10. Разработка универсальных полиморфных контейнеров

Механизм переопределения методов и свойство полиморфности объектных переменных позволяют создавать **универсальные** объекты-контейнеры, которые могут хранить и обрабатывать объекты разных классов, входящих в некоторую **общую** иерархию. При необходимости можно создать сверхуниверсальный контейнер, способный хранить объекты абсолютно любых классов. Для этого, как уже указывалось в разделе 9, достаточно в качестве базового типа использовать корневой класс Object/ TObject. Однако на практике вместо такого универсального хранилища часто удобнее использовать более **специализированные** контейнеры для некоторой **подиерархии** классов, которые **по смыслу** близки друг другу.

В качестве **первого примера** такого специализированного хранилища рассмотрим контейнер для графических объектов. Ранее в разделах 4 и 6 были рассмотрены два простейших варианта контейнера для хранения и обработки объектов-окружностей. Теперь, после изучения принципов **наследования** и **полиморфизма**, эти простейшие контейнеры можно сделать более универсальными, способными хранить и обрабатывать **любые** графические объекты.

Обязательным условием реализации такого контейнера является наличие **библиотеки графических классов**, удовлетворяющей следующим требованиям:

- в вершине иерархии находится **корневой** класс (например – класс Figure)
- в корневом классе объявлен **виртуальный абстрактный** метод отображения фигуры (например – Show)
- в корневом классе объявлен и реализован **обычный** метод перемещения, который **наследуется** в каждом дочернем классе
- в каждом дочернем классе **переопределен** метод отображения

Рассмотрим сначала реализацию контейнера для графических объектов на основе **обычного массива**. Превратить контейнер для окружностей в универсальный достаточно просто – надо лишь везде **заменить** классовый тип **Circle общеродовым** классовым типом **Figure**. Однако универсальный характер контейнера позволяет **расширить** базовую функциональность простейшего контейнера для окружностей за счет ряда **новых** методов, таких как:

- выборочное отображение объектов
- выборочное перемещение объектов
- получение имени объекта, связанного с заданным элементом массива

Вот описание класса для универсального контейнера, как обычно – на двух языках (для наглядности те изменения, которые появились по сравнению с начальным вариантом, выделены курсивом и подчеркнуты):

```
TArrayFigsContainer = class
 private
   count : integer;                  // текущее число объектов в контейнере
   Figs : array [1..100] of TFigure; // массив указателей на объекты
public
   constructor Create;   // создаем пустой контейнер
   function GetCount : integer;
   function Add (aFig : TFigure) : boolean;  // добавляем в конец набора
   function Delete (ai : integer) : TFigure; //  удаление объекта по номеру
   function SearchName (aName : string):integer;  // поиск по имени объекта
   procedure ShowAll;    //  метод-итератор для показа всех объектов
   procedure  ShowOnly (aName : string); // выборочное отображение
   procedure MoveAll (dx, dy : integer);  // общее перемещение
   procedure  MoveOnly (dx, dy : integer; aName : string);  // выборочное
   function  GetObjectName (ai : integer): string;  // получение имени объекта
end;
```

```
class  ArrayFigsContainer  {
  private   Figure [ ]   Figs;     // будущий массив указателей
  private   int   Count;            // счетчик числа объектов в контейнере
  public   ArrayFigsContainer (int  aSize)  {
           Figs = new  Figure [aSize];   //  создание массива
           Count = 0; }
  public   int   GetCount { return  Count;}
  public   int   GetSize { return  Figs.Length; }
  public   bool   Add (Figure   aFig) { // код добавления нового объекта }
  public   Figure   Delete (int   ai) {  //  код удаления объекта по его номеру}
  public   int   SearchName (string   aName) {  //  код поиска по имени  }
  public   void   ShowAll( )  {  //  код отображения всех объектов  }
  public   void   ShowOnly(string  aName) {//код выборочного отображения }
  public   void   MoveAll (int  dx, int  dy)  {   // код перемещения  }
  public   void   MoveOnly (int  dx, int  dy, string   aName ) { // код выборочного }
  public   string  GetObjectName (int  ai){ // код получения имени объекта }
}          //  конец описания класса
```

Некоторые комментарии к этим классам.

- объявление базового массива основано на свойстве полиморфности объектных переменных: элементы массива могут хранить адреса размещения в памяти любых графических объектов
- в методе добавления используется возможность объявления полиморфного параметра, через который передается адрес конкретного добавляемого в контейнер объекта
- в методе удаления тип возвращаемого значения также является полиморфным: можно получить ссылку на любой удаляемый объект
- в новых методах выборочного отображения и перемещения через входной параметр передается имя класса, который является

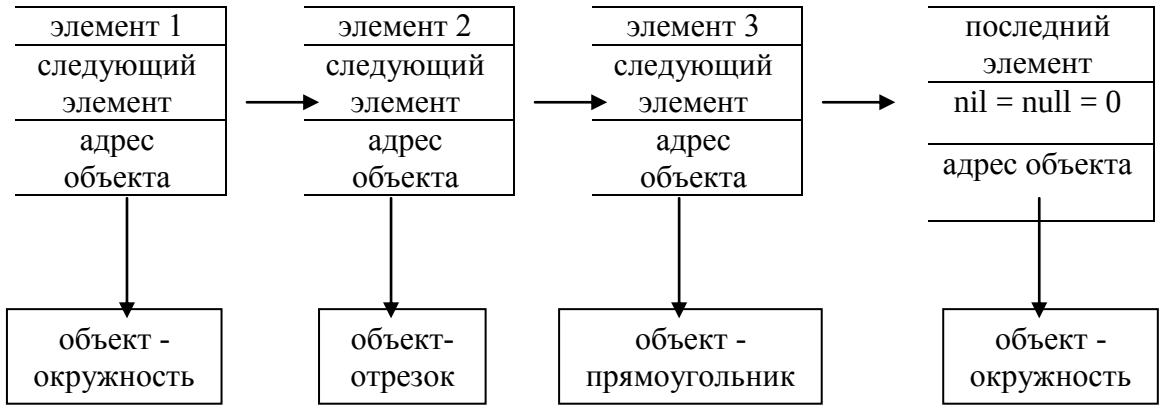
родоначальником той подиерархии, объекты которой надо показать или переместить; само отображение или перемещение реализуется с использованием операторов проверки динамического типа объектов, т.е. операторов **is** или **instanceof**

- новый метод *GetObjectName* для получения имени того класса, объект которого связан в данный момент с заданным элементом массива, должен использовать средства, предоставляемые механизмом рефлексии

Что касается использования универсального контейнера, то оно включает выполнение обычных действий:

- объявление объектной переменной контейнерного типа
- создание контейнера с помощью конструктора
- создание необходимых графических объектов с помощью соответствующих конструкторов
- добавление созданных объектов в контейнер
- полное или частичное отображение находящихся в контейнере объектов
- полное или частичное перемещение объектов
- удаление заданных объектов

Альтернативная реализация – на основе динамического списка полиморфных указателей.



В разделе 6 было отмечено, что для реализации спискового контейнера в общем случае надо использовать два класса – класс **элементов** списка и класс самого спискового **контейнера.** Для перехода от простого контейнера к универсальному необходимо заменить конкретный классовый тип **Circle** на общий тип **Figure**.

Описание измененных классов для элементов спискового контейнера:

```
ListItem = class
private
  ItemInf : string;   //  или другой необходимый тип
  Next : ListItem;   // свойство-указатель на следующий элемент
  Fig : TFigure;  // свойство-указатель адресуемого объекта
public
  constructor  Create (aInf : string;  aNext : ListItem;   aFig: TFigure);
  function   GetNext : ListItem;    // получение адреса след. элемента
  function  GetFig : TFigure;    // важно: получение адреса объекта!
  function   GetInf : string;
  procedure   SetNext (aNext : ListItem);   // изменение адреса след. эл-та
  procedure   SetFig (aFig : TFigure);  // для присоединения другого объекта
  procedure   SetInf (aInf : string);  // для изменения информационного поля
end;
```

```
class ListItem  {
  private   int   ItemInf;   //  или другой необходимый тип
  private   ListItem   Next;   // свойство-указатель на следующий элемент
  private   Figure   Fig;  // свойство-указатель адресуемого объекта
  public  ListItem (int  aInf,  ListItem   aNext,  Figure   aFig) {…;}
  public  ListItem   GetNext( ) {…;}    // получение адреса след. элемента
  public  Figure   GetFig( ) {…;}  // важно: получение адреса объекта!
  public  int   GetInf ( ) {…;}
```

```
  public  void   SetNext (ListItem   aNext) {…;}  // изменить адрес след. эл-та
  public void  SetFig (Figure  aFig) {…};  // для присоедин. другого объекта
  public  void   SetInf (int   aInf);  // для изменения информационного поля
};
```

Программная реализация методов вполне очевидна

Измененное и расширенное описание универсального спискового контейнера для графических объектов:

```
FigsListContainer = class
private
  ContInf : string;
  First : ListItem;  // указатель на первый элемент списка
public
  constructor   Create (ainf : string);  // создание пустого списка
  function  GetFirst : ListItem;
 //  здесь должны быть методы доступа к информационному свойству
  procedure  Add (ainf : string;  aFig : TFigure);  // добавление объекта
  function  Delete (ainf : string) : boolean;  // удаление эл-та по его инф. части
  function  Search (aName : sring) : ListItem;  // поиск по имени объекта
  procedure  ShowAll;   // отображение всех объектов
  procedure  ShowOnly (aName : string);  // выборочное отображение
  procedure  MoveAll (dx, dy : integer);  // перемещение всех объектов
  procedure  MoveOnly (dx, dy : integer; aName : string);  // выборочное
  function  GetObjectName (ai : integer): string;  // получение имени объекта
 end;
```

В качестве примера программной реализации приведем только метод ShowAll, поскольку реализация других методов практически совпадает с рассмотренной ранее. Да и новая реализация метода ShowAll мало отличается от старой и приводится здесь для того, чтобы еще раз обратить

внимание на технику использования методов доступа для вызова методов отображения графических объектов.

```
procedure  FigsListContainer.ShowAll;
   var   Temp : ListItem;      // вспом. переменная для прохода по списку
   begin
        Temp := First;
        while (Temp <> nil)  do
              begin
                Temp.GetFig.Show;    // доступ к методу текущего объекта
                Temp := Temp.GetNext;   //  получение адреса след. элемента
              end;
   end;
```

```
class  FigsListContainer  {
  private  ListItem  First;  // указатель на первый элемент списка
  private  int   ContInf;
  public  CircleListContainer  (int  ainf) { First = null ;  ContInf = ainf;}
  public  void  Add(Figure  aFig,  int  ainf)
     { First = new  ListItem (ainf,   First,   aFig) ;} // красиво и компактно!
  public  void  ShowAll ( )
      { ListItem   Temp = First;   // вспом. указатель для прохода по списку
         while  (Temp != null )
              { Temp.GetFig( ).Show ( );  // доступ к методу текущего объекта
                Temp = Temp.GetNext ( );  // доступ к адресному полю
                      }   // конец цикла
      }  //  конец метода отображения
  public   int   SearchName (string   aName) {  //  код поиска по имени  }
  public   void   ShowOnly(string  aName) {//код выборочного отображения }
  public   void   MoveAll (int  dx, int  dy)  {   // код перемещения  }
```

```
  public   void   MoveOnly (int  dx, int  dy, string   aName ) { // код выборочного }
  public   string  GetObjectName (int  ai){ // код получения имени объекта }
};   //  конец описания класса
```

В заключение отметим, что во всех стандартных библиотеках классов, поддерживающих объектные языки, реализованы различные стандартные контейнеры/коллекции. Реализация этих контейнеров использует понятие **интерфейса**, которое изучается в следующем разделе пособия, поэтому краткая характеристика стандартных контейнеров будет рассмотрена несколько позже.

Второй пример не связан с графическими объектами, но в то же время использует все рассмотренные ранее механизмы.

Постановка задачи. В организации работают сотрудники двух типов с разными формами оплаты:

- на основе фиксированного оклада и доли ставки: зарплата = оклад * ставка
- на основе почасовой оплаты: зарплата = число_отработанных_часов * стоимость_часа

Требуется создать универсальную объектную программу для организации хранения данных о сотрудниках и начисления им зарплаты.

Как обычно, разработку проведем по шагам.

Шаг 1. Анализ задачи с выделением возможных объектов.

Можно предложить следующие объекты:

- **сотрудник-окладник:** фамилия, должность, оклад, ставка, расчет зарплаты

- **сотрудник-почасовик:** фамилия, должность, отработанные часы, часовая ставка, расчет зарплаты

- **объект-контейнер** «Коллектив»: число сотрудников, полиморфное хранилище ссылок на сотрудников разных типов, добавление и удаление сотрудников, формирование зарплатной ведомости

Шаг 2. Анализ объектов.

Объекты первых двух типов имеют **общие** данные и **общее** поведение и могут рассматриваться как **разновидности** общего понятия «Сотрудник», следовательно, можно построить небольшую **иерархию** классов. Корневой класс должен содержать **общие** данные, такие как фамилия и должность, а также определять **общую** функциональность, такую как начисление зарплаты.

Шаг 3. Проектирование необходимых классов

Введем следующие классы:

- **- абстрактный** корневой класс **Сотрудник**: фамилия, должность, конструктор, методы доступа, **абстрактный** метод расчета зарплаты
- **- производный** класс **Окладник**: оклад, ставка, конструктор, методы доступа, **переопределенный** метод расчета зарплаты
- **- производный** класс **Почасовик**: часовая ставка, количество отработанных часов, конструктор, методы доступа, **переопределенный** метод расчета зарплаты
- - класс **контейнерных** объектов на основе массива полиморфных ссылок на сотрудников разных типов

Шаг 4. Формальное описание классов создаваемой иерархии.

Код на языке Delphi/Free Pascal

```
TSotr  = class                  // базовый абстрактный класс
  private   fam, dolgn : string ;              // общие свойства
  public   constructor  Create (aFam, aDolgn : string );
    // открытые методы доступа к общим свойствам
          function   Zarplata : real; virtual; abstract;  // абстрактный метод
end;
```

```
TSotrOklad = class (TSotr)            // первый дочерний класс
   private   oklad : integer;         // новые уникальные свойства
             stavka : real;
   public  constructor  Create (aFam, aDolgn : string; aOklad : integer; aStavka
          : real);
         // методы доступа к новым свойствам
            function   Zarplata : real; override;  // переопределение метода
end;

TSotrPochas = class (TSotr)           // второй дочерний класс
  private   OtrabChas, ChasStavka : integer;     // свои уникальные свойства
  public  constructor  Create ( aFam, aDolgn : string; aOtrChas, aChasSt :
          integer );
           // методы доступа к новым свойствам
           function   Zarplata : real; override;    // переопределение метода
end;

// реализация всех методов вполне очевидна
```

Код на языке C#

```
abstract   class   Сотрудник
{  private   string   Фам, Должн;
   public   Сотрудник (string   фам,   string   должн) { …}
   // открытые методы доступа
   public   abstract   virtual   float   Расчет( );
```

```
};

class  Окладник:  Сотрудник
{  private  int  Оклад;
   private  float  Ставка;
   public  Окладник (string  фам,  string  должн, int  оклад, float  ставка)
          : Сотрудник (фам, должн)  {…};
   public  override  float  Расчет( ) { return  Оклад * Ставка };
};

class  Почасовик: Сотрудник
{  private  int  ОтрабЧасов, ЧасСтавка;
   public  Почасовик(…) : Сотрудник (фам, должн) {…};
   public override  float  Расчет( ) { return  ОтрабЧасов * ЧасСтавка};
};
```

Шаг 5. Разработка контейнерного класса (минимальная конфигурация)

```
Container = class
 private
   Sotrs = array  of  TSotr ;  //динамический массив полиморфных ссылок
   count : integer;
 public   constructor  Create ( size : integer );
            procedure  AddSotr (aSotr : TSotr );
            function  DelSotr (aFam : string) : boolean;
            procedure  Vedomost;
```

```
            function   GetZarpl ( nom : integer) : real;
end;
// реализация некоторых методов
// простейший вариант метода  Vedomost (может быть использован в
            консольной программе)
procedure   Container.Vedomost;
var   i : integer;
        sumZarpl : real;   // накапливаем суммарную зарплату
begin
  sumZarpl := 0;
    // при проходе по массиву сотрудников используем полиморфизм!
  for  i := 0  to  count-1  do
  begin
      sumZarpl := sumZarpl + Sotrs[ i ].Zarplata;
                    // вызов разных версий  метода  Zarplata!
      writeln (Sotrs[ i ].GetFam + '  ' + Sotrs[ i ].Zarplata);
  end;
  writeln ( ' ВСЕГО: ' + sumZarpl );
end;
// простейший вариант метода получения зарплаты сотрудника с заданным
            номером (метод может быть использован в оконных приложениях)
function   Container.GetZarpl ( nom : integer) : real;
begin
  "проверка входного параметра  nom";
  result := Sotrs[ nom ].Zarplata;   // и здесь разные версии!
```

```
end;
```

```
class   КонтейнерСотрудников
{  private   Сотрудник [ ]   ВсеСотр;  //массив полиморфных ссылок
   private   int   Количество;
   public    КонтейнерСотрудников (int   Размер) { //код };
   public    void  ДобавитьСотр (Сотрудник   новый) { // код };
   public   bool   УдалитьСотр (string   фамилия)  { // код }
   public    void   Ведомость ( ) // далее - простейший вариант кода для
                                        // консольного приложения
     {   float   суммЗарплата = 0;
         for (int   i = 0;  i < Количество;  i++)
        // при проходе по массиву сотрудников используем полиморфизм!
           {  суммЗарплата =  суммЗарплата + ВсеСотр[ i].Расчет( );
              // вызов разных версий метода   Расчет!
              WriteLine( ВсеСотр[ i].GetFam( ) + '   ' + ВсеСотр[ i].Расчет( );}
           WriteLine ( 'ИТОГО: ' + суммЗарплата);
     };
  public   float   ЗарплатаСотрудника (int   номерСотр )
  // метод получения зарплаты сотрудника с заданным номером
  // (можно использовать в оконных приложениях)
  { " проверка входного параметра  номерСотр";
     return    ВсеСотр[ номерСотр ].Расчет( );}  // и здесь разные версии!
};
```

Шаг 6. Разработка тестовой программы (фрагмент консольного приложения)**.**

```
var   sotr : TSotr;  //  объявление полиморфной объектной переменной
      kollectiv : Container;  // переменная-контейнер
  . . . . . .
kollectiv := Container.Create(20);  //  создание контейнера на 20 сотрудников
sotr := TSotrOklad.Create ('Бендер', 'директор', 50000, 1.5); //создание
kollectiv.AddSotr (sotr);  // добавление сотрудника-окладника
kollectiv.AddSotr (TSotr.Pochas ('Балаганов', 'менеджер', 500, 100);
  // а здесь сразу всё – и создание, и добавление в одном флаконе!
. . . . . //  создание и добавление остальных сотрудников
kollectiv.Vedomost; // расчет и вывод зарплатной ведомости
```

```
КонтейнерСотрудников   Коллектив = new  Container(20);
        // создание контейнера на 20 сотрудников
 // создание и добавление сотрудника-окладника
Коллектив.ДобавитьСотр (new  Окладник('Бендер', 'директор', 50000, 1.5);
  // создание и добавление сотрудника-почасовика
Коллектив.ДобавитьСотр (new  Почасовик('Балаганов', 'менеджер', 500, 10);
. . . . . // создание и добавление остальных сотрудников
Коллектив. Ведомость( ); // расчет и вывод зарплатной ведомости
```