Вопросы

1. Отличие интерфейсного и виртуального методов
2. Почему в конструкторе/деструкторе не стоит вызывать виртуальные методы
3. «Посетитель»
4. Физическое представление делегата
5. Множественное наследование C++
6. Виртуальное наследование
7. Реализация интерфейсных вызовов в COM и в C#
8. Зачем нужна заглушка
9. Отличие метода от процедуры
10. Делегат
11. Отличие виртуального и обычного методов
12. «...и про COM интерфейсный»
13. Вызов виртуального метода
14. Зачем в паттерне «Одиночка» две проверки, какую из них можно опустить
15. Отличие делегата от события
16. Паттерн «Наблюдатель»
17. Отличие делегата и приёма «Наблюдатель»
18. «Мост»
19. «Одиночка»
20. Виды заместителей(сюда надо прокси)
21. Выдача номеров в таблице методов
22. Обертка (декоратор) и адаптер
23. Итератор

Интерфейс -набор методов которые обязан в себе иметь наследуемый класс.
Абстрактный класс – от обычного его отличает контракт(при наследование абстрактного класса у меня может быть какой-то необходимый метод
Абстрактный класс — это класс, у которого не реализован один или больше методов (некоторые языки требуют такие методы помечать специальными ключевыми словами).
Интерфейс — это абстрактный класс, у которого ни один метод не реализован, все они публичные и нет переменных класса.
Интерфейс нужен обычно когда описывается только интерфейс (тавтология). Например, один класс хочет дать другому возможность доступа к некоторым своим методам, но не хочет себя «раскрывать». Поэтому он просто реализует интерфейс.
Абстрактный класс нужен, когда нужно семейство классов, у которых есть много общего. Конечно, можно применить и интерфейс, но тогда нужно будет писать много идентичного кода.
В некоторых языках (C++) специального ключевого слова для обозначения интерфейсов нет.

Можно считать, что любой интерфейс — это уже абстрактный класс, но не наоборот. Виды методов проектирования: итератор, одиночка, заместитель, компоновщик ,мост , наблюдатель ,посетитель ,фабричный метод, а также адаптер и декоратор

1. Отличие интерфейсного и виртуального методов

Виртуальный метод не обязательно переопределять, в то время как каждый класс, реализующий интерфейс, должен реализовать все методы этого интерфейса. В этом собственно и цель интерфейса, чтобы переопределять методы.

У виртуальных и переопределенных методов должны быть одинаковые модификаторы доступа. В интерфейсах модификатор доступа можно не писать, он будет public по умолчанию.

Если сделать метод интерфейса приватным, то мы будем обязаны написать для него реализацию по умолчанию.

Вызов методов:

Для виртуального метода обращение к данным объекта идет через объектную переменную, а для интерфейсного – через интерфейсную. Потом из данных объекта извлекается таблица виртуальных методов, либо таблица методов интерфейса. Потом на основании порядкового номера метода (рассчитывается его смещение в таблице) извлекается адрес метода (этот пункт одинаковый и для того и для другого). После этого выполняется вызов метода. После этого вызов виртуального метода завершен, а для интерфейсного метода нужно еще корректировать аргумент this на вершине стека (привести к ожидаемому значению), а после этого сделать прямой переход к коду метода.

2. Почему в конструкторе/деструкторе не стоит вызывать виртуальные методы

Из конструктора в C# можно вызывать виртуальные методы. С одной стороны, это здорово. То есть вроде бы можно заставить конструктор предка работать по-разному, в зависимости от конкретизации в потомке. Это бывает очень полезно, ведь конструктор формы, например, может запрашивать какое-то виртуальное свойство и формировать на его основе вид формы. Форме-потомку будет достаточно перегрузить это самое свойство и форма соответственно изменится.

Но дальше мы упираемся в неприятное ограничение языка. А именно — базовый конструктор вызывается **всегда** до того, как выполнится конструктор потомка (что означает, что виртуальная таблица (содержащая адреса переопределенных виртуальных функций производного класса) еще не существует). То есть если наше пресловутое виртуальное свойство в форме-потомке зависит от параметра, переданного в конструкторе потомка, то система работать не будет — конструктор предка всегда вызывается до того, как какие-то параметры конструктора-потомка попадут в поля объекта.

Несуразность этого ограничения подчеркивается тем, что, как ни странно, инициализация полей объекта-потомка, для которых задано начальное значение (private int newField = 123 производится **до** вызова базового конструктора).

Когда конструируется объект(в нем нужно проинициализировать все поля+указатель на таблицу виртуальных методов(этого поля инициализацию делает конструктор)),мы вызываем виртуальный метод(вызовется так как будто он не виртуальный).В других языках сделано так:при конструирование объекта класса C в него компилятор кладет метод что нужно проинициализировать поле и конструктор класса C это поле инициализирует и потом конструктор класса B понимает что трогать это не надо и говорит конструкторов класса A что трогать это не надо.(признак в регистре (нужно инициализировать или нет))

3. «Посетитель»

{тезисно, для примера откройте док дениса с ответами}

Посетитель — поведенческий шаблон, описывающий действия, выполняемые с каждым объектом в некоторой структуре.

Используется для описания операций, которые выполняются с каждым объектом из некоторой структуры. Позволяет определить новую операцию без изменения классов этих объектов.

Пример:

- в структуре присутствуют объекты многих классов с различными интерфейсами и нам необходимо выполнить над ними операции, которые зависят от конкретных классов;

- необходимо выполнять несвязанные между собой операции над объектами, которые входят в состав структуры и мы не хотим добавлять эти операции в классы;

- классы, которые устанавливают структуру объектов редко изменяются, но часто добавляются новые операции над этой структурой.

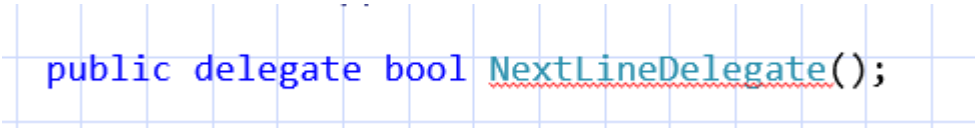
Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого вы будете передавать эти объекты в методы посетителя.

Код поведения, скорее всего, должен отличаться для объектов разных классов, поэтому и методов у посетителя должно быть несколько. Названия и принцип действия этих методов будет схож, но основное отличие будет в типе принимаемого в параметрах объекта.

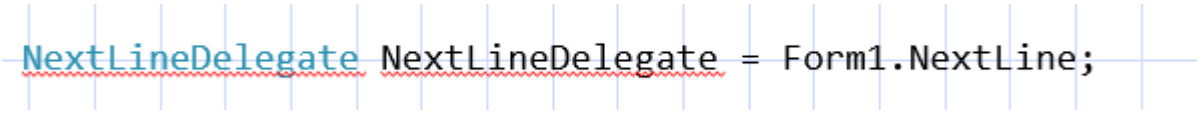
4. Физическое представление делегата

Делегат – ссылка на метод

Можно создать процедурную переменную, содержащую адрес метода. Предварительно для такой переменной определяется тип данных, называемый делегатом:



Теперь можно объявить переменную с таким типом и присвоить ей ссылку на метод какого-то объекта (сигнатура метода должна совпадать с сигнатурой делегата):



Наконец, можно сделать вызов через процедурную переменную
NextLineDelegate();
В результате этого оператора у объекта Form1 будет вызван метод bool NextLineNotification(). Это произойдет благодаря тому, что в переменной-делегате (NextLineDelegate) хранится пара указателей: указатель на объект и указатель на код метода. Когда делегату присваивается значение (NextLineDelegate = Form1.NextLine), в нем устанавливаются сразу оба указателя.

5. Множественное наследование C++

В Шарпах нету(потому что мешанина, может при наследовании повторяться методы гет которые возвращают разные значения и с этим нужно еще переопределять и работать)

При наследовании базовые классы и классы-потомки помещаются в памяти непосредственно друг за другом. И поэтому объект производного класса представляет собой последовательность объектов родительских классов.

2.11. Множественное наследование

В C++ множественное наследование подразумевает, что у одного класса может быть несколько базовых классов:

```
class TDelimitedReader : public TTextReader, public TStringList
{
  ...
};
```

Объект класса TDelimitedReader содержит все поля и методы базовых классов TTextReader и TStringList. При этом в классе TDelimitedReader можно переопределять виртуальные методы каждого базового класса.

Множественное наследование имеет ряд проблем:

- отсутствие эффективной реализации (неэффективность скрыта от программиста);
- неоднозначность, возникающая из-за того, что в базовых классах могут быть одноименные поля, а также методы с одинаковой сигнатурой;
- повторяющийся базовый класс в иерархии классов.

Неоднозначность при множественном наследовании:

```
class TTextReader
{
  virtual void NextLine();
  ...
};

class TStringList
{
  public:
    virtual void NextLine();
  ...
};

class TDelimitedReader: public TTextReader, public TStringList
{
  ...
};

TDelimitedReader*Reader;
Reader->NextLine(); // Ошибка. Неоднозначность.
```

Неоднозначность возникает потому, что в классе TDelimitedReader существуют две таблицы виртуальных методов и неизвестно, к какой из них надо обращаться за методом NextLine(). Поэтому последний оператор должен быть скорректирован на следующий:

```
Reader->TTextReader::NextLine();
```

или:

```
Reader->TStringList::NextLine();
```

В C++ для классов поддерживается столько таблиц виртуальных методов, сколько у него базовых классов. При перекрытии общего виртуального метода, существующего в нескольких базовых классах, происходит замещение адреса во всех таблицах виртуальных методов.

Перегрузка функций по типам аргументов не приводит к разрешению неоднозначности.

Если функция NextLine() была объявлена с различной сигнатурой в различных классах, то неоднозначность тоже остается.

В некоторых случаях наличие в базовых классах функций с одинаковыми именами (но различным количеством параметров или различными типами параметров) является преднамеренным решением. Чтобы в производном классе открыть нужную функцию нужного базового класса, применяется оператор **using**:

```
class TTextReader
{
  public:
    virtual void NextLine();
  ...
};

class TStringList
{
  public:
    virtual void NextLine(int);
  ...
};

class TDelimitedReader : public TTextReader, public TStringList
{
  public:
    using TStringList::NextLine;
    virtual void NextLine(int);
  ...
};
```

Таким образом, множественное наследование таит следующую проблему: заранее неизвестно от каких классов программист захочет унаследовать свой класс. Однако при создании класса использовать виртуальное наследование неэффективно, если наследуются поля, так как доступ к полям всегда будет осуществляться через дополнительный указатель.

Вывод: одинарное наследование в стиле Java, C++, Delphi допустимо только от классов, множественное — от интерфейсов. Иначе можно осуществлять множественное наследование лишь от классов, в которых отсутствуют поля.

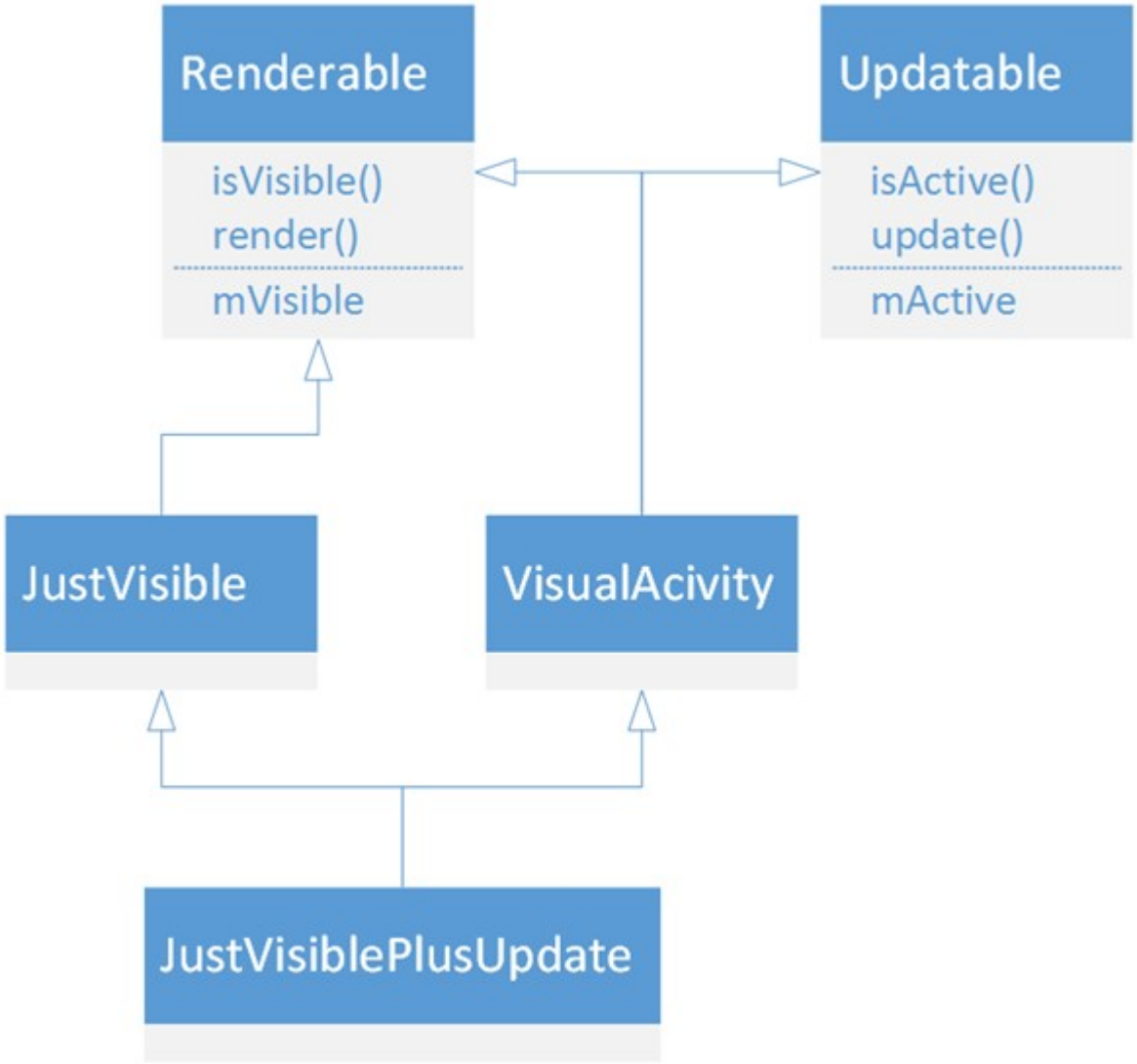
6. Виртуальное наследование C++

Википедия:

Виртуа́льное насле́дование в языке программирования C++ — один из вариантов наследования, который нужен для решения некоторых проблем, порождаемых наличием возможности множественного наследования, путём разрешения неоднозначности того, методы которого из суперклассов необходимо использовать.

Хабр:

Предположим, есть у нас проектик с такой иерархией:



1. **Renderable**: содержит признак видимости и метод рисования
2. **Updatable**: содержит признак активности и метод обновления состояния
3. **VisualActivity** = Renderable + Updatable
4. **JustVisible**: просто видимый объект

5. **JustVisiblePlusVisualActivity**: JustVisible с обновляемым состоянием

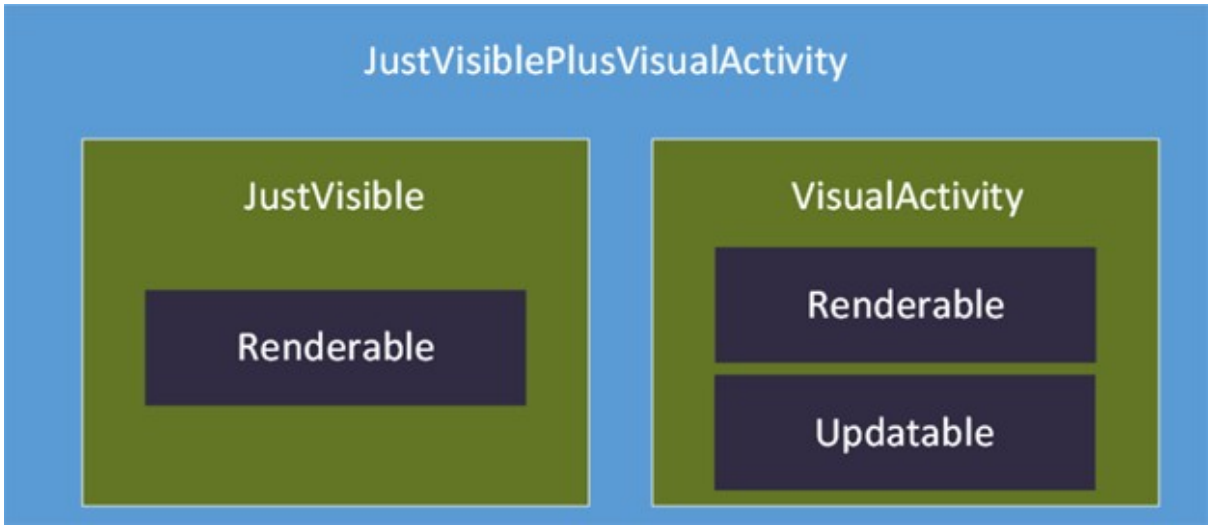
Сразу же видна проблема — конечный класс наследует Renderable дважды: как родитель JustVisible и VisualActivity. Это не дает нормально работать со списками отображаемых объектов.

```
JustVisiblePlusUpdate* object = new JustVisiblePlusUpdate;
std::vector<Renderable*> vector_visible;
vector_visible.push_back(object);
```

Получается неоднозначность (ambiguous conversions) — компилятор не может понять, об унаследованном по какой ветке Renderable идет речь. Ему можно помочь, уточнив направление путем явного приведения типа к одному из промежуточных.

```
JustVisiblePlusUpdate* object = new JustVisiblePlusUpdate;
std::vector<Renderable*> vector_visible;
vector_visible.push_back(static_cast<VisualActivity*>(object));
```

Компиляция пройдет успешно, только вот ошибка останется. В нашем случае требовался один и тот же Renderable вне зависимости от того, каким образом он был унаследован. Дело в том, что в случае обычного наследования в классе-потомке (JustVisiblePlusVisualActivity) содержится отдельный экземпляр родительского класса для каждой ветки.



Причем свойства каждого из них можно менять независимо. Выражаясь на c++, истинно выражение.

```
(&static_cast<VisualActivity*>(object)->mVisible) != (&static_cast<JustVisible*>(ob
ject)->mVisible)
```

Так что обычное множественное наследование для задачи не подходило. А вот виртуальное выглядело той самой серебряной пулей, которая была нужна… Все что требовалось — унаследовать базовые классы **Renderable** и **Updatable** виртуально, а остальные — обычным образом:

```
class VisualActivity : public virtual Updatable, public virtual Renderable
...
class JustVisible : public virtual Renderable
...
class JustVisiblePlusUpdate : public JustVisible, public VisualActivity
```

Все унаследованные виртуально классы представлены в потомке только один раз. И все бы работало, если бы базовые классы не имели конструкторов с параметрами. Но такие конструкторы существовали, и случился сюрприз. Каждый виртуально наследуемый класс имел как конструктор по умолчанию так и параметризованный. Классы-потомки содержали только конструкторы с параметрами.

И все равно при создании объекта вызывался конструктор Renderable по умолчанию. Породило проблему допущение, что Renderable чудесным образом разделится между JustVisible, VisualActivity и JustVisiblePlusUpdate. Но «чуду» не суждено было случиться. Ведь тогда можно было бы написать что-то типа

```
class JustVisiblePlusUpdate : public JustVisible, public VisualActivity
{
public:
        JustVisiblePlusUpdate(bool active)
                : JustVisible(true)
                , VisualActivity(false, active)
        {
        }
    //....
};
```

Это объясняет, почему вызывался конструктор по умолчанию — конструкторы виртуальных классов должны вызываться конечными наследниками, т.е. рабочим вариантом было бы что-то типа.

При виртуальном наследовании приходится, кроме конструкторов непосредственных родителей, явно вызывать конструкторы всех виртуально унаследованных классов. Это не очень очевидно и с легкостью может быть упущено в нетривиальном проекте. Так что лишний раз подтвердилась истина: не больше одного открытого наследования для каждого класса. Оно того не стоит.

7. Реализация интерфейсных вызовов в COM и в C#

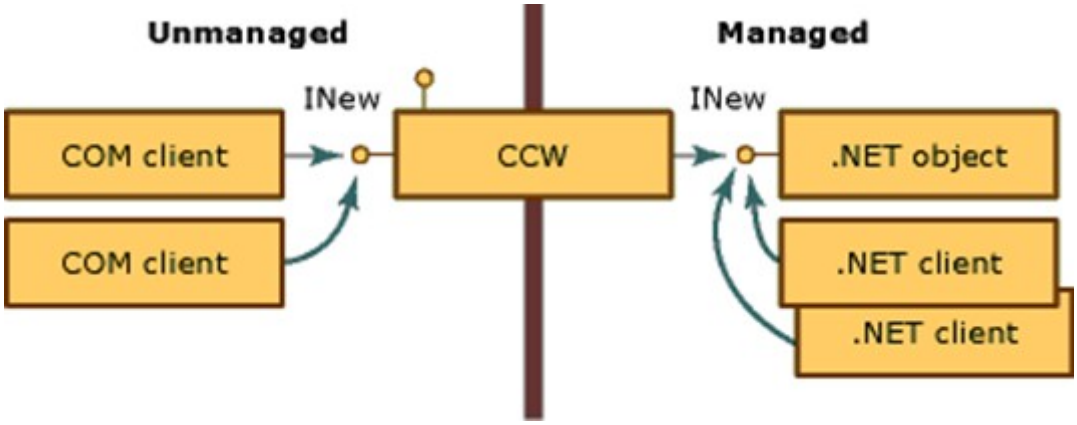
COM (Component Object Model) - это метод разработки программных компонентов, небольших двоичных исполняемых файлов, которые предоставляют необходимые сервисы приложениям, операционным системам и другим компонентам. Другими словами, COM определяет стандартный механизм, с помощью которого одна часть программного обеспечения предоставляет свои сервисы другой независимо от способа их реализации.

Там разница между COM механизмом и .Net механизмом вызова интерфейсных методов. Первый вызывается с помощью специальной заглушки(схема есть в презе), а второй вызывается с помощью нескольких таблиц интерфейсных методов (под каждый интерфейс)

Вызываемая оболочка COM:

Когда клиент COM вызывает объект .NET, среда CLR создает для этого объекта управляемый объект и вызываемую оболочку COM. Не имея возможности обращаться к объекту .NET напрямую, клиенты COM используют вызываемую оболочку COM в качестве посредника для управляемого объекта.

Среда выполнения создает одну вызываемую оболочку COM для управляемого объекта независимо от числа клиентов COM, которым требуются его службы. Как показано на рисунке ниже, несколько клиентов COM могут содержать ссылку на вызываемую оболочку COM, предоставляющую интерфейс INew. Вызываемая оболочка COM, в свою очередь, содержит единственную ссылку на управляемый объект, который реализует интерфейс и обрабатывается сборщиком мусора. Клиенты COM и .NET могут одновременно выполнять запросы к одному и тому же управляемому объекту.



Вызываемые оболочки COM невидимы для других классов, работающих в среде выполнения .NET. Их основной целью является маршалинг вызовов между управляемым и неуправляемым кодом. Однако вызываемые оболочки COM также управляют идентификацией и временем жизни управляемых объектов, которые в них упакованы.

Идентификация объектов:

Для объекта .NET среда выполнения выделяет память в куче, обработанной сборщиком мусора, что позволяет при необходимости перемещать объект в памяти. Для вызываемой же оболочки COM среда выполнения выделяет память из кучи, не обработанной сборщиком мусора, благодаря чему клиенты COM могут напрямую обращаться к оболочке.

Время жизни объекта:

В отличие от клиента .NET, учет ссылок для вызываемой оболочки COM, в которую инкапсулирован клиент, ведется обычным для модели COM образом. Когда счетчик ссылок на вызываемую оболочку COM достигает нуля, оболочка освобождает свою ссылку на управляемый объект. Управляемый объект, на который не осталось ссылок, обрабатывается сборщиком мусора в течение следующего цикла.

8. Зачем нужна функция-заглушка

Функция-заглушка — функция, не выполняющая никакого осмысленного действия, возвращающая пустой результат или входные данные в неизменном виде. То же самое, что **заглушка метода**.

Используется:

- Для наглядности при проектировании структуры классов приложения.
- Часть функций может быть «заглушена» для отладки других функций.
- · Для ограничения доступа к некоторым полям класса (например, к корню дерева).

Пример функции-заглушки на языке C:

```
void stub()

{

	return;

}
```

9. Отличие метода от процедуры

Метод – процедура над объектом. Во всех методах неявно передается параметр this с ссылкой на объект вызвавший метод.

Метод — это функция или процедура, принадлежащая какому-то классу или объекту.

Процедура — поименованная или иным образом идентифицированная часть компьютерной программы, содержащая описание определённого набора действий.

Как и процедура в процедурном программировании, метод состоит из некоторого количества операторов для выполнения какого-то действия и имеет набор входных аргументов.

Различают простые методы и статические методы (методы класса):

- простые методы имеют доступ к данным объекта (конкретного экземпляра данного класса),
- статические методы не имеют доступа к данным объекта, и для их использования не нужно создавать экземпляры (данного класса).

Методы предоставляют интерфейс, при помощи которого осуществляется доступ к данным объекта некоторого класса, тем самым, обеспечивая инкапсуляцию данных.

В зависимости от того, какой уровень доступа предоставляет тот или иной метод, выделяют:

- открытый (public) интерфейс — общий интерфейс для всех пользователей данного класса;
- защищённый (protected) интерфейс — внутренний интерфейс для всех наследников данного класса;
- закрытый (private) интерфейс — интерфейс, доступный только изнутри данного класса.

10. Делегат

Делегат – ссылка на метод.

Можно создать процедурную переменную, содержащую адрес метода. Предварительно для такой переменной определяется тип данных, называемый делегатом:

```
public delegate bool NextLineDelegate();
```

Теперь можно объявить переменную с таким типом и присвоить ей ссылку на метод какого-то объекта (сигнатура метода должна совпадать с сигнатурой делегата):

```
NextLineDelegate NextLineDelegate = Form1.NextLine;
```

Наконец, можно сделать вызов через процедурную переменную:

```
NextLineDelegate();
```

В результате этого оператора у объекта Form1 будет вызван метод bool NextLineNotification(). Это произойдет благодаря тому, что в переменной-делегате (NextLineDelegate) хранится пара указателей: указатель на объект и

указатель на код метода. Когда делегату присваивается значение (NextLineDelegate = Form1.NextLine), в нем устанавливаются сразу оба указателя.

Делегаты можно объединять в другие делегаты.

```
delegate void Message();
 Message mes1 = Hello;
 Message mes2 = HowAreYou;
 Message mes3 = mes1 + mes2; // объединяем делегаты
   mes3(); // вызываются все методы из mes1 и mes2
```

Также делегаты могут быть параметрами методов

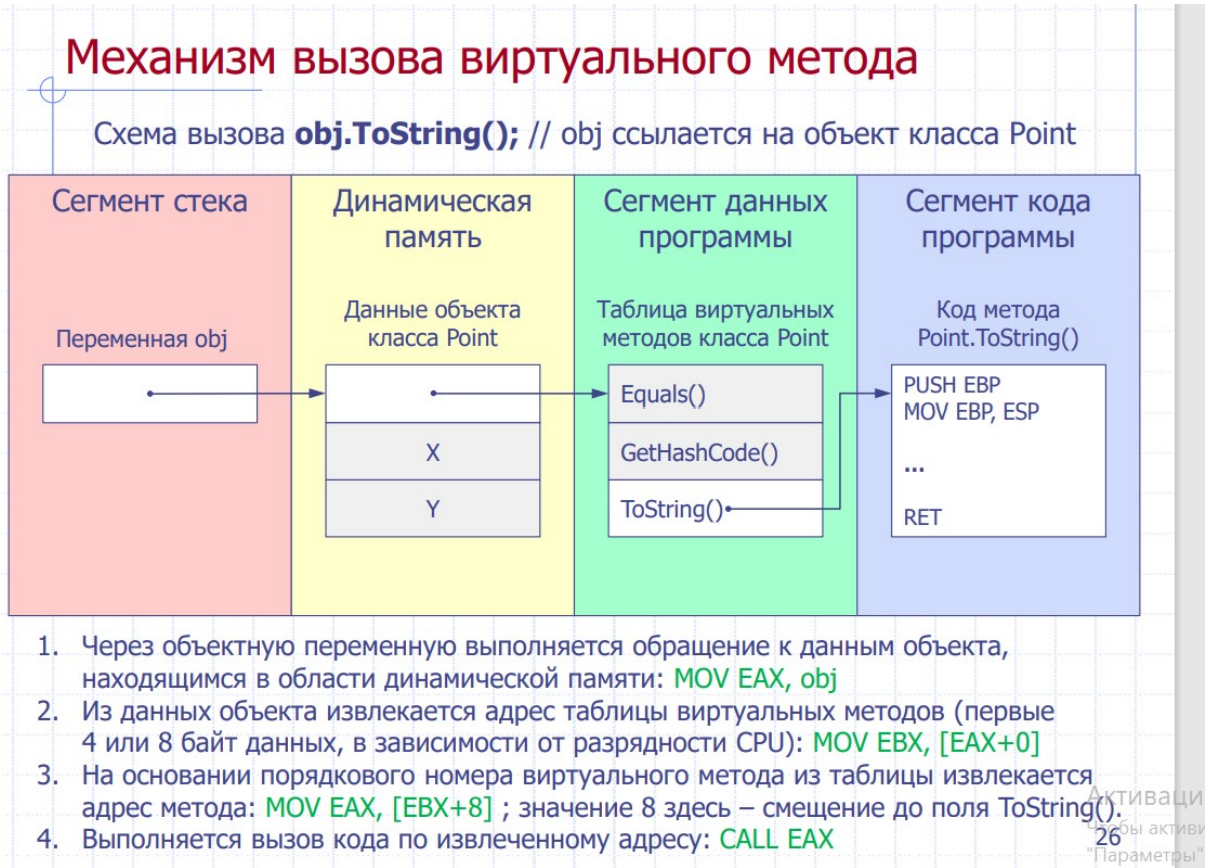
Делегаты могут быть обобщенными, например:

```
delegate T Operation<T, K>(K val);
class Program
{
    static void Main(string[] args)
    {
        Operation<decimal, int> op = Square;

        Console.WriteLine(op(5));
        Console.Read();
    }

    static decimal Square(int n)
    {
```

```
        return n * n;
    }
}
```

11. Отличие виртуального и обычного методов

- Виртуальный метод - переопределяемый обычный метод в классах потомках(обычный метод мы переопределить не можем), работают идентично
- Виртуальные методы используют другой механизм вызова(?)
- Обычный метод вызывается через функцию CALL в Assembler, виртуальный метод так: //Important будет спрашивать!!



//Сюда же будет спрашивать:
Что является идентификатором виртуального метода? Номер. Кто номер выдает? Компилятор. А адрес метода вычисляется динамически(см выше)
Что является идентификатором обычного метода? Адрес.

12. Что такое COM?

COM – интерфейсы = CC+ хуйня

(Component Object Model - COM)

Одним из главных преимуществ разработки с помощью объектно-ориентированных языков, таких как C++ и Java, является возможность эффективной инкапсуляции внутренних функций и данных. Это осуществимо именно благодаря объектной ориентированности этих языков. В объекте скрыты способы его реализации, а "наружу" предоставляется только хорошо определенный интерфейс, позволяющий внешним клиентам эффективно использовать функциональные возможности объекта. Технология COM обеспечивает эти возможности также с помощью определения стандартных способов реализации и предоставления интерфейсов COM-объекта.

Далее мы приведем определение класса для простого компонента, который будет построен в этой главе. Начнем с обыкновенного C++-класса, а затем преобразуем его в COM-объект. Особой необходимости строить COM-объекты с помощью C++ нет, но, как будет видно из дальнейшего, некоторые технические приемы C++ находят применение и при создании COM-компонентов.

```
class Math
{
// описание интерфейса
public:
   long Add( long Op1, long Op2 );
   long Subtract( long Op1, long Op2 );
   long Multiply( long Op1, long Op2 );
   long Divide( long Op1, long Op2 );

private:
   // реализация
   string m_strVersion;
   string get_Version( );
};

long Math::Add( long Op1, long Op2 )
{
   return Op1 + Op2;
}

long Math::Subtract( long Op1, long Op2 )
{
   return Op1 - Op2;
}
long Math::Multiply( long Op1, long Op2 ) {
   return Op1 * Op2;
}
long Math::Divide( long Op1, long Op2 )
{
   return Op1 / Op2;
}
```

Этот класс поддерживает выполнение основных математических операций. Вы передаете объекту два числа, а он будет прибавлять, вычитать либо делить их и возвращать результат. Нашей задачей в данной главе будет превращение указанного класса в COM-объект, который можно будет использовать в любом языке программирования, поддерживающем COM-интерфейс. Первым шагом будет определение для компонента интерфейса с помощью абстрактного класса такого вида:

```
class IMath
{
public:
   virtual long Add(long Op1, long Op2) = 0;
   virtual long Subtract(long Op1, long Op2) = 0;
   virtual long Multiply(long Op1, long Op2) = 0;
   virtual long Divide (long Op1, long Op2) = 0;
};
```

Затем мы создадим производный класс и опишем его методы точно так же, как это делалось раньше:

```
class Math : public IMath
{
public:
   long Add(long Op1, long Op2);
   long Subtract(long Op1, long Op2);
   long Multiply(long Op1, long Op2);
   long Divide (long Op1, long Op2);
};
```

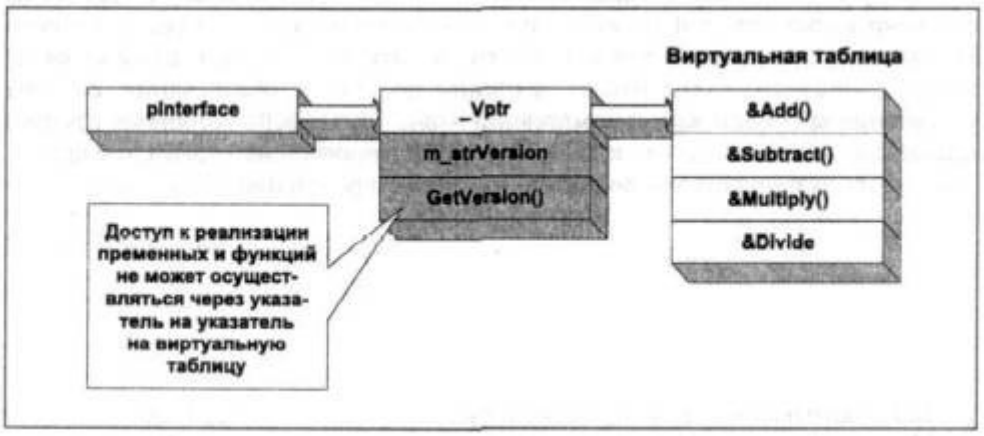
Это лишь первый шаг в преобразовании класса C++ в объект, доступный для программ, написанных на других языках. COM-технологии, такие как OLE и ActiveX, в основном реализуются посредством интерфейсов типа рассмотренного нами ранее класса IMath. Наш новый класс является абстрактным, т.е. он содержит, по крайней мере, одну чисто виртуальную функцию и одни лишь методы (без каких-либо данных) компонентного объекта.

Начальная буква "I" в названии класса IMath отражает тот факт, что он является объявлением интерфейса. Такие обозначения используются в технологии программирования COM повсеместно. Класс IMath объявляет внешний интерфейс для компонента Math. Наиболее важным аспектом этого нового класса является его способность порождать виртуальную таблицу функций C++ (Vtable) в любом производном классе.

Использование виртуальных функций в базовом классе является центральным моментом в проектировании COM-компонентов. Определение абстрактного класса порождает таблицу, содержащую только открытые методы (т.е. интерфейс) класса. Класс IMath не содержит переменных-членов и функций реализации объекта. Его единственной задачей является порождение производного класса Math для виртуальной реализации методов интерфейса компонента.

В технологии COM доступ к компонентам обеспечивается только с помощью указателей на виртуальные таблицы. Таким образом, прямой доступ к конкретным данным компонента становится невозможным. Внимательно изучите наш пример. Он достаточно прост, но отражает ключевую концепцию COM — использование виртуальных таблиц для доступа к функциональным возможностям компонента. Наконец, COM-интерфейс представляет собой просто указатель на указатель виртуальной таблицы (virtual table, или Vtable) C++. На рис. 1 отражено это взаимоотношение для случая компонента Math.

В нашем примере имеется несколько моментов, которые должны быть хорошо поняты. Во-первых, все COM-технологии, такие как ActiveX и OLE, содержат множество определений абстрактного интерфейса типа нашего класса IMath. В конечном счете, деятельность разработчика состоит в конкретной реализации этих интерфейсов. Это одна из причин того, почему ActiveX является стандартом. Технология ActiveX обеспечивает объявление интерфейса, а вы, разработчик, обеспечиваете его реализацию. Таким образом, несколько разработчиков могут по-разному реализовать стандартный компонент ActiveX. Эта концепция лежит в основе элементов управления и всех технологий ActiveX. Спецификация ActiveX определяет абстрактные классы, которые требуется реализовать для получения полноценного элемента управления ActiveX.



Во-вторых, Microsoft движется в направлении от создания библиотек функций (например, Win32 API) к созданию компонентов операционной системы, предоставляющих определенные COM-интерфейсы. В таком случае вы становитесь пользователем компонента. Вместо функций API вы получаете интерфейс для компонента системы Windows и доступ к его функциям посредством COM-интерфейса. Это дает возможность заменять реализацию компонента операционной системы, не оказывая воздействия на другие компоненты и приложения.

В-третьих, во многих случаях вы будете одновременно выступать в качестве производителя и потребителя COM-интерфейсов. Технологии ActiveX используют их очень широко. Взаимодействие между компонентами ActiveX достигается за счет интерфейсов. Запуск и согласование приложений осуществляется через интерфейсы. Фактически, COM-интерфейсы присутствуют везде.

Теперь вы поняли, что технология COM предусматривает наличие множества абстрактных классов, которые требуют реализации. При построении COM-компонента первым делом нужно реализовать интерфейс, который должны использовать все COM-компоненты: IUnknown. Компонент должен не только реализовать интерфейс IUnknown для себя самого, но и обеспечить его реализацию для каждого своего интерфейса. Вначале это может показаться вам слишком сложным, но именно так и обстоят дела. Большинство COM-компонентов предлагают несколько интерфейсов, и запомните: **COM-интерфейс — это просто указатель на C++-интерфейс.**

13. Вызов виртуального метода

Техника вызова виртуальных методов называется ещё «динамическим (поздним) связыванием». Имеется в виду, что имя метода, использованное в программе, связывается с адресом входа конкретного метода динамически (во время исполнения программы), а не статически (во время компиляции), так как в момент компиляции, в общем случае, невозможно определить, какая из существующих реализаций метода будет вызвана.

В компилируемых языках программирования динамическое связывание выполняется обычно с использованием таблицы виртуальных методов, которая создается компилятором для каждого класса, имеющего хотя бы один виртуальный метод. В элементах таблицы находятся указатели на реализации виртуальных методов, соответствующие данному классу (если в классе-потомке добавляется новый виртуальный метод, его адрес добавляется в таблицу, если в классе-потомке создаётся новая реализация виртуального

метода - соответствующее поле в таблице заполняется адресом этой реализации). Таким образом, для адреса каждого виртуального метода в дереве наследования имеется одно, фиксированное смещение в таблице виртуальных методов. Каждый объект имеет техническое поле, которое при создании объекта инициализируется указателем на таблицу виртуальных методов своего класса. Для вызова виртуального метода из объекта берётся указатель на соответствующую таблицу виртуальных методов, а из неё, по известному фиксированному смещению, — указатель на реализацию метода, используемого для данного класса. При использовании множественного наследования ситуация несколько усложняется за счёт того, что таблица виртуальных методов становится нелинейной.

14. Зачем в паттерне «Одиночка» две проверки, какую из них можно опустить

Если речь идет о «блокировке с двойной проверкой», то суть метода, согласно ВИКИ, следующая:

1. Сначала проверяется, инициализирована ли переменная (без получения блокировки). Если она инициализирована, её значение возвращается немедленно.
2. Получение блокировки.
3. Повторно проверяется, инициализирована ли переменная, так как вполне возможно, что после первой проверки другой поток инициализировал переменную. Если она инициализирована, её значение возвращается.
4. В противном случае, переменная инициализируется и возвращается.

Пример с Wikipedia: https://refactoring.guru/ru/java-dcl-issue

```
class Foo {

      private volatile Helper helper = null;

      public Helper getHelper() {
```

```
      if (helper == null) { // 1ая проверка
            synchronized(this) {
                  if (helper == null) // 2ая проверка
                     helper = new Helper();
            }
      }
      return helper;
      }

      // и остальные члены класса…
}
```

Скорее всего отказаться можно от 1ой проверки и будет достаточно только второй проверки внутри **synchronized**.

15. Отличие делегата от события

Главное отличие event от delegate состоит в том, что event может быть запущен только в классе, в котором объявлен. Помимо этого, при наличии event компилятор создает не только соответствующее приватное поле-делегат, но еще и два открытых метода для подписки и ее отмены на события.

Из меньших отличий стоит вспомнить о том, что событие, в отличие от делегата, может быть членом интерфейса. Это потому, что делегаты всегда являются полями, тогда как интерфейсы содержать поля не могут. Кроме того, событие, в отличие от делегата, не может быть локальной переменной в методе.

Событие хранит в себе список делегатов.

16. Паттерн «Наблюдатель»

- Паттерн Observer определяет зависимость "**один-ко-многим**" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.
- Паттерн Observer инкапсулирует главный (независимый) компонент в абстракцию Subject и изменяемые (зависимые) компоненты в иерархию Observer.
- Паттерн Observer **определяет часть "View"** в модели Model-View-Controller (MVC)

Решаемая проблема:

Имеется система, состоящая из множества взаимодействующих классов. При этом взаимодействующие объекты должны находиться в согласованных состояниях. Вы хотите избежать монолитности такой системы, сделав классы слабо связанными (или повторно используемыми).

Паттерн Observer определяет объект Subject, хранящий данные (модель), а всю функциональность "представлений" делегирует слабосвязанным отдельным объектам Observer. При создании наблюдатели Observer регистрируются у объекта Subject. Когда объект Subject изменяется, он извещает об этом всех зарегистрированных наблюдателей. После этого каждый обозреватель запрашивает у объекта Subject ту часть состояния, которая необходима для отображения данных.

Такая схема позволяет динамически настраивать количество и "типы" представлений объектов.

Описанный выше протокол взаимодействия соответствует модели вытягивания (pull), когда субъект информирует наблюдателей о своем изменении, и каждый наблюдатель ответственен за "вытягивание" у Subject нужных ему данных. Существует также модель проталкивания, когда субъект Subject посылает ("проталкивает") наблюдателям детальную информацию о своем изменении.

Реализация:

1. Смоделируйте "независимую" функциональность с помощью абстракции "субъект".

2. Смоделируйте "зависимую" функциональность с помощью иерархии "наблюдатель".
3. Класс Subject связан только с базовым классом Observer.
4. Наблюдатели регистрируются у субъекта.
5. Субъект извещает всех зарегистрированных наблюдателей.
6. Наблюдатели "вытягивают" необходимую им информацию от объекта Subject.
7. Клиент настраивает количество и типы наблюдателей.

У меня есть объект который должен сообщать своим подписчикам о том что он изменился

17. Отличие делегата и приёма «Наблюдатель»

Сам по себе шаблон делегата отсутствует. Скорее всего, имеется в виду шаблон делегирования.

Шаблон делегата:

- не очень гибкий - добавление более 1 делегата невозможно (подразумевает некоторую форму «мульти-делегата», то есть шаблон наблюдателя)
- отправка сообщения обходится дешево, O (1) - такая же стоимость, как и вызов любой другой функции или метода (не требуется поиск, очередь сообщений или другая инфраструктура)
- обычно не устойчивы - ожидается присутствие делегатов, которые выполняют свою часть работы, т. е. отправитель склонен к отказу, если не известен делегат
- легко понять, легко реализовать

Шаблон наблюдателя:

- очень гибкая - добавление n> 1 наблюдателей ожидается в дизайне
- стоимость отправки сообщения зависит от количества наблюдателей, O (n), то есть n наблюдателей занимают n времени и сообщений
- как правило, отказоустойчивые - обычно от наблюдателей не ожидается никакой работы со стороны отправителя. То есть, даже если нет наблюдателя, отправитель не затронут

- может стать довольно сложным для восприятия, в частности, ожидается, что наблюдатели будут реагировать на сообщения (имеет ли значение порядок? какой наблюдатель отвечает каким образом?)

Недостаток обработчиков событий или делегатов очевиден: только один наблюдатель.

Преимущество не так очевидно: производительность. С шаблоном наблюдателя вы можете добавить много наблюдателей. Когда происходит событие, о котором необходимо уведомить наблюдателей, вам нужно будет перечислить наблюдателей и отправить каждому уведомление. Это может быстро застопорить любой наблюдаемый экземпляр, особенно если число событий, требующих уведомления, также значительно.

Какая-то статья…

Как работают делегация и наблюдатель

С делегированием делегат точно выбирает, кто будет отвечать на конкретное событие в момент создания источника потенциального события. Вы можете думать об этом слушателе как о едином *наблюдателе*. В случае паттерна Observer наблюдатель выбирает, кого он наблюдает, когда ему это нравится; таким образом, зависимости обращаются, когда дело доходит до наблюдателя против делегирования. С моделью наблюдателя думайте о газете и подписчиках как о наблюдателях. Наблюдатели контролируют, когда создаются отношения. С делегацией подумайте о работнике и работодателе. Работодатель контролирует, когда создаются отношения и кто конкретно отвечает за конкретные события. Сотрудники не могут выбирать, над какими задачами они работают ... в общем.

Некоторые утверждают, что делегирование может иметь одного наблюдателя, но я думаю, что реальная разница между ними заключается в том, как назначается обработка событий. Вы никогда не увидите регистрацию делегата на событие. Он никогда не узнает, обрабатывает ли оно событие, пока оно не произойдет, и делегат не вызовет для него открытый метод.

Преимущество делегирования

Эта модель очень жесткая, а с большинством правильных конструкций она более простая и в целом более надежная. Это заставляет вас заранее объявить обработчик события во время инициализации источника потенциального события. Если вам нужен кто-то, чтобы направлять трафик, вы назначаете директора по трафику, прежде чем открыть улицу. В случае

наблюдателя вы могли бы позволить гаишнику выбирать, когда направлять трафик в любое время, когда он или она захотят этого.

Недостаток делегирования

Недостаток этого дизайна в том, что он не гибкий. Если бы вы внедрили какой-то код для подписки на газету, газете / делегату пришлось бы точно определить, кто может читать новостные сюжеты в момент их создания. По схеме наблюдателя они могут быть зарегистрированы позже в любое время, и газете нужно будет только знать, что новый человек зарегистрировался.

Когда выбрать делегацию?

Если вам нужен один конкретный (ые) наблюдатель (и), и у вас нет причин для изменения того, кто наблюдает, тогда будет полезен жесткий дизайн схемы делегирования.

Например, вам нужен класс / объект для обработки всплывающих окон для конкретной ошибки. Существует не так много причин, почему во время выполнения вам нужно было бы переключаться между тем, кто обрабатывает конкретную ошибку, поэтому имеет смысл делегировать ошибку «Недостаточно памяти» одному объекту. Создание массива потенциальных обработчиков и последующая регистрация этих обработчиков для ошибки «Недостаточно памяти» не имеет большого смысла; это было бы примером использования модели наблюдателя в этой ситуации. Во время выполнения вы можете захотеть изменить то, какие методы вызываются или какой «делегат» вызывается для переменных событий, но выгрузка обработчика события для определенного события во время выполнения не является нормальной.

Нельзя поменять местами делегатов, как в случае с наблюдателем, это просто сложно. В реальном мире, возможно, вы хотите поменять гаишников, чтобы новый делегат обрабатывал трафик. Можно утверждать, что лучший дизайн позволил бы сделать оригинального делегата полицейским участком, а не одного полицейского, но я отвлекся ...

18. «Мост»

Мост — делегирование функциональности метода другому объекту через интерфейс, чтобы иметь возможность независимо менять реализацию интерфейса.

Паттерн Bridge разделяет абстракцию и реализацию на две отдельные иерархии классов так, что их можно изменять независимо друг от друга.

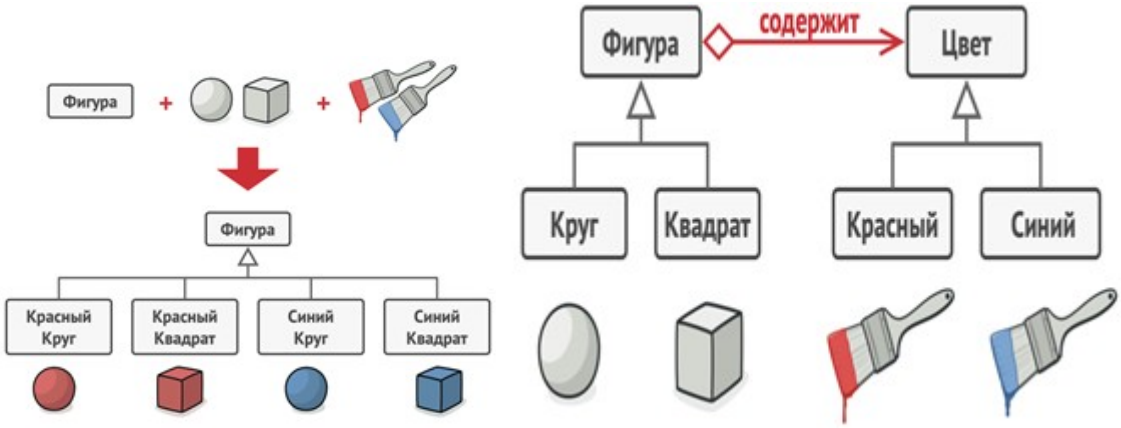
Первая иерархия определяет интерфейс абстракции, доступный пользователю. Все детали реализации, связанные с особенностями среды скрываются во второй иерархии.

Паттерн Bridge позволяет легко изменить реализацию во время выполнения программы. Применение паттерна Bridge также позволяет сократить общее число подклассов в системе, что делает ее более простой в поддержке.

Достоинства:

- Проще расширять систему новыми типами за счет сокращения общего числа родственных подклассов.
- Возможность динамического изменения реализации в процессе выполнения программы.
- Паттерн Bridge полностью скрывает реализацию от клиента. В случае модификации реализации пользовательский код не требует перекомпиляции.

- У тебя есть класс, в нем создаёшь объект типа интерфейс, через конструктор закидываешь в эту переменную значение, а после конструктора реализует методы этого интерфейса. В другом объекте



19. «Одиночка»

Назначение паттерна Singleton

Часто в системе могут существовать сущности только в единственном экземпляре, например, система ведения системного журнала сообщений или драйвер дисплея. В таких случаях необходимо уметь создавать единственный экземпляр некоторого типа, предоставлять к нему доступ извне и запрещать создание нескольких экземпляров того же типа. Паттерн Singleton предоставляет такие возможности.

Позволяет проверить существует ли уже объект, если объект существует то мы возвращает один и тот же объект при всех вызовах(transient при каждом обращение к жизненному циклу возвращает новый объект)

Описание паттерна Singleton

Архитектура паттерна Singleton основана на идее использования глобальной переменной, имеющей следующие важные свойства:

- Такая переменная доступна всегда. Время жизни глобальной переменной - от запуска программы до ее завершения.
- Предоставляет глобальный доступ, то есть, такая переменная может быть доступна из любой части программы.

Однако, использовать глобальную переменную некоторого типа непосредственно невозможно, так как существует проблема обеспечения единственности экземпляра, а именно, возможно создание нескольких переменных того же самого типа (например, стековых).

Для решения этой проблемы паттерн Singleton возлагает контроль над созданием единственного объекта на сам класс. Доступ к этому объекту осуществляется через статическую функцию-член класса, которая возвращает указатель или ссылку на него. Этот объект будет создан только при первом обращении к методу, а все последующие вызовы просто возвращают его адрес. Для обеспечения уникальности объекта, конструкторы и оператор присваивания объявляются закрытыми.

Паттерн Singleton часто называют усовершенствованной глобальной переменной.

Одиночка (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

Когда надо использовать Синглтон? Когда необходимо, чтобы для класса существовал только один экземпляр

Синглтон позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных.

Классическая реализация данного шаблона проектирования на C# выглядит следующим образом:

```
1
2   class Singleton
3   {
4
5       private static Singleton instance;
6
7
8       private Singleton()
9
10      {}
11
```

```
12
13      public static Singleton getInstance()
14      {
            if (instance == null)
                instance = new Singleton();
            return instance;
        }
    }
```

В классе определяется статическая переменная - ссылка на конкретный экземпляр данного объекта и приватный конструктор. В статическом методе getInstance() этот конструктор вызывается для создания объекта, если, конечно, объект отсутствует и равен null.

Для применения паттерна Одиночка создадим небольшую программу. Например, на каждом компьютере можно одномоментно запустить только одну операционную систему. В этом плане операционная система будет реализоваться через паттерн синглтон:

20. Виды заместителей

Заместитель- представляет суррогат или место хранения для другого объекта для контроля доступа к нему (*вообще речь шла про proxy, у заместителя нету определения потому что это группа паттернов*

***Прокси** (Proxy) или Суррогат (Surrogate) – легковесный объект заместитель, перенаправляющий вызовы к замещаемому тяжеловесному объекту.

***Обертка** (Wrapper) или Декоратор (Decorator) – объект-заместитель, содержащий в себе замещаемый объект и предоставляющий, по сравнению с ним, новые функции.

***Адаптер** (Adapter) – объект, реализующий некоторый интерфейс путем обращения к другому объекту через свойственный ему интерфейс.

Различия:
Декоратор добавляет поведение к объекту, заменяя тем самым наследование отчасти, адаптер адаптирует нужный клиенту интерфейс под тот что есть у него, не добавляя больше ничего, а прокси управляет доступом к объекту, подменяя его собой.

21. Выдача номеров в таблице методов

1. Через объектную переменную выполняется обращение к данным объекта, находящимся в области динамической памяти: MOV EAX, obj
2. Из данных объекта извлекается адрес таблицы виртуальных методов (первые 4 или 8 байт данных, в зависимости от разрядности CPU):

MOV EBX, [EAX+0]

3. На основании порядкового номера виртуального метода из таблицы извлекается адрес метода:

MOV EAX, [EBX+8]
значение 8 здесь – смещение до поля.

4. Выполняется вызов кода по извлеченному адресу: CALL EAX

номера выдаются в порядке добавления метода от 0.

22.Обертка(декоратор) и адаптер

Так, про разницу паттернов, паттерн декоратор меняет поведение динамически, делая обертку над классом, в то время как адаптер преобразует интерфейс класса в интерфейс который нужен клиенту, так как иначе интерфейсы не совместимы. То есть суть в том что первое добавляет поведение, а второе просто адаптирует уже то что есть

Обертка(тоже самое что и декоратор) – новые возможности объектов при использовании старых.

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.

Когда следует использовать декораторы?

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**: собственно декоратор, реализуется в виде абстрактного класса и имеет тот же базовый класс, что и декорируемые объекты. Поэтому базовый класс Component должен быть по возможности легким и определять только базовый интерфейс.

  Класс декоратора также хранит ссылку на декорируемый объект в виде объекта базового класса Component и реализует связь с базовым классом как через наследование, так и через отношение агрегации.

Адаптер (Adapter)

Паттерн Адаптер (Adapter) 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

23.Итератор

Формально
Абстрактный продвигаемый вперед указатель на элемент контейнера.

Понятным языком
Итератор нужен для классов которые работают с коллекциями, и так, мы разделим нашу функциональность на две части: то что мы будем перечислять, и то как мы будем это делать, создадим два generic интерфейса, один будет состоять из всего одного метода,-- возвращающего genaric интерфейс- перечисление, а само перечисление это и есть интерфейс обработки нашей коллекции(то как мы проходим по ней). Этот интерфейс будет возвращать текущее значение, говорить находимся ли мы в конце коллекции или нет, а также устанавливать следующий. Это базовый функционал чтобы пройтись в одну сторону по любой коллекции.

вот эти 2 части:
то что надо перечислить возвращает перечислитель

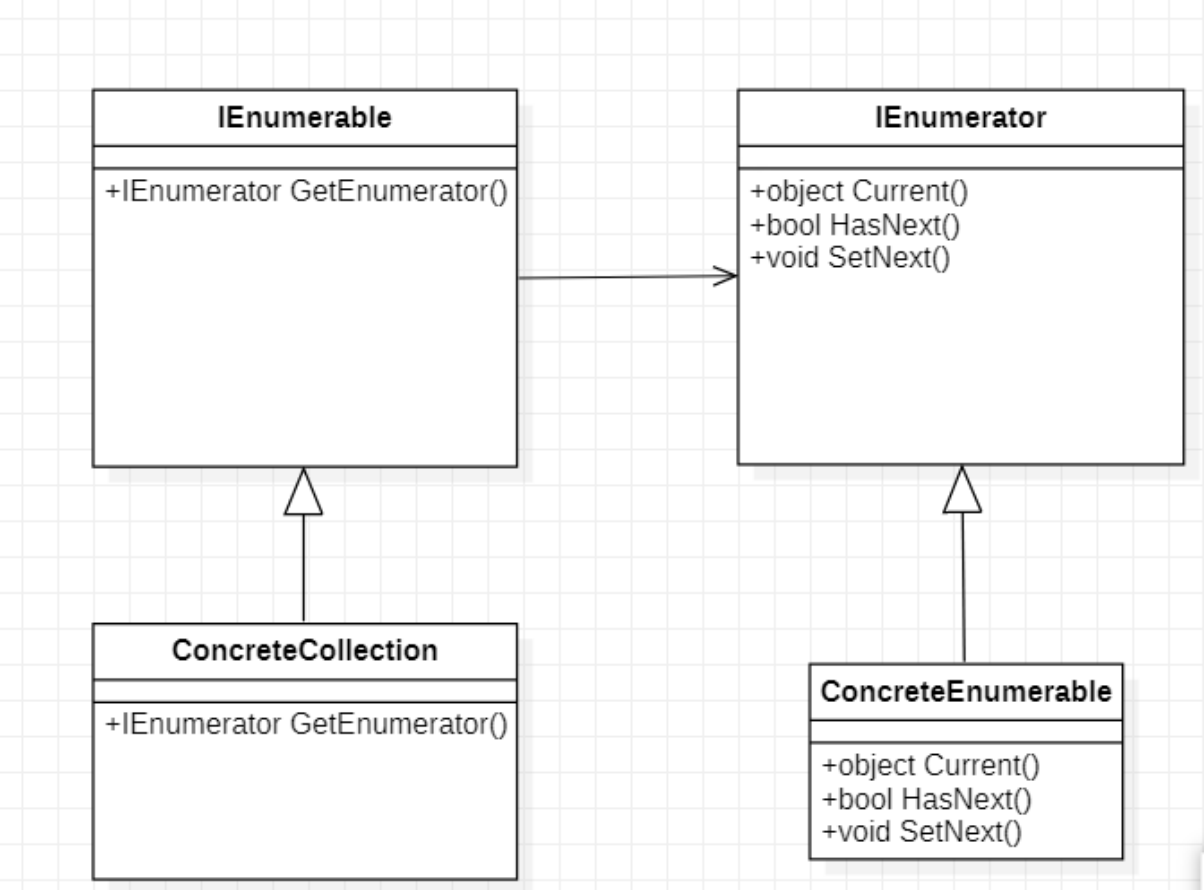
```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

и собственно сам перечислитель:

```
public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}

public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    T Current { get; }
}
```

Почему мы вообще так делаем? ну ответ лежит в принципах SOLID, конкретнее в SRP, класс не должен говорить нам как перечислять элементы своей коллекции, это вне зоны ответственности самой коллекции, вместо мы создаем новый интерфейс который отвечает за это(перечислитель).

UML:



Оператор yeld:

- Если функция возвращает значение типа IEnumerable, то такую функцию можно сделать итератором:

```
public static void Test(List<List<string>> lists)
{
    foreach (string s in PlainList(lists))
    {
        Console.WriteLine(s);
    }
}

public static IEnumerable<string> PlainList(List<List<string>> lists)
{
    foreach (List<string> list in lists)
        foreach (string s in list)
            yield return s;
    yield break; // здесь можно опустить
}
```