*Веками ковала мудрость ключи, подступаясь к человеку. Нарабатывала понятия, чтобы его объяснить. Время от времени приходил новый мудрец и с помощью нового ключа открывает тебе доступ к ещё неведомому.*

*Антуан де Сент-Экзюпери «Цитадель»*

# От структур и подпрограмм к объектам

О связи между структурным программированием и объектной технологией говорят очень нечасто и очень немного. Тем не менее, эта связь вполне очевидна, и её понимание позволяет оценить общие тенденции в области разработки программного обеспечения, а, следовательно, и роль объектной технологии. Именно поэтому цикл писем по объектной технологии начинается с рассмотрения её связи со структурным программированием.

Целями структурного программирования являлись структуризация данных и декомпозиция кода. Объединение данных в структуры позволяло, с одной стороны, более естественно описывать сущности реального мира, имеющие самые разнотипные наборы атрибутов. А, с другой стороны, делало процесс разработки программ более простым, поскольку однородные данные были сгруппированы в одном месте и под одним общим именем.

Декомпозиция кода заключалась в разделении исходного кода программы на отдельные подпрограммы. Подпрограммы решали две важные задачи: они снижали количество возможных связей между отдельными операторами (локализация кода) и, кроме того, позволяли устанавливать необходимую область видимости для переменных, используемых в подпрограммах (локализация данных). Локализация кода не позволяла произвольно переходить от одного оператора в одной подпрограмме к любому оператору в другой подпрограмме. Вызов подпрограммы приводил к передаче управления только в определённую точку входа, хотя некоторые языки допускали более одной точки входа в подпрограмму.

Такое решение не только повышало надёжность программ, но и, что более важно, позволяло многократно использовать одни и те же подпрограммы в различных программах. Как следствие, стали появляться библиотеки подпрограмм, как самостоятельные программные продукты. Часть библиотек поставлялась вместе с компиляторами с языков программирования и операционными системами, другая часть распространялась свободно или на коммерческой основе. Применение библиотек существенно ускоряло процесс разработки программного обеспечения, так как подпрограммы, применяемые многократно, требовалось отлаживать только единожды.

Локализация данных имела ничуть не меньшее значение, чем локализация кода. Подпрограммы имели дело с тремя видами данных:

- локальные данные подпрограммы, которые автоматически создавались при её вызове и уничтожались при возврате из подпрограммы;
- локальные данные, сохраняемые между несколькими вызовами одной подпрограммы;
- глобальные данные доступные нескольким или всем подпрограммам, и доступ к которым осуществлялся прямо из подпрограммы;
- фактические параметры, передаваемые подпрограммам, замещающие объявленные формальные параметры.

Примеры различных видов данных приведены в следующем фрагменте:

```
Locals      @@                  ; признак локальности данных и меток
```

```
DataSeg
Glob1       dd    0                 ; пример глобальных данных
Glob2       dd    100 dup (0)
      …
CodeSeg
Start:                              ; начало программы
      …
      ; при вызове подпрограмме ей передаются фактические параметры
      call  MyProc,     offset Glob1, eax
      …
Proc  MyProc
Arg   @@par1      :dword,           \ первый формальный параметр
      @@par2      :dword            ; второй формальный параметр
Local @@buf       :byte:16          ; локальная переменная, создаваемая
                                    ; при вызове подпрограммы
DataSeg
      @@loc1      dd    0           ; локальные переменные, сохраняющие
      @@loc2      db    80h dup (0) ; своё значение между вызовами
CodeSeg
      …                             ; тело подпрограммы
      ret
endp  MyProc
      …
end   Start
```

Глобальные данные применялись в основном для организации связей между подпрограммами, но они существенно ограничивали гибкость подпрограмм. Действительно, если подпрограмма работала с какими-то глобальными данными, то эти данные должны были совпадать по имени, типу во всех программах, которые использовали данную подпрограмму.

В отличие от этого, локальные данные, наоборот позволяли подпрограмме быть независимой от программ, использующих её. Но они не обеспечивали передачу данных между различными подпрограммами. Для передачи данных или ссылок на них использовался механизм формальных/фактических параметров. Об этом механизме надо сказать особо, так как его значение в структурном программировании велико.

Существо механизма формальных/фактических параметров состоит в том, что при описании подпрограммы объявляются формальные параметры, которые она принимает при вызове. Когда реально происходит вызов подпрограммы, на место формальных параметров подставляются фактические параметры того же типа. Это позволило, в частности, сделать глобальные переменные виртуальными. Действительно, в место формального параметра можно было передать значение любой переменной или ссылку на любую переменную, указанного в формальном параметре типа! Так, например, если у нас было два одинаковых массива, то механизм глобальных переменных не позволял одной и той же подпрограмме работать поочерёдно то с одним массивом, то с другим, поскольку внутри подпрограммы использовался физический адрес массива. Передача же ссылок на массивы в подпрограмму решила эту проблему. Подпрограмма теперь могла работать с любым массивом, достаточно было ей передать ссылку на нужный массив. Исходное положение массивов в памяти перестало играть какую-либо роль в работе подпрограммы. Виртуализация данных, передаваемых в подпрограмму, имеет самое непосредственное отношение к виртуализации самих подпрограмм. Но об этом поговорим чуть позже.

Образование библиотек подпрограмм происходило, как правило, по функциональному признаку. То есть, в одну библиотеку помещались подпрограммы, выполняющие близкую по составу работу. Например, библиотека подпрограмм, работающих с каким-то устройством, или библиотека подпрограмм, отображающих какие-то формы. Библиотеки подпрограмм, безусловно, были мощным стимулом в развитии программного обеспечения и преодолении его сложности. Однако они имели существенный недостаток. Как бы не был хорошо отлажен код подпрограмм, но результаты их работы могли быть неверны, если входные данные имели неправильные значения. Однако данные располагались в программе и могли изменяться произвольно. Любое неосторожное изменение данных могло непредсказуемым образом изменить результат работы программы. Появилась реальная потребность объединить и защитить данные и код, который их обрабатывает. Эта задача была решена в модульном программировании. Модуль, в отличие от библиотеки подпрограмм, мог содержать не только подпрограммы, но и данные.

Переход к модульному программированию был логичным следствием развития идеологии структурного программирования. Данные, располагаемые в модуле, были глобальными и для модуля и для программы, которая использовала данный модуль. А, следовательно, говорить о надёжности данных было преждевременно, поскольку данные по-прежнему могли быть изменены кодом, расположенным вне модуля. Для увеличения безопасности данных модуль разделили на несколько зон. Одна из зон обеспечивала взаимодействие между модулем и внешним программным обеспечением. Это, так называемая, зона интерфейса. Другая зона отвечала за реализацию модуля и была недоступна для внешнего программного обеспечения. Соответственно критичные к произвольным изменениям данные помещались в зону реализации и были недосягаемы для программы или других модулей. Про эти данные можно сказать, что они были инкапсулированы модулем.

В интерфейсной зоне оставались, как правило, только те данные, с которыми активно взаимодействовало внешнее программное обеспечение. Нельзя сказать, что эти данные можно было произвольно изменять без ущерба для работоспособности программы, но «прятать» все данные в зоне реализации означало, что подпрограммам, расположенным в модуле, было бы трудно взаимодействовать с другим программным обеспечением. Выход был найден в том, что практически все данные были инкапсулированы, то есть, помещены в зону реализации, а для взаимодействия с ними в интерфейсную зону помещались объявления специальных интерфейсных подпрограмм. Суть этих подпрограмм состояла в том, что только они могли изменять состояния переменных, расположенных внутри модуля, или возвращать значения этих переменных. Таким образом, можно было контролировать в модуле любые изменения данных и перехватывать некорректные действия до того, как они привели к разрушению программы или выдаче ошибочных результатов.

Однако перенос данных в зону реализации поставил ещё одну серьёзную проблему: данные необходимо было инициализировать перед началом работы и освобождать перед окончанием работы модуля. Например, перед началом работы модуля могла потребоваться динамически распределяемая область памяти, а в конце работы требовалось вернуть эту область памяти системе. Это привело к появлению в модулях областей инициализации, которые вызывались до начала работы модуля, и областей, которые автоматически вызывались перед тем, как модуль завершит работу.

Если теперь сложить все элементы полученной мозаики воедино, то получится знакомая картина: все подпрограммы модуля собраны вместе по функциональному признаку и работают над одними данными, инкапсулированными в модуль; модуль имеет области кода, которые вызываются при инициализации и окончании работы

модуля. Такой модуль можно назвать прообразом объекта. Но для получения более полного сходства необходимо вернуться к виртуализации, о которой говорилось выше.

Виртуализация данных посредством механизма формальных/фактических параметров раскрыла широкие возможности. Но её развитие сдерживалось строгой типизацией формальных/фактических параметров. На практике достаточно часто возникала необходимость применить одно и тоже действие по отношению к разным структурам данным. Проблему можно было разрешить очень просто: в подпрограмму в виде фактических параметров передавались тип структуры и ссылка на структуру. Внутри подпрограммы находился переключатель, который с учётом типа структуры выполнял требуемое действие. Можно рассмотреть следующий пример, иллюстрирующий данное решение:

```
enum  Figures     line, rectangle, circle
MaxFigure   =     circle
;объявляем типы
Struc tLine
      x           dd    ?     ; x - координата якорной точки
      y           dd    ?     ; y - координата якорной точки
      x1          dd    ?     ; x – координата конца линии
      y1          dd    ?     ; y – координата конца линии
ends  tLine

Struc tRectangle
      x           dd    ?     ; x – координата якорной точки
      y           dd    ?     ; y – координата якорной точки
      width       dd    ?     ; ширина прямоугольника
      height      dd    ?     ; высота прямоугольника
ends  tRectandle

Struc tCircle
      x           dd    ?     ; x – координата якорной точки
      y           dd    ?     ; y - координата якорной точки
      radius      dd    ?     ; радиус окружности
ends  tCircle

DataSeg
; создаём экземпляры типов, объявленных ранее
aLine       tLine       <20, 20, 10, 15>  ; линия
aRectangle  tRectangle  <10, 10, 20, 20>  ; прямоугольник
aCircle     tCircle     <20, 20, 10>      ; окружность

CodeSeg
Start:      ; начинаем программу
            …
            ; рисуем линию
            call  Draw, line, offset aLine
            …
            ;рисуем прямоугольник
            call  Draw, rectangle, offset aRectangle
            …
            ; рисуем окружность
            call  Draw, circle, offset aCircle
            …

Procedure Draw
Arg   @@figure    :dword,     \ это параметр типа фигуры
      @@ref :dword            ; это параметр ссылка на фигуру
```

```
DataSeg
Label @@vector    :dword                    ; создаём переключатель
            dd    offset @@draw_line        ; адрес кода рисования линии
            dd    offset @@draw_rectangle ; адрес кода рисования прямоугол.
            dd    offset @@draw_circle    ; адрес кода рисования окружности
CodeSeg
            mov   eax,[@@figure]            ; тип фигуры помещаем в eax
            cmp   eax,MaxFigure             ; проверка выхода за диапазон
            ja    @@type_mismatch           ; в случае выхода переход на
                                            ; обработчик ошибки
            mov   edx,[@@ref]               ; адрес структуры помещаем в edx
            jmp   [eax * 4 + offset @@vector]   ; переход на рисование
                                            ; фигуры заданного типа
@@exit:     ret                             ; возврат из подпрограммы

@@type_mismatch:                            ; обработка ошибки выхода
            …                               ; за диапазон
@@draw_line:                                ; рисование линии
            …
@@draw_rectangle:                           ; рисование прямоугольника
            …
@@draw_circle:                              ; рисование окружности
            …
endp  Draw
            …           ; продолжаем программу
end   Start             ; завершаем программу
```

Простой и эффективный переключатель, представленный в примере, позволяет связать виртуальность данных с кодом. Подпрограмме требуется всего несколько операторов, чтобы проверить правильность типа фигуры и перейти к заданному обработчику. Название «переключатель» заимствовано из языков высокого уровня и соответствует операторам switch-case в C или case в Pascal. Качественный компилятор с языка высокого уровня в подобной ситуации должен привести эти операторы к подобной конструкции, поскольку она является наиболее эффективной и компактной.

Недостатком данного решения с точки зрения структурного программирования является передача в подпрограмму ссылки на структуру произвольного типа. Это потенциальный источник ошибок. Например, возможна ситуация передача в подпрограмму ссылки на структуру некоторой фигуры, обработчик которой не был определён в подпрограмме. С точки зрения формальных параметров нарушения не будет, но результат работы программы может быть неожиданным. Однако реальная проблема лежит глубже.

В данном примере предполагается, что подпрограмма умеет работать с некоторыми геометрическими фигурами, в частности, рисовать их. Но для работы программы одного рисования может оказаться недостаточным. Вполне возможно, что фигуры потребуется выделять при выборе, перемещать, скрывать и т.п. Конечно, не так трудно реализовать эти действия по отношению к заданному множеству геометрических фигур, воспользовавшись подпрограммой Draw как шаблоном. Однако, что произойдёт в случае, если потребуется ввести в программу новую геометрическую фигуру? Прежде всего, потребуется серьёзно переделать и перекомпилировать всю программу. Нам придётся:

1. Добавить новую фигуру в тип перечисление Figures.
2. Изменить значение константы MaxFigures.
3. Описать структуру атрибутов новой фигуры.
4. Создать статически или динамически экземпляр новой фигуры.

5. В каждой подпрограмме, выполняющей некоторое действие над фигурой, необходимо добавить:
   a. новый элемент в вектор обработчиков, с указанием адреса обработчика;
   b. написать собственно сам обработчик действия для новой фигуры;
6. Перекомпилировать программу.

Процесс переделки и перекомпиляции рабочей программы является весьма болезненным даже для относительно небольших программ. Это обусловлено тем, что часть исходных текстов могла быть утрачена или изменена; могла смениться версия компилятора и при этом новая версия оказалась не полностью совместимой с предыдущей версией; и т.д. и т.п. Поэтому было бы желательно избежать лишних изменений в исходных текстах и, если это возможно, то исключить процесс перекомпиляции всей программы.

Для решения задачи попробуем несколько сместить точку зрения. До сих пор, реализовывалась виртуальность фигур в рамках действий над этими фигурами. Теперь попробуем реализовать виртуальность действий по отношению к фигурам, ведь, в конце концов, нам надо реализовать матрицу из множества действий для множества фигур. Поэтому переключающие вектора теперь извлечём из подпрограмм, реализующих действия, и разместим вне этих подпрограмм. Содержимым векторов по-прежнему останутся конкретные обработчики для конкретных фигур. Представим это решение:

```
enum  Figures     line, rectangle, circle
MaxFigure   =     circle
enum  Actions     Draw, Hide, Select, Drag
MaxActions  =     Drag

;объявляем типы
Struc tLine
      x           dd    ?     ; x - координата якорной точки
      y           dd    ?     ; y - координата якорной точки
      x1          dd    ?     ; x – координата конца линии
      y1          dd    ?     ; y – координата конца линии
ends  tLine
TYPEDEF     LineRef     PTR   tLine

Struc tRectangle
      x           dd    ?     ; x – координата якорной точки
      y           dd    ?     ; y – координата якорной точки
      width       dd    ?     ; ширина прямоугольника
      height      dd    ?     ; высота прямоугольника
ends  tRectangle
TYPEDEF     RectangleRef      PTR   tRectangle

Struc tCircle
      x           dd    ?     ; x – координата якорной точки
      y           dd    ?     ; y - координата якорной точки
      radius      dd    ?     ; радиус окружности
ends  tCircle
TYPEDEF     CircleRef   PTR   tCircle

DataSeg
; создаём вектора-переключатели (virtual tables)
Label       vectorLine        dword
            dd    offset      LineDraw
            dd    offset      LineHide
```

```
            dd    offset      LineSelect
            dd    offset      CommonDrag

Label       vectorRectangle   dword
            dd    offset      RectangleDraw
            dd    offset      RectangleHide
            dd    offset      RectangleSelect
            dd    offset      CommonDrag

Label       vectorCircle      dword
            dd    offset      CircleDraw
            dd    offset      CircleHide
            dd    offset      CircleSelect
            dd    offset      CommonDrag

; создаём экземпляры типов, объявленных ранее
aLine       tLine       <20, 20, 10, 15>  ; линия
aRectangle  tRectangle  <10, 10, 20, 20>  ; прямоугольник
aCircle     tCircle     <20, 20, 10>      ; окружность

CodeSeg
Start:      ; начинаем программу
            …
            ; рисуем линию
            call  [Draw * 4 + vectorLine], offset aLine
            …
            ; прячем прямоугольник
            call  [Hide * 4 + vectorRectangle], offset aRectangle
            …
            ; выбираем окружность
            call  [Select * 4 + vectorCircle], offset aCircle
            …

Proc  LineDraw
arg   @@ref :LineRef
            ; рисуем линию
            ret
endp  LineDraw

Proc  LineHide
arg   @@ref :LineRef
            ; прячем линию
            ret
endp  LineHide

Proc  LineSelect
arg   @@ref :LineRef
            ; выбираем линию
            ret
endp  LineSelect

Proc  CommonDrag
arg   @@ref :dword
            ; перетаскиваем фигуру
            ret
endp  CommonDrag

Proc  RectangleDraw
```

```
arg   @@ref :RectangleRef
            ; рисуем прямоугольник
            ret
endp  RectangleDraw

Proc  RectangleHide
arg   @@ref :RectangleRef
            ; прячем прямоугольник
            ret
endp  RectangleHide

Proc  RectangleSelect
arg   @@ref :RectangleRef
            ; выбираем прямоугольник
            ret
endp  RectangleSelect

Proc  CircleDraw
arg   @@ref :CircleRef
            ; рисуем окружность
            ret
endp  CircleDraw

Proc  CircleHide
arg   @@ref :CircleRef
            ; прячем окружность
            ret
endp  CircleHide

Proc  CircleSelect
arg   @@ref :CircleRef
            ; выбираем окружность
            ret
endp  CircleSelect

            …       ; продолжение программы
end   Start         ; завершение программы
```

Можно написать набор макроопределений, который бы позволил статически и динамически создавать фигуры и вектора-переключатели, а также вызывать подпрограммы. Это задача достаточно тривиальная, например, можно посмотреть те макроопределения, которые фирма Borland (теперь Inprise) предложила в TASM 3.0.

Гораздо важнее другое, теперь при добавлении новой фигуры не потребуется изменять существующие исходные тексты, достаточно просто добавить это описание к существующей программе. Мало того, достаточно легко создать run-time модуль, который позволит создавать и подключать новые фигуры прямо во время работы. Однако есть определённое неудобство в том описании структур фигур, которое приведено выше. Во время исполнения неизвестно какой фигуре принадлежит та или иная структура. Пока есть возможность работать с фигурами статически, особых проблем не возникает, но при динамичном создании структур и взаимодействии с ними, проблемы возникнут непременно. Самый простой способ решения этой проблемы состоит в том, чтобы включить описатель типа структуры непосредственно в саму структуру. Например,

```
Struc tLine
      kind        dd    ?     ; тип фигуры
      x           dd    ?     ; x - координата якорной точки
      y           dd    ?     ; y - координата якорной точки
```

```
      x1          dd    ?     ; x - координата конца линии
      y1          dd    ?     ; y - координата конца линии
ends  tLine
```

В поле «kind» можно поместить уникальный идентификатор той фигуры, к которой относится данная структура данных. Но обычно поступают проще, помещая в это поле ссылку на вектор-переключатель фигуры. В этом случае вызов подпрограмм, реализующих действия, для любой фигуры превратиться в следующую последовательность действий:

```
      ; пусть eax содержит ссылку на структуру, тогда
            push  eax
            ; здесь передаём в стек остальные параметры
            …
            mov   eax,[eax]          ; адрес вектора-переключателя
                                     ; поместим в eax
            call  [eax + Action4]    ; вызовем нужную подпрограмму
                                     ; (Action4 = Action * 4)
```

Другое важное отличие второго решения состоит в том, что в подпрограммы, реализующие некоторые действия, теперь можно передавать типизированные ссылки на конкретные структуры вместо произвольных ссылок. В этом случае компилятор с языка высокого уровня может взять на себя проверку правильности типов данных. Мало того, ссылку можно передавать неявно, избегая, тем самым, возможных ошибок программиста.

Вектора-переключатели обычно называют виртуальными таблицами, но в данном изложении мне казалось, что стоит повременить с переименованием, дабы как можно дольше сохранять связь с операторами языков высокого уровня таких, как switch-case (в C) или case (в Pascal). Сохранение этой связи преследовало вполне конкретную цель: показать связь между объектной технологией и технологией структурного программирования. Но теперь имеет смысл перейти к общепринятому названию векторов-переключателей – виртуальные таблицы. Таким образом, виртуальные таблицы, связывающие воедино структуры данных и код, взаимодействующий с ними, являются естественным решением при переходе от структурного программирования к объектной технологии. Конечно, виртуальные таблицы - не единственное и даже не лучшее решение, и об этом пойдёт речь в последующих письмах, но на сегодня это решение является весьма распространённым.

Подпрограммы, реализующие некоторые действия, обычно называют методами. Структура данных, связанная с методами, представляет собой объект, а описание структуры и методов обычно называют классом. Таким образом, объект является экземпляром класса. Методы класса, видимые для внешнего программного обеспечения, составляют интерфейс класса. Но, следует отметить, что интерфейс класса не всегда соответствует методам, как подпрограммам. Способ реализации интерфейса может определяться объектной средой.

Сейчас хотелось бы отметить, что мы вплотную подошли к возможности реализации объектной технологии. Фактически осталось только ввести поддержку полиморфизма и наследования, что сегодня, как правило, реализуется в рамках компиляторов. Введение поддержки этих парадигм не вызывает существенных сложностей. Однако наиболее важно понимать другое, то, что объектная технология, являясь прямым продолжением технологии структурной разработки программ, открыла новые принципиально иные подходы к проектированию и разработке сложного программного обеспечения. К сожалению, эти подходы пока очень редко применяются в практике и не отражены в литературе.