ГЛАВА 10

Объектно-ориентированное программирование

> В 1980-х годах объектно-ориентированное программирование будет занимать такое же место, какое занимало структурное программирование в 1970-х. Оно всем будет нравиться. Каждая фирма будет рекламировать свой продукт как созданный по этой технологии. Все программисты будут писать в этом стиле, причем все по-разному. Все менеджеры будут рассуждать о нем. И никто не будет знать, что же это такое.
>
> *Т. Ренч*

Ну вот, уважаемый читатель, мы и подошли к заключительной главе нашей книги. Нам кажется, что сейчас самое время остановиться, осмотреться и качественно оценить, что нам удалось изучить. Начали мы наше путешествие с основных этапов разработки программ: поговорили об анализе предметной области и постановке задачи, осознали важность построения математической модели и разработки алгоритма. Затем обсудили средства разработки и программирования в среде Windows, совершили стремительное восхождение к вершинам языка Object Pascal. Настало, наконец, время задаться последним вопросом нашей книги: «Что же такое *Object* в названии языка программирования, с которым мы работали все это время?».

В данной главе мы с вами познакомимся со значением этого слова в названии языка, узнаем, как сегодня разрабатываются по-настоящему большие программы. Выглядит заманчиво? Тогда за дело!

1. И снова о технологиях

Вернемся к материалу главы 5. В этой главе мы определили программирование как технологический процесс и рассмотрели понятие *технологии программирования*. В ходе дальнейшего изложения мы познакомились с двумя технологиями – структурным и модульным программированием. Достоинства этих подходов к разработке программ не вызывают сомнений. Это внесение порядка и стройности изложения в тексты программ, это разбиение задачи на более простые подзадачи и решение

подзадач по частям, это некоторые возможности по раздельной компиляции (а следовательно, и упрощению отладки) и повторному использованию кода и многое другое. Однако жизнь не стоит на месте. Аппаратная база компьютеров развивается семимильными шагами. Интуитивно ясно, что развитие вычислительной техники должно приводить к возможности решения при ее помощи все более и более сложных задач, и это действительно так, однако обратная сторона медали состоит в необходимости разработки все более и более сложных программ. Программы становятся большими (даже очень большими), а разработка – коллективной. Объем работы увеличивается, коллективы разрастаются, код «разбухает», и появляются новые проблемы, которых не было раньше. Неожиданно выясняется, что возможности структурного и модульного программирования ограничены и зачастую уже не позволяют добиваться желаемого результата (либо ничего не работает, либо проект не укладывается в сроки либо в бюджет, либо через год после написания программы выясняется, что ее невозможно модифицировать и т.д.).

Для большего понимания сказанного выше приведем пример из другой предметной области. Вспомним, как радовались люди 30 лет назад, когда им удавалось приобрести телевизор. Быть может, вы еще застали эти огромные ящики с лампами (например, используете их в качестве тумбочки). Современные телевизоры, стоящие у большинства из нас дома, сделаны по совершенно другой технологии. По своим техническим и прочим характеристикам они значительно превосходят своих ламповых предков. Понадобилось улучшить параметры – разработали новые технологии и новую элементную базу. В наши дни вы можете обнаружить в соответствующих магазинах телевизоры, в которых отсутствует электронно-лучевая трубка. Эти телевизоры являются «плоскими», занимают мало места, их можно повесить на стенку и т.д. Понадобилось улучшить параметры – вновь появились новые технологии и новая элементная база.

Аналогично и в программировании. Нужно оптимизировать процесс создания программ – разрабатываются новые технологии. Одна из них основана на объектном подходе и в русскоязычной литературе носит название «объектно-ориентированное программирование». В данной главе мы рассмотрим основные идеи этого подхода и способы его реализации в языке программирования Object Pascal. Сразу оговоримся, что данной теме посвящено немало прекрасных книг и монографий [5, 33, 36, 39, 44]. Мы не ставим целью полностью рассмотреть все аспекты этой большой и сложной темы. Мы сделаем введение в тему, полагая, что наш читатель при необходимости (которая у него наверняка возникнет, если он захочет связать свою жизнь с программированием) продолжит изучение самостоятельно. Сразу оговоримся, что данная тема является достаточно

сложной. При первой возможности мы будем иллюстрировать сказанное примерами для лучшего понимания материала.

2. Объектный подход

Не вдаваясь в историю развития *объектно-ориентированной технологии программирования* (прекрасный обзор содержится, например, в [5]), рассмотрим ее основное смысловое наполнение. В основе этой технологии лежит так называемый *объектный подход* [5, 33, 36, 39, 44]. Для того чтобы понять, в чем он заключается, изучим его основные отличия от других традиционных подходов к разработке программ.

2.1. Алгоритмическая и объектная декомпозиции

Известный нам к настоящему моменту способ разработки связан с представлением программы как совокупности данных и алгоритмов, связанных друг с другом. В рамках этого подхода анализ предметной области (а именно с него все и начинается) представляет собой попытку получить ответы на следующие вопросы:

- Какую информацию (данные) нам необходимо хранить?
- Какими алгоритмами мы будем обрабатывать эти данные?
- Каковы типовые сценарии работы с анализируемой областью (какие алгоритмы нужно и в каком порядке можно выполнять)?

Как мы обычно отвечаем на эти вопросы? Сначала определяем структуры для представления данных предметной области. После этого наращиваем функциональность программы, дописывая к ней алгоритмы, некоторые из которых связаны друг с другом. Эти алгоритмы получают на вход и возвращают в качестве выхода данные из предметной области. В результате, в качестве основного «*строительного блока*» в программе выступает *алгоритм*. Итак, наша обычная модульная программа есть

Данные (разбитые на модули) + Алгоритмы (разбитые на модули).

Обратим внимание на несколько моментов.

1. Как уже говорилось выше, алгоритмы связаны с обрабатываемыми ими данными.

2. Эти связи крайне трудно вычленить непосредственно из текста программы, они существуют лишь в голове программиста то непродолжительное время, пока он пишет соответствующий фрагмент

программы. Через год после ее написания понять чт∴, с чем и как связано – грандиозная проблема.

3. Замена представления реально существующих объектов предметной области абстрактными структурами и алгоритмами для их обработки отрицательно сказывается на понимании того, «как оно там на самом деле все устроено». Одним словом, подобное представление неестественно, что плохо.

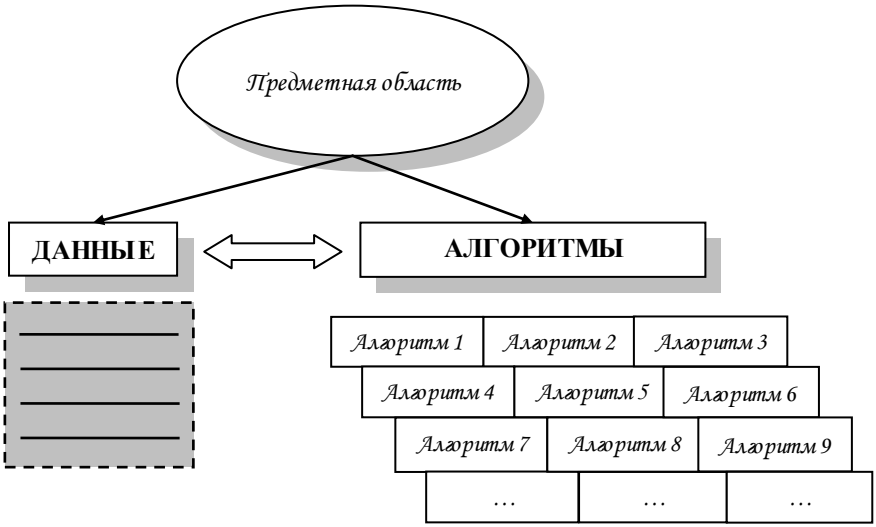

Рис. 10.1. Алгоритмическая декомпозиция.
Основной «строительный блок» – алгоритм

4. Сплошь и рядом встречаются «перекрестные ссылки» между модулями. В простейшем случае А использует В, а В использует А. Поверьте, все бывает намного более запутано. На этапе написания проекта подобные «обходные пути» – перекрестные ссылки – строго запрещаются. Но как проконтролировать, что этому запрету будут строго следовать? Наивно полагать, что все 100% исполнителей до конца понимают суть проблемы, которая может возникнуть от того, что они поленятся использовать специальный сложный механизм, пойдя вместо этого напрямик. Итог: в модульных программах почти нет средств контроля правильности действий ваших коллег (мы сами всегда действуем верно, не так ли?).

5. В модульных программах слабо представлены средства, которые позволили бы легко модифицировать код, заменяя отдельные структурные части программы другими, более совершенными. Декларировано, что такие средства есть, но их мощь оставляет желать лучшего.

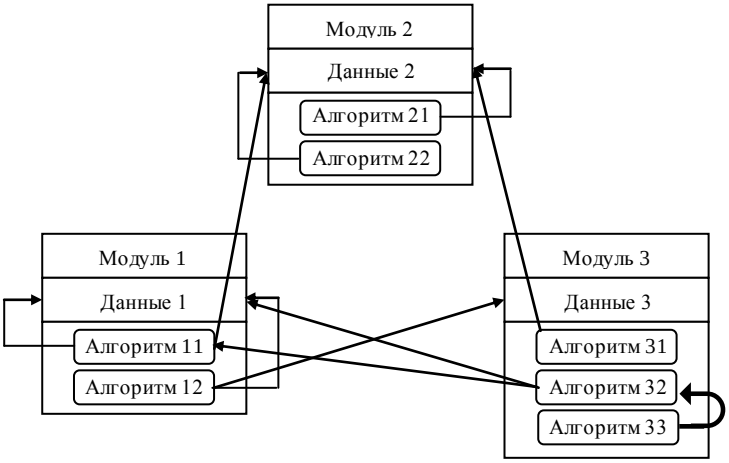

Рис. 10.2. Типовая схема модульной программы

Этот список можно продолжать. Мы показали лишь некоторые из проблем, но уже этого достаточно, чтобы понять необходимость изучения другого подхода к анализу, проектированию и разработке программ.

В основе объектного подхода лежит так называемая *объектная декомпозиция*, смысл которой состоит в том, что предметная область рассматривается как совокупность сущностей (абстракций), которые характеризуются своими данными и обладают строго определенным поведением, описываемым алгоритмами.

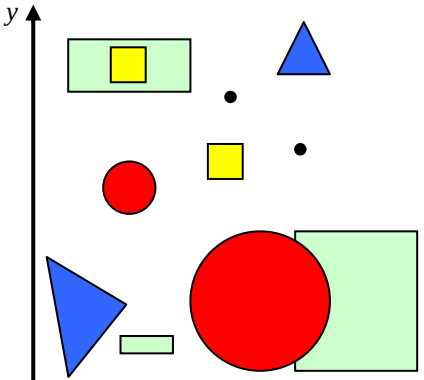

Рис. 10.3. Пример предметной области: координатная плоскость с фигурами

Тем самым, если раньше мы могли рассматривать координатную плоскость как иксы и игреки, а также алгоритмы рисования, скрытия, вычисления площадей и т.д., то теперь для нас координатная плоскость – прежде всего набор фигур определенных типов. Рассмотрим объектную декомпозицию для этого примера. Для начала нарисуем пример координатной плоскости с фигурами. Пусть нашей задачей является разработка программных средств для отображения подобных наборов фигур в декартовой системе координат на плоскости и выполнения над ними типовых операций (отображение, скрытие, перемещение, вычисление площадей). Будем считать, что на рисунке представлена типовая координатная плоскость для рассматриваемой задачи.

Следуя принципам объектной декомпозиции, проведем *анализ предметной области*. Для этого выделим основные ее элементы, концентрируя внимание не на том, сколько там каких фигур (на разных плоскостях будет по-разному), а на том, какие именно фигуры там есть.

Вот результат нашего анализа (основные *абстракции*, если говорить научно):

1. Точка.
2. Квадрат.
3. Круг.
4. Треугольник.
5. Прямоугольник.

Все? Больше ничего не видите? А если подумать?

Если подумать, то становится ясно, что сама координатная плоскость – шестой пункт нашего списка основных абстракций. Она тоже имеет некоторые свои данные и четко выраженное поведение (понятно, какие алгоритмы ее обрабатывают).

Вот мы и сделали *первый шаг – выделили абстракции*.

Второй шаг – необходимо понять, какими данными характеризуются выделенные нами абстракции.

1. *Точка*: (*x*, *y*) – координаты.
2. *Квадрат*: (*x*, *y*, *a*) – левый нижний угол и сторона. Стоп! Внимательный читатель должен здесь заметить, что так мы описываем квадраты, стороны которых параллельны осям координат. Пусть для простоты именно такие квадраты нас интересуют.
3. *Круг*: (*x*, *y*, *r*) – центр и радиус.
4. *Треугольник*: (*x*1, *y*1, *x*2, *y*2, *x*3, *y*3) – координаты вершин.

5. *Прямоугольник*: (*x*1, *y*1, *a*, *b*) – левый нижний угол и длины сторон (то же замечание, что и для квадрата).

6. *Координатная плоскость* – набор фигур типов 1–5. Хотелось бы написать «массив фигур». Хотелось бы, но… Элементы разнотипные. Бороться с этой проблемой мы научимся позже, а пока скажем обтекаемо – набор.

Некоторые соображения по поводу проведенного анализа данных:

- помимо указанного выше способа прямоугольник может быть представлен: координатами двух точек – противоположных углов – (*x*1, *y*1, *x*2, *y*2) или как два объекта «Точка» – Точка1 и Точка2;
- квадрат и круг с точки зрения реализации данных – одно и то же с точностью до обозначений;
- треугольник может быть представлен как (Точка1, Точка2, Точка3).

Как видим, результатом анализа редко бывает единственный возможный вариант описания предметной области. Искусство аналитика – сделать это описание наиболее удобным для дальнейшего использования.

Вот мы и сделали *второй шаг – определили, какими данными описываются наши абстракции.*

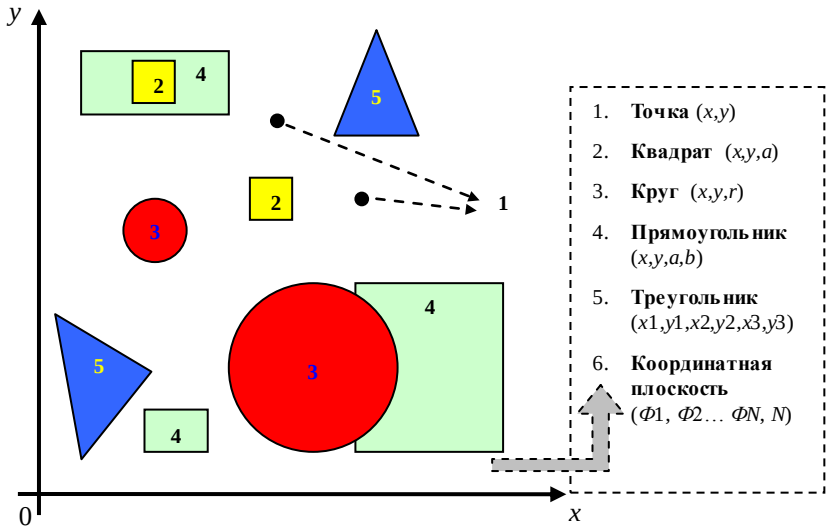

Рис. 10.4. Пример предметной области: координатная плоскость с фигурами. Параметры фигур

Третий шаг – определение, какими алгоритмами обрабатываются эти данные для каждого типа абстракций.

Введем основные операции:

1. **Show** – показать.
2. **Hide** – скрыть.
3. **S** – считать площадь (кроме точки).
4. **MoveTo**(dx, dy) – переместить на dx вправо и dy вверх (кроме координатной плоскости).

Список можно продолжать, но нам на первое время должно этого хватить. Заметим, что все эти операции есть у каждого типа абстракций 1–6, но вот работают они для них по-разному.

Ну вот, с анализом все или почти все. Осмотрим бегло складывающуюся картину и оценим, не забыли ли мы что-то важное? Конечно, забыли! Признак видимости. Если мы собираемся рисовать и скрывать фигуры, то у каждой из них обязательно должен быть признак видимости.

Итог: добавляем каждой из абстракций элемент данных Visible и операцию IsVisible – проверку, видна ли фигура на экране.

В итоге получаем:

1. *Точка*:
   — данные: (*x*, *y*, Visible);
   — операции: Show, Hide, MoveTo, IsVisible.
2. *Квадрат*:
   — данные: (*x*, *y*, *a*, Visible);
   — операции: Show, Hide, S, MoveTo, IsVisible.
3. *Круг*:
   — данные: (*x*, *y*, *r*, Visible);
   — операции: Show, Hide, S, MoveTo, IsVisible.
4. *Треугольник*:
   — данные: (*x*1, *y*1, *x*2, *y*2, *x*3, *y*3, Visible);
   — операции: Show, Hide, S, MoveTo, IsVisible.
5. *Прямоугольник*:
   — данные: (*x*1, *y*1, *a*, *b*, Visible);
   — операции: Show, Hide, S, MoveTo, IsVisible.
6. *Координатная плоскость*:
   — данные: (набор фигур, Visible);
   — операции: Show, Hide, S, IsVisible.

То, что мы получили, «в первом чтении» можно считать *результатом объектной декомпозиции*. Почему «в первом чтении»? Потому, что мы не выделили связи между обнаруженными абстракциями. Об этом несколько позже.

2.2. Немного терминологии

Прежде чем двигаться дальше, поговорим о терминах. Авторы книги надеются, что наш читатель будет активно изучать литературу по программированию. Без знания стандартной терминологии в ходе этого процесса не обойтись.

В прошлом разделе мы узнали, что процесс выделения основных абстракций в предметной области называется *объектной декомпозицией*. Заметим при этом, что сами выделенные абстракции называются *классами*, а их экземпляры с конкретными данными – *объектами*. Так, для приведенного на рис. 10.4 примера мы имеем 6 классов (точка, квадрат, круг, треугольник, прямоугольник, координатная плоскость) и 12 объектов (2 точки, 2 квадрата, 2 круга, 2 треугольника, 3 прямоугольника и 1 координатная плоскость).

Теперь вкратце остановимся на используемой в литературе терминологии, касающейся разных частей объектного подхода. В западной литературе принятым является следующее представление:

Объектный подход = ОО Анализ + ОО Проектирование + ОО Программирование.

Объектно-ориентированный анализ (**OOA** – object-oriented analysis) – это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области [5].

Объектно-ориентированное проектирование (**OOD** – object-oriented design) – это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы [5].

Объектно-ориентированное программирование (**OOP** – object-oriented programming) – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования [5].

Как видим, собственно программирование начинается на третьем этапе, а технология регламентирует также и предшествующие программированию действия. Обратим внимание на небольшую терминологическую путаницу: в отличие от западной литературы в русскоязычной под термином *объектно-ориентированное программирование* (**ООП**), как правило, понимают все три составляющие объектного подхода. В дальнейшем изложении и мы не будем отступать от этой традиции.

2.3. Основные идеи объектного подхода

Итак, выше мы познакомились с примером объектной декомпозиции в простейшей задаче о фигурах на координатной плоскости. На том же примере мы постараемся рассмотреть основные идеи объектного подхода.

Прежде всего, необходимо понять, чт.∴ хорошего в новом для нас способе анализа предметной области. Основной плюс состоит в том, что вместо «разрозненных» данных и алгоритмов теперь мы имеем в качестве результата набор абстракций (классов) с их данными и операциями (алгоритмами обработки этих данных). То есть предметная область предстает перед нами как совокупность классов и объектов, «внутри» которых находятся данные и алгоритмы.

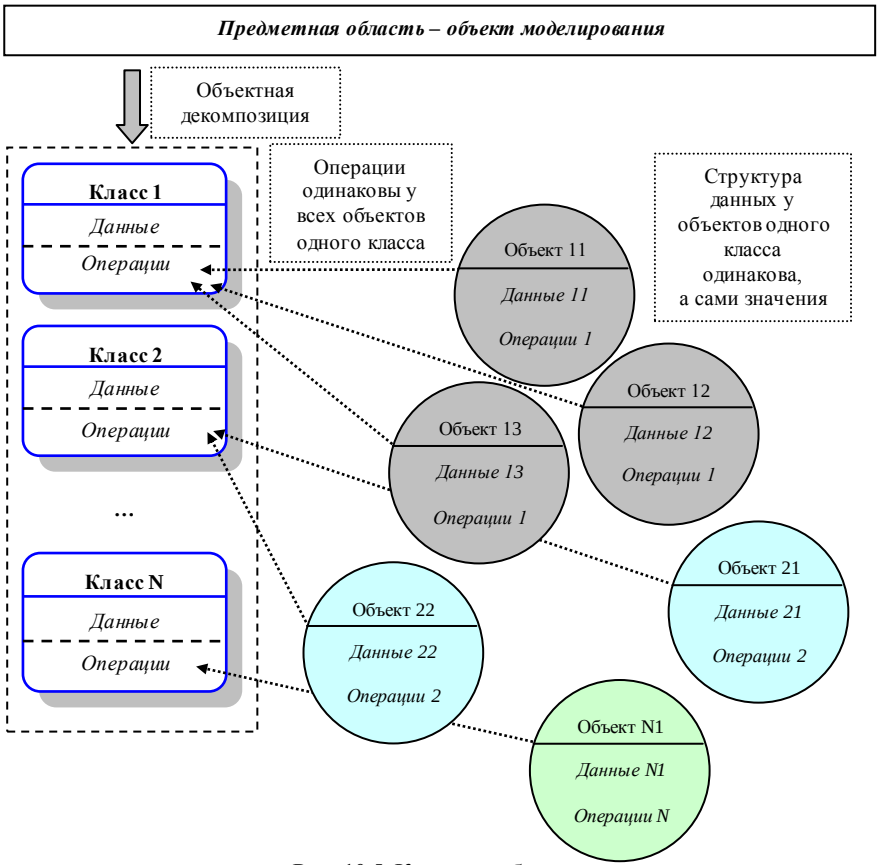

Рис. 10.5. Классы и объекты

Традиционно считается, что объектный подход базируется на трех основных понятиях: инкапсуляция, наследование и полиморфизм. Западные аналитики (см., например, в [5, 36, 37]) выделяют существенно больше составляющих объектного подхода, но мы ограничимся тремя перечисленными, считая их основными. Далее мы рассмотрим каждую из них более подробно.

2.3.1. Инкапсуляция. Под инкапсуляцией[1] понимаются следующие два тезиса:

1. Объединение данных и операций их обработки в рамках одной синтаксической структуры языка программирования.
2. Наличие механизма скрытия данных.

Первый тезис иллюстрирует важнейшее преимущество объектного подхода. Одна из основных проблем разработки программного обеспечения в рамках технологии модульного программирования состоит в том, что невозможно понять, какие алгоритмы обрабатывают те или иные данные. Синтаксически это никак не видно, информация об этом существует лишь в голове программиста то недолгое время, пока он пишет соответствующий фрагмент программы. Через несколько месяцев, чтобы понять, как связаны данные и алгоритмы, необходимо будет предпринять титанические усилия. В ООП мы вместо разрозненных алгоритмов и данных имеем классы, глядя на объявления которых можно сделать однозначный вывод о том, к каким данным какие операции в нашей модели предметной области применяются. Посмотрим на фрагмент объявления класса «Круг» (полностью правила объявления классов мы узнаем чуть позже):

```
TCircle = class
public
   x, y: Real;          { Координаты центра }
   r: Real;             { Радиус            }
   Visible: Boolean;    { Признак видимости }

   function GetS: Real; { Расчет площади круга }
end;
```

Совершенно ясно, что конструкция **class** объединила данные и операции воедино. Представленное объявление само «говорит», что функция `GetS` обрабатывает данные класса `TCircle`. Какие именно? Для ответа на этот вопрос достаточно посмотреть на реализацию функции:

---

[1] От английского «to encapsulate» – заключать в капсулу.

```
function TCircle.GetS: Real; {Расчет площади круга}
begin
   Result := PI * r * r;
end;
```

Итак, функция использует радиус и константу `PI`. Заметим также, что у функции отсутствуют аргументы – нужные ей данные она «берет» из класса, которому принадлежит.

Теперь перейдем ко второму тезису инкапсуляции. Что такое скрытие данных? В обычных модульных программах мы часто сталкиваемся с такой проблемой: как бы сделать так, чтобы наши коллеги своими действиями не смогли испортить результаты нашего труда. Не стоит думать при этом, что коллеги жаждут нам навредить. Понятно, что навредив они отодвинут сроки реализации проекта, чем добавят в том числе и себе проблем и головной боли. Но! Вы будете удивлены, узнав, насколько часто они способны что-то испортить неумышленно (конечно, это не про вас! Известно, что 99% программистов пишут код лучше, чем их коллеги).

Рассмотрим простой пример. Пусть вы написали некоторую библиотеку, реализовав там необходимые структуры для хранения данных и набор функций для их обработки. После этого вы передали библиотеку на использование другим программистам. В составленной вами инструкции черным по белому написано: «Не использовать прямые обращения к структурам данных! Использовать только специальные функции». Многие ли последуют вашим рекомендациям? Знайте, что как минимум половина программистов вообще не будет читать вашу инструкцию. Они сначала попробуют разобраться с библиотекой «методом научного тыка» (да и плох тот программист, который не попробует…). У них все заработает, и они успокоятся. Вы спросите – в чем проблема? Проблема в том, что когда вы захотите переписать структуры данных в библиотеке и поставить ее обновленную версию коллегам, у тех все перестанет работать и они извергнут на вас жуткие проклятья, искренне полагая, что правы. Ведь у них же раньше все работало! И на ваши оправдания, что на странице 347 инструкции было написано, что прямые обращения к структурам хранения запрещены (и это правильно, иначе вы не сможете развивать и оптимизировать свою библиотеку), никто не обратит внимания. Что с этим делать? Необходимы синтаксические механизмы скрытия данных и алгоритмов от несанкционированного использования. Такие механизмы – секции **`private`** (скрытые) и **`public`** (открытые) в классах языка Object Pascal.

Так, в следующем примере не удастся получить доступ извне к полю `Visible`, для этого предоставляется специальная функция `IsVisible`, которая позволяет узнать значение элемента данных `Visible`, но не

изменить его. Заметим, что соображения за и против скрытия данных могут быть разными. В данном примере разумность скрытия `Visible` вытекает из того, что в противном случае пользователь класса (коллега-программист) получит возможность менять признак `Visible` как угодно, в то время как круг может быть виден на координатной плоскости, а может быть нет. Возможность свободного изменения параметра `Visible` может привести к рассогласованию внутренних данных объекта и его внешнего представления, чего, конечно, быть не должно.

```
TCircle = class
private
   Visible: Boolean;          { Признак видимости }
public
   x, y: Real;                { Координаты центра }
   r: Real;                   { Радиус            }

   function GetS: Real;       { Расчет площади круга }
   function IsVisible: Boolean; { Возврат признака видимости }
end;
```

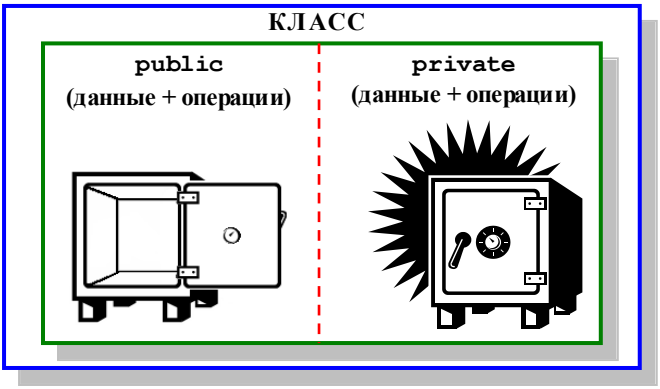

Рис. 10.6. Инкапсуляция – совместное проектирование данных и операций + разграничение доступа

Итак, инкапсуляция позволяет скрывать детали внутренней реализации, а значит, дает возможность проводить модификацию, не затрагивая код, который использует наши классы.

2.3.2. Агрегация и наследование. Одна из наиболее актуальных проблем в отрасли разработки программного обеспечения – многократная реализация одного и того же, так называемое «программирование в корзину». Удивительно, как много людей в одно и то же время программируют сходные реализации одинаковых алгоритмов в идентичных

задачах. Хуже того, один и тот же программист в своей практике неоднократно переписывает код, который уже был реализован ранее в одной из старых программ. В результате работа приобретает характер многократного бессмысленного преодоления одних и тех же препятствий. Предпосылки к возникновению подобной ситуации состоят еще и в том, что «выдрать» некие фрагменты кода из старой программы и перенести их в новую подчас является достаточно сложной задачей. Нередко оказывается проще переписать код заново, что и делается.

Одним из ключевых моментов в ООП является *возможность создания нового программного кода (реализующего новую функциональность) при помощи уже написанного ранее*. Преимущество такого подхода очевидно – ранее написанный код обычно является отлаженным и работоспособным. Достаточно важно и то, что мы можем использовать и новую, и старую редакции кода в одной и той же программе. Это обеспечивает существенное упрощение модификации и отладки. Посмотрим, как это делается.

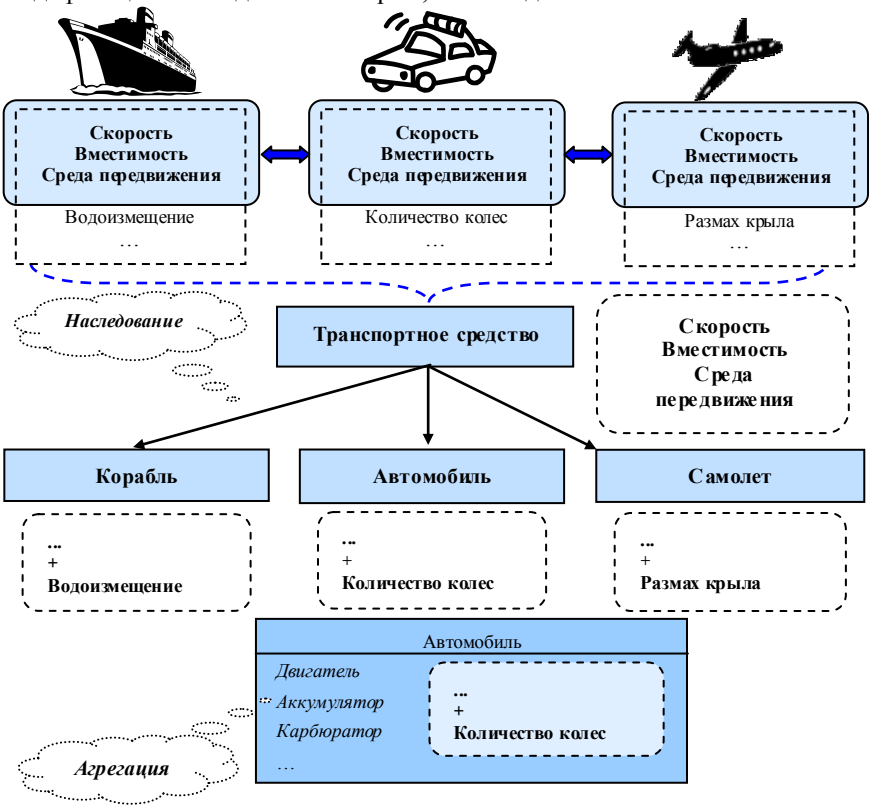

Рис. 10.7. Агрегация и наследование

В предыдущем пункте мы не только научились выделять абстракции в предметной области, но и слегка коснулись вопроса о скрытии деталей реализации. Применение этих составляющих объектного подхода существенно упрощает понимание задачи. Но и этого часто оказывается недостаточно.

В таких случаях на помощь приходит установление связей между классами. Принципиально можно выделить по крайней мере 2 типа отношений: «*это есть*» и «*быть частью*» [5]. Для их описания применяются два термина: *наследование* и *агрегация*.

Посмотрим на примеры.

Таблица 10.1

Примеры агрегации и наследования

«Это есть» (наследование)	«Быть частью» (агрегация)
КРУГ – это такая **ТОЧКА**, у которой есть все, что есть у точки, + добавился радиус и переписан ряд операций	**КРУГ** = **ТОЧКА** + радиус + некоторые новые операции
ТРЕУГОЛЬНИК – это такая точка, у которой есть еще 2 точки и переписан ряд операций	**ТРЕУГОЛЬНИК** = **ТОЧКА** + **ТОЧКА** + **ТОЧКА** + некоторые новые операции
ЖИВОТНОЕ – умеет перемещаться. **ПТИЦА** – **ЖИВОТНОЕ**, которое умеет летать. **СОЛОВЕЙ** – **ПТИЦА**, которая умеет не только летать, но еще и петь…	
	ПК = Монитор + Системный блок + Периферия Системный блок = Материнская плата (с устройствами) + Блок питания + Провода + … Материнская плата (с устройствами) = Процессор + Память + Видеоадаптер + … + Чипсет Видеоадаптер = …

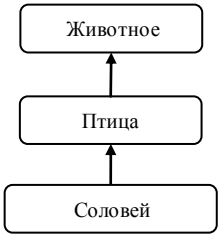

Рис. 10.8. Схема (иерархия) наследования для животных

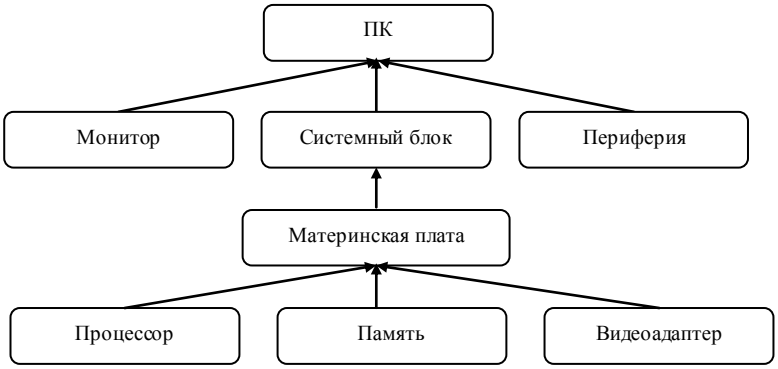

Рис. 10.9. Схема (иерархия) наследования для ПК

Объединение абстракций в иерархии позволяет рассматривать задачу последовательно на разных уровнях. Спускаясь вниз по дереву, мы увеличиваем степень детализации. Заметим, что кто-то понимает задачу на самом верхнем уровне и может лишь выкидывая половину компьютера устранять неисправность, кто-то на следующем уровне (может менять блоки), ну а кто-то для примера с ПК разбирается на уровне радиодеталей и может с паяльником в руках заменять вышедшие из строя запчасти. Чуть позже мы посмотрим, как абстракция и иерархия реализуются в синтаксисе языка Object Pascal.

2.3.3. Полиморфизм. Слово «полиморфизм» происходит от греческого polymorphos – многообразный. Примеры полиморфного поведения нам уже известны. Так, обратим внимание на операцию «+» в языке Pascal. Очевидно, что для целых чисел, вещественных чисел и, наконец, для строк она выполняет совершенно разные действия, сохраняя семантику (смысл)

использования и обозначение. Таким образом, «+» является полиморфной операцией.

Возникает следующая проблема: полиморфизм – это очень удобно (подумайте, каким кошмаром для программистов обернулась бы необходимость по-разному обозначать «+» для разных типов данных), но он существует для встроенных типов данных. А можно ли реализовать полиморфное поведение во вновь создаваемых типах данных (классах)? Одно из весьма существенных достоинств объектного подхода состоит в том, что он (и его реализация в языке Object Pascal) позволяет реализовывать полиморфное поведение в классах!

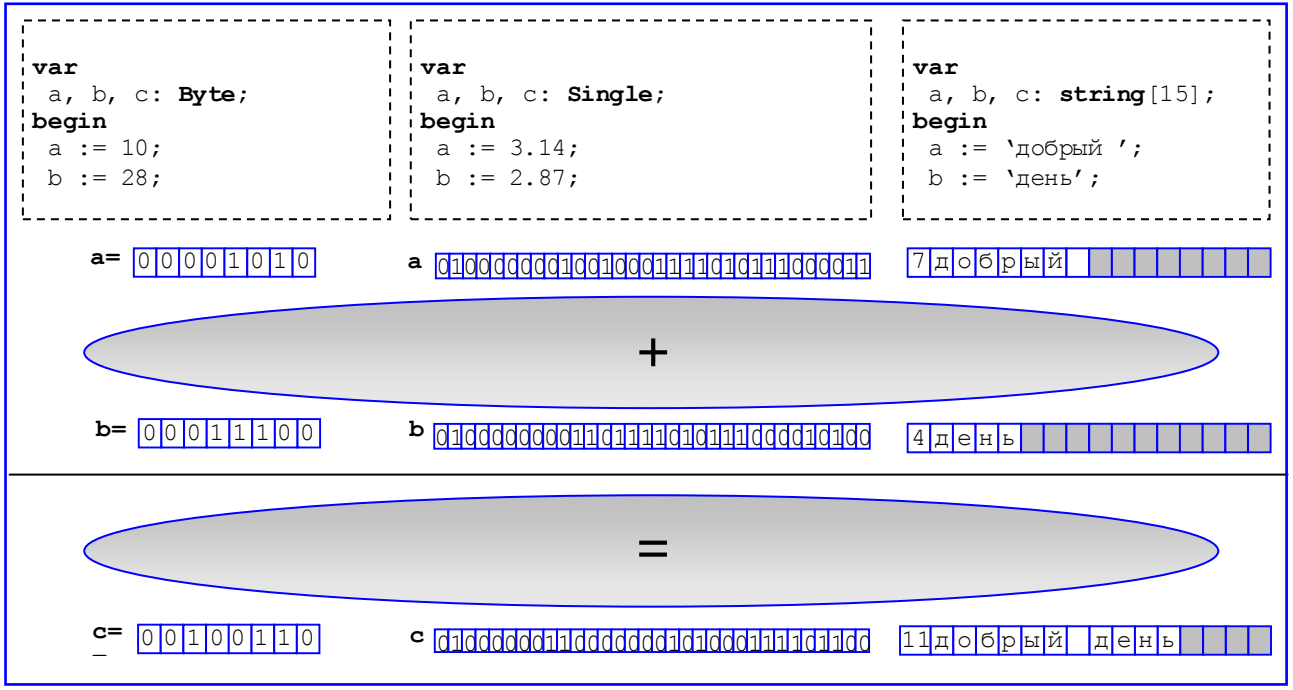

Рис. 10.10. Полиморфная операция «+» реализуется по-разному для разных типов данных

2.4. Резюме

В данном разделе мы сделали много анонсов, объясняя достоинства объектного подхода «на пальцах». Самое время перейти к рассказу о его реализации в языке Object Pascal.

3. Объектно-ориентированное программирование на языке Object Pascal

3.1. Вспоминая о записях

Для того чтобы наилучшим образом понять суть вопросов, связанных с синтаксисом ООП в языке Object Pascal, вновь обратим свой взгляд на записи (**record**). В частности, давайте опишем в виде записи новый тип данных «Круг».

```
type
  TCircleRec = record
    X, Y: Real;          { Координаты центра }
    R: Real;             { Радиус            }
    Visible: Boolean;    { Признак видимости }
  end;
```

Теперь, создавая переменные типа `TCircleRec`, мы получаем возможность хранить данные кругов. А как быть с операциями? Операции придется написать отдельно. Рассмотрим некоторые из них для примера. Дополним пример записью «Квадрат».

Будем «рисовать фигуры» при помощи условно существующих процедур рисования `Point`, `Circle` и т.д. Описания способов рисования в окне можно найти в справке по библиотеке `VCL` (см. методы класса `TCanvas`).

```
{ ================================================ }
{ Пример 10.1 }
{ Представление фигур на плоскости при помощи записей }
type
  TSquareRec = record
  X, Y: Real;          { Координаты левого нижнего угла }
  A: Real;             { Сторона                         }
  Visible: Boolean;    { Признак видимости               }
end;
```

```
TCircleRec = record
   X, Y: Real;          { Координаты центра }
   R: Real;             { Радиус            }
   Visible: Boolean;    { Признак видимости }
end;

var
   cr: TCircleRec;
   sq: TSquareRec;

procedure ShowSquare(_sq: TSquareRec); { Показать квадрат }
begin
   if not _sq.Visible then
   begin
      _sq.Visible := True;
      Square(_sq.X, _sq.Y, _sq.A);
   end;
end;

procedure ShowCircle(_cr: TCircleRec); { Показать круг }
begin
   if not _cr.Visible then
   begin
      _cr.Visible := True;
      Circle(_cr.X, _cr.Y, _cr.R);
   end;
end;

function SSquare(_sq: TSquareRec): Real; { Площадь квадрата }
begin
   Result := _sq.A * _sq.A;
end;

function SCircle(_cr: TCircleRec): Real; { Площадь круга }
begin
   Result := PI * _cr.R * _cr.R;
end;

{ Основная программа }
begin
   with cr do
   begin
      X := 100; Y := 100; R := 20;
   end;
   with sq do
   begin
      X := 200; Y := 100; A := 50;
   end;
```

```
   ShowCircle(cr);
   ShowSquare(sq);
   WriteLn('Площадь квадрата = ', SSquare(sq));
   WriteLn('Площадь круга = ', SCircle(cr));
end.
{ ================================================= }
```

Проанализировав пример, можно сделать следующие выводы:

1. Операции «оторваны» от данных. Нет никакой возможности привязать подпрограммы к записям.

2. Реализуя аналогичные по смыслу операции для круга и квадрата, мы вынуждены каждый раз придумывать новые имена подпрограмм (`ShowCircle, ShowSquare...`). А что если типов фигур будет 10, плюс у каждой по 15 аналогичных операций? 10 x 15 = 150 подпрограмм. Не много ли?

3. Наличие разных имен не позволит нам записать, к примеру, показ всех фигур в цикле:

```
for i := 1 to N do
   Show(Figures[i]);
```

Как нетрудно понять, вызвано это тем, что у каждого типа фигур операция «показать» называется по-разному. Более того, нам даже не удастся объявить «массив фигур», из-за того, что все они разнотипные. Эту проблему можно решить, объявив массив указателей на фигуры. Но как компилятор должен разобраться, как именно надо показывать фигуру `Figures[i]^`, если реальный тип содержимого указателя определится лишь на этапе работы программы?

4. Заметим, что любой программист может написать `cr.Visible := False`; в то время как на самом деле фигура видима на экране. В записях нет никаких средств, чтобы защитить поле `Visible` от несанкционированного доступа.

Это лишь вершина айсберга, те немногие, но очень существенные проблемы, которые видны даже из этого тривиального примера. Существуют и другие трудности, практически не преодолимые без использования ООП. Посмотрим, как решает эти и некоторые другие проблемы объектная модель языка программирования Object Pascal.

3.2. Объявление класса. Поля и методы

Для лучшего понимания материала обратимся к самому началу применения объектного подхода – к анализу предметной области. Пусть наша предметная область – уже знакомая нам координатная плоскость (см. раздел «Алгоритмическая и объектная декомпозиции»). В результате анализа

были выделены 6 абстракций, в том числе «Квадрат» со следующим описанием:

Квадрат:
— данные: (`x`, `y`, `a`, `Visible`);
— операции: `Show`, `Hide`, `S`, `MoveTo`, `IsVisible`.

Важное замечание: считаем стороны квадрата параллельными осям координат.

Первая часть описания квадрата – результат проектирования данных.

Вторая часть (вместе с алгоритмами осуществления указанных операций, в данном примере они тривиальны, но так бывает не всегда) – результат проектирования операций.

В итоге перечень выполненных нами действий выглядит так:

- выделена абстракция, существенная для предметной области;
- определено, какими данными характеризуется каждая такая абстракция;
- определено, какие операции можно выполнять над такими абстракциями.

Эти положения задают две основных составляющих типа данных – *типа данных* «Квадрат».

В терминах ООП мы получили характеристику класса. Таким образом, класс – абстрактный тип данных, средство моделирования объектов предметной области.

Заметим, что с языком программирования Object Pascal связаны две объектные модели – новая, появившаяся в Delphi, и старая, существовавшая еще в Borland Pascal для операционной системы MS DOS. В старых книгах вы еще можете увидеть объявления класса в виде:

```
type
  TMyClass = object
  ...
end;
```

Эта модель морально устарела, хотя и поддерживается до сих пор в целях обратной совместимости. Новая объектная модель является существенно более мощной. В данной книге мы рассматриваем новую модель, которую идентифицирует конструкция **`class`**.

Поскольку класс – тип данных, необходимо уметь его объявлять. Делается это как обычно в секции **`type`**. Простейший синтаксис объявления класса (в ходе изучения материала мы будем его уточнять) выглядит так:

```
type
  TMyClass = class
```

```
    < данные >
    < операции >
end;
```

Описание класса «Квадрат» может в первом приближении выглядеть так:

```
type
 TSquare = class
   { ---------- Данные ------------ }
   x, y: Real; { Координаты левого нижнего угла }
   a: Real; { Сторона квадрата }
   Visible: Boolean; { Признак видимости }
   { ---------- Операции ------------ }
   procedure Show; { Показать }
   procedure Hide; { Скрыть }
   function IsVisible: Boolean;{ Вернуть Visible }
   function S: Real; { Вычислить площадь }
   procedure MoveTo(dx, dy: Real); { Переместить на dx, dy }
end; { TSquare }
```

Таким образом, мы определили, какими данными будут обладать переменные класса «Квадрат», а также какими операциями их можно обрабатывать. Немного терминологии:

1. `TSquare` – класс, переменные типа `TSquare` – объекты.
2. Данные класса обычно называются *полями* (в англоязычной литературе *member-fields*).
3. Операции класса обычно называются *методами* (в англоязычной литературе *member-functions*).

Итак, *класс = поля + методы*.

Заметим, что приведенный выше программный код представляет собой именно объявление класса и не содержит реализации методов. Об этом позже.

3.3. Инкапсуляция в действии. Спецификаторы доступа

Рассматривая ранее принцип инкапсуляции, мы указали, что он включает в себя два момента: объединение данных и операций в рамках одной синтаксической структуры и разграничение доступа. Первый момент виден из приведенного объявления класса `TSquare`. А как быть со вторым?

Уточним формат объявления класса:

```
type
TMyClass = class
private
   <поля>
```

```
    <методы>
  protected
    <поля>
    <методы>
  public
    <поля>
    <методы>
  published
    <поля>
    <методы>
  end;
```

Как мы теперь видим, объявление класса содержит 4 секции: **public**, **private**, **protected** и **published**.

Заметим, что в целях обратной совместимости поддерживается еще и пятая секция – **automated**, но в последних версиях языка ее использовать не рекомендуется, поэтому мы и не будем говорить о ней в рамках данной книги.

Ключевые слова **public**, **private**, **protected** и **published** называются *спецификаторами доступа* и задают правила, по которым компилятор решает, предоставить программисту в данном месте программы доступ к полю (методу) или нет. Правила выглядят следующим образом.

Таблица 10.2

Спецификаторы доступа

Спецификатор доступа	Правила
public	Открытые поля и методы. Доступны из любого места программы
private	Скрытые поля и методы. Доступны только в том модуле, в котором объявлен класс. Из другого модуля нельзя вызвать скрытый метод или изменить, прочитать скрытое поле (на самом деле если нельзя, но очень хочется, то можно, но мы не будем рассматривать в книге приемы, вредные для коллективного программирования, в том числе и этот)
protected	Защищенные поля и методы. Доступны в рамках модуля, в котором объявлен класс, а также в потомках данного класса (см. Наследование)
published	Опубликованные поля и методы. Используются

	для поддержки работы технологии визуального программирования (библиотека визуальных компонентов VCL). В данной книге не рассматривается

Таким образом, мы можем управлять доступом к полям и методам, размещая их в разные секции в соответствии со смыслом их использования. Наша общая рекомендация по размещению по секциям выглядит так: объявляйте в секции **private** те поля и методы, которые не предназначены для использования извне и относятся к деталям внутренней реализации вашего класса. Все остальное размещайте в секцию **public** (использование секции **protected** будет объяснено в разделе «Наследование и полиморфизм. Виртуальные методы и позднее связывание»).

Вернемся к примеру о квадрате. Ранее было замечено, что предоставление доступа к полю-признаку `Visible` потенциально опасно и может привести к рассогласованному состоянию предметной области и ее объектного описания в памяти компьютера. Таким образом, необходимо исключить изменение поля `Visible` вне класса. Скорректируем объявление:

```
type
 TSquare = class
 private
  {---------- Данные ------------}
  Visible: Boolean; { Признак видимости }
 public
  {---------- Данные ------------}
  x, y: Real; { Координаты левого нижнего угла }
  a: Real; { Сторона квадрата }

  {---------- Операции -----------}
  procedure Show; { Показать }
  procedure Hide; { Скрыть }
  function IsVisible: Boolean;{ Вернуть Visible }
  function S: Real; { Вычислить площадь }
  procedure MoveTo(dx, dy: Real); { Переместить на dx, dy }
end; { TSquare }
```

Теперь ситуация выглядит так: поле `Visible` нельзя изменить извне, но можно узнать его значение посредством вызова функции `IsVisible`.

3.4. Реализация методов класса

Посмотрев, как в первом приближении выглядит объявление класса, самое время узнать, где и как пишется реализация методов. Общий вид заголовка метода при его реализации выглядит так:

```
procedure <Имя класса>.<Имя метода>(<список параметров>);
```

или

```
function <Имя класса>.<Имя метода>(<список параметров>):
  <тип результата>;
```

Таким образом, имя метода «расширяется» именем класса. Действительно, многие классы должны иметь метод `Show`, как же компилятор разберется, чей именно `Show` мы сейчас реализуем? При помощи расширенного имени `<Имя класса>.<Имя метода>` – вот и первое проявление полиморфизма. Более не надо писать `ShowSquare`, достаточно просто `Show`.

Теперь реализуем несколько методов:

```
procedure TSquare.Show; { Показать квадрат }
begin
  if not Visible then
  begin
    Visible := True;
    Square(x, y, a);
  end;
end;

function TSquare.S: Real; { Площадь квадрата }
begin
  Result := a * a;
end;
```

Следующий момент связан с тем, где это все писать. Вообще, класс – просто тип данных, который объявляется в секции **type**, а методы класса – процедуры и функции, которые реализуются в том же модуле, как обычно. Однако чаще всего один или несколько классов выносятся в отдельный модуль. Поступим так и мы, создав модуль `UFigures` для описания классов для представления фигур.

```
{ ================================================ }
{ Классы – фигуры на плоскости }
Unit UFigures;
interface

type
  TSquare = class
  private
```

```
      {---------- Данные -------------}
      Visible: Boolean; { Признак видимости }
   public
      {---------- Данные -------------}
      x, y: Real; { Координаты левого нижнего угла }
      a: Real; { Сторона квадрата }

      {---------- Операции -----------}
      procedure Show; { Показать }
      procedure Hide; { Скрыть }
      function IsVisible: Boolean;{ Вернуть Visible }
      function S: Real; { Вычислить площадь }
      procedure MoveTo(dx, dy: Real); { Переместить на dx, dy }
   end; { TSquare }

implementation

procedure TSquare.Show; { Показать квадрат }
begin
   if not Visible then
   begin
      Visible := True;
      Square(x, y, a);
   end;
end;

function TSquare.S: Real; { Площадь квадрата }
begin
   Result := a * a;
end;

...
begin
end.
{ ================================================= }
```

3.5. Свойства

Еще один важный элемент объектной модели языка Object Pascal – *свойства* (в англоязычной литературе – *properties*). Заметим, что

свойства – существенное новшество языка Pascal по сравнению с языком программирования C++, в котором нет ничего подобного (хотя, разумеется, есть много других достоинств). Концепция свойств оказалась настолько популярной, что проникла в новый язык программирования, разработанный специально для платформы .Net – C#. Рассмотрим смысл и синтаксис данного понятия на примере.

Заметим, что границы нашего квадрата при рисовании должны отображаться некоторым цветом. Для отражения этого факта в программном коде добавим в объявление типа `TSquare` открытое поле `Color`.

```
type
  TSquare = class
  private
    {---------- Данные -------------}
    Visible: Boolean; { Признак видимости }
  public
    {---------- Данные -------------}
    x, y: Real; { Координаты левого нижнего угла }
    a: Real; { Сторона квадрата }
    Color: Cardinal; { Цвет границ }
    {---------- Операции -----------}
    ...
end; { TSquare }
```

Подумаем, как все это будет работать. Допустим, что воображаемая функция `Square` рисует квадрат на плоскости, получая координаты его левого нижнего угла и цвет границ. Тогда реализация метода `Show` могла бы выглядеть так:

```
procedure TSquare.Show; { Показать квадрат }
begin
  if not Visible then
  begin
    Visible := True;
    Square(x, y, a, Color);
  end;
end;
```

Суть проблемы состоит в следующем: что если пользователь класса (коллега-программист) изменил значение `Color` на другое? Было бы логично, чтобы квадрат автоматически был перерисован другим цветом. Иначе придется писать в программе так:

```
MySquare.Color := NewColor;
MySquare.Hide;
MySquare.Show;
```

Слишком много для такой тривиальной операции, не правда ли? К тому же нет гарантии, что никто не забудет написать все 3 строки, а то получится как в известном анекдоте, где один копал ямы, а другой их сразу же закапывал. Исследование показало, что был еще и третий, он должен был сажать деревья, но не пришел.

Эта проблема решается посредством использования свойств. Синтаксис объявления свойства выглядит так:

```
property <Имя свойства>: <Тип данных>
  read <поле или метод> write <поле или метод>;
```

В отличие от поля свойство не хранит данных! Это просто механизм для доступа. Наиболее популярный вариант использования – создание скрытых полей и открытых свойств для доступа к ним.

В нашем примере мы можем поступить так:

```
TSquare = class
private
   {---------- Данные ------------}
   FVisible: Boolean; { Признак видимости }
   FColor: Cardinal; { Цвет границ }
   Fx, Fy: Real; { Координаты левого нижнего угла }
   Fa: Real; { Сторона квадрата }

   procedure SetColor(const Value: Cardinal);
public
   {---------- Свойства ------------}
   property x: Real read Fx write Fx;
   property y: Real read Fy write Fy;
   property a: Real read Fa write Fa;

   property Color: Cardinal read FColor write SetColor;
   property Visible: Boolean read FVisible;
   {---------- Операции ------------}
   procedure Show; { Показать }
   procedure Hide; { Скрыть }
   function S: Real; { Вычислить площадь }
   procedure MoveTo (dx,dy:Real);{Переместить на dx, dy}
end; { TSquare }
```

Посмотрим, что мы сделали. Во-первых, мы скрыли все данные. Во-вторых, мы создали свойства `x`, `y`, `a`, которые после ключевых слов **read** и **write** содержат названия полей `Fx`, `Fy`, `Fa`. Это означает, что запись `MyObject.x := 1` или `q := MyObject.x` будет означать доступ к полю `Fx`. На этих примерах особой выгоды не видно. Обратимся к более содержательным случаям. Так, для свойства `Visible` можно заметить

отсутствие секции **write**. Это означает, что хотя при чтении свойства мы берем значение из поля FVisible, запись является запрещенной. Метод IsVisible стал более не нужным и был исключен из объявления класса (предложение «**if** Visible **then**» теперь полностью его заменяет).

Отдельно рассмотрим свойство Color. В секции **read** мы имеем ссылку на поле FColor, а в секции **write** – упоминание метода SetColor. Метод SetColor получит управление при попытке изменить цвет посредством свойства Color. Параметр метода – новое значение цвета. Реализуем процедуру SetColor так:

```
procedure TSquare.SetColor(const Value: Cardinal);
begin
   Hide;
   FColor := Value;
   Show;
end;
```

Теперь при попытке изменить цвет моментально будет перерисован квадрат, что и требовалось получить.

Итак, мы можем создавать свойства, делать их свойствами только для чтения (часто применяется) или только для записи (крайне редко), связывать их с полями напрямую или посредством специальных методов.

Используя механизм свойств, мы можем предоставить доступ на запись в свойстве Visible, предусмотрев специальный метод, который будет отображать фигуру, вызывая ее метод Show.

```
TSquare = class
private
  {---------- Данные -------------}
  FVisible: Boolean; { Признак видимости }
  ...
  procedure SetVisible(const Value: Boolean);
public
  {---------- Свойства -------------}
  ...
  property Visible: Boolean read FVisible
    write SetVisible default False;
  {---------- Операции -------------}
  ...
end; { TSquare }

procedure TSquare.SetVisible(const Value: Boolean);
begin
  if FVisible <> Value then
  begin
    FVisible := Value;
```

```
    if FVisible then
      Show
    else
      Hide;
  end;
end;
```

Обратим внимание на ключевое слово **default**, которое в данном случае позволяет задать значение по умолчанию. Теперь свойство при создании объекта будет установлено в **False**.

Следующий важный тип свойства – индексированное или массивное свойство.

Пусть некоторый класс содержит среди данных массив.

```
TVector = class
private
  FData: array[1..100] of Real;
  function GetData(Index: Integer): Real;
  procedure SetData(Index: Integer; MyData: Real);
  ...
public
property Data[Index: Integer]: Real read GetData
   write SetData;
  ...
end; { TVector }

function TVector.GetData(Index: Integer): Real;
begin
  Result := FData[Index];
end;

procedure TVector.SetData(Index: Integer; MyData: Real);
begin
  FData[Index] := MyData;
end;
```

Теперь мы имеем возможность создать объект типа `TVector` (о том, как это делается, чуть позднее) и написать так: `v.Data[5] := 3.14;`

При этом вызовется метод `SetData` с параметрами `Index = 5, MyData = 3.14`.

Заметим, что методы могут выполнять некоторые дополнительные действия. В принципе, массивное свойство – просто интерфейсный элемент, оно может в действительности не быть связано с реальным массивом, но чаще всего дело обстоит так, как в рассмотренном примере.

Еще одно важное дополнение к рассмотренной функциональности состоит в том, что специально для массивных свойств ключевое слово

default не только может быть применено, но и работает совершенно по-другому. Добавим ключевое слово **default** так:

```
property Data[Index: Integer]: Real read GetData
  write SetData; default;
```

Теперь свойство Data становится свойством по умолчанию, и мы можем написать `v[5] := 3.14` вместо `v[5].Data := 3.14`.

Подводя итог, можно заключить, что свойства – мощный и удобный инструмент для разработки полноценных объектно-ориентированных программ.

3.6. Специальные виды методов. Конструкторы. Перегрузка конструкторов

Познакомившись с тем, как объявляются и реализуются методы в классах на языке Object Pascal, самое время изучить некоторые специальные виды методов. Сразу оговоримся, что их немало. Начнем с самого главного, с методов, без которых в реальности не существует ни один класс, – с *конструкторов* и *деструкторов*.

Конструктор – специальный метод, который вызывается для создания переменных объектного типа, т.е. в нашей терминологии – объектов. В чем основное предназначение данного метода, что отличает его от других? Прежде всего, вкладываемая в него смысловая нагрузка. Так, основные задачи конструктора можно сформулировать следующим образом:

- выделение памяти для хранения объекта;
- инициализация полей объекта начальными значениями;
- выделение памяти для специфических полей объекта (полей-объектов, полей указателей).

Все эти функции являются очень важными и заслуживают того, чтобы ради них ввели специальный вид метода.

Следующий вопрос состоит в том, как объявить конструктор. С точки зрения языка Object Pascal, конструктор есть подпрограмма, у которой вместо ключевого слова **procedure** присутствует ключевое слово **constructor**. Так, синтаксис объявления конструктора для класса `TSquare` может выглядеть так:

```
TSquare = class
private
   {---------- Данные ------------}
   FVisible: Boolean; { Признак видимости }
   FColor: Cardinal; { Цвет границ }
   Fx, Fy: Real;{ Координаты левого нижнего угла }
   Fa: Real; { Сторона квадрата }
```

```
    procedure SetColor(const Value: Cardinal);
  public
  ...
    constructor Create;
  end; { TSquare }
```

У вас наверняка возник ряд вопросов. Сейчас мы сформулируем основные принципы, касающиеся объявления конструкторов, которые должны существенно прояснить ситуацию.

Для начала обсудим вопрос об именовании конструкторов. Если у вас в классе объявлен один конструктор, настоятельно рекомендуется называть его `Create`. Конечно, вы можете пренебречь этой рекомендацией, и все будет работать, но… Стиль есть стиль. С большими шансами коллеги не поймут, если в классе `TMySuperGreatClass` конструктор будет называться `MySuperGreatClassConstr`. Это непонимание выльется в то, что они откажутся писать этот ужас в своем коде. Вообще, здравый смысл подсказывает, что изобретение велосипедов хорошо лишь в тех областях, где а) нет велосипедов и б) они вообще кому-то там нужны. Здесь нет ни а), ни б), поэтому мы все конструкторы по возможности будем именовать `Create`.

Ознакомившись для начала со стилистическим моментом написания программы, перейдем к существенно более сложным смысловым вещам. Так, рассмотрим, чт.∴ за операции выполняет приведенный выше конструктор, если мы создадим для него «пустую» реализацию.

```
constructor TSquare.Create;
begin
end;
```

Оказывается, данная реализация при запуске выполняет целую тучу разных действий. Кто об этом позаботился, если мы не написали ни строки кода? Конечно, наш друг компилятор. Такая реализация обеспечивает следующие действия, выполняемые *автоматически* любым конструктором:

- Выделение памяти для полей объекта. В данном случае выделится память, необходимая для хранения полей `FVisible`, `FColor`, `Fx`, `Fy`, `Fa`.
- Инициализация полей нулевыми значениями. Стоп! В этом месте необходимо задержаться. Документация по системам программирования Borland Delphi действительно утверждает, что обнуление происходит. Это означает, что обычные поля инициализируются нулем, поля-указатели – значением **`nil`**, поля-объекты – нулевыми ссылками. Но! Ни в коем случае не рекомендуем вам ставить работоспособность программы в зависимость

от того, обнулил что-то компилятор сам или нет. Если для программы важно гарантированное обнуление, выполняйте его самостоятельно. Тем самым вы страхуете себя от того, что ваша программа перестанет работать при переходе к другому компилятору.

- Создание в памяти довольно сложной структуры. Дело в том, что объект – это не только совокупность полей, но и информация о типе. Данный сложный вопрос будет рассмотрен нами чуть позже в разделе «Внутренняя структура объекта. Методы класса. Динамический контроль типов, операторы IS и AS».

Все это означает, что для создания полей объекта нам не нужно писать каких-то сложных конструкций, компилятор все делает сам. А бывает ли необходимость все-таки что-то написать или всегда нужно оставлять пустую реализацию? Конечно, бывает. Сейчас мы рассмотрим несколько примеров. Первый случай, когда необходимо вмешаться в создание объектов класса, связан с желанием параметризовать процесс создания, т.е. позволить создавать объекты с инициализацией полей ненулевыми значениями. Так, для класса `TSquare` это может выглядеть следующим образом:

```
TSquare = class
private
   {---------- Данные -------------}
   FVisible: Boolean; { Признак видимости }
   FColor: Cardinal; { Цвет границ }
   Fx, Fy: Real; { Координаты левого нижнего угла }
   Fa: Real; { Сторона квадрата }

   procedure SetColor(const Value: Cardinal);
public
   ...
   constructor Create(ax, ay, aa: Real;
      aColor: Cardinal; aVisible: Boolean);
end; { TSquare }

constructor TSquare.Create(ax, ay, aa: Real;
  aColor: Cardinal; aVisible: Boolean);
begin
   Fx := ax;
   Fy := ay;
   FColor := aColor;
   FVisible := False;
```

```
      if aVisible then
        Show;
   end;
```

Данный конструктор выполняет инициализацию полей переданными параметрами.

Второй случай вмешательства в действия компилятора – выделение памяти для полей объектов и полей указателей. Пусть мы создаем класс – динамический массив действительных чисел. Это может выглядеть так:

```
{ ================================================== }
{ Пример 10.2 }
{ Класс - динамический массив }
TArrDouble = class
private
   FDim: Integer;
   FValues: array of Double;

   procedure SetDim(const Value: Integer);
   function GetRValue(Index: Integer): Double;
   procedure SetRValue(Index: Integer; const Value: Double);
public
   { Значение }
   property Dim: Integer read FDim write SetDim;
   property Values[Index: Integer]: Double read GetRValue
     write SetRValue; default;

   constructor Create(aDim: Integer);
end; { TArrDouble }

constructor TArrDouble.Create(aDim: Integer);
begin
   FDim := aDim;
   SetLength(FValues, FDim);
end;

function TArrDouble.GetRValue(Index: Integer): Double;
begin
   Result := FValues[Index];
end;

procedure TArrDouble.SetRValue(Index: Integer;
  const Value: Double);
begin
   FValues[Index] := Value;
end;
```

```
procedure TArrDouble.SetDim(const Value: Integer);
begin
   FDim := Value;
   SetLength(FValues, FDim);
end;
{ ============================================== }
```

В представленном коде конструктор `Create` выделяет память под внутренний массив класса.

Следующий вопрос – а может ли в классе быть более одного конструктора? Ответ положительный. Несколько видоизменим предыдущий пример. Предоставим пользователю класса три способа создания массива – создание «пустого» массива, создание массива заданного размера и создание массива, данные которого хранятся в базе данных (БД).

Считаем, что схема хранения в БД выглядит так (рис. 10.11).

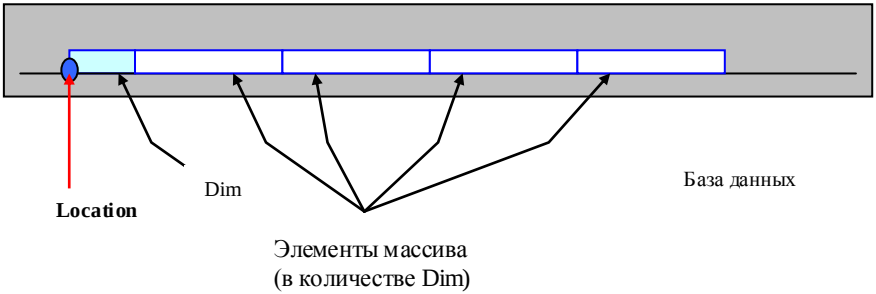

Рис. 10.11. Схема хранения массива в базе данных

```
{ ============================================== }
{ Класс – динамический массив. Конструкторы }
TArrDouble = class
private
   FDim: Integer;
   FValues: array of Double;

   procedure SetDim(const Value: Integer);
   function GetRValue(Index: Integer): Double;
   procedure SetRValue(Index: Integer; const Value: Double);
public
   { Значение }
   property Dim: Integer read FDim write SetDim;
   property Values[Index: Integer]: Double read GetRValue
     write SetRValue; default;
```

```
   { Создание пустого массива }
   constructor Create;
   { Создание массива из aDim элементов }
   constructor Init(aDim: Integer);
   { Создание массива с загрузкой данных из базы данных }
   { Используем гипотетический тип данных TDBLocation - }
   { местонахождение в БД }
   constructor Load(Location: TDBLocation);
end; { TArrDouble }
constructor TArrDouble.Create;
begin
   FDim := 0;
end;
constructor TArrDouble.Init(aDim: Integer);
begin
   FDim := ADim;
   SetLength(FValues, FDim);
end;
constructor TArrDouble.Load(Location: TDBLocation);
var
   i: Integer;
begin
   // Некоторый код загрузки массива из базы данных
   // Схема загрузки:
   // Читаем размерность (гипотетический метод, считаем, что
   // объект «знает», где он хранится в базе данных
   ReadFromDB(Location, 0, FDim);
   // Выделяем память под массив
   SetLength(FValues, FDim);
   // Читаем массив из базы данных
   for i := 1 to FDim do
      ReadFromDB(Location, i, FValues[i]);
end;
{ ================================================== }
```

Теперь мы имеем сразу три конструктора, позволяющих полностью контролировать процесс создания объектов и выполнять его в соответствии с необходимостью либо методом `Create` (создание пустого массива, потом при необходимости установка размера `Dim = <значение>` и дальнейшая работа), либо созданием массива заданной длины методом `Init`, либо загрузкой массива из базы данных методом `Load`. Осталось разобраться с нашей собственной рекомендацией об именовании. Ранее мы говорили, что лучше все конструкторы называть `Create`. В данном случае в классе их несколько, как же мы можем назвать несколько подпрограмм одинаково? Конечно, можем! Вспоминаем материал главы 7 о перегрузке

подпрограмм. Необходимо сопроводить все конструкторы ключевым словом **overload**.

```
TArrDouble = class
private
  FDim: Integer;
  FValues: array of Double;
  procedure SetDim(const Value: Integer);
  function GetRValue(Index: Integer): Double;
  procedure SetRValue(Index: Integer;
    const Value: Double);
public
  { Значение }
  property Dim: Integer read FDim write SetDim;
  property Values[Index: Integer]: Double read GetRValue
    write SetRValue; default;

  constructor Create; overload;
  constructor Create(aDim: Integer); overload;
  constructor Create(Location: TDBLocation); overload;
end; { TArrDouble }
```

Теперь мы имеем три *перегруженных* конструктора. Напомним, что факт перегрузки означает, что мы используем одно и то же имя для разных методов и эти методы должны быть различимы по списку параметров. Так, нам не удастся создать два конструктора с такими параметрами:

```
constructor Create(a: Integer);
constructor Create(b: Integer);
```

Компилятор в этом случае инициирует ошибку.

Для того чтобы завершить данный раздел, рассмотрим еще один пример.

```
{ ================================================= }
{ Обманем компилятор! }
// Любителям обмана компилятора посвящается
// Компилятор Object Pascal, среда Borland Delphi 7.0
TExample = class
public
   fd: Integer;
   constructor Create(a: Byte); overload;
   constructor Create(a: Integer); overload;
   procedure test;
end; { TExample }

constructor TExample.Create(a: Byte);
```

```
begin
   fd := a;
end;

constructor TExample.Create(a: Integer);
begin
   fd := a;
end;
```

Интересный вопрос – как компилятор собирается разбираться, какой именно конструктор вызвать, если типы `Integer` и `Byte` совместимы? Проведем эксперимент.

```
   // вызываем конструктор с параметром-переменной
   var
      TExample e1, e2, e3;
      a: Integer;
      b: Byte;
      ...

   e1 := TExample.Create(a);
   // Разобрался, вызвал второй конструктор
   e2 := TExample.Create(b);
   // Разобрался, вызвал первый конструктор
   e3 := TExample.Create(1);
   // Невозможно разобраться. Но компилятор вызвал
   // первый конструктор
{ ================================================== }
```

Если с первыми двумя случаями все ясно – компилятор «посмотрел» на тип переменной и сопоставил его с типом формального параметра в конструкторе `Create`, то третий случай вызывает некоторое недоумение. Каким образом компилятор решил, что 1 относится к типу `Byte`, а не к типу `Integer`? И тот ли это вариант, которого ждал программист?

Вывод: не нужно писать такой код, в котором невозможно разобраться. Хорошие программы – это не те программы, в которых не могут разобраться ваши коллеги и (или) компилятор. Хорошие программы – это программы, которые работают и работают понятным образом.

3.7. Специальные виды методов. Деструкторы

В предыдущем разделе мы научились управлять созданием объектов. А как управлять их уничтожением? Очень просто, для этого предназначен специальный тип метода – *деструктор*. Для объявления деструкторов используется ключевое слово **destructor**. Задачи деструктора – выполнить некоторые завершающие действия и освободить память. Снабдим деструкторами классы из рассмотренных ранее примеров.

```
TArrDouble = class
private
   FDim: Integer;
   FValues: array of Double;

   procedure SetDim(const Value: Integer);
   function GetRValue(Index: Integer): Double;
   procedure SetRValue(Index: Integer;
     const Value: Double);
public
   { Значение }
   property Dim: Integer read FDim write SetDim;
   property Values[Index: Integer]: Double read GetRValue
     write SetRValue; default;

   constructor Create; overload;
   constructor Create(aDim: Integer); overload;
   constructor Create(Location: TDBLocation); overload;
   destructor Destroy;
end; { TArrDouble }
destructor TArrDouble.Destroy;
begin
   SetLength(FValues, 0);
end;
```

Как видно из этого примера, память, выделенная в конструкторе, освобождается в деструкторе. Логично, не правда ли?

Заметим, что память для «обычных» полей, таких, как, например, `FDim`, освобождается автоматически, не требуя написания в деструкторе специальных инструкций.

Теперь посмотрим на проблему завершающих действий несколько в ином ракурсе. Вернемся к классу `TSquare`.

```
TSquare = class
private
   {---------- Данные ------------}
   FVisible: Boolean;  { Признак видимости              }
   FColor: Cardinal;   { Цвет границ                    }
   Fx, Fy: Real;       { Координаты левого нижнего угла }
   Fa: Real;           { Сторона квадрата               }

   procedure SetColor(const Value: Cardinal);
public
   ...
   constructor Create(ax, ay, aa: Real; aColor: Cardinal;
     aVisible: Boolean);
   destructor Destroy;
end; { TSquare }

constructor TSquare.Destroy;
begin
   Hide;
end;
```

Посмотрим внимательно на этот несложный пример. Здесь не требуется освобождать в деструкторе выделенную в конструкторе память, но требуется убрать фигуру с координатной плоскости, иначе объект исчезнет, а на экране останется видимым, странно, не правда ли?

В заключение еще одна рекомендация по стилю именования. Вероятно, вы заметили, что деструкторы в языке Object Pascal принято называть `Destroy`.

3.8. Объявление, создание, удаление объектов. Ссылочная модель объекта. Присваивание и копирование

Узнав уже довольно много о создании классов, самое время научиться их использовать. Именно этим мы сейчас и займемся. Для того чтобы понять, как использовать созданные классы, вспомним, что класс представляет собой тип данных, а значит, его использование непосредственно связано с созданием переменных. Как мы уже знаем, переменные класса называются объектами. Как же происходит работа с этими переменными?

Фундаментальный принцип программирования в языке Pascal выглядит так: *прежде чем переменная может быть использована, она должна быть объявлена.*

Объявление объектов ничем не отличается от объявления «обычных» переменных.

```
type
   TMyClass = class
   public
      ...
      constructor Create;
      destructor Destroy;
   end; { TMyClass }
...
var
   temp: TMyClass;
```

Важный для понимания момент – что реально происходит, когда компилятор встречает объявление переменной объектного типа? Какие инструкции он вставляет в объектный код?

Дело в том, что в языке Object Pascal *все переменные объектного типа – ссылки на области оперативной памяти*. Что это значит? Это значит, что в приведенном выше примере переменная temp – это не сам объект, а ссылка на объект (проще говоря, адрес некоторой области оперативной памяти). Важно помнить, что изначально эта ссылка равна нулю, и попытка использования объекта непосредственно после объявления в блоке **var** приведет к ошибке времени исполнения программы.

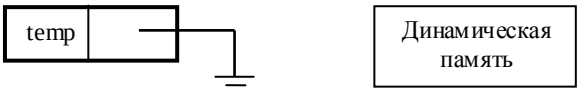

Рис. 10.12. Объявление объекта порождает нулевую ссылку

Сначала объект необходимо создать. Как вы помните, для создания объектов служит специальный метод, называемый конструктором.

Создадим объект – динамический массив (см. пример – класс `TArrDouble`).

```
var
   arr: TArrDouble;
   ...
begin
   arr := TArrDouble.Create(10);
end.
```

В представленном коде происходит создание объекта `arr` типа `TArrDouble` при помощи конструктора `Create` с параметром `aDim = 10`. При этом выполняется выделение необходимой памяти. Теперь допускается использование объекта.

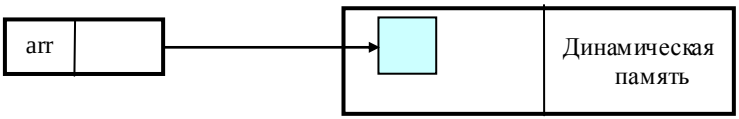

Рис. 10.13. Создание объекта посредством вызова конструктора

Доступ к полям и методам осуществляется через точку после имени объекта.

Заметим, что создание объекта синтаксически выглядит очень хитро. Метод `Create` вызывается не через переменную `arr` (как известно, это породит ошибку из-за того, что arr инициализирована нулевым адресом), а через имя класса `TArrDouble`. В данном случае мы видим пример так называемого *метода класса*, который вызывается не для объекта, а для класса. Подробнее о других случаях разработки и использования методов класса можно прочитать в документации.

Уничтожение созданного объекта производится посредством вызова деструктора. Так, мы можем написать:

```
var
   arr: TArrDouble;
...
begin
   arr := TArrDouble.Create(10);
   ...
   arr.Destroy;
end.
```

Освобождение памяти – важная процедура, о которой нельзя забывать, поэтому вызывать деструкторы, когда объект далее не будет использоваться, необходимо. Вопрос в том, как это делать.

Посмотрим на приведенный выше код «под микроскопом». В коде нет ошибки. Однако он небезопасен. Попробуйте в порядке эксперимента (не вздумайте делать это в реальных программах) вызвать деструктор для одного и того же объекта два раза. Или вызвать деструктор для объекта, который не был создан. Как вы увидите, это приведет к краху вашей программы. Как этого избежать?

Первый способ – проверять перед вызовом деструктора, инициализирована ли ссылка на объект.

```
var
   arr: TArrDouble;
...
begin
   arr := TArrDouble.Create(10);
   ...
   arr.Destroy;
   if arr <> nil then
      arr.Destroy;
end.
```

Неприятности связаны с тем, что вызов деструктора освобождает память, но не обнуляет ссылку. Так, следующий код приведет к ошибке:

```
var
   arr: TArrDouble;
...
begin
   arr := TArrDouble.Create(10);
   ...
   arr.Destroy;
   if arr <> nil then
      arr.Destroy;
   // Некоторые действия
   if arr <> nil then
      arr.Destroy; // Ошибка
   // или arr.<Обращение к любому члену класса> -
   // Ошибка
end.
```

С этим можно справиться так:

```
if arr <> nil then
begin
   arr.Destroy;
   arr := nil;
end;
```

Второй способ связан не столько со стандартом языка, сколько непосредственно с реализацией этого стандарта от фирмы Borland в рамках системы программирования Borland Delphi. Эта реализация предоставляет для всех классов метод `Free`, который выполняет проверку на **nil** и

только потом вызывает деструктор, решая часть проблемы. Так, программируя на Delphi, признак плохого стиля вызывать деструктор, вместо этого необходимо писать так:

```
arr.Free;
arr = nil; // Если вы опасаетесь за дальнейшее
```

Заметим, правда, что для этого вы должны создавать деструкторы виртуальными, используя в классах директиву **override** после имени деструктора (подробнее об этом в разделе «Наследование и полиморфизм. Виртуальные методы и позднее связывание»).

Рассмотрим теперь часто встречающуюся ситуацию с присваиванием и копированием объектов. Многие из вас, начав программировать с использованием ООП на Object Pascal, рано или поздно столкнутся со следующим примером:

```
var
   A, B: TArrDouble;
...
begin
   arr := TArrDouble.Create(10);
   ...
   B := A;
   ...
end.
```

Попробуем откомпилировать этот код – все в порядке, компиляция прошла успешно.

Вопрос: как это работает?

В голову приходят два варианта того, как это в принципе *могло бы* работать. Вариант первый состоит в том, что создается новый объект `B` и все содержимое объекта `A` копируется в `B`. Простая проверка показывает, что это не так.

```
var
   A, B: TArrDouble;
begin
   A := TArrDouble.Create(10);
   A[1] := 14; A[2] := 15;
   B := A;
   B[1] := 16; B[2] := 17;
   Writeln(A[1]:5:2, ' ', A[2]:5:2, ' ',
   B[1]:5:2, ' ', B[2]:5:2);
   Readln;
```

```
  A.Free;
end.
```

Запустив эту несложную программу, мы увидим на экране следующее:

```
16.00 17.00 16.00 17.00
```

Тем самым наше предположение не оправдалось. Оказывается, мы имеем дело со вторым возможным вариантом работы такого присваивания. Компилятор не выполнил создания нового объекта и копирования данных. Он настроил еще одну ссылку на ту же самую область памяти. Здесь необходимо быть очень внимательным. Дело в том, что после удаления объекта A ни в коем случае нельзя удалять объект B, ибо это одно и то же!

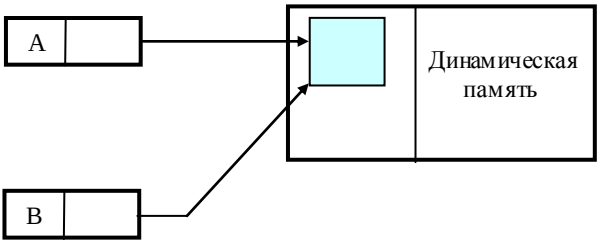

Рис. 10.14. Присваивание для объектов – присваивание адресов

А как же осуществить копирование?

Продемонстрируем на следующем примере возможный вариант организации копирования объектов на уровне класса. Попробуем понять, как это сделать. Очевидно, что для копирования данных для начала необходимо иметь объект, в который будет производиться копирование. Для этого его кто-то должен создать. Мы предлагаем создать в классе специальный конструктор с одним параметром – объектом того же типа. Задача этого конструктора – создать новый объект и скопировать в него все данные из образца, переданного в качестве параметра.

```
{ ================================================ }
{ Конструктор копирования }
TArrDouble = class
private
   FDim: Integer;
   FValues: array of Double;

   procedure SetDim(const Value: Integer);
```

```
   function GetRValue(Index: Integer): Double;
   procedure SetRValue(Index: Integer; const Value: Double);
public
   { Значение }
   property Dim: Integer read FDim write SetDim;
   property Values[Index: Integer]: Double read GetRValue
     write SetRValue; default;

   constructor Create; overload;
   constructor Create(aDim: Integer); overload;
   constructor Create(Location: TDBLocation); overload;

   { Конструктор копирования }
   constructor Create(aSource: TArrDouble); overload;
   destructor Destroy;
end; { TArrDouble }
constructor TArrDouble.Create(aSource: TArrDouble);
   overload;
var
   i: Integer;
begin
   SetDim(aSource.Dim);
   for i := 1 to Dim do
     FValues[i] := aSource[i];
end;
{ ================================================= }
```

Теперь напишем в головной программе:

```
begin
   A := TArrDouble.Create(10);
   A[1] := 14; A[2] := 15;
   C := TArrDouble.Create(A);
   C[1] := 16;
   Writeln(A[1]:5:2,'',A[2]:5:2,'',C[1]:5:2,'',
     C[2]:5:2);

   Readln;

   A.Free;
   C.Free;
end.
```

Запускаем… Работает!

На экране следующий результат:

```
14.00 15.00 16.00 15.00
```

Заметим, что в отличие от C++, Object Pascal не содержит *встроенных* средств для обеспечения операции копирования. В результате мы можем создавать в классах так называемый «конструктор копирования», действуя указанным выше способом. Однако в библиотеке VCL (библиотека классов, основа визуального программирования в Delphi) уже предприняты некоторые меры по автоматизации копирования. При рассмотрении наследования мы еще упомянем тот факт, что все классы в языке Object Pascal, в том числе и ваши собственные, для которых не указан предок, будут выведены из класса `TObject`. Сам по себе `TObject` содержит ряд полезных свойств и методов, но никак не помогает в вопросе копирования. А вот один из его потомков `TPersistent` имеет виртуальный метод `Assign`, предназначенный как раз для решения этой проблемы. Таким образом, выводя класс из `TPersistent`, вы можете перекрыть в вашем классе метод `Assign`, реализовав в нем копирование данных, что обеспечит копирование «в стиле библиотеки VCL».

3.9. Способы коммуникации между объектом и методами. Раннее связывание. Указатель Self

Поговорим о некоторых моментах внутреннего устройства и работы ООП в языке программирования Object Pascal. В данном разделе мы рассмотрим два вопроса – как объект «добирается» до своих методов и как методы класса получают доступ к данным объекта.

Суть первого вопроса состоит в следующем. Как было нами выяснено ранее, в отличие от обычных процедур и функций, методы вызываются не сами по себе, а для конкретных объектов. Так, мы можем написать следующий код:

```
var
   A: TArrDouble;
begin
   A := TArrDouble.Create;
   A.SetDim(20); // Вызов метода SetDim для объекта A
end.
```

В рассмотренном примере производится вызов метода `SetDim` для объекта `A`. Как компилятор разбирается, где этот метод и какой адрес нужно подставить? Для тех методов, которые мы умеем писать на настоящий момент, проблема решается на удивление просто. На этапе сборки программы подставляется адрес той области памяти, в которой находится скомпилированный код метода `SetDim` класса `TArrDouble`. Схема работы выглядит примерно так: объект `A` объявлен как переменная типа `TArrDouble`. Ищем среди методов класса `TArrDouble` метод `SetDim`. Подставляем его адрес.

В данном случае речь идет о так называемом *раннем связывании* (*early binding*), в рамках которого адрес вызываемого метода известен на этапе компиляции и сборки программы. Вы можете задаться вопросом – бывает ли по-другому? Бывает, но об этом при рассмотрении *виртуальных методов*.

Быстро разобравшись с первым вопросом, перейдем теперь ко второму. Как получается, что метод разбирается, какие именно данные использовать в теле метода? Рассмотрим пример.

```
procedure TArrDouble.SetDim(const Value: Integer);
begin
   FDim := Value;
   SetLength(FValues, FDim);
end;
```

Метод `SetDim` использует в своем теле имена `FDim`, `FValues`, `Value`. Если `Value` – параметр метода и в его использовании нет ничего необычного, то что такое `FValues` и `FDim`?

Ясно, что это поля класса `TArrDouble`. Действительно, `SetDim` – не просто отдельная функция, а метод класса `TArrDouble`, следовательно, использование внутри метода полей класса легально. Все хорошо, но как отрабатывает следующий код?

```
A.SetDim(20); // A – объект типа TArrDouble
```

Интуитивно понятно, что метод `SetDim` должен работать с данными того объекта, который его вызывает. В приведенном примере должны использоваться поля `FDim` и `FValues` объекта `A`.

А как достигают этого результата? На самом деле метод класса всегда получает один «неявный» параметр – ссылку на объект, который вызвал метод. Для приведенного примера метод `SetDim` в реальности получает два параметра – ссылку на объект `A` и значение `Value`. Этот неявный параметр имеет специальный идентификатор `Self`. Приведем условный код метода `SetDim` (такой, каким видит его компилятор):

```
procedure TArrDouble.SetDim(TArrDouble Self;
  const Value: Integer);
begin
   Self.FDim := Value;
   SetLength(Self.FValues, Self.FDim);
end;
```

Не правда ли, утомительно многократно писать префикс `Self`? Object Pascal избавляет нас от этой необходимости.

Бывают методы, которые не получают `Self`. Это так называемые «методы класса», о которых можно прочитать в разделе «Внутренняя структура объекта. Методы класса. Динамический контроль типов, операторы IS и AS».

Тем не менее в некоторых случаях указатель `Self` необходимо использовать явно. Как правило, речь идет о возврате объекта из метода.

```
function TSomeClass.SomeMethod(p: Integer): TSomeClass;
begin
   Fp := Fp + p;
   Result := Self;
end;
```

3.10. Пример разработки класса «рациональная дробь»

Узнав уже довольно много о разработке классов на языке Object Pascal, попробуем написать пример класса для представления и осуществления операций над рациональными дробями.

Посмотрим на последовательность действий, которую необходимо проделать для разработки полноценного класса. Начнем с анализа предметной области.

Как известно, рациональная дробь есть $R = \frac{p}{q}$, где p, q – целые числа. Имеет смысл говорить о представлении в виде класса несократимых дробей, т.е. дробей, для которых НОД(p, q) = 1. Дело в том, что в машинной реализации сокращение дроби выглядит очень важной процедурой, которая страхует нас от элементарного переполнения в результате выполнения операций над дробями. Как автоматически сокращать дробь? Пока оставим этот вопрос и перейдем к проектированию данных нового класса `TRational`.

Проектирование данных

Будем моделировать числитель и знаменатель дроби целыми числами типа `Integer`. Рассмотрим вопрос о помещении данных в открытую или закрытую часть класса. Можно ли открыть данные? Здравый смысл подсказывает, что нет. Действительно, открыв данные, мы утрачиваем контроль за тем, что дробь в любой момент работы программы будет

несократимой. Итак, мы скроем данные и предоставим свойства для доступа к ним, кроме того, при изменении p или q будем сокращать дробь.

Проектирование операций

Какие методы нам понадобятся?

- Конструктор: без параметров. Создает «нулевую дробь» – числитель равен 0, знаменатель равен 1.
- Конструктор: параметры метода – p и q. Создает и сокращает дробь.
- Конструктор копирования: параметр – объект типа `TRational`.
- Деструктор.
- Различные арифметические операции: сложение, вычитание, умножение, деление.
- Вывод на консоль.
- Метод сокращения дроби.

Учитывая, что алгоритмы осуществления арифметических операций хорошо известны, остановимся отдельно на алгоритме сокращения дроби.

«*Единица есть то, через что каждое из существующих считается единым. Число же – множество, составленное из единиц*»[1], – так говорил великий Евклид. Будем использовать для сокращения дроби так называемый *алгоритм Евклида* нахождения наибольшего общего делителя.

Пусть $a \le b$. Тогда согласно алгоритму

$$\text{НОД}\ (a, b) = \text{НОД}\ (b - a, a).$$

Будем считать, что НОД (0, 0) = 0, НОД (0, b) = b.

Известно, что вычитание в данном алгоритме может быть заменено на деление с остатком, что существенно ускоряет работу алгоритма.

$$\text{НОД}\ (a, b) = \text{НОД}\ (b \bmod a, a).$$

Во втором случае сложность алгоритма исчисляется в виде $O(\log B)$ (проверьте этот факт самостоятельно. Указание: показать, что на каждом шаге работы алгоритма размер задачи – число B – уменьшается по крайней мере в 2 раза).

Рассмотрев все необходимые вопросы, приведем программную реализацию. Выполним ее в виде модуля `URational`.

В примере нам встретятся незнакомые слова **`inherited`** и **`override`**. Догадаться об их назначении на самом деле не сложно, подробно мы познакомимся с ними ниже при рассмотрении наследования.

[1] Евклид. «Начала».

```
{ ============================================== }
{ Пример 10.3 }
{ Класс Рациональная дробь }
Unit URational;
interface
type
   { Рациональная дробь }
   TRational = class
   private
      { Числитель и знаменатель }
      Fp, Fq: Integer;

      { Поиск НОД(Fp, Fq) }
      function Euclid: Integer;
      { Сокращение дроби }
      procedure Reduce;
      { Установка Fp }
      procedure SetP(const Value: Integer);
      { Установка Fq }
      procedure SetQ(const Value: Integer);
   public
      { Числитель }
      property p: Integer read Fp write SetP;
      { Знаменатель }
      property q: Integer read Fq write SetQ;
      { Конструктор без параметров }
      constructor Create(); overload;
      { Конструктор копирования }
      constructor Create(R: TRational); overload;
      { Конструктор инициализатор }
      constructor Create(p, q: Integer); overload;
      { Деструктор }
      destructor Destroy(); override;

      { Группа операций - не создают новый объект }
      { Сложение R0 = R0 + R1 }
      function AddMe(R: TRational): TRational;
      { Вычитание R0 = R0 - R1 }
      function SubtMe(R: TRational): TRational;
      { Умножение R0 = R0 * R1 }
      function MultMe(R: TRational): TRational;
      { Деление R0 = R0 / R1 }
      function DivMe(R: TRational): TRational;

      { Группа операций - создают новый объект }
      { Сложение R3 = R1 + R2 }
```

```
      function AddNew(R: TRational): TRational;
      { Вычитание R3 = R1 - R2 }
      function SubtNew(R: TRational): TRational;
      { Умножение R3 = R1 * R2 }
      function MultNew(R: TRational): TRational;
      { Деление R3 = R1 / R2 }
      function DivNew(R: TRational): TRational;

      { Вывод на консоль }
      procedure Print;
   end; { TRational }
implementation

{ TRational }

{ Конструктор инициализатор }
constructor TRational.Create(p, q: Integer);
begin
   inherited Create;
   Fp := p;
   Fq := q;
   Reduce;
end;

{ Конструктор без параметров }
constructor TRational.Create;
begin
   inherited Create;
   Fp := 0;
   Fq := 1;
end;

{ Конструктор копирования }
constructor TRational.Create(R: TRational);
begin
   inherited Create;
   Fp := R.p;
   Fq := R.q;
end;

{ Деструктор }
destructor TRational.Destroy;
begin
   inherited;
end;

{ Сложение R0 = R0 + R1 }
function TRational.AddMe(R: TRational): TRational;
```

```
begin
   Fp := Fp * R.Fq + R.Fp * Fq;
   Fq := Fq * R.Fq;
   Reduce;
   Result := Self;
end;

{ Деление R0 = R0 / R1 }
function TRational.DivMe(R: TRational): TRational;
begin
   Fp := Fp * R.Fq;
   Fq := Fq * R.Fp;
   Reduce;
   Result := Self;
end;

{ Умножение R0 = R0 * R1 }
function TRational.MultMe(R: TRational): TRational;
begin
   Fp := Fp * R.Fp;
   Fq := Fq * R.Fq;
   Reduce;
   Result := Self;
end;

{ Вычитание R0 = R0 - R1 }
function TRational.SubtMe(R: TRational): TRational;
begin
   Fp := Fp * R.Fq - R.Fp * Fq;
   Fq := Fq * R.Fq;
   Reduce;
   Result := Self;
end;

{ Сложение R3 = R1 + R2 }
function TRational.AddNew(R: TRational): TRational;
var
   RNew: TRational;
begin
   RNew := TRational.Create(Self);
   RNew.Fp := RNew.Fp * R.Fq + R.Fp * RNew.Fq;
   RNew.Fq := RNew.Fq * R.Fq;
   Reduce;
   Result := RNew;
end;

{ Деление R3 = R1 / R2 }
function TRational.DivNew(R: TRational): TRational;
```

```
var
   RNew: TRational;
begin
   RNew := TRational.Create(Self);
   RNew.Fp := RNew.Fp * R.Fq;
   RNew.Fq := RNew.Fq * R.Fp;
   Reduce;
   Result := RNew;
end;

{ Умножение R3 = R1 * R2 }
function TRational.MultNew(R: TRational): TRational;
var
   RNew: TRational;
begin
   RNew := TRational.Create(Self);
   RNew.Fp := RNew.Fp * R.Fp;
   RNew.Fq := RNew.Fq * R.Fq;
   Reduce;
   Result := RNew;
end;

{ Вычитание R3 = R1 - R2 }
function TRational.SubtNew(R: TRational): TRational;
var
   RNew: TRational;
begin
   RNew := TRational.Create(Self);
   RNew.Fp := RNew.Fp * R.Fq - R.Fp * RNew.Fq;
   RNew.Fq := RNew.Fq * R.Fq;
   Reduce;
   Result := RNew;
end;

{ Поиск НОД(Fp, Fq) }
function TRational.Euclid: Integer;
var
   a, b, temp: Integer;
begin
   if Fp = 0 then
   begin
      Result := 0;
      Exit;
   end;
   if Fp > Fq then
   begin
      b := Fp;
      a := Fq;
```

```
   end
   else begin
      b := Fq;
      a := Fp;
   end;
   { b > a }
   while (b mod a <> 0) do
   begin
      temp := a;
      a := b mod a;
      b := temp;
   end;
   Result := a;
end;

{ Сокращение дроби }
procedure TRational.Reduce;
var
   gcd: Integer;
begin
   gcd := Euclid;
   if gcd <> 0 then
   begin
      Fp := Fp div gcd;
      Fq := Fq div gcd;
   end;
end;

procedure TRational.SetP(const Value: Integer);
begin
   Fp := Value;
   Reduce;
end;

procedure TRational.SetQ(const Value: Integer);
begin
   Fq := Value;
   Reduce;
end;

{ Вывод на консоль }
procedure TRational.Print;
begin
   WriteLn('R= ', p, '/', q);
end;

end.
{ ================================================ }
```

В примере представлены два варианта арифметических операций. Первый вариант (AddMe, SubtMe, MultMe, DivMe) рассчитан на то, что изменяется сам вызывающий операцию объект.

```
R1 := TRational.Create(x1, y1);
R2 := TRational.Create(x2, y2);
R1.AddMe(R2); // Математический аналог: R1 = R1 + R2;
```

Второй вариант (AddNew, SubtNew, MultNew, DivNew) создает новый объект внутри метода и возвращает его как результат.

Схема использования этой группы методов выглядит так:

```
R1 := TRational.Create(x1, y1);
R2 := TRational.Create(x2, y2);
// R3 создается внутри метода
R3 := R1.AddNew(R2); // Математический аналог:
R3 = R1 + R2;
```

3.11. Агрегация

В разговоре об основных идеях объектного подхода мы упомянули два способа повторного использования уже написанного кода – агрегацию и наследование.

Под агрегацией понимается такой способ создания новых классов, в рамках которого объекты уже созданных классов входят в объявление нового класса в качестве полей.

Проиллюстрируем сказанное на примере. Так, пусть мы имеем класс TPoint – точка на координатной плоскости.

```
TPoint = class
private
   Fx, Fy: Real;
...
public
   property x: Real read Fx write Fx;
   property y: Real read Fy write Fy;
   constructor Create(_x, _y: Real);
   ...
end; { TPoint }
```

Попробуем описать треугольник.

```
TTriangle = class
private
   Fx1, Fy1: Real;
   Fx2, Fy2: Real;
   Fx3, Fy3: Real;
```

```
    ...
  public
    property x1: Real read Fx1 write Fx1;
    property y1: Real read Fy1 write Fy1;
    property x2: Real read Fx2 write Fx2;
    property y2: Real read Fy2 write Fy2;
    property x3: Real read Fx3 write Fx3;
    property y3: Real read Fy3 write Fy3;
    constructor Create(_x1, _y1, _x2, _y2, _x3, _y3,: Real);
    ...
  end; { TTriangle }
```

Несколько громоздко, не правда ли? Учтем к тому же, что это только фрагмент описания класса.

Попробуем подойти к проблеме с другой стороны. Треугольник – это три точки. Попробуем описать класс «Треугольник» с использованием имеющегося класса «Точка».

```
  TTriangle = class
  private
    Fp1, Fp2, Fp3: TPoint;
    ...
  public
    property p1: TPoint read Fp1 write Fp1;
    property p2: TPoint read Fp2 write Fp2;
    property p3: TPoint read Fp3 write Fp3;
    ...
  end; { TTriangle }
```

Это и есть пример агрегации с использованием объектов класса `TPoint`.

Попробуем понять, как должен выглядеть конструктор и как будет происходить создание объекта. Логично сделать несколько конструкторов. Пусть один из них принимает в качестве параметров 6 координат (3 точки), а второй три объекта класса `TPoint`. В любом случае ясно следующее: именно конструктор класса `TTriangle` должен создавать объекты `Fp1`, `Fp2`, `Fp3`, а деструктор – удалять их. Как правило, это общая ситуация – память выделяется в конструкторе и освобождается в деструкторе. Заметим, что можно было организовать процесс по-другому, осуществляя в конструкторе лишь присваивание ссылок, оставляя вопрос о создании/уда-лении объектов головной программе, но по смыслу это неправильно. Треугольник должен быть полноправным хозяином своих точек, что дает ему определенные права (он может делать с ними все, что угодно), но и накладывает некоторые обязательства (он должен следить за своевременным созданием и удалением объектов-точек).

Для начала дополним фрагмент объявления класса «Точка» конструктором копирования.

```
TPoint = class
private
   Fx, Fy: Real;
   ...
public
   property x: Real read Fx write Fx;
   property y: Real read Fy write Fy;
   constructor Create(_x, _y: Real); overload;
   constructor Create(p: TPoint); overload;
   ...
end; { TPoint }

constructor TPoint.Create(_x, _y: Real);
begin
   Fx := _x;
   Fy := _y;
end;
constructor TPoint.Create(p: TPoint);
begin
   Fx := p.x;
   Fy := p.y;
end;

TTriangle = class
private
   Fp1, Fp2, Fp3: TPoint;
public
   property p1: TPoint read Fp1 write Fp1;
   property p2: TPoint read Fp2 write Fp2;
   property p3: TPoint read Fp3 write Fp3;

   constructor Create(_p1, _p2, _p3: TPoint); overload;
   constructor Create(_x1, _y1, _x2, _y2, _x3, _y3: Real);
     overload;
   destructor Destroy; override;
end; { TTriangle }

constructor TTriangle.Create(_p1, _p2, _p3: TPoint);
begin
   Fp1 := TPoint.Create(_p1);
   Fp2 := TPoint.Create(_p2);
   Fp3 := TPoint.Create(_p3);
end;

constructor TTriangle.Create(_x1, _y1, _x2, _y2, _x3,
  _y3: Real); overload;
```

```
begin
   Fp1 := TPoint.Create(_x1, _y1);
   Fp2 := TPoint.Create(_x2, _y2);
   Fp3 := TPoint.Create(_x3, _y3);
end;

destructor TTriangle.Destroy;
begin
   Fp1.Free;
   Fp2.Free;
   Fp3.Free;
end;
```

Иногда возникает потребность использовать при объявлении класса (например, при агрегации) другой класс, который будет объявлен далее в тексте программы. При этом компилятор не сможет обнаружить этот класс и выдаст сообщение об ошибке. Существует способ ему помочь – так называемое предварительное объявление класса. Выглядит это следующим образом:

```
TExample = class; { предварительное объявление }
TMyClass = class
public
   e: TExample;
   ...
end; { TMyClass }

TExample = class
   ...
end; { TExample }
```

Отметим также, что между предварительным объявлением и полным объявлением не должно быть ничего, кроме других объявлений типов, т.е. они должны находиться в одной секции **type**.

3.12. Наследование и полиморфизм. Виртуальные методы и позднее связывание

Рассмотрев пример агрегации, поговорим теперь о наследовании. Для лучшего понимания будем изучать материал на знакомом нам примере с координатной плоскостью. Вспомним, как выглядит результат нашего анализа предметной области:

1. *Точка*:
   — данные: (x, y, Visible);
   — операции: Show, Hide, MoveTo, IsVisible.
2. *Квадрат*:

— данные: (x, y, a, Visible);
— операции: Show, Hide, S, MoveTo, IsVisible.

3. *Круг*:
   — данные: (x, y, r, Visible);
   — операции: Show, Hide, S, MoveTo, IsVisible.
4. *Треугольник*:
   — данные: (x1, y1, x2, y2, x3, y3, Visible);
   — операции: Show, Hide, S, MoveTo, IsVisible.
5. *Прямоугольник*:
   — данные: (x1, y1, a, b, Visible);
   — операции: Show, Hide, S, MoveTo, IsVisible.
6. *Координатная плоскость*:
   — данные: (набор фигур, Visible);
   — операции: Show, Hide, S, IsVisible.

Сравним квадрат и точку. Очевидно, наблюдается явная преемственность по данным. Так, по сравнению с точкой у квадрата добавляется всего один новый параметр – сторона `a`. А в операциях? Совершенно ясно, что механизм работы с полем `Visible` абсолютно одинаков и у точки, и у квадрата. А вот операции `Show`, `Hide`, `MoveTo`, `S` работают по-разному.

Тем не менее фигуры имеют достаточно общего, а значит, было бы неплохо унаследовать квадрат от точки, перенимая все, что было у точки, добавляя сторону и механизмы для доступа к ее значению (длине) и заменяя (переписывая) основные операции.

В общем случае в языке Object Pascal это делается следующим образом. При описании нового класса мы можем указать тот класс, от которого производится наследование. В этом случае новый класс называется *потомком*, а тот класс, от которого производится наследование (выводимость), – *предком*.

Схема классов в программном комплексе, которая иллюстрирует отношения наследования между различными классами, называется *схемой выводимости* классов.

Заметим, что в языке Object Pascal предок может быть только один, тогда как, например, в языке C++ их может быть сколько угодно. В этом смысле говорят, что в Pascal принято *одиночное* наследование, а в C++ *множественное* наследование. Несмотря на то, что теоретически множественное наследование обеспечивает большую гибкость при разработке программ, чем одиночное, по мнению многих теоретиков и практиков ООП, одиночное наследование делает программы более понятными и защищенными от ошибок. Дело в том, что множественное наследование приводит к появлению большого

количества спорных ситуаций, в которых компилятор принимает то или иное решение, исходя из своих соображений, не всегда очевидных для разработчика, что приводит к возникновению ошибок, найти которые крайне трудно. В этом смысле отсутствие множественного наследования в Object Pascal вряд ли можно причислить к недостаткам. Строго говоря, в Object Pascal существует понятие *интерфейс* (существует оно и в других языках для обеспечения технологии компонентной разработки программ), которое позволяет использовать элементы множественного наследования там, где это необходимо. В данной книге конструкция **interface** не рассматривается.

Синтаксис организации наследования в общем случае выглядит так:

```
TBaseClass = class
   ...
end; { TBaseClass }
TDerivedClass = class(TBaseClass)
   ...
end; { TDerivedClass }
```

Таким образом, при наследовании имя класса-предка указывается в скобках. Унаследовав от класса `TBaseClass`, мы становимся обладателями всех его полей, свойств и методов.

Рассмотрим пример класса «Точка» и унаследованного от него класса «Квадрат». Для начала поработаем с классом «Точка».

```
{ ================================================ }
{ Пример 10.4 }
{ Класс Точка }
Unit UFigures;
interface
uses Graphics;

type
   { Класс Точка }
   TPoint = class
   private
      Fx, Fy: Word;           { Координаты            }
      FVisible: Boolean;      { Признак видимости     }
      Canvas: TCanvas;        { «Холст» для рисования }

      procedure SetX(const Value: Word);
      procedure SetY(const Value: Word);

   public
      property x: Word read Fx write SetX; { Координата X }
```

```
    property y: Word read Fy write SetY; { Координата Y }
    property Visible: Boolean read FVisible;
    { Признак видимости }

    procedure Show;        { Показать }
    procedure Hide;        { Скрыть   }
    function S: Word;      { Площадь  }

    constructor Create(_x, _y: Word; _Cnv: TCanvas);
      overload;
    constructor Create(p: TPoint); overload;
    destructor Destroy; override;
  end; { TPoint }

implementation

{ TPoint }
constructor TPoint.Create(p: TPoint);


begin
  Fx := p.x;
  Fy := p.y;
  FVisible := False;
  Canvas := P.Canvas;
end;

constructor TPoint.Create(_x, _y: Word; _Cnv: TCanvas);
begin
  Fx := _x;
  Fy := _y;
  FVisible := False;
  Canvas := _Cnv;
end;

procedure TPoint.Hide;
begin
  if Visible then
  begin
    Canvas.Pixels[Fx, Fy] := clBtnFace;
    FVisible := False;
  end;
end;

procedure TPoint.Show;
begin
  if not Visible then
```

```
   begin
      Canvas.Pixels[Fx, Fy] := clBlack;
      FVisible := True;
   end;
end;

function TPoint.S: Word;
begin
   Result := 0;
end;

procedure TPoint.SetX(const Value: Word);
begin
   Fx := Value;
   if Visible then
   begin
      Hide;
      Show;
   end;
end;
procedure TPoint.SetY(const Value: Word);
begin
   Fy := Value;
   if Visible then
   begin
      Hide;
      Show;
   end;
end;

destructor TPoint.Destroy;
begin
   Hide;
   inherited;
end;

end.
{ ============================================== }
```

Сделаем несколько замечаний по примеру:

1. В классе `TPoint` все данные скрыты.

2. Среди данных присутствует поле `Canvas` – холст для рисования. Это сделано для того, чтобы точка «знала», где ее нужно рисовать. Тип данных `TCanvas` – класс из библиотеки VCL (для его использования – **uses** `Graphics` в секции **interface** модуля). Этот класс управляет

отображением графики в окне. В дальнейшем мы будем пользоваться его методами, их назначение представляется очевидным по названию.

3. Для полей-координат прописаны соответствующие свойства. При установке новых значений координат производится перерисовка точки на холсте.

4. Поле `Visible` сделано доступным только для чтения.

5. Методы `Show` и `Hide` занимаются рисованием/скрытием точки в окне, используя класс `TCanvas` и его индексированное свойство `Pixels`. При этом для рисования используется цвет `clBlack` (черный – константа из модуля `Graphics`), а для скрытия – цвет `clBtnFace` (цвет фона окна – константа из модуля `Graphics`).

6. Деструктор скрывает точку на координатной плоскости.

Теперь напишем тестовую программу. Для этого создадим в Borland Delphi новый проект (оконное приложение) с одним окном, добавим к нему модуль `figures`, разместим на форме две кнопки `Show` и `Hide`. Добавим тестирующий код и получим в результате:

```
{ ================================================= }
{ Пример 10.5 }
{ Точки на форме }
Unit FMain;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls;

type
   TfrmMain = class(TForm)
      btnShow: TButton;
      btnHide: TButton;
      procedure btnShowClick(Sender: TObject);
      procedure btnHideClick(Sender: TObject);
   private
      { Private declarations }
   public
      { Public declarations }
   end;
var
   frmMain: TfrmMain;

implementation
{$R *.dfm}

uses figures;
```

```
var
   pt: array [1..1000] of TPoint;

procedure TfrmMain.btnShowClick(Sender: TObject);
var
   i, j, k: Word;
begin
   Randomize;
   for k := 1 to 1000 do
   begin
      i := random(Width);
      j := random(Height);
      pt[k] := TPoint.Create(i, j, Canvas);
      pt[k].Show;
   end;
end;

procedure TfrmMain.btnHideClick(Sender: TObject);
var
   k: Word;
begin
   for k := 1 to 1000 do
   begin
      pt[k].Free;
      pt[k] := nil;
   end;
end;

end.
{ ================================================== }
```

При нажатии на кнопку `Show` будем создавать динамически 1000 точек со случайными координатами (для генерации случайных чисел используем функцию `random`) и отображать их в окне. Процедура `Randomize` используется для инициализации датчика случайных чисел.

Рис. 10.15. Точки в окне – результат нескольких нажатий на кнопку `Show`

При нажатии на кнопку `Hide` производим удаление объектов-точек, находящихся в массиве точек `pt` (при этом точки скрываются в окне посредством метода `Hide` класса `TPoint`), после чего инициализируем значением **`nil`** элементы массива. В противном случае повторное нажатие на кнопку `Hide` приведет к краху программы.

Убедившись в работоспособности созданного класса `TPoint`, выведем из него класс `TSquare`.

```
TSquare = class(TPoint)
private
   Fa: Word;

   procedure SetA(const Value: Word);
public
   property a: Word read Fa write SetA;

   procedure Show;
   procedure Hide;
   constructor Create (_x, _y, _a: Word; _Cnv: TCanvas);
     overload;
   constructor Create(s: TSquare); overload;
   destructor Destroy; override;
 end; { TSquare }
```

Первое, что бросается в глаза, – простота и лаконичность объявления класса. Действительно, ведь существенная часть функциональности заимствована из `TPoint`. Проведем анализ объявления класса:

1. Добавлено поле `Fa` и свойство `a` для доступа к нему. Поля `Fx`, `Fy`, `FVisible` и вся связанная с ними функциональность (в том числе и свойства) унаследованы от предка.

2. Переопределены методы `Show` и `Hide`. Действительно, квадрат изображается и скрывается отличным от точки способом. Заметим, что переписав заголовки методов `Show` и `Hide`, мы осуществили так называемое «перекрытие методов». Новые реализации `Show` и `Hide` скрывают старые варианты. Заметим, тем не менее, что мы можем вызвать внутри, например, метода `Show` метод `Show` предка, написав **`inherited`** `Show`. Вот мы и выяснили, зачем нужно ключевое слово **`inherited`** – вызов перекрытого метода предка!

3. Для `Show` в данном примере это не актуально, а вот для конструкторов и деструкторов – крайне важно! Создание и уничтожение объектов должно происходить в полном соответствии с их внутренней структурой. Так, конструктор должен вызвать конструктор предка и только потом производить дополнительные действия. Деструктор, напротив, сначала должен совершить завершающие действия для новых данных и только потом вызвать деструктор предка для аналогичных действий над данными предка.

Проиллюстрируем сказанное выше реализацией методов класса `TSquare`.

```
constructor TSquare.Create(_x,_y,_a:Word;_Cnv:TCanvas);
begin
   inherited Create(_x, _y, _Cnv);
   Fa := _a;
end;

constructor TSquare.Create(s: TSquare);
begin
   inherited Create(s.x, s.y, s.Canvas);
   Fa := s.a;
end;

destructor TSquare.Destroy;
begin
   inherited;
end;
```

```
procedure TSquare.Show;
begin
   if not Visible then
   begin
      Canvas.Pen.Color := clBlue;
      Canvas.Rectangle(x, y + a, x + a, y);
      FVisible := True;
   end;
end;

procedure TSquare.Hide;
begin
   if Visible then
   begin
      Canvas.Pen.Color := clBtnFace;
      Canvas.Rectangle(x, y + a, x + a, y);
      FVisible := False;
   end;
end;

procedure TSquare.SetA(const Value: Word);
begin
   Fa := Value;
   if Visible then
   begin
      Hide;
      Show;
   end;
end;
```

Посмотрим, как все это работает. Дополним наше оконное приложение двумя кнопками на форме, объявим в секции **implementation** модуля формы `fmain` массив квадратов `sq` и реализуем методы реакции на нажатие новых кнопок.

```
...
var
   sq: array [1..200] of TSquare;
...
procedure TvbtnShow1.btnShow1Click(Sender: TObject);
var
   i, j, a, k: Word;
begin
   Randomize;
   for k := 1 to 200 do
```

```
    begin
      i := random(Width);
      j := random(Height);
      a := random(50);
      sq[k] := TSquare.Create(i, j, a, Canvas);
      sq[k].Show;
    end;
  end;

  procedure TvbtnShow1.btnHide1Click(Sender: TObject);
  var
    k: Word;
  begin
    for k := 1 to 200 do
    begin
      sq[k].Free;
      sq[k] := nil;
    end;
  end;
```

Запустим пример и посмотрим, что получилось, нажав несколько раз кнопку `btnShow1`.

На рисунке – возможный результат нескольких нажатий.

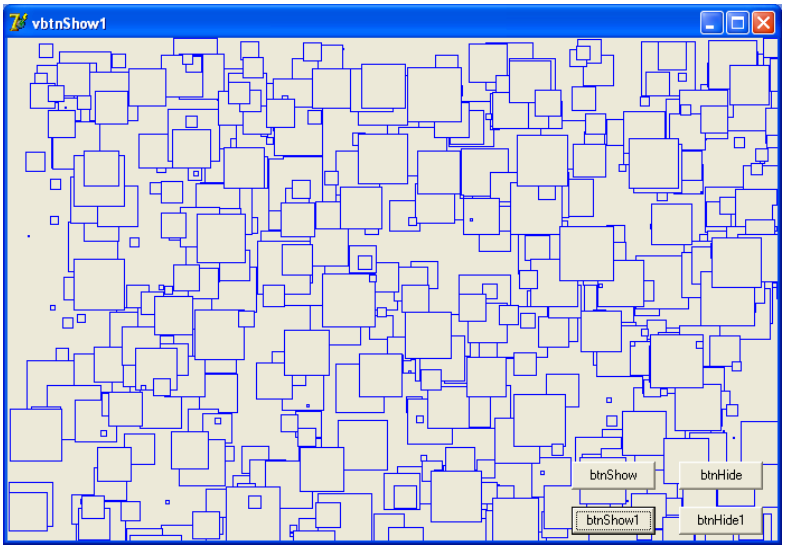

Рис. 10.16. Квадраты в окне – результат нескольких нажатий на кнопку `Show1`

Теперь нажмем на кнопку `btnHide1`. В чем дело? Почему ничего не происходит? Попробуем разобраться с происходящим. В режиме отладки проанализируем последовательность работы. Открывается следующая картина:

1. Попадаем в метод `btnHide1Click`.
2. В цикле вызываем метод `Free`.
3. Попадаем в деструктор класса `TSquare`.
4. Ключевое слово **`inherited`** переносит нас в деструктор класса `TPoint`.
5. Ага! Вызывается метод `Hide` класса `TPoint`, а не метод `Hide` класса `TSquare`, соответственно, у квадрата «стирается» лишь 1 точка – левый нижний угол.

Что делать? Нам нужно, чтобы в окне цветом фона рисовался квадрат, а не точка. Первое возможное решение состоит в том, чтобы переписать деструктор класса `TSquare`, поместив туда код, скопированный из `TPoint`, с заменой рисования точки рисованием квадрата. В результате все будет работать.

Как избежать этого дублирования? Как добиться того, чтобы, попав в деструктор класса `TPoint`, при вызове метода `Hide` компилятор «сообразил», что вызывающий деструктор объект на самом деле `TSquare`, и вызвал метод `Hide` квадрата?

Сложная задача для компилятора и компоновщика. Вернее, нерешаемая. Все дело в том, что на этапе компиляции и сборки адрес того метода, который необходимо вызвать в деструкторе класса `TPoint`, неизвестен. Лишь во время работы выяснится, кто именно его вызвал (`TPoint`, `TSquare` или кто-нибудь еще). В результате мы имеем дело с необходимостью «полиморфного» поведения, т.е. определения адреса метода, который необходимо вызвать, на этапе выполнения программы.

Случай, когда адрес вызываемого метода определяется не на этапе компиляции и сборки программы (ранее связывание), а на этапе работы программы, называется *поздним связыванием* (*late binding*). Язык Object Pascal предоставляет средства для использования этого сложного механизма в программах. Рассмотрим, в чем они состоят, но прежде отметим несколько важных моментов.

1. Позднее связывание возможно тогда и только тогда, когда классы находятся в иерархии наследования.

2. Все классы языка Object Pascal, у которых явно не указан предок, считаются выведенными из класса TObject, содержащего некоторую полезную функциональность (разные системные механизмы, которые будут упомянуты в разделе «Внутренняя структура объекта. Методы класса. Динамический контроль типов, операторы IS и AS»). Таким образом, класс TObject является общим предком для всех классов. Его данные и методы содержатся во всех объектах и могут быть использованы.

3. Чтобы указать компилятору и сборщику на необходимость использования механизма позднего связывания, требуется в конце заголовка метода при его объявлении поместить ключевое слово **`virtual`**. Далее в потомках необходимо в точности сохранять заголовок метода (реализация, разумеется, может быть изменена), за исключением замены слова **`virtual`** на **`override`** (вот мы и выяснили, что означало таинственное **`override`** после заголовка деструктора).

Методы, объявленные как **`virtual`** и в дальнейшем переопределяемые в потомках с ключевым словом **`override`**, называются *виртуальными*.

Изменим наш пример с учетом приобретенных знаний.

```
TPoint = class
private
   Fx, Fy: Word;       { Координаты             }
   FVisible: Boolean;  { Признак видимости      }
   Canvas: TCanvas;    { «Холст» для рисования }

   procedure SetX(const Value: Word);
   procedure SetY(const Value: Word);
```

```
  public
    property x: Word read Fx write SetX; { Координата X }
    property y: Word read Fy write SetY; { Координата Y }
    property Visible: Boolean read FVisible;
    {Признак видимости}

    procedure Show; virtual;      { Показать }
    procedure Hide; virtual;      { Скрыть   }
    function S: Word;             { Площадь  }

    constructor Create (_x, _y: Word; _Cnv: TCanvas);
      overload;
    constructor Create(p: TPoint); overload;
    destructor Destroy; override;
  end; { TPoint }

  { Класс Квадрат }
  TSquare = class(TPoint)
  private
    Fa: Word;
    procedure SetA(const Value: Word);
  public
    property a: Word read Fa write SetA;

    procedure Show; override;
    procedure Hide; override;
    constructor Create(_x, _y, _a: Word; _Cnv: TCanvas);
      overload;
    constructor Create(s: TSquare); overload;
    destructor Destroy; override;
  end; { TSquare }
```

Итак, мы сделали методы `Show` и `Hide` виртуальными. Реализация их при этом не изменится (указывать при реализации **`virtual`** и **`override`** не нужно). Компилируем и запускаем пример – теперь все работает!

Заметим, что деструктор у нас и так был виртуальным. Связано это с тем, что у класса `TObject` – общего предка деструктор `Destroy` является виртуальным. Сделано это для того, чтобы корректно происходило освобождение памяти по всей иерархии наследования.

Реализуем классы «Круг» (`TCircle`) и «Прямоугольник» (`TRectangle`). Будем выводить `TRectangle` из `TSquare`, добавляя сторону и перекрывая методы рисования/скрытия. Как реализовать `TCircle`? Сравнивая `TCircle` и `TSquare`, обнаруживаем, что они

полностью совпадают по данным, за исключением их интерпретации. Выведем TCircle из TSquare, перекрыв методы рисования.

Отдельно остановимся на классе «Координатная плоскость» (TPlane). Учитывая, что TPlane – набор фигур, не имеет смысла выводить класс из фигуры. Будем выводить TPlane из стандартного TObject. Попробуем описать данные TPlane. Как было указано ранее, данные этого класса составляет массив фигур. Тут нас, однако, подстерегает некоторая проблема. К сожалению, все фигуры принадлежат к разным типам. Значит, объявить массив нам не удастся? Или все-таки есть возможность?

Вспомним, что нам известно про массив. Массив – средство хранения больших объемов *однотипных* данных. Здесь налицо данные *разнотипные*. Как их объединить вместе? На помощь приходит важнейший элемент синтаксиса языка Object Pascal, специально предназначенный, в том числе, и для решения таких проблем. Вспомним, что объявляя переменную объектного типа мы на самом деле объявляем указатель на нее, т.е. адрес. Поскольку все адреса однотипны, объединить их в массив должно быть возможно. Единственно, нужно иметь в виду следующее соглашение: при объявлении таких массивов необходимо использовать базовый класс иерархии. Объявив FData: **array of** TPoint, мы получим возможность на этапе работы программы складывать туда различные фигуры, т.к. в языке Object Pascal допускается присваивание объектной переменной базового типа любого объекта-потомка.

```
var
   p: TPoint;
   q: TCircle;
begin
   ...
   p := q; // Так можно
   q := p; // Так нельзя
   ...
end.
```

Таким образом, объявляя массив из TPoint, мы сможем хранить в нем любые фигуры, выведенные из TPoint.

Следующий вопрос: как работать с этими фигурами? Ведь у каждого из классов есть свои методы, в том числе Show и Hide, которые работают по-разному по всей иерархии наследования. На помощь приходит следующее обстоятельство: каждый объект, в действительности, хранит «о себе любимом» некоторые данные, в частности информацию о своем типе. Сделав методы Show и Hide по всей иерархии виртуальными, мы добьемся того, что FData[i].Show

на этапе работы программы разберется, чт.∴ лежит на самом деле в *i*-й ячейке массива `FData`, и вызовет соответствующий метод `Show`. Это еще один пример работы механизма позднего связывания и полиморфного поведения.

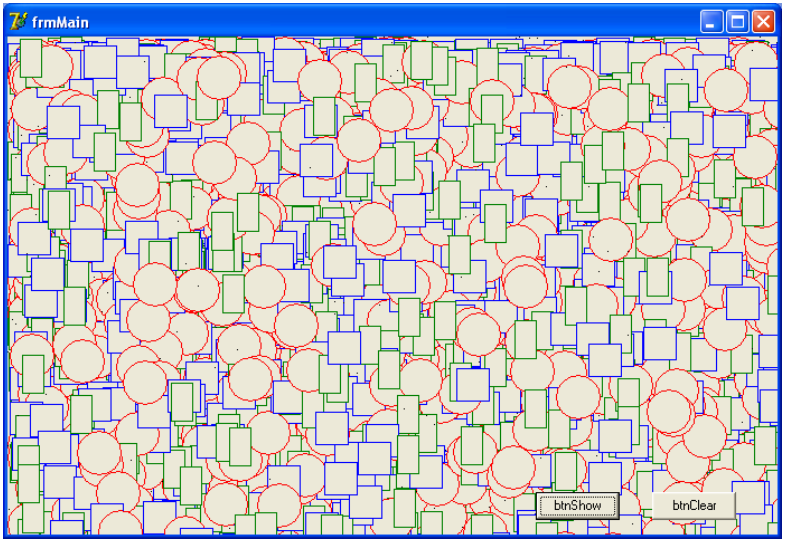

Рис. 10.17. Фигуры на координатной плоскости

Приведем полностью исходный код примера и демонстрационного приложения.

```
{ ======================================== }
{ Пример 10.6 }
{ Фигуры на форме }
Unit UFigures;
interface
uses Graphics;

type
   TPoint = class
   private
      Fx, Fy: Word;
      FVisible: Boolean;
      Canvas: TCanvas;

      procedure SetX(const Value: Word);
      procedure SetY(const Value: Word);
```

```
  public
    property x: Word read Fx write SetX;
    property y: Word read Fy write SetY;
    property Visible: Boolean read FVisible;

    procedure Show; virtual;
    procedure Hide; virtual;
    function S: Word;

    constructor Create(_x, _y: Word; _Cnv: TCanvas); overload;
    constructor Create(p: TPoint); overload;
    destructor Destroy; override;
  end; { TPoint }

  TSquare = class(TPoint)
  private
    Fa: Word;
    procedure SetA(const Value: Word);
  public
    property a: Word read Fa write SetA;

    procedure Show; override;
    procedure Hide; override;

    constructor Create(_x, _y, _a: Word; _Cnv: TCanvas);
      overload;
    constructor Create(s: TSquare); overload;
    destructor Destroy; override;
  end; { TSquare }

  TCircle = class(TSquare)
  public
    procedure Show; override;
    procedure Hide; override;

    constructor Create(_x, _y, _r: Word; _Cnv: TCanvas);
      overload;
    constructor Create(c: TSquare); overload;
    destructor Destroy; override;
  end; { TCircle }

  TRectangle = class(TSquare)
  private
    Fb: Word;
    procedure SetB(const Value: Word);
  public
    property B: Word read Fb write SetB;
```

```
      procedure Show; override;
      procedure Hide; override;
      constructor Create(_x, _y, _a, _b: Word; _Cnv: TCanvas);
        overload;
      constructor Create(r: TRectangle); overload;
      destructor Destroy; override;
   end; { TRectangle }

   TPlane = class(TObject)
   private
      FData: array of TPoint;
      FDim: Word;
      FCanvas: TCanvas;

      function GetData(Index: Word): TPoint;
      procedure SetData(Index: Word; const Value: TPoint);
   public
      property Dim: Word read FDim;
      property Canvas: TCanvas read FCanvas;
      property Data[Index: Word]:TPoint read GetData
      write SetData; default;

      procedure Show;
      procedure Clear;

      procedure Add(Index: Word; f: TPoint);

      constructor Create(_dim: Word; _Cnv: TCanvas);
      destructor Destroy; override;
   end; { TPlane }

implementation

{ TPoint }

constructor TPoint.Create(p: TPoint);
begin
   Fx := p.x;
   Fy := p.y;
   Canvas := P.Canvas;
   FVisible := False;
end;

constructor TPoint.Create(_x, _y: Word; _Cnv: TCanvas);
begin
   Fx := _x;
   Fy := _y;
   FVisible := False;
```

```
   Canvas := _Cnv;
end;
procedure TPoint.Hide;
begin
   if Visible then
   begin
      Canvas.Pixels[Fx, Fy] := clBtnFace;
      FVisible := False;
   end;
end;

procedure TPoint.Show;
begin
   if not Visible then
   begin
      Canvas.Pixels[Fx, Fy] := clBlack;
      FVisible := True;
   end;
end;

function TPoint.S: Word;
begin
   Result := 0;
end;

procedure TPoint.SetX(const Value: Word);
begin
   Fx := Value;
   if Visible then
   begin
      Hide;
      Show;
   end;
end;

procedure TPoint.SetY(const Value: Word);
begin
   Fy := Value;
   if Visible then
   begin
      Hide;
      Show;
   end;
end;

destructor TPoint.Destroy;
begin
   Hide;
```

```
   inherited;
end;
{ TSquare }

constructor TSquare.Create(_x,_y,_a:Word;_Cnv:TCanvas);
begin
   inherited Create(_x, _y, _Cnv);
   Fa := _a;
end;

constructor TSquare.Create(s: TSquare);
begin
   inherited Create(s.x, s.y, s.Canvas);
   Fa := s.a;
end;

destructor TSquare.Destroy;
begin
   inherited;
end;

procedure TSquare.Show;
begin
   if not Visible then
   begin
      Canvas.Pen.Color := clBlue;
      Canvas.Rectangle(x, y + a, x + a, y);
      FVisible := True;
   end;
end;

procedure TSquare.Hide;
begin
   if Visible then
   begin
      Canvas.Pen.Color := clBtnFace;
      Canvas.Rectangle(x, y + a, x + a, y);
      FVisible := False;
   end;
end;

procedure TSquare.SetA(const Value: Word);
begin
   Fa := Value;
   if Visible then
   begin
      Hide;
      Show;
```

```
   end;
end;
{ TCircle }

constructor TCircle.Create(_x,_y,_r:Word;_Cnv:TCanvas);
begin
   inherited Create(_x, _y, _r, _Cnv);
end;

constructor TCircle.Create(c: TSquare);
begin
   inherited Create(c.x, c.y, c.a, c.Canvas);
end;

destructor TCircle.Destroy;
begin
   inherited;
end;

procedure TCircle.Hide;
begin
   if Visible then
   begin
      Canvas.Pen.Color := clBtnFace;
      Canvas.Ellipse(x, y + a, x + a, y);
      FVisible := False;
   end;
end;

procedure TCircle.Show;
begin
   if not Visible then
   begin
      Canvas.Pen.Color := clRed;
      Canvas.Ellipse(x, y + a, x + a, y);
      FVisible := True;
   end;
end;

{ TRectangle }

constructor TRectangle.Create(_x, _y, _a, _b:
Word; _Cnv: TCanvas);
begin
   inherited Create(_x, _y, _a, _Cnv);
   Fb := _b;
end;
```

```
constructor TRectangle.Create(r: TRectangle);
begin
   inherited Create(r.x, r.y, r.a, r.Canvas);
   Fb := r.b;
end;

destructor TRectangle.Destroy;
begin
   inherited;
end;

procedure TRectangle.Hide;
begin
   if Visible then
   begin
      Canvas.Pen.Color := clBtnFace;
      Canvas.Rectangle(x, y + b, x + a, y);
      FVisible := False;
   end;
end;

procedure TRectangle.Show;
begin
   if not Visible then
   begin
      Canvas.Pen.Color := clGreen;
      Canvas.Rectangle(x, y + b, x + a, y);
      FVisible := True;
   end;
end;

procedure TRectangle.SetB(const Value: Word);
begin
   Fb := Value;
   if Visible then
   begin
      Hide;
      Show;
   end;
end;

{ TPlane }

constructor TPlane.Create(_dim: Word; _Cnv: TCanvas);
begin
   inherited Create;
   FDim := _dim;
   FCanvas := _Cnv;
```

```
   SetLength(FData, FDim);
end;
destructor TPlane.Destroy;
var
   i: Word;
begin
   for i := 0 to Dim - 1 do
     FData[i].Free;
   SetLength(FData, 0);
   inherited;
end;

function TPlane.GetData(Index: Word): TPoint;
begin
   Result := FData[Index];
end;

procedure TPlane.SetData (Index: Word; const Value: TPoint);
begin
   FData[Index] := Value;
end;

procedure TPlane.Clear;
var
   i: Word;
begin
   for i := 0 to Dim - 1 do
   begin
      FData[i].Free;
      FData[i] := nil;
   end;
end;

procedure TPlane.Show;
var
   i: Word;
begin
   for i := 0 to Dim - 1 do
     FData[i].Show;
end;

procedure TPlane.Add(Index: Word; f: TPoint);
begin
   FData[Index] := f;
end;

end.
```

```
{ Демонстрационное приложение }
Unit FMain;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes,  Graphics,
    Controls, Forms, Dialogs, StdCtrls, Figures;
type
   TfrmMain = class(TForm)
      btnShow: TButton;
      btnClear: TButton;
      procedure btnShowClick(Sender: TObject);
      procedure btnClearClick(Sender: TObject);
      procedure FormCreate(Sender: TObject);
      procedure FormDestroy(Sender: TObject);
   private
      { Private declarations }
      Plane: TPlane;
   public
      { Public declarations }
 end;

var
   frmMain: TfrmMain;

implementation
{$R *.dfm}

procedure TfrmMain.FormCreate(Sender: TObject);
begin
   Plane := TPlane.Create(10000, Canvas);
end;

procedure TfrmMain.FormDestroy(Sender: TObject);
begin
   Plane.Free;
end;

procedure TfrmMain.btnShowClick(Sender: TObject);
var
   i1, j1, i2, j2, k: Word;
   f: 0..3;
   p: TPoint;
begin
   Randomize;
   for k := 1 to Plane.Dim do
   begin
      i1 := Random(Width);
```

```
      j1 := Random(Height);
      f  := Random(4);
      case f of
         0: p := TPoint.Create(i1, j1, Plane.Canvas);
         1: p := TSquare.Create(i1, j1, 30, Plane.Canvas);
         2: p := TCircle.Create(i1, j1, 40, Plane.Canvas);
         3: p := TRectangle.Create(i1, j1, 20, 35,
              Plane.Canvas);
      end;
      Plane.Add(k - 1, p);
   end;
   Plane.Show;
end;

procedure TfrmMain.btnClearClick(Sender: TObject);
begin
   Plane.Clear;
end;

end.
{ ================================================== }
```

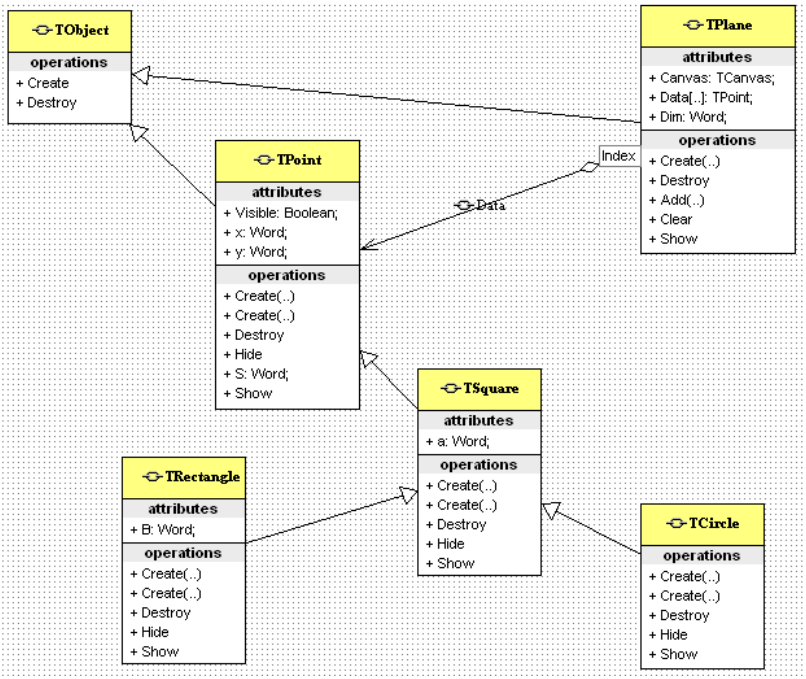

Рис. 10.19. Фигуры на координатной плоскости – схема выводимости

Нашу цепочку классов можно продолжать сколь угодно долго, выводя все новые и новые фигуры. Понятно, что у каждого из нас могут получиться несколько отличные схемы наследования. Построение удачной схемы наследования – важный шаг для того, чтобы проект оказался успешным. Искусство создания подобных схем приходит с годами и опытом. Экспериментируйте, обращайте внимание на закономерности, и у вас тоже будет получаться!

3.13. Абстрактные методы

Быть может, вам уже попадались в литературе по программированию «умные слова»: концепция, парадигма, абстракция, иерархия? Надеемся, вашей первой реакцией не было желание закрыть книгу и сложить ее куда-нибудь подальше или предложить знакомым как безвредное лекарство от бессонницы. Обещаем: в этом разделе не будет долгого и нудного философствования. Вместо этого мы вкратце познакомимся с одним из элементов объектно-ориентированного программирования – *абстрактными методами*.

Как стало ясно из предыдущего раздела, построение цепочек наследования – мощный инструмент проектирования и разработки программ. Иногда встречаются ситуации, когда мы хотим обеспечить *полиморфное поведение* – присваивание объектов производных типов элементу базового типа с последующим вызовом «правильных» (часто говорят «релевантных») методов в момент работы программы. Это обеспечивается технологией позднего связывания и механизмом виртуальных методов. В связи с этим во многих случаях является оправданным включение большого числа виртуальных методов в базовый класс. Но как быть, если для базового класса эти методы не имеют смысла и содержанием будут наполнены лишь в потомках?

В предыдущем примере мы не стали реализовывать класс «Треугольник». Подумаем, как он будет выглядеть. Очевидно, что мы хотели бы сохранить полиморфные `Show` и `Hide`. Получается, что для этого мы должны вывести его из `TPoint`. Это возможно, но несколько странно, а ведь объектный подход подразумевает естественность представления. Отсюда вывод – нужен базовый для всех класс «Фигура», из которого можно вывести, с одной стороны, «Точку», а с другой стороны, «Треугольник», сохранив дальше иерархию наследования.

Итак, мы хотим использовать класс «Фигура» в качестве базового для образования иерархии классов. При этом мы хотим предусмотреть

виртуальный метод Show, который будет заниматься отображением фигур на координатной плоскости. Зададимся вопросом – как должна выглядеть реализация этого метода? Для точки, круга, квадрата и даже для самой координатной плоскости все ясно, всем известно, как рисовать стандартные геометрические фигуры. А вот как быть с рисованием объектов класса «Фигура»? И вообще, должны ли быть в программе экземпляры этого типа, ведь в предметной области нет «Фигур», есть точки, круги, квадраты…

Возможный вариант – повесить у себя перед глазами плакат следующего содержания: «Никогда не создавать экземпляры класса “Фигура”» и написать в этом классе пустую реализацию метода Show.

```
TShape = class
private
   ...
public
   procedure Show; virtual;
   ...
end;

procedure TShape.Show;
begin
   { пустая реализация }
end;
```

Ясно, что это не лучший способ. Кто же будет страховать нас от ошибок? Для этого Object Pascal имеет специальный механизм – *абстрактные методы*.

Для начала дадим определение. *Абстрактные методы – виртуальные методы, для которых в классе отсутствует реализация*. Должны быть переопределены в потомках с указанной реализацией.

Так, для рассмотренного примера мы можем написать:

```
TShape = class
private
   ...
public
   procedure Show(); virtual; abstract;
   ...
end;
```

Заметим, что в этом случае мы не можем указать реализацию для метода Show.

Как быть с классом TShape? Обзаведясь хотя бы одним абстрактным методом, класс TShape становится *абстрактным классом*. В языке программирования C++ это означало бы, что мы не можем создавать

экземпляры класса `TShape`. В языке Object Pascal требования несколько понижены, мы можем создавать экземпляры, но компилятор выдает предупреждение о том, что мы создали экземпляр абстрактного класса (`constructing instance of 'TShape' containing abstract method TShape.Show`). А вот попытка вызова метода `Show` для объекта типа `TShape` инициирует ошибку во время работы программы (исключение `EAbstractError`), что совершенно логично. На то он и абстрактный метод, чтобы его нельзя было вызвать, компилятор контролирует это, страхуя нас от неприятностей.

В потомках мы должны переопределить метод `Show` обычным образом, используя директиву **`override`**.

```
TPoint = class(TShape)
private
   ...
public
   procedure Show(); override;
   ...
end;

procedure TPoint.Show;
begin
   { реализация }
...
end;
```

3.14. Внутренняя структура объекта. Методы класса. Динамический контроль типов, операторы IS и AS

Давно подмечено, что многие из нас с самого юного возраста обожают разбирать на запасные части все, что попадает в руки. Вы за собой такого не помните? Присмотритесь внимательнее к поведению ребенка, постепенно, с большим старанием отдирающего нос у какой-нибудь мягкой игрушки или пытающегося проникнуть внутрь игрушечного автомобиля, или с довольным видом и выражением полного счастья на лице вынимающего аккумулятор из сотового телефона родителей. Тяга к познанию окружающего мира незримо присутствует в каждом из нас.

К сожалению, формат данной книги не позволяет нам подробно, байт за байтом проанализировать содержимое переменных объектного типа и ассемблерный код текста программы, получающегося после того, как над ним на славу поработал компилятор. Справедливости ради надо сказать,

что эта информация является полезной для общего развития, но обычно не более того. Нам было бы довольно трудно привести понятный пример обозримого размера, иллюстрирующий необходимость знания того, по какому смещению находится ссылка на таблицу виртуальных методов и т.д. В конце концов, эти данные обычно не документированы, меняются от версии к версии компилятора, что делает их использование при программировании недопустимым (остается еще взлом программ, но мы надеемся, что вы не собираетесь этим заниматься). Поэтому мы ограничимся общим изложением, иллюстрирующим, как все устроено изнутри, и вполне, как мы считаем, достаточным для понимания принципов работы.

Итак, что такое объект? Во-первых, еще раз вспомним тот факт, что любой класс в языке Object Pascal является выведенным из класса `TObject`, если не указан другой предок. Класс `TObject` выполняет следующие основные функции:

1. Поддерживает унифицированный механизм для создания, обработки и уничтожения экземпляров объектов производных типов;
2. Осуществляет выделение, инициализацию и освобождение памяти для хранения экземпляров объектов;
3. Предоставляет информацию о типе объекта на этапе работы программы (RTTI – runtime type information);
4. Поддерживает механизм обработки сообщений (подробнее можно посмотреть в описании библиотеки VCL) и некоторые другие функции.

Из приведенного списка мы уже знакомы с пунктами 1 и 2, последний пункт выходит за рамки данной книги, поскольку относится не столько к языку программирования Object Pascal, сколько к реализации библиотеки визуальных компонентов VCL. В данном разделе мы будем говорить о третьем пункте – работа с информацией о типе объекта на этапе исполнения программы.

Итак, объект «изнутри» это:

1. Указатель на специальную структуру RTTI, содержащую информацию о том, к какому классу принадлежит объект и как этот класс устроен. Основными полями этой структуры являются:
   a) указатель на таблицу виртуальных методов с адресами методов;
   b) указатель на таблицу динамических методов с адресами методов;
   c) имя класса (в виде строки) и его размер;
   d) указатель на аналогичную RTTI-структуру предка;
   e) размер экземпляра (объекта).
2. Экземпляры предков со своими данными.
3. Данные объекта.

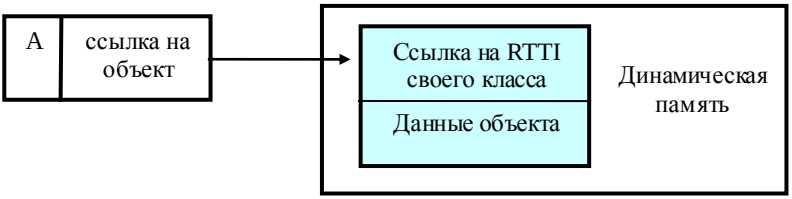

Рис. 10.19. Внутренняя структура объекта

Как видим, объект представляет собой совокупность данных и информации о своем классе. Как использовать данные объекта – мы уже знаем.

А как использовать информацию о классе?

Во-первых, эта информация сплошь и рядом используется автоматически. Так, таблицы виртуальных методов применяются в рамках работы механизма позднего связывания, таблицы динамических методов – при функционировании механизма обработки сообщений (см. описание библиотеки VCL) и т.д. Остальная информация тоже находит применение в программах. Мы рассмотрим один, самый важный из аспектов, связанный с использованием этой информации, – динамический контроль типов во время работы программы.

Заметим, что к части информации из RTTI можно получить доступ посредством вызова методов класса `TObject` (`ClassType`, `ClassInfo`, `ClassParent`). При этом вызов метода `ClassInfo` позволяет добраться до самой структуры RTTI, получив указатель на нее. Однако в документации не рекомендуется использовать непосредственно саму структуру, поскольку она может быть изменена при переходе к следующей версии Object Pascal. Отметим также тот факт, что указанные выше методы принадлежат к числу так называемых методов класса (в терминологии C++ – статических методов). Эти методы описываются с ключевым словом **`class`** перед **`procedure`** или **`function`**. Основной их особенностью является то, что они не получают указатель `Self` и, соответственно, не могут работать с полями объекта. Это методы, которые работают с данными класса, например с RTTI, как сделано в классе `TObject`.

Рассмотрим распространенную ситуацию.

Пусть мы имеем некоторую подпрограмму DoSomething, которая получает параметр объектного типа данных. Пусть эта подпрограмма такова, что мы не можем написать для этого параметра один конкретный тип данных, поскольку она должна осуществлять разные действия для разных типов данных. Посмотрим на следующий код:

```
procedure DoSomething(Obj: TObject);
var
   P: TPoint;
   S: TSquare;
begin
   if Obj IS TPoint then
   begin
      P := Obj AS TPoint;
      P.Hide;
   end
   else
      if Obj IS TSquare then
      begin
         S := Obj AS TSquare;
         S.Show;
      end;
end;
```

Код демонстрирует использование двух операторов языка – **IS** и **AS**.

Оператор **IS** осуществляет динамический контроль типов. Он проверяет, принадлежит ли пришедший в процедуру объект Obj тому или иному объектному типу данных. В момент срабатывания (не в момент компиляции, а в момент работы программы) у объекта Obj запрашивается информация о типе из RTTI, после чего производится сравнение с искомым типом данных (TPoint, TSquare в рассматриваемом примере). В результате **IS** возвращает значение **True** или **False**.

Оператор **AS** осуществляет безопасное преобразование типа на этапе работы программы. После того как **IS** подтвердил, что Obj – действительно объект типа TPoint, оператор **AS** производит приведение типа от TObject к TPoint, что позволяет вызывать для переменной P методы класса TPoint (напомним, что для Obj доступны лишь методы TObject).

Использование операторов **IS** и **AS** – важный способ обеспечения корректной работы программы.

В случае попытки некорректного преобразования оператор **AS** сгенерирует ошибочную ситуацию – исключение, которое может быть обработано в программе. В результате пользователь не увидит

сообщений типа «`Access violation`» или предложений обратиться к разработчику.

3.15. Выводы

Итак, мы закончили изложение материала по сложной и интересной теме – объектно-ориентированному программированию. К настоящему времени ООП из новой прогрессивной технологии превратилось в основной подход к разработке больших программных систем. Каждый программист сегодня должен владеть основными элементами этого подхода. К сожалению, нельзя объять необъятное, и наша книга, конечно, не претендует на абсолютную полноту изложения материала. Мы ставили своей целью рассмотреть основные элементы объектного подхода и их выражение в языке программирования Object Pascal. Некоторые частные возможности остались за кадром и могут быть изучены самостоятельно. К их числу относятся:

- ссылки на класс и виртуальные конструкторы;
- модификатор **`published`**, обработка событий (`events`);
- директива **`reintroduce`**.

Литература

1. Богатырев Р. Летопись языков Паскаль // Мир ПК. –№ 4.–2001.

2. Богатырев Р. От Паскаля к языку Zonnon: реализация новых идей на платформе .NET // Мир ПК.–2003.–№ 9.

3. Большая советская энциклопедия. Издание третье. – М.: Советская энциклопедия, 1970.

4. Брэдли Д. Программирование на языке ассемблера для персональной ЭВМ фирмы IBM.–М.: Радио и связь, 1988.

5. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Второе издание. – Бином, 1998.

6. Касперски К. Техника оптимизации программ. Эффективное использование памяти.– СПб.: БХВ-Петербург, 2003.

7. Кетков А., Кетков Ю. Практика программирования: Бейсик, Си, Паскаль. Самоучитель.–СПб.: БХВ-Петербург, 2001.

8. Кетков А., Кетков Ю. Практика программирования: Visual Basic, C++ Builder, Delphi. Самоучитель.–СПб.: БХВ-Петербург, 2002.

9. Кнут Д.Э. Искусство программирования. Т. 1. Основные алгоритмы.–М.:Вильямс, 2000.

10. Кнут Д.Э. Искусство программирования. Т. 3. Сортировка и поиск.–М.:Вильямс, 2004.

11. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы. Построение и анализ.–М.:МЦНМО, 1999.

12. Кэнту М. Delphi 7 для профессионалов.–Спб.:Питер, 2004.

13. Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования.–М.: Мир, 1982.

14. Модернизация и ремонт персонального компьютера.– [http://www.allcompinfo.com]

15. Новичков А. Система генерации проектной документации Rational SoDA.– [http://www.citforum.ru/programming/digest/soda.shtml]

16. Олифер В.Г., Олифер Н.А. Сетевые операционные системы.–СПб.: Питер, 2002.

17. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Пер. с англ.– 4-е изд.–СПб.: Питер; М.: Издательско-торговый дом «Русская Редакция», 2001.

18. Служба тематических толковых словарей. – [http://www.glossary.ru/]

19. Стивенс Р. Delphi. Готовые алгоритмы.– Спб.:Питер, 2004.

20. Тейксейра С., Пачеко К. Borland Delphi 6. Руководство разработчика. –Вильямс, 2002.

21. Шень А. Программирование: теоремы и задачи.–М.:МЦНМО, 2004.

22. Юров В. Assembler.–СПб.: Питер, 2002.

23. Cantu M. Essential Delphi.– [www.marcocantu.com/edelphi]

24. Cantu M. Essential Pascal.– [www.marcocantu.com/epascal]

25. Cantu M. Mastering Borland Delphi 2005.–Sybex, 2005.

26. Cantu M. Mastering Delphi 7.–Sybex, 2003.

27. Clark R., Koehler S. The UCSD Pascal Handbook.–Prentice-Hall, 1982.

28. Dahl O., Dijkstra E., Hoare C.A.R. Structured Programming.–London, England: Academic Press, 1972.

29. Gutknecht J., Zueff E. Zonnon Language Report. Draft.–ETH Zurich, June 2003.

30. Gutknecht J., Zueff E. Zonnon for .NET, A Language and Compiler Experiment. // Computer Systems Institute, ETH Zürich, Switzerland. JMLC Conference.–August 2003.

31. Gutknecht J. Zonnon: A .NET Language Beyond C# // Moscow: Microsoft Conference, June 15-17, 2003.

32. Gutknecht J., Wirth N. The Oberon System // Software – Practice and Experience.–Vol.19, № 9.–1989.–p.857-893.

33. Jacobson I., Christerson M., Jonsson P., Overgaard G. Object-oriented Software Engineering.–Workingham, England: Addison-Wesley Publishing Company, 1992.

34. Jensen K., Wirth N. PASCAL – User Manual and Report, ISO Pascal Standard.–1974.

35. Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture.– Intel Corporation: 1997.

36. Meyer B. Object-oriented Software Construction.–Prentice Hall, 1988.

37. Rumbaugh H., Blaha M., Premarlani W., Eddy F., Lorensen W. Object-Oriented Modeling and Design.–Prentice-Hall, 1995.

38. Sedgewick R. Algorithms.–Reading, MA: Addison-Wesley, 1983.

39. Tesler L. Object Pascal Report. // Structured Language World, Vol.9, No.3.– 1985.–p. 10-14.

40. Thiriez H. Modelling of an interactive scheduling system in a complex environment // European Jornal of Operational Research. –1991.– №50.–p.37-47.

41. Wirth N. Program Development by Stepwise Refinement // Communications of the ACM vol.26(1).– January 1983.

42. Wirth N. The Programming Language Pascal // Acta Informatica, 1.–1971.–p. 35-63.

43. Wirth N. Programming in Modula-2.–Springer-Verlag, 1982.

44. Wirth N. The programming language Oberon // Software – Practice and Experience, Vol.18, № 7.–1988.–p.671-690.

45. Wirth N. Programming in Modula-2.–Springer, 1974.

46. Wirth N. Pascal and its Successors // Conference on Computer Pioneers.–Bonn, 2001.

Заключение

Всякое наше знание ограничено,
лишь незнание бесконечно.

Пьер Симон Лаплас

Что ж! Пришла пора прощаться, уважаемый читатель. С сожалением вынуждены констатировать: наше с вами совместное путешествие подошло к концу.

В бурном море современных информационных технологий, как ни в какой другой области человеческой деятельности, справедлив принцип «все течет, все изменяется». Причем изменяется настолько быстро, что любой, кто хочет считаться специалистом в данной сфере, должен постоянно идти вверх, только чтобы оставаться на месте, ну а чтобы действительно двигаться, приходится бежать со всей скоростью. Другими словами, современный IT-специалист — это человек, постоянно пребывающей в состоянии изучения нового. Нам кажется, в этом состоит большой плюс нашей профессии!

Однако, несмотря на всю изменчивость и молодость компьютерного мира, уже существуют островки знания, признанные сообществом как классические, без овладения которыми стать профессионалом невозможно. Мы с вами на протяжении книги посетили и составили подробные карты нескольких таких островов: острова, где царят три алгоритмические конструкции; острова, где правят бал процедуры и функции, предпочитающие объединяться в библиотеки; и, наконец, острова, где в большом почете находится «его величество» Объект.

При этом главной нашей задачей было отнюдь не составление «географического» справочника, напротив, мы пытались показать, как на основе данных, извлекаемых путем анализа из постановки задачи, можно перекинуть через эти острова мост до самого главного — острова Работающей Программы. Насколько нам это удалось — судить вам.

В качестве прощального слова хотим выразить искреннюю надежду, что вы продолжите знакомство с компьютерными науками и что почерпнутые вами из данной книги знания помогут вам в этом процессе, а также пригодятся в дальнейшей профессиональной деятельности.