6. Разработка контейнера на основе адресного списка

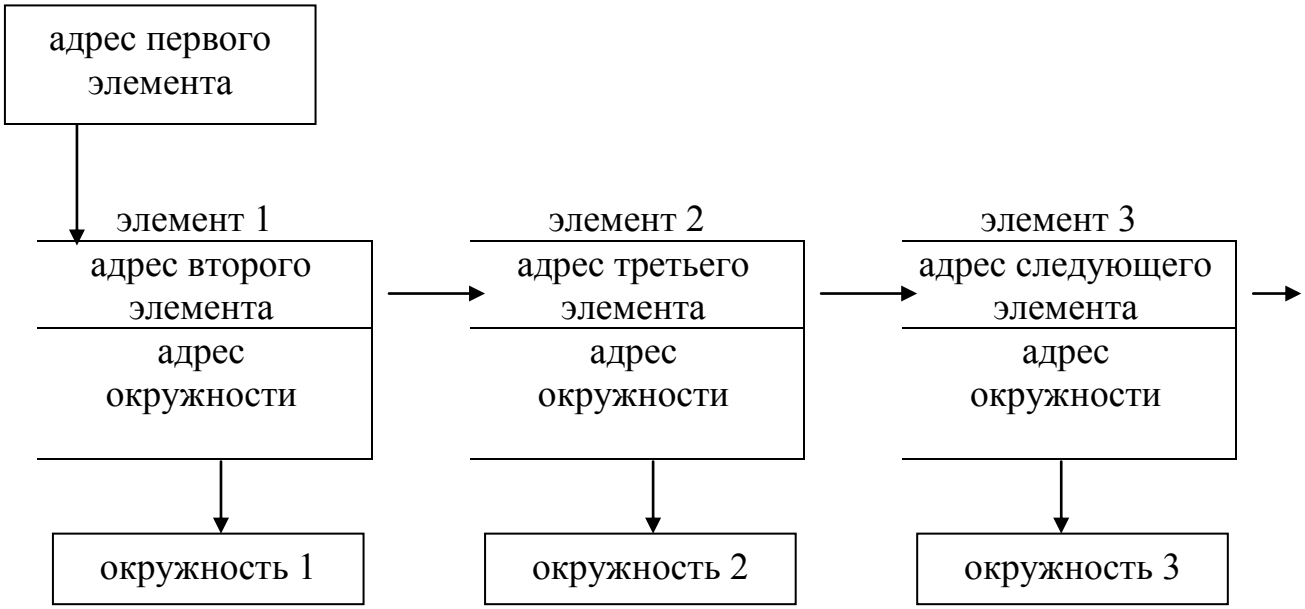
В разделе 4 в качестве примера описания и использования классов был представлен контейнер на основе массива. Как известно, существует и альтернативная реализация контейнера, которая использует **адресные** связи между элементами и механизм **динамического** распределения памяти. Сравнение и алгоритмическая реализация этих вариантов представлена в классических учебниках по Структурам Данных. Краткое введение в этот материал можно найти в учебном пособии [8].

Динамический адресный список представляет из себя набор элементов, занимающих свои **отдельные** области памяти. Элементы связываются в логическую последовательность с помощью **адресных** полей.

Можно предложить **два** варианта объектной реализации адресного списка. Более **общий** вариант предполагает использование для связи собираемых в контейнер объектов вспомогательных **объектов-посредников.** Более **простой** вариант основан на включении служебных адресных полей **непосредственно** в сами объекты, поэтому он возможен только если доступно **исходное** описание класса хранимых объектов.

Рассмотрим первый вариант, причем в качестве примера возьмем объектную реализацию контейнера для хранения и обработки графических объектов-окружностей. В дальнейшем, после изучения принципа **полиморфизма**, этот простейший вариант контейнера будет **модифицирован** и получит способность хранить **любые** графические объекты.

Схематично списковый контейнер для объектов-окружностей можно представить следующим образом.



Каждый отдельный элемент списка будем рассматривать как **самостоятельный** объект, отвечающий за хранение адреса **следующего** элемента и адреса **связанной** с ним окружности. Сам контейнер как набор элементов-объектов тоже является объектом, реализующим основную функциональность динамического списка.

Для программной реализации прежде всего необходимо ввести два класса: класс «**ЭлементСписка**» и класс «**СписковыйКонтейнер**». Неформально структуру этих классов можно описать следующим образом.

Класс «**ЭлементСписка**» должен содержать:

- адрес следующего элемента-объекта (поле данных объектного типа)
- адрес окружности (поле данных классового типа Circle)
- одно или несколько информационных полей (при необходимости)
- конструктор(ы) для создания нового элемента-объекта с необходимыми значениями свойств
- методы доступа к закрытым адресным данным

Класс «**СписковыйКонтейнер**» должен содержать:

- адрес первого элемента списка (поле объектного типа)
- адрес последнего элемента списка (при необходимости)
- одно или несколько информационных полей (при необходимости)
- конструктор(ы) для создания контейнера в начальном состоянии (чаще всего пустом)
- методы доступа к информационным полям (при необходимости)
- методы добавления, удаления, поиска и итеративной обработки

Формализованное описание класса «**ЭлементСписка**» :

```
TPosrednik  =  class
private
   ItemInf : string;   //  или другой необходимый тип
   Next : TPosrednik;   // свойство-указатель на следующий элемент
   Circ : TCircle;  // свойство-указатель адресуемой окружности
public
   constructor  Create (aInf : string;  aNext : TPosrednik;   aCirc : TCircle );
   function   GetNext : TPosrednik;    // получение адреса след. элемента
   function  GetCirc : TCircle;    // важно: метод для доступа к окружности!
   function   GetInf : string;
   procedure   SetNext (aNext : TPosrednik);   // изменение адреса след. эл-та
   procedure   SetCirc (aCirc : TCircle);  // для присоединения другой окр-ти
   procedure   SetInf (aInf : string);  // для изменения информационного поля
end;
```

```
class   Posrednik  {
   private   int   ItemInf;   //  или другой необходимый тип
   private   Posrednik   Next;   // свойство-указатель на следующий элемент
   private   Circle   Circ;  // свойство-указатель адресуемой окружности
```

```
  public  Posrednik (int  aInf,  Posrednik   aNext,  Circle  aCirc) {…;}
  public  Posrednik   GetNext( ) {…;}    // получение адреса след. элемента
  public  Circle   GetCirc( ) {…;}  // важно: метод для доступа к окр-ти!
  public  int   GetInf ( ) {…;}
  public  void  SetNext (Posrednik   aNext) {…;}  // изменить адрес след. эл-та
  public void  SetCirc (Circle  aCirc) {…};  // для присоединения другой окр.
  public  void   SetInf (int   aInf) {…};  // для изменения информац. поля
};
```

Описание основного класса «**СписковыйКонтейнер**» (для упрощения изложения добавление нового элемента будем производить в начало списка):

```
CircleListContainer = class
private
  ContInf : string;
  First : TPosrednik;  // указатель на первый элемент списка
public
  constructor   Create (ainf : string);  // создание пустого списка
  function  GetFirst : TPosrednik;
 //  здесь должны быть методы доступа к информационному свойству
  procedure  Add (ainf : string;  aCirc : TCircle);  // добавление объекта
  function  Delete (ainf : string) : boolean;  // удаление эл-та по его инф. части
  function  Search (aRad : integer) : TPosrednik;  // поиск эл-та по радиусу окр.
  procedure  ShowAll;   // итератор для отображения всех окружностей
  procedure  MoveAll (dx, dy : integer);  // итератор-перемещатель
 end;
```

Программная реализация некоторых методов (**обратить внимание на использование Get-методов для доступа к закрытым данным объектов-элементов списка и объектов-окружностей!**):

```
constructor   CircleListContainer.Create (ainf : string);
```

```
  begin   First := nil;   ContInf := ainf;
  end;
procedure   CircleListContainer.Add ( ainf : string;  aCirc : TCircle);
  begin   // напомним, что добавление — в начало списка!
      First := TPosrednik.Create (ainf, First, aCirc);  //  красиво и компактно!
  end;
procedure  CircleListContainer.ShowAll;
  var   Temp : TPosrednik;   // вспом. объектная перем. для прохода по списку
  begin
      Temp := First;
      while (Temp <> nil)  do
            begin
              Temp.GetCirc.Show;    // доступ к методу объекта-окружности
              Temp := Temp.GetNext;   //  получение адреса след. элемента
            end;
  end;
function  CircleListContainer.Search (aRad : integer) : TPosrednik;
 var   Temp : TPosrednik;
 begin
   result := nil;
   Temp := First;
   while (Temp <> nil)  do
     if  (Temp.GetCirc.GetR = aRad)   then
     begin
       result := Temp;
       break;
     end
     else  Temp := Temp.GetNext
 end;
```

```
class  CircleListContainer  {
  private  Posrednik  First;  // указатель на первый элемент списка
  private  int   ContInf;
  public  CircleListContainer  (int  ainf) { First = null ;  ContInf = ainf;}
  public  void  Add(Circle  aCirc,  int  ainf)
     { First = new  Posrednik (ainf,   First,   aCirc) ;} // красиво и компактно!
  public  Circle   Delete (int   ainf) {…}  // удаление
  public  Circle   Search (int   aRad) {…}  // поиск по радиусу
  public  void  ShowAll ( )
      { Posrednik   Temp = First;   // вспом. указатель для прохода по списку
        while  (Temp != null )
            { Temp.GetCirc ( ).Show ( );  // доступ к методу объекта-окружности
              Temp = Temp.GetNext ( );  // доступ к адресному полю
                  }   // конец цикла
      }  //  конец метода отображения
  public  void  MoveAll (int  ax,  int  ay ) {…}
};   //  конец описания класса
```

Замечание. В языках Java и C# вместо двух **отдельных** классов можно воспользоваться возможностью объявления **вложенных** классов: класс «ЭлементСписка» можно объявить **внутри** класса «СписковыйКонтейнер». В этом случае доступ к полям данных объектов-элементов можно выполнять напрямую, без использования методов доступа.

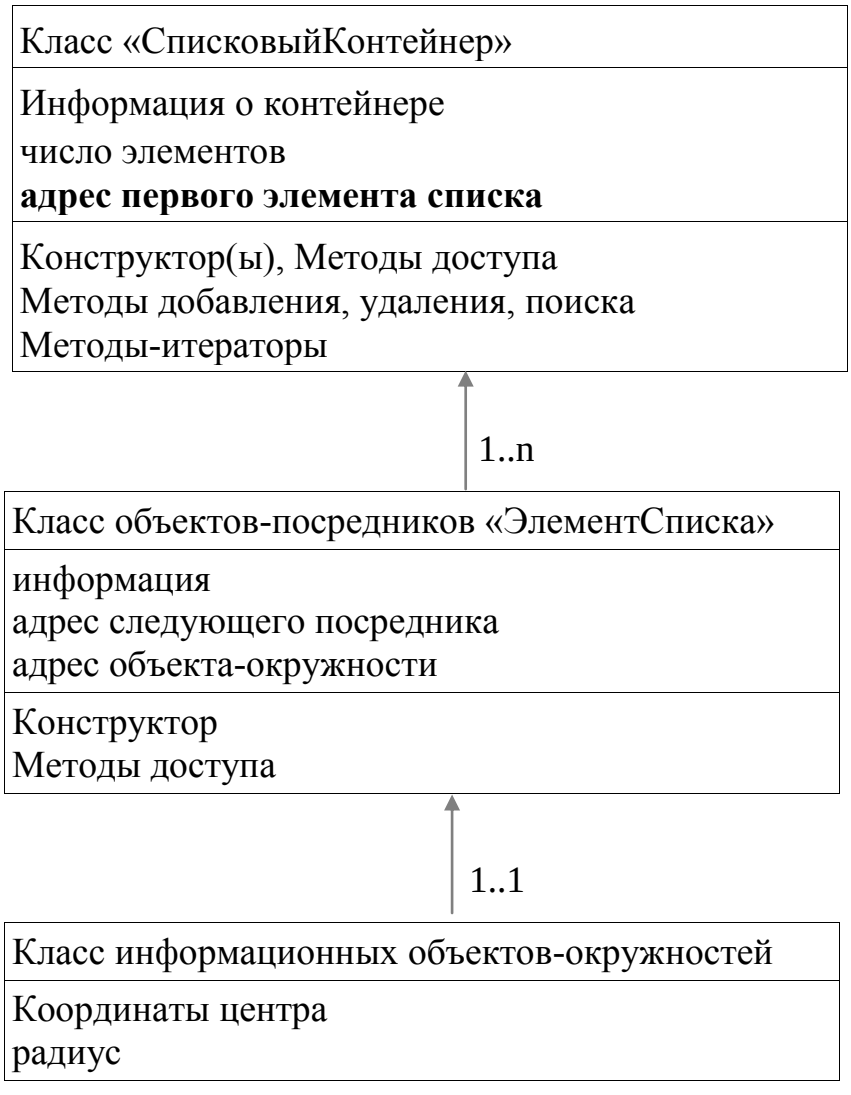
В заключение приведем фрагменты программ, демонстрирующих использование спискового контейнера.

```
  var  MyCont : CircleListContainer;
  MyCont := CircleListContainer.Create (…);
  MyCirc := TCircle.Create(random(…), random(…), random(…));
  MyCont.Add ('текст',  MyCirc);
```

```
// создали и добавили еще несколько окружностей
MyCont.ShowAll;
MyCont.MoveTo (…);
```

```
CircleListContainer   MyCont = new  CircleListContainer (…);
Circle   MyCirc = new  Circle (. . . . .);
MyCont.Add (MyCirc,  1);
  // повторить два последних действия необходимое число раз
MyCont.ShowAll ( );
MyCont.MoveAll (50, 50);
```

Упрощенная диаграмма классов для спискового контейнера объектов-окружностей имеет следующий вид:



Конструктор(ы) Методы доступа Методы отображения и перемещения

Во втором, более **простом** варианте класс объектов-посредников не нужен, но использовать данный способ можно не всегда. В качестве примера приведем описание (сокращенное) класса «**Студент**» со служебным адресным полем.

```
TStudent = class
    private
      Fam : string;
      NextStud : TStudent;        // служебное связующее поле
. . . . . .
    public
      constructor  Create (aFam : string;  aNext : TStudent);
      function  GetNext : TStudent;     // получить ссылку на следующего
      procedure  SetNext (aStud : TStudent);     // изменить ссылку
    // остальные методы — как ранее было описано
  end;
```

Класс «**СписковыйКонтейнер**» для объектов-студентов:

```
StudListContainer = class
  private
    infCont : string;    //  или другой необходимый тип
    First : TStudent;     // указатель на первого студента в списке
  public
    constructor   Create (ainf : string);  // конструктор создает пустой список
    function  GetFirst : TStudent;
```

```
    procedure  Add (aFam : string );     // добавление - в начало списка
    function  Delete (aFam : string) : boolean;  // удаление
    function  GetStud (aFam : string) : TStudent;  // поиск по фамилии
// остальные методы
 end;
```

Реализация метода добавления студента в начало списка:

```
procedure   StudListContainer.Add (aFam : string);
   begin   First := TStudent.Create (aFam, First);
   end;
```

В свою очередь, объекты-контейнеры сами могут быть элементами контейнера более высокого уровня: студенческие группы могут собираться в рамках контейнера «Факультет», объекты-факультеты могут объединяться на уровне контейнера «Институт» и т. д. При этом способы объединения могут быть разными, как на основе массивов, так и на основе адресных связей.