Искусство алгоритмизации

В. Потопахин

ДМК
издательство

УДК 004.421
ББК 32.973.26-018
П64

Потопахин В. В.

П64 Искусство алгоритмизации. – М.: ДМК Пресс, 2011. – 320 с.: ил.

ISBN 978-5-94074-621-8

Эта книга для тех, кто хорошо, владея языком программирования и устойчивыми навыками решения задач, желает наработать свой программистский инструментарий. В книге, неформально и довольно детально, разобран значительный набор алгоритмов и методов. Большая часть представленных алгоритмов доведена до реализации на языке Компонентный Паскаль. Для большей прозрачности изложения реализация выполнена пошагово с четкой формулировкой задач каждого шага и записью программного фрагмента. Изложение сопровождается заданиями для самостоятельной работы, количество и сложность которых достаточны для хорошего усвоения материала. Требования к математическим знаниям минимальны, некоторые важные математические понятия и темы кратко изложены в приложении.

УДК 004.421
ББК 32.973.26-018

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.
Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-94074-621-8

© Потопахин В., 2010
© Оформление, издание, ДМК Пресс, 2011

Содержание

Введение 6

Глава1. Парадигма структурного программирования 9
Зачем нужны общие принципы? 10
Нисходящее проектирование 12
Три базовых элемента структурного программирования 14
Пример разработки 17

Глава 2. Вычислительные алгоритмы 26
Моделирование непрерывных процессов дискретными 27
Метод половинного деления. Общая задача поиска величины 31
Метод касательных 34
Метод хорд 35
Метод итераций (последовательных приближений) 36
Обобщение метода половинного деления 37
Метод наименьших квадратов 38
Задача вычисления площадей криволинейных фигур 42
Метод Симпсона 45
Метод Монте-Карло 48

Глава 3. Числовые алгоритмы 54
Алгоритм Евклида 55
Алгоритмы факторизации и поиска простых 57
Выделение полного квадрата (алгоритм Ферма) 58
Квадратичное решето 60
Алгоритм Полларда 66
Алгоритмы поиска простых чисел 69
Решето Аткина 71
Решето Сундарама 72
Тесты простоты 73
Числа Мерсенна 75
Тест Люка-Лемера 76
Числа Ферма 78
Тест Пепина 78
Псевдослучайные числа 78
Критерии правильности случайных чисел 81
Критерий, основанный на квадратичном отклонении 81
Линейный конгруэнтный метод 81
Методы перемешивания 85

Глава 4. Арифметика 89
Представление числа в позиционной системе счисления 90
Проблемы технической реализации арифметики 93
Двоичный сумматор 94
Ускорение операции сложения 95
Представление чисел в форме с фиксированной и плавающей десятичной точкой 96
Реализация арифметики на уровне алгоритмического языка 97
Сложение двух чисел 97
Вычитание из большего меньшего 99
Умножение 102
Деление 107
Некоторые другие алгоритмы 115
Алгоритм быстрого возведения в степень 115
Быстрый перевод из десятичной в двоичную систему счисления 116
Решение диофантовых уравнений 117
Двоичная арифметика 119
Сложение двоичных чисел 120
Как преобразовать в двоичное число дробную часть 122
Вычитание двоичных чисел 124
Умножение в двоичной системе счисления 125
Деление в двоичной системе счисления 126

Глава 5. Рекурсия и динамическое программирование 131
Общее определение 132
Задача о ханойской башне 135
Переход от рекурсивного к нерекурсивному решению 138
Рекурсия как метод поиска 143
Динамическое программирование 144
Задача обхода конем шахматной доски 146
Факторизация числа 153

Глава 6. Сортировки 166
Общая постановка задачи 167
Обменные сортировки. Сортировка пузырьком 168
Шейкерная сортировка 170
Анализ качеств алгоритма 171
Сортировка выбором 174
Сортировка вставками 176
Сортировка Шелла 178
Быстрая сортировка 181
Двоичная сортировка 186
Сортировка слияниями 191

Глава 7. Комбинаторные задачи 204
Общая постановка задачи 205
Оптимизация перебора 207
Связь комбинаторики с алгоритмами на графах 209
Основные комбинаторные задачи 210
Задача получения перестановок на множестве из N элементов 210
Построение сочетаний без повторений на множестве элементов 216
Сочетания с повторениями 221
Задача получения размещений 223

Глава 8. Динамические структуры данных 224
Понятие о динамической величине 225
Линейный связный список 226
Зачем рекурсивные структуры нужны? 229
Использование рекурсивных определений для создания деревьев данных 233

Глава 9. Алгоритмы принятия решений 237
Постановка задачи. Понятие эвристического алгоритма 238
Оценочная функция 240
Метод минимакса 241
Альфа-бета алгоритм 245

Глава 10. Алгоритмы на графах 250
Стратегии обхода 251
Обход графа в ширину 251
Обход графа в глубину 253
Построение остовного дерева 253
Алгоритм Прима 254
Алгоритм Краскала 258
Алгоритм поиска компонент связности 263
Волновой алгоритм 265
Алгоритм Дейкстры 269
Алгоритм Флойда 276
Нахождение максимального потока 280

Глава 11. Приложения 296
Приложение 1. Элементы комбинаторики 297
Приложение 2. Теория графов 301
Приложение 3. Элементы теории вероятности 309
Приложение 4. Синтаксис языка Компонентный Паскаль 315

Список литературы 319

Глава 1

Парадигма структурного программирования

Зачем нужны общие принципы.....10
Нисходящее проектирование.....12
Три базовых элемента структурного программирования..... 14
Пример разработки..... 17

Зачем нужны общие принципы?

Попробуем понять, что такое хорошая программа. Очевидно, этот вопрос надо решить дважды: во-первых, с точки зрения пользователя, так как программа в конечном итоге делается для него, и во-вторых, с точки зрения программиста, так как он, наверное, не может быть безразличен к качеству своего продукта. Пользователь программного обеспечения ожидает, что результат будет получен гарантированно и за приемлемое время. Еще одно, важное требование – это стоимость ПО. Если пользователь платит за программу, то естественно, он желает, чтобы стоимость была по возможности низкой. Это почти все. Различные программистские вопросы, вроде того, на каком языке написана программа, как она устроена, сколько в её разработку вложено программистской энергии, его не интересуют.

Важное замечание. Есть один исключительно важный момент, выпадающий из логики дальнейшего изложения, но пропустить который нельзя. Это вопрос надежности ПО. Грубо говоря, нажимая на кнопку, мы ожидаем вполне определенный результат, а не какой-нибудь. Иногда это вопрос больших материальных потерь и даже человеческих жизней. И надежность обеспечивается отнюдь не талантом программиста, а скорее выбором и строгим соблюдением общих принципов и правил.

Позиция программиста в вопросе оценки качества программы куда более сложная. Например, программист желает получить заданный результат с минимальными усилиями. Отчасти из такого желания и возникает понятие технологии. В промышленности из желания делать много малыми усилиями возникли разделение труда и конвейер. В программировании результатом такого желания можно считать понятие процедуры, коллекционирование процедур в библиотеки и сборку новой программы из уже готовых процедур. С этой точки зрения задачу программирования можно сформулировать так:

Определите, какие процедуры необходимы, и соберите из них новую процедуру, реализующую поставленную задачу.

Так звучит парадигма процедурного программирования. Конечно же компоновать программу из процедур и языковых конструкций можно по-разному. Процедурная парадигма ничего не говорит о том, как организовать процесс программирования, она лишь определяет процедуру главным строительным блоком. Вопрос, что такое хорошо и что такое плохо, с этой позиции остается открытым.

Анализ с точки зрения функциональности программы мы опустим, это можно отнести к ожиданиям пользователя, разговор о которых уже закрыт. С точки же зрения программиста существуют еще два важных фактора: скорость написания программы и её читаемость. Читаемость – собственно то же фактор скорости, скорости понимания программы.

Конечно, лучше всего было бы быстро писать и легко читать. Но, к сожалению, в реальном программировании приходится искать золотую середину, отдавая предпочтение либо скорости написания, либо читаемости. Что важнее? На первый взгляд может показаться, что скорость важнее. Чем быстрее мы напишем программу, тем меньше потратим времени и энергии. Но это только на первый

взгляд. Во-первых, любой серьезный проект не пишется за одну попытку. Сначала создается его первая, рабочая версия, которая затем развивается и улучшается, быть может, до бесконечности. Для развития и улучшения программа должна читаться, и иногда не теми людьми, которые писали первую версию. Следовательно, все-таки программа должна быть читаемой.

Даже если программу пишет один программист, то и он делает её не за один присест. Скорее всего, на разработку уходит значительное время, в течение которого у программиста может возникнуть потребность вернуться к тому или иному фрагменту, что-то вспомнить, что-то уточнить. То есть и для автора первой версии программа должна быть читаемой. Сказанное можно выразить так:

Программа пишется один раз, а читается многократно. Поэтому читабельная программа сокращает время разработки значительно эффективнее, чем «быстро написанная».

Необходимо ответить на вопрос: а как писать код, чтобы он был легко читаем? Взглянем на программу немного с другой стороны. Исполнение программы можно рассматривать как процесс передачи управления, а текст – соответственно, как описание последовательности передачи управления. Отсюда следует простая и естественная идея. Читаемая программа – это программа с максимально простой последовательностью передачи управления. Наиболее простая структура управления – это линейная последовательность программных блоков. Для того чтобы можно было выстроить линейную последовательность, необходимо и достаточно, чтобы каждый логически замкнутый блок имел один вход и один выход. Программа, построенная из таких блоков, имеет, очевидно, структуру простейшую из возможных, и такая программа называется структурной.

Структурность может быть, и как правило, бывает многоуровневой. Блоки линейной структуры могут оказаться достаточно крупными. В этом случае каждый такой блок должен допускать представление в виде линейной структуры блоков меньшего размера и т. д.

Почему программа, устроенная таким образом, будет читаемой? Ответ, видимо, лежит в области психологии человеческого восприятия. Когда мы смотрим на картину, то сначала воспринимаем крупный план, затем детали. Также работает и наше мышление. Есть крупный план проблемы, есть подзадачи, которые можно рассматривать последовательно одну за другой. Также анализируется и программа. Сначала крупный план – программа как последовательность логически законченных блоков, затем каждый блок – процедура как самостоятельная задача.

Идея структурирования программы последовательностью логически замкнутых блоков выглядит естественной. Поэтому может возникнуть вопрос зачем вообще об этом говорить. Есть много естественных вещей, и им никто не учит. Ведь действительно, никто не будет строить дом с крыши. Однако в программировании ситуация несколько хитрее. Естественно не значит просто и совершенно не означает, что структурное программирование дается каждому с лету. Структурное программирование предполагает дисциплинированный ум, умеющий хорошо ор-

ганизовать свою работу. Однако дисциплина ума, технологичность мышления – это то, что подлежит развитию. Поэтому структурное программирование – это не просто свод правил, как правильно писать программу, это прежде всего некоторая технология мыслительной деятельности.

Нисходящее проектирование

Сейчас речь пойдет не просто о написании программы. Речь пойдет о поиске решения задачи, и написание программы в этом процессе есть только этап, пусть и наиболее ощутимый, но все же только этап. Решение же предполагает многие вещи. Например, поиск математического решения, выбор структур данных, разработку алгоритмов. Это замечание нам нужно для того, чтобы обезопасить себя от узкого понимания вопроса *„что значит решить программистскую задачу?"* К сожалению, очень часто это действие сводят к написанию некоторого кода, что, конечно, неправильно.

А сейчас главная идея. Большая программистская удача заключается в том, что любая достаточно серьезная задача не представляет собой логически монолитного куска гранита. Скорее, это набор камней, уложенных в определенную конфигурацию. А если не увлекаться аналогиями, можно сказать, что программистская задача допускает разбиение на подзадачи, каждая из которых формулируется независимо от Большой Задачи и от других подзадач и соответственно решается так, как будто других подзадач и Большой Задачи просто не существует. После решения всех подзадач решение Большой Задачи компонуется из полученных меньших.

Такой процесс разбиения называется **ДЕКОМПОЗИЦИЕЙ**. Декомпозиция может быть многоуровневой, каждая из полученных подзадач также может оказаться достаточно большой, и тогда к ней тоже можно применить операцию декомпозиции.

Последовательное применение операции декомпозиции к подзадачам различного уровня называется нисходящим проектированием.

Рассмотрим простой пример. Дано уравнение вида

$$\sum_{k=0}^{n} a_k x^k = 0.$$

Определить все его целые корни.

Решение. Алгебра дает нам нужный для решения факт: все целочисленные корни являются делителями свободного члена, то есть величины a_0. Следовательно, решение Большой Задачи должно заключаться в переборе всех делителей числа a_0 и выяснении, какие из них являются корнями уравнения. Отсюда и очевидное разбиение:

Задача 1. Дано некоторое целое число. Найти все его делители.

Задача 2. Дан многочлен вида

$$\sum_{k=0}^{n} a_k x^k = 0$$

и некоторое значение x, вычислить значение многочлена.

Легко увидеть, что сформулированные задачи логически независимы. Вторая задача нужна для расчета значения многочлена от делителя, но то, что число x является делителем, ровным счетом никого ни к чему не обязывает. Если мы напишем процедуру, считающую значение многочлена от любого x, то, естественно, мы сможем вычислить и его значение от x, являющегося делителем.

Предположим, что используемая среда программирования не умеет считать степень числа, тогда есть смысл вторую подзадачу разбить еще на две:

Задача 2.1. Вычислить сумму ряда $A_1 + A_2 + ... + A_n$.

Задача 2.2. Вычислить величину $A_k = a_k x^k$.

Но, в общем-то, мы выполнили декомпозицию второго уровня только для того, чтобы показать, что это возможно. Глубина декомпозиции должна определяться прежде всего из соображений разумности. Подзадача вычисления степени величины, даже для не слишком опытного программиста, не должна создавать проблем, и специально её проектировать нет необходимости. Задача поиска делителей числа решается так:

Листинг 1.1

```
FOR x:=1 TO B DO
  IF B MOD x=0 THEN
    StdLog.Strng('Очередной делитель =');StdLog.Int(x);
  END;
END;
```

Данный фрагмент решает поставленную задачу. Предположим теперь, что вторая задача также решена и реализована в виде функции **Расчет**. Для того чтобы готовый фрагмент объединить с функцией **Расчет**, необходимо принять решение о том, что функция получает на входе и что она дает в качестве результата. На выходе необходимо сообщение, является ли некоторое число корнем. Поэтому разумно возвращать из **Расчет** логическое значение (TRUE – корень, FALSE – не корень). Для работы функции потребуется массив коэффициентов многочлена и значение x. Величина x в нашем фрагменте определена, единственное – уточним, что проверке подлежат два числа: x и $-x$. И фрагмент можно переписать так:

Листинг 1.2

```
FOR x:=1 TO B DO
  IF B MOD x=0 THEN
    IF Расчет (a,x) THEN
```

```
      StdLog.String('Очередной корень =');StdLog.Int(x);
    END;
    IF Расчет (a,-x) THEN
      StdLog.String('Очередной корень =');StdLog.Int(-x);
    END;
  END;
END;
```

Обратите внимание, мы использовали процедуру – функцию **Расчет**, не написав для неё ни одной строчки кода. Это означает, что две наши подзадачи действительно независимы. Первая подзадача использует вторую, но ей все равно, как вторая устроена внутри. Является ли написанный фрагмент структурным? Да, безусловно. Его верхний уровень – это один цикл. Следующий уровень – это единственный оператор IF, записанный в структуре цикла, и наконец, тело оператора IF B MOD x=0 THEN – две последовательно выполняемые проверки. Завершив написание фрагмента, программист вполне может приниматься за написание функции **Расчет**, не оборачиваясь на сделанное.

В наши цели входило только привести простой пример декомпозиции, поэтому доводить работу до завершенной программы не будем.

Три базовых элемента структурного программирования

Приведенный пример несет в себе еще один важный момент. Структурная программа – это последовательность блоков с одним входом и одним выходом, но представлять такой блок только как линейную последовательность операторов не содержательно. Любой язык программирования содержит в себе конструкции циклов и конструкции выбора. Этот факт отражен в структурной парадигме вводом трех видов структурных блоков. Первый (рис. 1.1) – это простой блок, второй (рис. 1.2) – это конструкция цикла, и третий (рис. 1.3) – конструкция выбора.

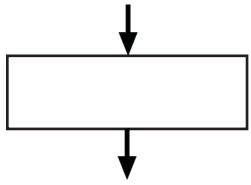


Рис. 1.1

В своей простейшей форме простой блок – это последовательность операторов присваивания. В общем виде простой блок – это процедура или любой программный фрагмент, который можно отделить от остальной программы и определить для него только ему присущий смысл. Простой блок может иметь сколь угодно сложную внутреннюю структуру. Существенно важно лишь то, что есть только одна точка в его тексте, в которой ему передается выполнение, и есть только одна точка, в которой его выполнение завершается.

Цикл – это сложная конструкция, описывающая процесс многократного выполнения простого блока. Обратите внимание: в блок-схеме речь идет об условии продолжения. Говоря в терминах Компонентного Паскаля, здесь описана форма цикла WHILE. Разумеется, это не означает недопустимости других форм. Необхо-

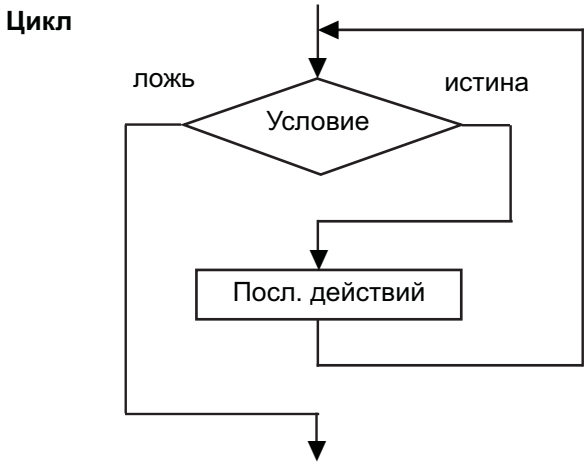


Рис. 1.2

димо понимать, что прежде всего здесь описана не определенная форма цикла, а принципиальная возможность многократного выполнения.

Для конструкции условного перехода (конструкции выбора), так же как и для цикла отметим, что любой существующий язык программирования предлагает существенно больше возможностей для организации ветвлений. Но любая форма ветвления представима в виде комбинации структур, описанных на рис. 1.3.

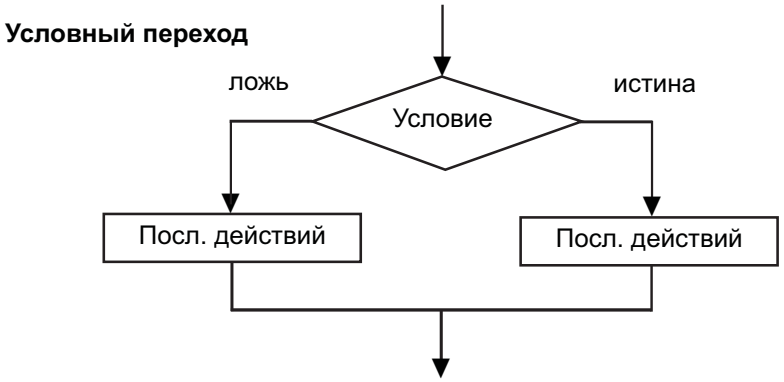


Рис. 1.3

Три приведенных своими блок-схемами конструкции и есть **единственные** блоки для построения структурных программ. Обратите внимание на выделенное слово. То, что эти блоки единственные, момент принципиальный.

Идея структурного программирования заключается в максимальной простоте при достаточной функциональности, именно поэтому структурные программы прозрачны для понимания и легко читаемы.

Конечно, можно искусственно ввести какие-то дополнительные конструкции для увеличения функциональности, но, наверное, вы согласитесь, что любая палка о двух концах. Потеря читабельности сведет на нет все достижения в области функциональности. Конечно же, сказанное не означает запрета на поиск новых, эффективных конструкций. Это лишь требование осторожности в таком поиске. Примеры изобретения более мощных структур конечно же есть. Например, так называемый цикл Дейкстры.

Листинг 1.3

```
WHILE условие цикла DO
   последовательность операторов
{
ELSIF условие  DO
   Последовательность операторов
}
END;
```

Фигурные скобки означают, возможно, многократное повторение конструкций ELSIF. Смысл цикла – в том, что сложная конструкция из вложенных циклов и проверок выстраивается на одном уровне, это если не упрощает логику алгоритма, то во всяком случае делает более прозрачной запись программного текста. В КП нет конструкции, непосредственно реализующей цикл Дейкстры, но его легко реализовать циклом LOOP:

Листинг 1.4

```
LOOP
    IF логическое выражение THEN
      Последовательность операторов
   {
    ELSIF логическое выражение THEN
      Последовательность операторов
    }
    ELSE EXIT
    END;
END;
```

Структурное программирование и нисходящее проектирование позволяют сделать процесс разработки ПО высокотехнологичной дисциплиной, что жизненно важно для программирования вообще и промышленного программирования в частности.

Как ни странно, талантливый программист, интуитивно находящий красивые решения, может оказаться фактором повышенной опасности для промышленного проекта. Дело в том, что программистская работа в массе своей является работой коллективной, и зачастую не столько важна красивость решения, сколько сроки, в которые оно получено, возможность его анализа, совместимость его с другими частями проекта. То есть работа программиста должна подчиняться строгой мыс-

лительной дисциплине, а структурное программирование вкупе с идеей нисходящего проектирования, по большому счету, и есть основа для такой дисциплины. Разумеется, сказанное ни в коем случае нельзя воспринимать как попытку принизить личный талант. Талантливый специалист всегда лучше бесталанного. Разумеется только то, что личный талант – это лишь основа, нуждающаяся в жестком дисциплинирующем каркасе.

Замечание об операторе GOTO. Альтернатива – структурное или неструктурное программирование при переходе к языку программирования отчасти переходит в альтернативу: использовать или не использовать оператор безусловного перехода GOTO. В языке Компонентный Паскаль этот оператор отсутствует, но, как правило, языки программирования проектируются с этим оператором, есть он и в диалектах языка Паскаль. GOTO – это главная возможность «грубого» прерывания хода исполнения программы и передачи управления почти в любое место её текста без проверки каких-либо условий. Этот оператор и обеспечивает возможность появления программных блоков с более чем одним входом и более чем одним выходом. Спор о том, нужен или нет оператор GOTO, не имеет большого смысла. Каждый программист должен ответить для себя на более фундаментальный вопрос: будет ли его программа структурной, если да, то потребности в GOTO просто нет. Если же программист допускает появление неструктурных элементов, то, естественно, он допускает и появление в своих программах GOTO.

Пример разработки

Для построения примера возьмем известную и очень сложную задачу разработки расписаний. Естественно, мы её ограничим и будем говорить не о любых расписаниях, а о расписании учебных занятий в учебном заведении.

Данная задача сложна не только с идейной стороны. Она очень трудоемка, и детальное её описание может потребовать слишком много места. Поэтому договоримся прежде всего об упрощении структуры данных и не будем требовать от алгоритма реального успеха в любых условиях. Мы будем использовать алгоритм достаточно разумный и достаточно интересный, но если кто решит попробовать довести его до коммерческой версии, он должен быть готов к очень большой работе.

Краткое описание идеи

Если говорить о структурах данных, на которых решается задача, то их две: во-первых, набор занятий, и, во-вторых, набор аудиторий, в которых занятия можно проводить. Занятие можно идентифицировать именем или номером. С аудиторией дело обстоит несколько сложнее. Объектом распределения является не сама аудитория, а время, в течение которого аудиторию можно занять. Договоримся отрезок времени, в течение которого аудитория может быть занята одним занятием, условно называть вакансией. Тогда вместо множества аудиторий есть смысл рассматривать множество вакансий.

Структура составляемого расписания должна удовлетворять некоторому набо-

ру требований. Эти требования можно разбить на два класса. Назовем их условно требования класса *A* и требования класса *B*. Требования класса *A* определяют принципиальное соответствие занятия и вакансии. Это, например, требование вместимости, аудитория должна иметь достаточное количество посадочных мест. Аудитория должна иметь требуемое оборудование, например для лекции нужна доска, для лабораторной работы – соответствующие приборы и т. д. Сколько таких требований описывается для расписания и как их формализовать, сейчас не важно. Заметим только, что полный набор требований составляется к каждому занятию, и набор этот определяет на множестве вакансий подмножество, которое назовем областью определения занятия. Соответственно, множество занятий, претендующих на данную вакансию, является областью определения вакансии.

Если бы требования класса *A* были единственными, то можно было бы сформулировать достаточные условия существования расписания и описать алгоритм, гарантирующий положительный результат, если таковой вообще возможен. А именно было бы достаточно для каждого занятия выполнить следующие операции:

1. Найти в области определения занятия вакансию, отсутствующую в областях определения еще не распределенных занятий.
2. Составить новую пару расписания.
3. Вычеркнуть распределенное занятие из областей определения всех свободных вакансий.

Возможна ситуация, когда первый пункт выполнить не удастся, но это не самая большая проблема. Ситуацию резко ухудшает наличие требований *B*. Это требования взаимного положения занятий в расписании. Например, у группы студентов не должно быть дыр (пропусков между занятиями). Это может быть требование чередования предметов. Нельзя в один день поставить все занятия по одному предмету, в другой – все занятия по другому и т. д.

Формализация такого рода требований – тема отдельного разговора. Мы отметим только, что требования группы *B* усложняют понятие области определения занятия. Придется ввести два понятия области определения. Первую область, сформированную требованиями *A*, назовем абсолютной. Эта область может быть рассчитана один раз, до начала процесса составления расписания, и она не меняется. Вторую область назовем текущей, это набор вакансий, доступных для занятия с точки зрения полного множества требований *B*. Ясно, что эта область определения имеет переменный размер, она может по ходу составления расписания как увеличиваться, так и уменьшаться. К сожалению, уменьшение более типично для такого процесса.

Претензии разных занятий на одни и те же вакансии и существование требований *B* приводят к тому, что при наличии большого количества свободных вакансий текущая область определения какого-то занятия может оказаться пустой. Такую ситуацию будем называть конфликтом. Введем понятия риска конфликта для занятия. Пусть D – это размер области определения для некоторого занятия. Тогда назовем величину $1/D$ риском. Ясно, что чем D меньше, тем величина риска

больше. При D=0 величина риска становится бесконечно большой, что и означает конфликт.

Главная идея. Пусть некоторое количество пар расписания уже построено. Рассчитаем риски для нераспределенных занятий и выполним сортировку в порядке убывания. Тогда очередным распределяемым занятием становится занятие, имеющее наибольшую величину риска. Идея очень естественная. Её разумность, наверное, очевидна, но борьба за эффективность потребует серьезной доработки.

Главную идею можно существенно дополнить. Размер области определения вакансии также является величиной, характеризующей степень риска. Если область определения вакансии велика, то, связав вакансию с занятием, мы уменьшаем области определения значительного количества занятий. Поэтому желательно привлекать к распределению вакансии с наименьшей областью определения.

Декомпозиция

Процесс составления расписания заключается в формировании пар (занятие, вакансия) и состоит из решения следующих подзадач:

- Расчет текущей области определения занятия.
- Расчет области определения вакансий.
- Сортировка множества занятий.
- Сортировка множества вакансий.
- Составление пар (занятие, вакансия).

И еще один важный пункт – выход из конфликтной ситуации. Анализируемый алгоритм позволяет уменьшить вероятность конфликтной ситуации на каждом шаге распределения, но вряд ли он дает возможность её избежать. Мы не будем сейчас заниматься разработкой алгоритма разрешения конфликтов, для наших целей достаточно понимания, что такой алгоритм необходим. Возможно, алгоритм разрешения конфликта будет сбрасывать часть расписания и как-то изменять ход последующего распределения, наверное, здесь возможны различные идеи. Но пора заняться написанием алгоритма. Ниже дан псевдокод, описывающий структуру алгоритма.

1. Рассчитать текущие области определения и величины рисков для каждого занятия. Величина риска рассчитывается, как величина обратная к величине области определения.
2. Рассчитать области определения для каждой вакансии и величины рисков. Величина риска равна размеру области определения.
3. Упорядочить множество занятий в порядке убывания рисков.
4. Упорядочить множество вакансий в порядке убывания рисков.
5. Взять для очередной пары расписания из множества занятий первое.
6. Выбрать из области определения взятого занятия вакансию, имеющую наименьший номер в списке вакансий. Построить пару расписания.

7. Пересчитать текущие области определения занятий.
8. Пересчитать текущие области определения вакансий.
9. Если в списке занятий есть занятие с пустой текущей областью определения, то вызвать процедуру разрешения конфликтов.
10. Если список занятий не пуст, то перейти на п. 3.
11. Расписание составлено. Работу завершить.

Мы выполнили декомпозицию исходной задачи. Получившийся в результате алгоритм представляет собой один большой цикл, в теле которого вызываются несколько процедур, каждая из которых реализует собой достаточно большую задачу. Процедуры вызываются последовательно, независимо друг от друга, их связь осуществляется через общий набор данных. Каждая из подзадач-процедур идентифицируется фразой, смысл которой обозначает цель подзадачи-процедуры, дальнейшую разработку каждой из подзадач могут вести разные программисты, единственное, что требуется – это точное описание структур данных.

Уточним понятие «Передача управления»

Передача управления от программного блока *A* к программному блоку *B* означает завершение работы блока *A* и начало работы блока *B*. Это во-первых. Во-вторых, передача управления означает завершение подготовки данных, необходимых для работы очередного программного блока. Работу программы можно таким образом воспринимать как передвижение набора данных между программными блоками. Тогда передачу управления можно понимать как передачу набора данных.

Замечание о процедурах и функциях

Иногда в учебниках программирования можно встретить понимание процедуры как средства, позволяющего многократно использовать код, записанный один раз. Это, конечно, правильное понимание, но, кроме того, процедура – это средство структурирования программы. В рассмотренном выше примере задачи составления расписания учебных занятий результат представляет собой набор процедур, ни одна из которых не вызывается дважды. Таким образом, имя процедуры – это некоторая метка, поясняющая, какая подзадача в данной точке текста программы должна быть решена. Это, в свою очередь, позволяет разделить процесс разработки на различные уровни абстракции с определенной последовательностью передачи управления и структур данных.

Еще один пример

Рассмотрим следующую задачу. Некий путешественник выходит из пункта *A* и следует в пункт *B*, находясь в глубоком тумане и имея компас, всегда показывающий на пункт *B*. Необходимо построить траекторию пути от *A* до *B* через поле, заполненное препятствиями, построенными из прямоугольников (препятствие может иметь достаточно сложную форму). Эта задача детально разобрана в [3] и [4]. Мы же сейчас используем её как ещё один пример декомпозиции. Обозначим текущие координаты путешественника через *x*, *y*, исходные координаты как Ax, Ay

и координаты пункта назначения через Bx, By. Тогда верхний уровень разработки дает следующее решение:

Вариант 1.

```
x:=Ax; y:=Ay;
WHILE (x # Bx) OR (y # By) DO
    Путешествие;
END;
```

В этом варианте предполагается, что процедура **Путешествие** смещает путешественника на один шаг за один свой вызов. В этом варианте текущие координаты разумно объявить как глобальные. Цикл проверки в принципе можно включить в тело процедуры и получить следующий вариант:

Вариант 2.

```
x:=Ax; y:=Ay;
Путешествие;
```

Но такая запись несодержательна, она не несет в себе никакой информации о решении, кроме того, что путь начинается из пункта *A*. Сказанное выделим еще раз, как важнейший принцип:

Каждый уровень разработки должен решать содержательную задачу, а не сводиться к констатации факта, что нечто должно быть сделано.

С этой точки зрения второй вариант полностью не содержателен, первый вариант лучше, в нем описана некоторая логика, но если учесть, что проблема задачи все же заключается в построении пути, то сведение исходной задачи к процедуре **Путешествие** мало что дает. Поэтому еще немного поработаем на верхнем уровне.

Особенностью нашего путешественника является то, что он в процессе движения попадает в две принципиально отличные ситуации:

- Путь по пустому пространству.
- Обход препятствия.

Заметим также, что если путешественник не дошел до пункта *B*, то в отношении его состояния справедливы следующие утверждения:

- По завершении обхода препятствия путешественник попадает на пустое пространство.
- Завершение пути по пустому пространству возможно только при столкновении с препятствием.

Сказанное позволяет описать верхний уровень существенно более содержательным:

Вариант 3.

```
x:=Ax; y:=Ay;
WHILE (x # Bx) OR (y # By) DO
```

```
    ПутьВперед;  (*Движение по прямой*)
    Обход;  (*Обход препятствия*)
END;
```

Программный фрагмент стал существенно более содержательным, но в то же время и ошибочным. Здесь движение по прямой всегда должно завершиться обходом препятствия. И только завершение обхода гарантирует проверку достижимости точки назначения. Это означает, что цикл путешествия завершится только в том случае, если пункт назначения окажется на границе некоторого препятствия. Отсюда мораль:

Структура передачи управления на каждом уровне разработки не может не зависеть от формулировки подзадач, полученных в результате декомпозиции.

В третьем же варианте этот важнейший принцип был нарушен. В варианте 1 путник двигался пошагово, на каждом шагу проверяя свои координаты. В варианте 3 мы существенно изменили характер движения. Теперь он выполняет движение определенного рода до тех пор, пока это возможно. Выход из положения может заключаться в переносе проверки координат в тело процедур, но тогда процедуры должны сообщать наверх о причине прекращения своей деятельности. Введем переменную – флаг, который будет истинным тогда и только тогда, когда пункт назначения не достигнут. И третий вариант запишется так:

Вариант 3. Исправленный

```
x:=Ax; y:=Ay; Flag:=TRUE;
WHILE Flag DO
    IF Flag THEN Flag:=ПутьВперед; END;    (*Движение по прямой*)
    IF Flag THEN Flag:=Обход; END;   (*Обход препятствия*)
END;
```

Верхний уровень полностью завершен. Отметим только, что для проверки координат ранее была только одна запись, сейчас их две, по одной в каждой процедуре. Проиграли мы или выиграли, продублировав текст? Согласно следующему правилу мы, выиграли, и довольно существенно:

Между краткостью текста и логической независимостью подзадач необходимо выбирать второе. Возможно, удлинение текста будет полностью компенсировано более простой структурой всего проекта.

Разбирать задачу детально не входит в наши планы, но еще немного продвинемся вглубь. Начнем с **ПутьВперед**. Работа этой процедуры заключается в смещении путника на некоторый малый вектор в направлении пункта *B* до тех пор, пока либо не будет достигнут пункт *B,* либо не встретится препятствие. Способ проверки первого условия очевиден. О втором условии необходимо поговорить. Пусть поле, по которому происходит движение, – это экран монитора, пусть его

цвет черный, препятствия обозначим линиями белого цвета. Тогда столкновение с препятствием – это обнаружение по ходу движения точки белого цвета. Отсюда следует необходимость в следующих процедурах:

- **ЦветТочки**(x,y) – функция, получающая на вход координаты точки и возвращающая её цвет.
- **ВекторСмещения**(OUT dx, dy:REAL) – процедура, вычисляющая вектор смещения. Величины dx, dy – координаты вектора. Длина вектора, очевидно, мала, поэтому разумно для его координат использовать действительный тип. Вектор параллелен вектору (Bx − x, By − y).
- **ШагСмещения**() – процедура, выполняющая шаг смещения на величину вектора.

Запишем вариант текста процедуры **ПутьВперед**:

```
ВекторСмещения(dx,dy);
(*Пока не достигнута точка B и не встретилось
  препятствие выполнять движение*)
WHILE (x # Bx) OR (y # By) & (ЦветТочки(x, y) # White) DO
    ШагСмещения();
END;
IF (x=Bx) & (y=By) THEN Flag:=FALSE; END;
```

Детально обсуждать устройство и работы процедур **ВекторСмещения**() и **ШагСмещения**() не будем. Заметим только очевидный факт, что построение этого более низкого структурного уровня можно выполнить независимо от ранее сделанной работы. Чтобы дополнительно убедиться в сказанном, поработаем немного над процедурой обхода препятствия. Для её построения требуется ответ на два вопроса:

- Как организовать движение вдоль препятствия?
- Как определить, что препятствие пройдено?

Подробные ответы на оба вопроса можно найти в [4], сейчас существенно следующее:

Организация движения. Движение вдоль препятствия – это движение вдоль стенок, то есть движение с изменяющимся направлением.

Для ответа на второй вопрос нужен критерий, который можно было бы вычислять специальной процедурой. И конечно, нужна процедура, собственно выполняющая шаг путника вдоль стенки. Эти рассуждения приводят нас к следующему программному фрагменту:

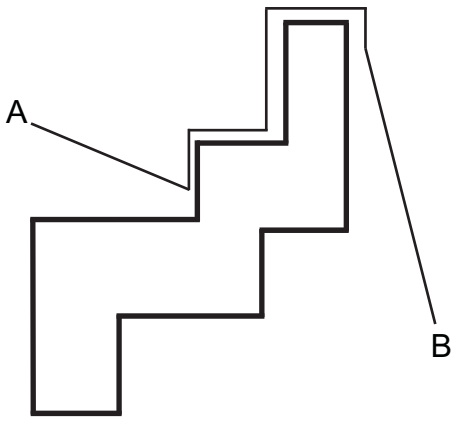


Рис. 1.4