ПОВЕДЕНЧСКИЕ ПАТТЕРНЫ(НАБЛЮДАТЕЛЬ)

Поведенческие паттерны

Эти паттерны позволяют организовать эффективное и безопасное ***взаимодействие*** между ***объектами*** программы.

Общий список поведенческих паттернов следующий

1. Цепочка обязанностей
2. Команда
3. Итератор
4. Посредник
5. Снимок
6. Наблюдатель(изучаем)
7. Состояние
8. Стратегия(изучаем)
9. Шаблонный метод(изучаем)
10. Посетитель

Наблюдатель- это *поведенческий паттерн* проектирования, который создаёт ***механизм подписки,*** позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

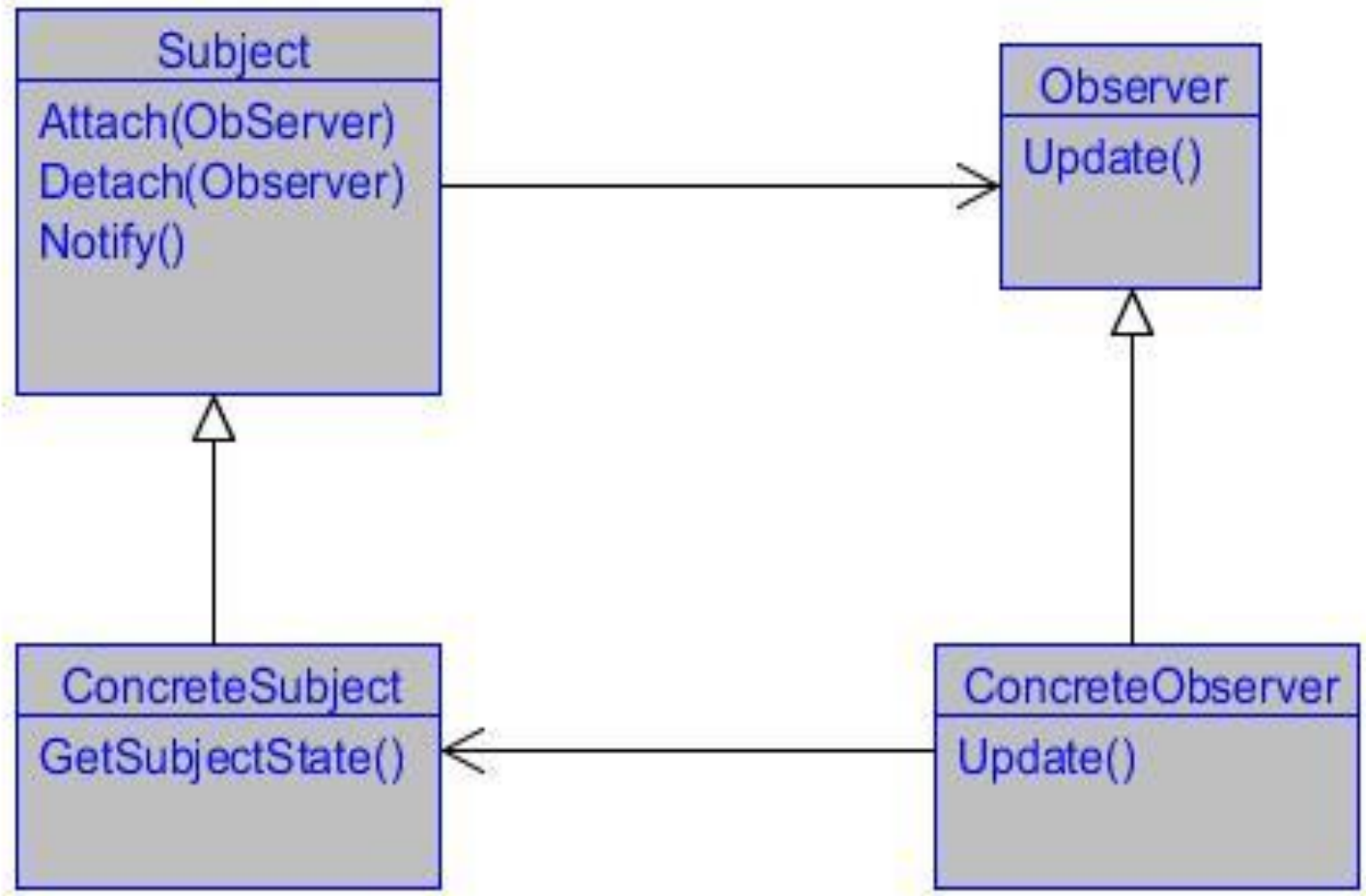
Без паттерна **Наблюдатель,** не обходится ни один SDK разработки графических интерфейсов.

Любой объект пользовательского интерфейса является источником сигналов.

Это могут быть щелчки или движения мышью, нажатия клавиш на клавиатуре и т.д.

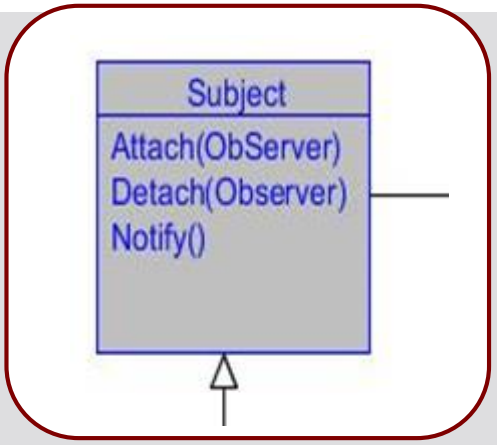
Например в Java для этих целей предусмотрен интерфейс **Listener,** а в **Qt** создана целая схема **сигналов-слотов,** которая расширяет синтаксис стандартного C++.

UML схема паттерна наблюдателя



1. Subject

Этот класс *отслеживает* всех ***наблюдателей*** и предоставляет возможность ***добавлять*** или удалять ***наблюдателей***.



Кроме того, именно класс отвечает *за обновление наблюдателей при любых изменениях*.

Предлагается внутри объекта **Subject** хранить список ссылок на объекты наблюдателей,

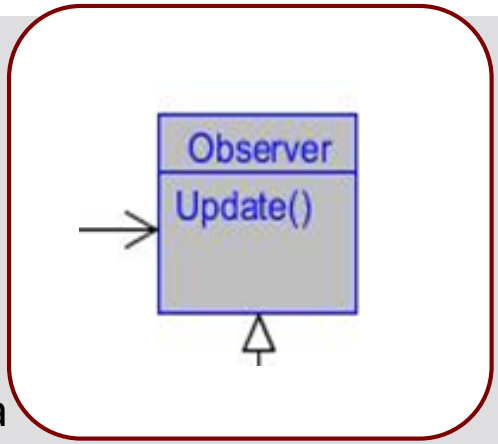
При этом **Subject** не должен вести список подписки самостоятельно.

Он предоставляет методы, с помощью которых ***наблюдатели*** могли бы добавлять или убирать себя из списка.

Когда в **Subject** будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-наблюдателей.

ConcreteSubject : - Этот класс является реальным классом, который реализует **Subject**. ***Этот класс является сущностью, изменение которой повлияет на другие объекты***.

Observer - представляет *интерфейс*, который определяет метод, который должен вызываться при каждом изменении.



Concrete Observer выполняют что что -то в ответ на оповещение, пришедшее от **Subject.**

Эти классы должны следовать общему интерфейсу ***Observer*** чтобы ***Subject*** не зависел от конкретных классов наблюдателей.

Client создает объекты **ConcreteSubject** и **ConcreteObserver,**а затем регистрирует **ConcreteObserver** на обновления в **ConcreteSubject.**

Рассмотрим на конкретном примере

Легенда

У нас есть магазины, которые хотят обновлять цену, установленную владельцем продукта.

Нужно определить сущности и понять какую роль эти сущности будут играть при реализации паттерна Наблюдатель.

Пример

```
class IObserver
{public:
virtual void Update(float price) = 0;
};
class Shop : IObserver
{//Название магазина

std::string name;
float price;
public:
Shop(std::string n);
void Update(float price);
};

void Shop::Update(float price)
{
this->price = price;
std::cout << "Price at " << name << " is now " << price << "\n";
}
```

```
class ASubject
{
// Давайте отслеживать все магазины, которые мы наблюдаем
std::vector<Shop*> list;

public:
void Attach(Shop *product);
void Detach(Shop *product);
void Notify(float price);
};
```

```
void ASubject::Attach(Shop *shop)
{
list.push_back(shop);
}

void ASubject::Detach(Shop *shop)
{
list.erase(std::remove(list.begin(), list.end(), shop), list.end());
}

void ASubject::Notify(float price)
{
for (vector<Shop*>::const_iterator iter = list.begin(); iter !=
list.end(); ++iter)
{
if (*iter != 0)
{
(*iter)->Update(price);
}
}
}
```

```
class ConcreteProduct : public ASubject
{
public:
void ChangePrice(float price);
};

void ConcreteProduct::ChangePrice(float price)
{
        Notify(price);
}
```

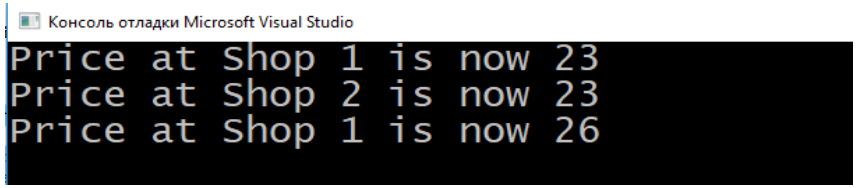
```
int main()
{
ConcreteProduct product;
 // У нас есть два магазина, которые хотят обновлять цену,
установленную владельцем продукта
Shop shop1("Shop 1");
Shop shop2("Shop 2");
product.Attach(&shop1);
product.Attach(&shop2);

// Теперь попробуем изменить цену товара, это должно автоматически
обновить магазины
product.ChangePrice(23.0f);

// Теперь shop2 не интересуется новыми ценами, поэтому они
отписываются
product.Detach(&shop2);

// Теперь попробуем снова изменить цену товара
product.ChangePrice(26.0f);

}
```



ПАТТЕРН **СТРАТЕГИЯ**

ПОВЕДЕНЧЕСКИЙ ПАТТЕРН: СТРАТЕГИЯ

Паттерн Стратегия.

Правило - «Определите семейство алгоритмов, инкапсулируйте каждый и сделайте их взаимозаменяемыми. **Взаимозаменять** можно прямо во время исполнения программы

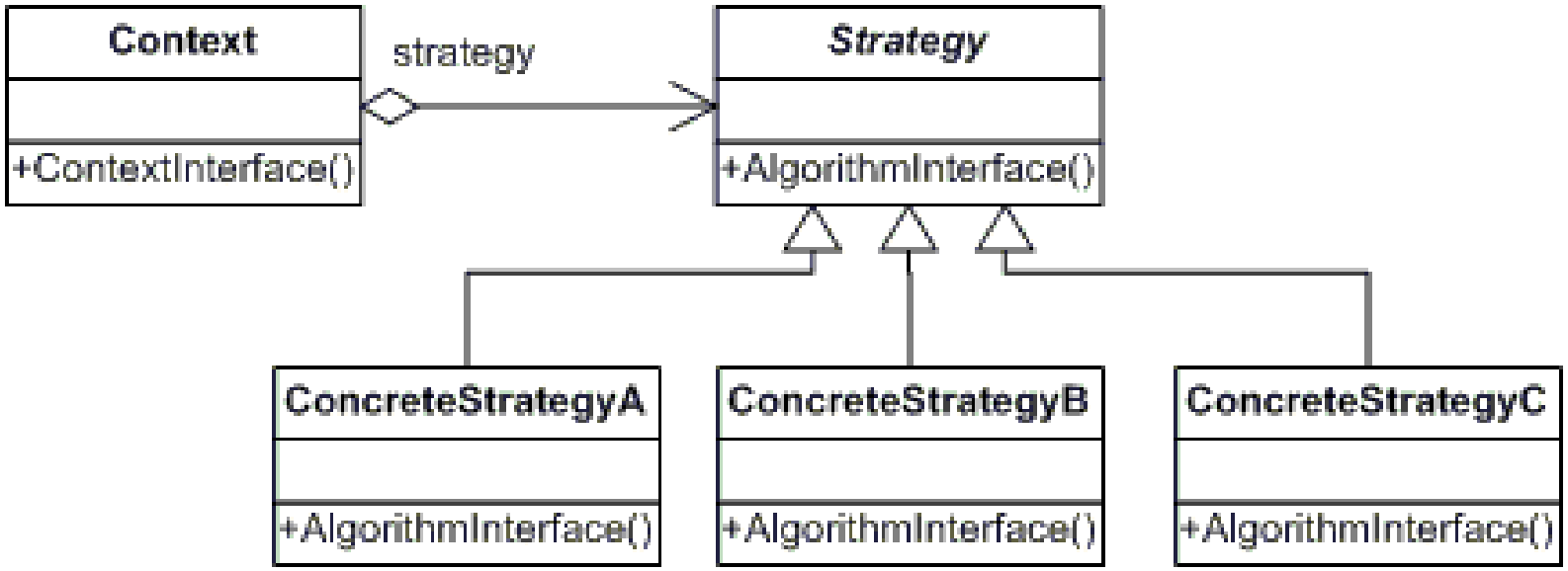
Ключевыми фразами здесь являются «Семейство алгоритмов», «инкапсуляция» и «взаимозаменяемость».

Другими словами, шаблон стратегии содержит набор функций, выполняющих похожие, но не идентичные задания.

Паттерн Стратегия предлагает определить семейство схожих алгоритмов, которые часто **изменяются** или **расширяются**, и вынести их в собственные классы, называемые ***стратегиями***.

Т.Е. изначальный класс сам не выполняет какой либо алгоритм, он будет являться контекстом(context class) и будет ссылаться на одну из стратегий делегируя ей выполнение. Для смены алгоритма потребуется добавить в класс другой объект-стратегию.

ПОВЕДЕНЧЕСКИЙ ПАТТЕРН: СТРАТЕГИЯ



ПОВЕДЕНЧЕСКИЙ ПАТТЕРН: ОПИСАНИЕ

1. **Контекст** хранит ссылку на объект конкретной стратегии, работая с ним через общий интерфейс стратегий.

2. **Стратегия** определяет интерфейс, общий для всех вариаций алгоритма. Контекст использует этот интерфейс для вызова алгоритма.

3. **Конкретные стратегии** реализуют различные вариации алгоритма.

4. Во время выполнения программы контекст получает вызовы от клиента и делегирует их объекту конкретной стратегии.

5. Клиент должен создать объект конкретной стратегии и передать его в конструктор контекста. Кроме этого, клиент должен иметь возможность заменить стратегию на лету, используя сеттер(Set)). Благодаря этому, **контекст** не будет знать о том, какая именно стратегия сейчас выбрана.

ПОВЕДЕНЧЕСКИЙ ПАТТЕРН: ИСПОЛЬЗОВАНИЕ

Паттерн стратегию используют тогда, когда

1. нужно использовать различные вариации какого-то алгоритма внутри одного объекта.

2. есть множество похожих классов, отличающихся только некоторым поведением.

3. нужно скрыть детали реализации алгоритмов для других классов. *Стратегия позволяет изолировать код, данные и зависимости алгоритмов от других объектов, скрыв эти детали внутри классов-стратегий.*

4. различные вариации алгоритмов реализованы в виде ***многочисленных веток условного оператора.*** Каждая ветка такого оператора представляет собой вариацию алгоритма.

ПОВЕДЕНЧЕСКИЙ ПАТТЕРН: ПРИМЕНЕНИЕ

Для того чтобы применить паттерн стратегию нужно

1. Определить алгоритм, который подвержен частым изменениям. Также подойдёт алгоритм, имеющий несколько вариаций, которые выбираются во время выполнения программы.

2. Создать интерфейс стратегий, описывающий этот алгоритм. Он должен быть общим для всех вариантов алгоритма.

3. Поместить вариации алгоритма в собственные классы, которые реализуют этот интерфейс.

4. В классе контекста создать поле для хранения ссылки на текущий объект-стратегию, а также метод для её изменения.

Контекст должен работать с объектом только через общий интерфейс стратегий.

ПОВЕДЕНЧЕСКИЙ ПАТТЕРН: ПРИМЕР

Рассмотрим задачу сжатия файлов. Существует несколько стратегий сжатия файлов(ZIP,RAR,ARJ).

```
#include <iostream>
#include <string>

// Иерархия классов, определяющая алгоритмы сжатия файлов
class ICompression
{
  public:
    virtual ~Compression() {}
    virtual void compress( const string & file ) = 0;
};

class ZIP_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "ZIP compression" << endl;
    }
};
```

ПОВЕДЕНЧЕСКИЙ ПАТТЕРН: ПРИМЕР

```
class ARJ_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "ARJ compression" << endl;
    }
};

class RAR_Compression : public Compression
{
  public:
    void compress( const string & file ) {
        cout << "RAR compression" << endl;
    }
};
```

ПОВЕДЕНЧЕСКИЙ ПАТТЕРН: ПРИМЕР

```
// Класс для использования
class Compressor
{
  public:
    Compressor( Compression* comp): p(comp) {}
   ~Compressor() { delete p; }
    void compress( const string & file ) {
      p->compress( file);
    }
  private:
    Compression* p;
};
```

ПОВЕДЕНЧЕСКИЙ ПАТТЕРН: ПРИМЕР

```
int main()
{
  Compressor* p = new Compressor( new ZIP_Compression);
  p->compress( "file.txt");
  delete p;
  return 0;
}
```

ПОВЕДЕНЧЕСКИЕ ПАТТЕРНЫ: ШАБЛОННЫЙ МЕТОД

ШАБЛОННЫЙ МЕТОД

Шаблонный метод — это поведенческий паттерн проектирования, который определяет ***скелет алгоритма***, перекладывая ответственность за некоторые его шаги на подклассы.

Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

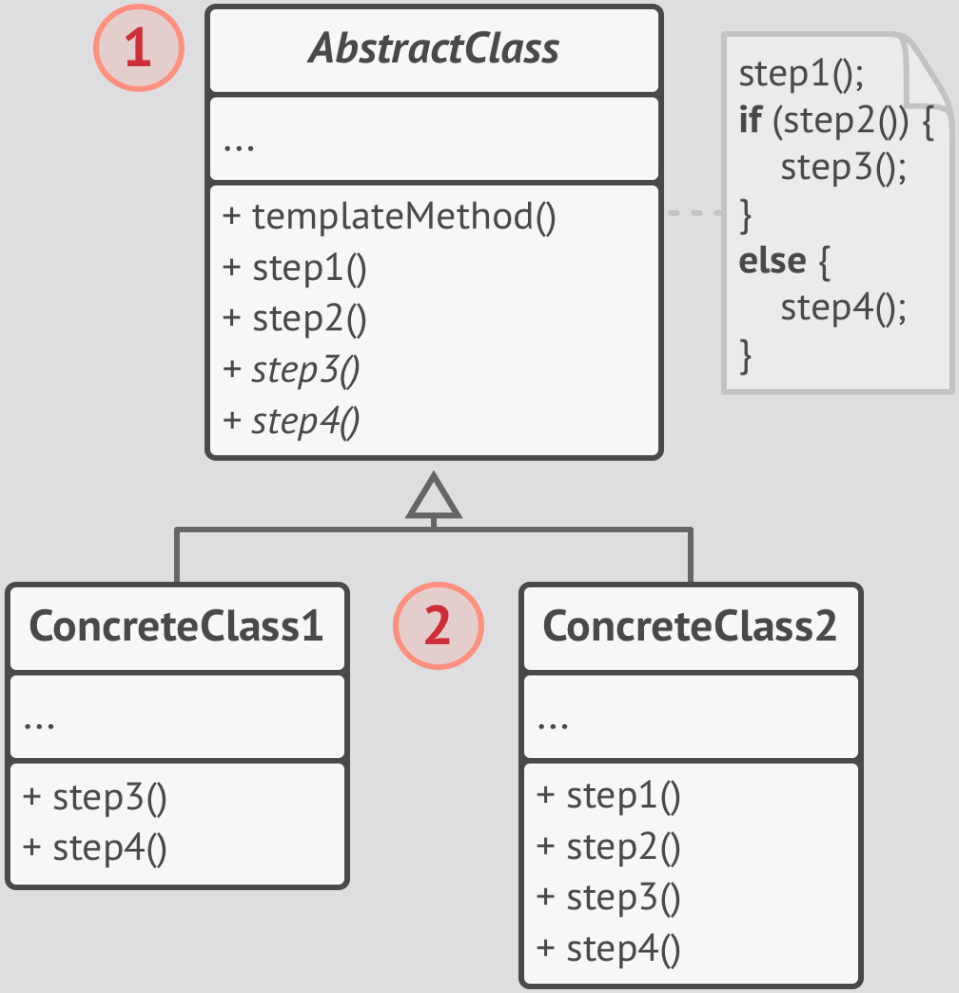
Базовый класс определяет шаги алгоритма с помощью *абстрактных операций*, а производные классы их реализуют.

Что решаем?

Например, имеются два разных, но в тоже время очень похожих компонента.

Требуется внести изменения в оба компонента, избежав дублирования кода.

ШАБЛОННЫЙ МЕТОД



ШАБЛОННЫЙ МЕТОД

1. Абстрактный класс определяет шаги алгоритма и содержит ***шаблонный метод***, *состоящий из вызовов этих шагов.*

Шаги могут быть как ***абстрактными***, так и содержать ***реализацию по умолчанию.***

2. Конкретный класс переопределяет некоторые (или все) шаги алгоритма. Конкретные классы *не переопределяют* сам шаблонный метод.

ШАБЛОННЫЙ МЕТОД

Шаблонный метод нужно применять

1. Когда подклассы должны расширять базовый алгоритм, не меняя его структуры.

Шаблонный метод позволяет подклассам расширять определённые шаги алгоритма через наследование, не меняя при этом структуру алгоритмов, объявленную в базовом классе.

2.Когда существует несколько классов, делающих одно и то же с незначительными отличиями. При редактировании одного класса, приходится вносить такие же правки и в остальные классы.

Паттерн шаблонный метод предлагает создать для похожих классов ***общий суперкласс*** *и оформить в нём главный алгоритм в* ***виде шагов****.*

Отличающиеся шаги *можно переопределить в* ***подклассах****.*

Это позволит убрать дублирование кода в нескольких классах с похожим поведением, но отличающихся в деталях.

ШАБЛОННЫЙ МЕТОД

Шаги реализации

1. Необходимо **исследовать алгоритм** и решить, **разбивается ли он на шаги**. Далее **определить**, какие шаги будут стандартными для всех, а какие должны изменяться(*определяться подклассами*).

2. Создать абстрактный базовый класс и определить в нём **шаблонный метод**. Этот метод состоит из вызовов шагов алгоритма.

3. Добавить в абстрактный класс методы для каждого из шагов алгоритма. Можно сделать эти методы ***абстрактными*** или добавить какую-то ***реализацию по умолчанию***.

Все подклассы должны будут реализовать абстрактные методы.

4. Продумать применение **хуков** в алгоритме. Чаще всего, хуки располагают между основными шагами алгоритма, а также до и после всех шагов.

5. Создать конкретные классы, унаследовав их от абстрактного класса. Реализовать в них все недостающие шаги и хуки.

ШАБЛОННЫЙ МЕТОД

Методы шаблонов являются основополагающим методом повторного использования кода.

Методы шаблонов приводят к перевернутой структуре управления, которая иногда упоминается как «**принцип Голливуда**», то есть принцип ***«Не звоните нам, мы вам сами перезвоним».***

Это относится к тому, что родительский класс вызывает операции подкласса, а не наоборот.

ХУКИ В ПАТТЕРНЕ ШАБЛОННЫЙ МЕТОД

Для шаблонных методов важно указать, какие операции являются **Хуками(Hook)**(которые могут быть переопределены) и какие абстрактными операциями (которые ***должны*** быть переопределены).

Хуки это ***методы***, которые ничего не делают, или, другими словами ***реализуют поведение по умолчанию в абстрактном классе***, но могут быть переопределены в подклассах.

Чтобы эффективно использовать абстрактный класс, авторы подкласса должны понимать, какие операции предназначены для переопределения.

Подкласс может распространять поведение операции родительского класса, переопределяя операцию и вызывая родительскую операцию явно:

Например:

```
void DerivedClass :: Operation () {
      // Расширение поведения DerivedClass
      ParentClass :: Операция ();
  }
```

К сожалению, легко забыть вызвать унаследованную операцию.

ХУКИ В ПАТТЕРНЕ ШАБЛОННЫЙ МЕТОД

Мы можем преобразовать такую операцию в шаблонный метод, чтобы обеспечить родительский контроль над тем, как подклассы расширяют его.

Идея заключается в том, чтобы вызвать операцию Hook из метода шаблона в родительском классе.

Затем подклассы могут переопределить эту операцию.

```
void ParentClass :: Operation () {
        // Поведение ParentClass
        HookOperation ();  }
```

HookOperation ничего не делает в ParentClass:

```
void ParentClass :: HookOperation () {}
```

Подклассы переопределяют HookOperation, чтобы расширить свое поведение:

```
void DerivedClass :: HookOperation () {
        // расширение производного класса
    }
```

ПРИМЕР

Рассмотрим реализацию шаблонного метода на конкретном примере.

Рассмотрим два класса, один из которых реализует ***процесс варки кофе***, ***второй заваривания чая.***

ПРИМЕР

```
class Coffee
{
public:
    void boilWater ( ){
        cout<<"Boiling water\n";
    }
    void brewCoffeeGrinds ( ){
    cout<<"Dripping coffee thru filter/n";
    }
    void pourInCup(){
        cout<<"Pouring into cup/n";
    }
    void addSugarAndMilk(){
        cout<<"Adding Sugar and Milk/n";
    }

    void prepareRecipe ( ){
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }
};
```

ПРИМЕР

```
class Tea
{
public:

    void boilWater ( ){
        cout<<"Boiling water\n";
    }
    void steepTeaBag ( ){
        cout<<"Steeping the Tea\n";
    }
    void pourInCup ( ){
        cout<<"Pouring into cup\n";
    }
    void addLemon ( ) {cout<<"Adding Lemon\n";
    }
    void prepareRecipe ( )
    {
        boilWater  ( );
        steepTeaBag( );
        pourInCup  ( );
        addLemon   ( );
    }
};
```

ПРИМЕР

В методе **`prepareRecipe`** приготовление напитка реализуется отдельными методами.

В нашем примере есть дублирование кода — следовательно, это хороший признак того, что нам нужно изменить дизайн.

Нужно абстрагировать общность и перенести в базовый класс, так как процесс приготовления кофе и чая похож.

В соответствии со схемой шаблонного метода нужно выделить общность в процессе приготовления чая и кофе.

И так, что общего?

1. Вскипятить немного воды;
2. Заварить чай или кофе;
3. Вылить напиток в чашку
4. Использовать подходящие добавки к чаю или кофе.

ПРИМЕР

Выделим общий интерфейс для различных методов.

И так, для приготовления кофе используются методы **brewCoffeeGrinds()** и **addSugarAndMilk()**,

а для приготовления чая методы **steepTeaBag( )** и **addLemon( ).**

Здесь методы **brewCoffeeGrinds()** и **steepTeaBag()** похожи их можно объединить в абстрактный метод **brew( )**,

методы **addLemon( )** и **addSugarAndMilk()** в метод **addCondiments().**

Тогда реализацию можно представить

```
class CaffeineBeverage
{
public:
    void prepareRecipe ( ){// Это Шаблонный метод, так
представляет алгоритм приготовления напитков. Здесь каждый шаг
алгоритма является методом.
        boilWater ( );
        brew ( );
        pourInCup ( );
        addCondiments ( );
    }
protected:
    virtual void brew ( )= 0;
    virtual void addCondiments( )= 0;
    void boilWater ( ){
        cout<<"Boiling Water\n";
    }
    void pourInCup ( ){
        cout<<"Pouring into cup\n";
    }
};
```

```
class Tea: public CaffeineBeverage
{
protected:
    void brew ( ){
        cout<<"Steeping the Tea\n";
    }
    void addCondiments ( ){
        cout<<"Adding Lemon\n";
    }
};


class Coffee: public CaffeineBeverage
{
public:
    void brew ( ){
        cout<<"Dripping Coffee Thru the Filters\n";
    }
    void addCondiments ( ){
        cout<<"Adding Sugar and Milk\n";
    }
};
```

Клиентский код

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

CaffeineBeverage *array[] ={new Tea(), new Coffee()};
  for (int i = 0; i < 2; i++){
    array[i]->prepareRecipe();
    cout << '\n';
  }
    for (int i = 0; i < 2; i++){
        delete array[i];
    }
    return 0;
}
```

Замечание:

В реализации на языке C++, все операции шаблонного метода должны быть объявлены как защищенные(protected) методы, что обеспечит вызов этих операций только в шаблонном методе. Сам шаблонный метод нельзя переопределять в порожденных классах, следовательно он должен быть не виртуальным(non-virtual).

ПРИМЕР

Рассмотрим реализацию класса **CaffeineBeverage** в которой определим одну операцию Хук(Hook).

В качестве хука определим метод, определяющий поведение по умолчанию.

```
bool customerWantsCondiments( ){
      return true;
      }
```

В метод `void prepareRecipe ( )` добавим небольшое условие вида,

```
if (customerWantsCondiments ( ) ){
addCondiments ( );
}
```

Т.е.метод **addCondiments ( )**;будет вызван тогда когда заказчик хочет получить дополнения к напитку. По умолчанию, если хук не будет переопределен, **addCondiments** будет вызываться всегда.

```
class CaffeineBeverageWithHook{
public:
    void prepareRecipe ( ){
        boilWater ( );
        brew ( );
        pourInCup ( );
        addCondiments ( );
    }
protected:
    virtual void brew ( )= 0;
    virtual void addCondiments( )= 0;
    void boilWater ( ){
        cout<<"Boiling Water\n";
    }
    void pourInCup ( ){
        cout<<"Pouring into cup\n";
    }
    // Это Hook. Этот метод можно переопределить, но не
обязательно.Он задает поведение по умолчанию
    bool customerWantsCondiments( ){
        return true;
    }
};
```