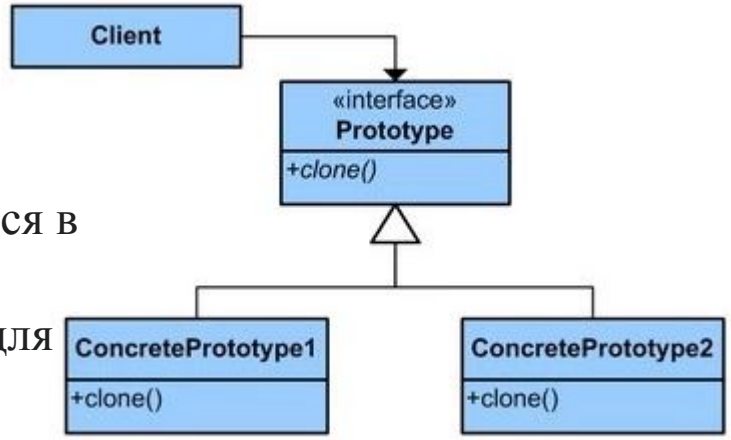
Паттерны проектирования

Prototype



Паттерн используется чтобы:

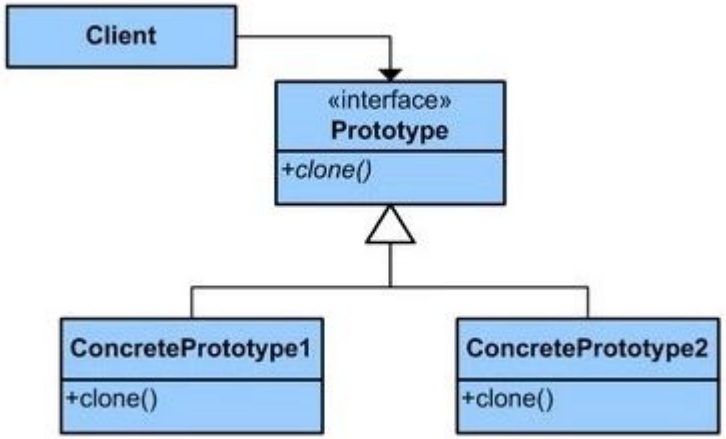
- избежать дополнительных усилий по созданию объекта стандартным путём (имеется в виду использование конструктора, так как в этом случае также будут вызваны конструкторы всей иерархии предков объекта), когда это непозволительно дорого для приложения.
- избежать наследования создателя объекта (object creator) в клиентском приложении, как это делает паттерн abstract factory.

Используйте этот шаблон проектирования, когда системе безразлично как именно в ней создаются, компонуются и представляются продукты:

- Создаваемые экземпл. класса определяются во время выполнения, например с помощью динамической загрузки;
- избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов;
- экземпляры класса могут находиться в одном из нескольких различных состояний. Может оказаться удобнее установить соответствующее число прототипов и клонировать их, а не создавать их каждый раз экземпл. класса вручную в подходящем состоянии.

Prototype

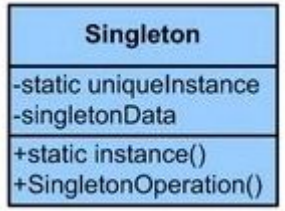
```
class Meal {
    public: virtual ~Meal();
    virtual void eat() = 0;
    virtual Meal *clone() const = 0; //...
};
class Spaghetti : public Meal {
    public: Spaghetti( const Spaghetti &);
    void eat();
    Spaghetti *clone() const { return new Spaghetti( *this ); } //...
};
```



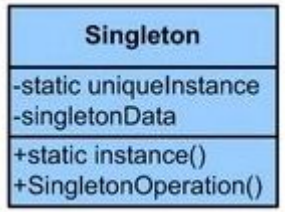

```
/** * Prototype Class */
public class Cookie implements Cloneable {
    protected int weight;
   @Override public Cookie clone() throws CloneNotSupportedException {
        Cookie copy = (Cookie) super.clone();
//В реальной реализации этого шаблона теперь вы можете изменить ссылки на трудно производимую
деталь из копий, которые хранятся внутри прототипа.
        return copy;
       } }
/** * Concrete Prototypes to clone */
public class CoconutCookie extends Cookie { }
```

Singleton

```
public class Singleton {
  private static Singleton instance;
  private Singleton () {};

  public static synchronized Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```



Singleton

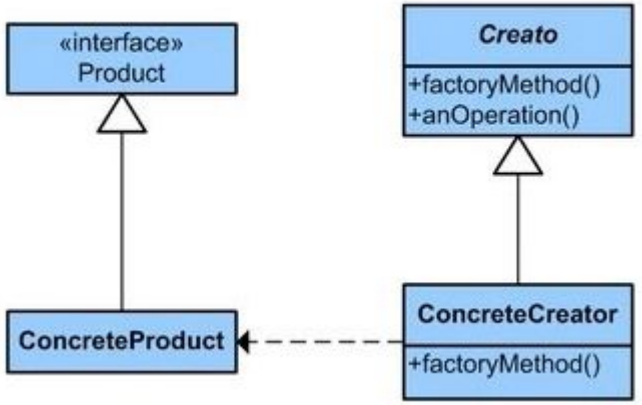


Применение:
должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам; единственный экземпляр должен расширяться путём порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

Достоинства:
Наведение порядка в глобальном пространстве имён.
Ускорение начального запуска программы, если есть множество одиночек, которые не нужны для запуска.
Упрощение кода инициализации — система автоматически неявно отделит нужные компоненты от ненужных и проведёт топологическую сортировку.

Недостатки:
Усложняется контроль за межпоточными гонками и задержками
Требуются особые функции для модульного тестирования.
Компоненты не должны иметь неявных связей между собой, иначе небольшое изменение — в программном коде, файле настроек, сценарии пользования — может спутать порядок и вызвать трудноуловимую ошибку.

Factory method (virtual constructor)

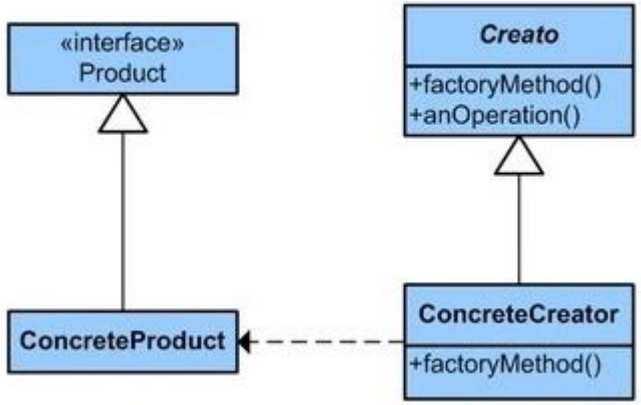


Фабричный метод - это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

Когда надо применять паттерн:

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

Factory method (virtual constructor)

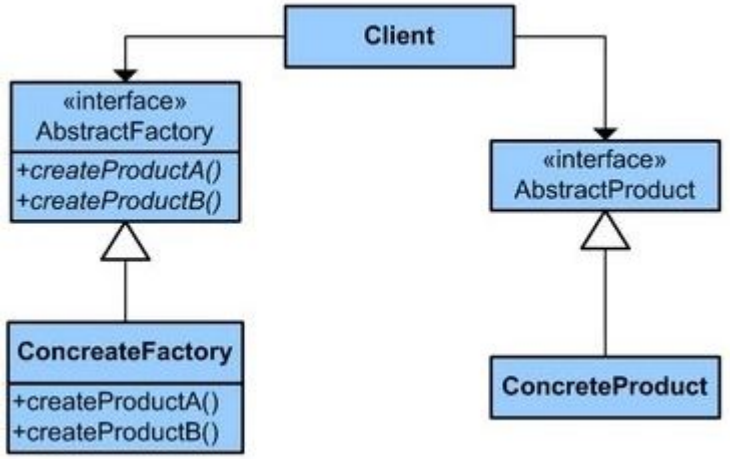



```
abstract class Product{}

class ProductA extends Product{}

class ProductB extends Product{}

abstract class FactoryAbstract
{
    public function create($type)
    {
        switch ($type) {
            case 'A':
                return new ProductA();
            case 'B':
            default:
                return new ProductB();
        }
    }
}
```

```
class Factory extends FactoryAbstract{}

$factory = new Factory();
$productA = $factory->create('A');
```

Abstract Factory

Паттерн "**Абстрактная фабрика**" (Abstract Factory) предоставляет интерфейс для создания семейств взаимосвязанных объектов с определенными интерфейсами без указания конкретных типов данных объектов.

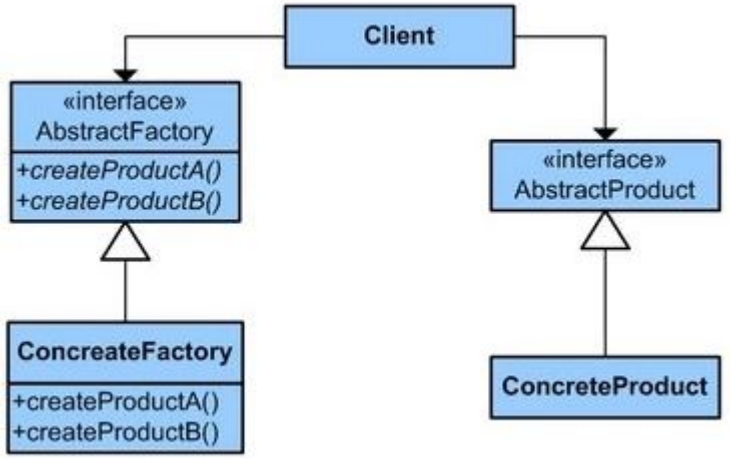


Когда использовать абстрактную фабрику

- Когда система не должна зависеть от способа создания и компоновки новых объектов
- Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными
- Система должна конфигурироваться одним из семейств составляющих её объектов.
- Требуется предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Abstract Factory

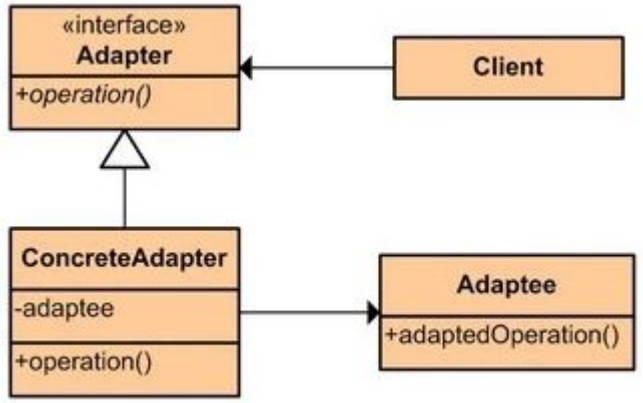
Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.



Плюсы:
изолирует конкретные классы;
упрощает замену семейств продуктов;
гарантирует сочетаемость продуктов.
Минусы:
сложно добавить поддержку нового вида продуктов.

Adapter

Паттерн **Адаптер** предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.

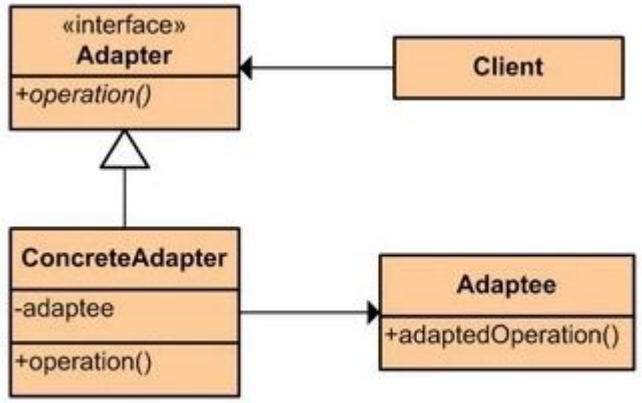


Когда надо использовать Адаптер:

- Когда необходимо использовать имеющийся класс, но его интерфейс не соответствует потребностям
- Когда надо использовать уже существующий класс совместно с другими классами, интерфейсы которых не совместимы
- Паттерн Adapter позволяет повторно использовать уже имеющийся код, адаптируя его несовместимый интерфейс к виду, пригодному для использования.

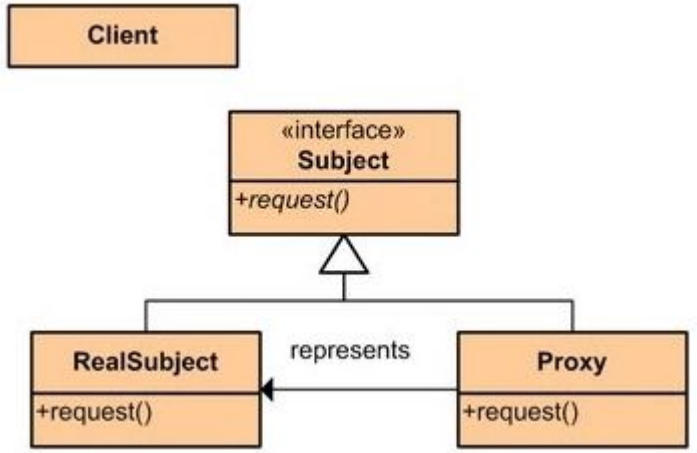
Adapter

```
#include <iostream>

// Уже существующий класс температурного датчика окружающей среды
class FahrenheitSensor
{
  public:
    // Получить показания температуры в градусах Фаренгейта
    float getFahrenheitTemp() {
      float t = 32.0;
      // ... какой то код
      return t;
    }
};

class Sensor
{
  public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
};
```




```
class Adapter : public Sensor
{
private:
    FahrenheitSensor* p_fsensor;
  public:
    Adapter( FahrenheitSensor* p ) : p_fsensor(p) {
    }
float getTemperature() {
      return (p_fsensor->getFahrenheitTemp()-32.0)*5.0/9.0;
    }
};

int main()
{
  Sensor* p = new Adapter( new FahrenheitSensor);
  cout << "Celsius temperature = " << p->getTemperature() << endl;
```

Proxy

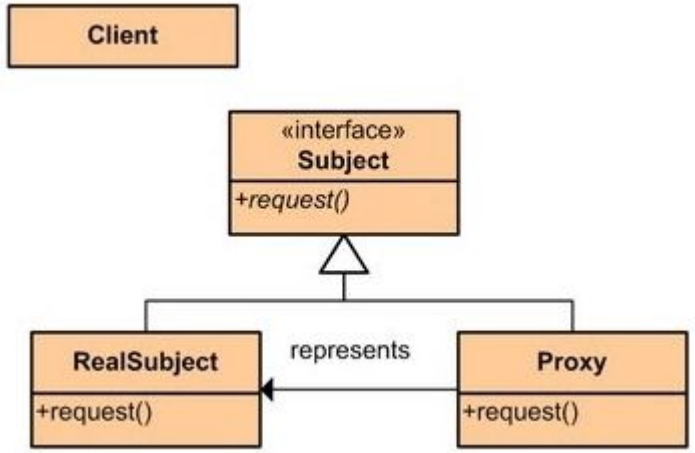


Паттерн **Заместитель** (Proxy) предоставляет объект-заместитель, который управляет доступом к другому объекту. То есть создается объект-суррогат, который может выступать в роли другого объекта и замещать его.

Когда использовать прокси?

- Когда надо осуществлять взаимодействие по сети, а объект-проси должен имитировать поведения объекта в другом адресном пространстве. Использование прокси позволяет снизить накладные издержки при передачи данных через сеть. Подобная ситуация еще называется **удалённый заместитель (remote proxies)**
- Когда нужно управлять доступом к ресурсу, создание которого требует больших затрат. Реальный объект создается только тогда, когда он действительно может понадобится, а до этого все запросы к нему обрабатывает прокси-объект. Подобная ситуация еще называется **виртуальный заместитель (virtual proxies)**
- Когда необходимо разграничить доступ к вызываемому объекту в зависимости от прав вызывающего объекта. Подобная ситуация еще называется **защищающий заместитель (protection proxies)**
- Когда нужно вести подсчет ссылок на объект или обеспечить потокобезопасную работу с реальным объектом. Подобная ситуация называется **"умные ссылки" (smart reference)**

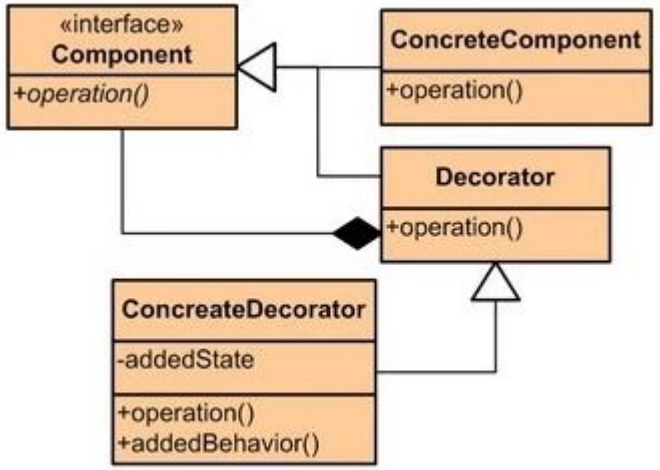
Proxy




```
class Client
{
   void Main()
   {
      Subject subject = new Proxy();
      subject.Request();
   }
}
abstract class Subject
{
   public abstract void Request();
}
```

```
class RealSubject : Subject
{
   public override void Request()  {}
}
class Proxy : Subject
{
   RealSubject realSubject;
   public override void Request()
   {
      if (realSubject == null)
         realSubject = new RealSubject();
      realSubject.Request();
   }
}
```

Decorator



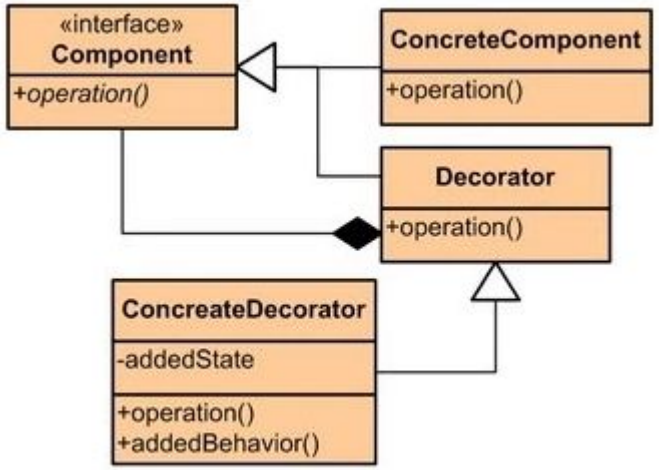
Декоратор (Decorator) представляет структурный шаблон проектирования, который позволяет динамически подключать к объекту дополнительную функциональность. Для определения нового функционала в классах нередко используется наследование. Декораторы же предоставляет наследованию более гибкую альтернативу, поскольку позволяют динамически в процессе выполнения определять новые возможности у объектов.

Применять следует:

- Когда надо динамически добавлять к объекту новые функциональные возможности. При этом данные возможности могут быть сняты с объекта
- Когда применение наследования неприемлемо. Например, если нам надо определить множество различных функциональностей и для каждой функциональности наследовать отдельный класс, то структура классов может очень сильно разрастись. Еще больше она может разрастись, если нам необходимо создать классы, реализующие все возможные сочетания добавляемых функциональностей.

Decorator

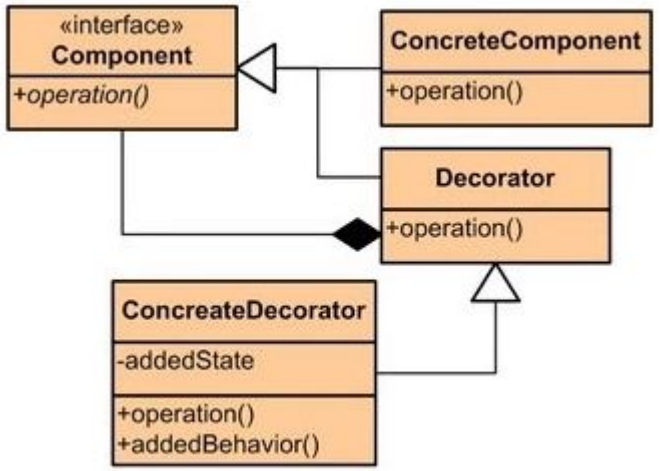
```
abstract class Pizza
{
    public Pizza(string n)
    {
        this.Name = n;
    }
    public string Name {get; protected set;}
    public abstract int GetCost();
}

class BulgerianPizza : Pizza
{
    public BulgerianPizza()
        : base("Болгарская пицца")
    { }
    public override int GetCost()
    {
        return 8;
    }
}
```




```
Pizza pizza3 = new BulgerianPizza();
        pizza3 = new TomatoPizza(pizza3);
        pizza3 = new CheesePizza(pizza3);// болгарская пиццы с томатами и сыром
```

Сначала объект BulgerianPizza обертывается декоратором TomatoPizza, а затем CheesePizza. И таких обертываний мы можем сделать множество. Просто достаточно унаследовать от декоратора класс, который будет определять дополнительный функционал.

Decorator

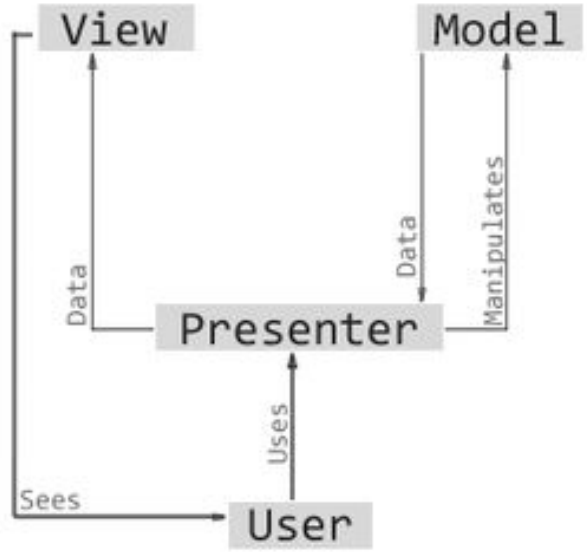



```
class TomatoPizza : PizzaDecorator
{
   public TomatoPizza(Pizza p)
      : base(p.Name + ", с томатами", p)
   { }

   public override int GetCost()
   {
      return pizza.GetCost() + 3;
   }
}
```

```
class CheesePizza : PizzaDecorator
{
   public CheesePizza(Pizza p)
      : base(p.Name + ", с сыром", p)
   { }

   public override int GetCost()
   {
      return pizza.GetCost() + 5;
   }
}
```

```
abstract class PizzaDecorator : Pizza
{
   protected Pizza pizza;
   public PizzaDecorator(string n, Pizza pizza) : base(n)
   {
      this.pizza = pizza;
   }
}
```

MVP

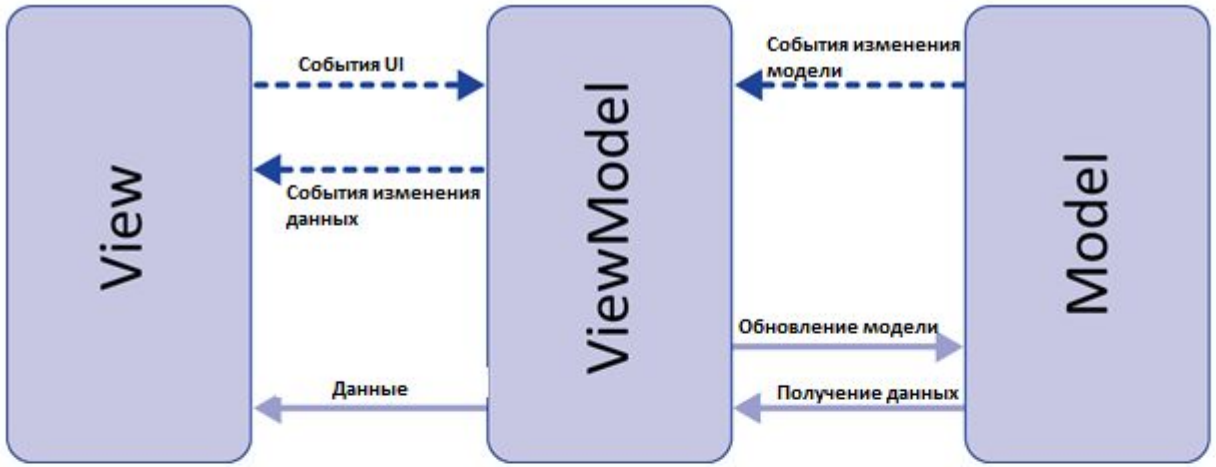


Всё это можно сравнить с работой издательства:

1. Автор готовит текст (модель).
2. Текст получает издатель (представитель).
3. Если с текстом всё в порядке, издатель передаёт его в отдел вёрстки (вид).
4. Верстальщики готовят книгу, которую начинают продавать читателям (пользователи).
5. Если пользователи как-то реагируют на книгу, например, пишут письма в издательство, то работа может начаться заново. Допустим, кто-то может заметить в книге неточность, тогда издатель передаст информацию автору, автор её обновит и так далее.

```
public class MyModel
{
    private int _state = 0;
    public MyModel(initState)
    {
        _state = initState;
    }
    public getState(){
        return _state;
    }
}
public class MyView: IView
{
    private IPresenter _presenter;

    public MyView()
    {
        _presenter = new MyPresenter(this);
    }
}

public class MyPresenter: IPresenter
{
    private IView _view;
    private MyModel _myModel;

    public MyPresenter(IView view)
    {
        _view = view;
        _myModel= new MyModel(1);
    }
}
```

MVVP



Данный подход позволяет связывать элементы представления со свойствами и событиями View-модели. Можно утверждать, что каждый слой этого паттерна не знает о существовании другого слоя.

Признаки View-модели:

- Двухсторонняя коммуникация с представлением;
- View-модель — это абстракция представления. Обычно означает, что свойства представления совпадают со свойствами View-модели / модели
- View-модель не имеет ссылки на интерфейс представления (IView). Изменение состояния View-модели автоматически изменяет представление и наоборот, поскольку используется механизм связывания данных (Bindings)
- *Один экземпляр View-модели связан с одним отображением.*

Используется в ситуации, когда возможно связывание данных без необходимости ввода специальных интерфейсов представления (т.е. отсутствует необходимость реализовывать IView); Частым примером является технология WPF.

```
vm.data.observe(this){
        t.text=""+vm.get()
    }

class MVM(): ViewModel() {
    var data=MutableLiveData<Int>()
    init {
        data.value=0
    }
```