Тема:Паттерн Абстрактная фабрика

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.

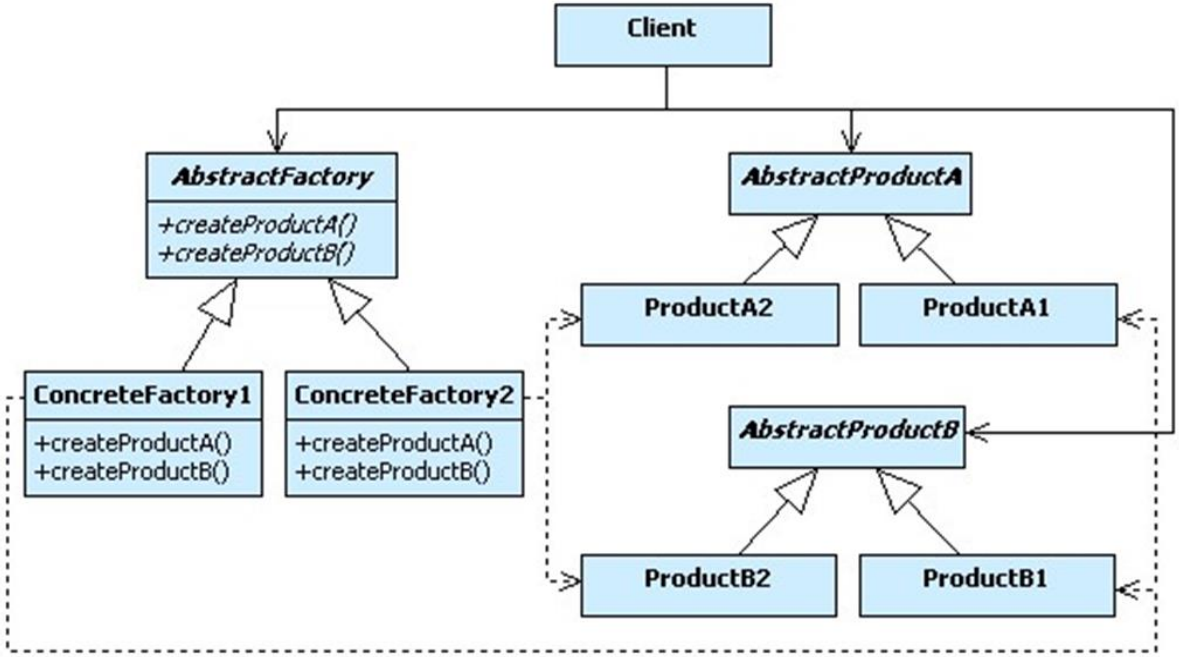
Для того чтобы ***система*** оставалась независимой от специфики того или иного семейства продуктов необходимо использовать **общие интерфейсы для всех основных типов продуктов.**

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

Паттерн Abstract Factory(абстрактная фабрика)

Абстрактная фабрикапредставляет 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, ProductB1**), вторая фабрика **ConcreteFactory2**, также, создаёт все типы объектов, но для второго семейства **(ProductA2, ProductB2).**



Паттерн Abstract Factory(абстрактная фабрика)

Например, формально можно описать так.

Рассмотрим множество групп

Groups = {GroupA, GroupB, GroupC }

Рассмотрим множестово 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 = $SetObj^B$ = {$ObjectX^B$, $ObjectX^B$, $ObjectZ^B$}

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

Представим реализацию абстрактной фабрики на C++

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

```cpp
#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 следующие 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 создании 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:

Реализована 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