Министерство образования Российской Федерации
Пермский государственный технический университет
Кафедра автоматизированных систем управления

А.М. НОТКИН

ТЕОРИЯ И ПРАКТИКА ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Утверждено Редакционно-издательским советом университета в качестве учебного пособия

Пермь 2001

Содержани е

ПРЕДИСЛОВИЕ

Цель практикума лабораторных работ – закрепить знания, полученные при изучении теоретической части курсов и получить практические навыки разработки объектно-ориентированных программ и использования объектно-ориентированных библиотек. Практикум включает выполнение восьми лабораторных работ. Первые четыре работы связаны с базовыми понятиями объектно-ориентированного программирования, такими как объекты и классы, наследование, полиморфизм и виртуальные методы, обработка событий. Последние четыре работы посвящены использованию объектно-ориентированной библиотеки Turbo Vision и охватывают такие разделы профессионального программирования, как просмотр информации в окнах, организация диалогов в программах, хранение объектов в коллекциях и потоках. После первых четырех и последних четырех работ предусмотрено выполнение итоговых лабораторных работ.

Лабораторные работы выполняются в среде Borland Pascal 7.0. Необходимо проследить, чтобы установленная у вас среда Borland Pascal 7.0 содержала модули библиотеки Turbo Vision.

Лабораторная работа № 1
МОДУЛЬНАЯ СТРУКТУРА ПРОГРАММЫ

Цель. Получить практические навыки разработки программ с модульной структурой. Освоить разработку модулей на языке Turbo Pascal.

Основное содержание работы.

Определить структуру данных в соответствии с заданием. Разместить структуры в динамической памяти, объединив их в группу. Сохранить группу в файле.

Краткие теоретические сведения.

- **Модули.** Модули предназначены для поддержки принципов модульного программирования при разработке программ, основным из которых является принцип *скрытия информации.*

Большие программы сложны для разработки, понимания и ведения. Чтобы прийти к лучшим и более надежным программам, требуются методы, позволяющие разбивать программы на меньшие и более удобные части. Частичное решение этих проблем дают подпрограммы (процедуры и функции в Паскале). Однако подпрограммы лишь частично помогают решить проблемы, связанные с эффективностью разработки программ. Успешное решение этих проблем дают модули.

Модули позволяют легко и надежно вести разработку программ группами программистов (бригадами), хранить компоненты программ в библиотеке и не дублировать их в каждой использующей их программе. Модули позволяют использовать *раздельную компиляцию*. Это означает, что модули запоминаются в библиотеке программ не в виде исходных текстов, а в откомпилированной форме.

•Модули в Turbo Pascal.

Модули в Turbo Pascal – это средства, заимствованные из языка **Модула-2.**

Модуль в Turbo Pascal имеет следующий формат:

<заголовок>;<раздел интерфейса> <раздел реализации> <раздел инициализации>.

Формат заголовка:
unit <идентификатор модуля>

Формат раздела интерфейса:
interface
<uses-предложения>
<разделы>
где **разделы** – это:
– раздел констант;
– раздел типов;
– раздел переменных;
– раздел заголовков процедур и функций.

Формат раздела реализации:
implementation
<uses-предложения>
<разделы>
где **разделы** – это:
– раздел меток;
– раздел констант;
– раздел типов;
– раздел переменных;
– раздел процедур и функций.

Формат раздела инициализации:
либо **end**, либо **раздел операторов**.

В списке импорта интерфейсного раздела (uses-предложения) перечисляются имена модулей, информация интерфейсных частей которых должна быть доступна в данном модуле.

В списке экспорта интерфейсного раздела (в разделах) описываются имена, которые определены в данном модуле и использовать которые разрешено во всех других модулях и программах, включающих имя данного модуля в своем списке импорта.

В списке импорта раздела реализации перечисляются имена модулей, информация интерфейсных частей которых используется (доступна) в данном модуле (в его разделе реализации).

Описания раздела реализации доступны только в этом модуле и не доступны ни в одном другом.

Операторы раздела инициализации выполняются при запуске программы в том же порядке, в каком имена модулей описаны в предложении uses.

При разработке модулей необходимо соблюдать следующие правила:

1) не допускается одновременное использование модулей с одинаковыми именами;

2) идентификатор модуля, указанный в заголовке unit должен совпадать с именами файлов, содержащих исходный(.pas) и объектный(.tpu) код.

3) если идентификатор модуля длиннее восьми символов, то он должен совпадать с именами файлов по первым восьми символам.

- **Группа .** Объект (структура), в который включены другие объекты (структуры). Объекты, входящие в группу, называются *элементами группы.* Элементы группы, в свою очередь, могут быть группой.

Примеры групп:

1) Окно в интерактивной программе, которое владеет такими элементами, как поля ввода и редактирования данных, кнопки, списки выбора, диалоговые окна и т.д.

2) Агрегат, состоящий из более мелких узлов.

3) Огород, состоящий из растений, системы полива и плана выращивания.

4) Некая организационная структура (например, ФАКУЛЬТЕТ, КАФЕДРА, СТУДЕНЧЕСКАЯ ГРУППА).

Реализовать группу можно следующим образом.

Создается связанный список записей типа *TItem:*

```
Type
    PItem=^TItem;
    TItem=record
            next   :PItem;
            item   :PObject;
```

end;

Поле *item* указывает на объект, включенный в группу. Группа содержит поле *last* типа *PItem*, которое указывает на начало связанного списка объектов, включенных в группу.

Для работы с группой должны быть определены следующие процедуры и функции:

1) P*rocedure GroupInsert(var group:TGroup;p:PObject);*

Включает в группу group структуру, на которую указывает p.

2) *Procedure GroupShow(group:TGroup);*

Позволяет просмотреть группу.

3) Function GroupEmpty(*group:TGroup)*:boolean;

Показывает, есть ли хотя бы один элемент в группе.

4) Procedure GroupDelete(var *group:TGroup; p:PObject*);

Удаляет элемент из группы.

- **Итератор.** Итераторы позволяют выполнять некоторые действия для каждого элемента определенного набора данных.

  Fo*r all элементов набора* ***do*** *дейст****в*** *ия*

Такой цикл мог бы быть выполнен для всего набора, например чтобы напечатать все элементы набора. Или мог бы искать некоторый элемент, который удовлетворяет определенному условию, и в этом случае такой цикл может закончиться, как только будет найден требуемый элемент.

Мы будем рассматривать итераторы как специальные процедуры группы, позволяющие выполнять некоторые действия для всех объектов, включенных в группу. Примером итератора является метод Group*Show*.

Нам бы хотелось иметь такой итератор, который позволял бы выполнять над всеми элементами группы действия заданные произвольной процедурой или функцией пользователя. Такой итератор можно реализовать, если эту процедуру (функцию) передавать ему через параметр процедурного типа.

Type TProc=procedure(p:PObject);

Процедура-итератор может объявляться таким образом:

Procedure GroupIter(group:TGroup;proc:TProc);

Передаваемая итератору процедура (функция) должна быть откомпилирована с атрибутом ***far.***

Procedure MyProc(p:PObject);far;

Итератор можно сделать более удобным, если передавать ему указатель на процедуру:

Procedure GroupIter(group:TGroup;proc : pointer);

•**Процедурные типы.** В стандартном Паскале процедуры и функции рассматриваются только как части программы, которые можно выполнять с помощью вызова процедуры или функции. В Borland Pascal процедуры и функции трактуются гораздо шире: здесь допускается интерпретация процедур и функций, как объектов, которые можно присваивать переменным и передавать в качестве параметров. Такие действия можно выполнять с помощью процедурных типов.

В описании процедурного типа задаются параметры, а для функции – еще и результат функции.

Формат описания процедурного типа:

procedure <список формальных параметров>

или

function <список формальных параметров>:тип результата

Характерно, что синтаксис записи процедурного типа в точности совпадает с записью заголовка процедуры или функции, только опускается идентификатор после ключевого слова procedure или function. Приведем некоторые примеры описаний процедурного типа:

```
type
  Proc = procedure;
  SwapProc = procedure(var X, Y: Integer);
  StrProc = procedure(S: String);
  MathFunc = function(X: Real): Real;
  DeviceFunc = function(var F: text): Integer;
  MaxFunc = function(A, B: Real; F: MathFunc): Real;
```

Имена параметров в описании процедурного типа играют чисто декоративную роль – на смысл описания они не влияют.

Borland Pascal не позволяет описывать функции, которые возвращают значения процедурного типа. Результат функции должен быть указателем

строкового, вещественного, целого, символьного, булевского типа или иметь перечислимый тип, определенный пользователем.

Переменной процедурного типа можно присвоить процедурное значение. Процедурные значения могут быть следующими:

– значениями nil;

– ссылкой на переменную процедурного типа;

– идентификатором процедуры или функции.

В контексте процедурных значений описание процедуры или функции можно рассматривать как специальный вид описаний констант, когда значением константы является процедура или функция.

Использование процедурных переменных, которым в операторе вызова процедуры или функции присваивается значение nil, приводит к ошибке. Значение nil предназначено для указания того, что процедурной переменной не присвоено значение, и там, где процедурная переменная может получить значение nil, участвующие в этой процедурной переменной вызовы процедур и функций следует подвергать проверке:

```
if @P <> nil then P(I, J);
```

Обратите внимание на использование операции @ для указания того, что P проверяется, а не вызывается.

- **Параметры процедурного типа.** Поскольку процедурные типы допускается использовать в любом контексте, то можно описывать процедуры или функции, которые воспринимают процедуры и функции в качестве параметров.

Параметры процедурного типа особенно полезны в том случае, когда над множеством процедур или функций нужно выполнить какие-то общие действия. В следующем примере показывается использование параметров процедурного типа для вывода трех таблиц различных арифметических функций:

```
program Tables;
type
  Func = function(X,Y: integer): integer;
function Add(X,Y: integer): integer; far;
begin
   Add := X + Y;
 end;
```

```
function Multiply(X,Y: integer): integer; far;
begin
  Multiply := X*Y;
end;
function Funny(X,Y: integer): integer; far;
begin
   Funny := (X+Y) * (X-Y);
end;
procedure PrintTable(W,H: integer; Operation: Func);
var
  X,Y : integer;
begin
  for Y := 1 to H do
  begin
    for X := 1 to W do Write(Operation(X,Y):5);
    Writeln;
  end;
  Writeln;
end;

begin
 PrintTable(10,10,Add);
 PrintTable(10,10,Multiply);
 PrintTable(10,10,Funny);
end.
```

В данном случае процедура PrintTable представляет собой общее действие, выполняемое над функциями Add, Multiply и Funny.

Если процедура или функция должны передаваться в качестве параметра, они должны удовлетворять тем же правилам совместимости типа, что и при присваивании, то есть:

1) такие процедуры или функции должны компилироваться с директивой far;

2) они не могут быть встроенными функциями;

3) они не могут быть вложенными;

4) они не могут описываться с атрибутами inline или interrupt.

Порядок выполнения работы.

1. Определить структуру (в соответствии с вариантом задания), а также процедуры и функции для работы с ней (см. п. "Методические указания"). Поместить определения в модуль **unit1.**

2. Написать программу тестирования модуля **unit1.**

3. Отладить модуль **unit1.**

4. Определить типы, процедуры и функции для организации работы с группой. Поместить определения в модуль **unit2.**

5. Написать программу тестирования модуля **unit2.**

6.Отладить модуль **unit2.**

7. Определить типы, процедуры и функции для организации работы с файлом (сохранение группы в файле и загрузка ее из файла). Поместить определения в модуль **unit3.**

8. Написать программу тестирования модуля **unit3.**

9. Отладить модуль **unit3.**

10. Определить итератор (в соответствии с вариантом задания) и поместить определение в модуль **unit2.**

11. Написать программу тестирования итератора и отладить его.

12. Написать программу, в которой создаются структуры, помещаются в группу, группа просматривается, затем сохраняется в файле, после сохранения удаляется, затем вновь создается путем считывания из файла и просматривается.

13. Отладить эту программу.

14. Написать отчет.

Методические указания.

1. Модуль **unit1** содержит определения структуры, а также процедуры и функции для работы с ней. В Pascal структура – это запись(**record**). Например, структура ПЕРСОНА определяется так:

```
TPerson=record
        name:string35;   {имя}
        age:integer;     {возраст}
        end;
```

Следует определить также тип указателя на структуру:

PPerson=^TPerson;

2. Для работы со структурой следует описать следующие процедуры и функции:

– функцию, которая размещает структуру в памяти и возвращает указатель на нее, например:

function PersonCreate(name1:string35;age1:integer):PPerson;

– процедуру, удаляющую структуру из памяти, например:

procedure PersonDestroy(Var p:PPerson); где p – указатель на удаляемую структуру;

– процедуры для установки полей структуры, например:

procedure PersonInit(p:PPerson;name1:string35;age1:integer);

procedure SetName(p:PPerson;name1:string35);

procedure SetAge(p:PPerson;age1:integer);

– функции, возвращающие значения полей, например:

function GetName(p:PPerson):string35;

function GetAge(p:PPerson):integer;

– процедуру для просмотра структуры (вывода значений ее полей), например: procedure PersonShow(p:PPerson);

3. Второй модуль(**unit2**) содержит определения типов, процедур и функций для организации работы с группой (см. п. “Краткие теоретические сведения”).

4. Следует определить процедуру типа:

procedure GroupInsert*(var group:TGroup;p:PPerson);*

которая включает в группу group структуру PPerson.

Следует определить также процедуры для создания (инициализации) группы, для ее просмотра и удаления: GroupCreate, GroupShow, GroupDestroy.

5. Третий модуль (**unit3**) содержит процедуры, необходимые для сохранения группы в файле и загрузке ее из файла. Здесь необходимо определить следующие процедуры:

AssignFile – связать файловую переменную с внешним файлом;

OpenFileRead – открыть файл для чтения;

OpenFileWrite – открыть файл для записи;

ReadPersons – читать записи из файла и поместить их в группу;

WritePersons – сохранить группу (входящие в нее структуры) в файле;

CloseFile – закрыть файл.

6. Примерный вид основной программы:

```
uses Unit1,Unit2,Unit3;
Type TProc=procedure(person:TPerson);
Var p:PPerson;
   a,b:TGroup;
   f:TFile;
Procedure Show(person:PPerson;age1:integer);far;
{Показывает студентов, чей возраст больше age1}
begin

if person^.age>age1 then PersonShow(person);
end;
Begin
AssignFile(f1,'person1.dat'  );
AssignFile(f2,'person2.dat'  );
GroupCreate(a,'ГРУППА А');
GroupCreate(b,'ГРУППА Б');
p:=PersonCreate('Иванов' 25);
GroupInsert(a,p);
p:=PersonCreate('Петров' 35);
GroupInsert(a,p);
p:=PersonCreate('Сидоров',45);
GroupInsert(a,p);
{аналогично для группы b}
GroupShow(a);
OpenFileWrite(f1);
WritePersons(f1,a);
CloseFile(f1);
GroupDestroy(a);
GroupShow(a); {показываем группу, чтобы убедиться, что она пустая}
{ аналогично для группы b }
OpenFileRead(f1);
```

```
ReadPersons(f1,a);
GroupShow(a);
{ аналогично для группы b }
GroupIter(a,Show,35);
GroupDestroy(a);
GroupDestroy(b);
End.
```

Содержание отчета.

1. Титульный лист по следующей форме:

Министерство образования Российской Федерации

Пермский государственный технический университет

Кафедра автоматизированных систем управления

Отчет по лабораторной работе № 1
по дисциплине
ТЕОРИЯ И ПРАКТИКА ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

студента II-го курса специальности АСУ
группы АСУ-99-1
Иванова И.И.

ТЕМА: Модульная структура программы

Вариант № 13

Пермь 2000

2. Формулировка задания в соответствии с вариантом (тип структуры и итератор).

3. Интерфейсные части всех модулей.

4. Объяснить выполнение следующих процедур:

– просмотр структуры;

– просмотр группы;

– включение структуры в группу;

– удаление структур из группы;
– загрузка группы из файла.
5. Объяснить выполнение реализованного итератора.
6. Приложение (на дискете):
а) листинги всех модулей и основная программа : pas-файлы;
б) exe-файл программы.

Рекомендуемая литература.

1. Епанешников А., Епанешников В. Программирование в среде Турбо-Паскаль 7.0. – М.: Диалог-МИФИ ,1993.

2. Марченко А.И. Программирование в среде Borland Pascal 7.0. – Киев – Юниор, 1996.

3. Фаронов В.В. Турбо-Паскаль 7.0.: В 2-х кн. – М.: Нолидж, 1997.

Приложение. Варианты заданий

#	Структура	Итератор
1	СТУДЕНТ	Имена всех юношей (девушек)
2	СТУДЕНТ	Имена студентов указанного курса
3	СТУДЕНТ	Количество юношей (девушек)
4	СТУДЕНТ	Количество студентов на указанном курсе
5	СЛУЖАЩИЙ	Имена служащих со стажем не меньше заданного
6	СЛУЖАЩИЙ	Имена служащих заданной профессии
7	СЛУЖАЩИЙ	Количество служащих со стажем не меньше заданного
8	СЛУЖАЩИЙ	Количество служащих заданной профессии
9	КАДРЫ	Имена рабочих в заданном цехе

#	Структура	Итератор
10	КАДРЫ	Имена рабочих заданного разряда
11	ЦЕХ	Количество продукции заданного наименования
12	ЦЕХ	Наименование продукции, количество которой не менее заданного
13	БИБЛИО-ТЕКА	Наименование книг, стоимость которых выше заданной
14	БИБЛИО-ТЕКА	Количество книг указанного автора
15	ЭКЗАМЕН	Имена студентов, сдавших экзамен на отлично
16	ЭКЗАМЕН	Имена студентов, сдававших экзамен в заданный день
17	ЭКЗАМЕН	Количество студентов, не сдавших экзамен
18	АДРЕС	Имена живущих на заданной улице
19	АДРЕС	Имена живущих на четной стороне заданной улицы
20	АДРЕС	Количество жителей заданной улицы

#	Структура	Итератор
21	ТОВАР	Количество товара заданного наименования
22	ТОВАР	Наименование товара, количество которого превышает заданную величину
23	ТОВАР	Наименование товара, стоимостью не выше заданной

#	Структура	Итератор
24	КВИТАН-ЦИЯ	Номера квитанций указанной даты
25	КВИТАН-ЦИЯ	Общая сумма всех квитанций указанной даты

Описания структур

1. СТУДЕНТ
имя – string
курс – integer
пол – boolean

2. СЛУЖАЩИЙ
имя – string
профессия – string
рабочий стаж – integer

3. КАДРЫ
имя – string
номер цеха – unteger
разряд – integer

4. ЦЕХ
имя – string
шифр – string
количество – integer

5. БИБЛИОТЕКА
имя – string
автор – string
стоимость – real

6. ЭКЗАМЕН
имя студента – string
дата – integer
оценка – integer

7.АДРЕС
имя – string
улица – string
номер дома – integer

8.ТОВАР
имя – string
количество – integer
стоимость – real

9.КВИТАНЦИЯ
номер – integer
дата – integer
сумма – real

Лабораторная работа № 2
ПОЛИМОРФНЫЕ ОБЪЕКТЫ И НАСЛЕДОВАНИЕ

Цель. Получить практические навыки создания иерархии классов, создания и удаления объектов, использования виртуальных методов.

Краткие теоретические сведения.

- **Объектно-ориентированное программирование (ООП).** Методология, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса (типа), а классы образуют иерархию на принципах наследования.
- **Объект.** Осязаемая реальность, имеющая четко определенное поведение. Объект обладает состоянием, поведением и индивидуальностью. Структура и поведение сходных объектов определены в общем для них классе. Термины *"объект"* и *"экземпляр"* взаимозаменяемы. В Borland Pascal объект – это переменная объектного типа.
- **Класс(объектный тип).** Множество объектов с общей структурой и поведением. Термины *"класс"* и *"объектный тип"* взаимозаменяемы. В языке Borland Pascal(но не Object Pascal!) используется термин "объектный тип". Объектный тип в Borland Pascal – это структура, аналогичная записи(record), которая наряду с *полями данных* содержит *поля процедур и функций*(то есть *методы*).
- **Наследование.** Отношение между классами, при котором класс использует структуры и/или поведение другого (*одиночное наследование*) или других (*множественное наследование*) классов. Наследование вводит иерархию "общее/частное" (*иерархия классов*), в которой класс (потомок) наследует от одного или нескольких более общих суперклассов (предков). Подклассы обычно дополняют или переопределяют унаследованные структуру и поведение. Посредством наследования экземпляры класса получают доступ к данным и методам классов-предков без их повторного определения.
- **Полиморфизм.** Свойство, позволяющее называть разные действия одним именем. Например, объекты родственных классов могут иметь одноименные методы, которые выполняют различные действия в зависимости от того, объект какого класса вызвал этот метод. При этом свя-

зывание объекта с конкретным методом может быть на этапе компиляции (*раннее связывание - статические методы*) или на этапе выполнения (*позднее связывание - виртуальные методы*).

- **Полиморфные объекты.** Объекты разных (но имеющих общего родителя) классов, названные одним именем. Следовательно, любой полиморфный объект может по-своему реагировать на некий общий набор операций. В Borland Pascal именем полиморфных объектов может быть имя указателя объектного типа или имя формального параметра объектного типа.
- **Конструктор.** Метод, используемый для создания нового объекта. Обеспечивает решение двух задач: он выделяет память под новую переменную – объект и гарантирует, что переменная инициализируется надлежащим образом. В Borland Pascal конструктор только инициализирует объект (память выделяется либо статически, либо динамически с помощью процедуры *NEW*). Кроме этого конструктор выполняет определенную работу по настройке для механизма обработки виртуальных методов. Поэтому каждый объектный тип, который имеет виртуальный метод, должен иметь конструктор.
- **Деструктор.** Метод, который используется для разрушения объекта и освобождения занимаемой им памяти. Для полиморфных объектов вызов деструктора гарантирует освобождение ровно столько байт, сколько занимает объект. Деструктор – это, как правило, виртуальный метод.

Порядок выполнения работы.

1. Создать иерархию классов для определенной предметной области. Например, для предметной области ФАКУЛЬТЕТ:

```
Type
PPerson=^TPerson;
TPerson=object
 name:string;   {фамилия, имя, отчество }
 sex:boolean; {пол: true-мужской}
 age:byte;      {возраст}
 Constructor Init(name1:string;sex1:boolean;age1:byte);
 Destructor Done;virtual;
```

```
 Function GetName:string;
 Function GetSex:boolean;
 Function GetAge:byte;
 Procedure SetName(NewName:string);
 Procedure SetAge(NewAge:byte);
 Procedure Show;virtual;
end;

PStudent=^TStudent;
TStudent=object(TPerson)
 year:byte;         {курс}
 rating:byte;       {рейтинг}

 Constructor Init(name1:string;sex1:boolean;age1:byte;
  year1:byte;rating1:byte);
 Destructor Done;virtual;
 Function GetYear:byte;
 Function GetRating:byte;
 Procedure SetYear(NewYear:byte);
 Procedure SetRating(NewRating:byte);
 Procedure Show;virtual;
end;

{ Служащий }
PEmployee=^TEmployee;
TEmployee=object(TPerson)
 post:string;{должность }
 work:byte;{рабочий стаж}
 Constructor Init(name1:string;sex1:boolean;age1:byte;
             post1:string;work1:byte);
 Destructor Done;virtual;
 Function GetPost:string;
 Function GetWork:byte;
 Procedure SetPost(NewPost:string);
 Procedure SetWork(NewWork:byte);
 Procedure Show;virtual;
end;
```

```
{Преподаватель }
PTeacher=^TTeacher;
TTeacher=object(TEmployee)
 TeacherWork:byte;{педагогический стаж}
 Constructor Init(name1:string;sex1:boolean;age1:byte;
 post1:string;work1:byte;teacherwork1:byte);
 Destructor Done;virtual;
 Function GetTeacherWork:byte;
 Procedure SetTeacherWork(NewTeacherWork:byte);
 Procedure Show;virtual;
end;
```

2. Определить для классов виртуальный метод *Show,* который показывает состояние объекта.

Например:

```
Procedure TPerson.Show;
Var stsex:string;
Begin
writeln;
if sex then stsex:='мужской' else stsex:='женский' ;
writeln('имя:' ,name,' пол:' ,stsex,' возраст' ,age);
End;
```

3. Определить внешнюю процедуру *View* с параметром объектного типа, которая показывает состояние объекта, передаваемого ей через параметр.

Например:

```
Procedure View(AObject:PPerson);
```

4. Написать демонстрационную программу, в которой создаются, показываются и разрушаются объекты всех классах.

Методические указания.

1. Предметная область задается преподавателем или выбирается студентом самостоятельно. Для студентов заочного отделения эта область должна быть связана с местом работы студента.

2. Предусмотреть размещение объектов как в статической, так и в динамической памяти.

3. Предусмотреть просмотр объектов как с помощью методов *Show*, так и с помощью процедуры *View*.

4. Определение классов (объектных типов), процедур и функций поместить в модуль *L02UXXX*, где *XXX*- аббревиатура фамилии, имени и отчества, например, Иванов Петр Григорьевич – *IPG.*

5.Демонстрационную программу поместить в файл *L02XXX.PAS.*

Содержание отчета.

1. Титульный лист.
2. Постановка задачи.
3. Определение классов.
5. Реализация методов для одного из классов (выбирается самим студентом).
6. Реализация процедуры *View.*
7. Приложение: листинг демонстрационной программы.

Лабораторная работа № 3
ИЕРАРХИЯ ОБЪЕКТОВ И ГРУППА. ИТЕРАТОРЫ

Цель. Получить практические навыки создания объектов-групп и использования методов-итераторов.

Основные теоретические сведения.

- **Абстрактный класс.** Класс, который представляет абстрактную концепцию, для которой не могут существовать объекты. Абстрактный класс используется не для создания экземпляров, а только как интерфейс и в качестве базы для других классов. Абстрактный класс пишется в предположении, что его конкретные подклассы дополняют его структуру и поведение, скорее всего, реализовав *абстрактные методы*.

Полезно, чтобы корневой класс в иерархии классов был абстрактным классом. Он должен иметь основные общие свойства со своими производными классами так, чтобы эти свойства наследовались из него, но сам не может использоваться для объявления объектов. Вместо этого он используется для объявления указателей, которые могут иметь доступ к объектам наследуемых классов.

Абстрактный класс также обеспечивает интерфейс для изменений в иерархии классов в целом. Если добавляются некоторые новые свойства, которые применяются ко всем объектам иерархии, то их можно просто добавить к абстрактному классу.

- **Абстрактный метод.** Объявленный, но не реализованный метод в абстрактном классе. Абстрактный метод может рассматриваться как обобщение *переопределения*. В обоих случаях поведение родительского класса изменяется для потомка. Для абстрактного метода, однако, поведение просто не определено. Любое поведение задается в производном классе.

Одно из преимуществ абстрактного метода является чисто концептуальным: программист может мысленно наделить нужным действием абстракцию сколь угодно высокого уровня. Например, для геометрических фигур мы можем определить метод *Draw*, который их рисует: треугольник *TTriangle*, окружность *TCircle*, квадрат *TSquare*. Мы определим аналогичный метод и для абстрактного родительского класса *TGraphObject*. Однако

такой метод не может выполнять полезную работу, поскольку в классе *TGraphObject* просто нет достаточной информации для рисования чего бы то ни было. Тем не менее присутствие метода *Draw* позволяет связать функциональность (рисование) только один раз с классом *TGraphObject,* а не вводить три независимые концепции для подклассов *TTriangle*, *TCircle*, *TSquare.*

Имеется и вторая, более актуальная причина использования абстрактного метода. В объектно-ориентированных языках программирования со статическими типами данных (C++, Borland Pascal) программист может вызвать метод класса, только если компилятор может определить, что класс действительно имеет такой метод. Предположим, что программист хочет определить полиморфную переменную типа *TGraphObject,* которая будет в различные моменты времени содержать фигуры различного типа. Это допустимо для полиморфных объектов. Тем не менее компилятор разрешит использовать метод *Draw* для переменной, только если он сможет гарантировать, что в классе переменной имеется этот метод. Присоединение метода *Draw* к классу *TGraphObject* обеспечивает такую гарантию, даже если метод *Draw* для класса *TGraphObject* никогда не выполняется. Естественно, для того, чтобы каждая фигура рисовалась по-своему, метод *Draw* должен быть *виртуальным.*

- **Группа .** Объект, в который включены другие объекты. Объекты, входящие в группу, называются *элементами группы.* Элементы группы, в свою очередь, могут быть группой.

  Примеры групп:

  1) окно в интерактивной программе, которое владеет такими элементами, как поля ввода и редактирования данных, кнопки, списки выбора, диалоговые окна и т.д. Примерами таких окон являются объекты классов, порожденных от абстрактного класса *Tgroup* (TDeskTop, TWindow, TDialog) в иерархии классов библиотеки **Turbo Vision** и объекты классов, порожденных от *TWindowObject* в иерархии классов библиотеки **OWL**;

  2) агрегат, состоящий из более мелких узлов;

  3) огород, состоящий из растений, системы полива и плана выращивания;

  4) некая организационная структура (например, ФАКУЛЬТЕТ, КАФЕДРА, СТУДЕНЧЕСКАЯ ГРУППА).

Мы отличаем «группу» от «контейнера». Контейнер используется для хранения других данных. Примеры контейнеров: объекты класса *TCollection* библиотеки **Turbo Vision** и объекты контейнерных классов библиотеки **STL** в **C++** (массивы, списки, очереди).

В отличие от контейнера мы понимаем группу как класс, который не только хранит объекты других классов, но и обладает собственными свойствами, не вытекающими из свойств его элементов.

Группа дает второй вид иерархии (первый вид – *иерархия классов*, построенная на основе наследования) – *иерархию объектов* (иерархию типа *целое/часть*), построенную на основе агрегации.

Реализовать группу можно несколькими способами.

1. Класс “группа” содержит поля данных объектного типа. Таким образом, объект “группа” в качестве данных содержит либо непосредственно свои элементы, либо указатели на них:

```
Type
    TWindowDialog=object(TGroup)
    Input1          :TInputLine;
    Edit1           :TEdit;
    Button1         :TButton;
{другие поля и методы}
end;
```

Такой способ реализации группы используется в **Delphi.**

2. Группа содержит поле *last* типа *PObject*, которое указывает на начало связанного списка объектов, включенных в группу. В этом случае объекты должны иметь поле *next* типа *PObject,* указывающее на следующий элемент в списке.

3. Создается связанный список записей типа *TItem:*

```
Type
    PItem=^TItem;
    TItem=record
            next    :PItem;
            item    :PObject;
    end;
```

Поле *item* указывает на объект, включенный в группу. Группа содержит поле *last* типа *PItem*, которое указывает на начало связанного списка записей типа TItem.

Если необходим доступ элементов группы к ее полям и методам, объект типа *TObject* должен иметь поле *owner* типа *TGroup*, которое указывает на собственника этого элемента.

Методы группы.

Имеется два метода, которые необходимы для функционирования группы:

1) P*rocedure Insert(p:PObject);*

Вставляет элемент в группу.

2) *Procedure Show;*

Позволяет просмотреть группу.

Кроме этого, группа может содержать следующие методы:

1) Function Empty:boolean;

Показывает, есть ли хотя бы один элемент в группе.

2) Procedure Delete (p:PObject);

Удаляет элемент из группы, но сохраняет его в памяти.

3) Procedure DelDisp (Var p:PObject);

Удаляет элемент из группы и из памяти.

- **Итератор.** Итераторы позволяют выполнять некоторые действия для каждого элемента определенного набора данных.

***For** all элементов набора **do** действия*

Такой цикл мог бы быть выполнен для всего набора, например, чтобы напечатать все элементы набора. Или мог бы искать некоторый элемент, который удовлетворяет определенному условию, и в этом случае такой цикл может закончиться, как только будет найден требуемый элемент.

Мы будем рассматривать итераторы как специальные методы класса-группы, позволяющие выполнять некоторые действия для всех объектов, включенных в группу. Примером итератора является метод *Show*.

Нам бы хотелось иметь такой итератор, который позволял бы выполнять над всеми элементами группы действия заданные не одним из методов объекта, а произвольной процедурой или функцией пользователя. Такой

итератор можно реализовать, если эту процедуру(функцию) передавать ему через параметр процедурного типа.

Type TProc=procedure(p:PObject);

Метод-итератор может объявляться таким образом:

Procedure ForEach(action:TProc);

Передаваемая итератору процедура (функция) должна быть откомпилирована с атрибутом ***far:***

Procedure MyProc (p:PObject);far;

Метод-итератор можно сделать более удобным, если передавать ему указатель на процедуру:

Procedure ForEach (action : pointer);

- **Динамическая идентификация типов.** Определение типа объекта во время выполнения программы. Необходимость в этом может возникнуть, когда необходимо определить, какой объект содержит полиморфная переменная.

Например, группа содержит объекты различных классов и необходимо выполнить некоторые действия только для объектов определенного класса. Тогда в итераторе мы должны распознавать тип очередного объекта.

В *Borland Pascal* для динамической идентификации типа используется функция ***TypeOf.***

Function TypeOf(x):pointer;

Функция возвращает указатель на **VMT** объектного типа.

X-идентификатор класса или экземпляра класса.

Используем TypeOf для проверки принадлежности указателя на полиморфный объект определенному классу.

Var p:PObject;

.....

if TypeOf(p^)=TypeOf(TCircle) then {это окружность}

Порядок выполнения работы.

1. Дополнить иерархию классов лабораторной работы № 2 классами “группа”.

Например, для предметной области ФАКУЛЬТЕТ можно предложить классы "*факультет*", "*студенческая группа*", "*кафедра*". Рекомендуется создать абстрактный класс – "*подразделение*", который будет предком всех групп и абстрактный класс *TObject*, находящийся во главе всей иерархии.

2. Написать для класса-группы метод-итератор.

3. Написать процедуру или функции, которая выполняется для всех объектов, входящих в группу.

4. Написать демонстрационную программу, в которой создаются, показываются и разрушаются объекты-группы, а также демонстрируется использование итератора.

Методические указания.1.

Предметная область задается преподавателем или выбирается студентом самостоятельно. Для студентов заочного отделения эта область должна быть связана с местом работы студента.

2. Пример добавленных классов:

```
PObject=^TObject; {абстрактный класс}
TObject=object
 Constructor Init;
 Destructor Done;virtual;
Procedure Show;virtual;
end;

PDepartment=^TDepartment;
TDepartment=object(TObject) {абстрактный класс-группа}
 name:string;        {наименование}
 head:PPerson;     {руководитель}
 last:PItem; {указатель на начало связанного списка записей типа TItem }
 Constructor Init(Name1:string;Head1:PPerson);
 Destructor Done;virtual;
 Function GetName:string;;
 Function GetHead:PPerson;
 Procedure SetName(NewName:string);
 Procedure SetHead(NewHead:PPerson);
```

```
 Procedure Insert(p:PObject); {вставить объект в группу}
end;

PFaculty=^TFaculty; {факультет}
TFaculty=object(TDepartment)
 counte:word;{ количество сотрудников на факультете  }
 counts:word;{количество студентов на факультете }
                                  Constructor                    Init
(name1:string;head1:PPerson;counte1:word;counts1:word);
 Function GetCountE:byte;
 Function GetCountS:byte;
 Procedure SetCountE(NewCountE:byte);
 Procedure SetCountS(NewCountS:byte);
 Procedure Show;virtual;
end;

PStudentGroup=^TStudentGroup ;{студенческая группа}
TStudentGroup=object(TDepartment)
 count:byte;{ количество студентов в группе }
 speciality:string;{специальность }
 Constructor Init(name1:string;head1:PPerson;
          count1:byte;speciality1:string);
 Function GetCount:byte;
 Function GetSpec:string;
 Procedure SetCount(NewCount:byte);
 Procedure SetSpec(NewSpec:string);
 Procedure Show;virtual;
end;
```

3. Для включения объектов в группу использовать третий способ (см. п. “Основные теоретические сведения”).

4. факультет создается следующим образом:

а) создается пустой ФАКУЛЬТЕТ;

б) создается пустая КАФЕДРА;

в) создаются ПРЕПОДАВАТЕЛИ и включаются в КАФЕДРУ;

г) кАФЕДРА включается в ФАКУЛЬТЕТ;

д) создается пустая ГРУППА;

е) создаются СТУДЕНТЫ и включаются в ГРУППУ;

ж) ГРУППА включается в ФАКУЛЬТЕТ.

5. Удаляется ФАКУЛЬТЕТ в обратном порядке.

6. Метод-итератор и класс выбирается на основе контрольной работы № 1.

Например, для класса *TStudentGroup* может быть предложен итератор

Procedure TStudentGroup.ForEach(action:pointer;parametr:real);

где *action* – указатель на процедуру или функцию, которая должна быть выполнена для всех объектов, включенных в группу (в данном случае для всех СТУДЕНТОВ), *parametr* – передаваемая процедуре дополнительная информация.

В качестве передаваемой методу функции может быть предложена, например, такая: “Вывести список студентов, имеющих рейтинг не ниже заданного”:

```
Procedure MyProc(p:PObject,rate:real);far;
Begin
if PStudent(p)^.GetRating>=rate then writeln(PStudent(p)^.GetName);
End;
```

Если *pg* – указатель на объект *TStudentGroup*, то вызов итератора будет иметь следующий вид:

```
pg^.ForEach(@MyProc,45); {список студентов, чей рейтинг выше 45}
```

7. Определение классов (объектных типов), процедур и функций поместить в модуль *L02UXXX*, где *XXX*- аббревиатура фамилии, имени и отчества, например, Иванов Петр Григорьевич – *IPG.*

8. Демонстрационную программу поместить в файл *L02XXX.PAS.*

Содержание отчета.

1. Титульный лист.
2. Постановка задачи.
3. Иерархия классов.
4. Иерархия объектов.

5. Определение классов (добавленных или измененных по сравнению с лабораторной работой № 2).

6. Реализация для одного не абстрактного класса-группы всех методов.

7. Реализация итератора.

8. Реализация передаваемой итератору процедуры или функции.

9. Листинг демонстрационной программы.

Лабораторная работа № 4
ОБРАБОТКА СОБЫТИЙ

Цель. Получить практические навыки разработки объектно-ориентированной программы, управляемой событиями.

Краткие теоретические сведения.

- **Объектно-ориентированная программа как программа, управляемая событиями.** При использовании ООП все объекты являются в некотором смысле обособленными друг от друга и возникают определенные трудности в передаче информации от объекта к объекту. В ООП для передачи информации между объектами используется механизм обработки событий.

События лучше всего представить себе как пакеты информации, которыми обмениваются объекты и которые создаются объектно-ориентированной средой в ответ на те или иные действия пользователя. Нажатие на клавишу или манипуляция мышью порождает событие, которое передается по цепочке объектов, пока не найдется объект, знающий как обрабатывать это событие. Для того чтобы событие могло передаваться от объекта к объекту, все объекты программы должны быть объединены в группу. Отсюда следует, что прикладная программа должна быть объектом–группой, в которую должны быть включены все объекты, используемые в программе.

Таким образом, объектно-ориентированная программа – это программа, управляемая событиями. События сами по себе не производят никаких действий в программе, но в ответ на событие могут создаваться новые объекты, модифицироваться или уничтожаться существующие, что и приводит к изменению состояния программы. Иными словами, все действия по обработке данных реализуются объектами, а события лишь управляют их работой.

Принцип независимости обработки от процесса создания объектов приводит к появлению двух параллельных процессов в рамках одной программы: процесса создания объектов и процесса обработки данных.

Это означает, что действия по созданию, например, интерактивных элементов программы (окон, меню и пр.) можно осуществлять, не заботясь о действиях пользователя, которые будут связаны с ними.

И, наоборот, мы можем разрабатывать части программы, ответственные за обработку действий пользователя, не связывая эти части с созданием нужных интерактивных элементов.

- **Событие.** Событие с точки зрения языка Паскаль – это запись, отдельные поля которой характеризуют те или иные свойства передаваемой информации, например:

```
Type
TEvent = record
What : word;
case What of
evNothing : ( );
evMouse : ({поля}
evKeyDown : ({поля});
evMessage : (
command : word;
Info : integer)  end;
```

Запись TEvent состоит из двух частей. Первая (What) задает тип события, определяющий источник данного события. Вторая задает информацию, передаваемую с событием. Для разных типов событий содержание информации различно.

evNothing – это пустое событие, которое означает, что ничего делать не надо. Полю what присваивается значение evNothing, когда событие обработано каким-либо объектом.

evMouse – событие от «мыши».

evKeyDown –событие от клавиатуры.

Для события от объекта (evMassage) задаются два параметра:

command – код команды, который необходимо выполнить при появлении данного события;

Info – передаваемая с событием информация.

- **Методы обработки событий.** Следующие методы необходимы для организации обработки событий.

GetEvent – формирование события;

Execute – реализует главный цикл обработки событий. Он постоянно получает событие путем вызова GetEvent и обрабатывает их с помощью HandleEvent. Этот цикл завершается, когда поступит событие "конец".

HandleEvent – обработчик событий. Обрабатывает каждое событие нужным для него образом. Если объект должен обрабатывать определенное событие (сообщение), то его метод HandleEvent должен распознавать это событие и реагировать на него должным образом. Событие может распознаваться, например, по коду команды (поле command).

ClearEvent – очищает событие, когда оно обработано, чтобы оно не обрабатывалось далее.

- **Обработчик событий (метод HandleEvent)**. Получив событие (запись типа TEvent), обработчик событий для класса TObject обрабатывает его по следующей схеме:

```
Procedure TObject. HandleEvent (Var Event: TEvent);
Begin
  if Event. What = evMessage then begin
    case Event. Command of
      cmCommand1: Action1 (Event. Info);
      cmCommand2: Action2 (Event. Info);
  …
  else exit end;
  ClearEvent (Event) end
End;
```

Обработчик событий группы вначале обрабатывает команды группы, а затем, если событие не обработано, передает его своим элементам, вызывая их обработчики событий.

```
Procedure TGroup.HandleEvent (Var Event: TEvent);
Begin
  if Event. What = evMessage then begin
   case Event. Command of
    {обработка команд объекта-группы}
  else begin
    {получить доступ к первому элементу группы}
```

```
while (Event. What <> evNothing and {просмотрены не все элементы} do begin
    {вызвать HandleEvent текущего элемента}
    {перейти к следующему элементу группы}
  end;
  exit;
  end end;
  ClearEvent (Event);
  end;
End;
```

- **Метод ClearEvent – очистка события.** ClearEvent очищает событие, присваивая полю Event.What значение evNothing.

- **Главный цикл обработки событий (метод Execute)**. Главный цикл обработки событий реализуется в методе Execute главной группы – объекта “прикладная программа” по следующей схеме:

```
Procedure TApp.Execute;
Var Event: TEvent;
Begin
  repeat
   EndState := 0;
   GetEvent (Event);
   HandleEvent (Event);
  until Valid;
End;
```

Метод HandleEvent программы обрабатывает событие “конец работы”, вызывая метод EndExec.EndExec изменяет значение private – переменной EndState. Значение этой переменной проверяет метод-функция Valid, возвращающая значение true, если “конец работы”. Такой несколько сложный способ завершения работы программы связан с тем, что в активном состоянии могут находиться несколько элементов группы. Тогда метод Valid группы, вызывая методы Valid своих подэлементов, возвратит true, если все они возвратят true. Это гарантирует, что программа завершит свою работу, когда завершат работу все ее элементы.

- **Пример обработки событий**. Рассмотрим простейший калькулятор, воспринимающий команды в командной строке. Здесь приводится упрощенный вариант. Рабочий вариант приведен в “Приложении”.

Формат команды:

знак параметр

Знаки +, –, *, /, =, ?, q

Параметр – целое число

```
Const       evNothing = $0000;
            evMessage = $0100;
            cmSet = 1;    {занести число}
            cmGet = 2;    {посмотреть значение}
            cmAdd = 3;    {добавить}
```

и т.д. cmQuit = 100; {выход}

Запись-событие:

```
Type TEvent = record
       what: word;
       case what of
         evNothing: ( );
         evMessage: (
            command: word;
            A: integer;) end;
```

Объект-калькулятор, работающий с целыми числами: PInt = ^TInt;

```
TInt = object
  private
    EndState: byte;
  public
    x: integer;
  Constructor Init (x1: integer);
  Destructor Done; virtual;
  Procedure GetEvent (Var Event: TEvent); virtual;
  Procedure Execute; virtual;
  Procedure HandleEvent (Var Event: TEvent); virtual;
  Procedure ClearEvent (Var Event: TEvent); virtual;
```

```
  Function Valid: boolean;
  Procedure EndExec;
  Function GetX: integer;
  Procedure SetX (newX: integer);
  Procedure AddY (Y: integer);
…
end;
```

Рассмотрим возможную реализацию основных методов:

```
Procedure TInit. GetEvent;
Const OpInt: set of char = ['+', '-', '*', '/', '=', '?', 'q'];
Var s: string;
     code: char;
Begin
  write('>') ;
  readln(s); code := s[1];
  with Event do
    if code in OpInt then begin
      what:= evMessage;
      case code of
        '+': command:= cmAdd;
        …
        'q': command;= cmQuit;
      end;
        {выделить второй параметр, перевести его в тип integer и присвоить полю A}
    end
    else what:= evNothing
End.

Procedure TInt.Execute;
Var E: TEvent;
Begin
  repeat
    EndState:= 0;
    GetEvent (E);
```

```
    HandleEvent (E);
  until Valid;
End;

Procedure TInt.HandleEvent;
Begin
  if Event. what = evMessage then begin
    case Event. command of
      cmAdd: AddY (Event. A);
      …
      cmQuit: EndExec;
    else exit end;
  ClearEvent (Event) end
End;

Function TInt.Valid;
Begin
  if EndState = 0 then Valid:= false
  else Valid:= true
End;

Procedure TInt.ClearEvent;
Begin
  Event. what:= evNothing
End;

Procedure TInt.EndExec;
Begin
  EndState:= 1
End;

Procedure TInt.AddY;
Begin
  X:= X+Y
End;
```

и т.д.

```
Var MyApp: TInt;
Begin
  MyApp. Init(0);
  MyApp.Execute;
End;
```

Основное содержание работы.

Написать интерактивную программу, выполняющую команды, вводимые пользователем с клавиатуры.

Порядок выполнения работы.

1. Разобрать пример, представленный в “Приложении”. Ответить на следующие вопросы:

а) Какова здесь иерархия классов?

б) Какова здесь иерархия объектов?

в) Как КАЛЬКУЛЯТОРУ передаются аргументы операции? Где они хранятся? Каким образом получают к ним доступ устройства СЛОЖЕНИЯ, ВЫЧИТАНИЯ и т.д.?

г) Как обрабатываются события группой?

д) Каковы все маршруты события TEvent?

е) Как выполняются HandleEvent всех классов?

2. Для выбранной группы объектов определить перечень операций, которые должны выполняться по командам пользователя. Например, команды: **сложить, вычесть, умножить, разделить** для объекта **КАЛЬКУЛЯТОР** (см. “Приложение”).

3. Определить вид командной строки <код_операции><параметры>. Решить вопросы: Как кодируются операции? Какие передаются параметры?

4. Определить иерархию объектов. Если необходимо, добавить новые объекты (группы объектов).

5. Определить иерархию классов. Если необходимо, добавить новые классы.

6. Определить какой объект в программе играет роль приложения. В случае необходимости добавить в иерархию классов класс ***TApp.*** Решить, в каком классе будет метод ***Execute (***или ***Run)***, организующий главный цикл обработки событий.

7. Определить и реализовать необходимые для обработки событий методы.

8. Написать основную программу.

Методические указания.

1. Группу, для которой организуется обработка событий, рекомендуется выбрать группу из лабораторной работы № 3. Например, объект ФАКУЛЬТЕТ или КАЛЬКУЛЯТОР. Однако эти группы не выбираются, так как они уже рассмотрены в методических указаниях.

2. Определение и реализацию классов поместить в модуль.

3. Для констант, связанных с командами, использовать мнемонические имена *cmXXXX.*

4. Пример модуля для класса-группы КАЛЬКУЛЯТОР приведен в “Приложении”). Это образец для вас.

5. Основная программа имеет примерно следующий вид:

```
Program Prog01;
Uses Prog01U;
Var ACalc:TCalc;
Begin
ACalc.Init;
ACalc.Run;
ACalc.Done;
End
```

Содержание отчета.

1. Титульный лист.

2. Постановка задачи.

3. Схема иерархии классов.

4. Схема иерархии объектов.

5. Описание маршрута, который проходит событие *TEvent* от формирования до очистки.

6. Определение классов (добавленных или измененных по сравнению с работой № 3).

7. Реализация методов обработки событий *GetEvent, Exicute, EndExec, Valid*.

8. Реализация всех методов *HandleEvent*.

9. Листинг основной программы.

Приложение.

```
{ Объект ПРОСТОЙ КАЛЬКУЛЯТОР выполняет сложение, вычитание, умножение, деление вещественных чисел}
{Модуль: Определение и реализация классов}
Unit Prog01U;

INTERFACE
Const
   evNothing=$0000;
   evMessage=$0100;
   cmSet   = 1; {занести число в сумматор}
   cmGet   = 2; {посмотреть число в сумматор}
   cmAdd  = 3; {сложить}
   cmSub  = 4;  {вычесть}
   cmMult = 5; {умножить}
   cmDel   = 6; {делить}
   cmHelp = 7; {помощь}
   cmQuit=100;{выход}

Type TEvent=record
    What:word;
    case word of
    evNothing:();
    evMessage:(
    command:word; {команда}
    A:real;) {параметр, передаваемый с командой}
    end;

  {**КЛАССЫ**}
```

```
POject=^TObject;
PScheme=^TScheme;
PDevice=^TDevice;
PCalc=^TCalc;
PReg=^TReg;

PItem=^TItem;
TItem=record
next:PItem;
ptr:PObject;
end;

TObject=object {абстрактный класс}
private
owner:PScheme; {указатель на владельца}
public
Constructor Init;
Destructor Done;virtual;
Procedure HandleEvent(Var Event:TEvent);virtual;

Procedure ClearEvent(Var Event:TEvent);virtual;
end;

TScheme=object(TObject) {абстрактная группа}
private
last:PItem;
public
Constructor Init;
Destructor Done;virtual;
Procedure Insert(p:PObject);virtual;
Procedure HandleEvent(Var Event:TEvent);virtual;
end;

    TDevice=object(TScheme) {абстрактная группа – устройство управления}
private
```

```
EndState:byte;
public
Procedure GetEvent(Var Event:TEvent);virtual;
Procedure Run;virtual;
Function Valid:boolean;virtual;
Procedure EndExec;virtual;
end;

TCalc=object(TDevice) {калькулятор}
private
sum,r1:PReg;  {указатели на регистр и сумматор}
public
Constructor Init;
Procedure HandleEvent(Var Event:TEvent);virtual;
Procedure GetEvent(Var Event:TEvent);virtual;
Procedure Run;virtual;
Procedure OutPutX;
Procedure InPutX(x1:real);
Procedure Help;virtual;
end;

TReg=object(TObject) {регистровая память}
private
x:real;
public
Constructor Init;
Function GetX:real;
Procedure SetX(NewX:real);
end;

PAdd=^TAdd;
TAdd=object(TObject) {устройство сложения}
Procedure HandleEvent(Var Event:TEvent);virtual;
Procedure Add;virtual;
end;
```

```
    PSub=^TSub;
    TSub=object(TObject) {устройство вычитания}
    Procedure HandleEvent(Var Event:TEvent);virtual;
    Procedure Sub;virtual;
    end;

    PMult=^TMult;
    TMult=object(TObject) {устройство умножения}
    Procedure HandleEvent(Var Event:TEvent);virtual;
    Procedure Mult;virtual;
    end;

    PDel=^TDel;
    TDel=object(TObject) {устройство деления}
    Procedure HandleEvent(Var Event:TEvent);virtual;
    Procedure Del;virtual;
    end;

IMPLEMENTATION
{приведена реализация только некоторых методов}
Constructor TObject.Init;
begin
owner:=nil;
end;

Procedure  TObject.ClearEvent(Var Event:TEvent);
begin
Event.What:=evNothing;
end;


Constructor TScheme.Init;
Begin
inherited Init;
last:=nil;
End;
```

```
Destructor TScheme.Done;
Var p,r:PItem;
begin
Для всех элементов группы выполнить:
    begin
    dispose(r^.ptr,Done);
    dispose(p);
    end;
end;

Procedure TScheme.HandleEvent(Var Event:TEvent);
Var r:PItem;
Begin
inherited HandleEvent(Event);
if Event.What=evMessage then
 begin
 {получить доступ к первому элементу}
  while (Event.What<>evNothing)and(r<>nil) do
  begin
  {вызвать HandleEvent  очередного элемента}
  {перейти к следующему элементу}
  end
  end
  End;

Procedure TDevice.Run;
Var E:TEvent;
Begin
repeat
 EndState:=0;
 GetEvent(E);
 HandleEvent(E);
until Valid;
End;
Function TDevice.Valid:boolean;
```

```
Begin
if EndState=0
  then Valid:=false
  else Valid:=true;
End;

Procedure TDevice.EndExec;
begin
EndState:=1;
end;

Constructor TCalc.Init;
Var p:PObject;
begin
inherited Init;
sum:=new(PReg,Init);
r1:=new(PReg,Init);
{вставить в КАЛЬКУЛЯТОР элементы TAdd,TSub,TMult,TDel }
end;

Procedure  TCalc.HandleEvent(Var Event:TEvent);
Begin
if Event.What=evMessage then
{обработка команд TCalc}
begin
 case Event.Command of
 cmQuit:EndExec;
 cmGet:OutPutX;
 cmSet:InPutX(Event.A);
 cmHelp:Help
 else {обработка команд устройств }
     begin
     inherited HandleEvent(Event);
     exit;
     end;
end;{case}
```

```
ClearEvent(Event)
end;{if then}
End;

Procedure TCalc.Run;
Begin
writeln('Вводите команды. “q” – выход,“h” – помощь);
inherited Run;
End;

Procedure TCalc.OutPutX;
begin
writeln(sum^.GetX:0:8)
end;
Procedure TCalc.InPutX(x1:real);
begin
sum^.SetX(x1);
end;

Procedure TAdd.HandleEvent(Var Event:TEvent);
begin
if Event.What=evMessage
 then begin
 case Event.Command of
 cmAdd:begin PCalc(owner)^.r1^.SetX(Event.A); Add;end
 else exit end;
ClearEvent(Event) end
end;

Procedure TAdd.Add;
begin
PCalc(owner)^.sum^.SetX(PCalc(owner)^.sum^.GetX+PCalc(owner)
^.r1^.GetX)
end
END.
```

Итоговая лабораторная работа № 1
ИЕРАРХИЯ КЛАССОВ И ОБЪЕКТОВ

Цель. Получить практические навыки создания иерархий объектов и выполнения запросов с использованием итераторов.

Порядок выполнения работы.

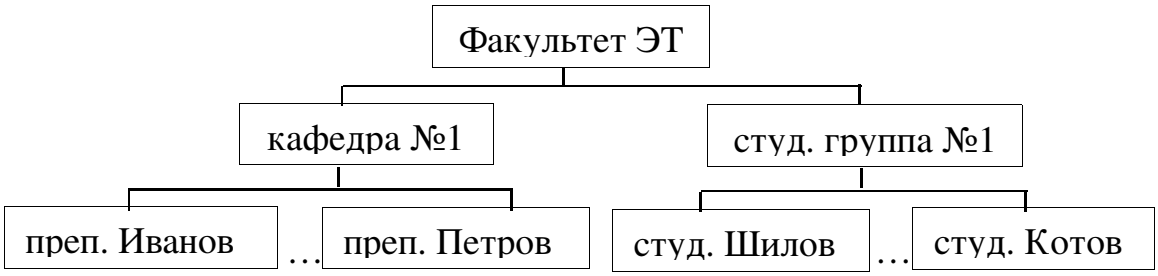
1. Создать иерархию объектов определенной предметной области.
2. На основе иерархии объектов определить иерархию классов.
3. Реализовать классы. Определения и реализации классов поместить в модуль.
4. Определить, какие запросы должна выполнять программа.
5. Написать методы-итераторы.

6. Написать процедуры или функции, передаваемые итераторам для выполнения запросов.
7. Написать демонстрационную программу, в которой создаются, показываются и разрушаются группы, а также демонстрируется выполнение запросов.

Методические указания.

1. Предметная область выбирается студентом самостоятельно или выдается преподавателем. Для студентов заочного отделения эта область должна быть ***связана с местом работы*** студента.
2. Для иерархии объектов и классов могут быть использованы материалы лабораторных работ № 2 и 3.

Пример иерархии объектов для предметной области “Высшее учебное заведение”:



Количество объектов, включенных в каждую группу на самом нижнем уровне, должно быть не менее 5.

3. Запросы выполняются путем вызова итератора для головного объекта (в нашем примере "факультет") и передачи ему необходимой процедуры или функции и, если необходимо, других параметров.

Например:

pf:PGroup;

pf:=new(PFac,Init("ЭТФ", <другие параметры конструктора>));

pf^.ForEach(@MyFunc,<другие параметры итератора>)

4. Примеры запросов:

– вывести список студентов, чей рейтинг не менее заданного (заданный рейтинг передается итератору через параметр);

– повысить зарплату всем доцентам на 20% ("доцент" – значение поля post объекта класса TTeacher);

– вывести список ассистентов заданной кафедры (наименование кафедры – значение поля name объекта класса TSubFac – передается итератору через параметр).

Количество запросов, выполняемых в демонстрационной программе должно быть не менее 3.

5. Демонстрационная программа не должна быть диалоговой – объекты создаются, помещаются в группы, удаляются, выполняют итераторы непосредственно в тексте программы.

Содержание отчета.

1. Титульный лист.

2. Постановка задачи.
3. Иерархия классов.
4. Иерархия объектов.
5. Реализация всех итераторов.
6. Описание запросов.
7. Реализация функций (процедур), передаваемых итератору.
8. Листинг демонстрационной программы.
9. Результаты выполнения программы.

Итоговая лабораторная работа № 2
ПРОГРАММА, УПРАВЛЯЕМАЯ СОБЫТИЯМИ

Цель. Закрепить знания, полученные при выполнении предыдущих лабораторных работ. Получить практические навыки создания диалоговой объектно-ориентированной программы, обрабатывающей события-запросы.

Содержание работы

Работа основана на лабораторной работе № 4 и итоговой лабораторной работе № 1. Отличие от итоговой работы № 1 состоит только в том, что запросы вводятся в диалоге в виде команд, как в лабораторной работе № 4.

Содержание отчета.

1. Титульный лист.
2. Постановка задачи.
3. Иерархия классов.
4. Иерархия объектов.
5. Описание запросов.
6. Реализация всех обработчиков HandleEvent с объяснением их выполнения.
7. Листинг демонстрационной программы.
8. Тестовые данные и результаты тестирования программы.

Лабораторная работа № 5
ПРОСМОТР ТЕКСТОВОГО ФАЙЛА В ОКНЕ СО СКРОЛЛИНГОМ

Цель. Получить практические навыки создания простой интерактивной программы с использованием библиотеки Turbo Vision в среде Borland Pascal.

Основное содержание работы.

Написать и отладить объектно-ориентированную программу для просмотра текстового файла. Программа должна содержать меню (объект **TMenuBar**) и строку состояния (объект **TStatusLine**). Для просмотра файла использовать объект **TWindow**, для скроллинга – **TScroller**. Написать обработчик событий программы (метод **HandleEvent**).

Краткие теоретические сведения.

•**Программирование в Turbo Vision.** Сущность программирования в TV – это проектирование того, что видит пользователь на экране, и того, что делать, когда получена та или иная команда. В связи с этим программирование делится на две относительно независимые части:

1) проектирование элементов экрана, с которыми взаимодействует пользователь. В TV – это отображаемые объекты;

2) проектирование реакции программы на те или иные действия пользователя или на сообщения, посылаемые объектами. В TV – это обработка событий.

Библиотека Turbo Vision содержит достаточное число отображаемых объектов для создания стандартного интерфейса прикладной программы. Задача программиста состоит в выборе необходимых объектов и перекрытия некоторых методов, применительно к требованиям конкретной прикладной программы.

Интерфейс интерактивной программы содержит обычно панель экрана (**TDeskTop**), меню (**TMenuView**) и строку состояния (**TStatusLine**). Эти элементы создаются и вставляются в программу при вызове метода TProgram.Init (MyApp.Init). На панели экрана размещаются окна (**TWindow**), в

том числе и диалоговые (**TDialog**) и различные элементы управления – кнопки, кластеры, строки ввода.

Программа на Turbo Vision выглядит следующим образом:

```
Type TMyApp=object(TApplication)
Procedure HandleEvent(Var Event:TEvent);virtual;
Procedure InitMenuBar;virtual;
Procedure InitStatusLine;virtual;
{объявление других методов, если они необходимы}
end;
{объявление других классов и их методов}
{определение методов}
Var MyApp:TMyApp;
begin
MyApp.Init;
MyApp.Run;
MyApp.Done;
end.
```

- **Обработка событий.** Программы, работающие в среде TV – это программы, управляемые событиями. События лучше всего представить себе как небольшие пакеты информации, которыми обмениваются отображаемые элементы и которые создаются средой TV в ответ на те или иные действия пользователя. События сами по себе не производят никаких действий в программе, но в ответ на событие создаются новые объекты, модифицируются или уничтожаются существующие, что и приводит к изменению состояния программы.

Программы, управляемые событиями, имеют центральный механизм диспетчеризации событий так, что программа не должна заботиться о получении ввода и о решении, что делать с ним. Программа просто ждет, когда центральный диспетчер обработает их ввод.

Почти вся работа программы в TV выполняется внутри метода **Run**, который наследуется от TApplication. Run состоит главным образом из цикла repeat...until, общая схема которого:

```
repeat
Получить событие
```

Обработать событие
until Quit

С точки зрения языка Pascal событие – это запись, отдельные поля которой характеризуют те или иные свойства передаваемой информации.

```
TEvent = record
  What: Word;
  case Word of
   evNothing: ();
   evMouse: (
    Buttons: Byte;
    Double: Boolean;
    Where: TPoint);
   evKeyDown: (
    case Integer of
     0: (KeyCode: Word);
     1: (CharCode: Char;
  ScanCode: Byte));
   evMessage: (
    Command: Word;
    case Word of
     0: (InfoPtr: Pointer);
     1: (InfoLong: Longint);
     2: (InfoWord: Word);
     3: (InfoInt: Integer);
     4: (InfoByte: Byte);
     5: (InfoChar: Char));
 end;
```

Первая часть записи (поле What) задает тип события, определяющий источник данного события, вторая часть записи задает информацию, передаваемую с событием.

События формируются с помощью метода **TView.GetEvent**. Для стандартных случаев они формируются автоматически с помощью имеющихся средств Turbo Vision. Обработка же событий, как правило, сугубо индивидуальна для каждой прикладной программы.

Обработка событий всегда начинается с текущего модального объекта, после чего, если необходимо, оно передается тем или иным подэлементам этого модального объекта. В зависимости от последовательности обработки событий объектами все события можно разбить на три группы:

I – события от “мыши”(позиционированные события).

II – события от клавиатуры и команды.

III – события-сообщения.

Обрабатываются события методом **HandleEvent.**

- **Обработчик событий – метод HandleEvent.** Это центральный метод, через который реализуется вся обработка событий TV. Все отображаемые объекты имеют этот метод.

Procedure TView.HandleEvent(Var Event:TEvent);virtual;

Обрабатывает событие *evMouseDown*, выбирая объект. Почти всегда перекрывается в потомках.

Procedure TProgram.HandleEvent(Var Event:TEvent);virtual;

Обрабатывает команду *cmQuit* и события от клавиш *Alt1..Alt9.*

Procedure TApplication.HandleEvent(Var Event:TEvent);virtual;

Обрабатывает команды *cmTile,cmCascade,CmDosShell.*

Procedure TGroup.HandleEvent(Var Event:TEvent);virtual;

Группа обрабатывает события, передавая их в методы **HandleEvent** своих подэлементов. Действительный маршрут зависит от группы событий.

Если объект, наследуемый от стандартного объекта TV, должен обрабатывать события, не обрабатываемые стандартным объектом, перекрываемый метод HandleEvent записывается по следующей схеме:

```
TMyObject=object(TBaseObject)
...
Procedure HandleEvent(Var Event:TEvent);virtual;
...
end;

Procedure TMyObject.HandleEvent(Var Event:TEvent);
begin
inherited HandleEvent(Event);
if Event.What=evCommand then
```

```
 case EventCommand of
cmCommand1: Proc1;
cmCommand2: Proc2;
...
else exit
end;
ClearEvent(Event)
end;
```

Если событие не обработано, осуществляется выход по Exit и событие направляется следующему элементу в соответствии с маршрутом обработки событий. Если событие обработано, то оно “очищается” вызовом метода ClearEvent.

Если наследуемый объект должен обрабатывать событие базового объекта отличным от базового образом, то перекрываемый метод HandleEvent записывается по следующей схеме:

```
Procedure TMyObject.HandleEvent(Var Event:TEvent);
begin
if Event.What=evCommand then
 case EventCommand of
cmCommand1: Proc1;
cmCommand2: Proc2;
...
else begin
      inherited HandleEvent(Event);
      exit
     end
end;
ClearEvent(Event)
end;
```

•**Команды.** События от объектов имеют в качестве одного из параметров поле *Command:word,* задающее код команды, которую необходимо выполнить при обработке этого события.

Все используемые в TV команды делятся на 4 группы:

1) команды, зарезервированные за системой и которые можно маскировать и демаскировать, – коды с 0 по 99;

2) команды, которые вводит программист и которые можно маскировать и демаскировать, – коды с 100 по 255;

3) команды, зарезервированные за системой, но которые нельзя маскировать и демаскировать, – коды с 256 по 999;

4) команды, которые вводит программист, но которые нельзя маскировать и демаскировать, – коды с 999 по 65535.

Команды, зарезервированные за системой, имеют стандартные имена (cmXXXX), и TV определенным образом реагирует на них.

•**Создание меню.** В TV любое меню составляется из 3 элементов:

1) собственно элемент меню (определяет команду, которую следует выполнить при выборе этого элемента);

2) подменю, при выборе которого на экране раскрывается соответствующий пункт исходного меню;

3) разделительная линия, которая имеет чисто декоративное назначение и позволяет отделить в подменю те или иные группы элементов друг от друга.

Для создания меню используется объект класса **TMenuBar**, который наследуется от абстрактного класса **TMenuView**.

Для создания элемента меню, определяющего выполняемую команду, используется функция **NewItem.**

Для создания элемента меню, определяющего подменю, используется функция **NewSubMenu.**

Для создания элемента меню, определяющего разделительную линию, используется функция **NewLine.**

Связанные друг с другом элементы одного уровня объединяются вместе и образуют структуру **TMenu.** Для создания такой структуры используется функция **NewMenu.**

Создается меню вызовом метода **Procedure TProgram.InitMenuBar**, который следует перекрыть в конкретной программе, например:

```
Procedure TMyApp.InitMenuBar;
Var r:TRect;
```

```
Begin
GetExtent(r);
r.b.y:=r.a.y+1;
 MenuBar:=New(PMenuBar,Init(r,NewMenu(
                NewSubMenu('~F~ile',hcNoContext,
                 NewMenu(
                 NewItem('~O~pen','F3',kb F3,cmOpenFile,hcNoContext,
                 NewItem('~S~ave','F2',kbF2,cmSaveFile,hcNoContext,
                        NewLine(
                                                              NewItem
('~Q~uit','Alt+X',kbAltX,cmQuit,0,nil))))),
               NewSubMenu('~E~dit',hcNoContext,
                 NewMenu(
                 NewItem('~C~ut','Shift+Del',kbShiftDel,cmCut,0,
                                                              NewItem
('c~O~py','Ctrl+Ins',kbCtrlIns,cmCopy,0,nil))),
                              nil)))))
                             end;
```

Указатель на созданное меню помещается в переменную **MenuBar.**

•**Создание строки статуса.** Для формирования строки статуса необходимо создать объект **TStatusLine.** Этот объект создается методом **InitStatusLine**, который сохраняет указатель на него в глобальной переменной **StatusLine.** Метод InitStatusLine всегда перекрывается в потомках от TApplication. InitStatusLine создает объект TStatusLine, размещая его в динамической памяти процедурой New:

StatusLine:=New(PStatusLine,Init(аргументы));

В процедуре New используется конструктор Init объекта TStatusLine.

Constructor TStatusLine.Init(Var Bounds:TRect;ADefs:PStatusDef);

где *PStatusDef=^TStatusDef;*

```
   TStatusDef=record
                    next:PStatusDef;
                    Min,Max:Word;
                    Items:PStatusItem;
                    end;
```

Записи **TStatusDef** создаются функцией **NewStatusDef:**

Function NewStatusDef(AMin,Amax:Word;AItems:PStatusItem;ANext: PStatusDef):PStatusDef;

где *PStatusItem=^TStatusItem;*

```
TStatusItem=record
                next:PStatusItem;
                Text:PString;
                KeyCode:Word;
                Command:Word;
                end;
```

В свою очередь записи **TStatusItem** создаются функцией **NewStatusKey.**

Пример:

```
Procedure TMyApp.InitStatusLine;
Var r:TRect;
Begin
GetExtent(r);
r.a.y:=pred(r.b.y);
StatusLine:=New(PStatusLine,Init(r,NewStatusDef(0,$FFFF,
          NewStatusKey('Alt-X~ Выход',kbAltX,cmQuit,
          NewStatusKey('F2~ Закрыть',kbF2,cmSaveFile,
          NewStatusKey('F3~ Открыть',kbF3,cmOpenFile,
          NewStatusKey('F4~ Работа',kbF4,cmWork,
          NewStatusKey('F10~ Меню',kbF10,cmMenu,
          Nil))))),Nil)));
End;
```

•**Окна.** Окно в TV – это объект класса **TWindow**, владеющий специальной рамкой(**TFrame**) и, как правило, объектами **TScroller** и **TScrollBar**, которые позволяют осуществлять скроллинг не размещающейся в окне информации.

Как правило, в программе используется потомок TWindow, так как для конкретных окон необходимо переопределить метод Init для размеще-

ния в окне его компонентов и метод HandleEvent для соответствующей обработки событий, предназначенных окну.

Для того чтобы отобразить что-нибудь в окне (например, текст), необходимо поместить внутрь окна специальный отображаемый объект, который отвечал бы за содержимое окна. Для этой цели используется объект класса **TScroller.** Следует создать потомок TScroller и переопределить в нем метод **Draw** для того, чтобы наполнить поле скроллера соответствующей информацией. Имеются четыре метода класса TView, которые используются в методе Draw для размещения в скроллере текстовых строк: **WriteBuf, WriteLine, WriteStr, WriteChar.** Для копирования в буфер используются следующие глобальные процедуры: **MoveBuf, MoveStr, MoveChar, MoveCStr.**

Пример:

```
PInterior  =^TInterior;
TInterior  =object(TScroller)
 Constructor Init(Var Bounds:TRect;HS,VS:PScrollBar);
 Procedure Draw;Virtual;
 end;

Lines   :array[1..MaxLine] of string[LLines]; {массив для хранения прочитанных из файла  стр ок}
Constructor TInterior.Init(Var Bounds:Trect;HS,VS:PScrollBar);
Begin
inherited Init(Bounds,HS,VS);
ReadFile; {прочитать  в память текстовый файл}
GrowMode:=gfGrowHiX+gfGrowHiY;
SetLimit(LLines,Nlines)
End;

Procedure TInterior.Draw;
var
 Color: Byte;
 n,k: Integer;
 B: TDrawBuffer;
begin
```

```
  Color := GetColor(1);
  for n := 0 to Size.Y - 1 do
  begin
    MoveChar(B, ' ,'Color, Size.X);
    k := Delta.Y + n+1;
    MoveStr(B, Copy(Lines[k], Delta.X + 1, Size.X), Color);
    WriteLine(0, n, Size.X, 1, B);
  end;
end;
```

Порядок выполнения лабораторной работы.

1. Сформировать меню прикладной TV-программы. Горизонтальное меню должно содержать следующие пункты:

- Файл
- Работа

Меню **Файл** содержит следующие подпункты:

- Открыть файл
- Закрыть файл
- Сменить каталог
- Выход

При выборе пункта **Работа** просматривается открытый текстовый файл.

2. Сформировать строку статуса.

3. Отладить программу. Программа должна выполнять только навигацию по меню.

4. Дополнить программу обработчиком событий HandleEvent. В обработчике событий вместо вызова реальных процедур использовать заглушки. Отладить программу.

5. Определить необходимые методы класса.

TMyApp = object (TApplecation).

6. Отладить программу, используя заглушки для еще нереализованных процедур.

7. Включить в программу классы TMyWindow=object(TWindow) и TInterior=object(TScroller). Определить необходимые методы этих классов.

8. Отладить целиком всю программу.

Методические указания.

1. Для смены каталога использовать стандартный диалог TChDirDialog.
1. При открытии файл для ввода имени файла использовать стандартный диалог TFileDialog.
1. Пока файл не открыт, пункт Работа (просмотр файла) не доступен. Для управления доступом использовать методы DisableCommands и EnableCommands.
1. К скроллеру следует подключить полосы скроллинга. Для этого использовать объекты TScrollBar. Для создания этих объектов лучше использовать метод TWindow.StandardScrollBar.
1. Объект TMyWindow включают в рабочую область (TDeskTop) методом Insert.
1. Объект TInterior включают в окно также методом Insert.
1. При открытии файла строки считываются в массив типа string.

Содержание отчета.

1. Титульный лист.

2. Определение классов **TMyApp, TMyWindow, TInterior** с комментариями**.**

3. Реализация методов **TMyApp.HandleEvent, TMyApp.FileOpen** с комментариями**.**

4. Реализация метода **TMyWindow.Init** с комментариями**.**

5. Реализация методов **TInterior.Init, TInterior.Draw** с комментариями**.**

6. Описание процедуры просмотра текста в окне.

Лабораторная работа № 6
ДИАЛОГОВЫЕ ОКНА В ПРОГРАММАХ TURBO VISION

Цель: Получить практические навыки использования диалоговых окон в TV-программе.

Основное содержание работы.

Написать программу, создающую объекты пользовательского класса и помещающую их в линейный список. Предусмотреть диалог для ввода исходных данных, просмотра объектов, их корректировки и сохранения данных в файле.

Краткие теоретические сведения.

Все программы в среде Turbo Vision рассчитаны на диалоговый способ взаимодействия с пользователем. Точка ветвления, управляемая командой пользователя, называется точкой диалога. В точке диалога создается активный видимый элемент, который называется модальным элементом. Примером модального элемента является диалоговое окно. Когда в программе создается и активизируется модальный элемент, только этот элемент и его подэлементы могут взаимодействовать с пользователем. Исключением являются только командные клавиши и соответствующие поля для мыши, определенные в строке статуса. Эти поля всегда доступны пользователю и нажатие на них обрабатывается модальным элементом так же как, как если бы они были определены в нем.

Диалоговое окно - это специальный тип окна. Класс TDialog порожден от TWindow и предназначен для реализации взаимодействия с пользователем. Его обработчик событий генерирует команду cmCancel в ответ на нажатие клавиши Esc и команду cmDefault в ответ на нажатие клавиши Enter, а также обрабатывает cmOk, cmCancel, cmYes и cmNo, завершая модальное состояние диалогового окна.

Существует несколько отличий между диалоговым окном и другими окнами:

1. Цвет диалогового окна по умолчанию серый вместо синего.
2. Диалоговое окно не может изменять размер.
3. Диалоговое окно не имеет номера.

- **Создание диалогового окна.** Для создания диалогового окна можно определить специальный тип:

```
PDialogWin=^TDialogWin;
TDialogWin=object(TDialog)
Constructor Init(var R:TRect;ATitle:TTitleStr);
end;
```

где R – размеры окна; ATitle – заголовок окна.

```
var PD: PDialogWin;
```

Поскольку в диалоговое окно вставляются различные управляющие элементы, в программе, как правило, используется не объект TDialog, а создается производный от TDialog класс, конструктор которого создает диалоговое окно и вставляет в него необходимые элементы.

Чтобы диалоговое окно стало модальным, его надо вставить в группу DeskTop с помощью метода ExecView:

```
Control : = DeskTop ^ . ExecView ( PD);
```

Когда окно закроется, ExecView удаляет диалоговое окно из группы и осуществляет выход.

Вместо метода ExecView целесообразно использовать метод Execute-Dilog класса TProgram:

```
Function TProgram.ExecuteDialog (P : TDialog; Data : Pointer ) : Word;
```

где Data – указатель на запись с передаваемыми данными.

Тогда диалог инициируется следующим образом:

```
Control := MyApp^.ExecuteDialog(P,nil);
```

Здесь данные окну не передаются. Обе функции возвращают код команды, завершившей диалог.

- **Элементы управления диалоговым окном.** ***Кнопка*** (объект TButton) – это прямоугольник с надписью, имитирующей кнопку панели управления. Обычно TButton является элементом группы TDialog и “нажатие” на кнопку инициирует событие, связанное с какой-либо стандартной командой или командой пользователя. Большинство диалоговых окон имеет по крайней мере 2 кнопки: “O K” и “Cancel”. Модуль TDialogs определяет 5 стандартных диалоговых команд, которые могут быть связаны с TButton: cmOk,

cmCancel, cmYes, cmNo, cmDefault. Первые 4 команды одновременно еще и закрывают диалоговое окно, так что модальным становится предыдущий видимый элемент:

Constructor TButton.Init (Var R : TRect, ATitle : TTitleStr; ACommand : Word; AFlags : byte);

где R – область, занимаемая кнопкой;

ATitle – текст, который помещается в кнопке;

ACommand – команда связанная с кнопкой;

AFlags – флаг типа кнопки.

Когда создается кнопка, ее флаг может быть установлен bfNormal или bfDefault . Кнопка, помеченная как bfDefault, будет кнопкой по умолчанию, т.е. она “нажимается” при нажатии пользователем клавиши “Enter”.Обычно кнопка “Ok” является кнопкой по умолчанию. Например:

```
r.Assign(15,12,27,14);
Insert(New(PButton,Init(r,'~O~k',cmOk,bfDefault)));        {добавляется кнопка Ok}
r.Assign(32,12,43,14);
Insert(New(PButton,Init(r,'Cancel',cmCancel,bfNormal)));{добавляется кнопка Cancel}
```

Статиче ский текст (объект TStaticText) – это видимый элемент, который просто отображает строку, переданную в него. Текст будет центрироваться, если строка начинается с #3. Символ #13 в тексте указывает на начало новой строки.

Constructor TStaticText.Init (Var R : TRect, const AText : String);

Текст автоматически делится на слова и размещается внутри прямоугольника видимого элемента без переноса.

```
R.Assign(1,1,35,2);
DepText:=New(PStaticText,Init(r,' Текст'));
Insert(DepText);
```

Строка ввода (объект TInputLine) используется для ввода различных текстовых строк с клавиатуры:

Constructor TInputLine.Init (Var R : TRect, AMaxLen : integer);

где AMaxLen – размер буфера (≤ 255) для вводимых данных.

Например:

```
R.Assign(3,8,37,9);
```

B:=New(PInputLine,Init(r,128));{добавляется строка ввода длиной не более 128 символов}

Insert(b);

Метки управляющих элементов (объект TLabel) служат для отображения поясняющего текста и, кроме того, метка связывается с другим видимым элементов, так что отметка мышкой метки приводит к активизации связанного с ней элемента.

Constructor TLabel.Init (Var Bounds : TRect, const AText : String; ALink : PView);

где Bounds – размер поля метки;

AText – текст метки;

ALink – указатель на объект, который связан с меткой.

Например:

R.Assign(2,7,30,8);

Insert(New(PLabel,Init(R,'Enter the number, please',B)));{добавляется метка, связанная со строкой ввода}

- **Установка и получение данных.** Элементы управления бесполезны, если неизвестно как получить от них информацию. Надо иметь возможность сделать 2 вещи:

1) установить начальные значения элементов;

2) прочитать значения при закрытии диалогового окна;

Для этого используются соответственно методы **SetData** (копирует данные в видимый элемент) и **GetData** (копирует данные из видимого элемента). Каждый видимый элемент имеет методы SetData и GetData. Когда группа (например, TDialog) инициализируется с помощью вызова SetData, она передает данные дальше, вызывая методы SetData для каждого из подэлементов. При вызове SetData для группы методу передается запись данных, которая содержит данные для каждого видимого элемента в группе. Данные для каждого элемента надо расположить в том виде, в котором они были вставлены в группу.

Для установки правильного размера данных для каждого видимого элемента используется метод **DataSize**, который возвращает размер видимого элемента.

После выполнения диалога, надо проверить не была ли выполнена отмена окна (команда cmCancel), а затем вызвать метод GetData для передачи информации в программу. Видимые элементы, которые не имеют данных (например, метки и кнопки), используют метод GetData, который они наследуют от TView и который ничего не делает, т.е. при установке и получении информации их можно пропустить. Можно установить запись данных для диалогового окна в глобальном типе:

```
DialogData=record
CheckBoxData : word;
RadioButtonData : word;
InputLineData : string[128];
. . . . . .
end;
var Ddata: DialogData;
```

Затем до выполнения диалогового окна надо установить данные и прочитать их, когда окно успешно закрыто.

```
PW^ . SetData (Ddata);
Control : = DeskTop ^ . ExecView ( PW);
if Control<> cmCancel then PW^ . GetData(Ddata);
```

Установка начальных значений диалогового окна выполняется в конструкторе.

- **Обработка команд пользователя.** Обработчик событий диалогового окна обрабатывает следующие стандартные команды cmClose, cmCancel, cmOk, cmYes, cmNo, cmResize. Чтобы заставить его реагировать на команды пользователя, нужно перекрыть стандартный обработчик событий.

```
PDialogWin=^TDialogWin;
TDialogWin=object(TDialog)
Constructor Init(R:TRect;ATitle:TTitleStr);
Procedure HandleEvent(var Event:TEvent);virtual; {перекрыли обработчик событий}
end;
```

```
var PD: PDialogWin;
```

В новом методе сначала вызывается стандартный обработчик, а затем анализируется событие; если оно не очищено и содержит команду, то надо заставить обработчик выполнить процедуру, связанную с этой командой. Например, так:

```
Procedure TDialogWin . HandleEvent;
Begin {HandleEvent}
inherited HandleEvent(Event);
with Event do
 if What = evCommand then
 case Command of
 cmContinue   :ViewNext;
. . . . . .
 end;
ClearEvent(Event);
End;{HandleEvent}
```

- **Другие элементы управления.** Модуль Dialogs предоставляет дополнительные возможности. Они используются аналогичным способом: создается новый экземпляр объекта, вставляется в диалоговое окно и соответствующие данные включаются в запись данных.

Кластеры кнопок представляют собой прямоугольные видимые элементы, имитирующие несколько зависимых или независимых кнопок. Для создания и использования кластера предусмотрен абстрактный объект **TCluster** и его потомки **TRadioButtons** (для создания независимых кнопок) и **TCheckBoxes** (для создания зависимых кнопок).

Просмотр списка (TListViewer) – выводит список в одну или несколько колонок и пользователь может выбрать элемент из этого списка.

TListViewer может взаимодействовать с двумя полосами скроллинга. Tlist Viewer предназначен для построения блока и не используется отдельно. Он может обрабатывать список, но сам не содержит списка. Его абстрактный

метод GetText загружает элементы списка для его метода Draw. Наследник TListViewer должен перекрывать GetText для загрузки актуальных данных.

Окно списка (TlistBox) – наследуется от TListViewer, он владеет объектом TCollection, который должен содержать указатели на строки. TListBox поддерживает одну полосу скроллинга. При получении или установке данных окна списка удобно использовать запись TListBoxRec, которая хранит указатель на список строк и слово, указывающее на текущий выбранный элемент списка.

История (THistory) – реализует объект, который работает со строкой ввода и связанным окном списка. С его помощью пользователь может вызвать список предыдущих значений для строк ввода и выбрать любое значение из этого списка.

Таким образом, для организации диалога нужно выполнить следующие действия:

1) определить класс, производный от TDialog, в этом классе определить конструктор Init и обработчик событий HandleEvent.

2) определить запись для хранения данных диалогового окна;

3) определить методы, связанные с нестандартными командами диалогового окна;

4) инициировать диалог методом ExecView или ExecuteDialog;

5) если диалог завершен успешно, прочитать данные окна.

Порядок выполнения работы.

1. Определить класс объектов (TMyObject), которые будут созданы и сохранены в файле.

2. Сформировать меню и строку статуса TV-программы. Меню должно содержать следующие пункты:

- Добавить объект
- Просмотреть объект
- Найти объект
- Редактировать объект
- Удалить объект
- Сохранить объекты в файле
- Загрузить объекты из файла

- Выход

Предусмотреть удобную иерархию в меню. Например, главное меню может содержать два пункта Файл и Работа, а все вышеперечисленные пункты размещены в них как пункты подменю.

3. Ввести в программу объект TMyDialog = object (TDialog) и с его помощью создать диалоговое окно для ввода информации об объекте.

4. В класс TMyApp добавить методы для создания пользовательских объектов и добавления их в список. Например, метод NewObject, в котором инициируется диалог для ввода значений полей объекта, создается объект и помещается в список. Реализовать этот метод и добавить его вызов в обработчик событий класса TMyApp.

5. Реализовать метод TMyObject.Show, используя окно TWindow и статический текст или строки ввода (TInputLine), для вывода информации об объекте.

6. В класс TMyApp добавить метод ShowObject, который просматривает список и показывает объекты, используя метод Show объекта. Реализовать этот метод и добавить его вызов в обработчик событий класса TMyApp.

7. В класс TMyApp добавить методы для корректировки и поиска требуемого объекта по ключевому полю (например, по имени сотрудника). Реализовать эти методы и добавить их вызов в обработчик событий класса TMyApp.

8. В класс TMyApp добавить метод для удаления требуемого объекта. Реализовать этот метод и добавить его вызов в обработчик событий класса TMyApp.

9. В класс TMyApp добавить метод для сохранения полей объектов в файле (метод SaveFile). Метод SaveFile использует стандартный диалог TFileDialog для выбора имени файла и записывает в файл значения полей объектов. Реализовать метод и добавить его вызов в обработчик событий класса TMyApp.

10. Отладить программу и выполнить ее тестирование.

11. В класс TMyApp добавить метод для загрузки полей объектов из файла (метод LoadFile). Метод LoadFile использует стандартный диалог TFileDialog для выбора имени файла, читает из файла значения полей

объектов, создает объекты и помещает их в список. Реализовать метод и добавить его вызов в обработчик событий класса TMyApp.

12. Отладить программу и выполнить ее тестирование.

Методические указания.

1. Тип объектов выбирается в соответствии с вариантом задания (приведен в “Приложении”).

2. Пример определения класса TMyApp:

```
TMyApp=object(TApplication)
Constructor Init;
Procedure HandleEvent(Var Event : TEvent);Virtual;
Procedure InitStatusLine; Virtual;
Procedure InitMenuBar; Virtual;
Procedure LoadFile; Virtual;
Procedure SaveFile; Virtual;
Procedure NewObject; Virtual;
Procedure FindObject;
Procedure DeleteObject;
Procedure ShowObject;
Procedure ChangeObject;
Procedure ChangeDir;
end;
```

3. Конструктор TMyApp.Init должен содержать оператор, в котором глобальной переменной, указывающей на начало списка, присваивается nil.

4. Диалоговое окно создается с помощью специального объекта TNewDialog=object (TDialog), в конструкторе которого определяются элементы управления диалоговым окном.

5. Чтобы получить информацию из диалогового окна, надо определить метод NewObject, в котором это окно сделать модальным, прочитать из него информацию с помощью стандартного метода GetData и записать эту информацию в поля объекта, используя конструктор этого объекта. Например:

```
pwd:=New(PNewDialog,Init(R,'Новый сотрудник' ));
```

```
if DeskTop^.ExecView(pw)<> cmCancel then
begin
pwd^.GetData(rec);
Dispose(pwd,Done);
val(rec.age,age,ok);
if ok<>0 then exit;
name:=rec.name;
. . . . . .
employee:=New(PEmployee,Init(name,sex,age,post,work));
InsertEmpl(employee); {вставляет объект в список}
end;
```

где rec – это переменная, в которую заносится введенная пользователем с помощью строк ввода диалогового окна информация;

InsertEmpl – глобальная процедура, которая вставляет объект в список.

6. Для организации более удобного ввода информации об объектах, можно, кроме конструктора, реализовать для TNewDialog метод HandleEvent (var Event : TEvent), который и будет осуществлять ввод информации об объекте. В этот метод можно добавить обработку нестандартной команды cmNewObject, которая выполняет ввод информации и добавление объекта в список:

```
Procedure  TNewDialog(Var Event:TEvent);
Begin
inherited HandleEvent(Event);
case Event.Command of
 cmNewObject:
  begin
  FillChar(rec,sizeof(rec),' ');
  GetData(rec);
. . . . . .
employee:=New(PEmployee,Init(rec.name, sex, age, rec.post, work));
InsertEmpl(employee);
 else exit;
```

```
end;{case}
ClearEvent(Event);
End;
```

Окно TNewDialog должно содержать кнопку, связанную с командой cmNewObject. Это может быть кнопка “Ok”.

При такой реализации метод TMyApp.NewObject будет только выводить на экран модальное окно для ввода информации.

7. Метод TMyObject.Show показывает объект, выводя значения его полей, используя объекты TStaticText.

```
pw:=New(PWindow,Init(r,' Сотрудник',0));
r.assign(1,1,35,2);
StatText:=New(PStaticText,Init(r,'Имя: '+GetName));
pw^.Insert(StatText);
```

и т.д.

Для закрытия окна следует добавить в окно кнопку “Ok”:

```
r.assign(15,17,27,19);
Insert(New(PButton,Init(r,'~O~k',cmOk,bfDefault)));
SelectNext(false);
```

8. Для просмотра всех объектов, надо определить метод TMyApp.Show, в котором реализуется проход по списку и обращение к методу Show каждого объекта списка.

9. Для поиска объекта в списке можно реализовать вспомогательную функцию Find, в которую передается ключ поиска, например find_name : string. Функция возвращает указатель на объект, если объект найден и nil в случае неудачного поиска.

```
Function Find(find_name:string):PEmployee;
Var p:PItem;
Begin
p:=beg;
while (p<>nil) and (p^.Item^.GetName<>find_name) do p:=p^.next;
if p^.item^.GetName=find_name then
Find:=p^.Item
else Find:=nil;
End;
```

Метод, который показывает найденный объект (например, метод TMApp.FindObject), может использовать стандартные функции **InputBoxRect** для ввода ключа поиска и **MessageBox** для вывода сообщения:

```
if InputBoxRect(r,'Поиск по имени ',' ,s,35)=cmOk then
begin
employee:=Find(s);
if employee<>nil then employee^.Show
else MessageBox ('Объекта с таким именем нет', nil,+mfOkButton);
end
```

10. Метод, который выполняет корректировку объекта, может быть реализован следующим образом: сначала выполняется поиск корректируемого объекта с помощью вспомогательной функции Find, которая возвращает указатель на нужный объект. Если этот указатель не равен nil, то формируется диалоговое окно, в которое заносятся поля объекта, обработчик команд диалогового окна выполняет добавление откорректированного объекта в список, а старый объект удаляется с помощью метода DeleteObject:

```
Procedure TMyApp.DeleteObject;
Var employee:PEmployee;
    pwd:PNewDialog;
    r:TRect;
Begin
…
employee:=Find(name);

if employee <>nil then
begin
r.assign(10,3,54,22);
pwd:=new(PNewDialog,init(r,' Сотрудник'));
{Формируется запись data, в которую записываются поля объекта employee
^. Затем информация из data передается в строки TInputLine окна pwd^,
вызовом метода SetData}
 mdw^.SetData(data);
if DeskTop^.ExecView(pwd)<>cmCancel then begin{Создается объект и
включается в список}
end
```

```
else MessageBox('Объекта с таким именем нет' nil,mfInformation+mfOkBut-
ton);
End;
```

Можно поступить и по-другому. Определить в классе TEmployee метод Set, который устанавливает поля объекта, используя данные, переданные ему через параметры. Тогда достаточно просто установить новые значения полей объекта, считанные с диалогового окна методом GetData, не удаляя старый объект и создавая новый.

11. Для удаления объекта из списка можно написать процедуру DeleteObject(p:PEmployee), в которую передается указатель на удаляемый объект. В процедуре осуществляется поиск объекта в списке и, если он найден, его удаление.

Содержание отчета.

1. Титульный лист.
2. Графическая схема иерархии классов.
3. Графическая схема иерархии объектов.
4. Определение всех классов и глобальных имен (констант, типов, переменных, процедур и функций).
5. Определение методов TmyApp (с комментариями), используемых в программе.
6. Определение конструктора диалогового окна с комментариями.
7. Определение обработчика событий для TMyApp с комментариями.

Приложение. Варианты заданий.

Номер варианта	Структура
1	СТУДЕНТ
2	СЛУЖАЩИЙ
3	КАДРЫ
4	ЦЕХ

5	БИБЛИОТЕКА
6	ЭКЗАМЕН
7	АДРЕС
8	ТОВАР
9	КВИТАНЦИЯ
10	АВТОМОБИЛЬ

Описания структур

1. СТУДЕНТ
имя – string
курс – integer
пол – boolean

2. СЛУЖАЩИЙ
имя – string
возраст – integer
рабочий стаж – integer

3. КАДРЫ
имя – string
номер цеха – unteger
разряд – integer

4. ЦЕХ
имя – string
шифр – string
количество – integer

5. БИБЛИОТЕКА
имя – string
автор – string
стоимость – real

6. ЭКЗАМЕН
имя студента – string
дата – integer
оценка – integer

7. АДРЕС
имя – string
улица – string
номер дома – integer

8. ТОВАР
имя – string
количество – integer
стоимость – real

9. КВИТАНЦИЯ
номер – integer
дата – integer
сумма – real

10. АВТОМОБИЛЬ
тип – string
мощность – integer
регистрационный номер – string

Лабораторная работа № 7
КОЛЛЕКЦИИ. ХРАНЕНИЕ И ПОИСК ОБЪЕКТОВ

Цель. Получить практические навыки использования коллекций в TV-программе.

Основное содержание работы.

Создание объектов пользовательского класса, сохранение их в коллекции и в файле. Загрузка объектов с файла и занесение их в коллекцию. Поиск в коллекции.

Краткие теоретические сведения.

- **Коллекция** – это класс объектов TV, предназначенных для хранения элементов. В этом смысле коллекция служит тем же целям, что и массивы. Однако коллекции существенно отличаются от массивов.

Во-первых, размер коллекции может динамически меняться в ходе работы программы, фактически ограничиваясь лишь доступной памятью. Во-вторых, в коллекции могут храниться элементы разных типов, в том числе и объекты. Это свойство называется полиморфизмом коллекции.

В Turbo Vision коллекции технически реализованы как массивы нетипизированных указателей на размещённые в динамической памяти элементы коллекций.

- **Реализация коллекции в Turbo Vision**. Для создания коллекции используют объекты классов TCollection, TSortedCollection и TStringCollection. Это классы так называемых неотображаемых объектов TV.

В лабораторной работе используются объекты TCollection или TSortedCollection. TSortedCollection прямой потомок от TCollection и отличается от последнего тем, что реализует коллекцию, отсортированную по ключу. TsortedCollection – абстрактный класс и, чтобы использовать его, надо создать производный класс и перекрыть методы KeyOf и Compare.

В принципе коллекция может хранить данные любых типов. Однако некоторые методы TCollection предназначены для обработки коллекции объектов, порождённых от TObject (например, FreeItem – освободить эле-

мент коллекции). Поэтому либо все включённые в коллекцию объекты должны быть потомком от TObject, либо следует перекрывать соответствующие методы.

TSortedCollection наследует почти все методы TCollection, за исключением IndexOf и Insert, которые перекрыты. Кроме этого, добавлены три метода: Compare, KeyOf и Search.

- **Создание коллекции**. Коллекция создаётся вызовом конструктора:

```
Constructor Init (ALimit, ADelta: integer);
```

где Alimit – начальная длина коллекции (количества размещаемых в динамической памяти указателей);

Adelta – шаг приращения коллекции. Если в ходе пополнения коллекции её длина превысит начальную, TV будет наращивать по ADelta указателей.

Например:

```
…
Var Persons : PCollection;
…
Begin
    Persons : = New(PCollection , Init (50 , 20) );
…
```

Для отсортированной коллекции

```
Type
    PMySortedCollection= ^ TMySortedCollection;
    TMySortedCollection=object(TSortedCollection)
    Function KeyOf (item:pointer):pointer;virtual;
    Function Compare(key1,key2:pointer):integer;virtual;
    end;
Var Persons: PMySortedCollection;
```

- **Включение элементов в коллекцию**. Элементы в коллекцию включаются методом

```
Procedure Insert (Item:pointer);virtual;
```

где Item – указатель на вставленный в коллекцию объект.

Для отсортированной коллекции предварительно проверяется, есть ли уже в коллекции элемент с таким значением ключа. Если такого элемен-

та нет, то он вставляется в позицию, определённую своим ключом. Если элемент найден, дальнейшие действия будут определены значениями поля Duplication: при False вставка не осуществляется, при True вставляется новый элемент перед элементом с одинаковым ключом.

- **Методы KeyOf и Compare**. Эти методы использует класс TSortedCollection для упорядочивания элементов в коллекции.

- Function KeyOf (item:pointer):pointer;virtual;

Для данного элемента item^ коллекции KeyOf возвращает указатель на соответствующий ключ. TSortedCollection. KeyOf просто возвращает Item. KeyOf перекрывается в случае, когда ключ элемента не совпадает с самим элементом.

- Function Compare(key1,key2:pointer):integer;virtual;

Это абстрактный метод, который должен быть перекрыт во всех порождённых типах. Он сравнивает два ключа key1 и key2 и возвращает:

-1, если key1^ < key2^

0, если key1^ = key2^

1, если key1^ > key2^

В этом случае коллекции сортируются по возрастанию ключей. Key1 и Key2 – это значения указателей, извлечённых из коллекции методом KeyOf.

- **Удаление элементов из коллекции**. Элемент из коллекции можно удалить с помощью следующих методов:

- Procedure Delete (item : pointer); удаляет элемент item из коллекции.
- Procedure AtDelete (index : integer); удаляет элемент в позиции index.
- Procedure DeleteAll; удаляет из коллекции все элементы, устанавливая count в 0.

Все эти методы удаляют элементы только из коллекции, не освобождая занимаемую ими память. Следующие методы освобождают память, занимаемую элементами коллекции:

- Procedure FreeItem (item : pointer); virtual; освобождает item, вызывая if item <> nil then Dispose(PObject (item), Done); Сам item из коллекции не удаляется. Если требуется удалить и item, то необходимо вызвать
- Procedure Free (item:pointer); эквивалентно FreeItem (item); Delete (item);

- Procedure AtFree (index : integer); удаляет из коллекции элемент в позиции index и освобождает выделенную память.
- Procedure FreeAll; удаляет и освобождает все элементы коллекции.

- **Просмотр коллекции**. Метод Function At (index : integer) : pointer; возвращает указатель на элемент с индексом index в коллекции. Этот метод позволяет интерпретировать коллекцию как индексированный массив.

Поле count : integer; возвращает текущее число элементов в коллекции. Если коллекция содержит объекты клсссов, производных от TPerson, то просмотреть ее можно следующим образом:

```
for k : = 0 to Persons^ . count - 1 do
View (Persons ^. At ( k ) );
```

где View – внешняя процедура вида

```
Procedure View (P : PPerson )
begin
        P^ . Show;
end;
```

где Show – виртуальный метод просмотра объектов классов, производных от PPerson.

Если коллекция содержит объекты одного класса, например, TStudent, то просмотреть её можно следующим образом:

```
for k : =0 to Persons ^ . count - 1 do
(PStudent ( Persons^ . At ( k ) ) ) ^ . Show;
```

- **Итерационные методы**. Коллекции имеют три итерационных метода.

- Итератор ForEach.

```
Procedure ForEach (Action : pointer );
```

Применяет действие, определённое процедурой, на которую указывает action, для каждого элемента коллекции.

Используя ForEach, коллекцию можно просмотреть следующим образом:

```
Procedure ViewCollection (p : PCollection);
```

```
Procedure View (p : PPerson) ; far;
begin
  p^ . Show
end;
Begin
  p^ . ForEach( @View );
End;
```

- Итератор FirstThat.

Function FirstThat (Test : pointer) : pointer;

Применяет логическую функцию, заданную указателем Test, к каждому элементу коллекции до тех пор, пока Test, возвращает False. Результат – указатель на элемент, для которого Test возвращает True, или nil, если такого элемента нет. Коллекция просматривается от начала к концу.

- Итератор LastThat.

Function LastThat (Test:pointer):pointer;

То же, что и FirstThat, но коллекция просматривается от конца к началу.

- **Поиск в коллекции**. Для поиска требуемого элемента в коллекции можно использовать итератор FirstThat или LastThat, в который передаётся функция, возвращающая True, если удовлетворено условие поиска.

Например, требуется найти в коллекции, представляющей собой телефонный справочник, абонента по его имени.

```
Function Find (p : PCollection , name:string35) : PPhone;

Function Equal (p : PPhone) : boolean ;f ar;
begin
Equal :=p^ . name=name
end;
Begin
Find:=p^ . FirstThat (@ Equal);
End;
```

Если коллекция – TSortedCollection и требуется выполнить поиск по ключу, то можно использовать метод Search.

Function Search (key:pointer ; var Index : integer) : boolean; virtual;

Возвращает True, если элемент, с заданным ключом key^, найден в отсортированной коллекции. Если элемент найден, в Index помещается его индекс, в противном случае индекс, который будет присвоен новому элементу при его вставке в коллекцию.

```
Function Find (p:PMySortedCollection , name : string35): PPhone;
var k : integer;
    ok : boolean;
Begin
    ok := p^ . Search(@ name , k);
    if ok then Find :=p^ . At(k)
    else Find:=nil;
End;
```

Порядок выполнения работы.

1. Определить классы объектов, которые будут храниться в коллекции. Использовать пользовательские классы предыдущей лабораторной работы № 6 в соответствии со своим вариантом. Например, Temploуee – служащий.

2. Сформировать меню и строку статуса TV-программы. Меню должно содержать следующие пункты:

- Поместить (в коллекцию служащего)
- Просмотреть (коллекцию)
- Поиск (служащего)
- Редактировать (служащего)
- Удалить (служащего)
- Сохранить (коллекцию)
- Загрузить (коллекцию)
- Сменить директорий
- Выход

Предусмотреть удобную иерархию в меню. Это меню составлено для рассмотренного выше примера. Естественно, в пунктах “поместить” и т.д. следует использовать объекты своих пользовательских классов. Продумать и разместить строку статуса.

3. В классс TMyApp добавить:

а) метод для ввода данных пользовательских объектов, например, метод PutEmployee;

б) метод Show – просмотреть коллекцию.

4. Реализовать эти методы (см. п. “Методические указания”) .

5. Реализовать обработчик событий объекта TMyApp. Предусмотреть в нем обработку команд типа cmShow – просмотреть коллекцию, cmChDir-изменить каталог,cmPutEmployee – ввести данные о служащем.

6. Ввести в программу объект TMyDialog = object (TDialog) и определить в нем обработчик событий, который обрабатывает команды типа cmAppendEmployee – добавить в коллекцию служащего.

7. Отладить программу и выполнить ее тестирование.

2. Реализовать метод для просмотра объекта TMyObject.Show, используя окно TDialog.

3. Реализовать метод для просмотра коллекции TMyApp.Show.

4. Отладить программу и выполнить ее тестирование.

5. В класс TMyApp добавить методы для корректировки и поиска требуемого объекта по ключевому полю (например, по имени сотрудника). Реализовать эти методы и добавить их вызов в обработчик событий класса TMyApp. Для корректировки объекта использовать окно TDialog.

6. Отладить программу и выполнить ее тестирование.

7. В класс TMyApp добавить методы : Save – сохранить объекты из коллекции в файле: Load - загрузить объекты из файла в коллекцию.

8. Добавить в TMyApp.HandleEvent вызов этих методов как результат обработки соответствующих команд меню.

9. Реализовать методы Save и Open.

10. Отладить программу и выполнить ее тестирование.

11. Сформулировать условие для поиска элемент в коллекции. Например, найти служащего с заданным именем.

12. Добавить в программу методы для поиска элемента и внести изменения в обработчик событий.

13. Отладить программу и выполнить ее тестирование.

Методические указания.

1. Тип используемой в работе коллекции (TCollection или TSorted-Collection) выбирается в зависимости от варианта задания: четный вариант – TCollection, нечетный – TSortedCollection.

2. Пример определения класса TMyApp:

```
TMyApp =object(TApplication)
   Constructor Init;
   Procedure InitStatusLine;Virtual;
   Procedure InitMenuBar;Virtual;
   Procedure HandleEvent(Var Event:TEvent);Virtual;
   Procedure Show;
   Procedure PutEmployee;virtual;
   Procedure ChangeDir;
   Procedure Load;
   Procedure Save;
   Procedure LoadCollection;
   Procedure SaveCollection;
   Procedure Find;
end;
```

3. Коллекция создается:

а) при инициализации приложения, например:

```
Constructor TMyApp.Init;
begin
inherited Init;
MyCollection := new( PCollection , Init(50 , 10) );
end;
```

б) при загрузке данных из файла. При этом старая коллекция предварительно очищается MyCollection . FreeAll .

4. Можно рекомендовать следующую схему добавления объектов в коллекцию (на примере объекта TEmployee):

а) метод TMyApp.HandleEvent обрабатывает команду cmPutEmployee, вызывая метод TMyApp.PutEmployee;

б) метод TMyApp.PutEmployee создает диалоговое окно (объект TMyDialog) для ввода необходимой информации. В окно помещается две кнопки:

```
Insert(New(PButton,Init(r,'~Д~оба-
вить',cmAppendEmployee,bfDefault)));
Insert(New(PButton,Init(r,'~З~акончить',cmCancel,bfNormal)));
```

в) метод TMyDialog.HandleEvent обрабатывает команду cmAppendEmployee, создавая объект класса TEmployee и добавляя его в коллекцию:

```
Procedure TMyDialog.HandleEvent(Var Event:TEvent);
….
Begin
inherited HandleEvent(Event);
case EventCommand of
cmAppendEmployee :
begin
FillChar( Data1 , sizeof (Data1) , ‘ ‘ );
GetData ( Data1);
Employee := new ( PEmployee , Init (. . . . . .));
MyCollection^ . Insert (Employee)
end;
End;
```

Перед добавлением объекта в коллекцию можно запросить подтверждение вызовом функции MessageBox:

```
if MessageBox (‘Объект будет добавлен в коллекцию’ , nil, mfOkBut-
ton) = cmOk then {объект добавляется в коллекцию}.
```

5. Для просмотра коллекции лучше использовать итератор ForEach:

```
Procedure TMyApp.Show;
Procedure CallShow (p : Pperson); far;
begin
p^ . Show;
end;
begin
MyCollection ^. ForEach ( @CallShow ) ;
end;
```

6. Поиск элемента в коллекции можно организовать по следующей схеме:

а) метод TMyApp.HandleEvent обрабатывает команду cmFindEmployee, вызывая метод TMyApp. FindEmployee;

б) метод FindEmployee создает диалоговое окно для ввода ключа поиска (например, фамилии служащего)

```
R . Assign(. . . . . );
MyDialog := New ( PMyDialog , Init( R, ‘ Поиск’ ));
R . Assign(. . . . . );
PInput := New ( PInputLine, Init (R, 30));
Insert ( PInput) );
R . Assign(. . . . . );
Insert ( New (PLabel, Init ( R, ‘Введите имя :’, P Input )));
R . Assign(. . . . . );
Insert ( New (PButton, Init ( R, ‘OK’,cmOk, bfDefault )));
R . Assign(. . . . . );
Insert ( New (PButton, Init ( R,’Cancel’, cmCansel, bfNormal )));
SelectNext(False);
```

в) после ввода ключа поиска и окончании диалога по кнопке ОК вызывается итератор FirstThat:

```
if DeskTop^ . ExecView (MyDialog)=cmCancel then exit
name := PInput^ . Data^;
Employ := Find( MyCollection , name);
```

где Find – внешняя функция поиска, эту функцию надо написать самостоятельно, предусмотрев в ней вызов итератора.

Или

```
if Application^. ExecuteDialog ( MyDialog, @name)= cmCancel then exit;
```

Для ввода значения ключевого поля можно использовать также стандартное диалоговое окно, создаваемое функцией InputBox

```
if InputBox (‘Введите фамилию’ , ‘ Фамилия : ’, name, 35) = cmOk then begin
    Employ := Find ( MyCollection, name);
```

и т.д.;

г) затем вызывается метод Show для показа найденного объекта Employ^. Show.

7. Для сохранения коллекции следует использовать типизированный файл

```
Type TFile = file of TData1;
```

где TData1 – запись, содержащая поля объекта.

Просматривая коллекцию, мы переносим поля объекта в запись Date1: TDate1и записываем ее в файл.

8. При загрузке коллекции из файла сначала удаляется старая коллекция MyCollection . FreeAll, затем в коллекцию записываются объекты, созданные на основе информации, прочитанной из файла:

```
K := 0;
while not eof ( f ) do begin
read ( f, Data1);
k := k + 1;
MyCollection^ . Insert(New (PEmployee, Init( . . . . . )));
end;
```

После этого можно выдать информационное сообщение

```
if K <> 0 then
MessageBox('Коллекция сформирована и содержит %d объектов', @k, mfInformation)
else
MessageBox('Коллекция пустая', nil, mfInformation);
```

9. При сохранении (загрузке) коллекции следует организовать диалог для ввода имени файла. Для этого лучше всего использовать объект TFileDialog (модуль StdDlg). Например:

```
var FileDialog : PFileDialog;
    S :        PathStr;
begin

FileDialog := New(PFileDialog, Init ( '*.dat', ' загрузить из файла', ' имя файла', fdOkButton , 0));
if ExecuteDialog ( FileDialog, @S)<>cmCancel then  begin
assign(f , s);
```

и т.д.

10. Для изменения текущего каталога следует использовать объект TChDirDialog (модуль StdDlg).

```
Var ChDirDialog : PChDirDialog;
begin
ChDirDialog := New (PChDirDialog, Init ( cdNormal, 0));
if ValidView(ChDirDialog) <> nil then begin
DeskTop^. ExecView ( ChDirDialog);
```

```
Dispose (ChDirDialog , Done);
end;
```

Содержание отчета.

1. Титульный лист.
2. Графическая схема иерархии классов.
3. Графическая схема иерархии объектов.
4. Определение всех классов и глобальных имен (констант, типов, переменных, процедур и функций) с необходимыми пояснениями.
5. Описание схемы обработки событий.
6. Описание методов Tcollection (TSortedCollection), используемых в программе.
7. Описание диалогов и их реализация.

Лабораторная работа № 8
СОХРАНЕНИЕ ОБЪЕКТОВ В ПОТОКЕ

Цель. Получить практические навыки использования потоков в TV-прграмме.

Основное содержание работы.

Написать программу, создающую объекты пользовательского класса и помещающую их сначала в коллекцию, а затем в поток. Объекты могут читаться из потока и помещаться в коллекцию. Предусмотреть диалог для ввода исходных данных и просмотра объектов.

Краткие теоретические сведения.

Поток. Поток **TStream** и его потомки **TDosStream, TBufStream, TEMSStream** предназначены для хранения в них объектов. В отличие от файлов потоки могут существовать не только на внешних устройствах, но и в оперативной памяти. Также поток отличается от файла тем, что в нём можно хранить данные разных типов, в том числе и объекты.

Работа с потоком напоминает работу с файлом последовательного доступа. При создании потока в нём не находится никакой информации. Записываемая в поток информация последовательно приписывается к его концу.

Для того чтобы было возможно сохранять в потоке объекты разных классов (типов) используется механизм регистрации классов (типов объектов). Зарегистрировать класс, означает присвоить ему какой-то уникальный номер. Этот регистрационный номер записывается в файл и считывается первым. Прочитав его, TV сразу же определяет тип объекта.

Регистрация объектов. Для регистрации любого класса используется глобальная процедура **RegisterType** (RObjects:TStreamRec). Единственным параметром обращения к этой процедуре является запись типа **TStreamRec**:

```
TStreamRec=record
Objtype:word;
VMTLink:word;
Load:pointer;
```

```
Store:pointer;
Next:word;
End;
```

Для каждого типа объектов должна быть создана своя запись TStreamRec, если вы собираетесь помещать объект этого типа в поток. Однако для стандартных классов такие записи уже существуют. Итак, для каждого нестандартного класса вы должны подготовить запись типа TStreamRec.

В поле **ObjType** вы должны поместить константу-идентификатор класса. Константы 0..99 уже зарезервированы в TV, поэтому в вашем распоряжении константы в диапазоне 100..65535.

Поле **VMTLink** должно содержать смещение VMT класса. Получить адрес VMT можно функцией **typeof**, а смещение – функцией **ofs**.

Поля **Load, Stor**e должны содержать адреса методов Load и Store регистрируемого класса. Для получения адреса метода применяется операция @.

Поле **Next** организует связь записей в списке.

Обычно регистрация организуется в конструкторе приложения:

```
Const
RMyWindow:TStreamRec=(
Objtype:100;
VMTLink:ofs(typeof(TMyWindow)^);
Load:@TMyWindow.load;
Store:@TMyWindow.store;
)
constructor TMyApp.Init
begin
inherited Init;
RegisterType(RMyWindow)
end;
```

Можно также выделить отдельную процедуру для регистрации всех объектов.

Для упрощения регистрации стандартных объектов предусмотрены процедуры – RegisterXXX, где XXX – имя соответствующего модуля (RegisterDialogs, RegisterViews и т.д.).

- **Создание и удаление потока.** Поток TStream является абстрактным потомком, потому что он не привязан ни к какому конкретному носителю информации.

TDosStream – реализует не буферизованный доступ к файлу.

TBufStream – реализует буферизованный доступ к файлу. Буферизация позволяет согласовать формат данных с размерами дискового сектора и обычно значительно ускоряет доступ к потоку.

TEMSStream – реализует доступ к отображаемой памяти. Здесь чтение и запись объектов происходит с предельно возможной скоростью. Однако после выключения компьютера вся информация теряется. Такие потоки в основном предназначены для промежуточных записей.

В каждом потоковом классе предусмотрен свой конструктор Init.

Constructor TDosStream.Init(FileName:FnameStr;Mode:Word);
FileName – имя файла;
Mode – режим доступа к данным:
StCreate =$3c00 создать файл;
StOpenRead =$3d00 только для чтения;
StOpenWrite =$3d01 только для записи;
StOpen =$3d02 чтение и запись.

Constructor TBufStream(FileName:FnameStr;Mode:Word;Size:Word);
FileName – имя файла;
Mode – режим доступа к данным;

Constructor TEMSStream.Init(MinSize,MaxSize:longint)
MinSize,MaxSize – минимальный и максимальный размеры блока, который будет передаваться в EMS память.

После завершения работы с потоком необходимо вызвать его деструктор Done.

- **Работа с потоком.** После регистрации классов объектов объекты можно помещать в поток и читать из потока.

Чтобы поместить объект в поток, необходимо обратиться к методу Put, передав ему в качестве параметра инициированный экземпляр объекта.

```
MyStream.Put(MyWindow);
```

Метод Put отыскивает запись класса объекта в списке регистрационных записей, получает его регистрационный номер. Затем в поток записывается идентификатор класса и вызывается метод Store, который записывает поля объекта.

По такой же схеме работает и метод Get. Вначале он считывает идентификатор класса, потом отыскивает соответствующий метод Load, который уже считывает поля.

```
PWindow:=MyStream.Get;
```

Таким образом, действительное чтение и запись объектов в поток производится методами Load и Store. Каждый объект должен иметь эти методы для использования потока, поэтому вы никогда не вызываете их непосредственно (они вызываются из методов Get и Put). Все что вам нужно сделать, это убедиться в том, что объект знает, как записать себя в поток, когда это потребуется, то есть он имеет методы Load и Store.

Методы Store и Load обеспечивают сохранение полей в потоке и чтение их из потока. При наследовании классов достаточно в перекрытом методе вызвать метод предка и записать сохранение/чтение добавленных полей.

Например:

```
TMyDilalog=object (TDialog)
 St : PString;
 Procedure Store(var s:TStream);
 Constructor Load(var s:TStream);
 …
end;

Procedure TMyDilalog.Store(var s:TStream);
Begin
inherited Store(s);
S^.WriteStr(st);
End;
```

```
Constructor TMyDilalog.Load(var s:TStream);
Begin
inherited Load (s);
S^.ReadStr(st);
End;
```

Еще раз подчеркнем, если ли вы предполагаете сохранять свои объекты в потоке, то должны обязательно определить для них методы **Load** и **Store**.

В методах Store и Load для записи/чтения полей объектов в поток (из потока) вы должны использовать методы **TStream.Write/TStream.Read**, **TStream.WriteStr/TStream.ReadStr**, а в некоторых случаях методы **TGroup.PutSubViewPtr/TGroup.GetSubViewPtr** и **TView. PutPeerViewPtr/ TView.GetPeerViewPtr.**

Порядок выполнения работы.

1. Определить классы объектов, которые будут храниться в коллекции и потоке (см. п. “Методические указания”).
2. Сформировать меню и строку статуса TV-программы. Меню должно содержать следующие пункты:

- Добавить объект в коллекцию
- Просмотреть объект (выбранный)
- Просмотреть коллекцию (все объекты)
- Найти объект
- Редактировать объект (выбранный)
- Сохранить объекты в потоке
- Загрузить объекты из потока
- Сменить директорий
- Выход

Предусмотреть удобную иерархию в меню. Например, главное меню может содержать два пункта Файл и Работа, а все вышеперечисленные пункты размещены в них как пункты подменю.

3. В класс TMyApp добавить методы для ввода пользовательских объектов, например, метод NewEmployee – получить информацию о служащем, метод Show – просмотреть коллекцию, метод ShowEmployee – посмотреть объект. Реализовать эти методы.

4. Реализовать обработчик событий объекта TMyApp. Предусмотреть в нем обработку команд типа cmShow, cmChDir и т.д.

5. Ввести в программу объект TMyDialog1 = object (TDialog) и определить в нем обработчик событий, который обрабатывает команды типа cmAddEmployee – добавить в коллекцию служащего.

6. Ввести в программу объект TMyDialog2 = object (TDialog) и определить в нем обработчик событий, который обрабатывает команды типа cmShowEmployee – просмотр служащего.

7. Для просмотра объекта организовать поиск объекта в коллекции по его имени (поле Name).

8. Отладить программу и выполнить ее тестирование.

9. В класс TMyApp добавить методы для корректировки и поиска требуемого объекта по ключевому полю (например, по имени сотрудника). Реализовать эти методы и добавить их вызов в обработчик событий класса TMyApp. Для корректировки объекта использовать окно TDialog.

10. Отладить программу и выполнить ее тестирование.

11. Добавить в класс объекта методы Load и Store и реализовать их.

12. В класс TMyApp добавить методы : SaveInStream – сохранить объекты из коллекции в потоке; LoadOutStream – загрузить объекты из потока в коллекцию.

13. Добавить в TMyApp.HandleEvent вызов этих методов как результат обработки соответствующих команд меню.

14. Реализовать методы SaveInStream и LoadOutStream.

15. Отладить программу и выполнить ее тестирование.

Методические указания.

1. В качестве типов объектов использовать пользовательские классы лабораторной работы № 6 в соответствии со своим вариантом. Тип используемой в работе коллекции (TCollection или TSortedCollection) выбирается в зависимости от варианта задания: нечетный вариант – TCollection,

четный – TsortedCollection (**обратите внимание** – тип коллекции отличается от типа, который был у вас в лабораторной работе № 7.

2. Во избежание проблем при сохранении и загрузки объектов классы объектов пользователя обязательно должны наследоваться от абстрактного класса TObject.

3. В каждом классе, объекты которого помещаются в поток или считываются из потока, должны быть определены методы Store и Load, к которым обращаются методы Put и Get потока. Например:

```
Constructor TEmployee.Load(Var S:TStream);
Begin
s.read(name,sizeof(name));
s.read(post,sizeof(post));
s.read(salary,sizeof(salary));
End;

Procedure TEmployee.Store(Var S:TStream);
Begin
s.write(name,sizeof(name));
s.write(post,sizeof(post));
s.write(salary,sizeof(salary));
End;
```

4. Регистрация объекта (например “Сотрудник”) может быть выполнена следующим образом:

```
Const
REmployee:TStreamRec=(ObjType:101;
VMTLink:Ofs(Typeof(TEmployee)^);
Load:@TEmployee.Load;
Store:@TEmployee.Store);
```

5. Пример определения класса TMyApp:

```
TMyApp =object(TApplication)
Constructor Init;
Procedure InitStatusLine;virtual;
Procedure InitMenuBar;virtual;
```

```
    Procedure HandleEvent(Var Event:TEvent);virtual;
    Procedure Show;
    Procedure ShowEmployee;
    Procedure NewEmployee;
    ……. .
Procedure LoadOutStream;
    Procedure SaveInStream;
    end;
```

6. Конструктор Init должен содержать операторы, в которых выполняется регистрация типов объектов: Например:

```
Constructor TMyApp.Init;
Begin
{регистрация стандартных типов объектов}
RegisterApp;
RegisterObjects;
RegisterMenus;
RegisterViews;
RegisterStdDlg;
{регистрация пользовательского типа}
RegisterType(REmployee);
inherited Init;
{создание коллекции}
MyCollection:=New(PCollection,Init(50,10));
End;
```

7. Чтобы загрузить объекты из потока используется процедура LoadOutStream, в которой с помощью стандартного окна диалога выбирается нужный файл и инициируется поток:

```
MyStream:=New(PBufStream,Init(s,stOpenRead,1024));{открываем
поток для чтения}
if MyStream^.Status=stOk then{если нет ошибки}
begin
LoadCollection;
Dispose(MyStream,Done)
end;
```

LoadCollection – это вспомогательная процедура, которая получает очередной объект из потока и записывает его в коллекцию. В ней выполняются следующие действия:

```
MyCollection^.FreeAll;{освобождаются элементы коллекции}
MyStream^.Seek(0); {встаем на начало потока}
while MyStream^.status=0 do
begin
  p:=PPerson(MyStream^.Get);{читаем очередной объект из потока, PPerson – абстрактный объект, наследниками которого являются  PEmployee и PStudent}
if MyStream^.status=0 then begin {если нет ошибки}
inc(k);
MyCollection^.Insert(p);              {объект вставляется в коллекцию}
end;
```

8. Для записи объектов в поток из коллекции можно использовать процедуру SaveInStream, в которой с помощью стандартного окна диалога выбирается нужный файл, создается поток, затем выбираются из коллекции объекты и помещаются в поток:

```
MyStream:=New(PBufStream,Init(s,stCreate,1024));{открываем поток}
if MyStream^.Status=stOk then{если нет  ошибки}

begin
SaveCollection;
Dispose(MyStream,Done);
end;
```

SaveCollection – вспомогательная процедура, которая все объекты из коллекции записывает в поток. В ней выполняются следующие действия:

```
for k:=0 to MyCollection^.count-1 do
begin
p:=MyCollection^.At(k);
MyStream^.Put(p);{p^ помещается в поток}
end;
```

Если необходимо объекты из коллекции добавить в существующий файл, то поток создается в режиме mode=stOpenWrite

```
MyStream:=New(PBufStream,Init(s,stOpenWrite,1024));
```

Затем устанавливаем текущую позицию в конец потока:
MyStream^.Seek(MyStream^.GetSize);
после этого добавляем объекты из коллекции в поток.

Содержание отчета.

1. Титульный лист.
2. Графическая схема иерархии классов.
3. Графическая схема иерархии объектов.
4. Определение всех классов и глобальных имен (констант, типов, переменных, процедур и функций).
5. Реализация методов **Load** и **Store** с комментариями**.**
6. Реализация методов **LoadOutStream** и **SaveInStream** с комментариями**.**
7. Описание методов класса TMyApp, используемых в программе.

Итоговая лабораторная работа № 3
СОЗДАНИЕ И СОХРАНЕНИЕ ОБЪЕКТОВ

Цель. Закрепить полученные при выполнении лабораторных работ № 5 – 8 навыки разработки объектно-ориентированных приложений с использованием библиотеки Turbo Vision.

Содержание работы

1. Взять за основу лабораторную работу № 8.
2. Добавить еще один класс объектов.
3. В меню “Создать” ввести подменю – “Создать <имя первого класса>” и “Создать <имя второго класса>”. При выборе этих меню вызывается соответствующий классу TDialog.
4. Объекты заносятся в TsortedCollection (коллекция одна – полиморфная).
5. При выборе пункта меню “Просмотр” инициируется диалог “Введите тип объекта” (т.е. объекты какого класса будут просматриваться).
6. Организовать поиск объектов в коллекции (пункт меню «Поиск»).
7. Организовать сохранение объектов из коллекции в потоке.
8. Организовать загрузку объектов из потока в коллекцию.

Методические указания

1. Второй класс объектов выбирается студентом самостоятельно. Для студентов заочного отделения он берется из предметной области, связанной с местом работы студента.
2. Для каждого класса объектов создается свое окно для ввода значений полей при создании объекта (объект TDialog). Эти окна содержат две кнопки – “Ok” и “Cancel”. Их назначение понятно.
3. Просмотр объектов реализуется следующим образом. Просматривается коллекция. Из нее извлекается очередной объект. Определяется класс этого объекта (функция ***typeof***). Если это объект нужного класса (того, который был введен в диалоге “Введите тип объекта”), то он показывается в окне TDialog. Естественно, для каждого класса создается свое окно просмотра TDialog.

4. Окно TDialog для просмотра объекта должно иметь две кнопки – “Next” и “Cancel”. При выборе “Next” показывается следующий объект. Просмотр идет по кольцу, т.е. после последнего объекта показывается вновь первый. При выборе “Cancel” просмотр заканчивается.

5. Диалог “Поиск” организуется следующим образом. Запрашивается и вводится ключ поиска, который должен быть уникальным в общей области объектов двух классов. Далее просматривается коллекция и, если такой объект находится, то он показывается в соответствующем его классу окне TDialog (или TWindow). В противном случае (объект не находится) выдается соответствующее сообщение.

6. Сохранение объектов в потоке и загрузка их из потока выполняется так же, как и в лобораторной работе № 8. Поскольку поток полиморфный, в нем можно одновременно сохранять объекты любого класса. Необходимо только, чтобы они имели общего предка – TObject, т.е. были объектами Turbo Vision.

7. Редактирование и удаление объектов можно в итоговой работе не делать.

Содержание отчета

1. Титульный лист.
2. Постановка задачи.
3. Графическая схема иерархии классов.
4. Графическая схема иерархии объектов.
5. Определения пользовательских классов с комментариями.
6. Листинг демонстрационной программы с комментариями.
7. Результаты тестирования программы.

СПИСОК ЛИТЕРАТУРЫ

Основная

1. Бадд Т. Объектно-ориентированное программирование в действии. – СПб: Питер, 1997.
2. Фаронов В.В. Турбо Паскаль 7.0: В 2 кн. – М.: Нолидж, 1997. Кн. 1.
3. Фаронов В.В. Турбо Паскаль: В 3 кн. Кн. 2: Библиотека Turbo Vision. – М.: МВТУ – Фесто-Дидактик, 1993.

Дополнительная

1. Буч Г. Объектно-ориентированное проектирование. – М.: Конкорд, 1992.
2. Епанешников А., Епанешников В. Turbo Vision 2.0: Основы практического использования. – М.: Диалог-МИФИ, 1995.

3. Федоров А. Borland Pascal: практическое использование Turbo Vision 2.0. – Киев: Диалектика, 1993.
4. Borland Pascal with Object 7.0: Документация фирмы Borland International / Пер. с англ. – на сервере кафедры АСУ Перм. гос. техн. ун-та.
5. Borland Pascal 7.0: Turbo Vision. Документация фирмы Borland International / Пер. с англ. – на сервере кафедры АСУ Перм. гос. техн. ун-та.