5. Взаимодействие объектов: агрегация и композиция

Как уже было отмечено, объектная программа – это набор **взаимодействующих** объектов **разных** классов, совместно реализующих объектную модель. Создание программ, содержащих объекты одного-двух классов вряд ли принесет ощутимый результат. Объектный подход – это средство борьбы со **сложностью** создаваемых программных систем, в которых используются десятки классов, связанных между собой **различными** способами.

Существует **несколько** основных способов взаимодействия объектов, причем отражением этих взаимодействий на **формальном** уровне являются отношения между соответствующими **классами.** Поэтому построение объектной модели – это выделение необходимых **классов** и установление между ними определенных **отношений.**

Пожалуй, наиболее понятным способом взаимодействия объектов является случай, когда один объект включает в себя в качестве **составляющих частей** объекты других классов. В этом случае между классами устанавливается отношение типа **“часть-целое”.** В оригинальной англоязычной литературе такое отношение описывается термином **“has-a”** (имеет, содержит, включает в себя). В теории ООП такой тип отношения называют **агрегацией**. При этом различают строгую и нестрогую агрегацию. Нестрогая агрегация – это такое взаимодействие объектов, при котором составной объект и образующие его части могут существовать **независимо** и **отдельно** друг от друга. Строгая агрегация или **композиция** возникает тогда, когда существование составного объекта **зависит** от существования входящих в него частей.

Очень важно понимать, что отношение агрегации/композиции можно устанавливать далеко не между любыми объектами. Поэтому первое, с чего следует начинать, это **выяснение возможности** установления подобной связи.

Предположим, имеются два объекта A и B разных классов. Между ними можно установить отношение агрегации, если утверждение «Объект A можно рассматривать как **составную часть** объекта B» не противоречит здравому смыслу.

Приведем несколько примеров из разных предметных областей.

1. Объект «**Компьютер**» можно рассматривать как набор более простых объектов, таких как «**Основная** (материнская) **плата**», «**Монитор**», «**Жесткий диск**», «**Мышь**», «**Клавиатура**». В свою очередь, объект «**Основная плата**» можно рассматривать состоящим из более простых объектов типа «**Процессор**», «**Основная память**» и др.

2. Геометрический объект «**Окружность**» может включать в себя более простой объект **«Точка»** для хранения координат своего центра; аналогично объект «**Прямоугольник**» может состоять из двух объектов-точек, которые задают координаты двух его противоположных углов. Более того, составной объект «**Деталь**» можно разложить (декомпозировать) на отдельные составляющие – объекты «**Окружность**», «**Отрезок**», «**Прямоугольник**» и т.д.

3. Объект «**Оконное приложение**» содержит хотя бы один объект типа «**Оконная форма**», который в свою очередь может включать такие объекты как «**Кнопка**», «**Поле ввода**», «**Список**», «**Таблица**», «**Переключатель**» и т.д.

4. Объект «**Контейнер-массив динамических списков**» состоит из отдельных объектов типа «**Динамический список**», каждый из которых в свою очередь состоит из объектов типа «**Элемент динамического списка**».

С другой стороны – **контрпример**: объект «Стул» вряд ли стоит рассматривать как часть объекта «Стол» и поэтому эти объекты **не следует** связывать отношением агрегации/композиции.

Необходимо отметить, что с **течением времени** связи между объектами могут **меняться**, что является отражением изменения самой объектной

модели, вызванное изменением исходных объектов. Например, если лет 20 назад утверждение «Компьютер можно рассматривать как ЧАСТЬ автомобиля» противоречило общепринятым понятиям, то в **настоящее** время оно вполне имеет право на существование.

Следующим шагом после выяснения **возможности** использования композиционной связи является **разработка необходимых классов**. Очевидно, что в любой объектной информационной модели будет существовать некоторый набор классов, которые описывают **неделимые** объекты, т.е. объекты, не содержащие других объектов. Такие классы объявляются обычным **стандартным** образом. А вот классы для **составных** объектов имеют некоторые **особенности** описания.

Прежде всего, в состав свойств таких классов должны быть включены **свойства объектного типа** или другими словами – переменные соответствующих **классовых** типов. Поскольку такие переменные являются неявными указателями на объекты, с их помощью можно во время **выполнения** программы **связывать** между собой необходимые объекты.

Пример фрагмента описания **составного** класса ComboClass с использованием объектных свойств классовых типов SimpleClass1 и SimpleClass2:

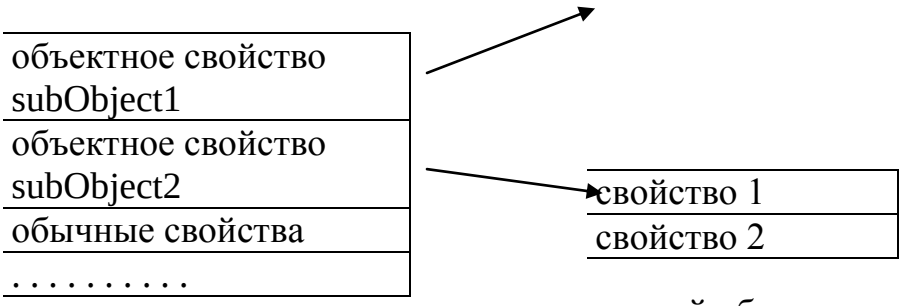
class = ComboClass
subObject1 : SimpleClass1;
subObject2 : SimpleClass2;
// обычные свойства
. . . . . .
end;
	ComboClass **class** {
	SimpleClass1 subObject1;
	SimpleClass2 subObject2;
	// обычные свойства
	. . . . . .
	};

Схематично связь составного объекта с его объектами-частями можно представить следующим образом:

составной объект
класса ComboClass

простой объект класса SimpleClass1

свойство 1
свойство 2



простой объект класса SimpleClass2

Наличие адресных связей между объектами предоставляет составному объекту следующие возможности:

- использовать **данные**, которые хранятся внутри связанных с ним объектов (как правило – через методы доступа)
- **вызывать** открытые методы связанных объектов, т.е. использовать **ранее созданный** программный код

В целом, механизм агрегации/композиции позволяет конструировать **сложные** объекты на основе **ранее созданных** объектов, **постепенно** наращивая уровень сложности.

В качестве **примера** можно привести два варианта описания класса объектов «Компьютер». В первом варианте для хранения информации о процессоре и основной памяти используются строковые поля:

TComputer = **class**
  cpu : **string**;
  memory : **string**;
  . . . . . .
end;
	Computer **class**
	{ **string** cpu;
	  string memory;
	  . . . . . .
	};

Этот вариант можно использовать в простых случаях, но когда о процессоре и памяти надо иметь **более подробную** информацию, целесообразно ввести **отдельные** (самостоятельные) классы для объектов «Процессор» и «Память», а в классе «Компьютер» использовать механизм композиционных **связей** с помощью полей объектного типа.

TCPU = **class**
// поля и методы класса «Процессор»
 	
TMem = **class**
// поля и методы класса «Память»
 	
TComputer = **class**
cpu : **TCPU**; // связующее поле
mem : **TMem**; // связующее поле
// другие поля
// методы
	class CPU
	// поля и методы класса «Процессор»
	
	class Memory
	// поля и методы класса «Память»
	
	class Computer
	CPU cpu; // связующее поле
	Memory mem; // связующее поле
	// другие поля
	// методы

При разработке составных классов обязательно надо продумать механизм **создания** объектов таких классов. При этом возможны **два** основных варианта реализации **конструкторов** в составных классах.

1. Конструктор **составного** класса **должен создать** все необходимые объекты, для чего он должен **вызвать** конструкторы соответствующих классов, при необходимости **передавая** им входные параметры и **устанавливая** значение связующих объектных переменных в адреса созданных этими конструкторами объектов. Пример для составного класса «Компьютер»:

```
constructor  TComputer.Create (входные параметры);
begin
  cpu := TCPU.Create (…);   // установка связи с создаваемым объектом
  mem := TMem.Create (…); // установка связи с создаваемым объектом
  // установка обычных свойств
end;
```

```
public  Computer (входные параметры)  {
  cpu =  new  CPU (…);        // установка связи с создаваемым объектом
  mem =  new  Memory (…); // установка связи с создаваемым объектом
  // установка обычных свойств
```

```
};
```

2. Если объекты-части **уже созданы,** подходят для составного объекта и **известны** адреса их размещения в памяти, то можно **передать** эти адреса как входные параметры конструктору, который должен лишь **использовать** эти значения для установки соответствующих **связующих** объектных свойств. Для этого достаточно объявить входные параметры конструктора как переменные **классового** типа:

```
constructor  TComputer.Create (aCPU : TCPU;   aMem : TMem);
begin
  cpu := aCPU;  // установка связи с существующим объектом
  mem := aMem;   // установка связи с существующим объектом
  // установка обычных свойств
end;
```

```
public  Computer (CPU   aCPU,  Memory   aMem) {
  cpu =  aCPU;   // установка связи с существующим объектом
  mem =  aMem;   // установка связи с существующим объектом
  // установка обычных свойств
};
```

Возможна и **комбинация** этих способов, когда часть объектов **создается**, а часть **используется** в качестве уже существующих.

Что касается **вызова методов** объектов-частей в методах составных классов, то он выполняется **стандартным** образом с использованием **имени** объектного **свойства** и имени **метода**:

```
cpu.SomeMethod( );
```

В заключение приведем перечень **основных шагов**, которые необходимо выполнить для реализации агрегационного (композиционного) взаимодействия объектов:

- определить **возможность** и **целесообразность** применения данного способа для выявленных объектов
- **разработать** (при необходимости) классы для **объектов-частей**
- разработать класс **составных** объектов с включением в его структуру **объектных** (связующих) свойств
- реализовать **конструктор(ы)** составных классов
- реализовать **методы** составных классов с **вызовом** (при необходимости) **ранее созданных** методов связанных с ними объектов.

В качестве примера рассмотрим задачу разработки классов для графических объектов. Решение данной задачи проведем в соответствии с описанными выше шагами. Целью является использование механизма агрегации/композиции для построения **эффективного** набора классов графических объектов.

Исходные объекты (графические примитивы): **Окружность, Прямоугольник**, **Отрезок** и др.

Анализ объектов:

Окружность: содержит **координаты** центра и **методы** доступа к ним

Прямоугольник: содержит **координаты** базовой точки (например — левый верхний угол) и **методы** доступа к ним

Отрезок: содержит **координаты** одного из концов и **методы** доступа к ним

Вывод: объекты имеют **повторяющиеся** данные и методы, которые целесообразно **вынести** во **вспомогательный** класс точек.

Объекты класса «Точка» можно включать как **составные части** в основные объекты-примитивы.

В свою очередь, объекты-примитивы можно использовать как части более сложных **составных** графических объектов (например, типа «Деталь»)

Такой подход несомненно является более эффективным по сравнению с описанием каждого класса «с нуля». Для примера можно сравнить два описания класса для объекта «Простая деталь», состоящего из прямоугольника и окружности:

- описание «с нуля» потребует введения семи свойств (три для окружности и четыре для прямоугольника) и (в общем случае) 14-ти методов доступа!
- описание на основе композиции, как будет видно из дальнейшего материала, потребует лишь ДВУХ объектных свойств!

Описание классов начинается с класса точек, который в рамках данной информационной модели определяет элементарные «неделимые» объекты. Класс очень простой, отвечает за хранение двух целых чисел-координат и доступ к ним через соответствующие методы.

TPoint = **class**
private
x, y : **integer**;
public
constructor Create(ax, ay: integer);
procedure SetXY (ax, ay : integer);
function GetX : integer;
function GetY : integer;
end;
	class Point {
	private int x, y;
	public Point (**int** ax, **int** ay) {…};
	public void SetXY (**int** ax, **int** ay) {…};
	public int GetX( ) {…};
	public int GetY( ) {…};
	};

Новый (улучшенный) класс окружностей будет реализован уже как **составной** и поэтому будет иметь следующие отличия от реализованного «с нуля»:

- вместо двух свойств целого типа с именами x и y вводится **объектное** свойство с именем **Center**, в котором во время **выполнения** программы будет содержаться **адрес** объекта-точки, используемого для хранения координат центра окружности; можно сказать, что класс окружностей **делегирует** работу с координатами своего центра классу точек
- класс может иметь **два конструктора** с разными наборами параметров для создания окружностей разными способами, т.е. либо с **созданием** необходимого объекта-точки, либо с **использованием** уже существующей точки
- методы доступа к координатам центра уже не нужны, т.к. они **реализованы** в классе точек и могут быть **вызваны** при необходимости; в классе остаются только методы доступа к радиусу
- методы прорисовки и перемещения окружности вместо **прямого** использования координат центра должны **обращаться** к **методам доступа** класса точек, поскольку эти координаты теперь объявлены как **закрытые** свойства «чужого» для окружностей класса

```
TCircle = class
  private
    Center : TPoint;   // объектное свойство-указатель на точку
    r : integer;
  public
    constructor  Create (ax, ay, ar : integer); overload;
    constructor  Create (aCenter : TPoint;  ar : integer); overload;
    procedure  SetRad (ar : integer);
```

```
        function  GetRad : integer;
        procedure  Show;
        procedure  MoveTo (ax, ay : integer);
      end;
```

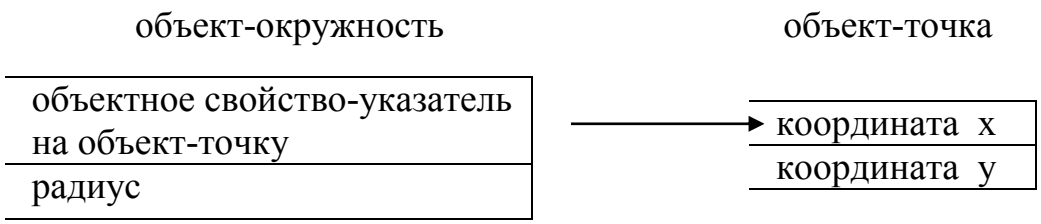
Реализация некоторых методов:

```
    constructor  TCircle.Create (ax, ay, ar : integer);
      begin
        Center := TPoint.Create (ax, ay);  // сначала создаем объект-точку
        r := ar;
      end;
    constructor  TCircle.Create (acenter : TPoint; ar : integer);
      begin
        center := acenter;   // а здесь используем ранее созданную точку
        r := ar;
      end;
    procedure  TCircle.Show;
      begin
         // прорисовка окружности с использованием методов доступа
         // Center.GetX  и  Center.GetY
      end;
    procedure  TCircle.MoveTo (ax, ay : integer);
       begin
            Show;     Center.SetXY(ax, ay);      Show;
       end;
```

```
class   Circle  {
  private    Point   Center;   // объектное свойство-указатель на точку
  private    int   r;
```

```
  public  Circle (int  ax, int  ay, int  ar)
    { Center = new  Point (ax, ay);   r = ar;} // создание объекта-точки
  public  Circle (Point   aCenter,  int  ar)
    { Center = aCenter;  r = ar;}  // использование существующей точки
  public   void  SetRad (int   ar) {…}
  public   int   GetRad ( ) {…}
  public   void  Show( ) { // прорисовка окружности с использованием
                    // методов доступа    Center.GetX( )  и  Center.GetY( ) }
  public  void  MoveTo (int   ax,  int   ay)
    { Show( );  Center.SetXY(ax, ay);   Show( ); }
};
```

Схема взаимодействия объектов:



Аналогично можно определить составной класс прямоугольников в одном из двух вариантов:

- через координаты верхнего левого угла, ширину и высоту (одно объектное свойство-указатель на точку)
- через координаты верхнего левого и правого нижнего углов (два объектных свойства-указателя на две точки)

```
  TRect = class       // первый способ
    private
      TopLeft : TPoint;  // объектное свойство для координат угловой точки
      h, w : integer;
    public
```

```
        constructor  Create (ax, ay, ah, aw : integer); overload;
        constructor  Create (aTL : TPoint;  ah, aw : integer); overload;
        // не забыть: методы доступа к полям h и w
        procedure  Show;                // реализация - через методы доступа
        procedure  MoveTo (ax, ay : integer);  // аналогично
    end;
```

Реализация конструкторов:

```
    constructor  TRect.Create (ax, ay, ah, aw : integer);
      begin
        TopLeft := TPoint.Create (ax, ay);  // создание объекта-точки
         h := ah;   w := aw;
      end;
    constructor  TCircle.Create (aTL : Tpoint;  ah, aw : integer);
      begin
        TopLeft := aTL;   // использование ранее созданной точки
        h := ah;   w := aw;
      end;
```

```
class  Rect  {
  private   Point   TopLeft;   // объектное свойство для координат угла
  private   int   h, w;

  public  Rect (int  ax, int  ay, int  ah,  int  aw)
    { TopLeft = new  Point (ax, ay); // создание объекта для угловой точки
      h = ah;   w = aw;}
  public  Rect (Point   aTL,  int  ah,  int  aw)  // использование объекта-точки
    { TopLeft = aTL;  h = ah;   w = aw;}
  public . . . . .    // методы доступа к полям h и w
  public   void  Show( ) { // прорисовка прямоугольника с использованием
```

```
                    // методов доступа   TopLeft.GetX( ),  TopLeft.GetY( ) }
  public void MoveTo (int  ax,  int  ay)
    { Show( );  TopLeft.SetXY(ax, ay);   Show( ); }
};
```

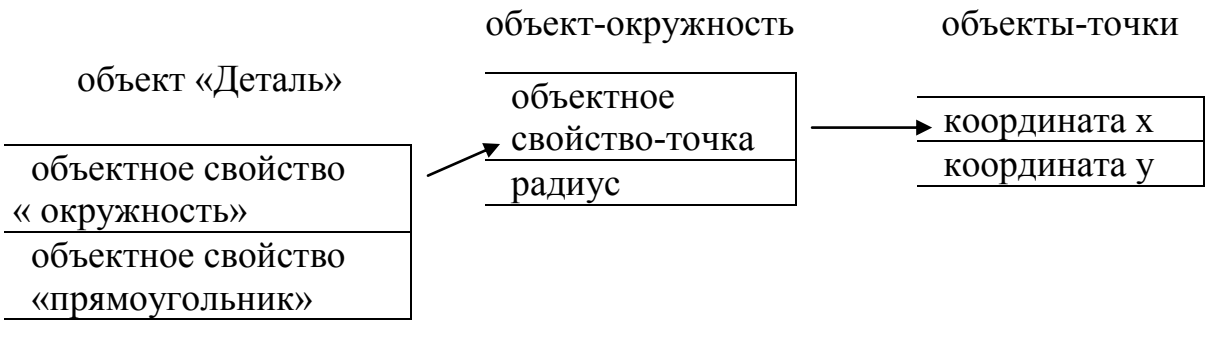
После создания классов для окружностей и прямоугольников можно легко **собирать** объекты этих классов в рамках более **сложных** графических объектов. Например, можно **объединить** окружность и прямоугольник в рамках составного объекта «**Простая деталь**», причем практически вся функциональность такого объекта будет обеспечена ранее реализованными методами соответствующих классов. Для этого достаточно в класс «Простая деталь» включить **два объектных свойства** для адресации объектов «Окружность» и «Прямоугольник».

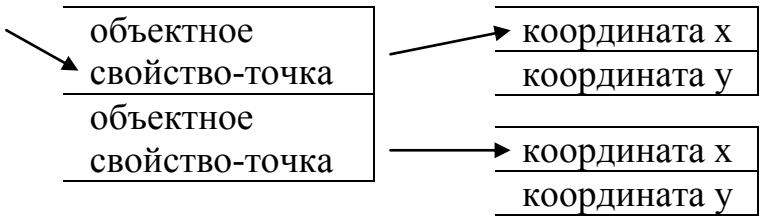
```
TDetal = class
  private
     Circ : TCircle;    // поле для связи с объектом-окружностью
     Rect :  TRect;      // поле для связи с объектом-прямоугольником
  public
    constructor  Create (aCx, aCy, ar, aRx, aRy, ah, aw : integer); overload;
    constructor  Create (aCir : TCircle;  aRect : TRect); overload;
    procedure  Show;
    procedure  MoveTo (ax, ay : integer);
end;

Реализация методов:
  constructor  TDetal.Create (aCx, aCy, ar, aRx, aRy, ah, aw : integer);
     begin
        Circ := TCircle.Create (aCx, aCy, ar);  // создание объекта-окружности
       Rect := TRect.Create (aRx, aRy, ah, aw); //создание прямоугольника
```

```
    end;
  constructor  TDetal.Create (aCir : TCircle;  aRect : TRect);
    begin
        Circ := aCir;   // использование ранее созданной окружности
        Rect := aRect;   // использование прямоугольника
    end;
  procedure   TDetal.Show;   // аналогично - метод перемещения
    begin
      Circ.Show;   Rect.Show;    // и все, больше ничего не надо!
    end;
```

```
class   Detal  {
  private    Circle   circ;   // поле для связи с объектом-окружностью
  private    Rect    rect;    // поле для связи с объектом-прямоугольником

  public  Detal (int  aCx, int  aCy, int  ar, int  aRx,  int  aRy, int  ah, int  aw)
    { circ = new  Circle (aCx, aCy, ar); // создание окружности
       rect = new  Rect (aRx, aRy, ah, aw);} // и прямоугольника
  public  Detal (Circle   aCirc,  Rect   aRect)  // использование
    { circ = aCirc;  rect = aRect;} // существующих объектов
  public   void  Show( ) { circ.Show( );  rect.Show( ); }  // очень просто!
  public  void  MoveTo (int   ax,  int   ay)
    { circ.MoveTo(ax, ay);  rect.MoveTo(ax, ay); }
 };
```

Схема взаимодействия объектов при выполнении программы:





объект-прямоугольник

Ниже приводятся фрагменты **демонстрационных** программ, в которых **разными** способами создаются объекты описанных выше классов.

```
var
  MyPoint1, MyPoint2 : TPoint;
  MyCirc1, MyCirc2 : TCircle;
  MyRect : TRect;
  MyDetal : TDetal;
begin
  MyCirc1 := TCircle.Create (100, 100, 50);   // создание окружности «с нуля»
  MyPoint1 := TPoint.Create (100, 100);       // создание точки
  MyCirc2 := TCircle.Create (MyPoint1, 50);   // окружность: второй способ
  MyPoint2 := TPoint.Create (300, 200);       // еще одна точка
  MyRect := TRect.Create (MyPoint2, 200, 100);   // прямоугольник
  MyDetal := TDetal.Create (100, 100, 50, 50, 50, 200, 300);   //  деталь «с нуля»
  MyDetal := TDetal.Create (MyCirc1, MyRect);   //  используем созданные
  MyDetal.Show;
  MyDetal.MoveTo (200, 200);
```

```
  Point   MyPoint1, MyPoint2;
  Circle   MyCirc1, MyCirc2;
  Rect   MyRect;
```

```
Detal   MyDetal;
MyCirc1 = new  Circle (100, 100, 50);   // создание окружности «с нуля»
MyPoint1 = new  Point (100, 100);       // создание точки
MyCirc2 = new  Circle (MyPoint1, 50);   // окружность: второй способ
MyPoint2 = new  Point (300, 200);       // еще одна точка
MyRect = new  Rect (MyPoint2, 200,  100);   // прямоугольник
MyDetal = new  Detal (100, 100, 50, 50, 50, 200, 300);   //  деталь «с нуля»
MyDetal = new  Detal (MyCirc1, MyRect);   //  используем созданные
MyDetal.Show( );
MyDetal.MoveTo (200, 200);
```