Аргов Д.И.

Объектно-ориентированное программирование: новый взгляд и новое мышление

учебное пособие

Рыбинск, 2017 г.

Содержание:

- Причины появления ООП
- Иерархия классов
- Ответственность программиста
- Готовность к изменениям
- Зацепление и связность
- Методы и данные
- Сообщения
- Размещение и инициализация объектов
- Наследование
- Замещение и уточнение
- Преобразование типов. Полиморфизм
- Отложенные методы
- Программа Figure
- Понятие объекта
- Инкапсуляция
- Наследование
- Полиморфизм
- Виртуальные методы
- Программа «Фигура»
- Рисование всех объектов
- Метод Insert
- Обмен сообщениями
- Задание
- Общая схема построения интерфейсной оболочки
- Иерархия классов
- Объект tView
- Объект tGroup
- Создание объектов
- Рисование объектов
- Передача сообщений объектам
- Создание своего окна
- Организация списков
- Элемент ListBox (строка с памятью)
- Организация окон диалога
- Организация меню
- Строка статуса
- Метод Idle
- Объект tScrollBar
- Дальнейшее развитие оболочки

Причины появления ООП

- Современное программное обеспечение имеет стабильную тенденцию к постоянному усложнению, один программист уже не в состоянии справиться с постоянно возрастающей по объему программой. Поэтому приходится разделять обязанности по созданию программы между коллективом программистов. Однако сложность программы возрастает не линейно, то есть, если один программист пишет программу за пять месяцев, то это не значит, что пять программистов напишут эту же программу за один месяц.
- Из-за существенного усложнения программного обеспечения и увеличения времени его создания резко возрастает его стоимость, что бы это как-то компенсировать фирмы пытаются использовать в программах ранее созданный код.

Причины появления ООП

- ООП меняет традиционный подход к программированию, меняет мышление программиста. Если в традиционном классическом программировании программист сам решал все проблемы: разрабатывал структуру данных, организовывал циклы перебора, поиска нужной информации, то в ООП он выступает в роли диспетчера, который только посылает сообщение – сделать то-то. Нужный объект, получив это сообщение, сам решает, как ему на него реагировать, как решить поставленную задачу. Рассмотрим простой пример. Пусть необходимо послать цветы бабушке на день рождения.

Причины появления ООП

- **I способ решения. Традиционный.**
- Программист разрабатывает структуру данных для представления цветов, посылки, разрабатывает алгоритм поиска пути для передачи посылки до бабушки. На основе структуры данных разрабатывается сам алгоритм:

1. взять деньги;
2. пойти в магазин;
3. купить цветы;
4. пойти на почту;
5. послать цветы;
6. доставить цветы бабушке.

- Все фрагменты программы предельно связаны друг с другом через структуру данных, поэтому использовать данную программу (для посылки открытки) становится уже крайне проблематично.

Причины появления ООП

- **II способ. Решение с использованием ООП.**
- Все действующие в программе объекты (я, цветочница, магазин, цветы и т.д.) разбиваются на группы, выделяются их ключевые свойства. По этим свойствам выстраивается иерархия объектов. Каждый уровень объектов обладает некоторыми своими дополнительными свойствами. Работа программы сводится к взаимодействию между объектами, которое осуществляется посылкой сообщений: я не иду в магазин покупать цветы, а лишь посылаю сообщение владельцу магазина, в котором сказано кому, когда и что я хочу послать. Причем мне совершенно не важно, как это будет сделано, поэтому я не завишу от класса продавцов цветов, наше взаимодействие сводится только к передаче и получению сообщения. Программист может менять любой класс, и это ни как не скажется на другом классе, главное чтобы неизменным оставался формат сообщений.

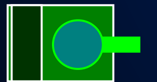
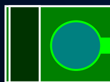
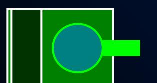
- Главной обязанностью каждого объекта является свойство удовлетворять запросы, содержащиеся в сообщениях. Для их удовлетворения существуют специальные алгоритмы (подпрограммы) - методы. Задача метода не только выполнить запрос, но и скрыть механизм его выполнения от других. Это позволяет не загромождать программу частными алгоритмами, а так же повторно использовать эти методы в других программах. Согласитесь, что мне совершенно не важно, как владелец магазина пошлет цветы бабушке. Этот алгоритм вообще не должен влиять на ход всей программы.
- **Первый принцип ООП** - *действия в ООП инициируются посредством передачи сообщения агенту (объекту) ответственному за его выполнение. Интерпретация сообщения зависит от объекта*.
- Действительно, пусть у нас в программе есть несколько геометрических фигур: круг, квадрат, треугольник. Выполнение приказа "нарисуй себя" зависит от типа объекта.
- ***ООП подразумевает наделение всех объектов свободой действий. Именно сам объект решает, как ему лучше выполнить тот или иной запрос.***

- Представим себе стратегическую игрушку Real time strategy, в которой двигается несколько танков:
- в традиционном программировании программист организует цикл, в котором переберет все танки, для каждого изменяет его текущие координаты на величину DX, затем заново перебирает все танки и рисует видимые на экране;
- в ООП программист пошлет два приказа: СООБЩЕНИЕ(всем танкам, сместиться), СООБЩЕНИЕ(всем танкам, нарисоваться). Каждый танк сам будет изменять свои координаты, сам определять виден ли он на экране и т.п.

Традиционный подход

```
For i:=1 to N do
   Передвинуть i-ый танк
```

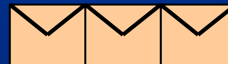
i 3

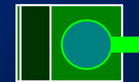
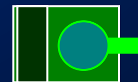
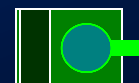




Программист лично не занимается каждым танком, он лишь посылает им сообщение: сместиться. Танк, получив сообщение, самостоятельно выполняет команду

ООП подход

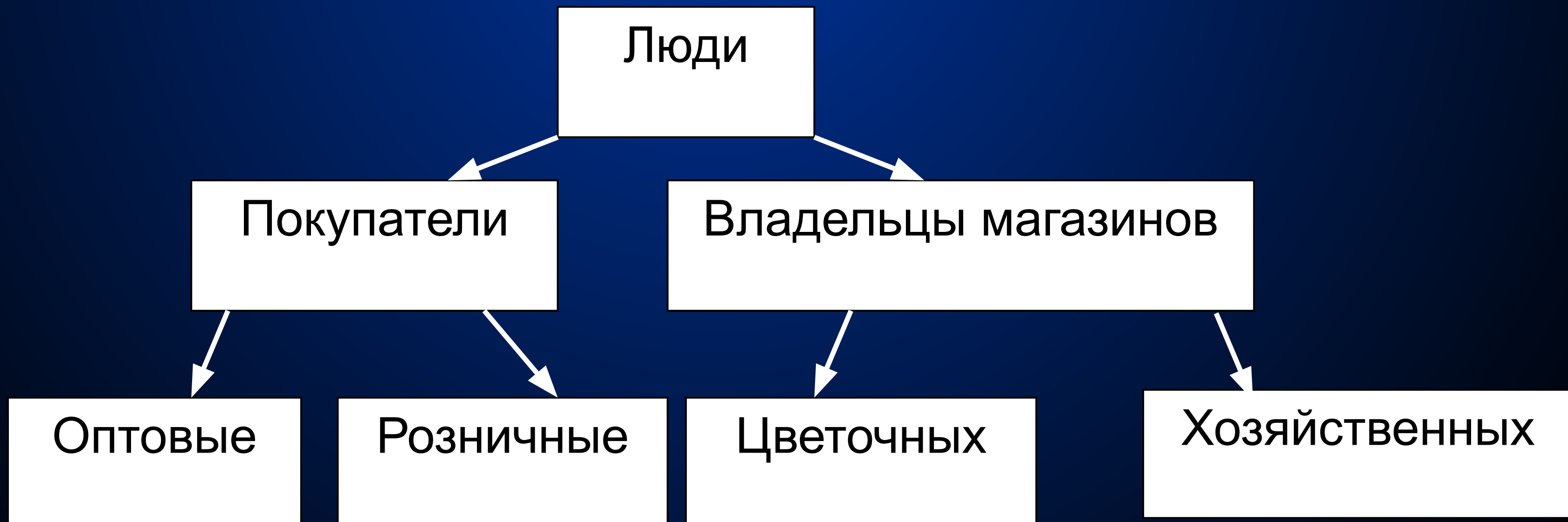
послать сообщение(всем танкам, сместиться)



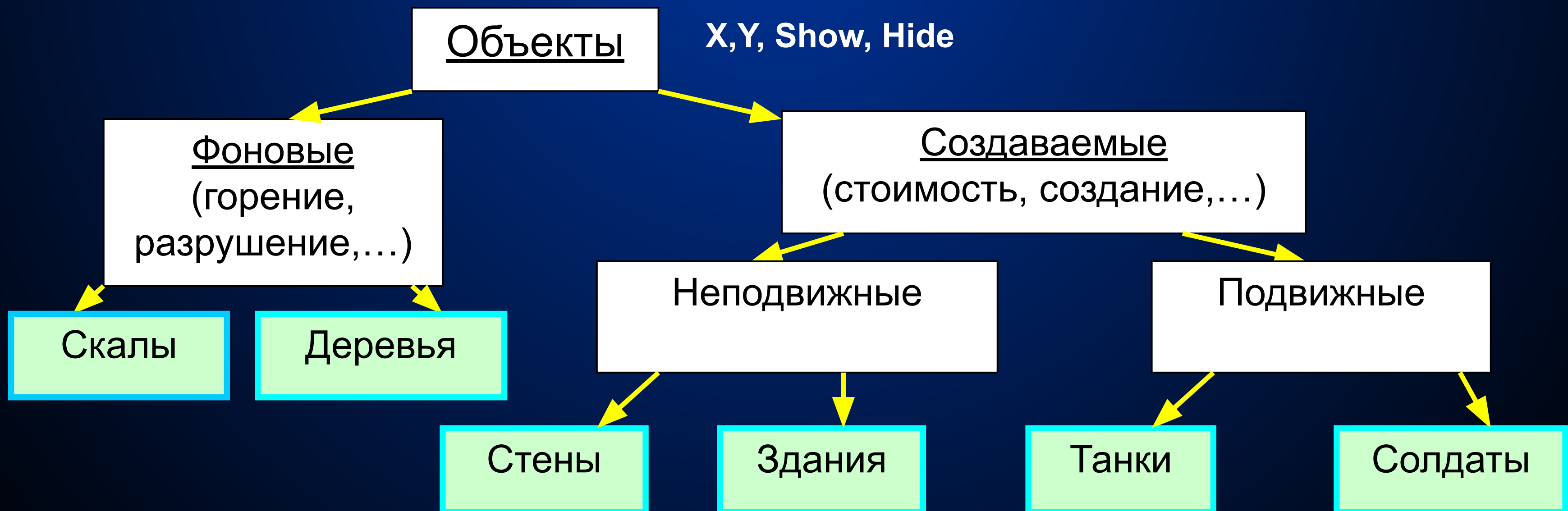
Иерархия классов

- При разработке программы решающую роль играет разделение объектов на подчиненные классы. Задачей данной иерархической структуры является выделение общих свойств и добавление новых свойств, присущих каждому объекту. Это позволяет избежать в дальнейшем дублирование свойств объектов, упростить понимание всей конструкции программы.
- Объекты, существующие в верхней части дерева обладают более общими свойствами, присущими всем объектам потомкам. Каждый последующий класс уточняет предыдущий, делает его более узким, добавляет новые свойства.



Иерархия классов

- Рассмотрим иерархию объектов в любой стратегической игрушке. Реально существующие объекты обведены голубой линией, они имеют физический смысл. Большинство верхних объектов иерархического дерева физического смысла могут и не иметь. Их называют абстрактными объектами. Объекты нижних уровней обладают всеми свойствами верхних плюс некоторыми дополнительными.

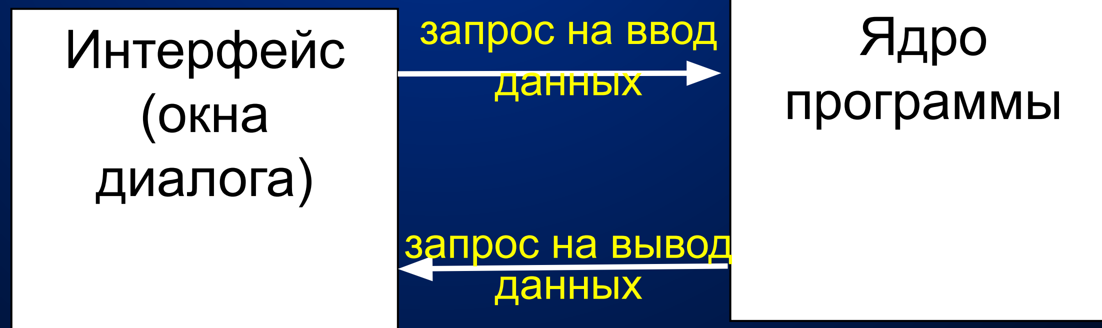


Ответственность программиста

- В ООП каждому объекту программы предоставляется право выбора метода выполнения того или иного запроса. Другие объекты не должны вмешиваться в процесс выполнения задания. Это позволяет уменьшить связь между элементами, что позволяет легко модифицировать программу и повторно использовать ранее созданный код.
- При организации больших проектов в процессе создания программы участвуют несколько человек. Возникает проблема разделения программы на отдельные части и распределение этих частей между программистами. Большой проблемой так же является организация взаимодействия между разными частями программы.
- В разработке сценария работы программы участвуют все программисты, используется метод "мозгового штурма", каждый предлагает свой сценарий, который подвергается конструктивной критике, выявляются слабые места проекта. Авторы аргументировано доказывают преимущества данной модели. Сценарий разрабатывается при активном участии клиента. Он записывает на бумаге все свои требования и пожелания к создаваемой программе. Затем сценарий разбивается на логически законченные компоненты. За каждым компонентом закрепляется определенная обязанность. На данном этапе еще не важно, как данный компонент будет реализован. При разбиении программы руководствуются двумя моментами:
- компонент должен иметь небольшой объем четко определенных обязанностей;
- компонент должен минимально взаимодействовать с другими компонентами.

Готовность к изменениям

- Каждый программист разбивает список своих подпрограмм на модули. Основная задача при этом - организовать минимальное взаимодействие между модулями. Любое изменение в программе должно затрагивать максимум 2 модуля. Наиболее часто источником изменений служит интерфейс, поэтому при разработке программы его необходимо отделить от решающего ядра программы. Программа должна быть максимально независима от аппаратуры. Поэтому не рекомендуется в программах использовать ассемблерные вставки, которые напрямую обращаются к аппаратуре (винчестеру).
- При такой организации решающую роль играет механизм запроса, его структуру менять нельзя, замена же интерфейсной части ни как не отразится на решающем ядре.



Зацепление и связность

- Каждый компонент программы характеризуется:
- ***поведением*** - набором действий, которые может выполнить данный компонент;
- ***состоянием*** - его содержимым (значением его переменных (полей)).
- *Зацепление* - это взаимодействие между компонентами, их взаимное влияние. Оно должно стремиться к минимуму. Минимальное зацепление позволяет легко модифицировать программу, то есть изменение одного компонента не должно приводить к изменению другого.
- *Связность* – характеризует, насколько компонент образует логически законченную осмысленную единицу. Связность компонент должна стремиться к максимуму.

Разделение интерфейса и реализации

- Интерфейс - это внешняя сторона программы, а реализация - как это все сделано. При разработке программы используется принцип Д. Парнаса:
- 1) *разработчик программы должен предоставлять пользователю всю информацию, которая необходима для эффективного использования программы и ничего кроме этого;*
- 2) *разработчик программы должен знать только требуемое поведение компоненты и ничего кроме этого*.
- Использование этих двух правил позволяет уменьшить зацепление между компонентами и отделить интерфейс (правила взаимодействия между программой и пользователем) от реализации программы.

Методы и данные

- Чтобы описать объект на языке Паскаль используют слово object. Например,

```
Type  Figure=object
             x,y: integer;
             Col:byte;
             procedure Draw;
             procedure Move(nx,ny:integer);
          end;
```

- Подпрограммы, которые описаны в объекте, называются методами. Объекты обладают свойством инкапсуляции, то есть включают в себя данные (как записи) и методы их обработки. Это позволяет уменьшить зацепление между компонентами. Например, объект Tank обладает всеми характеристиками танка (координатами, скоростью, силой и дальностью стрельбы и так далее) и одновременно процедурами работы с танками. Изменение объекта Tank ни как не отражается на других объектах, не являющихся его потомками.

Сообщения

- Работа любой программы, написанной на ООП, основана на передаче сообщений по цепочке объектов. Адресат, получая сообщение, изымает его. Реакция на полученное сообщение не одинакова и зависит от типа получателя (например, сообщение "нарисуй себя" должно по-разному выполняться для каждого из объектов).
  В сообщении обычно говорится:
- кому предназначено (всем или указатель на адресата);
- тип сообщения (клавиатура, мышь, внутреннее событие);
- список аргументов.
- Передача сообщений обычно сопровождается вызовом подпрограммы:
- p.Draw(...); -статический объект p;
- p^.Draw(...); -динамический объект p.
- Message(кому, что)

Размещение и инициализация объектов

- Объекты могут быть созданы и размещены либо в статической, либо в динамической памяти. Предпочтение отдают последней, так как это позволяет:
- создавать и удалять объекты по мере необходимости;
- создавать виртуальные методы. Создание кода вызова такого метода (call [addr]) происходит не в момент компиляции, а в момент выполнения программы, когда становится известно, какого типа данный объект. Для этого у каждого объекта строится специальная таблица виртуальных методов. Ее настройкой занимается специальная подпрограмма, называемая конструктором. Кроме этого конструктор может инициализировать поля объекта, выделять необходимую объекту память и тому подобное.
- По традиции конструктор называют Init. Конструктор может вызываться явно и автоматически.

```
Type
   pObj=^tObj;
   tObj=object
      constructor Init;
      destructor Done;
      ...
     end;
Var a:pObj;
```

- 1) явный вызов: new(a); a^.Init;
- 2) автоматический вызов: new(a, Init);
- Деструктор занимается ликвидацией объекта. Например, деструктор окна удаляет подэлементы окна, освобождает выделенную конструктором память и т.п. Dispose(a, Done);

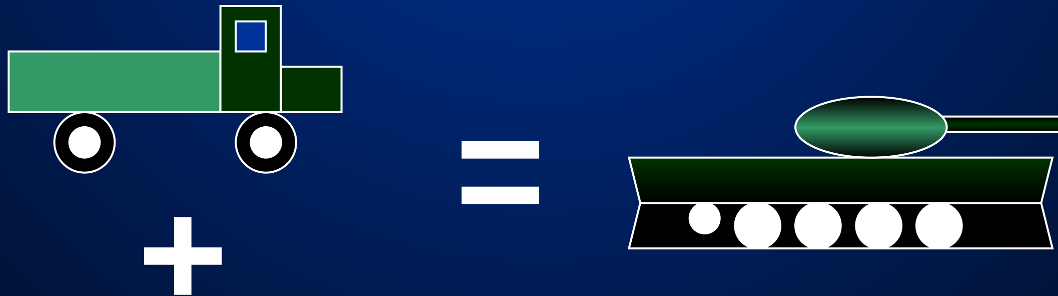
Наследование

- Наследование означает, что данные объекта и его поведение передаются (наследуются) дочерним классом от родительского. Дочерний класс всегда расширяет набор полей и методов родительского класса. Кроме того, дочерние подклассы могут переопределять (заменять) некоторые методы родительского класса или дополнять их.
- Преимущества наследования:
- 1) повторное использование программы или ее алгоритмов;
- 2) повторное использование кода без общей перекомпиляции (TPU модули);
- 3) согласование интерфейса.

- **Пример 1**: Пусть у нас имеется объект Automobile, который полностью характеризует автомобиль и предоставляет методы работы с объектом (перемещение, рисование, выбор пути среди препятствий и т. д.). Чтобы создать объект Tank вовсе не обязательно заново описывать все алгоритмы, достаточно обратить внимание на то, что автомобиль и танк схожи по большинству свойств: одинаково перемещаются по экрану, одинаково ищут путь, имеют одинаковые характеристики: скорость, координаты, координаты пункта назначения, угол поворота и так далее. Объект Tank можно создать, как потомок от объекта Automobile, при этом новый создаваемый объект наследует от родительского все поля и методы. Некоторые методы полностью сохраняют без всяких изменений, например, выбор пути, некоторые дополняют, некоторые полностью заменяют, например, рисование танка на экране. Объект Tank получает дополнительные поля и методы, так как у танков по сравнению с автомобилями появляются новые свойства (дальность стрельбы, выбор цели и т.д.).

Type
Возможность стрелять

```
  pAutom=^tAutom;
  tAutom=object
  x,y:integer;{координаты}
  dx,dy:integer;{направление}
  procedure Move;
  procedure Show;
  ...
    end;
  tTank=object(tAutom){наследник от класса автомобиль}
  fireF:boolean;{флаг стрельбы}
  procedure Fire; {стрельба}
  procedure FindTarget; {поиск цели}
  procedure Show;
  ...
    end;
```



Возможность стрелять

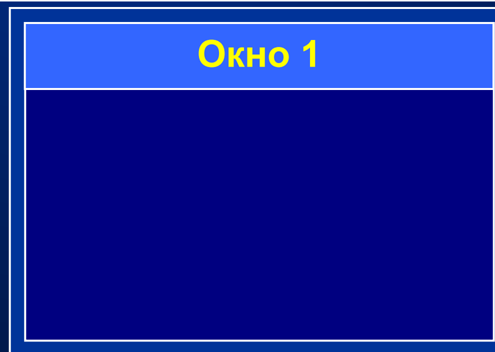
Пример 2: Пусть у нас имеется TPU модуль, который позволяет нарисовать окно. Оно нас не устраивает тем, что не имеет заголовка. Что же делать? Заново создавать новый модуль? Нет! Достаточно создать свой объект tMyWind и подкорректировать вывод изображение окна.

```
Type pMyWind=^tMyWind;
     tMyWind=object(tWindow) {tWindow - родительский класс}
       constructor Init;
    destructor Done;
    procedure Draw(...);
      end;
```

Вызов метода предка осуществляется при помощи ключевого слова inherited, причем вызов может осуществляться в любой точке метода.

```
procedure tMyWind.Draw(...);
begin
  Inherited Draw(...); {вызов метода предка, который нарисует окно}
  нарисовать заголовок
end;
```

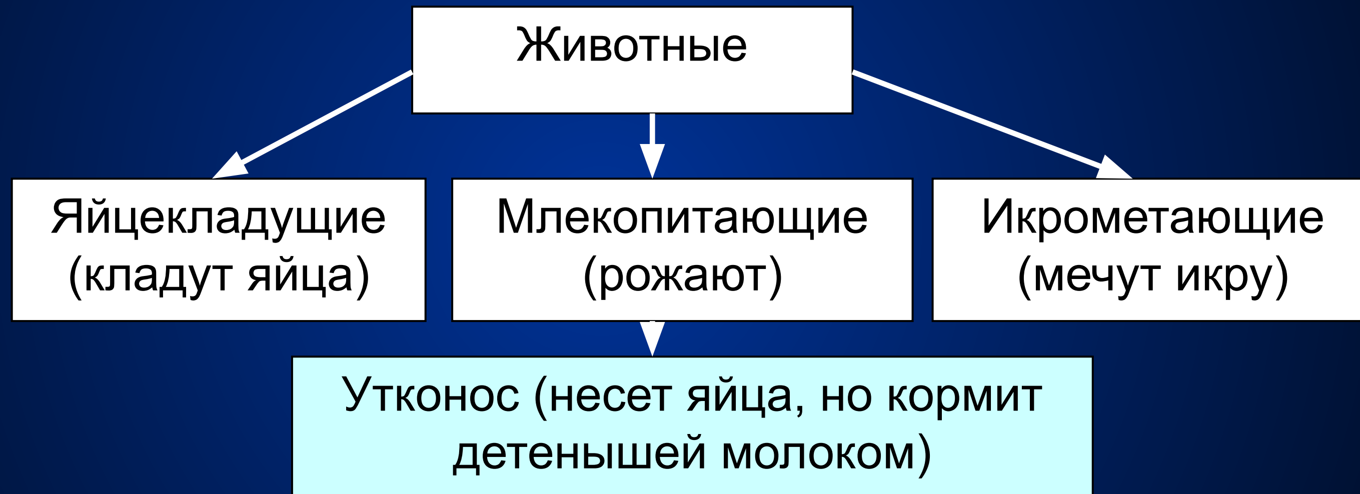
Окно базового класса tWindow



Окно с заголовком класса tMyWind

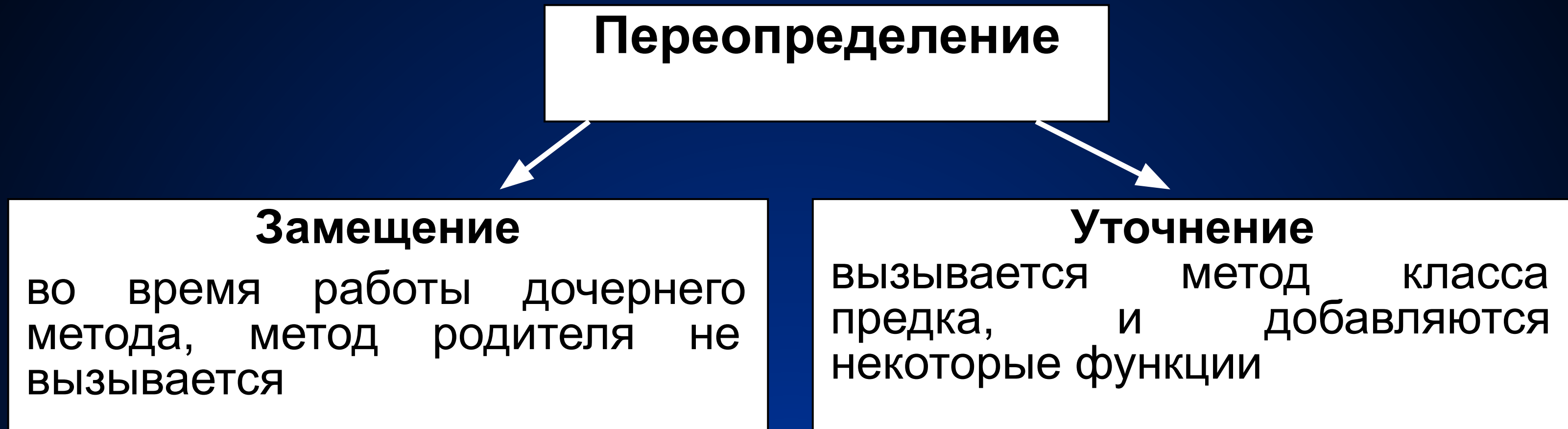
Замещение и уточнение

- До сих пор мы просто предполагали, что дочерние классы просто добавляют некоторые свойства, тем самым, расширяя родительский класс. Однако дочерний подкласс может видоизменять (переопределять) некоторые методы родительского класса. Рассмотрим следующую схему:



- Куда отнести объект "Утконос", по всем характеристикам он относится к млекопитающим, но у него другой метод размножения - несет яйца. Поэтому, объект "утконос" наследует все свойства млекопитающих, но метод размножения должен быть изменен.

- Существует два способа переопределения методов:



- При переопределении методов в Паскале необходимо выполнить следующее:
- имя методов должно совпадать;
- параметры могут не совпадать;
- метод может быть виртуальным.
- При передаче сообщения объекту поиск методов начинается с проверки класса, которому принадлежит данный объект. Если в данном классе есть метод с таким именем, то вызывается именно он. Если такого метода нет, то просматривается родительский класс и т.д.

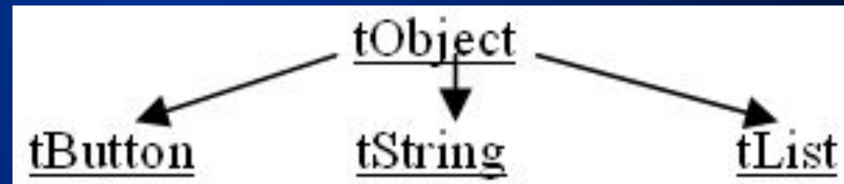
Следствие наследования

- Рассмотрим два способа размещения объектов: в динамической памяти и в стеке. Использование стека эффективнее динамической памяти, так как позволяет автоматически создавать и ликвидировать переменные. Однако возникает серьезная проблема: необходимо заранее знать размер памяти, необходимой объекту, причем, до объявления объекта в момент входа в подпрограмму. Из данной ситуации есть три выхода:
- а) выделять память, достаточную для базового класса;
- б) выделять максимальный размер, достаточный для любого класса;
- в) разместить указатель в стеке, а сам объект - в динамической памяти. Это позволит выделить память под объект в тот момент, когда его размеры уже известны.
- Над указателем возможны три операции:
- присваивание x:=y;
- сравнение x<>y; x=y;
- в некоторых случаях объекту необходимо настроить указатель на себя. Для этого в Паскале введена константа @Self, которая равна адресу объекта, которому принадлежит данный метод.
-   **if** p=@Self
-   **then** p указывает на меня
-   **else** p не указывает на меня

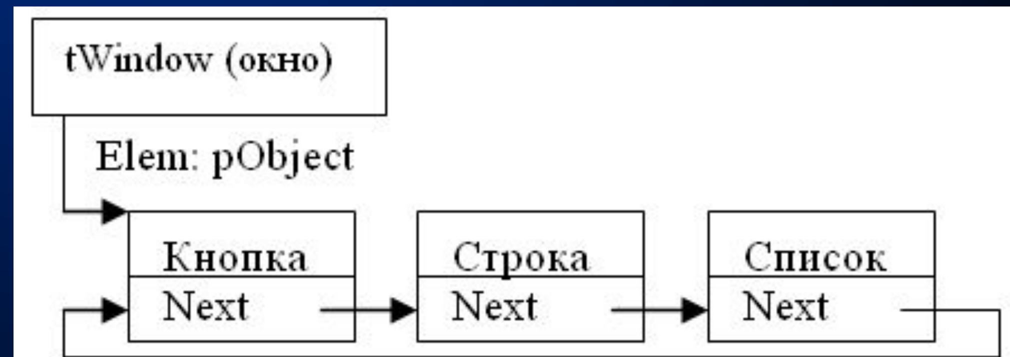
Преобразование типов. Полиморфизм

- Пусть нам необходимо написать программу, в которой некое окно владеет несколькими элементами: строкой ввода, кнопкой, списком. Все эти элементы равноправны и подчинены окну. Чтобы объединить их в один список, необходимо чтобы все эти элементы были одного типа. В рамках традиционного Паскаля это приводит к неэффективному использованию памяти, так как все элементы должны будут иметь одинаковые поля, некоторые из которых вообще не будут использоваться.
- В рамках наследования в ООП применяют следующий метод:

1. создают общий родительский класс, в котором размещают единые для всех объектов характеристики (tObject)
2. на основании класса tObject создаются дочерние классы tString (строка), tButton (кнопка), tList(список).



3. объект tWindow имеет специальный указатель Elem типа pObject, который указывает на список подэлементов окна. Каждый подэлемент, в свою очередь, имеет ссылку Next на следующий подэлемент в кольцевом списке.



- **Var w:pWindow;{*указатель на окно*}**
- **p:pObject;**
- **begin**
- **p:=w^.Elem^.Next;** …
- p указывает на строку, но компилятор этого не знает, он знает только то, что объект p^ владеет полями и методами класса tObject и все! Для обращения к полям строки необходимо явное преобразование типов: pString(p)^.Title:='Значение строки';
- Любая ошибка при выборе объекта не того типа не может быть обнаружена на этапе компиляции. Полиморфизм заключается в том, что во время выполнения программы полиморфные переменные могут хранить значения различных типов (указывать на объекты различных типов). Полиморфизм предусматривает передачу одноименным функциям различных параметров. При вызове процедуры рисования p^.Draw будет вызываться метод tObject.Draw (если метод не виртуальный), и Draw того объекта, на который указывает указатель, если он виртуальный.

Отложенные методы

- Пусть в родительском классе вводится метод, которым будет обладать любой дочерний класс, например, метод Show. В родительском классе этот метод не может быть реализован. Такие методы называются отложенными.

```
procedure tObject.Show; virtual;
begin
end;

procedure tButton.Show; virtual;
begin
   Bar(...); OutTextXY(...);...
end;

procedure tWindow.Show; virtual;
Var p:pObject;
begin
      Нарисовать себя (окно);
      p:=Elem;{Нарисовать свои подэлементы (обычно так не делают)}
      while p<>nil do
       begin
         p^.Show;
         p:=p^.Next
       end;
end;
```

Строка ввода

Список:

( ) вариант 1

Очень важно!
Переменная p имеет тип pObject.
Подумайте, к чему приведет вызов метода p^.Show?

Очень важно!
Если метод Show не будет виртуальным, то будет вызываться метод tObject.Show и ничего рисоваться не будет. Если метод Show будет виртуальным, то через переменную p будет вызываться метод Show того класса, на который указывает переменная p.

сообщение своим элементам. Каждый рисует себя сам.

Пусть имеется окно с тремя элементами: строкой ввода, кнопкой и списком.

Строка ввода

Список:

( ) вариант 1

( ) вариант 2

(*) вариант 3

кнопка

```
procedure tWindow.Show; virtual;
Var p:pObject;
begin
      Нарисовать себя (окно);
      p:=Elem;
      while p<>nil do
       begin
         p^.Show;
         p:=p^.Next
       end;
end;
```

Программа Figure

- В основе ООП лежит понятие объекта (Object), сочетающего в себе данные и действия над ними. Объект в некотором роде похож на запись, но сочетает в себе не только данные, но и подпрограммы их обработки, называемые методами. Таким образом, в объекте сосредоточены его свойства и поведение.
- ООП характеризуется тремя новыми свойствами: инкапсуляция (encapsulation), наследование (inherited), полиморфизм (polimorphism).
- **Инкапсуляция** означает объединение в одном объекте данных и действий над ними.
- **Наследование** позволяет создавать иерархию объектов, начиная с некоторого простого первоначального предка и заканчивая более сложными, но включающими свойства предшествующих элементов. Каждый потомок несет в себе характеристики своего предка, а также обладает своими собственными полями и методами. При этом наследуемые поля и методы нет необходимости описывать еще раз.
- **Полиморфизм** означает, что для родственных объектов можно задать единый класс действий (например, перемещение по экрану любой геометрической фигуры). Затем для каждого конкретного объекта составляется своя подпрограмма, выполняющая это действие непосредственно для данного объекта. Причем все подпрограммы могут иметь одно и тоже имя. Когда потребуется перемещать конкретную фигуру, из вашего класса будет выбрана соответствующая подпрограмма.

Понятие объекта

Тип-объект в Паскале напоминает собой тип-запись. После зарезервированного слова object перечисляются имена полей с указанием их типов, а также имена подпрограмм их обработки, называемых методами, после чего пишется **end**.

```
Type Figure=object
        X,Y:integer;
        Col:byte;
        procedure Move(dx,dy:integer);
        procedure Draw;
    end;
```

В этом примере поля X и Y задают координаты фигуры, Col ее цвет, а метод Draw позволяет нарисовать фигуру на экране, и т. д.

Естественно, что затем все используемые методы должны быть описаны:

```
procedure Figure.Draw;
begin
   ...
end;
```

Указывать имя типа необходимо потому, что методы разных типов могут иметь одинаковые имена. Некоторые объекты программы, особенно находящиеся в начале иерархического дерева, могут не соответствовать реальным объектам. Например, procedure Figure.Draw не может быть реализована, так как каждая геометрическая фигура должна рисоваться по-своему. Однако выделение общих свойств бывает очень удобно, так как это позволяет избежать многократного дублирования при описании конкретных объектов, объединить разные объекты в один список.

Инкапсуляция

Под термином "инкапсуляция" понимается совмещение в одном объекте, как параметров, так и действий над ними. При этом включенные в объект подпрограммы (методы), как правило, оперируют с данными этого объекта или обращаются к методам предков. Это позволяет объединить в одном месте все свойства объекта, что облегчает понимание работы программы, ее отладку аи модификацию. Как правило, к данным объекта извне непосредственно не обращаются, хотя это возможно. Для обращения используют соответствующие методы, например, SetColor - установить новый цвет объекта:

```
Procedure Figure.SetColor(NewCol:byte);
    begin
     Col:=NewCol;
    end;
```

Такое опосредованное обращение к данным позволяет избежать во многих случаях нежелательного изменения параметров. В Паскале с этой целью используется специальное слово Private (приватный), в принципе запрещающее непосредственное обращение к тем или иным данным и методам объекта вне модуля, в котором они описаны. Данные и методы, к которым разрешено обращение, описываются в общей части, после слова public.

```
Type Figure=object
       private
        X,Y:integer;
        Col:byte;
       public
        procedure Move(dx,dy:integer);
        procedure Draw;
        procedure SetColor(NewCol:byte);
      end;
```

Доступ к полям X, Y, Col из программы не возможен, хотя любой метод Figure может к ним обратиться.

Наследование

Пусть нам необходимо написать программу, в которой по экрану в случайном направлении перемещаются различные фигуры (окружность, точка, квадрат). Все геометрические фигуры обладают общими признаками: координатами, процедурой перемещения фигуры, процедурой смены ее цвета. Логично объединить все эти признаки в абстрактный класс Figure, который задает общие характеристики. Объект окружность (Krug) наследует общие признаки от объекта Figure (предка).

```
Type Krug=object(Figure)
         R:integer;
         procedure SetRadius(NewRad:integer);
       end;
```

Объект Krug наследует поля X,Y,Col и методы SetColor, Draw, и т. д. от своего предка объекта Figure, а также обладает своими полями и методами, которые характеризуют его как конкретную фигуру - круг. Некоторые методы придется переопределить, так как они не могут быть реализованы у предка.

```
 procedure Krug.Draw;
 begin
    SetColor(Col);    Circle(x,y,R)
 end;
```

```
Kvadr=object(Figure)
    SizeX:integer;{Сторона квадрата}
         procedure Draw;virtual;
          ...
        end;
```

Можно аналогично ввести объект квадрат Kvadrat:

```
procedure Kvadr.Draw;
begin {A=половине длины стороны}
  ... A:=SizeX div 2; Rectangle(x-A,y-A,x+A,y+A);
end;
```

Полиморфизм

- Все вращающиеся фигуры обладают одними и теми же характеристиками (скорость поворота, цвет, размер) и над ними возможны одинаковые по смыслу действия: нарисовать фигуру, стереть, установить новый цвет. Все эти действия почти не зависят от типа фигуры (точка, круг, квадрат). В ООП разрешается подпрограммам разных объектов давать одинаковые имена:
- Draw - рисует объект,
- Hide - стирает объект,
- SetColorMy - устанавливает новый цвет объекта.
- В возможности иметь несколько подпрограмм с одинаковыми именами и заключается полиморфизм ООП Turbo Pascal. Вопрос, какая же конкретно подпрограмма будет использована в том или ином случае, определяется типом конкретного объекта, использующего эту подпрограмму. Так, если вызывается подпрограмма Draw объекта Krug, то будет нарисован круг, если же это подпрограмма объекта типа Kvadr, то будет нарисован квадрат и т. п. Причем список формальных параметров у этих подпрограмм может отличаться.

Виртуальные методы

В ряде случаев при описании тех или иных объектов приходится писать методы, также схожие друг с другом и отличающиеся только отдельными деталями. Так методы Draw у объектов круг и квадрат работают по одинаковому алгоритму:

1. нарисовать себя;
2. нарисовать все свои подэлементы.

Рисование подэлементов реализуется одинаково для любой фигуры и не зависит от ее формы:

```
procedure tGroup.Draw;
Var t:pFigure;
begin
    t:=Elem; {встать на начала списка подэлементов }
    While t<>nil do
     begin
        t^.Draw; {нарисовать подэлемент}
        t:=t^.Next {перейти к следующему}
     end;
end;
```

Встаем на начало списка t:=Elem, затем, в цикле пока t<>nil, рисуем объект и сдвигаемся к следующему элементу. Рисование же самой фигуры отличается. Поэтому рисование подэлементов всех фигур реализовано в абстрактном классе tGroup - группа. Именно этот класс отвечает за все свойства группы: добавить объект в группу, опросить группу и т.д.

Каждый объект потомок может вызвать метод родителя в любой момент своей работы:

```
procedure Krug.Draw;
begin
  SetColor(Col+Num); {Нарисовать себя}
  Circle(x,y,Rad);
  Inherited Draw;{Вызвать метод предка для прорисовки подэлементов
   данной фигуры}
end;
```

Все элементы программы (круги, квадраты, точки) находятся в одном списке элементов типа pFigure. Какой же метод должен вызваться при вызове t^.Draw, где t: pFigure? Ответ очевиден - Figure.Draw, но этот метод не может ничего нарисовать, так как является отложенным абстрактным методом и содержит только **begin end**. Для выхода из данной ситуации метод Draw во всех объектах должен быть объявлен виртуальным (Virtual), что позволит вызвать метод Draw именно того объекта, к которому мы в данный момент обращаемся через любой указатель. Если метод где-то был объявлен виртуальным, то и все другие методы с тем же заголовком должны быть объявлены виртуальными, а списки формальных параметров должны быть эквивалентными.

Метод, использующий виртуальные методы, должен быть расположен в доступном всем другим методам месте. Для вызова метода предка перед его именем ставится зарезервированное слово **Inherited** <имя предка>.

Основное отличие виртуальных методов заключается в том, что необходимые связи с ними в программе устанавливаются не на этапе компиляции или компоновки, а на этапе выполнения программы. С этой целью у объекта создается таблица виртуальных методов. При выполнении программы в случае необходимости из этой таблицы выбирается адрес соответствующего варианта виртуального метода с целью использования именно этого варианта. Однако для использования такой таблицы ее необходимо предварительно заполнить нужными адресами. Для этой цели используют специальные методы, называемые конструкторами (constructor). Он должен быть исполнен до вызова виртуального метода. Основное назначение конструктора - заполнение таблицы виртуальных методов, однако, он может выполнять и другие действия по инициализации создаваемого объекта (выделять нужную память, присваивать начальные значения переменным и др.).

```
    Constructor Figure.Init(Xn,Yn,Radn:integer;Coln:byte);
begin
  X:=Xn;Y:=Yn; Col:=Coln; Radorb:=Radn;
  Next:=nil;    Elem:=nil;    Owner:=nil;
end;
```

Для упрощения программы стандартная процедура new в Паскале была дополнена вторым необязательным параметром - именем конструктора создаваемого объекта. После создания динамического объекта автоматически будет вызван конструктор:

New(Desk, Init(GetMaxX div 2,GetMaxY div 2,300,0));

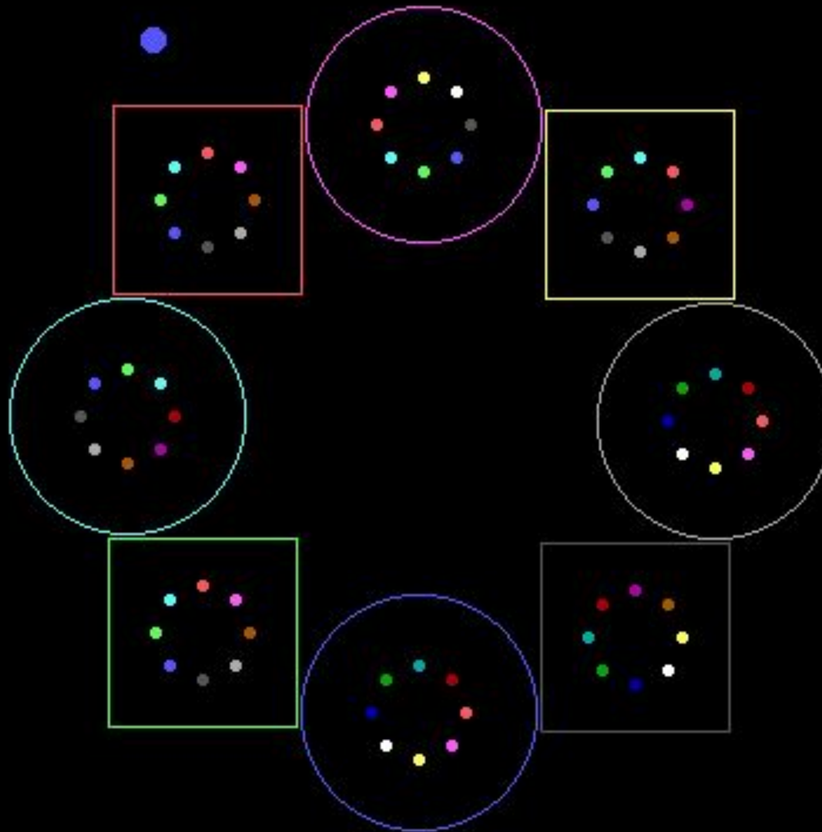
Аналогично при удалении объекта из динамической памяти процедурой Dispose можно использовать подпрограмму, называемую деструктором (Destructor), предназначенной для проведения операций, связанных с ликвидацией объекта (освобождение выделенной памяти, исключение себя из списка и т. п.). Деструктор, как правило, наследуется потомком и почти всегда бывает виртуальным

Одной из причин использования виртуальных методов является возможность упрощения программы за счет устранения повторяющихся частей. Другая, более существенная причина заключается в возможности модификации виртуальных методов или расширение функций программы без ее глобальной перекомпиляции (использование стандартных TPU-модулей). Внести какие-либо изменения в обычный метод невозможно, для виртуального же метода можно либо внести какие-либо изменения, либо полностью его заменить. Так метод Draw был переопределен для каждой конкретной фигуры, так как на уровне абстрактного объекта figure невозможно нарисовать разные геометрические фигуры.

```
procedure krug.Draw;
begin
 x:=owner^.x+round(radorb*cos(uu));
 y:=owner^.y+round(radorb*sin(uu));
 setcolor(col+num); circle(x,y,rad);
 Inherited Draw;
end;
```

Здесь введен новый тип krug, являющийся наследником типа tGroup. Его виртуальный метод Draw сначала рисует окружность нужным цветом и в нужных координатах, а затем пытается нарисовать свои подэлементы, вызвав для этого метод предка.

Программа «Фигура»



Вокруг центра экрана вращаются круги и квадраты. Каждый круг или квадрат содержит вращающиеся точки

Если щелкнуть мышкой по любому объекту – он отреагирует, меняя свой цвет. Кнопки «+» и «-» меняют скорость вращения системы.

По экрану в режиме «свободного полета» перемещаются свободные точки. При столкновении с кругами они изменяют направление своего движения на противоположное, а на квадраты никак не реагируют.

DeskTop -является рабочей областью программы. Она обладает методом Run - главным рабочим циклом обработки событий (смена угла поворота всей системы).

```
Procedure DeskTop.Run;
Var    Ug:real; Page:byte; e:tEvent;
 begin
 Ug:=0;Page:=0;SetColor(15);  HideMouse;
 OutTextXY(10,10,'"+"-увеличить скорость,"-"- уменьшить скорость.');
 Repeat
    HideMouse;   Hide;{стереть все объекты}
    SetUgol(Speed); {изменить угол у объектов}
    Move; {сместить все объекты}
    SetColorMy(random(15));{установить цвет объектам}
    HideMouse;    Draw; ShowMouse;    {нарисовать объекты}
    GetEvent(e);    {получить событие}
    HandleEvent(e); {обработать событие}
 Until Quit;
end;
```

В цикле, пока не взведен флаг Quit, выполняются следующие действия:

1) стереть все подэлементы (Hide;);
2) изменить угол поворота(SetUgol(Speed););
3) сдвинуть все объекты (Move);
4) если возможно, то все сменить цвет.
5) нарисовать свои подэлементы (Draw;);
6) получить текущие события (GetEvent(e););
7) обработать текущие события (HandleEvent(e);)

- Причем подэлементами рабочей области DeskTop являются объекты типа Krug, Kvadr и FreePoint, первые два, в свою очередь, представляют собой подсистему вращения объектов точка (Point) вокруг их центра. Для установления столь сложных связей владелец-подэлемент служат три указателя:
- 1) Elem - указывает на список подэлементов данного объекта;
- 2) Next - указывает на следующий объект данного уровня;
- 3) Owner - указывает на владельца данного элемента
- Установленные связи показаны на схеме.
- В начале программы все объекты создаются и размещаются в динамической памяти при помощи оператора New, при этом, автоматически вызывается конструктор каждого объекта, он инициализирует объект и заполняет таблицу виртуальных методов. Метод Insert позволяет разместить созданный объект в качестве подэлемента любого объекта. Он устанавливает соответствующие связи. Обычно все элементы программы создаются в конструкторе DeskTop’a. Для этого пользователь на основе объекта tDeskTop создает свой объект tMyDeskTop и переопределяет у него конструктор. В нем он создает и вставляет в список все необходимые элементы.

```
Type pMyDesk=^tMyDesk;
     tMyDesk=object(DeskTop)
        constructor Init(Xn,Yn,Radn:integer;Coln:byte);
     end;
Constructor tMyDesk.Init(Xn,Yn,Radn:integer;Coln:byte);
Var i:integer; ug2,ug1:real; rr,j:integer; pp:pGroup;
begin
  Inherited Init(Xn,Yn,Radn,Coln);{вызвать конструктор предка для
   инициализации полей}
  for i:=1 to 3+random(5) do {создать несколько точек}
    insert(new(pFreePoint,  Init(random(640),random(480))));

  Ug1:=0;RR:=150;
  for i:=1 to Max do
   begin {Создать подэлементы круг или квадрат}
    if odd(i)
    then pp:=New(pKrug,Init(x+Round(rr*cos(Ug1)),Round(rr*sin(Ug1)),
                 rr-25,50,8+i,Ug1))
    else pp:=New(pKvadr,Init(Round(rr*cos(Ug1)), Round(rr*sin(Ug1)),rr-20,80,Ug1));
    Ug2:=0;  {Создать подэлементы точки для объекта круг или квадрат}
    for j:=1 to max do
     begin
       pp^.Insert(New(pPoint, Init(pp^.x+Round(20*cos(Ug2)),
                 pp^.y+Round(20*sin(Ug2)),20,Ug2)));
       Ug2:=Ug2+(2*Pi/Max)
     end;
    Insert(pp);{Разместить круг или квадрат в подэлементах  рабочей области}
    Ug1:=Ug1+(2*Pi/Max)
   end;
end;
```

Рисование всех объектов

Рассмотрим работу подпрограммы Draw:

1) в методе Desk^.Run вызывается метод Draw, он является потомком метода tGroup.Draw, который вызывает метод Draw для всех подэлементов рабочей области DeskTop. Это приводит к вызову соответствующего метода объекта Kvadr, так как он является первым элементом рабочей области. Он обладает следующим методом Draw:

```
procedure Kvadr.Draw;
 Var x1,y1:Longint; uu:real;
 begin
   uu:=Ugol+myUg;
   x:=Owner^.x+round(Radorb*cos(Uu)); y:=Owner^.y+round(Radorb*sin(Uu));
   SetColor(Col+Num);
   Rectangle(x-SizeX div 2,y-SizeX div 2, x+SizeX div 2,y+SizeX div 2);
   Inherited Draw; {вызов предка tGroup.Draw}
 end;
```

Сначала рисуется квадрат, а затем вызывается виртуальный метод предка, который выглядит следующим образом:

```
procedure tGroup.Draw;
Var t:pFigure;
begin
   t:=Elem;
   While t<>nil do
    begin
      t^.Draw;
      t:=t^.Next
    end;
  end;
```

2) Метод предка tGroup.Draw вызывает методы Point.Draw для всех подэлементов квадрата.

```
procedure Point.Draw;
Var  uu:real;
begin
   uu:=-2*Ugol+myUg; SetColor(Col+Num);
   x:=Owner^.x+round(Radorb*cos(Uu));
   y:=Owner^.y+round(Radorb*sin(Uu));
   SetFillStyle(1,Col+Num);FillEllipse(x,y,2,2);
end;
```

Так как точка является потомком объекта Figure, а не tGroup, то подэлементов у нее нет. Следовательно, процедура рисования точки завершаетсяи возвращает управление tGroup.Draw. Затем метод tGroup.Draw рисует следующую точку и т. д. После завершения этого процесса управление возвращается методу DeskTop.Draw, он рисует свой второй элемент - Круг, который рисует себя, свои подэлементы и передает управление дальше по цепочке. В конце списка размещается объект FreePoint, он отличается по своей структуре от предыдущих и имеет свой метод Draw. Так как методы Draw виртуальные, то происходит вызов именно того метода, которого нужно. В конце концов, все элементы нарисуют себя. Рассмотрим еще раз схему размещения объектов.

DeskTop

Elem

<u>Квадрат</u>
Next
Elem

<u>Круг</u>
Next
Elem

. . .

<u>Круг</u>
Next=nil
Elem

свободные точки

Список точек

Чтобы нарисовать все элементы, достаточно вызвать метод DeskTop.Draw.
DeskTop.Draw

```
TGroup.Draw
    P:=Elem;
    While p<> nil do
        Begin
          P^.Draw;
          P:=p^.next
        End;
```

Цикл по элементам DeskTop (кругам и квадратам)

```
tKrug.Draw
нарисовать круг
Inherited Draw
```

TGroup.Draw
Цикл по элементам круга или квадрата, т.е. точкам.

```
While p<> nil do
  Begin
     P^.Draw;
     {Point.Show}
     p:=p^.Next…
  End;
```

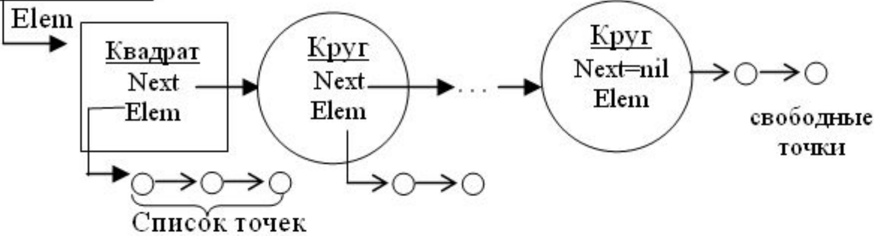
```
Procedure DeskTop.Run;
 …
 repeat
   Move;
   Draw;
 until quit;
end;
procedure tGroup.Draw;
begin
  t:=Elem;
  while t<>nil do
   begin
    t^.Draw;
    t:=t^.Next
   end;
end;
```

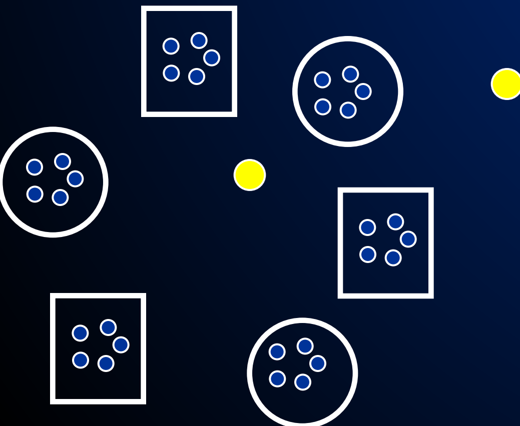
```
procedure Krug.Draw;
begin
 if RedGrenState=0
 then Circle(x,y,Rad);
 else FillEllipse(x,y,Rad,Rad);
 Inherited Draw;
end;

procedure Kvadr.Draw;
begin
  if RedGrenState=0
  then Rectangle( …);
  else Bar( … );
  Inherited Draw;
 end;
```

```
procedure Point.Draw;
begin
  SetColor(Col);
  SetFillStyle(1,Col);
   FillEllipse(x,y,2,2);
end;

procedure FreePoint.Draw;
begin
  SetColor(Col);
  SetFillStyle(1,Col);
  FillEllipse(x,y,Rad,Rad);
end;
```





Методы Krug.Draw или Kvadr.Draw являются потомками tGroup и вызов метода предка приводит к вызову метода tGroup.Draw, который прорисовывает подэлементы (точки). Точки не являются потомками группы, то есть просто рисуют себя

Метод Insert

Метод Insert позволяет разместить созданный объект в качестве подэлемента любого объекта. Этим методом обладают все объекты типа tGroup и их потомки. Выбор типа списка (кольцевой или обычный, заканчивающийся nil) – не принципиален и зависит от задачи и Вашего вкуса.

```
procedure tGroup.Insert(p:pFigure);
begin
   p^.Owner:=@Self;{Вставляет элемент p первым в список Elem}
   p^.Next:=Elem;
   Elem:=p
end;
```

Процедуре Insert в качестве параметра передается указатель P на объект типа Figure или его потомок, вход в список подэлементов Elem объект берет свой. Следовательно, крайне важно, у какого объекта вызывать метод Insert. Если у DeskTop, то объект вставится в список DeskTop, если у круга, то – в список круга. Метод устанавливает указатель P^.Owner на себя (p^.Owner:=@Self), так как именно он будет являться владельцем вставляемого объекта P, затем просматривает список своих подэлементов и размещает P первым в этом списке.

Обмен сообщениями

- Чтобы инициировать какое-либо действие объект должен послать сообщение. И наоборот, чтобы отреагировать на какое-либо событие (нажатие кнопки, перемещение мышки) объект должен уметь получать и обрабатывать сообщения. Для обмена сообщениями между объектами служат четыре метода:
- GetEvent - получить внешнее событие. Этот метод опрашивает устройства ввода, и если есть внешнее событие, то настраивает запись E:tEvent. Если событий нет, то полю e.What присваивается значение константы cmNothing, то есть ничего не произошло.
- ClearEvent - очистить событие. Любая обработка события должна завершаться вызовом этого метода, он очищает значение поля e.What. Это позволяет другим объектам не реагировать на уже обработанное событие и завершить обработку события досрочно.
- Message(Addr,Command,Code,InfoPtr); послать сообщение. Addr-кому, Command-тип сообщения (клавиатура, мышь, внутреннее событие), Code-код внутреннего события, InfoPtr-адрес отправителя. Если поле Addr=nil, то событие будет послано всем объектам.
- HandleEvent - обработать событие. Каждый тип объектов по-разному реагирует на то или иное событие. Поэтому у каждого типа объектов свой обработчик событий HandleEvent. Вначале он классифицирует тип события (мышь, клавиатура, внутреннее), затем кому оно принадлежит (если мышь, то по координатам ее курсора, если внутреннее, то по полю Addr). И если событие принадлежит данному объекту, то он, так или иначе, реагирует на него.

Рассмотрим обмен сообщениями между кругами и свободными точками.

Каждый раз при своем перемещении свободные точки посылают всем объектам событие cmCollision (столкновение).

```
message(nil,cmBroadCast,cmCollision,@Self);
```

Реагируют на него только круги, так как в их методе HandleEvent это предусмотрено. Получив такое сообщение, каждый круг проверяет расстояние от себя до отправителя сообщения:

```
Dist(x,y,e.InfoPtr^.x+pFreePoint(e.InfoPtr)^.dx, e.InfoPtr^.y +
pFreePoint(e.InfoPtr)^.dy)
```

Если это расстояние оказывается меньше критического (то есть столкнулись), то круг посылает внутреннее сообщение данной точке с кодом cmRed. Точка, получив его, переходит в особое "красное" состояние. Круг переводит в него и себя, а затем вызывает перерисовку своего владельца (Owner^.Draw;).

```
procedure Krug.HandleEvent(Var e:tEvent);
Begin Inherited HandleEvent(e);{Передать событие подэлементам}
  case e.what of          {Если оно еще не обработано, то}
   cmMouse: {Если пришло событие от мышки}
            if Dist(x,y,e.mx,e.my)<Rad {Если Мышь "внутри" меня, то}
            then begin {послать событие своему владельцу, что нажат круг}
         Message(Owner,cmBroadCast,cmPressKrug,@Self); clearEvent(e);{Очистить событие}
              end;
  cmBroadCast:begin {Если пришло внутреннее событие}
     if e.Code=cmCollision {Проверка на столкновение со свободными точками }
     then if Dist(x,y,… )<(Rad+5) then begin
                  Owner^.Hide;{Всем скрыться}
             RedGrenState:=Red;{Взвести флаг у себя}
             Message(e.InfoPtr,cmBroadCast,cmRed,@Self);{послать сообщение о столкновении свободной
    точке}
             Owner^.Draw;  Owner^.Hide;
                  end;
     if e.Addr=@Self{Если событие для меня}
     then case e.Code of
            cmRed: RedGrenState:=12;
            CmGreen:RedGrenState:=10;
            cmPressPoint: begin{если событие "нажата точка"}
                           AddrEv:=e.InfoPtr;HideMouse;
                           for i:=1 to 5 do begin
                              Hide;
                              if odd(i)  then  Message(AddrEv,cmBroadCast,cmRed,@Self)
                              Else  Message(AddrEv,cmBroadCast,cmGreen,@Self);
                              Draw;
                            end; ShowMouse;
                         end;
            end;{case}
     end; {cmBroadCast}
   end; {case e.what }
 end;
```

```
procedure FreePoint.HandleEvent(Var e:tEvent);
begin
   case e.what of
     cmBroadCast: case e.Code of
                   cmRed:begin
                          if e.Addr=@Self
                          then begin
                                 dx:=-dx;dy:=-dy; x:=x+5*dx;y:=y+5*dy
                               end
                         end;
                  end;
   end;
end;
```

Отдельно стоит поговорить о типе переменной E:

```
tEvent=record
       case what:word of
         cmMouse:  (  {Событие от мышки}
                    mx,my:word; Mask:byte);
         cmKeyBoard: ({Событие от клавиатуры}
                    Key,Scan:char;);
         cmBroadCast:(Addr,InfoPtr:pFigure;{Общее событие}
                      Code:word)
       end;
```

Это запись с вариантами. Название, количество и тип ее полей зависит от значения поля What (что).

Если What=cmMouse, то переменная E имеет следующие поля:

mx, my - координаты курсора мыши; mask - маска нажатых кнопок мыши.

Если What=cmKeyBoard, то переменная E имеет поля:

Key - ASCII-код клавиши, Scan- скан-код клавиши.

Если What=cmBroadCast, то переменная E имеет поля:

Addr- кому послано сообщение; InfoPtr - кто послал сообщение;

Code - какой код сообщения.

```
message(nil,cmBroadCast,cmCollision,@Self);
```

Каждый раз при своем перемещении свободные точки

```
procedure Figure.Message(Addr:pFigure;Command,Code:word
var e:tEvent;              {кому,         что,    код соб
begin     {Посылает сообщение. Если Addr=nil, то собы
  e.what:=Command;  e.infoPtr:=InfoPtr;  e.Code:=Code
  if Addr<> nil {Если сообщение не всем: то послать и
  then Addr^.HandleEvent(e)
  else PutEvent(e) {Иначе послать всем.}
end;
```

Метод DeskTop.GetEvent определят, есть ли отложенное событие, если есть, то рассылает его всем подэдементам через механизм HandleEvent. Если события нет, то анализируется нажатие клавиш и события от мышки

```
procedure Figure.PutEvent(var e:tEvent);
Begin if owner<>nil  then Owner^.PutEvent(e)
```

```
procedure DeskTop.PutEvent(var e:tEvent);
Begin   SaveEvent:=e  end;
```

```
procedure DeskTop.GetEvent(var e:tEvent);
begin
 if SaveEvent.what<>cmNothing{Если было отложенное событие, то}
 then begin                  {послать его подэлементам}
       e:=SaveEvent; SaveEvent.what:=cmNothing
      end
 else if keypressed {Если была нажата клавиша}
      then begin
        e.what:=cmKeyboard; e.Key:=readkey; if e.Key=#0 thene.Scan:=readkey
      end
      else begin {если была нажата мышь}
           ReadMouse;
           if Mask<>0 then begin
              e.what:=cmMouse; e.mx:=mx; e.MY:=my; e.mask:=Mask;  end
           else e.what:=cmNothing
      end
end;
```

ресылает DeskTop, пока сверху.

ent пересылать н сохраняет его ю **SaveEvent,** **влечет метод** **t.**

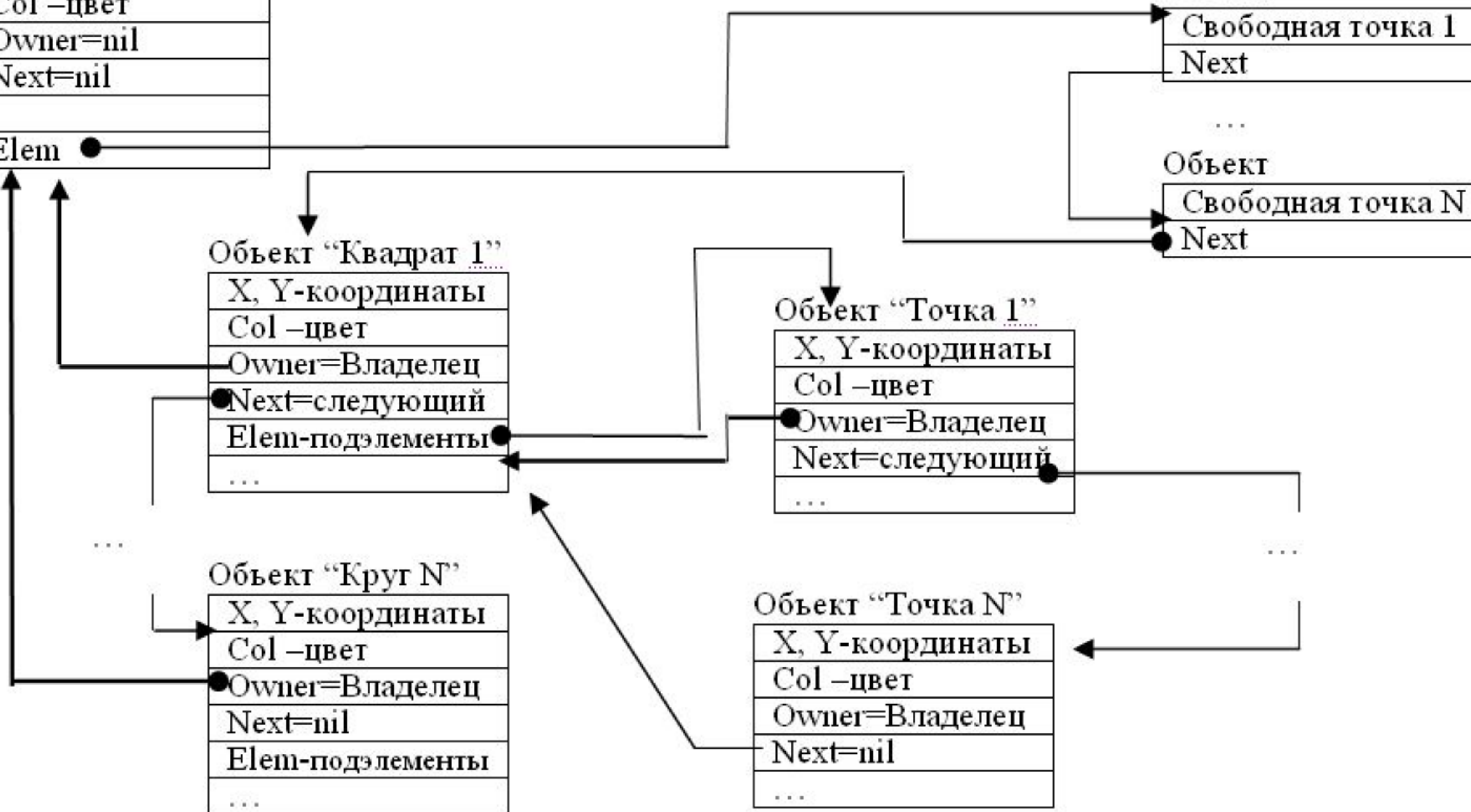
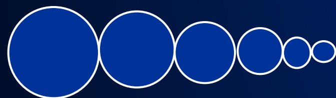
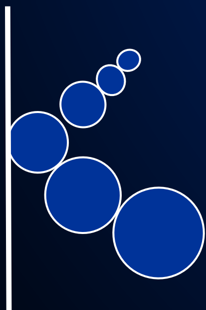


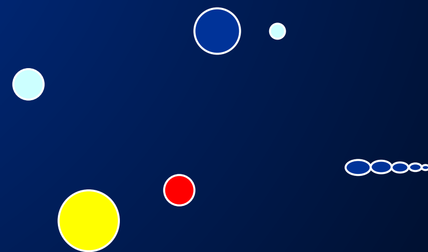
СХЕМА СВЯЗЕЙ МЕЖДУ ОБЪЕКТАМИ
(Desktop, Krug, Kvadr, Point)
Обьект DeskTop
X, Y-координаты
Col –цвет
Owner=nil
Next=nil
Elem
Обьект
Свободная точка 1
Next
…
Обьект
Свободная точка N
Next
Обьект "Квадрат 1"
X, Y-координаты
Col –цвет
Owner=Владелец
Next=следующий
Elem-подэлементы
…
Обьект "Точка 1"
X, Y-координаты
Col –цвет
Owner=Владелец
Next=следующий
…
…
…
Обьект "Круг N"
X, Y-координаты
Col –цвет
Owner=Владелец
Next=nil
Elem-подэлементы
…
Обьект "Точка N"
X, Y-координаты
Col –цвет
Owner=Владелец
Next=nil
…

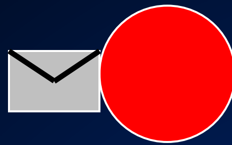
Задание

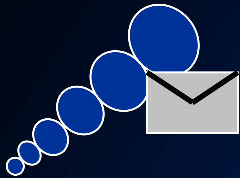
1. Исправить программу Figure2 так, чтобы круги снова начали реагировать на столкновение со свободными точками, как это сделано в Figure. Обратите внимание на то, что это можно сделать двумя способами, один из которых неправильный, хотя и дает результат.
2. *На основе программы Figure создать собственную программу, в которой будет демонстрироваться модель Солнечной системы. Вокруг Солнца вращаются 9 планет, каждая может иметь несколько спутников. Если «ткнуть» мышкой в планету, то она должна написать свое название. Вместо свободных точек по экрану должны летать кометы, состоящие из уменьшающихся кругов. Каждый круг «отражается» от стенки или планеты самостоятельно.*











При каждом перемещении, комета посылает планетам событие cmCollision (столкновение) через механизм message.

DeskTop.SaveEvent

Это приводит к тому, что событие попадает в переменную ...aveEvent, а затем метод ...ссылает событие всем ...

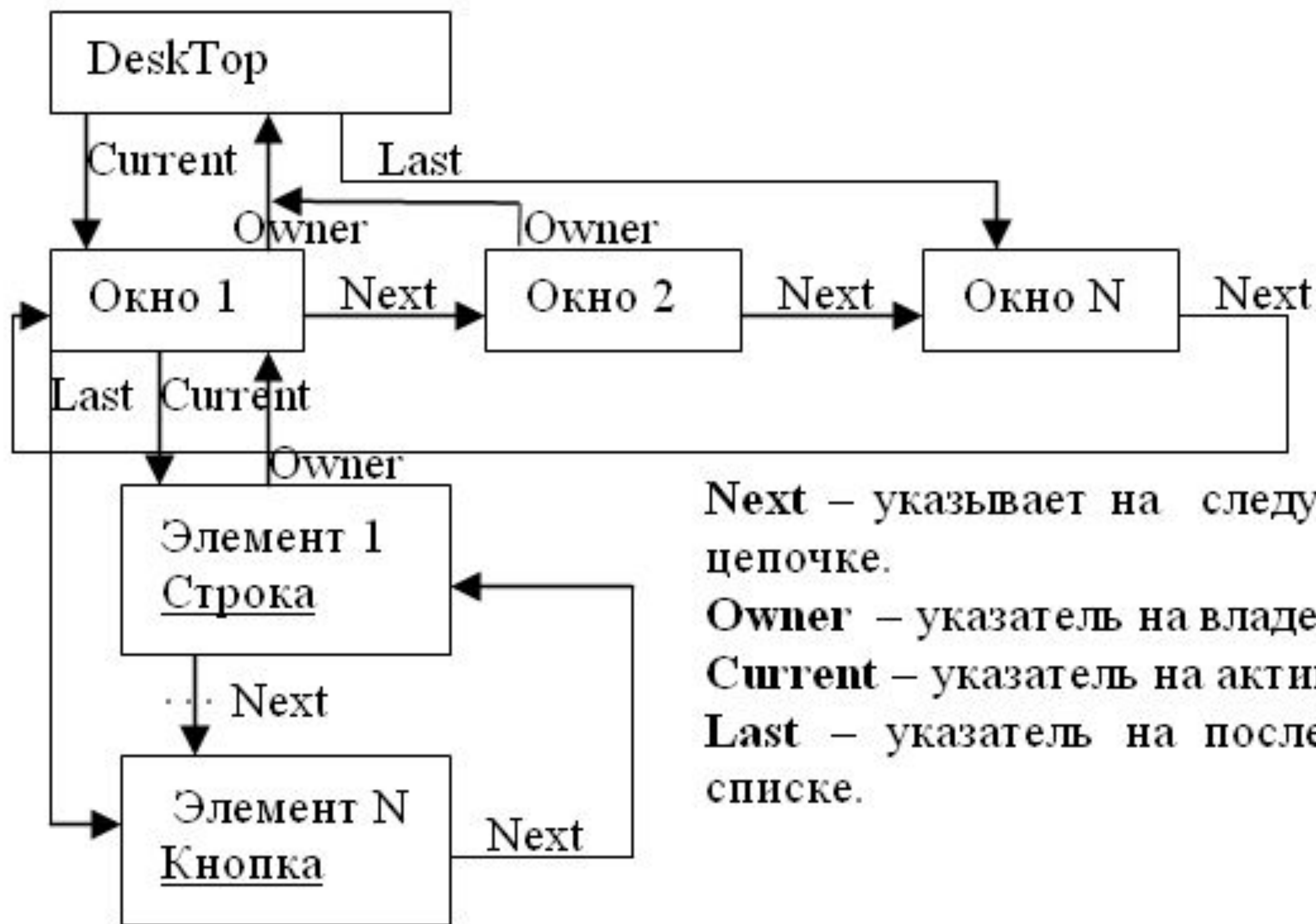
message(nil,cmBroadCast,cmCollision,@Self);

```
procedure Krug.HandleEvent(var e:tEvent);
   …
begin
  inherited HandleEvent(e);{Передать событие подэлементам}
  case e.what of          {Если оно еще не обработан...
   cmMouse: …
   cmBroadCast: begin
       if e.Code=cmCollision {Проверка на столкнове...
       then begin            {свободными точками}
             if Dist(x,y,e. InfoPtr^.x +
             pFreePoint(e.InfoPtr)^.dx, e.InfoPtr^. ...
             pFreePoint(e.InfoPtr)^.dy)<(Rad+5)
             then begin
                   Owner^.Hide;{Всем скрыться}
                   RedGrenState:=Red;{Взвести флаг у себя}
                   Message(e.InfoPtr,cmBroadCast,cmRed,@Self);
              {послать сообщение о столкновении свободной точке}
                   Owner^.Draw; sound(800); delay(100); nosound;
                   Owner^.Hide;
                  end;
            end;
```

После того, как круг обнаружил столкновение с кометой, он сообщает ей об этом, посылая личное сообщение (адрес кометы круг берет из подписи предыдущего послания).

Общая схема построения интерфейсной оболочки

- Для закрепления полученной информации вам предлагается написать интерфейсную оболочку, в которой следует учесть следующие рекомендации. В дальнейшем ее можно будет использовать при создании интерфейса в любых ваших программах. Основное требование, предъявляемое к ней, - независимость и универсальность. То есть оболочка должна предоставлять пользователю средства создания окон диалога (любой набор элементов), меню и т.д.
- В интерфейсной оболочке можно выделить следующие элементы:
- десктоп (DeskTop) - рабочее поле экрана, в нем размещаются все программные элементы: меню, окна, строка подсказки (Status Line);
- меню (горизонтальное и вертикальное);
- окна и окна диалога;
- элементы окна: кнопки, списки (селективный и триггерный);
- строка ввода, метка и т.д.
- Все элементы размещаются в кольцевом списке. Поле Next указывает на следующий элемент в цепочке. Каждый элемент имеет ссылку на владельца - Owner, она указывает либо на окно, либо на DeskTop. Поле Last указывает на последний элемент в списке подэлементов. У владельца группы элементов есть указатель Current, он показывает, какой элемент в группе является активным.



Next – указывает на следующий элемент в цепочке.

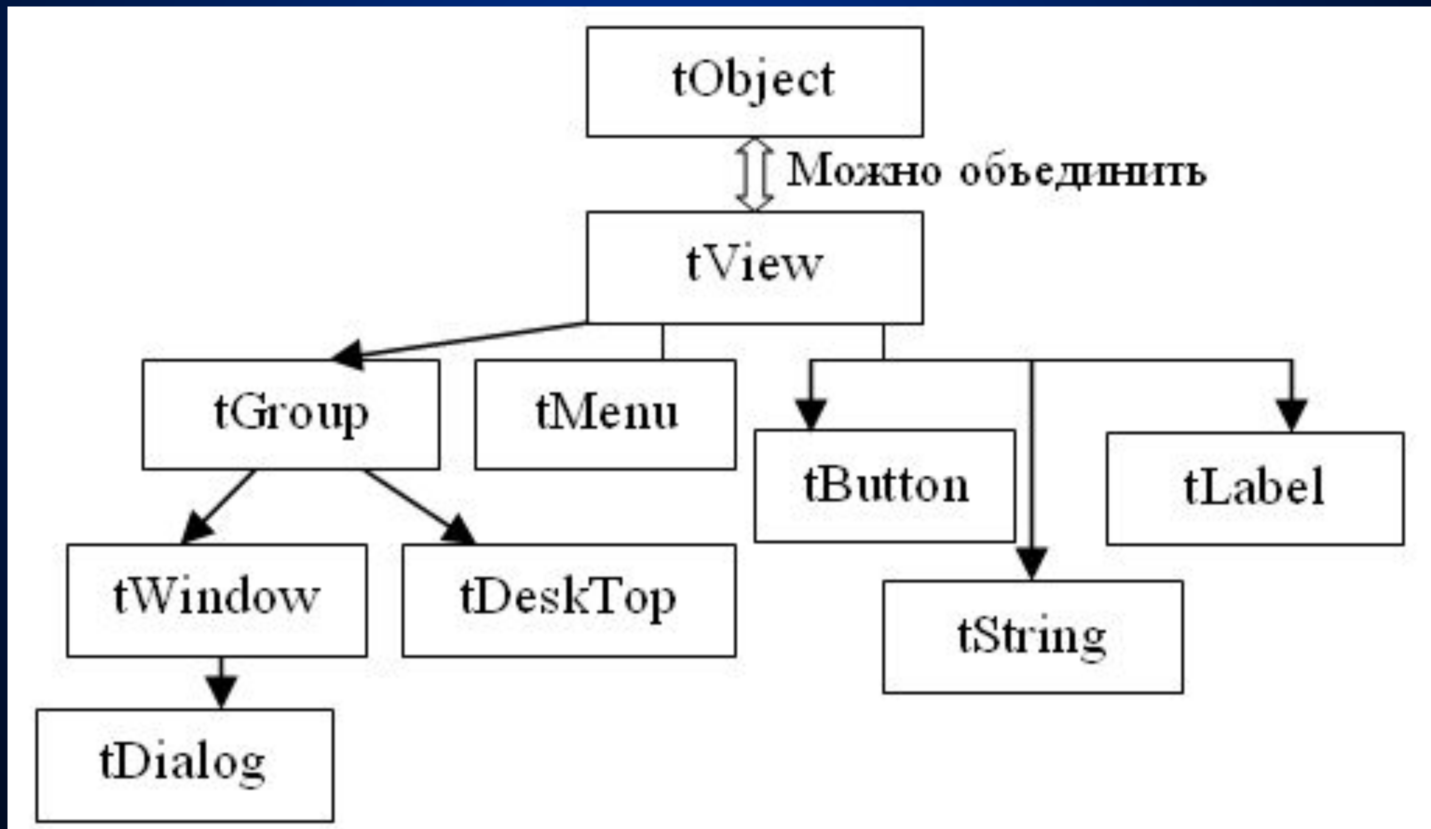
Owner – указатель на владельца.

Current – указатель на активный элемент.

Last – указатель на последний элемент в списке.

Иерархия классов

При создании программы в ООП решающую роль играет разбиение объектов на классы и установка между классами иерархической взаимосвязи. При плохом или неправильном разбиении программа становится плохо читаемой, неудобной для модификаций.



Объект tView

- Хранит все свойства видимых объектов. Он имеет следующие поля и методы:
- Owner:pGroup- указатель на владельца данного объекта, nil - у Desktop;
- Next:pView- указатель на следующий элемент в кольцевом списке подэлементов;
- X, У - координаты в рамках владельца;
- SizeX, SizeY - размеры объекта;
- State:word- статус объекта, его характеристики в данный момент (видимый, активный и т. д.);
- Options:Word- задает возможные операции над объектом (перемещаемый, разрешено менять размеры и т.п.);
- EventMask:word- маска для разрешенных в данный момент событий;
- ClearEvent(Var E:tEvent)- очистить событие;
- DataSize- возвращает размер параметров, которые необходимо передать объекту (получить от объекта);
- Execute- отложенный метод для работы с окном диалога;
- Focus:boolean - устанавливается для сфокусированных объектов;
- GetData- возвращает текущие параметры объекта, используется при окончании работы окна диалога для получения результатов его работы;
- GetEvent(Var E:tEvent)- получить текущее событие;
- HandleEvent(Var E:tEvent)- обработать событие;
- Hide- скрыть видимый объект;
- MakeLocal(Var x,y:integer)-преобразовать координаты экрана в координаты объекта;
- MakeGlobal(Var x,y:integer)-преобразовать координаты объекта в координаты экрана;
- Prev:pView- функция, возвращает указатель на предыдущий элемент;
- SetData- передает начальные параметры объекту;
- Show- нарисовать видимый объект;
- MouseContaine- определяет, находиться ли курсор мыши в координатах объекта.

Объект tGroup

- Определяет возможности работы с группой элементов, когда один объект может иметь несколько подэлементов. К данному классу объектов можно отнести окно и DeskTop. Они имеет следующие поля и методы:
- **Last:pView-** указывает на последний элемент в списке;
- **Current:pView-** указывает на текущий (активный в данный момент) элемент;
- **First:pView-** возвращает указатель на первый подэлемент списка;
- **Phase:byte-** фаза обработки события (PreProcess, Focused, PostProcess);
- **Execute-** организует работу окна диалога;
- **ForEach-** выполняет некоторую процедуру для каждого подэлемента списка;
- **Insert(p:pView)**-добавляет подэлелемент в список, делает его активным;
- **SelectNext-** делает активным следующий видимый элемент.

Создание объектов

- **Создание объектов производится при помощи функции New и размещения созданных объектов в списке при помощи метода Insert.**

```
Var p:pWindow;
begin
    p:=new(pWindow, Init(10,10,20,8,'Мое окно'));
      {создать окно}
    p^.Insert(new(pButton,Init(5,5,'OK')));
      {создать элементы окна}
    p^.Insert(new(pButton,Init(15,5,'Cancel')));
    p^.Insert(new(pString,Init(2,2,'')));
    DeskTop^.Insert(p);
      {вставить окно в список DeskTop'a}
```

- **Обратите внимание на то, что метод Insert должен принадлежать тому объекту, в список которого мы хотим вставить наш подэлемент. DeskTop - глобальная переменная, типа pMyDeskTop, созданная и описанная в программе.**

Рисование объектов

Каждый объект, являющийся потомком tView, и будучи видимым, должен переопределять метод Show. Именно этот метод задает, как будет выглядеть на экране Ваша кнопка, строка ввода, окно. Метод Show для объекта tGroup имеет несколько иные задачи - передать сообщение "нарисуйтесь" всем подэлементам.

```
procedure tButton.Show; virtual;
 begin   SetFillStile();Bar();OutTextXY();... end;

procedure tString.Show;virtual;
 begin   SetFillStile();Bar();OutTextXY();... end;

procedure tGroup.Show;virtual;
Procedure RecDraw(p:pView );
Begin
   if p <> nil
   then begin
    recDraw(p^.next);
    p^.Show;
        end;
End;
begin
    RecDraw(Elem);  {рисовать элементы в обратном порядке}
end;
```

```
procedure tWindow.Show;virtual;
begin
    SetFillStile();Bar();OutTextXY();...{нарисовать окно}
    Inherited show {нарисовать свои подэлементы за счет объекта tGroup}
End;
```

1. Рисует окно1
2. вызывает предка tWindow.Show
1. Рисует окно2
2. вызывает предка…tButton.ShowtString.Show…
end;

- Когда DeskTop получает сообщение "нарисуй себя", он сначала рисует рабочую область (фон), а затем вызывает метод своего предка **Inherited** tGroup.Show. Этот метод последовательно вызывает метод Show для всех подэлементов DeskTop'a (окон). Так как метод Show виртуальный, то для каждого подэлемента вызывается именно его метод Show. Окно сначала рисуют себя, своим цветом и в своих координатах, а затем вызывает метод предка (tGroup), который, в свою очередь, рисует подэлементы данного окна.

Передача сообщений объектам

Передача внутренних сообщений между объектами опирается на три метода:

GetEvent- получить событие;

PutEvent- поместить событие в очередь;

HandleEvent- обработать событие.

Первые два метода являются методами объекта tView, в их задачу входит добраться с нижних элементов до самого верха, до DeskTop, который помещает событие в очередь или извлекает его оттуда. Само событие хранится в записи с вариантами, у которой для каждого типа события имеются свои поля:

```
Type
tEvent=record
        case what:word of
         cmMouse: (  {Событие от мышки}
              mx,my:word; {координаты}
              Mask:byte{статус кнопок} );
         cmKeyBoard: ({Событие от клавиатуры}
         Key,Scan:char;);
         cmBroadCast:( {Общее событие}
              Addr,InfoPtr:pFigure;{кому, от кого}
              Code:word) {код события - константа}
       end;
procedure tView.PutEvent(Var e:tEvent);virtual;
begin
   if Owner<> nil
   then Owner^.PutEvent(e){если не DeskTop, то двигаемся вверх}
end;
procedure tDeskTop.PutEvent(Var e:tEvent);virtual;
begin
   SaveEvent:=e; {сохранить событие в глобальной  переменной}
   ClearEvent(e) {очистить событие}
end;
```

```
procedure tDeskTop.GetEvent(Var e:tEvent);virtual;
begin
 if SaveEvent.What<>cmNothing {если событие было сохранено}
 then begin {извлечь событие из глобальной  переменной}
       e:=SaveEvent;SaveEvent.What:=cmNothing
     end
 else begin
   ReadMouse;
   if bmask<>0 {если была нажата кнопка мыши, то}
   then begin
          e.What:=cmMouse;   e.mx:=MX;e.my:=MY;e.bMask:=bMask
        end
   else if keypressed
        then begin {если была нажата кнопка, то}
               e.What:=cmKeyBoard; e.Key:=Readkey;
               if e.Key=#0 then e.Scan:=Readkey
               else e.Scan:=#0
            end
        else e.What:=cmNothing {иначе - ничего}
    end
end;
```

Метод GetEvent объекта tDeskTop сначала анализирует отложенное событие, которое сохраняется для простоты не в очереди, а в глобальной переменной SaveEvent. Если отложенного события не было, то анализируется мышь, если кнопка мыши не была нажата, то анализируется клавиатура. Если ни одно из перечисленных событий не произошло, то переменной e присваивается код cmNothing, то есть "ничего".

Метод ClearEvent отвечает за "очистку" события:

```
procedure tView.ClearEvent(Var e:tEvent);virtual;
 begin
  e.What:=cmNothing
 end;
```

Его вызывает подэлемент, когда выполнит обработку события, это необходимо для того, чтобы остальные элементы не реагировали на уже обработанное событие.

Самым важным элементом системы обработки событий является метод HandleEvent, именно он отвечает за анализ всех событий и реакцию на них объекта. Каждый объект должен переопределять этот метод. Метод tGroup просто рассылает сообщение всем своим подэлементам.

```
procedure tGroup.HandleEvent(Var e:tEvent);virtual;
Var p:pView;
 begin
    p:=First;  {встать на первый элемент в списке}
    while (p<> nil) and (e.What<>cmNothing) do
      begin
       p^.HandleEvent(e)
       p:=p^.Next; {перейти к следующему элементу}
      end;
   {закончить, когда дойдем до конца или событие кто-то обработал}
     end
 end;
```

Каждый подэлемент должен, так или иначе, реагировать на события. Если объект обработал событие, то он вызывает метод ClearEvent.

```
procedure tButton.HandleEvent(Var e:tEvent);virtual;
begin
  case e.What of
   cmMouse:begin {если произошло событие от мышки, то}
        if MouseContain {мышь находится на мне}
        then begin
              нажаться;
              ClearEvent(e); {очистить событие}
              e.What:=cmBroadCast; {код общего события}
              e.Code:=Code {код нажатой кнопки, задается при инициализации}
              e.InfoPtr:=@Self; {от кого сообщение}
              e.Addr:=Owner;  {кому сообщение? - владельцу}
              PutEvent(e); {послать сообщение}
             end;
        end;
   cmKeyBoard: begin
               if e.Key=моей горячей клавише
               then послать сообщение владельцу
              end;
    end;{case}
 end;
```

Из примера видно, что кнопка анализирует тип события (мышь, клавиатура и т. п.), а затем определяет принадлежность этого события. Если событие принадлежит ей, то оно обрабатывается, при этом владельцу посылается уникальный код - константу, которая указывается пользователем при создании кнопки. Обработчик окна HandleEvent, которому принадлежит кнопка, анализирует поступающие события, и, если получен код, равный коду его кнопки, то окно реагирует на это событие, вызывая соответствующую процедуру. Существуют несколько стандартных кодов: CmYes, cmNo, cmOK, cmCancel, которые можно определить в модуле.

```
procedure tWindow.HandleEvent(Var e:tEvent);virtual;
begin
 case e.What of
  cmMouse:begin
    if MouseContain {мышь находится на мне}
    then begin
          if (Status and fActive)=0 {я не активно}
          then begin
                {послать сообщение активному элементу владельца о смене
    его статуса (сбросить активность у элемента  Owner^.Current^)}
            Message(Owner, cmBroadcast, cmNotActive, @Self);
            Status:=Status or fActive;{взвести активность у себя}
            {сообщить владельцу о смене активного элемента}
            Message(Owner, cmBroadcast, cmActive, @Self);
            {перерисоваться}
            Message(Owner,cmBroadcast, cmShow, @Self);
            ClearEvent(e); {очистить событие}
           end
         else begin {если окно активно}
         Inherited HandleEvent(e);{передать событие подэлементам}
         if e.What<>cmNothing  {если событие еще не обработано,}
         then begin
                      анализ возможности  перемещения, закрытия окна;
                  ClearEvent(e); {очистить событие}
                    end
          end;
       end;
    end;
```

```
cmKeyBoard: begin {клавиатура}
      case e.Key of
       tab: selectNext;{выбрать активным след. элем.}
       Shift+Tab:SelectPrev;
       else Inherited HandleEvent(e) {послать событие подэлементам}
      end
   end;
cmBroadCast:begin
      if e.Addr=@Self {если событие прислано мне}
      then case e.Code of{здесь анализируются коды,сообщаемые подэлементами}
            cmYes, cmOk:begin    {закрыть окно, сообщить об успехе}
                       e.What:=BroadCast;
                       e.Code:=cmClose;{закрыть}
                       e.Addr:=Owner; {кому}
                       e.InfoPtr:=@Self {кого закрыть}
                       PutEvent(e);  {послать сообщение}
                       ClearEvent(e)
                     End;
cmNotActive:Status:=Status and(not fActive){сбросить активность у себя}
  end {case}
 end
end
end;
```

Обработчик событий DeskTop.HandleEvent должен следить за событиями cmClose, cmActive и, так или иначе, реагировать на них. Например, в первом случае, он должен закрыть окно, которое прислало это сообщение. Во втором- изменить указатель Current на новый активный элемент или (что удобнее) удалить окно из списка и вставить его обратно. В этом случае активный объект будет всегда первым в списке, а вторым – тот объект, которым был активным до этого. Это позволит перерисовывать не все объекты, а только активный и бывший активный, а так же правильно передавать сообщения. Обработчик tWindow.HandleEvent должен реагировать на событие cmNotActive, сбрасывая соответствующие флаги.

За организацию передачи и обработки событий отвечает метод Run, именно он опрашивает источники событий и рассылает сообщения.

```
procedure tDeskTop.Run;
Var e:tEvent;
begin
  настройка параметров;
  Repeat
   GetEvent(e);
       HandleEvent(e);
  Until Quit
end;
```

Все описанные ранее объекты собираются в TPU модуль, а сама программа выглядит следующим образом:

1) Сначала создается новый класс tMyDeskTop, который является потомком tDeskTop. Переопределяется его конструктор и деструктор:

```
constructor tMyDeskTop.Init;
begin
  Inherited init; {вызвать конструктор предка}
  создать и разместить в списке все элементы программы: окна, меню и т. д.
end;
destructor tMyDeskTop.Done;
begin
 Inherited Done;{вызвать деструктор предка:ликвидировать подэлементы}
 освободить выделенную дополнительную память(загруженные
   картинки)
end;
procedure tMyDeskTop.HandleEvent;
begin
  анализировать сообщения, присылаемые созданными окнами
   (закрыть окно, сообщения от меню и т.д.);
  Inherited HandleEvent; {послать сообщения подэлементам}
end;
```

2) сама программа состоит из трех действий:

```
 begin
    Desktop:=New(pMyDeskTop,Init);
    DeskTop^.Run;
    Dispose(DeskTop,Done)
 end.
```

Метод Quit должен следить за нажатием горячей кнопки выхода Alt+X и вызывать метод Valid для всех подэлементов. По умолчанию этот метод всегда согласен на выход:

```
function tView.Valid:boolean;virtual;
 begin
     Valid:=true
 end;
```

Если пользователю необходимо запретить выход до наступления некоторого события (например, до ввода имени пользователя), то он перекрывает метод Valid своей подпрограммой, которая анализирует условие завершения программы (диалога). Если хотя бы один из подэлементов вернул False, то tGroup.Valid тоже возвращает False. Например, в текстовом редакторе перед выходом необходимо убедиться в том, что пользователь сохранит измененный текст. Если он этого не сделал, то выход необходимо запретить.

Создание своего окна

Для создания "своего" окна диалога, например, загрузки файлов, необходимо действовать аналогично:

1) Создать новый класс tMyWind1, который является потомком tWindow;

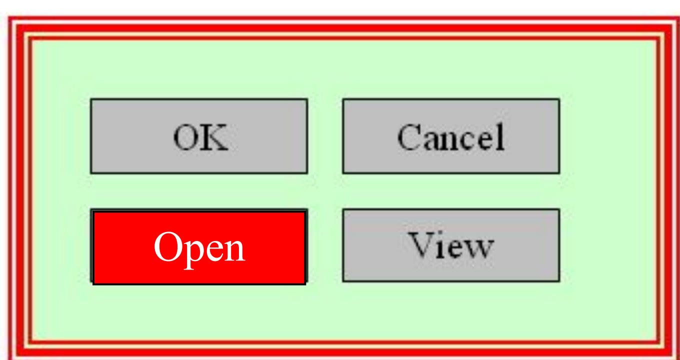
2) переопределить конструктор, который создаст все подэлементы окна (кнопки, строки ввода, списки ...), переопределить деструктор, который все это ликвидирует;

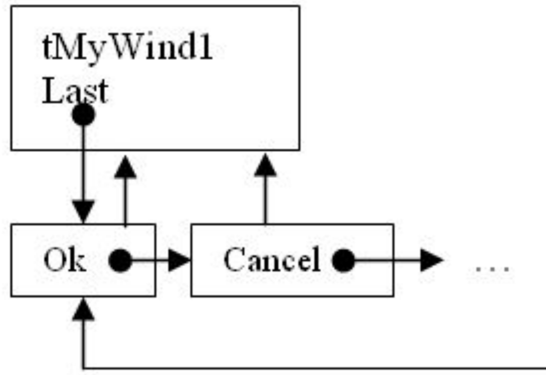
```
constructor tMyWind1.Init(…);
Var p:pButton;
begin
  Inherited init(…); {вызвать конструктор предка}
  Insert(New(pButton,Init(10,120,'OK',cmOK)));
  Insert(New(pButton,Init(50,120,'Cancel',cmCancel)));
  Insert(New(pButton,Init(90,120,'Open',cmOpen)));
  Insert(New(pButton,Init(130,120,'View',cmView)));
  …
end;
```

3) переопределить метод обработки событий

```
procedure tMyWind.HandleEvent(Var e:tEvent);virtual;
begin
  case e.What of
    cmBroadCast: begin
       if e.Addr=@Self {если это мне}
       then   case e.Code of
{анализировать коды кнопок, вызывать соответствующие подпрограммы}
                 cmOpen: begin
            OpenFile;  ClearEvent(e)
                End;
                 cmView: begin
            ViewFile; ClearEvent(e)
                End
                end
          end
     end;
  if e.What<>cmNothing {Если еще не обработали}
  then Inherited HandleEvent(e); {вызвать метод окна tWindow}
 end;
```

Когда пользователь нажимает мышкой на кнопку созданного окна, то обработчик HandleEvent посылает сообщение своему владельцу, то есть окну. Обработчик событий окна следит за кодами, которые присылают ему подэлементы и другие объекты программы. Если пришло общее событие (cmBroadcast) и его код равен cmOpen, то окно вызывает соответствующую подпрограмму.






```
procedure tButton.HandleEvent(Var e:tEvent);virtual;
begin
 case e.What of
  cmMouse:begin {если произошло событие от мышки, то}
   if MouseContain {мышь находится на мне}
   then begin
       нажаться;
       ClearEvent(e); {очистить событие}
       e.What:=cmBroadCast; {код общего события}
       e.Code:=Code {код нажатой кнопки=cmOpen}
       e.InfoPtr:=@Self; {от кого сообщение}
       e.Addr:=Owner;  {кому сообщение? - владельцу}
       PutEvent(e); {послать сообщение}
    end;
   end;
```

Пусть пользователь щелкнул мышкой по кнопке Open. Это приводит к рассылке события cmMouse

Обработчик кнопки Open, получив событие cmMouse, реагирует на него, посылая владельцу (окну) событие cmOpen

```
procedure tMyWind.HandleEvent(Var e:tEvent);virtual;
begin
 case e.What of
  cmBroadCast: begin
    if e.Addr=@Self {если это мне}
    then   case e.Code of
            cmOpen: begin
            OpenFile;  ClearEvent(e)
               End;
```

Обработчик окна, получив событие cmOpen, реагирует на него, вызывая соответствующую процедуру

Организация списков

- Список – это важнейший элемент окон диалога, позволяющий пользователю сделать выбор одного или несколько вариантов из предложенных. Различают два вида списков:
- *Селективный* – пользователю предлагается несколько вариантов, он может выбрать только один.
- *Триггерный* - пользователю предлагается несколько вариантов, каждый вариант может быть выбран отдельно.

Вопрос, заданный пользователем:
( ) вариант ответа 0;
( ) вариант ответа 1;
(•) вариант ответа 2;
( ) вариант ответа 3.

Вопрос, заданный пользователем:
□ вариант ответа 0;
■ вариант ответа 1;
□ вариант ответа 2;
■ вариант ответа 3.

Рассмотрим проблему организации списков на примере селективного списка:

```
Type    pStrElem=^ tStrElem; {хранение вариантов в динамическом списке}
   tStrElem=record
      St:string; {формулировка варианта ответа (№0..№N-1)}
      Next: pStrElem {указатель на следующий вариант}
        End;
pSelect=^tSelect;
tSelect=object(tView)
      Number:word;{номер, выбранного элемента}
      Quest:string;{вопрос пользователя}
      pElem:pStrElem;{указатель на список вариантов ответа}
    Constructor Init(ax,ay:integer; aQuest:string; aElem:pStrElem);
    …
end;
```

Схематично объект «Селективный список» можно представить так:

Чтобы создать такой список можно воспользоваться специально созданной функцией:

```
Function NewStr(aStr:string;aNext:pStrElem): pStrElem;
Var p: pStrElem;
Begin
   New(p);    p^.st:=aStr;
   p^.Next:=aNext;   NewStr:=p
End;
Тогда сам список можно создать так:

New(p, Init(ax, ay,'Вопрос…',
     NewStr('Вариант 1',
     NewStr('Вариант 2',…,
     NewStr('Вариант N', nil))));
```

Вывод списка на экран довольно тривиален, поэтому рассмотрим лишь обработку событий. Напомню, что один элемент окна является выбранным (активным или сфокусированным). Такой объект должен первым получать события от клавиатуры, и только если он не обработал событие, то его получают другие подэлементы. То есть, если список выбран, то активным вариантом ответа можно управлять при помощи стрелок.

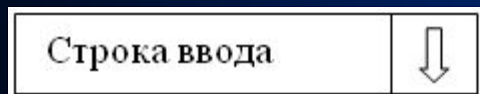
```
Procedure tSelect.HandleEvent(Var e:tEvent);
Begin
  Case e.What of
    CmMouse: begin
      If  MouseContain {мышь на мне}
      {размеры могут быть вычислены конструктором}
      Then begin
             Вычислить новый активный элемент;
             Number:=его номер;
             Перерисовать точку у активного элемента;
             ClearEvent(e)
           end
    End;
   CmKeyBoard: begin
        Case e.Scan of
         #75: if Number>0 then dec(Number);
         #77:…;
        End;
       Перерисоваться;     ClearEvent(e);
     end
  End
End;
```

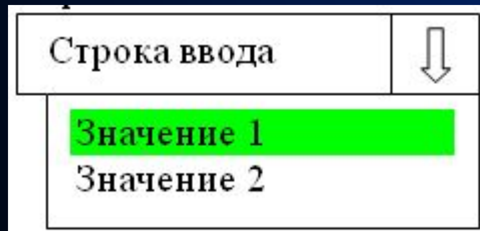
Триггерный список отличается от селективного тем, что поле Number хранит не номер выбранного элемента, а каждый бит определяет состояние каждого варианта. Если бит D0 равен 1, то вариант 0 выбран, если D0=0, то вариант 0 не выбран.

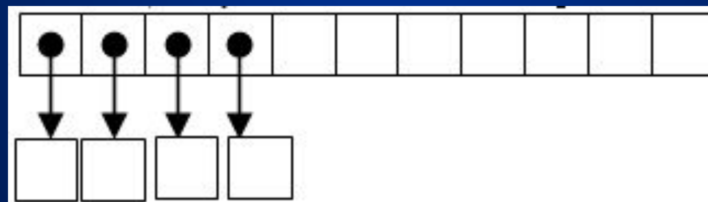
Элемент ListBox (строка с памятью)

Очень часто пользователь сталкивается со строкой ввода, рядом с которой нарисована стрелка вниз. Если нажать на эту стрелку, то появится небольшое окно либо со всевозможными вариантами значений строки, либо с вариантами, которые пользователь уже вводил. Это окошко называется объектом ListBox. Он как бы «цепляется» к строке и хранит различные варианты значения этой строки.

Работа элемента ListBox основывается на особом объекте – коллекции. Коллекция – это объект, предназначенный для хранения элементов неопределенного типа. В идеале объект вообще не должен зависеть от типа хранимых элементов. Это становится возможным, если хранить не объекты, а указатели на них. Пользователю предоставляются методы: AddElem (добавить новый элемент), DeleteElem (удалить элемент), GetElem (получить элемент) и переопределяемый метод GetText (получить текстовое представление элемента).







```
Type pCollection=^tCollection;
tCollection=object
      Collect:array[1..Max] of pointer;{коллекция (хранилище элементов)}
      Number:word;{количество элементов в коллекции}
      Procedure AddElem(Var El); virtual; {добавить элемент}
      …
end;

Procedure tCollection.AddElem(Var El);virtual;
{параметр может быть любого типа}
Begin
  If Number<Max {если в коллекцию можно еще что-то добавить}
  Then begin
          Inc(Number); Collect[Number]:=pointer(El)
        End
  Else ErrorCollection
End;

Function tCollection.GetText(i:integer):string; virtual;
{Получить элемент с номером i}
Var xx:тип элемента коллекции;
Begin   {переопределяется пользователем, заранее не известен тип XX}
  If  (i<Number)and(i>0) {если элемент в коллекции}
  Then begin
         Xx:=тип элемента(GetElem(i)^);
         GetText:=StrText(xx) {преобразовать в текстовое представление}
        End
  Else GetText:=''
End;
```

В нашем случае (для простоты) можно обойтись массивом строк. Объект ListBox должен иметь доступ к двум объектам: строке ввода и коллекции. При создании объекта ему передается готовая коллекция (либо пустая, либо с уже готовыми значениями):Коллекция значенийСтрока вводаКоличество видимых эл.

PLB:=New(pListBox,Init(New(pCollection,Init(…)),New(pString,Init(…)), N))

При инициализации объект должен рассчитать координаты своей «видимой стрелки»:

X:=S^.x+S^.Width; Y:=S^.y; {*координаты строки ввода + ее ширина*}

Самой важной частью объекта ListBox является обработчик событий:

```
Procedure tListBox.HandleEvent (Var e:tEvent);
Begin
Case e.What of
  CmKeyBoard: begin
    If e.Scan=#80 {нажата стрелка «вниз»}
    Then begin
          Стать активным (сфокусированным);
          Нарисоваться;   Вывести список:
          N:=min(количество видимых элем., Number);
          If n>Nvid then создать строку ScrollBar;
          For i:=0 to  N-1 do Begin
             St:=Copy(Collect^.GetText(Tek+i),1,ширина);
             Вывод (St)
          End;
        Repeat
    Анализ нажатия клавиш и сдвиг списка
        Until клавиша Enter;
        S^.SetData(Collect^.GetText(выбранный элемент)) {передать строке выбранный элемент}
      end
    end
  End {case}
End;
```

Организация окон диалога

- Окно диалога предназначено для организации интерфейса между программой и пользователем. Оно получает от пользователя какие-то параметры и передает их программе. Работа с окнами диалога опирается на три метода: SetData, ExecuteDialog, GetData.
- Основной проблемой при организации диалога является передача начальных значений подэлементам окна и получение результата от них (ведь, сколько всего подэлементов, какого они типа, в каком порядке они расположены заранее не известно). Для организации обмена данными программист должен создать запись, поля которой располагаются в том же порядке, что и подэлементы окна диалога. Каждое поле записи предназначено одному подэлементу, который точно знает, какого типа будет параметр и каков его размер. Например, строка будет получать начальное значение типа String, ее размер 256 байт должен точно соответствовать размеру строки при передаче параметров или при их приеме. Список получает поле типа Word – номер активного элемента, его размер 2 байта и так далее.
- Каждый объект должен иметь поле DataSize:word, которое будет хранить размер взятого или введенного с клавиатуры параметра. И метод SetData, который должен скопировать значение передаваемого параметра во внутреннее поле объекта.

```
constructor tString.Init(ax,ay,aSizeX:integer;Len:byte; s:string);
begin
  x:=ax; y:=ay; DataSize:=len; SizeX:=aSizeX;
  {где x,y-координаты,Str- значение строки, DataSize- ее размер }
  GetMem(Str,Len+1); {выделить память под значение. Можно просто}
  Str^:=s        {использовать тип string}
end;
Procedure tString.SetData(Var Rec);
type tStr=string;
begin
  Str^:=copy (tStr(Rec),1,DataSize) {скопировать строку}
end;
procedure tGroup.SetData(Var Rec);
type   Bytes=array[0..65535] of Byte;
Var     I:word; V:pView;
begin
    V:=First; I:=0;
    while v<> nil do
      begin
          V:=V^.Next;
          V^.SetData(Bytes(rec)[i]);
          Inc(i,V^.DataSize);
      end;
end;
```

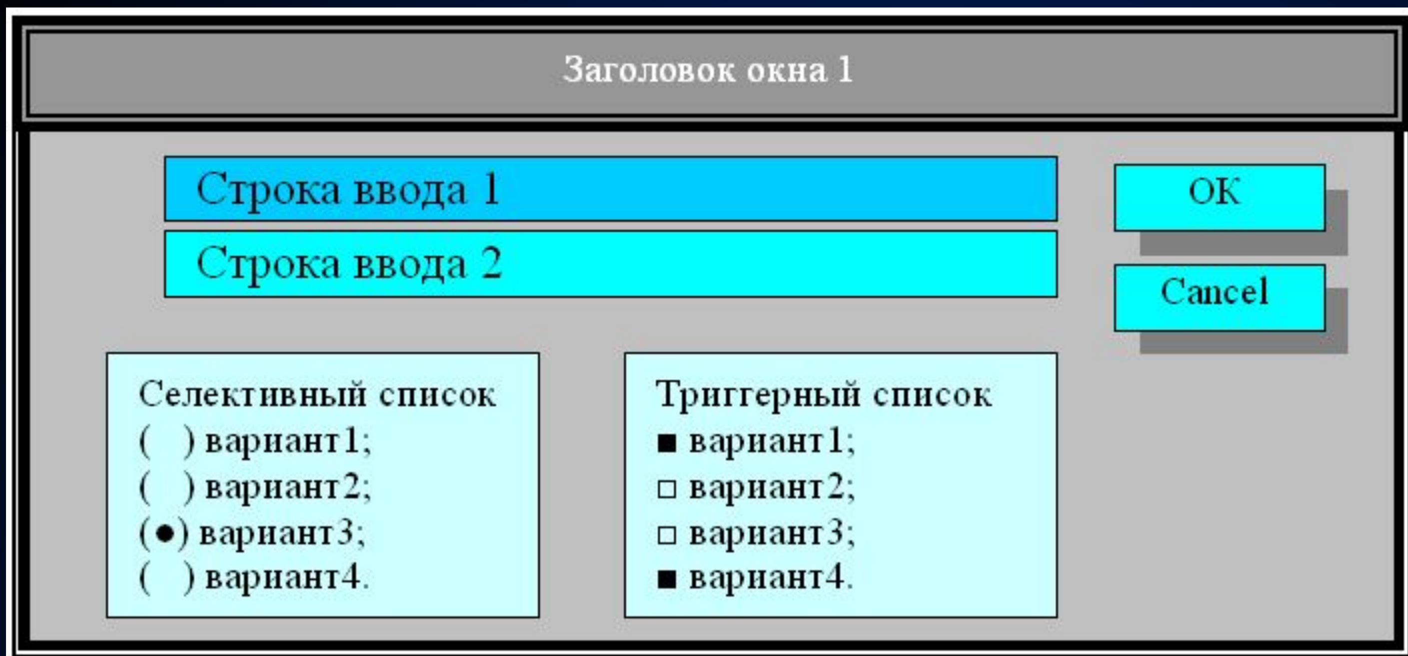
Двигаемся по списку подэлементов, для каждого из них вызываем метод SetData, а затем смещаем указатель l на размер взятого параметра. (!) Обратите внимание на необходимое преобразование типов.

Метод GetData выполняется аналогично. Следует обратить внимание, что каждый объект должен сам следить за значением DataSize.

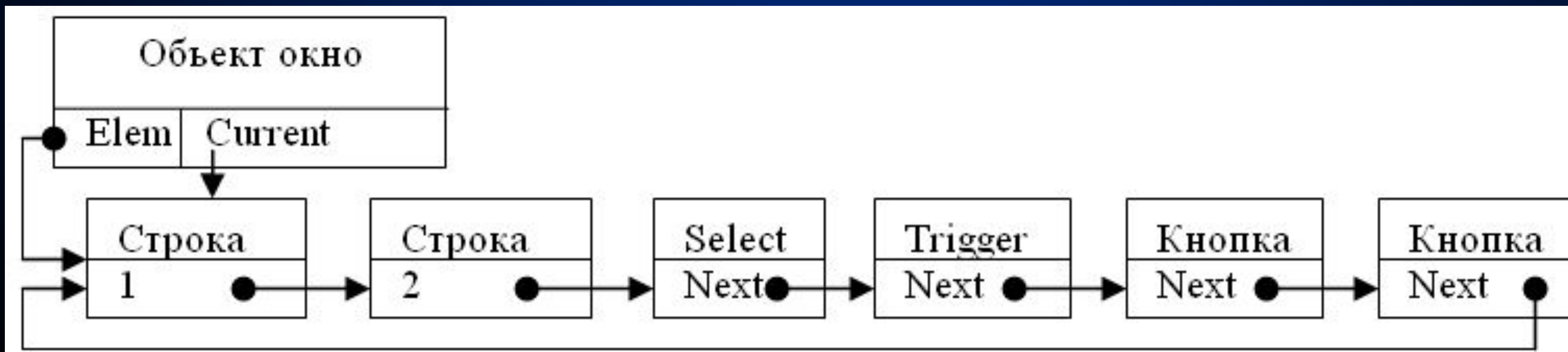
За организацию работы окна диалога отвечает метод объекта tDeskTop.ExecuteDialog.

```
function tDeskTop.ExecuteDialog(p:pWindow; Data:pointer):word;virtual;
Var res:word;
begin
   ExecuteDialog:=cmCancel;
   insert(p); {вставить окно диалога в список}
   if Data<>nil then p^.SetData(data^); {обратите внимание на ^}
   Res:=p^.Execute;
   if (Res<>cmCancel)and(Data<>nil)
   then p^.GetData(data^);
   ExecuteDialog:=res
   delete(p); {Удаление объекта из кольцевого списка}
   dispose(p,Done); {ликвидировать окно диалога}
 end;
function tGroup.Execute:word;
Var e:tEvent;
begin
  Repeat
      GetEvent(e);    HandleEvent(e);
  Until EventStatus<>0;
  Execute:=EventStatus
end;
```

Переменная EventStatus является полем объекта tGroup, она равна коду нажатой кнопки (cmOK или cmCancel).



Данное окно содержит 6 элементов: две строки ввода, два списка, две кнопки. Каждый элемент может стать активным, сейчас активной является первая строка ввода. Все события сначала передаются ей, и лишь затем (если она не обработала событие) всем остальным объектам. Структура окна как объекта выглядит так:



Создание такого окна диалога выполняется в три этапа:

Создается новый объект

```
Type
pMyDialog=^tMyDialog;
tMyDialog=object (tDialog)
     Constructor Init(ax,ay,aSizeX,aSizeY:integer;Title:string);
     Procedure HandleEvent(Var e:tEvent); virtual;
 End;
```

Переопределяем конструктор, в котором создаем все нужные подэлементы окна.

```
Constructor tMyDialog.Init
                    (ax,ay,aSizeX,aSizeY:integer;Title:string);
Begin
 Inherited Init(ax,ay,aSizeX,aSizeY,Title);
 P:=New(pString,Init(…));Insert(p);{создание и вставка первой строки}
 P:=New(pString,Init(…));Insert(p);{создание и вставка второй строки}
 P:=New(pSelect,Init(…));Insert(p);{создание и вставка селективного списка}
  …
End;
```

В процедуре, которая нуждается в диалоге с пользователем, создаем окно диалога и переменную запись, порядок полей которой полностью соответствует порядку размещения подэлементов в окне диалога.

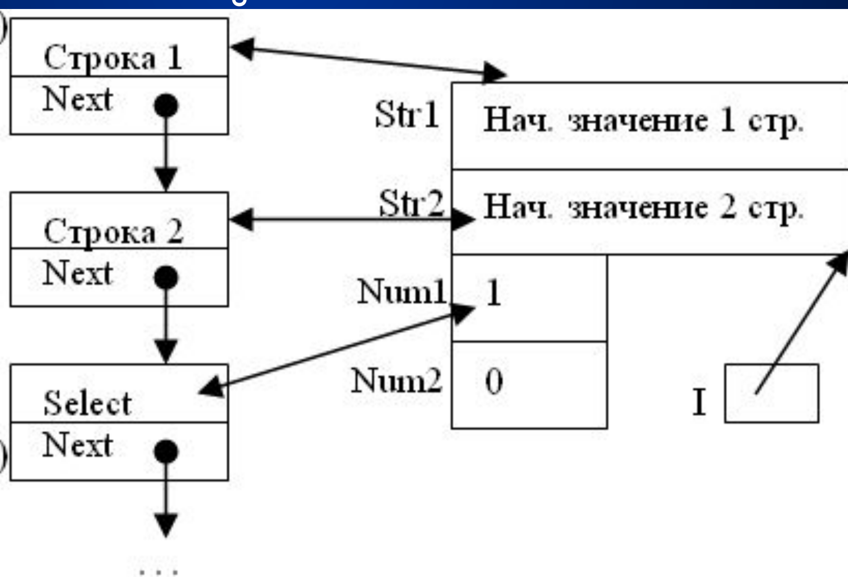
```
Procedure ExampDial;
Type tRec=record
           Str1,Str2:string; {начальное значение 1 и 2 строк}
           Num1,Num2:word {начальное состояние списков}
         End;
Const Rec:tRec=( (Srt1:'начальное значение 1 строки');
     (Srt2:'начальное значение 2 строки'); (Num1:1); (Num2:0 ));
Var Wnd:pMyDialog;    Res:word;
begin
  New(Wnd, Init(100,100,100,100,'Заголовок окна')); {Создать диалог}
  Res:=ExecuteDialog(Wnd,Rec);{запустить диалог и передать ему параметры}
  If Res=CmOK
  then Rec содержит новые значения полей, веденные пользователем
end;
```

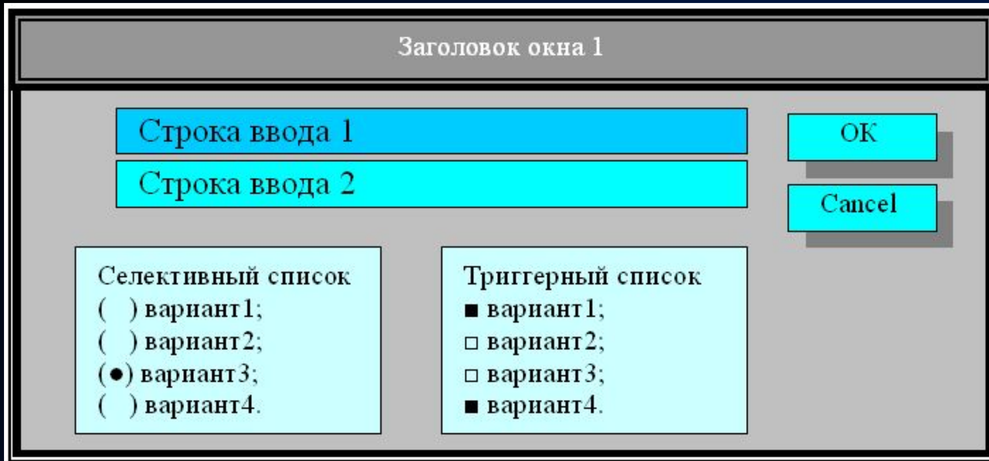
Вспомним, как работает метод ExecuteDialog.

```
function tDeskTop.ExecuteDialog(p:pWindow; Data:pointer):word; virtual;
Var res:word;
begin
  ExecuteDialog:=cmCancel;
  insert(p); {вставить окно диалога в список}
  if Data<>nil then p^.SetData(data^); {обратите внимание на ^}
  Res:=p^.Execute;
  if (Res<>cmCancel)and(Data<>nil) then p^.GetData(data^);
  ExecuteDialog:=res
  delete(p); {Удаление объекта из кольцевого списка}
  dispose(p,Done); {ликвидировать окно диалога}
end;
```

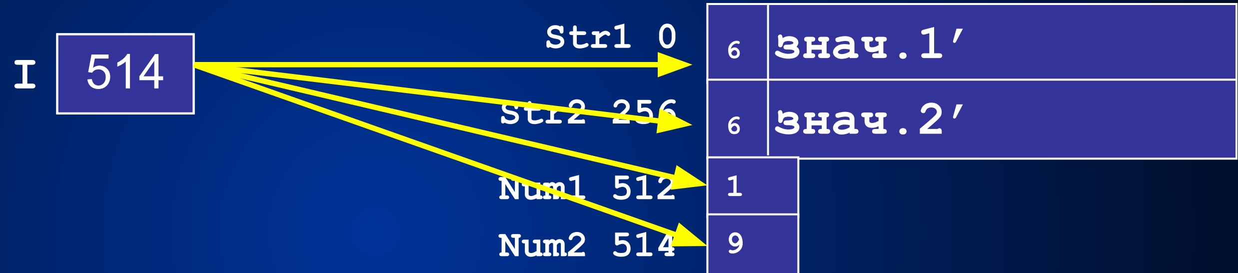
Сначала подэлементам окна диалога передаются начальные параметры. Это реализуется при помощи метода SetData. В начале вызывается метод tDialog.SetData, это идентично вызову tGroup.SetData. Он последовательно вызывает методы SetData каждого подэлемента и двигает указатель по записи параметров, то есть каждый подэлемент «откусывает» от блока параметров столько, сколько ему надо (Смотри схему). Счетчик I двигается по записи как по массиву байт. После вызова метода SetData первой строки ввода, она скопирует себе начальное значение из поля Str1 и установит значение поля DataSize в 256 (размер выделенной строки). Значение счетчика I увеличится соответственно на 256, и теперь он будет уже указывать на поле Str2. Указатель V переместится к следующему объекту в списке подэлементов окна, и будет указывать на вторую строку. Далее вызовется метод SetData второй строки, затем списка. Метод tSelect.SetData скопирует всего 2 байта (word) – номер активного элемента и т.д. Как вы видите, порядок следования элементов и

```
procedure tGroup.SetData(Var Rec);
type Bytes=array[0..65535] of Byte;
Var  I:word;     V:pView;
begin
  If Last<>nil
  Then begin
         V:=Last; I:=0;
         Repeat
          V:=V^.Next;
          V^.SetData(Bytes(rec)[i]);
          Inc(i,V^.DataSize);
         Until V=Last;
       end
end;
```

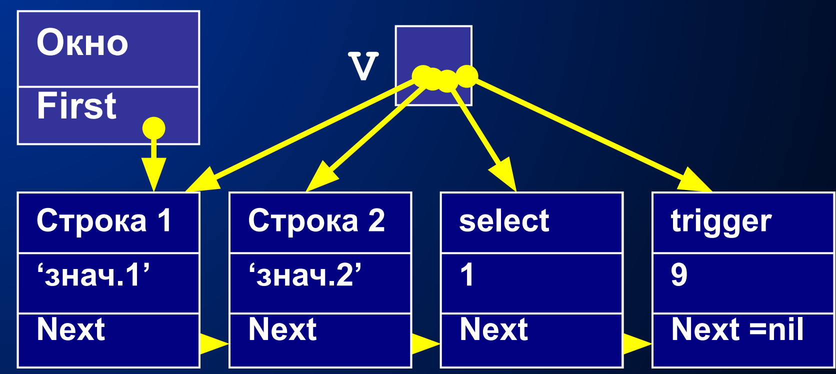





```
Type tRec=record
        Str1,Str2:string;
        Num1,Num2:word
     End;
Const Rec:tRec=(
(Srt1:'знач.1');
(Srt2:'знач.2');
(Num1:1); (Num2:9 ));
```



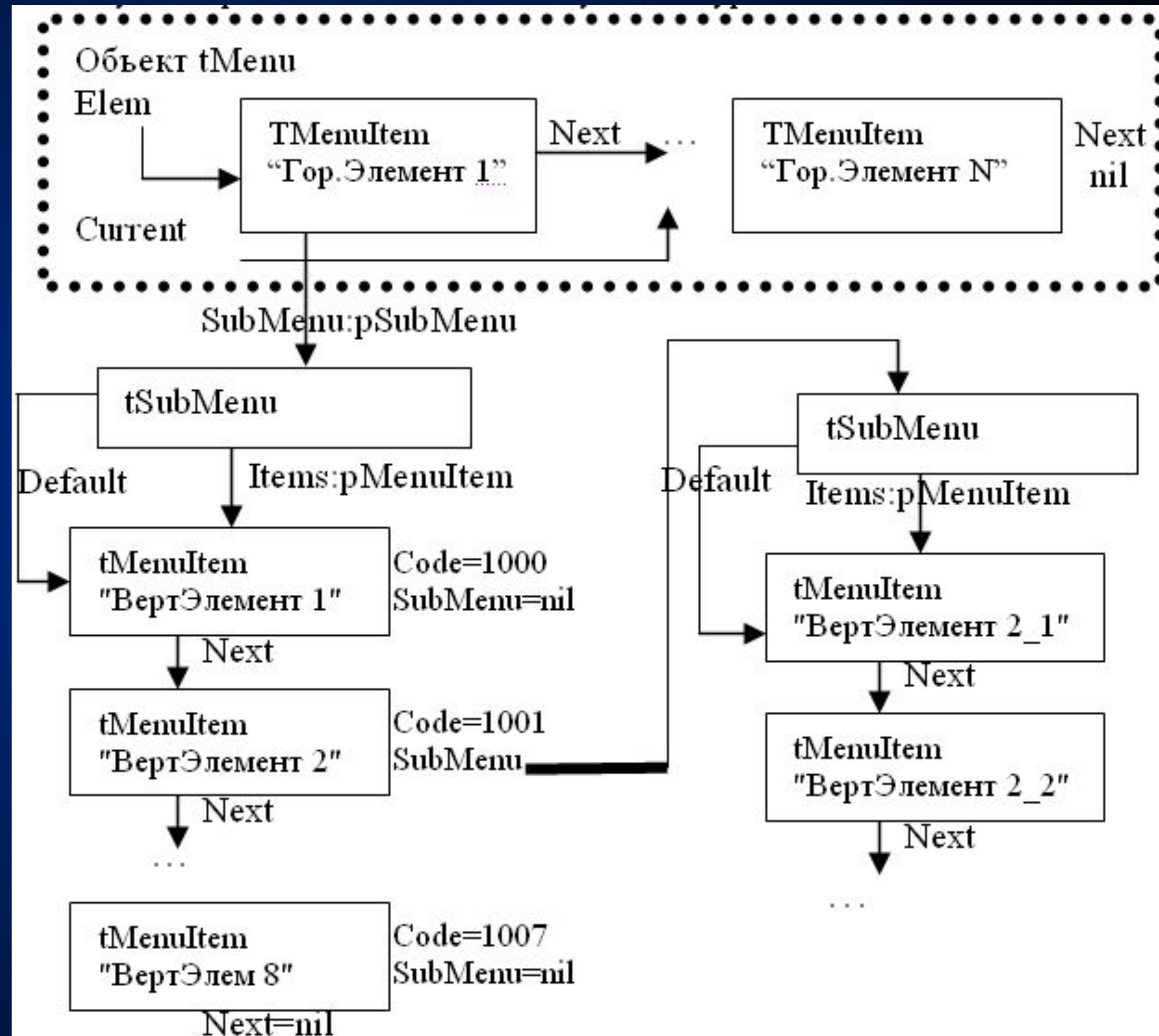

```
procedure tGroup.SetData(Var Rec);
type Bytes=array[0..65535] of Byte;
Var I:word; V:pView;
begin
 If Last<>nil
 Then begin
       V:=First; I:=0;
       Repeat
        V^.SetData(Bytes(rec)[i]);
        Inc(i,V^.DataSize);
        V:=V^.Next;
       Until V=nil;
      end
end;
```



Далее процесс аналогичен: каждый объект списка копирует из записи свой кусок и увеличивает смещение i на его размер

Организация меню

Меню - это список услуг, предоставляемых пользователю. Меню делится на две части: горизонтальное меню и вертикальное меню. Каждый пункт горизонтального меню имеет ссылку на соответствующую структуру вертикального меню. А каждый элемент вертикального меню имеет либо уникальный код, который он должен послать DeskTop'у в случае своего выбора, либо еще одну ссылку на



Объект tMenu является наследником tView, вставляется в список подэлементов DeskTop наравне с окнами. Данный объект имеет размеры, которые определяются количеством и положением на экране элементов горизонтального меню, то есть данный объект похож на прямоугольник, который полностью закрывает элементы горизонтального меню. Объект tMenu обладает теми же методами, что и tView. Задачей метода tMenu.Show является нарисовать пункты горизонтального меню. На первый из них ссылается указатель Elem:pMenuItem, на активный - Current:pMenuItem. Метод tMenu.HandleEvent анализирует нажатие кнопки F10 и кнопки мыши в своих координатах (пунктирная линия). Если это событие произошло, то объект tMenu становится модальным (единственным получателем всех событий), он вызывает метод ExecuteDialog объекта DeskTop. При нажатии на Enter или кнопку мыши вызывается подпрограмма, которая вычисляет размер, необходимый для подэлементов, рисует прямоугольник, заполняет его названиями пунктов вертикального меню. Дальнейшее нажатие Enter приводит либо к завершению модальности элемента tMenu и посылке соответствующего кода Code объекту DeskTop, либо к появлению нового уровня вертикального меню (если SubMenu<>nil).

С точки зрения Паскаля меню состоит из трех частей (Смотри схему).

```
tMenu=object(tView)
    Elem, Current: pMenuItem;
    Constructor Init(x1,y1,Sx,Sy:integer;p:pMenuItem);
    destructor Done; virtual;
    procedure Show; virtual;
    procedure HandleEvent(e:tEvent);virtual;
    procedure Execute; virtual;
   end;
procedure tMenu.HandleEvent(Var e:tEvent);virtual;
Var MyCode:word;
begin
  case e.What of
  cmMouse: if MouseContain(e.mx,e.my)
         then begin
           MyCode:=ExecuteDialog(@Self,nil);
            e.Code:=MyCode;
            e.What:=cmBroadCast;
            e.Addr:=DeskTop;
            e.InfoPrt:=@Self;
            PutEvent(e);
            ClearEvent(e)
          end;
     ... для клавиатуры выполняются аналогичные действия.
end;
```

Вызов ExecuteDialog в конечном итоге приведет к вызову tMenu.Execute, именно он должен отвечать за вызов подменю и анализ нажатий на кнопки. Объекты горизонтального меню как таковыми объектами не являются - это просто записи, объединенные в линейный однонаправленный список. Для работы с ними есть глобальная подпрограмма NewItem.

```
function NewItem(Name:string;Code:word; Commands:word;
                  Next:pMenuItem):pMenuItem;
Var p:pMenuItem;
begin
   new(p);
   p^.Name:=Name;{название пункта}
   p^.Code:=Code;{код, посылаемый при выборе данного пункта меню}
   p^.Command:=Command;{горячая клавиша}
   p^.SubMenu:=nil;{подменю нет}
   p^.Next:=Next;{прицепить следующий элемент}
   NewItem:=p;
end;
pMenuItem=^tMenuItem
pSubMenu=^tSubMenu;
tMenuItem=record
             Name:string[20];
             Command,Code:word;
             Next:pMenuItem;
             SubMenu:pSubMenu
          end;
tSubMenu=record
       Items:pMenuItem; {указатель на первый подэлемент списка}
       Default:pMenuItem; {указатель на элемент по умолчанию}
     end;
```

Запись типа tSubMenu предшествует каждому вертикальному подменю. Для создания вертикального подменю служит глобальная подпрограмма NewSubMenu.

```
function NewSubMenu(Name:string;Code:word;
Commands:word;SubMenu:pSubMenu Next:pMenuItem):pMenuItem;
Var p:pMenuItem;
begin
   new(p);   p^.Name:=Name;{название пункта}
   p^.Code:=Code;{код, посылаемый при выборе данного пункта меню}
   p^.Command:=Command;{горячая клавиша}
   p^.SubMenu:=SubMenu;{прицепить подменю}
   p^.Next:=Next;{прицепить следующий элемент}
   NewSubMenu:=p;
end;
```

Для создания заголовка подменю служит подпрограмма NewMenu:

```
function NewMenu(Items:pMenuItem):pSubMenu;
Var p:pSubMenu;
begin
   new(p);     p^.Items:=Items;   p^.default:=Items;
   NewMenu:=p
end;
```

Процесс создания меню будет выглядеть следующим образом:

```
Var menu:pMenu;
...
 new(Menu,Init(0,0,640,20,{указатель на первый эл. гор. меню}
     newSubMenu('ЭлГор1',1000,Alt_1,
      NewMenu( newItem('ЭлВер1',1001,Alt_a,
                  newItem('ЭлВер2',1002,Alt_b,
                  newItem('ЭлВер3',1003,Alt_c,nil)))
      ),
     newSubMenu('ЭлГор2',1010,Alt_2,
      NewMenu(  newItem('ЭлВер2_1',1011,Alt_d,
                  newItem('ЭлВер2_2',1012,Alt_e,
                  newItem('ЭлВер2_3',1013,Alt_f,nil)))
      ),
     ),nil)
   ));
 DeskTop^.Insert(Menu);
```

Указатель на Menu можно сохранить в специальном поле объекта tDeskTop.

Строка статуса

Строка статуса - необходимый элемент в широко развитых интерфейсах, в ее задачу входит сообщение пользователю справочной информации о том, что делает данный пункт меню, как выполнить ту или иную операцию и т.п. Работа со строками статуса чрезвычайно проста:

1) организуется глобальный массив строк, в котором перечисляются все строки-подсказки:

```
Const    StatusLine:array[0..100] of
  string=('', 'Строка 1','Строка 2', ...);
```

2) каждому видимому элементу при инициализации (через конструктор) передается константа, соответствующая номеру строки статуса в массиве StatusLine. Когда объект понимает, что он становится активным, то он посылает объекту tStatusLine сообщение типа cmStatusLine с кодом - номер элемента массива. Данный объект, получив сообщение, перерисовывает строку статуса.

Метод Idle

Этот метод вызывается в тот момент, когда нет событий, предназначенных для обработки, то есть e.What=cmNothing. Метод Run несколько изменяется:

```
procedure tDeskTop.Run Var e:tEvent;
begin
    настройка параметров;
    Repeat
      GetEvent(e);
      if e.What=cmNothing
      then Idle
      else HandleEvent(e);
  Until Quit
end;
```

Метод Idle – виртуальный, может содержать подсчет времени с момента последнего события, по прошествии 2 секунд, метод определяет какой объект находится под курсором мыши и рисует "облако" подсказки. Пользователь может уточнить метод Idle, расширив его, и, добавив в него свои функции, например, вывод текущего времени.

Объект tScrollBar

Данный объект используется для предоставления пользователю возможности быстрого путешествия по спискам (при помощи мышки) и определения масштаба окна (по строке прокрутки). Объект tScrollBar создается на основе объекта tView и имеет следующие характеристики:

```
tScrolBar=object(tView)
     Max,min:integer;{наибольшее и наименьшее значение счетчика}
    Value:integer; {текущее значение счетчика}
    Constructor Init(координаты и размеры);
  {Настройка параметров}
   procedure SetParams(AValue,AMin,AMax, APgStep:integer);
   procedure SetRange(AMax,AMin:integer); {настройка границ}
   procedure SetValue(AValue:integer);{установка текущего положения ползунка}
   procedure HandleEvent(Var e:tEvent);virtual;
end;
```

Работа строки скроллинга основана на методе обработки событий HandleEvent, который перехватывает события от мышки и клавиатуры. Если произошло событие, то строка скроллинга вычисляет новое положение ползунка (значение Value), а затем посылает своему владельцу (какому-либо списку или окну диалога) сообщение:

Message(Owner, cmBroadCast, cmScrollBarClicked, @Self);

кому, общее событие, сдвиг строки скрол., подпись

При инициализации владельцу строки скроллинга (списку) передается созданный объект tScrollBar: new(pList, Init(координаты, new(pScrollBar,init(...))));

Кстати, строк скроллинга может быть две: горизонтальная и вертикальная. Объект-владелец в методе обработки событий следит за сообщением своего объекта tScrollBar и перерисовывает себя в зависимости от положения ползунка в строке скроллинга:

```
procedure tList.HandleEvent(Var e:tEvent);virtual;
begin
  Inherited HandleEvent(e);
  case e.What of
    cmBroadCast:
          if (e.Code=cmScrollBarClicked) and (e.Addr=@Self)
          then begin
              if HorBar=e.InfoPtr{если это горизонтальная строка}
          then {обработать}
              if VertBar=e.InfoPtr {если это вертикальная строка}
              then {обработать}
        ...    end;
  end;
end;
```

Обработка события сводится к получению владельцем строки скроллинга значения Value при помощи метода GetPos и перерисовки себя в новом качестве. Указатели HorBar и VertBar настраиваются конструктором при создании объекта, если соответствующие строки нет, то передается nil. Конструктор так же должен настроить параметры строки скроллинга: начальное значение указателя, его шаг, шаг страницы и так далее. Строка скроллинга не может быть выделенным объектом (сфокусированным), чтобы объект правильно реагировал на нажатие клавиш на клавиатуре у него взводится флаг ofPostProcess, то есть если после опроса всех объектов ни один из них не обработал нажатие стрелки, то строка скроллинга забирает это событие себе. Линейки скроллинга вставляются в список подэлементов обычным образом, при помощи метода Insert. Метод Run должен быть несколько видоизменен, то есть три раза опрашивать объекты, каждый раз изменяя фазу обработки события.

```
procedure tDeskTop.Run
Var e:tEvent;
begin
   настройка параметров;
   Repeat
     GetEvent(e);
     Phase:=OfPreProcess;    HandleEvent(e);
     Phase:=OfProcess;            HandleEvent(e);
     Phase:=OfPostProcess;   HandleEvent(e);
   Until Quit
end;
```

Каждый объект, в соответствии со своими опциями, реагирует только на события, которые происходят во время нужной фазы.

OfPreProcess - облако подсказки;

OfProcess - все обычные объекты;

OfPostProcess - скроллинг.

Дальнейшее развитие оболочки

- Самой неприятной проблемой, возникшей при создании данной оболочки, является проблема потери события. Она возникает, когда два события происходят почти одновременно. Например, произошло внутреннее событие, а когда его обработка еще не завершена, возникло событие от мышки, оно будет потеряно, так как мышь в этот момент никто не опрашивает. Такие же проблемы могут возникнуть, когда два объекта захотят одновременно сгенерировать внутреннее событие. Одно из них будет потеряно. Выходом из такой ситуации является организация очереди событий. Метод PutEvent помещает событие не в переменную SaveEvent, а в кольцевой список. Метод GetEvent извлекает событие только из очереди, при этом он не читает ни мышь, ни клавиатуру. Для их опроса необходимы два обработчика прерываний: 09h и функцию 0Ch прерывания Int 33h (установить обработчик события). Обработчик от мышки вызывается, когда происходит какое-либо событие с мышкой (сдвинулась, нажали кнопки), он определяет, что же в действительности произошло с мышкой, затем формирует событие (координаты курсора, состояние кнопок) и помещает это событие в кольцевую очередь. Аналогично работает обработчик клавиатуры. В результате события в очередь поступают как бы по трем каналам.
- В данный момент можно считать, что вы уже познакомились с классическим механизмом передачи сообщений между объектами. Как уже говорилось ранее, существует альтернативный подход, связанный с передачей объекту указателя на подпрограмму, которая будет обрабатывать все его сообщения. Такой механизм используется в Windows. У каждого окна Windows имеется указатель на дальнюю подпрограмму WinProc, которая вызывается при наступлении любого события: нажатие клавиши, перемещение мыши, срабатывании кнопки окна, изменение состояния элементов окна и т.д. Подпрограмме WinProc передается указатель на параметры произошедшего события и код самого события.

Procedure WinProc (Event: word; Var e:tEvent); far;

Переменная Event хранит код события (их может быть очень много), а переменная е указывает на параметры события, кстати, запись с вариантами
tEvent можно расширить. При инициализации окна конструктору, вместо основных параметров передается структура-запись, описывающая окно:

```
Type   Proc= Procedure (Event: word; Var e: tEvent);
WindStruct=record
    X, Y, Width, High: integer; {координаты и размеры окна}
    Title: string;   {заголовок окна}
    WindP: Proc;  {адрес подпрограммы обработки событий}
    Color: byte;  {цвет окна}
    Attr: word;{атрибуты окна (возможность перемещения, изм. размеров) }
    …
end;
```

Перед созданием окна пользователь заполняет соответствующую структуру, а затем передает ее конструктору окна в качестве параметров. При возникновении какого-либо события вызывается не обработчик событий
DeskTop.HandleEvent, который передает события всем подэлементам, а конкретный обработчик нужного окна. Все подэлементы конкретного окна хранят адрес обработчика окна своего владельца, и при необходимости вызывают его.

```
Procedure WinProcMy1 (Event: word; Var e: tEvent); far;
Begin
  Case Event of
   CmMouseMove: {анализ перемещения мыши по окну}
   CmKeyDown:{анализ нажатия кнопок}
   CmSetFocus: {изменение активного элемента}
   CmBroadCast: {общее событие, возникает при изменении статуса подэлементов окна}
      Begin
          If e.Code=cmCOMMAND
          Then  begin {если сработал некоторый подэлемент}
                 Case e.addr of
                   IDB_Button1: {Нажата кнопка Button 1}
                        Begin
                         Выполнение действий, связанных с этой кнопкой;
                        End;
                    IDB_Button2: { Нажата кнопка Button 2}
                         Выполнение действий, связанных с этой кнопкой;
                    …
                  End;
              End;
    End;
  Else DefaultWindowProc(Event,e) {вызов обработчика по умолчанию}
 End;
End;
```

Каждому объекту при инициализации присваивается уникальный код – идентификатор, при изменении статуса подэлемент подписывается этим кодов (раньше мы использовали адрес объекта). Если окно не обработало событие, то оно передается стандартному обработчику событий DefaultWindowProc. Данная схема несколько проще классической, более компактна. Перед своим уничтожение окно получает сообщение cm_DESTROY, в этот момент оно может опросить свои элементы и вернуть их значения через параметры.