[i];
        for(i=0; i< y.size(); ++i) delete y[i];
        for(i=0; i< z.size(); ++i) delete z[i];
    }
    void info() {
        int i;
        for(i=0; i<x.size(); ++i) x[i]->info();
        for(i=0; i<y.size(); ++i) y[i]->info();
        for(i=0; i<z.size(); ++i) z[i]->info();
    }
    vector<ObjectX*> x;
    vector<ObjectY*> y;
    vector<ObjectZ*> z;
};
```



```
class ConcreteGroup
{
public:
    Group* createGroup(GroupFactory &factory ){
        Group* p = new Group;
        p->x.push_back( factory.createObjectX());
        p->y.push_back( factory.createObjectY());
        p->z.push_back( factory.createObjectZ());
        return p;
    }
};
```

```
int main(int argc, char** argv) {  
  
    ConcreteGroup cgroup;  
  
    GroupAFactory ga_factory;  
    GroupBFactory gb_factory;  
  
    Group * ga = cgroup.createGroup( ga_factory);  
    Group * gb = cgroup.createGroup( gb_factory);  
  
    cout << "GroupA:" << endl;  
    ga->info();  
    cout << "\nGroupB" << endl;  
    gb->info();  
  
    return 0;  
}
```

Теперь рассмотрим создание абстрактной фабрики на конкретном примере:

Например, необходимо организовать сборку компьютеров различных конфигураций.

При сборке, уделим внимание трем компонентам, процессору, жесткому диску и монитору.

Опишем соответствующие интерфейсы, например, следующим образом. В каждом интерфейсе определим абстрактные методы.

```
class IProcessor
{
public:
virtual void PerformOperation() = 0;
};
class IHardDisk
{
public:
virtual void StoreData() = 0;
};
class IMonitor
{
public:
virtual void DisplayPicture() = 0;
};
```


Далее, по схеме, определим ConcreteProduct, в соответствии с различными, возможными семействами или группами, которые можно выделить для этих объектов, например
Процессор, жесткий диск могут быть дорогими по цене, либо нет, таким образом выделим два семейства объектов

1)дорогие по цене;

2)дешевые по цене;

Далее определим соответствующие классы, для процессора:

```
class IHardDisk
{
public:
virtual void StoreData() = 0;
};

class IMonitor
{
public:
virtual void DisplayPicture() = 0;
};

class ExpensiveProcessor : public IProcessor
{
public:
void PerformOperation()
{ cout << "Operation will perform quickly" << endl; }
};

class CheapProcessor : public IProcessor
{
public:
void PerformOperation() { cout << "Operation will perform Slowly" << endl; }
};
```

Для жесткого диска.

```
class IHardDisk
{
public:
virtual void StoreData() = 0;
};
class IMonitor
{
public:    virtual void DisplayPicture() = 0;
};
```

Для монитора, рассмотрим относительно разрешения.

```
class HighResolutionMonitor : public IMonitor
{
public:
    void DisplayPicture()
    {
        cout<<"Picture quality is Best"<<endl;
    }
};

class LowResolutionMonitor : public IMonitor
{
public:
    void DisplayPicture()
    {
        cout<< "Picture quality is Average"<< endl;
    }
};
```


Создадим абстрактный класс фабрики

```
class IMachineFactory
{
public:
virtual IProcessor* GetRam() = 0;
virtual IHardDisk * GetHardDisk() = 0;
virtual IMonitor  * GetMonitor() = 0;
};
```

Здесь, создаем конкретные фабрики для каждого семейства

```
class HighBudgetMachine :public IMachineFactory
{
public:
IProcessor* GetRam() { return new ExpensiveProcessor();}
IHardDisk*   GetHardDisk() { return new ExpensiveHDD(); }
IMonitor*    GetMonitor() { return new HighResolutionMonitor(); } };
```

```
class LowBudgetMachine : public IMachineFactory
{
public:
IProcessor* GetRam() { return new CheapProcessor(); }
IHardDisk*  GetHardDisk() { return new CheapHDD(); }
IMonitor*   GetMonitor() { return new LowResolutionMonitor();}
};
```

Здесь выполняем создание сборки относительно заданной категории

```
class ComputerShop
{
    IMachineFactory *category;
public:
    ComputerShop(IMachineFactory *_category) { category =
        _category; }
    void AssembleMachine() {
        IProcessor* processor = category->GetRam();
        IHardDisk* hdd = category->GetHardDisk();
        IMonitor* monitor = category->GetMonitor();
        //используем все три категории для создания машины
        processor->PerformOperation();
        hdd->StoreData();
        monitor->DisplayPicture();
    }
};
```

Клиентский код выглядит следующим образом

```
MachineFactory *factory = new HighBudgetMachine();// или new  
LowBudgetMachine();  
ComputerShop *shop = new ComputerShop(factory);  
shop->AssembleMachine();
```


Достоинства паттерна Abstract Factory

- Скрывает сам процесс порождения объектов, а также делает систему независимой от типов создаваемых объектов, специфичных для различных семейств или групп (пользователи оперируют этими объектами через соответствующие абстрактные интерфейсы).
- Позволяет быстро настраивать систему на нужное семейство создаваемых объектов.

Недостатки паттерна Abstract Factory

- Трудно добавлять новые типы создаваемых продуктов или заменять существующие, так как интерфейс базового класса абстрактной фабрики фиксирован.

Легенда

Реализован некоторая система, которая способна генерировать код на языке **C++**, причем, программы только определенного вида.

■ Задание

Требуется реализовать подобную генерацию программ на **C#** и **Java**. Таким образом, необходимо расширить возможности предложенной реализации. Предлагается рассмотреть и реализовать **фабричные подходы** для расширения возможностей текущей реализации.

ЗАДАНИЕ

Лабораторная
№2