7. Взаимодействие классов на основе наследования

Вторым важнейшим способом взаимодействия классов является **обобщение**, при котором один из классов описывает некоторое достаточно **общее** понятие, а другие – более **конкретные разновидности** этого понятия. Тем самым появляется возможность моделировать связи типа **«Общее – частное»**. Это позволяет проводить классификацию понятий.

Аналогично принципу агрегации/композиции, далеко **не все** классы можно связывать подобным образом. Вот некоторые примеры **общих** понятий и их более конкретных **разновидностей**:

1. Общее понятие: **компьютер**

    разновидности: **стационарные** компьютеры, **мобильные** компьютеры

        разновидности **мобильных** компьютеров: ноутбуки, нетбуки, планшетные компьютеры

        разновидности **стационарных:** настольный компьютер, сервер, суперкомпьютер

2. Общее понятие: **графическая фигура**

    разновидности: **окружность, отрезок, прямоугольник**

3. Общее понятие: **структура данных**

    разновидности: **список, дерево**, **хеш-таблица**

        разновидности **списка**: упорядоченный, двунаправленный

        разновидности **дерева**: двоичное, поисковое, недвоичное

4. Общее понятие: **транспортное средство (ТС)**

    разновидности ТС: **наземное, воздушное, водное**

        разновидности **наземного** ТС : автомобиль, трактор, трамвай и др.

            разновидности **автомобилей**: грузовые, легковые, спортивные

        разновидности **воздушного** ТС: самолет, вертолет, воздушный шар

5. Общее понятие: **человек**

    разновидности: **учащийся, сотрудник**, **пациент**, **клиент**

        разновидности **учащихся**: **школьник, студент**, **курсант**

На основе этих примеров, а также исходя из здравого смысла, можно сформулировать следующее правило для выяснения **возможности** установления отношения обобщения между некоторыми классами:

класс В описывает понятие, которое является частным случаем понятия, описываемого классом А

В англоязычной литературе этот тип взаимодействия кратко определяется как **“is-a”** (является, есть).

С другой стороны, можно привести множество примеров классов, между которыми данное отношение устанавливать **не следует**. Например класс **Стол** и класс **Стул** напрямую связать **нельзя,** но косвенно можно – через **общее** для них понятие **Мебель.**

В объектных языках программирования отношение обобщения реализуется с помощью механизма **наследования**, которое безусловно является одним из фундаментальных отличий объектных языков от не-объектных. Очень многие мощные возможности объектных языков (в частности, рассматриваемый далее полиморфизм) основаны на использовании наследования.

Суть механизма наследования состоит в следующем. Пусть имеется некоторый класс со своими свойствами и методами. На основе этого класса можно создать **новый** класс, связав его с исходным классом отношением наследования. Исходный класс называют **родительским** или **базовым** (parent class), а производный от него – **дочерним** или классом-наследником (child class).

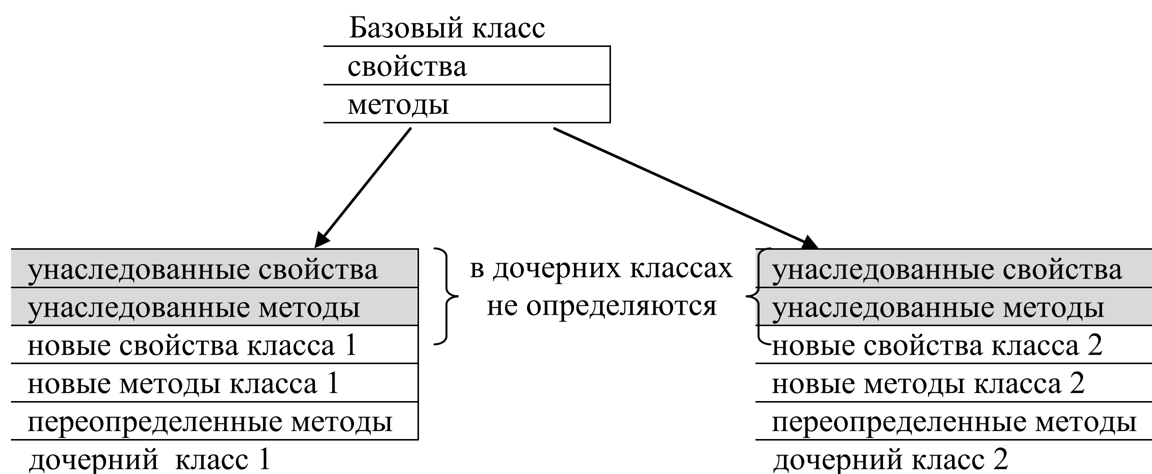
Отношение наследования между двумя классами означает,
что в дочернем классе можно использовать свойства и методы базового родительского класса БЕЗ их определения!
Эти свойства и методы просто наследуются из родительского класса.

При создании дочернего класса в него надо добавить **новые** свойства и методы, которых не было в родительском классе. Тем самым дочерний класс **расширяет, дополняет** набор свойств и методов своего родителя и именно поэтому описывает более **конкретное** понятие. Например, класс **«Человек»** может включать такие «общечеловеческие» данные как фамилия и имя, дата рождения, пол и т.д. К этим данным в классе **«Студент»** могут быть **добавлены** такие специфические данные как специальность и форма обучения, полученные оценки, шифр группы и др.

На основе одного базового класса можно создать любое **разумное** количество дочерних классов. У всех дочерних классов есть что-то **общее,** определяемое унаследованными свойствами и методами, и в то же время каждый класс **индивидуален,** так как содержит свои собственные свойства и методы.

Дополнительно в дочерних классах можно выполнить так называемое **переопределение** некоторых унаследованных методов. Этот очень мощный, но сложный для понимания механизм связан с принципом полиморфизма и рассматривается далее в разделе 11.

Схематично взаимодействие **базового** класса и созданных на его основе **производных** классов показано на следующем рисунке.



Мощь механизма наследования заключается в том, что дочерний класс, в свою очередь, может выступать в качестве базового родительского класса,

на основе которого можно создать **свои** дочерние классы. В итоге возникает **многоуровневая иерархическая структура классов**. На верхних уровнях этой иерархии находятся **наиболее общие** классы с относительно небольшим числом свойств и методов. Чем ниже класс в иерархии, тем больше он имеет свойств и методов и, тем самым, носит более **конкретный** характер. Необходимо четко понимать, что каждый нижележащий класс наследует свойства и методы от **всех** своих потомков, начиная с самого **верхнего** уровня. Это позволяет создавать мощные по своим функциям классы путем **постепенного** наращивания возможностей.

Классы верхних уровней часто объявляются **абстрактными.** Эти классы нужны для придания **общности** нижележащим подклассам.

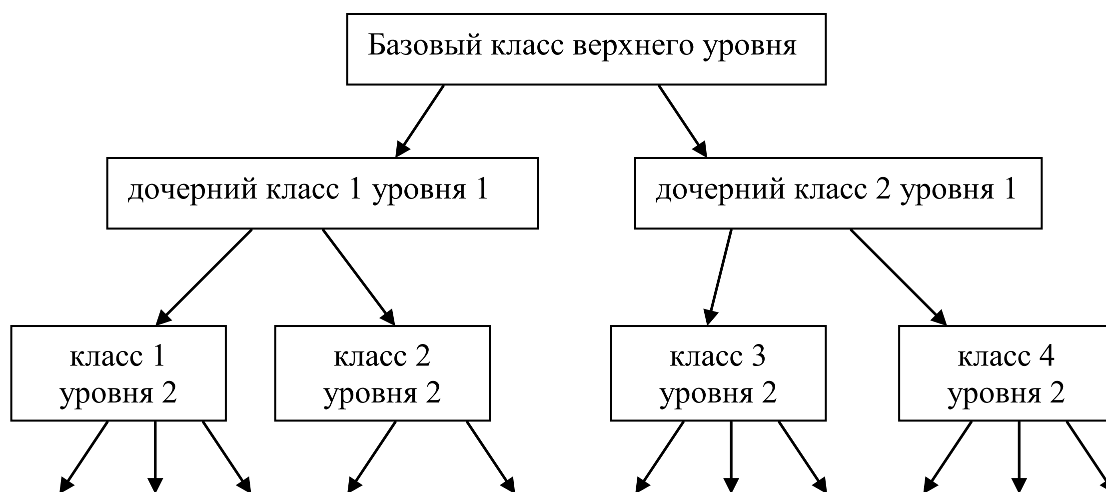
Различают два типа наследования – **простое** и **множественное**:

- **простое (единичное)**: дочерний класс может иметь **только одного** родителя
- **множественное**: родителей может быть **несколько**

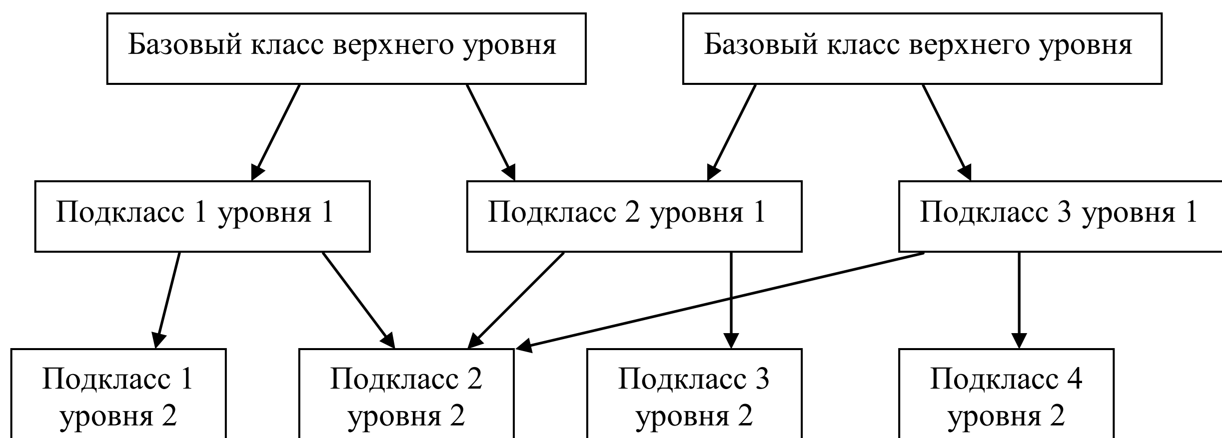
Очевидно, что множественное наследование дает **больше** возможностей при создании дочерних классов, поскольку можно наследовать свойства и методы сразу от **нескольких** классов. Первоначально множественное наследование было реализовано в языке С++, но в более поздних языках (Java, C#, Delphi/Free Pascal)) от него отказались, оставив только простое наследование. Это связано с тем, что как показала многолетняя практика использования языка С++, множественное наследование часто приводит к ряду проблем. Поэтому разработчики более поздних языков отказались от множественного наследования в чистом (как в С++) виде, но ввели некоторую частичную **замену** этого механизма на уровне так называемых **интерфейсных** классов.

Необходимо отметить, что есть современные языки, где множественное наследование реализовано в классическом виде (например - **Python**, **Eiffel**, **Perl**). В дальнейшем основное внимание будет уделено использованию механизма простого наследования.

Нетрудно догадаться, что в случае простого наследования (родитель один!) иерархия классов имеет **древовидный разветвляющийся** характер, что конечно же проще чем структура в виде ориентированного **графа**, которая возникает в случае множественного наследования.



Общесетевая иерархия классов (множественное наследование):



Все стандартные библиотеки классов, поддерживающие языки Java, платформу .NET и Delphi/FP построены на основе простого наследования и имеют древовидную структуру. В этих библиотеках на самом верхнем уровне находится **единственный корневой** класс, который является общим предком **абсолютно любых** классов, как стандартных, так и создаваемых программистом при разработке объектной программы. В этом классе

объявляются свойства и методы, наследуемые всеми классами. В языках Java и C# этот класс называют **Object**, в Delphi Pascal – **TObject**. Несмотря на общность названия, структура этих классов в разных языках различная.

В чем же состоит мощь принципа наследования? Для трех-четырех классов, связанных отношением наследования, вряд ли стоит ожидать каких-то существенных преимуществ. Мощь иерархии классов проявляется для **больших разветвленных** наборов, содержащих десятки и сотни взаимосвязанных классов. В такой иерархии можно найти один или несколько наиболее подходящих для решения конкретной задачи классов, причем если какой-то существующий класс не полностью отвечает необходимым требованиям, на его основе можно легко создать класс-потомок, добавив недостающие свойства и методы и даже изменив реализацию некоторых из унаследованных методов. В целом это существенно сокращает время разработки новой программы, позволяет максимально использовать ранее созданный и отлаженный код, повышает надежность программы и облегчает ее модернизацию.

Механизм наследования между классами в разных языках включается очень легко и практически одинаково: имя родительского класса прописывается в заголовке нового класса.

В языке Delphi Pascal:

```
TChildClass = class (TParentClass)
   private
      новые свойства;
   public
      новые методы;
 end;
```

Если родительский класс не задан, то компилятор **автоматически** в качестве родителя подставляет класс TObject. Тем самым все новые классы **включаются в общую иерархию,** что позволяет использовать ряд мощных механизмов, рассматриваемых в следующих разделах пособия.

В языке Java:

```
class   ChildClass   extends   ParentClass
// очень удачная директива extends  (расширение)
   { новые свойства;
     новые методы;
    };
```

Здесь также при отсутствии родительского класса автоматически подставляется базовый класс Object и тем самым все классы включаются в общую иерархию.

В языке C#:

```
class   ChildClass : ParentClass
   { новые свойства;
     новые методы;
    };
```

Практически также – для языка C++, только список родителей может содержать несколько классов:

```
class   ChildClass : Parent1, Parent2, Parent3     // список родителей
```

При наследовании свойств и методов немного изменяется механизм **доступа** к элементам классов. Закрытые элементы класса по-прежнему доступны для прямого использования только в методах данного класса. Важно отметить, что эти элементы **остаются закрытыми** и для дочерних классов, т.е. закрытые элементы базового класса присутствуют в дочерних классах, но доступ к ним разрешен **только** через открытые методы родителя.

Довольно часто такая закрытость является неудобной, поэтому кроме двух стандартных уровней доступа (**private** и **public**) введен **промежуточный** уровень, описываемый директивой **protected.** Элементы класса, объявленные с этой директивой, называют **защищенными,** и они доступны для **прямого** использования в любых **производных классах** и

только в них. Для всех остальных классов, не входящих в подиерархию с заданным корневым классом, эти элементы остаются закрытыми.

Большое значение при использовании наследования имеет **правильное создание** объектов дочерних классов. Поскольку дочерние классы всегда содержат больше свойств, чем их родители, то при создании дочернего объекта необходимо правильно инициализировать унаследованные свойства. За установку унаследованных свойств отвечает **конструктор родительского класса,** поэтому конструктор дочернего класса должен начинать свою работу с вызова конструктора родительского класса. Для этого используются специальные синтаксические приемы, отличающиеся для различных языков программирования.

В языке DP для вызова родительского конструктора используется директива **inherited:**

inherited Create ( ); // вызов конструктора родительского класса

В языке Java – директива **super:**

super (параметры);

В языках C# и C++ вызов родительского конструктора можно вставить в заголовок дочернего конструктора:

ChildClass (. . .) : ParentClass (. . .);

В итоге создается цепочка вложенных вызовов конструкторов. В **первую** очередь устанавливаются поля, объявленные в классе **самого верхнего** уровня. После этого управление возвращается на уровень ниже с установкой соответствующих полей и т.д., и в последнюю очередь устанавливаются уникальные поля создаваемого объекта.

В качестве **примера** использования механизма наследования рассмотрим построение небольшой иерархии классов для основных графических примитивов. Необходимо отметить, что построение иерархии классов не всегда бывает простым и очевидным и часто может быть выполнено различными способами.

На **верхний** уровень создаваемой иерархии необходимо поместить **общий класс фигур**, который целесообразно объявить **абстрактным**. Для фигуры надо ввести два свойства, которые затем будут **наследоваться** всеми производными классами. Эти свойства – координаты базовой точки – должны иметь **любые** конкретные фигуры.

В классе фигур можно ввести два **абстрактных** метода: показ фигуры (метод с именем, например, Show), перемещение базовой точки (метод с именем MoveTo). Целесообразность их введения объясняется тем, что в этом случае в базовом классе как корневом для создаваемой подиерархии определяется **общая** функциональность, которую должны обеспечивать все конкретные классы. Заголовки этих абстрактных методов можно рассматривать как **прототипы** будущих реальных методов для конкретных фигур.

Можно в классе ввести конструктор, но использовать его не для создания объектов (это запрещено для абстрактных классов), а для **инициализации** свойств-координат. Что касается методов доступа к полям данных, то здесь возможны два варианта:

- поля данных (координаты точки) объявляются закрытыми и для работы с ними как обычно вводятся методы доступа
- поля данных объявляются защищенными и тогда методы доступа можно не вводить

Сам класс фигур достаточно простой и особых комментариев не требует.

```
TFigure = class   // по умолчанию родитель – класс  TObject
   private        //  альтернативный вариант – защищенные свойства
        x, y : integer;
   public
        constructor  Create (ax, ay : integer);
        procedure  SetXY (ax, ay : integer);  // можно не включать
        function  GetX : integer;   // можно не включать
```

```
      function  GetY : integer;   // можно не включать
      procedure Show;  abstract;
      procedure  MoveTo(ax, ay : integer);  abstract;
   end;
   //  реализация неабстрактных методов
```

```
class   Figure  {              // по умолчанию родитель – класс  Object
   private   int  x, y;     //  альтернативный вариант – защищенные свойства
   public   Figure (int  ax,  int  ay) {…;}
   public   void  SetXY (int  ax,  int  ay) {…;}  // можно не включать
   public   int   GetX( ) {…;}   // можно не включать
   public   int   GetY( ) {…;}   // можно не включать
   public   abstract   void   Show( );
   public   abstract   void   MoveTo(int  ax,  int  ay);
 };
```

Первый дочерний класс – это класс окружностей со следующей структурой:

- одно закрытое или защищенное поле для радиуса окружности (а вот поля для координат центра будут наследоваться из базового класса фигур)
- конструктор для создания окружности
- методы доступа к радиусу (методы доступа к координатам будут наследоваться)
- реальные методы отображения и перемещения окружности

Прежде чем привести формальное описание класса окружностей, надо сделать одно замечание. Какие имена надо присвоить методам отображения и перемещения? На первый взгляд – любые осмысленные, типа ShowCircle, MoveCircle. Однако на самом деле следует использовать **те же имена**, что были введены в **базовом** классе, например – Show и MoveTo. Объясняется это тем, что использование **одинаковых** имен для обозначения

одинакового поведения объектов **разных** классов дает возможность включить ряд интересных и мощных механизмов, основанных на понятии полиморфизма. В последующих разделах эти вопросы будут рассмотрены более подробно.

Теперь можно привести формальное описание класса окружностей как производного от базового абстрактного класса фигур.

```
TCircle = class (TFigure)   //  включаем механизм наследования!
    private                      // а можно - protected
        r : integer;             // x, y    унаследованы и могут использоваться
    public
        constructor  Create (ax, ay, ar : integer);
        procedure  SetRad (ar : integer);  //  остальные методы  унаследованы!
        function  GetRad : integer;
        procedure  Show;   //  надо реализовать
        procedure  MoveTo (ax, ay : integer);  // и это надо
end;
```

Фрагменты программной реализации:

```
constructor  TCircle.Create (ax, ay, ar : integer);
  begin
    inherited   Create (ax, ay);      r := ar;
  end;
procedure  TCircle.MoveTo (ax, ay : integer);
  begin
    Show;
    SetXY(GetX+ax, GetY+ay);    // если координаты закрыты
   // или так:     x := x+ax;   y := y+ay;   если они  protected
    Show;
  end;
```

```
class  Circle  extends  Figure   {   //  включаем механизм наследования!
    private   int   r;                  // а можно - protected
    public    Circle (int  ax, int  ay, int  ar) { super (ax, ay);  r = ar;}
    public    void   SetRad (int   ar) {…};  //  остальные методы  унаследованы!
    public    int   GetRad( ) {…;}
    public    void   Show( ) {…;}   //  надо реализовать
    public    void   MoveTo (int  ax, int  ay)  // и это надо
       { Show( );  SetXY(GetX+ax, GetY+ay);
             // или так:     x = x+ax;   y = y+ay;   если они  protected
         Show( );  }
};
```

Второй дочерний класс – это класс прямоугольников со следующей структурой:

- два закрытых или защищенных поля для ширины и высоты (поля для координат центра будут наследоваться из базового класса фигур)
- конструктор для создания прямоугольника
- методы доступа к ширине и высоте (методы доступа к координатам будут наследоваться)
- реальные методы отображения (Show, однако!) и перемещения (конечно же - MoveTo) прямоугольников

```
TRect = class (TFigure)   //  включаем механизм наследования!
    private    //   или   protected
        h, w : integer;   // x, y   унаследованы и могут использоваться
     public
        constructor  Create (ax, ay, ah, aw : integer);
        procedure  SetHW (ah, aw : integer);  //  остальные  унаследованы!
        function  GetH : integer;
        function  GetW : integer;
        procedure  Show;
```

```
    procedure  MoveTo (ax, ay : integer);
end;
  //  реализация методов
```

```
class  Rect  extends  Figure  {   //  включаем механизм наследования!
   private   int  h, w;   //  или  protected
   public  Rect(int ax, int ay, int ah, int aw) { super(ax, ay); h = ah; w = aw;}
   public  void  SetHW (int ah, int aw);  //  остальные унаследованы!
   public  int  GetH( ) {…;}
   public  int  GetW( ) {…;}
   public  void  Show( ) {…;}
   public  void  MoveTo (int ax, int ay) {…;}
};
```

Н основе класса окружностей можно построить свои производные классы, например – класс эллипсов и класс дуг. Для класса эллипсов к трем полям данных надо добавить еще одно – для вертикальной полуоси (радиус надо теперь трактовать как горизонтальную полуось). Можно добавить и новый метод – поворот эллипса на 90 градусов (для окружности этот метод не имеет смысла). Введение нового свойства и нового метода показывает возможность расширения родительского класса с одновременным использованием всех ранее реализованных методов. В сокращенной форме класс эллипсов можно описать следующим образом:

```
TEllipse = class (TCircle)   // включаем наследование
   private   // объявляем только одно свойство
      r2 : integer;   // x, y  и  r  наследуются
    public
      constructor  Create (ax, ay, ar, ar2 : integer);
     // методы доступа к  r2
     procedure  Show;   // имя – одинаковое во всей иерархии!
```

```
    procedure  MoveTo (ax, ay : integer); // аналогично
    procedure  Rotate90;   // новый метод
  end;
  //  программная реализация методов
```

```
class   Ellipse   extends  Circle         // включаем наследование
 {   private  int    r2;   // x, y  и  r  наследуются
     public    Ellipse (int  ax, int  ay, int  ar, int  ar2) {…;}
       // методы доступа к  r2
     public   void  Show( ) {…;}   // имя – одинаковое во всей иерархии!
     public   void  MoveTo (int  ax, int  ay) {…;} // аналогично
     public   void  Rotate90( ) {…;}   // новый метод
 };
```

Этот класс можно сравнить с аналогичным, но созданным «с нуля», т.е. без использования наследования и родительских классов. Такой класс содержал бы 4 свойства и 8 методов доступа. Даже на таком простом примере можно увидеть преимущества механизма наследования.

Ну а как на основе класса окружностей построить класс дуг? Очень просто – надо добавить лишь два новых поля данных для значений двух углов! Все остальное (координаты центра, радиус) уже есть.

```
TArc = class (TCircle)
   private
       a1, a2 : integer;   // x, y  и  r  наследуются
    public
       // конструктор, методы доступа к a1 и a2, методы Show и MoveTo
       procedure  Rotate (angle : integer);  // новый метод поворота
  end;
```

```
class   Arc  extends   Circle
```

```
{   private   int   a1, a2;   // x, y  и  r  наследуются
    public
       // конструктор, методы доступа к a1 и a2, методы Show и MoveTo
    public   void   Rotate (int   angle) {…;}  // новый метод поворота дуги
};
```

В свою очередь, на основе класса прямоугольников можно создать такие производные классы как класс повернутых прямоугольников (добавляется новое поле данных – угол поворота) и класс закрашенных прямоугольников (добавляется новое поле – цвет закраски).

```
TRotateRect = class (TRect)
   private
       angle : integer;   // x, y, h, w  унаследованы и могут использоваться
    public
       //  конструктор, методы доступа к углу, методы Show и  MoveTo
      procedure  Rotate (ang : integer); // новый метод поворота прям-ника
  end;
```

```
class   RotateRect   extends   Rect
{   private   int   angle;   // x, y, h, w  унаследованы и могут использоваться
    public
       //  конструктор, методы доступа к углу, методы Show и  MoveTo
    public  void  Rotate (int  ang) {…;} // новый метод поворота прям-ника
 };
```

Замечание. Рассмотренный вариант иерархии графических объектов не является окончательным – его можно улучшить, сделать более универсальным, но только после изучения механизмов, основанных на принципе полиморфизма.

Графическое представление иерархии классов с использованием элементов языка UML выглядит следующим образом.