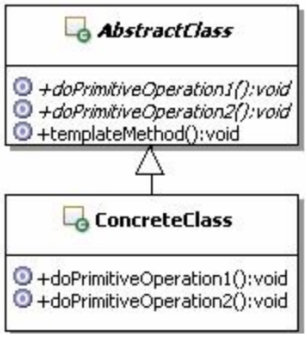
模板方法:定义一个操作中的算法的骨架(稳定)，而将一些步骤延迟(变化)到子类中。模板方法使得子类可以不改变(复用)一个算法的结构即可重定义该算法的某些特定步骤。



- AbstractClass：抽象类。用来定义算法骨架和原语操作，在这个类里面，还可以提供算法中通用的实现
- ConcreteClass：具体实现类。用来实现算法骨架中的某些步骤，完成跟特定子类相关的功能。

当多个类有相同的方法，并且逻辑相同，只是细节上有差异时，可以考虑使用模板模式。具体的实现上可以将相同的核心算法设计为模板方法，具体的实现细节有子类实现。

要点总结: Template Method模式是一种非常基础性的设计模式，在面向对象系统中有着大量的应用。它用最简洁的机制（虚函数的多态性）为很多应用程序框架提供了灵活的扩展点，是代码复用方面的基本实现结构。除了可以灵活应对子步骤的变化外，“不要调用我，让我来调用你”的反向控制结构是Template Method的典型应用。在具体实现方面，被Template Method调用的虚方法可以具有实现，也可以没有任何实现（抽象方法、纯虚方法），但一般推荐将它们设置为protected方法。

主要解决：一些方法通用，却在每一个子类都重新写了这一方法。

何时使用：有一些通用的方法。

如何解决：将这些通用算法抽象出来。

关键代码：在抽象类实现，其他步骤在子类实现。

优点： 1、封装不变部分，扩展可变部分。 2、提取公共代码，便于维护。 3、行为由父类控制，子类实现。

缺点：每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。

使用场景： 1、有多个子类共有的方法，且逻辑相同。 2、重要的、复杂的方法，可以考虑作为模板方法。

注意事项：为防止恶意操作，一般模板方法都加上 final 关键词。

```
#include <iostream>
using namespace std;
class Computer
{
public:
    void Product()
    {
        InstallCPU();
        InstallRam();
         InstallGPU();
     }
     virtual void InstallCPU() = 0;
```

```
    virtual void InstallRam() = 0;
    virtual void InstallGPU() = 0;
    virtual ~Computer(){};
};
class AppleComputer : public Computer
{
public:
    void InstallCPU() { cout << "Apple Install a itel i7" << endl; }
    void InstallRam() { cout << "Apple Install a 8G Ram" << endl; }
    void InstallGPU() { cout << "Apple Install a High speed GPU" << endl; }
};
class LenovoComputer : public Computer
{
public:
    void InstallCPU() { cout << "Lenovo Install a itel i5" << endl; }
    void InstallRam() { cout << "Lenovo Install a 4G Ram" << endl; }
    void InstallGPU() { cout << "Lenovo Install a powerfull GPU" << endl; }
};
int main()
{
    Computer * pApp = new AppleComputer();
    pApp->Product();
    delete pApp;

    Computer * pLenevo = new LenovoComputer();
    pLenevo->Product();
    delete pLenevo;
    return 0;
}
```

```
#include <iostream>
using namespace std;

class AbstractClass {
// 抽象类，其实就是一个抽象模板，定义并实现了一个模板方法。
// 这个模板方法一般是一个具体方法，给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中，推迟到了子类实现。
// 顶级逻辑也可能调用一些具体的方法。
public:
    void TemplateMethod() { // 模板方法，给出了逻辑骨架，而逻辑的组成是一些相应的抽象操作，推迟到子类实现.
        PrimitiveOperation1();
        PrimitiveOperation2();
    }
    virtual void PrimitiveOperation1() = 0;
    virtual void PrimitiveOperation2() = 0; // 一些抽象行为，放到子类实现
    virtual ~AbstractClass() {}
};

class ConcreteClassA : public AbstractClass {
// ConcreteClass实现父类所定义的一个或多个抽象方法。
// 每个AbstractClass都可以有任意多个ConcreteClass与之对应，
// 而每一个ConcreteClass都可以给出这些抽象方法的不同实现，从而使得顶级逻辑的实现各不相同。
public:
    void PrimitiveOperation1() {
```

```
        cout << "ConcreteClassA: PrimitiveOperation1 & ";
    }
    void PrimitiveOperation2() {
        cout << "PrimitiveOperation2" << endl;
    }
};

class ConcreteClassB : public AbstractClass {
public:
    void PrimitiveOperation1() {
        cout << "ConcreteClassB: PrimitiveOperation1 & ";
    }
    void PrimitiveOperation2() {
        cout << "PrimitiveOperation2" << endl;
    }
};

int main() {
    AbstractClass* aa = new ConcreteClassA();
    aa->TemplateMethod(); // ConcreteClassA: PrimitiveOperation1 & PrimitiveOperation2

    AbstractClass* ab = new ConcreteClassB();
    ab->TemplateMethod(); // ConcreteClassB: PrimitiveOperation1 & PrimitiveOperation2

    delete aa;
    delete ab;
    return 0;
}
```

```
192:DesignPattnsStudy weishichun$ ls TemplateMethod
TemplateMethod_1.cpp        TemplateMethod_2.cpp        TemplateMethod模板方法.pdf
192:DesignPattnsStudy weishichun$ g++ -o TemplateMethod1.out TemplateMethod_1.cpp
192:DesignPattnsStudy weishichun$ ./TemplateMethod1.out
Apple Install a itel i7
Apple Install a 8G Ram
Apple Install a High speed GPU
Lenovo Install a itel i5
Lenovo Install a 4G Ram
Lenovo Install a powerfull GPU
192:DesignPattnsStudy weishichun$ g++ -o TemplateMethod2.out TemplateMethod_2.cpp
192:DesignPattnsStudy weishichun$ ./TemplateMethod2.out
ConcreteClassA: PrimitiveOperation1 & PrimitiveOperation2
ConcreteClassB: PrimitiveOperation1 & PrimitiveOperation2
192:DesignPattnsStudy weishichun$
```