参考：https://design-patterns.readthedocs.io/zh_CN/latest/

概念：UML类图：后面假如有机会看的话再补充

父类和子类的关系：子类是父类更具体地某一类, 是is a 关系。

类之间地关系：

- 泛化关系：继承非抽象类，继承关系为is a关系
- 实现关系：继承抽象类
- 聚合关系：B由A组成，和组合关系的差别，B没有了，A依旧存在
- 组合关系：B由A组成，B没有了，A也不存在了
- 关联关系：不同类的对象之间的结构关系，比如乘车人和车票之间就是一种关联关系。在最终代码中，关联对象通常是以成员变量的形式实现。
- 依赖关系：描述一个对象在运行期间会用到另一个对象的关系，是一种临时性关系。在最终代码中，体现为类构造方法及类方法的传入参数。

创建型模式

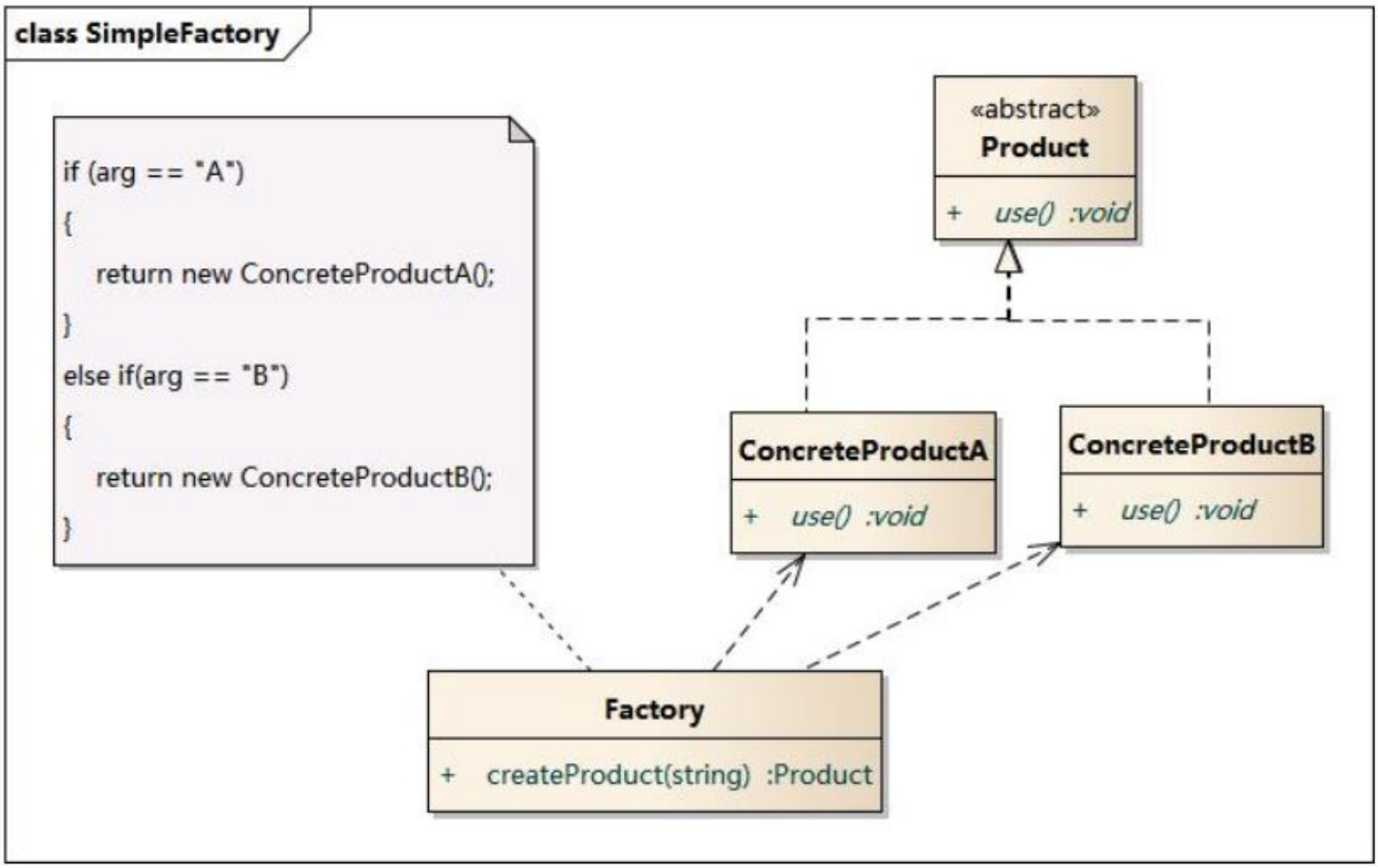
创建型模式对类的实例化过程进行了抽象，能够将软件模块中对象的创建和对象的使用分离。外界对于这些对象只需要直到它们共同的接口，而不清楚其具体的实现细节。

包含的模式：

- 简单工厂模式
- 工厂方法模式
- 抽象工厂模式
- 建造者模式
- 原型模式
- 单例模式

1 简单工厂模式

1.1 模式定义
可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常具有共同的父类。



角色：

- 工厂角色：负责实现创建所有实例的内部逻辑
- 抽象产品角色：时所创建的所有对象的父类，负责描述所有实例所共有的公共接口
- 具体产品角色：创建目标，所有创建的对象都充当这个角色的某个具体类的实例。

要点：当你需要什么，只需要传入一个正确的参数，就可以获取你所需要的对象，而无须直到其创建细节。

2.1 简单工厂类的优点
最大的优点：实现对象的创建和对象的使用分离，将对象的创建交给专门的工厂类负责。

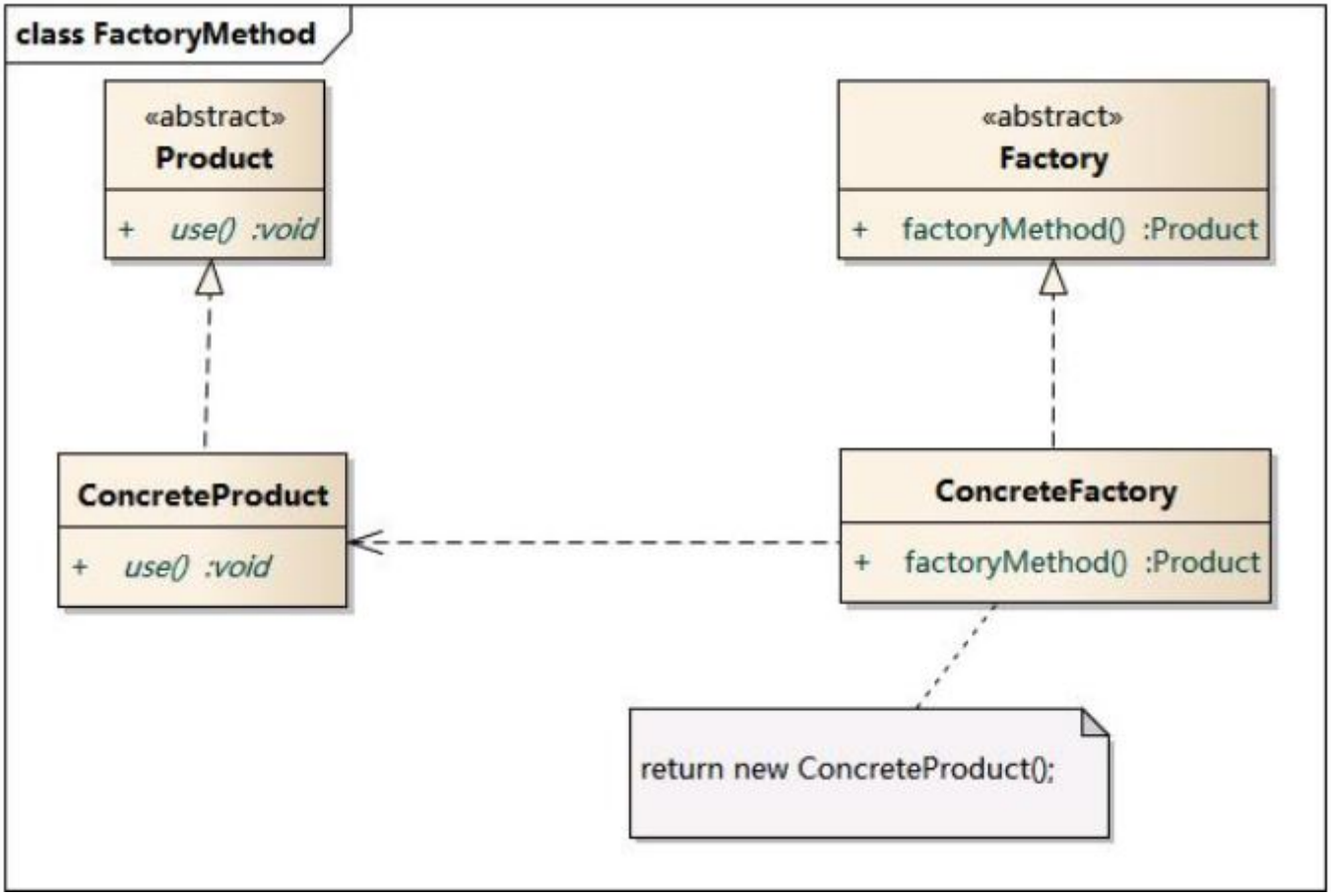
2.2 缺点
工厂不够灵活，增加新的具体产品需要修改工厂类的判断逻辑代码，而且产品比较多时，工厂方法代码将会非常复杂。

2 工厂方法模式

工厂方法模式又称为工厂模式，也叫虚拟构造器模式或者多态工厂模式。属于类创建型模式。在工厂模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象。目的是将产品类的实例化操作延迟到工厂子类中完成。通过工厂子类来确定究竟应该实例哪一个具体产品类

2.3 模式结构
角色:

- 抽象产品
- 具体产品
- 抽象工厂
- 具体工厂




```
#include "Factory.h"
#include "ConcreteFactory.h"
#include "Product.h"
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
        Factory * fc = new ConcreteFactory();
        Product * prod = fc->factoryMethod();
        prod->use();

        delete fc;
        delete prod;

        return 0;
}
```

2.4 模式分析
使用了面向对象的多态性。在工厂模式中，核心工厂不在负责所有产品的创建，而是将具体创建工作交给子类去做，核心工厂仅仅负责给具体工厂必须实现的接口。这使得工厂模式可以允许系统在不修改工厂角色的情况下引进新产品。

工厂方法模式的主要优点是增加新的产品类时无须修改现有系统，并封装了产品对象的创建细节，系统具有良好的灵活性和可扩展性；其缺点在于增加新产品的同时需要增加新的工厂，导致系统类的个数成对增加，在一定程度上增加了系统的复杂性。

3 抽象工厂模式

4 建造者模式

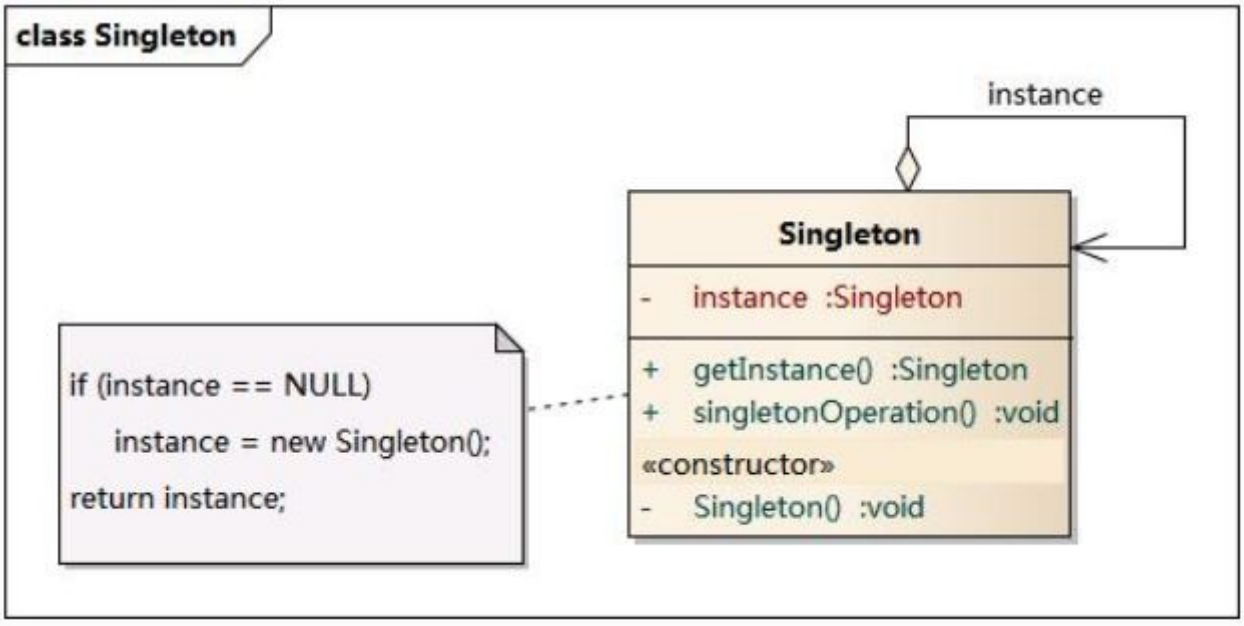
5 单例模式

单例模式主要解决一个全局使用的类频繁的创建和销毁的问题。

单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。

要点： 1.某个类只能有一个实例；2. 它必须自行创造这个实例；3. 它必须自行向整个系统提供这个实例。

5.1 模式结构
角色：单例



饿汉式实现线程安全：

```
#include<iostream>
using namespace std;
class singleton
{
private:
    static singleton* instance;
    singleton(){};
public:
    static singleton*  getsinstance(){
        return instance;
    }
    void operation(){
        cout << "singletion operation" << endl;
    }
    ~singleton() { delete instance; }
};
singleton* singleton::instance = new singleton();

int main()
{
    singleton *s = singleton::getsinstance();
    s->operation();
    return 0;
}
```

懒汉式实现线程安全，使用互斥锁进行实现
运行环境：linux

```
#include<iostream>
#include<pthread.h>
using namespace std;

class singleton
{
private:
    static singleton *instance;
    singleton(){pthread_mutex_init(&mutex,NULL);};
public:
    static pthread_mutex_t mutex;
    static singleton * getinstance()
    {
        if(instance == NULL){
            pthread_mutex_lock(&mutex);
            if(instance == NULL){
                instance = new singleton();
            }
            pthread_mutex_unlock(&mutex);
        }
        return instance;
    }
    void operation(){
        cout<<"operation"<<endl;
    }
};
singleton* singleton::instance = NULL;
pthread_mutex_t singleton::mutex;
int main()
{
    singleton *s = singleton::getinstance();
    s->operation();
    return 0;
}
```

5.2 模式分析

单例模式的目的是保证一个类仅有一个实例，并提供一个访问它的全局访问点。单例类拥有一个私有构造函数，确保用户无法通过new关键字直接实例化它。除此之外，该模式中包含一个静态私有成员变量与静态共有的工厂方法。该工厂方法负责检验实例的存在并实例化自己，然后存储在静态成员变量中，以确保只有一个实例被创建。

5.3 优点

提供了对唯一实例的受控访问并可以节约资源。

5.4 缺点

因为缺少抽象层而难以扩展，且单例类职责过重。