**UML - Unified Modeling Language**

**Unified Modeling Language**, сокращённо **UML**, применяется на различных этапах разработки программного обеспечения (ПО). UML переводится как **унифицированный язык моделирования**.

Если посмотреть спецификацию UML, то можно заметить некоторую её избыточность. Сама спецификация занимает около 900 страниц формата А4. К счастью, для чтения UML-диаграмм нужно знать только условные обозначения, применяемые в UML.

В UML программы представляются диаграммами. В UML диаграммах представляется общая архитектура программы или какие-то её особенности. Это значит, что в UML-диаграммах создаётся только модель будущей программы. Язык UML является довольно высоким уровнем абстракции, поэтому программы, написанные на UML, впоследствии можно реализовать на любом языке, в котором есть достаточно возможностей объектно-ориентированного программирования.

Unified Modeling Language может использоваться на различных этапах разработки ПО: как во время проектирования ПО, так и во время кодирования на конкретном языке. Так как UML представлен диаграммами, то этот язык используется не только программистами, но и, например, менеджерами в компаниях, разрабатывающих ПО (но при этом нужно знать некоторые концепции ООП).

Теперь давайте уйдём от скучных определений и подумаем, а для чего нам, простым программистам, нужен этот самый UML? Представим такую ситуацию: у нас есть три класса, каждый по 300 строк кода. Между классами определены сложные связи. Уследить за кодом в данной ситуации довольно сложно. А вот если эти классы представить UML-диаграммой, то все классы и связи между ними будут видны на одной небольшой картинке (диаграмме).

Если посмотреть на спецификацию Unified Modeling Language, то может показаться, что язык очень сложный. На самом деле читать UML-диаграммы довольно легко. Главное разобраться с условными обозначениями.

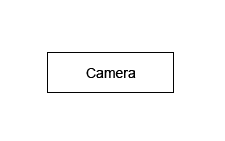
И последнее замечание прежде чем мы начнём: uml довольно новый язык, был создан в середине 90-х годов (1994-1996). На данный момент есть спецификация uml 2.2. Именно версию 2 мы будем рассматривать. Отличия между uml 1 и uml 2 нам не важны.

**Диаграммы классов UML (Class diagram)**

В UML можно создавать несколько типов диаграмм. В большинстве случаев мы будем пользоваться диаграммами классов (Class diagram). В данном типе диаграмм показывается взаимодействие классов программы.

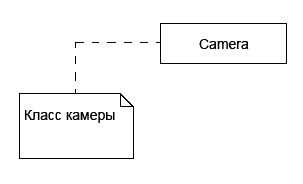
**Элементы диаграмм UML**

Диаграммы UML состоят из элементов. Элементы представляются прямоугольниками, в которых пишется имя элемента. Например, изобразим в UML-диаграмме какой-нибудь класс (для примера я взял, написанный нами ранее, класс Camera):



**Комментарии в UML**

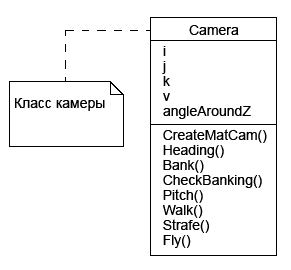
Для комментариев в UML используется прямоугольник с "загнутым" правым верхним уголком. Пунктирной линией показывается, какому элементу принадлежит комментарий:



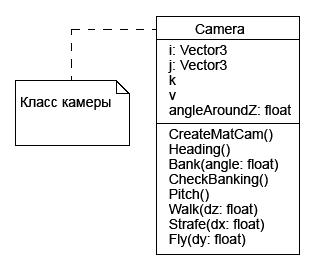
**Атрибуты (attribute) и операции (operation) в UML-диаграммах**

Если в C++ переменная, принадлежащая классу, называется полем класса или переменной-членом, то в UML такая переменная называется атрибутом. Также и с функцией/методом класса - в UML это операция.

Для атрибутов и операций в элементах отводится отдельный блок. Каждый блок разделяется горизонтальной чертой. Например, для класса Camera элемент с атрибутами и операциями будет выглядеть вот так:



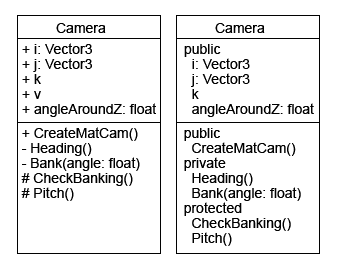
Тип атрибута (как и тип аргумента операции) задаётся через двоеточие:



Здесь можно увидеть все достоинства UML. В Unified Modeling Language необязательно расписывать **все** детали классов. Это будет сделано при написании кода на конкретном языке (в нашем случае - C++). В UML-диаграмме можно опускать ненужные детали. Например, в диаграмму элемента можно добавить только те операции/атрибуты, которые важны для данной диаграммы, неважные особенности класса в UML можно опускать.

**Видимость атрибутов и операций в UML: +, -, # (спецификаторы доступа)**

Спецификатора доступа языка C++ (public, private, protected) в UML отображаются символами + (public), - (private), # (protected), которые ставятся перед именем атрибута/операции. Также возможен вариант с ключевыми словами public, private, protected. На картинке ниже показаны оба варианта:



Напомню значение спецификаторов доступа: public - поля/методы класса видны снаружи класса. Т.е. к ним могут получать доступ объекты класса. private - поля/методы класса видны только внутри определения класса. protected - поля/методы класса видны в определении самого класса и в определениях производных классов.

**Отношения между классами в ООП (UML, С++)**

В программах между классами существуют различные виды взаимодействия (или связи): один класс может быть производным другого, третий может содержать объект четвёртого в виде поля. Для различных видов взаимодействия в UML есть специальные умные названия.

**Ассоциация/объединение/связь (association)**

Первый вид связи - association. На русский можно перевести по-разному: ассоциация, связь, объединение. На мой взгляд, наилучший вариант - связь, но я это слово использую для всех видов взаимодействия классов. Поэтому для обозначения вида связи association мы воспользуемся словом ассоциация.

Ассоциация - самый слабый вид связи. Обычно ассоциация возникает, когда один класс вызывает метод другого или если при вызове метода в качестве аргумента передаётся объект другого класса.

На диаграммах ассоциация обозначается сплошной линией.

Для примера напишем простой класс:

class MonstAr

{

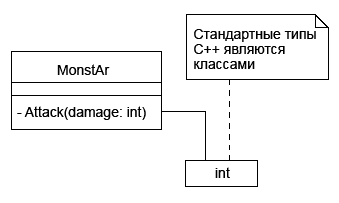
private:

attack(int damage) // damage - урон

{}

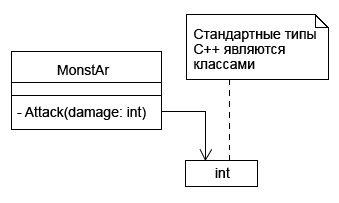
};

Напоминаю, что стандартные типы C++ являются классами. Вот как будет выглядеть взаимодействие классов MonstAr и int на диаграмме UML:



Обратите внимание на то, как в этой диаграмме показано отсутствие атрибутов у элемента.

Иногда при ассоциации показывают направленность (если это имеет значение). В спецификации UML используется слово navigable. На мой взгляд, на русском здесь нужно использовать *направленность*, так как это слово правильно отражает суть. Направленность показывается с помощью стрелочки (обратите внимание, как рисуется стрелочка, это имеет значение):



Заметьте, что стрелочка указывает на int. В данном случае направленность ассоциации говорит нам, что в методе MonstAr::Attack используется объект типа int.

**Обобщение (generalization)**

Для представления наследования в UML используется обобщение (generalization, напоминаю, что все термины берутся из спецификации UML). Пример:

MonstAr

{

private:

attack(int damage) // damage - урон

{}

};

BigMonstAr : public MonstAr // большой (big) MonstAr

{

// определение класса

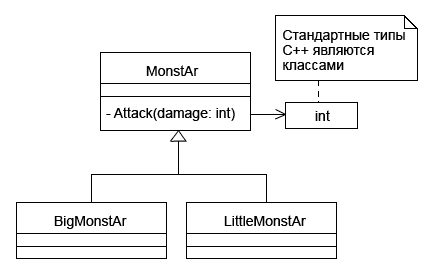
};

SmallMonstAr : public MonstAr // маленький (small) MonstAr

{

// определение класса

};



При обобщении рисуется сплошная линия. Обратите внимание как рисуется стрелочка - пустой треугольник.

Теперь насчёт слова *обобщение* (generalization). В UML используется именно оно, а не *наследование*, так как в данном виде связи один из классов (базовый) является общим, а остальные классы (производные) - более специализированными.

**Aggregation - агрегация, агрегирование, включение в UML**

Следующий тип связи между классами - aggregation (слово происходит от латинского aggregatio - присоединение). По-русски это будет агрегация, агрегирование или соединение частей. Мы будем использовать слово *агрегация*.

Итак, в UML агрегация отражает связь классов, когда объект одного класса является атрибутом другого. Пример:

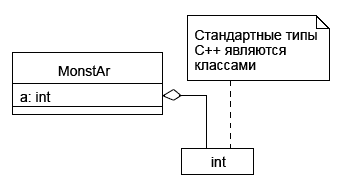
class MonstAr

{

public:

int a;

};



На диаграммах агрегация показывается незакрашенным ромбом.

**Композиция классов - composition в UML**

Композиция классов - более сильная связь между классами, чем агрегация. Между агрегацией и композицией довольно тонкая грань. Особенностью композиции является то, что объекты, из которых создаётся композиция, могут принадлежать только классу, с которым они образуют композицию. При этом время жизни объекта и класса, в который встраивается объект, совпадает.

Для начала рассмотрим два примера из жизни. Например, dvd-привод и диски, которые он читает, образуют агрегацию. Диски можно свободно менять. Примером композиции может служить хлеб и мука. Извлечь муку уже невозможно. На этих двух примерах хорошо видна разница между композицией и агрегацией: компоненты собранные агрегацией можно разъединить, а с композицией этого сделать не получится. Но вернёмся к программированию.

Одним из признаков агрегации является использование указателей. И наоборот, если при связи классов указатели не используются, то существует большая вероятность, что перед нами композиция классов.

class Claws; // claws - когти

class MonstAr

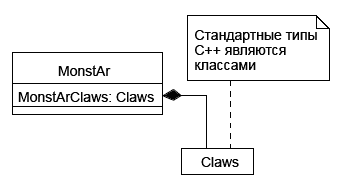
{

public:

Claws MonstArClaws;

};

В данном случае у монстра "есть когти" (определённые в отдельном классе). Возможно, пример не слишком удачный, но здесь хорошо видна композиция классов: нельзя от монстра отделить его когти (он будет сильно недоволен). В UML композиция выглядит вот так:



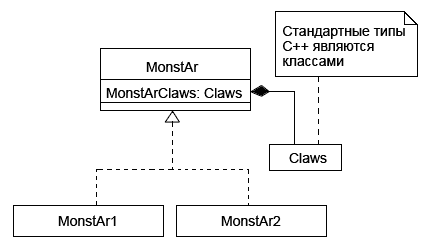
На диаграммах композиция показывается закрашенным ромбом.

Впереди у нас много примеров, в которых можно будет потренироваться в определении типа связи между классами.

**Реализация - realization в UML**

Последнее отношение, которое мы рассмотрим, будет realization - реализация. Данная связь показывает отношение: класс - объект.

На диаграмме реализация показывается пунктирной линией и незакрашенной стрелочкой:



**Заключение**

Конечно, за рамками урока осталось много важных в UML вещей. Но по крайней мере у нас теперь есть основа, которая позволит нам понимать (и рисовать) диаграммы UML. В ближайшее время я доработаю урок по конечным автоматам, где мы воспользуемся новыми знаниями.