**UML-диаграммы классов**

Диаграммы классов показывают набор классов, интерфейсов, а также их связи. Диаграммы этого вида чаще всего используются для моделирования объектно-ориентированных систем. Они предназначены для статического представления системы.

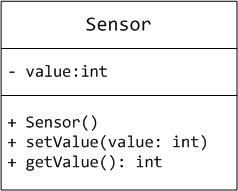
***Класс*** – это описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой.  
  
Графически класс изображается в виде прямоугольника, разделенного на 3 блока горизонтальными линиями:

* имя класса
* атрибуты (свойства) класса
* операции (методы) класса.

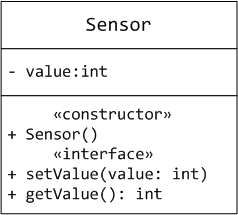
Для атрибутов и операций может быть указан один из трех типов видимости:

* **—** — private (частный)
* **#** — protected (защищенный)
* **+** — public (общий)

Видимость для полей и методов указывается в виде левого символа в строке с именем соответствующего элемента.  
  
Каждый класс должен обладать именем, отличающим его от других классов. ***Имя*** – это текстовая строка. Имя класса может состоять из любого числа букв, цифр и знаков препинания (за исключением двоеточия и точки) и может записываться в несколько строк.

На практике обычно используются краткие имена классов, взятые из словаря моделируемой системы. Каждое слово в имени класса традиционно пишут с заглавной буквы, например Sensor (Датчик) или TemperatureSensor (ДатчикТемпературы).  
  
Для абстрактного класса имя класса записывается курсивом.  
  
   
***Атрибут*** (свойство) – это именованное свойство класса, описывающее диапазон значений, которые может принимать экземпляр атрибута. Класс может иметь любое число атрибутов или не иметь ни одного. В последнем случае блок атрибутов оставляют пустым.  
Атрибут представляет некоторое свойство моделируемой сущности, которым обладают все объекты данного класса. Имя атрибута, как и имя класса, может представлять собой текст. На практике для именования атрибута используются одно или несколько коротких существительных, выражающих некое свойство класса, к которому относится атрибут.

Можно уточнить спецификацию атрибута, указав его тип, кратность (если атрибут представляет собой массив некоторых значений) и начальное значение по умолчанию.  
  
Статические атрибуты класса обозначаются подчеркиванием.  
  
***Операция*** (метод) – это реализация метода класса. Класс может иметь любое число операций либо не иметь ни одной. Часто вызов операции объекта изменяет его атрибуты.  
Графически операции представлены в нижнем блоке описания класса.  
Допускается указание только имен операций. Имя операции, как и имя класса, должно представлять собой текст. На практике для именования операции используются короткие глагольные конструкции, описывающие некое поведение класса, которому принадлежит операция. Обычно каждое слово в имени операции пишется с заглавной буквы, за исключением первого, например move (переместить) или isEmpty (проверка на пустоту).  
Можно специфицировать операцию, устанавливая ее сигнатуру, включающую имя, тип и значение по умолчанию всех параметров, а применительно к функциям – тип возвращаемого значения.  
  
Абстрактные методы класса обозначаются курсивным шрифтом.

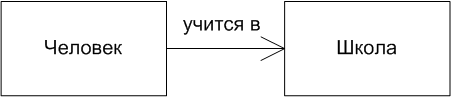
Статические методы класса обозначаются подчеркиванием.  
  
Изображая класс, не обязательно показывать сразу все его атрибуты и операции. Для конкретного представления, как правило, существенна только часть атрибутов и операций класса. В силу этих причин допускается упрощенное представление класса, то есть для графического представления выбираются только некоторые из его атрибутов. Если помимо указанных существуют другие атрибуты и операции, вы даете это понять, завершая каждый список многоточием.  
Чтобы легче воспринимать длинные списки атрибутов и операций, желательно снабдить префиксом (именем стереотипа) каждую категорию в них. В данном случае ***стереотип*** – это слово, заключенное в угловые кавычки, которое указывает то, что за ним следует.  


## Отношения между классами

Существует шесть типов связей в диаграмме классов UML:

* Ассоциация
* Агрегация
* Композиция
* Обобщение
* Реализация
* Зависимость

Эти связи представляют собой базовые строительные блоки для описания отношений в UML, используемые для разработки хорошо согласованных моделей.  
  
   
   
***Ассоциация*** – это структурная связь между элементами модели, которая описывает набор связей, существующих между объектами.

Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности таким образом, что можно перемещаться от объектов одного класса к другому.  
Например, класс **Человек** и класс **Школа** имеют ассоциацию, так как человек может учиться в школе. Ассоциации можно присвоить имя «учится в». В представлении однонаправленной ассоциации добавляется стрелка, указывающая на направление ассоциации.  


Двойные ассоциации представляются линией без стрелок на концах, соединяющей два классовых блока.  
Ассоциация может быть именованной, и тогда на концах представляющей её линии будут подписаны роли, принадлежности, индикаторы, мультипликаторы, видимости или другие свойства.  
  
***Множественность ассоциации*** представляет собой диапазон целых чисел, указывающий возможное количество связанных объектов. Он записывается в виде выражения с минимальным и максимальным значением; для их разделения используются две точки. Устанавливая множественность дальнего конца ассоциации, вы указываете, сколько объектов может существовать на дальнем конце ассоциации для каждого объекта класса, находящегося на ближнем ее конце. Количество объектов должно находиться в пределах заданного диапазона. Множественность может быть определена как единица 1, ноль или один 0..1, любое значение 0..\* или \*, один или несколько 1..\*. Можно также задавать диапазон целых значений, например 2..5, или устанавливать точное число, например 3.



***Агрегация*** – особая разновидность ассоциации, представляющая структурную связь целого с его частями. Как тип ассоциации, агрегация может быть именованной. Одно отношение агрегации не может включать более двух классов (контейнер и содержимое).  
Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём, по умолчанию агрегацией называют агрегацию по ссылке, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.

Графически агрегация представляется пустым ромбом на блоке класса «целое», и линией, идущей от этого ромба к классу «часть».  
Агрегация

***Композиция*** — более строгий вариант агрегации. Известна также как агрегация по значению.  
Композиция – это форма агрегации с четко выраженными отношениями владения и совпадением времени жизни частей и целого. Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.  
Графически представляется как и агрегация, но с закрашенным ромбиком.  
Композиция

Агрегация и композиция похожи

Схожие черты:

- Агрегация (связь целого и части, по смыслу «состоит из»)

- Композиция (связь части и целого)

Различия:

- Для агрегации обьекты части создаются отедльно от класса целого. Они существует сам по себе, так и обьект целое сам по себе, хоть он и содержит в себе. Для композиции все, что находится в контейнере, оно создается классом целое, и если класс целое ликвидируется, то класс часть тоже ликвидиурется

- Класс целое содержит в себе поле. Это поле является массивом из обьектов другого класса

- Время существования

***обобщение*** – выражает специализацию или ***наследование***, в котором специализированный элемент (потомок) строится по спецификациям обобщенного элемента (родителя). Потомок разделяет структуру и поведение родителя. Графически обобщение представлено в виде сплошной линии с пустой стрелкой, указывающей на родителя.  
Обобщение

***реализация*** – это семантическая связь между классами, когда один из них (**поставщик**) определяет соглашение, которого второй (**клиент**) обязан придерживаться. Это связи между интерфейсами и классами, которые реализуют эти интерфейсы. Это, своего рода, отношение «целое-часть». Поставщик, как правило, представлен абстрактным классом. В графическом исполнении связь реализации – это гибрид связей обобщения и зависимости: треугольник указывает на поставщика, а второй конец пунктирной линии – на клиента.  
Реализация

***зависимость*** – семантически представляет собой связь между двумя элементами модели, в которой изменение одного элемента (независимого) может привести к изменению семантики другого элемента (зависимого). Графически представлена пунктирной линией, иногда со стрелкой, направленной к той сущности, от которой зависит еще одна; может быть снабжена меткой.  
   
Зависимость  
Зависимость – это связь ***использования***, указывающая, что изменение спецификаций одной сущности может повлиять на другие сущности, которые используют ее.

## Пример кода и диаграммы классов для него

Программа получает данные с датчика температуры (вводятся с консоли) — по 5 измерений для каждого из двух объектов класса TemperatureMeasure и усредняет их. Также предусмотрен класс ShowMeasure для вывода измеренных значений.

#include <iostream>  
using namespace std;

class Sensor {  
  int value;  
public:  
  Sensor() { value = 0; }  
  void setValue(int value) { this->value += value; }  
  int getValue() { return value; }  
};

class MeasureCount  
{  
  int number;  
  static int total;  
public:  
  MeasureCount() { number = 0; }  
  void increment() { number++; total++; }  
  int getNumber() { return number; }  
  static int getTotal() { return total; }  
};

int MeasureCount::total = 0;

class ITemperatureMeasure// интерфейс  
{  
public:  
  virtual void setValue() = 0;  
  virtual double getValue() = 0;  
};

class TemperatureMeasure : public virtual ITemperatureMeasure  
{  
private:  
  Sensor \*h;        // агрегация  
  MeasureCount \*measure;  // композиция  
public:  
  TemperatureMeasure(Sensor \*h);  
  int getNumber() { return measure->getNumber(); }  
  double getValue()

{ return (double)h->getValue()/measure->getNumber(); }

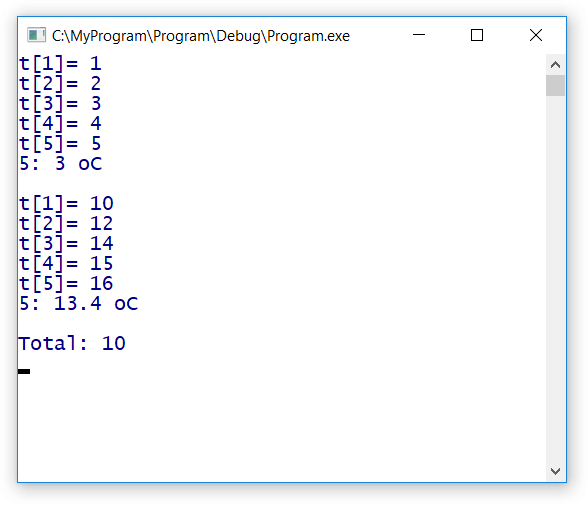
  void setValue()  
  {  
    int value;  
    measure->increment();  
    cout << "t[" << measure->getNumber() << "]= ";   
    cin >> value;  
    h->setValue(value);  
  }  
  static int getTotal() { return MeasureCount::getTotal(); }  
};

TemperatureMeasure::TemperatureMeasure(Sensor \*h)  
{  
  measure = new MeasureCount();  
  this->h = h;  
}

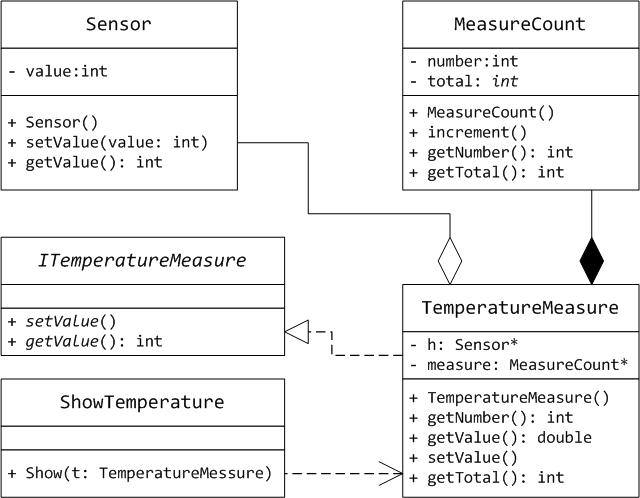
class ShowTemperature //зависимость  
{  
public:  
  static void Show(TemperatureMeasure t)  
  {  
    cout << t.getNumber() << ": ";  
    cout << t.getValue() << " oC" << endl;  
  }  
};  
  
int main()  
{  
  Sensor \*h1 = new Sensor();  
  TemperatureMeasure tc1(h1);  
  for(int i=0; i<5; i++)  
    tc1.setValue();

  ShowTemperature::Show(tc1);  
  cout << endl;  
  Sensor \*h2 = new Sensor();  
  TemperatureMeasure tc2(h2);

  for (int i = 0; i<5; i++)  
    tc2.setValue();  
  ShowTemperature::Show(tc2);  
  cout << endl;  
  cout << "Total: " << TemperatureMeasure::getTotal() << endl;  
  cin.get(); cin.get();  
  return 0;  
}

Результат выполнения  


UML-диаграмма классов для приведенного выше кода будет выглядеть следующим образом:



На диаграмме классов основным классом является класс TemperatureMeasure, который и является измерителем температуры. В качестве измеренного значения формируется среднее арифметическое всех измерений - сумма всех измерений, деленная на их количество.

Для получения измерений и их суммирования используется класс Sensor (в качестве датчика температуры). В консольной задаче сами измерения передаются в этот класс для суммирования. Класс состоит в отношении агрегации с основным классом TemperatureMeasure: мы сначала создаем объект класса Sensor, а потом передаем его в качестве параметра конструктора классу TemperatureMeasure, чтобы использовать его в качестве части класса.

Количество измерений формируется классом MeasureCount, который содержит статическое свойство total для подсчета общего измерений, а также свойство count для подсчета количества измерителей конкретного объекта TemperatureMeasure. Класс MeasureCount находится в отношении композиции с классом TemperatureMeasure: объект MeasureCount создается непосредственно при создании объекта TemperatureMeasure (в его конструкторе).  
Класс ITemperatureMeasure представляет собой интерфейс класса TemperatureMeasure и является своего рода **поставщиком** в отношении реализации.  
Наконец, класс ShowTemperature находится в отношении зависимости с классом TemperatureMeasure, поскольку реализация единственного метода Show класса ShowTemperature зависит от структуры класса TemperatureMeasure.

Замечание: На картинке класс TemperatureMeasure реализует интерфейс ITemperatureMeasure, который содержит контракт "getValue(): int". Однако класс TemperatureMeasure, имеет метод getValue, который возвращает double, хотя по контракту должен возвращать int. TemperatureMeasure нарушил контракт.

Ошибка в картинке - метод getValue() интерфейса должен возвращать double

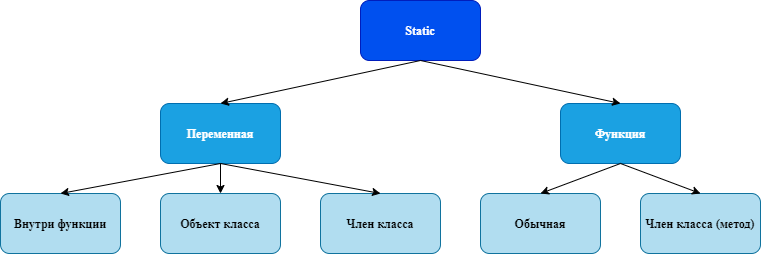
Дополнения

Про статические переменные

https://habr.com/ru/post/527044/

**Static -** это ключевое слово в C++, используемое для придания элементу особых характеристик. Для статических элементов выделение памяти происходит только один раз и существуют эти элементы до завершения программы. Хранятся все эти элементы не в heap и не на stack, а в специальных сегментах памяти, которые называются *.data* и *.bss* (зависит от того инициализированы статические данные или нет).

Ниже приведена схема, как и где используется **static** в программе.



**Статические переменные внутри функции**

Статические переменные при использовании внутри функции инициализируются только один раз, а затем они сохраняют свое значение. Эти статические переменные хранятся в статической области памяти (*.data или .bss*), а не в стеке, что позволяет хранить и использовать значение переменной на протяжении всей жизни программы. Давайте рассмотрим две почти одинаковые программы и их поведение. Отличие в них только в том, что одна использует статическую переменную, а вторая нет.

https://prog-cpp.ru/cpp-virtual/

***Виртуальные функции*** — специальный вид функций-членов класса. Виртуальная функция отличается об обычной функции тем, что для обычной функции связывание вызова функции с ее определением осуществляется на этапе компиляции. Для виртуальных функций это происходит во время выполнения программы. Это функция, которая определяется в базовом классе, а любой порожденный класс может ее переопределить. Виртуальная функция вызывается только через указатель или ссылку на базовый класс.  
   
Для объявления виртуальной функции используется ключевое слово virtual. Функция-член класса может быть объявлена как виртуальная, если

- класс, содержащий виртуальную функцию, базовый в иерархии порождения;

- реализация функции зависит от класса и будет различной в каждом порожденном классе.