1. Решение задач с использованием вычислительной техники

Представьте себе такую ситуацию: Вы руководитель отдела в программистской фирме. К Вам пришел заказчик со следующим предложением: “Мне нужна программа для нахождения нулей произвольной функции на заданном отрезке”.

Вы: “Отлично. Вы пришли в нужное место. У нас лучшие специалисты по нахождению нулей функции и именно на заданном отрезке”.

Что? Что-то не так? Вы думаете, так явно себя рекламировать не стоит? Может быть. Впрочем, суть не в этом. Допустим, Вы с заказчиком обо всем договорились, убедили его, что лучших исполнителей ему не найти, и он окрыленный ушел, насвистывая: “Мы рождены, чтоб сказку сделать былью”. А Вы остались и призадумались. А что теперь делать?

Описанная ситуация отнюдь не является надуманной. И вопрос этот можно более четко переформулировать так: “Пусть у нас есть задача, для решения которой мы хотим (нам необходимо, мы не можем обойтись без того, чтобы) использовать компьютер. Какие действия мы должны для этого выполнить?” Очевидный ответ типа: “Раз заказчик хочет программу – надо ее написать”, – на самом деле порождает еще больше вопросов. Как минимум: “А как? С чего начать?” Попытаемся разобраться.

Прежде всего, поскольку задача математическая (надеемся, этот факт не вызывает у Вас сомнения), начать наверное надо с попытки ее решения средствами математики.

Из курса математики известно, что, если функция $y = f(x)$ является *непрерывной* на $[a, b]$ и $f(a) \cdot f(b) < 0$, то эта функция имеет *корень* на $[a, b]$, т.е. такое $x = x_0$, что $x_0 \in [a, b]$, $f(x_0) = 0$.

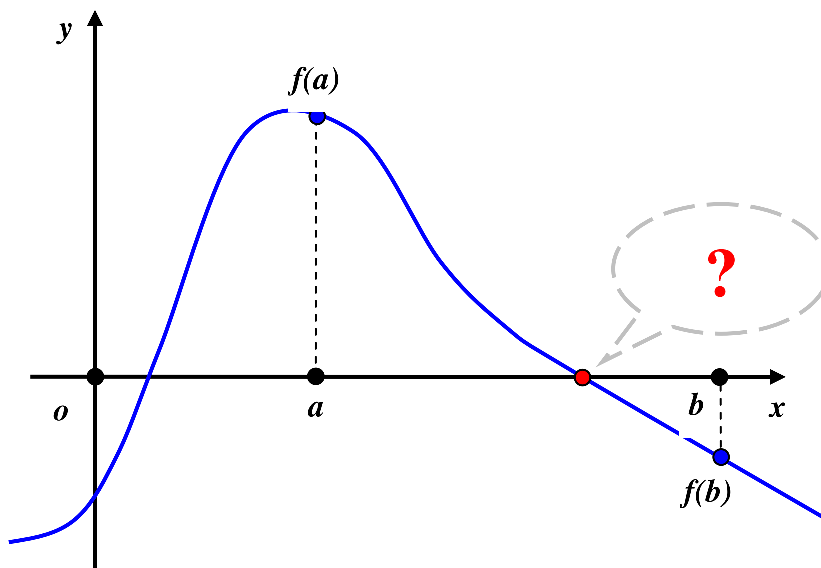

Рис 1.1. Корни уравнения f(x) = 0

Проблема нахождения значения x_0 состоит в том, что отнюдь не всегда уравнение $f(x) = 0$ может быть решено *аналитически*, то есть выведены формулы расчета его корней. Если аналитическое решение невозможно, то уравнение решают *численно*, – некоторым способом подбирают x', такое что значение функции $f(x)$ в этой точке достаточно близко к нулю. При этом, как правило, не удается найти точное решение, но удается отыскать его с некоторой *удовлетворительной точностью*.

Формально, вышесказанное может быть записано так: для решения задачи задается некоторое $\varepsilon > 0$, достаточно близкое к нулю, и требуется найти такое

$$x' \in [a, b], \text{ что } |f(x')| \le \varepsilon. \qquad (1.1)$$

Значение ε обычно вытекает из специфики задачи и определяет требуемую точность решения.

Для численного нахождения корней уравнения $f(x) = 0$ разработано немало *методов*. Среди них широко известными являются так называемые *итерационные методы*, которые строят цепочку точек $x_1, x_2, \ldots, x_n$, последовательно приближаясь к решению.

Общий вид формулы, задающей такой метод, выглядит следующим образом:

$$x_{n+1} = f^*(x_1, x_2, \ldots, x_n), \qquad (1.2)$$

где $x_1, x_2, \ldots, x_n$ – последовательность “кандидатов” на почетное звание корня, а f^* – некоторая функция, определяющая сам метод.

В качестве примера таких методов можно привести: метод половинного деления (деления отрезка пополам, дихотомии), метод касательных, метод секущих.

Рассмотрим кратко один из методов – *метод половинного деления*.

Формула, задающая метод, выглядит следующим образом:

$$x_{n+1} = \frac{a_n + b_n}{2}, \qquad (1.3)$$

где a_n и b_n есть текущие значения границ отрезка, на котором происходит поиск корня.

В начале работы метода $a_0 = a$, $b_0 = b$.

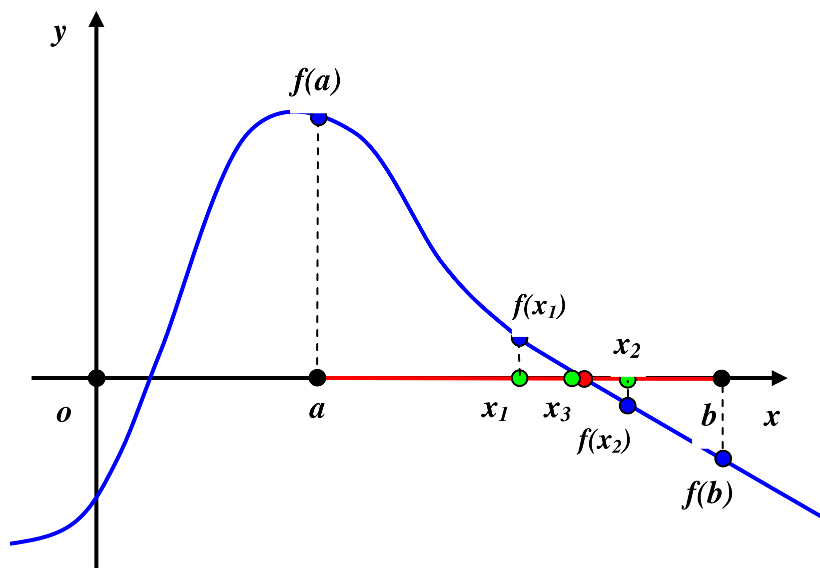

Рис 1.2. Метод половинного деления

На каждом шаге метода происходит вычисление середины отрезка x_{n+1} по указанной формуле, после чего производится вычисление функции в точке x_{n+1}. Если выполнено условие останова ($|f(x_{n+1})| \le \varepsilon$), то найденная точка x_{n+1} считается решением, и метод заканчивает работу.

В случае если условие не выполнено, в качестве очередного отрезка для поиска выбирается тот из отрезков $[a_n, x_n]$ или $[x_n, b_n]$, для которого функция на концах принимает значения разных знаков.

Теперь дело за малым. Осталось реализовать приведенный метод (записать его на языке понятном компьютеру) и получить *программу* для нахождения корня уравнения f(x) = 0 на отрезке [a, b].

Все просто, не правда ли? Помимо того, что мы не объяснили, что же такое “язык понятный компьютеру”, и как на нем что-нибудь написать, кажущаяся легкость связана еще и с тем, что в результате проведенного анализа нам стали хорошо известны ответы на следующие вопросы:

1. В чем суть задачи, что нам дано, и что нужно найти?
2. Какими математическими соотношениями описывается связь между исходными данными и требуемым результатом?
3. Какой метод необходимо применить для решения этой системы математических соотношений? В чем суть этого метода?

Думается, для того, чтобы ответить на эти вопросы, в рассмотренной задаче должно быть достаточно знаний выпускника средней школы. В реальной ситуации все будет значительно сложнее, и получение ответов может поставить серьезные проблемы даже перед квалифицированными специалистами. Однако лишь после этого можно двигаться дальше, то есть, собственно, переходить к процессу, который обычно называют “программирование”. О деталях этого процесса мы поговорим позже, а пока подведем промежуточные итоги.

Итак, решение любой сколько-нибудь серьезной задачи с использованием компьютера отнюдь не начинается с составления программы (справедливости ради надо сказать, что и не заканчивается на этом). Прежде, чем можно будет приступить непосредственно к программированию, нужно, отталкиваясь от формулировки задачи, пройти ряд обязательных этапов. Далее в этой главе мы рассмотрим эти этапы, их назначение и выполняемые на каждом из них действия. Когда программа, наконец-то, будет готова, наступит очередь других важных этапов, о которых мы тоже поговорим.

Что касается самого процесса программирования (написания текста программы), то он в чем-то сродни написанию литературного произведения (хотя и более формализован), и является процессом творческим. Таким образом, для овладения им отнюдь не достаточно выучить “язык понятный компьютеру”. Так же как Александру Сергеевичу Пушкину для создания поэмы “Евгений Онегин” отнюдь не достаточно было глубокого знания русского языка, но потребовался и его великолепный талант и предшествующий опыт написания стихов и поэм и, что не менее важно, владение методами и техникой стихосложения, так и для создания, или как говорят профессионалы “разработки”, программ требуется освоение определенных приемов и методов, в совокупности образующих технологии программирования. О технологиях речь пойдет, начиная с пятой главы книги. А до этого мы успеем обсудить этапы решения задач с использованием компьютера, средства, которые помогают нам в этом процессе, поговорим о вещах, не связанных, казалось бы, с программированием напрямую, но имеющих, тем не менее, весьма важное значение (процессор, оперативная память, операционная система и т.д.), а также научимся составлять простейшие программы.

Впереди длинный путь, надеемся, что нам удастся сделать его достаточно интересным для Вас. Итак, приступим.

Постановка задачи

В правильной постановке вопроса содержится половина ответа.
Научная мудрость

У разработчика программ не было бы особых проблем, если бы задачу ему формулировали примерно в таком виде: "Напиши оператор ввода трех чисел, вычисли значение по такой-то формуле, сравни результат с нулем…", то есть прямо задавали некоторый план (*алгоритм*) действий.

Нетрудно догадаться, что на практике дело обстоит по-другому. Вот как, например, описывает один французский специалист данную ему Национальным географическим институтом постановку задачи: "Имеются соответствующие данные о самолетах, экипажах, доступном оборудовании, аэропортах, полетных задачах (пункты назначения, высота полета, скорость, степень срочности и т.д.) и карты ежедневных метеорологических наблюдений со спутников. Программная система должна предлагать эффективные решения по распределению самолетов, персонала и оборудования на каждый день работы и допускать оперативное изменение параметров и перераспределение ресурсов"[1].

В таком весьма общем виде формулируется множество заданий на разработку программных комплексов, по крайней мере, изначально. И хоть они выглядят несколько более внятно, чем известное "Пойди туда – не знаю куда, найди то – не знаю что", все же разработать программу по такой постановке, конечно, невозможно. Добиться от заказчика ответов на все необходимые для воплощения его потребностей в жизнь вопросы (а заодно и выяснить, что же он в реальности хочет – часто это не вполне совпадает с тем, что он говорит) – Ваша центральная, как исполнителя, задача. При этом в ходе получения ответов первоначальная постановка, скорее всего, существенно изменится.

Принципиальные вопросы, которые должны быть решены, прежде чем можно будет приступить к реализации программы, мы частично озвучили выше. Далее мы более детально рассмотрим все этапы решения задачи с использованием компьютера на конкретном достаточно простом примере.

Итак, пусть заказчику требуется программная система для выполнения регулярных расчетов *арендной платы за земельные участки*. Примем, что *арендная плата* – произведение стоимости одного квадратного метра земли на площадь участка. На самом деле это не всегда так, поскольку участок может состоять из земель разных типов, стоимость квадратного метра которых может быть различна. Таким образом, мы уже сделали первое *допущение*, которое на самом деле должно быть согласовано с заказчиком – должна ли наша будущая программа быть рассчитана на такую ситуацию или нет.

Допустим, заказчик клятвенно заверил нас, что учитывать возможную разную стоимость квадратного метра не требуется. Следующая наша задача – вычисление *площади участка*. Начиная ее решение, мы переходим к следующему этапу – *построение модели*.

[1] Herve Thiriez, Modelling of an interactive scheduling system in a complex environment. European Jornal of Operational Research 50 (1991), 37-47

Модель

Модель – формальное (как правило приближенное) описание изучаемого объекта или явления, отражающее интересуюшие нас аспекты.

Зачем нужна модель? Реальные объекты, о которых идет речь в задаче, чаще всего достаточно сложны, описываются массой параметров, существенной частью которых можно и нужно пренебречь. Например, пусть земельный участок имеет форму прямоугольника. Означает ли это, что его площадь есть произведение длины на ширину? Да? Вы хорошо подумали? А если участок имеет вид холма? Никто не говорил, что уровень земли по всему участку одинаков. Вот и еще одно *допущение*. Все вместе подобные допущения, ограничения, не принимаемые в расчет параметры и составляют модель объекта или явления.

Итак, формализуем условие нашей задачи, т.е. введем обозначения для исходных данных, требуемого результата и результатов промежуточных вычислений. Прежде всего, требуется формализовать понятие *земельный участок*.

Для начала, допустим, что в результате изучения плана местности и бесед с заказчиком, выяснилось, что участки имеют прямоугольную форму и уровень земли по всему участку одинаков. Тогда анализ постановки задачи приводит к следующей *системе параметров*.

Исходные данные:

a, ***b*** – размеры участка (стороны прямоугольника);

Price – стоимость одного квадратного метра земли.

Требуемый результат:

Rent – арендная плата.

Важные промежуточные результаты:

S – площадь участка.

Попробуем построить модель для этого случая. Так как участок имеет прямоугольную форму, вычисление его площади не представляет труда.

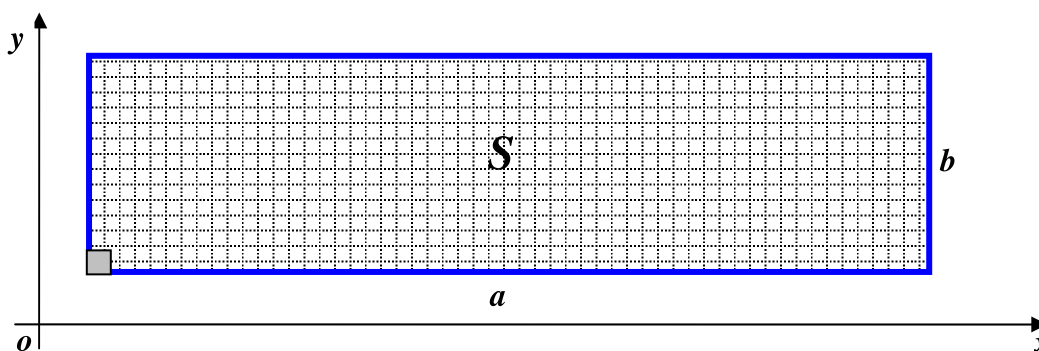

Рис 1.3. Модель земельного участка прямоугольной формы

Как известно, площадь прямоугольника с учетом принятых обозначений вычисляется следующим образом:

$$S = a \times b \qquad (1.4)$$

Посмотрим теперь, что мы должны получить на выходе модели? Арендную плату ***Rent***. При принятых нами допущениях она вычисляется по следующей формуле:

$$Rent = S \times Price \qquad (1.5)$$

Формулы (1.4) и (1.5) совместно составляют *математическую модель* для решаемой задачи (для случая прямоугольной формы участка), связывая исходные данные и требуемый результат.

В принципе уже можно переходить к следующему этапу, но поскольку вариант с прямоугольной формой участка слишком прост, давайте рассмотрим чуть более общую ситуацию, имеющую к тому же под собой реальное обоснование – краевые участки, одна из сторон которых примыкает к реке или дороге и представляет собой кривую. Как сообщил нам заказчик, ни владелец земли, ни арендаторы не согласны "спрямить" эту сторону и настаивают на точном расчете.

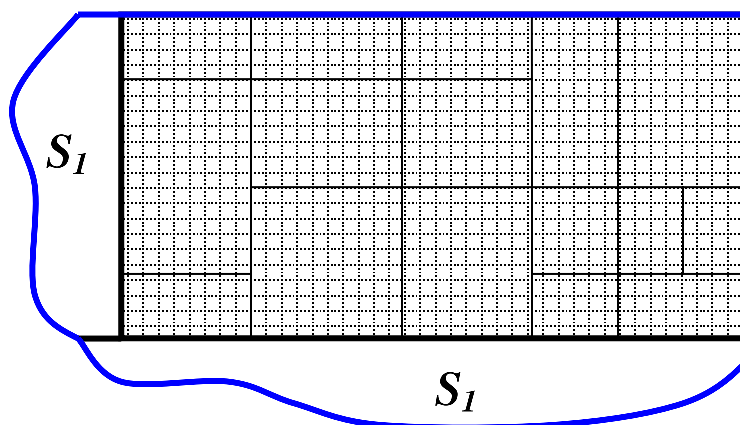

Рис 1.4. Модель земельного участка общего вида

Чтобы справиться с этой проблемой, требуется немного больше знаний. Геометрической моделью объектов такого вида является так называемая *криволинейная трапеция*.

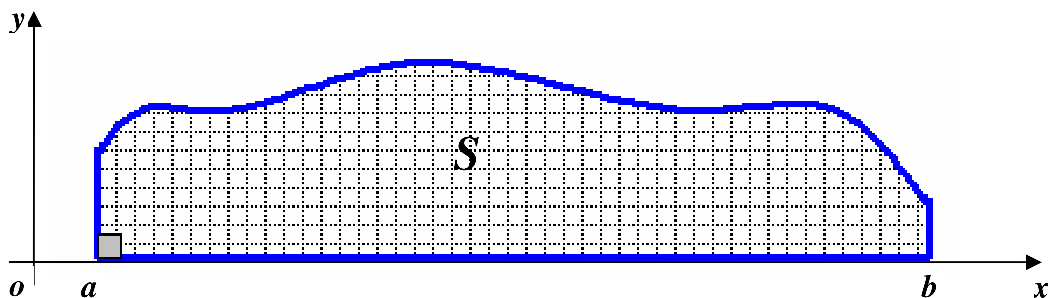

Рис 1.5. Криволинейная трапеция

Внесем изменения в систему параметров.

Исходные данные:

a, ***b*** – границы участка;

f(x) – функция, описывающая "криволинейную часть" участка;

Price – стоимость одного квадратного метра земли.

Требуемый результат:

Rent – арендная плата.

Важные промежуточные результаты:

S – площадь участка.

Площадь криволинейной трапеции выражается при помощи определенного интеграла, требующего знания параметров трапеции, например, f(x) – функции, задающей график кривой (одной из “сторон” трапеции).

$$S = \int_a^b f(x)dx \qquad (1.6)$$

Читателю, не знакомому с понятием интеграла, достаточно знать, что существуют специальные правила интегрирования – их искусное применение позволяет получить формульное выражение, подставив в которое пределы интегрирования (параметры участка) мы получим численное значение площади. Соотношение (1.6) и формула вычисления арендной платы (1.5) задают математическую модель нашей задачи для случая криволинейных участков. Математическая модель является основой построения *информационной модели* задачи. Любой обрабатываемый в программе объект, помимо параметров, участвующих в расчетах, может характеризоваться информационными (описательными) параметрами. Например, в дополнение к данным, определяющим площадь участка и стоимость одного квадратного метра, заказчик может потребовать включить в информационную модель некоторые данные, идентифицирующие участок: его номер и фамилию владельца. Разумеется, это требование отразится на списке параметров, которыми мы описываем объекты задачи.

Построив модель, то есть, определившись, в конечном счете, со схемой получения из исходных данных требуемого результата, мы должны теперь для каждого выбранного для реализации варианта четко сформулировать способ расчета, то есть построить *метод вычислений*.

Метод

На этапе построения модели мы выделили два возможных варианта: участки прямоугольной формы и участки с одной криволинейной стороной.

Метод вычислений для первого случая тривиален, он в точности определяется формулами (1.4) и (1.5), расчет которых не представляет никаких проблем при любых исходных данных.

Со вторым вариантом несколько сложнее. Он тоже задается двумя формулами: (1.5) и (1.6), однако расчет площади по формуле (1.6) уже не так прост. Если для входящей в *исходные данные* функции f(x) известно аналитическое (формульное) выражение, то можно попытаться проинтегрировать формулу (1.6) и получить способ вычисления площади. Правда, функция f(x) может оказаться сложной и применить правила интегрирования будет непросто, но главная проблема даже не в этом. Дело в том, что точную формулу, выражающую кривизну реки или дороги на всех участках, взять просто неоткуда, и столь красивая математическая модель оказывается практически бесполезной.

Выходов из сложившейся ситуации три: первый – найти (придумать, если его не существует) метод расчета для выбранной модели, второй – изменить модель так, чтобы метод расчета для нее был известен, третий – вернуться к предыдущему этапу и попытаться построить другую модель. В данном случае мы пойдем по второму пути.

Начать надо с ответа на вопрос: “Чем вызвана возникшая проблема?”. Кажется, тем, что сторона участка имеет вид, не позволяющий выполнить точный расчет площади. Это конечно правильно, однако напомним – на самом деле мы работаем не с самим *объектом*, а с его *моделью*. А в модель криволинейная сторона попала потому, что владелец и арендатор не соглашаются ее “спрямить”. А можем мы сделать так, чтобы согласились? Можем. Для этого надо разбить криволинейную сторону на некоторое количество частей так, чтобы кривизна каждой части была достаточно мала, тогда ее можно будет спрямить с приемлемой погрешностью. Конечно, в

результате мы не найдем точную площадь участка, однако если суммарная погрешность от замены исходной формы криволинейной стороны на ломаную линию будет достаточно мала, то такой подход можно использовать.

В результате мы приходим к следующему *методу вычисления* площади участка. На основе изучения карты или обследования на местности для каждого участка проводится разбиение криволинейной стороны на фрагменты, пригодные для замены отрезками прямой. Это разбиение порождает разбиение всего участка на обычные трапеции, в которых одна из боковых сторон является высотой. Их площади легко вычисляются по известной формуле, а площадь всего участка полагается равной сумме площадей трапеций. Число трапеций и длины их высот индивидуальны для каждого участка и определяются как разумный компромисс между точностью приближения кривой и трудоемкостью измерений.

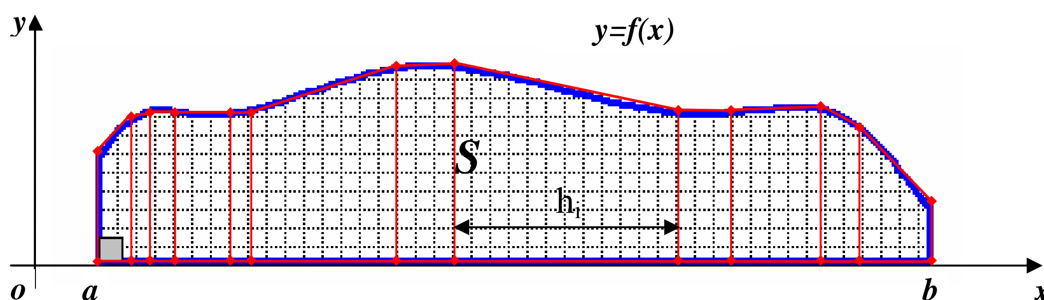

Рис 1.6. Приближенное вычисление площади криволинейной трапеции

Представленный нами *метод* порождает новую *модель*, описывающую задачу, в соответствии с этим меняется и система используемых нами параметров.

Исходные данные:

a, ***b*** – границы участка;

N – число трапеций;

y_i – длины оснований трапеций;

h_i – высоты трапеций;

Price – стоимость одного квадратного метра земли.

Требуемый результат:

Rent – арендная плата.

Важные промежуточные результаты:

S – площадь участка.

В сделанных обозначениях формулу расчета площади участка мы можем записать так:

$$S = \sum_{i=0}^{N-1} h_i \times \frac{(y_{i+1} + y_i)}{2} \quad , \qquad (1.7)$$

Теперь, имея в распоряжении исходные данные, мы сможем без труда выполнить все расчеты.

Итак, *метод вычислений* найден. Пора переходить к реализации? Не совсем так. Прежде нам предстоит пройти еще один достаточно важный этап – *построение алгоритма*.

Алгоритм

Алгоритм – точный план действий по решению задачи.

На этапе построения *модели* мы определили систему параметров, описывающих задачу, и установили соответствие между исходными данными и требуемым результатом. На этапе построения *метода вычислений* мы уточнили модель и сформировали точную схему выполнения расчетов. Но раз так, значит алгоритм, то есть план решения, готов! Берем исходные данные, подставляем в формулы расчета и получаем результат. Правильно? На самом деле нет. Подумайте, что значит: "Берем исходные данные"? Откуда берем? Не забудьте, что выполнять расчет будет программа, а она не сможет попросить пользователя: "Дай-ка, дружок, мне вон те циферки, я их сложу и умножу". Итак, тот факт, что мы решили задачу математически, не означает, что мы сформировали алгоритм, который можно будет превратить далее в программу.

Чего же не хватает? Не хватает учета возможностей исполнителя, то есть компьютера. Возможности эти естественно ограничены, так что, несмотря на непрекращающийся с момента создания первой ЭВМ рост мощности компьютеров, в каждый текущий момент времени существуют задачи, которые современной вычислительной технике не под силу. Эти и множество других ограничений, присущих компьютерам в силу их внутреннего устройства, необходимо учитывать для грамотного составления алгоритмов, для чего может потребоваться снова вернуться к этапам построения модели и метода. Однако подробные рассуждения на эту тему уведут нас далеко в сторону, поэтому мы их отложим. Пока же достаточно отметить, что любой алгоритм, предназначенный для последующего воплощения в программу, должен предусматривать действия по получению исходных данных, выполнению на их основе указанных расчетов, и выдаче с возможной предварительной обработкой результатов.

В нашей задаче исходные данные будут формироваться в результате измерений, выполняемых на участках. Будем считать, что эти результаты накапливаются в *текстовом файле*, а наша программа должна будет этот файл "читать" при запуске. Осталось лишь определиться с формой представления данных в этом файле. Например, она может быть следующей: номер участка, фамилия и инициалы владельца, количество отрезков разбиения, длины оснований трапеций (в метрах) и, наконец, высоты трапеций (в метрах).

Фрагмент такого файла может выглядеть следующим образом:

```
154
Иванов И.И.
3
40
37
50
45
7
20
10
```

Теперь нужно решить вопрос с представлением результатов расчетов. Результат расчёта должен быть представлен в виде справки на экране и содержать идентифицирующую информацию участка, значения площади и арендной платы.

Итак, все необходимые решения приняты, можно записывать алгоритм. Вот только как? Модель и метод описываются с помощью общепринятой математической символики, а также просто словесно. А как выглядит язык записи алгоритма? Во-первых, алгоритм можно записать обычным "человеческим" языком. Однако каждое действие алгоритма должно пониматься однозначно, а разговорные языки обычно "грешат" многозначностью. Во-вторых, формальными

языками записи алгоритмов являются *языки программирования*. О них речь пойдёт в следующих главах книги. Однако изложить алгоритм на формальном языке сразу бывает довольно сложно, требуется предварительная неформальная его запись в промежуточном, “черновом” варианте. В качестве такого промежуточного языка используется *язык блок-схем*. Надо отметить, что детальная запись сложного алгоритма на этом языке – дело трудоемкое, а результат, представляющий собой нечто вроде принципиальной схемы телевизора, не так уж и нагляден. Поэтому блок-схемы применяются в основном для изображения укрупненной общей схемы алгоритма, либо отдельных его фрагментов. Вот пример такого укрупнённого изображения алгоритма для нашей задачи:

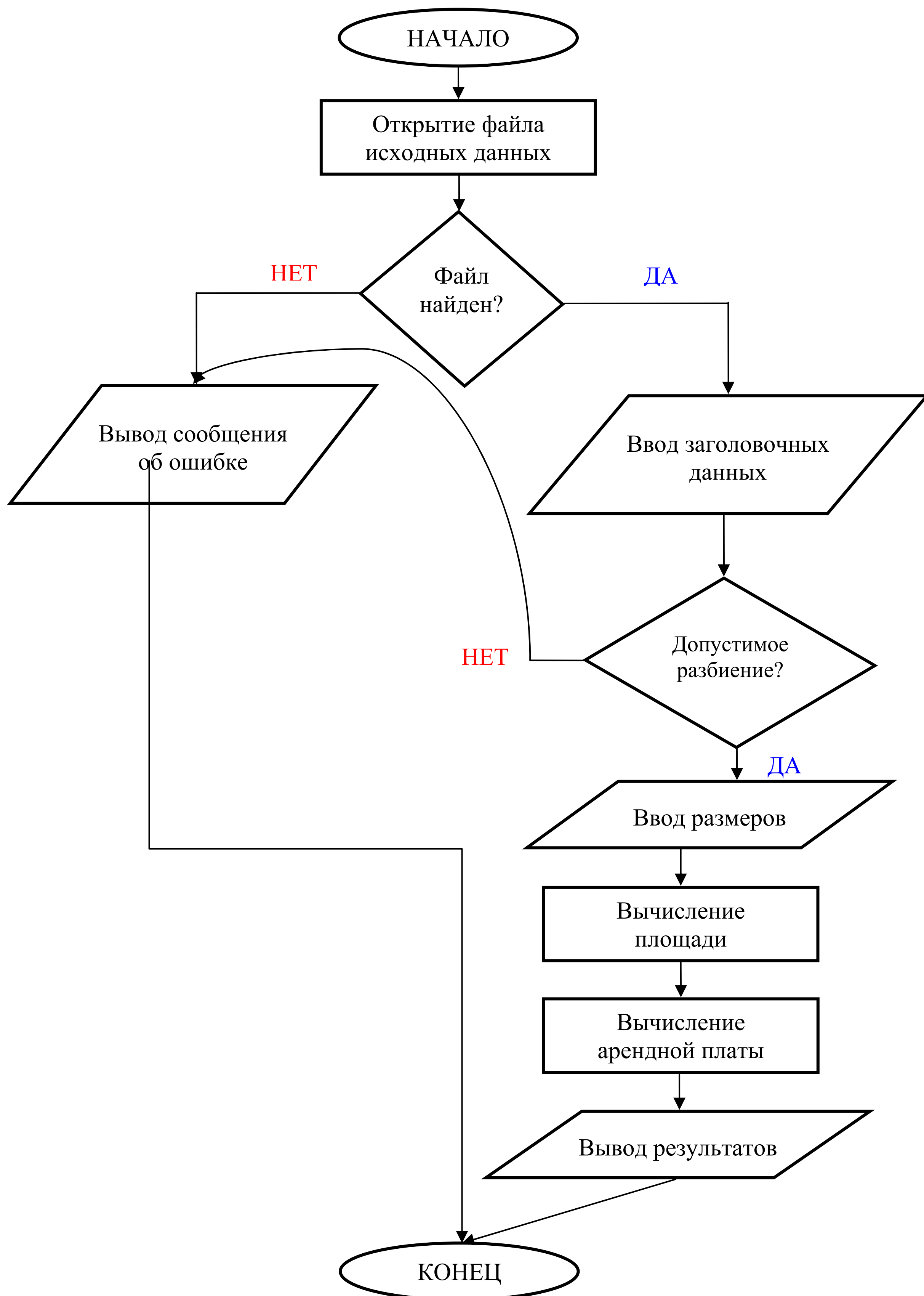

Рис 1.7. Блок-схема алгоритма расчета арендной платы

Вообще, для представления алгоритма в литературе в настоящий момент чаще всего применяется следующая форма описания: словесное изложение с элементами того или иного языка программирования. Мы в дальнейшем тоже будем пользоваться такой формой.

В заключение раздела, отметим, что разработка алгоритма, так же как построение модели и метода – это, конечно, творческий неформализуемый процесс. Вместе с тем, за время существования программирования как научной и технической дисциплины, конечно же, наработаны некоторые правила и рекомендации, облегчающие "тяжелую программистскую долю". Одним из таких фундаментальных приёмов является иерархическое разбиение сложной задачи на ряд подзадач. Современные системы программирования обеспечивают пошаговую, поэтапную реализацию алгоритма, как говорят, разработку "сверху-вниз", т.е. от общего, укрупненного представления алгоритма ко все более детальному виду. Вопросы полномасштабного освещения подобных технологий выходят за рамки данной книги. В то же время, положенные в их основу технологии модульного и структурного программирования, вопросы конструирования новых типов данных, введение в объектно-ориентированное программирование будут нами рассмотрены.

Программа

Ну вот, наконец, мы добрались и до этапа составления *программы*, то есть записи *алгоритма* на языке, "понимаемом" компьютером. Разговор о языках мы снова отложим (до следующей главы), отметим только, что одним из таких языков является *язык программирования* Zonnon. А сейчас в полном соответствии с высказыванием "Лучше один раз увидеть, чем сто раз услышать" давайте сразу посмотрим на текст программы, реализующей решение рассмотренной в предыдущих разделах задачи. Из него мы сможем составить первое представление о языке, с которым будем работать далее на протяжении всей книги. Для того чтобы облегчить процесс знакомства, мы сопроводили весь текст пояснениями, оформив их в виде комментариев, заключённых в фигурные скобки. Жирным шрифтом выделены так называемые ключевые слова языка, выражающие его основные синтаксические конструкции. Более подробно с этими и другими элементами языка мы познакомимся чуть позднее.

```
(* ================================================================== *)
(* Пример 1.1 *)
(* Вычисление арендной платы за земельный участок *)
module  Rent; (* программа Rent (заголовок) *)
import
  System.Int32 as Integer,
  System.Single as Single,
  System.IO as IO,
  System.IO.StreamReader as StreamReader;
(* декларативная часть программы - различные объявления *)

(* константы *)
const
  MaxSeg = 20; (* максимальное число отрезков разбиения *)
  Rate = 0.10; (* плата за 1 кв. метр в рублях *)

(* типы данных *)
type
  Coord = array MaxSeg of real; (* вектор вещественных     *)
                                  (* координат из MaxSeg + 1 *)
                                  (* элементов               *)
```

```
(* переменные и их типы *)
var
  sr: StreamReader;
  s: string;
  y, h: Coord;      (* векторы длин оснований трапеций    *)
                    (* и длин отрезков разбиения          *)
                    (* (высот трапеций)                   *)
  N, i: integer;    (* текущее число отрезков разбиения   *)
                    (* и переменная цикла (целые числа)   *)
  Number,           (* номер участка                      *)
  Name: string;     (* и имя владельца (текстовые строки) *)
  Area,             (* площадь и                          *)
  Cost: real;       (* стоимость (вещественные числа)     *)

(* начало исполняемой части программы *)
begin
  if IO.File.Exists('AREADATA.TXT') = false then
    writeln('ОШИБКА 1: Не найден файл AREADATA.TXT');
    halt(1);                (* завершение работы       *)
  end;

  sr := new StreamReader('AREADATA.TXT');
  (* чтение файла *)
  Number :=  sr.ReadLine();        (* номер участка *)
  Name := sr.ReadLine();           (* имя владельца *)
  s := sr.ReadLine();
  N := Integer.Parse(s);           (* количество отрезков разбиения *)

  if N > MaxSeg then (* если количество отрезков больше максимально *)
                     (* допустимого, то вывод на дисплей            *)
    writeln('ОШИБКА 2: Недопустимо большое число отрезков');
    halt(2);              (* завершение работы *)
  end;

  (* чтение файла *)
  for i := 0  to  N  do
    s := sr.ReadLine();
    y[i] := Single.Parse(s); (* длины оснований трапеций *)
  end;

  for i := 1  to  N  do
    s := sr.ReadLine();
    h[i] := Single.Parse(s); (* длины высот трапеций *)
  end;

  (* вычисление площади участка *)
  (*      "метод трапеций"      *)
  Area := 0.0;
  for i := 1  to  N  do
    Area := Area + (y[i - 1] + y[i]) * h[i];
  end;
  Area := Area / 2;

  Cost := Area * Rate;    (* арендная плата *)

  (* вывод на экран *)
  writeln('Участок ', Number);
  writeln('Владелец ', Name);
  writeln('Площадь участка ', Area, ' кв. м');
```

```
  writeln('Арендная плата ', Cost:10:2, ' руб.');
  (* конец программы *)
end Rent.
(* ================================================================ *)
```

Читатель, взявший на себя труд разобрать представленный текст программы, должен заметить, что собственно расчётная ее часть занимает всего лишь пять строк из общего текста. Может быть эта ситуация вызвана простотой решаемой нами задачи? На самом деле нет. На примере многих программ, рассматриваемых в книге, мы неоднократно сможем убедиться – почти всегда "обслуживающая" часть программы превышает по размеру (иногда многократно), собственно содержательные преобразования исходных данных в требуемый результат. Тому много причин. Во-первых, любая программа должна не просто работать, а работать надежно, то есть проверять все потенциально опасные для правильной работы ситуации (например, отсутствие файла с исходными данными). Во-вторых, существенную часть любой программы обычно составляет реализация интерфейса с пользователем (в простом случае это организация ввода и вывода информации). В-третьих, о чем мы неоднократно будем упоминать дальше, правильно написанная программа обязательно должна обладать тем, что на профессиональном языке называется "самодокументированность", и что означает использование при оформлении текста программы общепринятых стилистических правил (отступы, именование переменных и т.д). Есть и в-четвертых и в-пятых. Но об этом в свое время.

Итак, кажется, мы добрались до конца процесса – программа готова, можно пользоваться. В принципе, все верно, однако есть одно "но". Задача, которую мы рассматривали на протяжении пяти разделов, достаточно проста, но даже для ее реализации нам понадобилась программа, содержащая почти сотню строк. А то ли еще будет. Программа среднего по нынешним меркам размера это десяток другой тысяч строк, что равносильно книге страниц на триста. Спросите себя, сколько раз вы ошибетесь просто при наборе ее текста? Перефразируя классика, можно сказать: "О, сколько нам ошибок чудных готовит просвещенья дух"[2]. Но не пугайтесь, на самом деле не все так плохо, просто вот так плавно мы переходим к следующему "наиболее горячо любимому" всеми без исключения программистами этапу – *отладке* программы.

Стоит упомянуть распространенную ошибку: многие люди считают, что главное, это правильно произвести анализ предметной области, построить модель, разработать алгоритм, а уж программирование – кодирование алгоритма – дело десятое. Надо сказать, это совершенно не соответствует действительности. *Промышленное программирование* – сложный технологический процесс со своими специальностями, технологиями, проблемами.

Конечно, вопрос успешной разработки программного продукта в существенной степени зависит от качества подбора персонала и бюджета. Известен так называемый закон Лермана: "*Имея достаточно времени и денег, можно преодолеть любую техническую проблему*". Но вместе с законом известно также и его следствие: "*Вам всегда будет не хватать либо времени, либо денег*". Отсюда вывод – необходимо не просто учиться программировать, а учиться делать это *быстро* и *качественно*.

---

[2] А.С. Пушкин: "О, сколько нам открытий чудных готовят просвещенья дух…"

Отладка

Ошибки, возникающие в процессе создания программы, помимо упомянутой выше причины могут быть вызваны и неадекватным моделированием, и некорректностью метода или алгоритма, и, наконец, неправильным применением самих средств программирования.

В целом типы ошибок принято разделять на два неравнозначных класса. Один из них – это класс *синтаксических ошибок*, то есть ошибок, связанных с неправильной записью или употреблением языковых конструкций. Эти ошибки легко исправимы, так как соответствующее программное обеспечение – транслятор языка – осуществляет автоматический контроль синтаксической правильности программы пользователя, а с помощью программы контекстно-зависимой помощи можно получить как разъяснение ошибки, так и узнать правильный вид языковой конструкции.

Другой вид ошибок, действительно представляющий проблему программирования, – *смысловые ошибки*. Обнаружение и исправление их, что собственно и представляет собой процесс *отладки*, дело сложное, а порой, как это ни парадоксально звучит, и безнадёжное. Как определить, что программа имеет смысловую ошибку? В лучшем случае программа не работает, то есть её работа прерывается в некоторый момент и система выдаёт какое-нибудь туманное сообщение типа “исчезновение порядка числа с плавающей точкой”. В худшем случае программа успешно завершает свою работу и выдаёт результаты, отвечающие интуитивным представлениям о характере решения задачи, а о наличии ошибки в программе мы узнаём только после практического внедрения результатов, например, когда по нашим прочностным расчётам построили мост, а он тут же обвалился под собственной тяжестью.

Как обнаружить такие скрытые ошибки? Самый популярный метод – так называемое *тестирование*. Следует взять такие исходные данные, правильный результат расчёта для которых известен заранее, и выполнить программу с этими данными. Если полученный результат совпадает с известным, то, как говорят, “тест прошёл”. Беда в том, что, это совсем не означает, что программа не содержит ошибок. Рассмотрим простой пример. Допустим, нас попросили реализовать программу умножения двух вещественных чисел, и мы предложили следующий вариант на языке Zonnon:

```
(* ================================================================ *)
(* Пример 1.2 *)
module Mult;                                (* заголовок                *)
var                                         (* объявление               *)
  a, b, c: real;                            (* вещественных переменных  *)
begin
  writeln('Введите сомножители');           (* вывод запроса на дисплей *)
  readln(a, b);                             (* чтение с клавиатуры      *)
  c := a + b;                               (* ОШИБКА!!! + вместо *     *)
  writeln('Произведение равно ', c:4:4);    (* вывод результата на дисплей *)
end Mult.                                   (* конец программы          *)
(* ================================================================ *)
```

Чтобы доказать заказчику правильность предложенной программы, мы предлагаем ему тест: **2 x 2 = 4**, который, очевидно, проходит. Тем не менее, программа содержит существенную ошибку. Можно, конечно, сказать, что надо провести несколько тестов, но это означает, что мы должны знать все особые случаи. Для сложного алгоритма такая информация, как правило, неизвестна.

Ситуация усугубляется тем, что часто получение тестовых данных выливается в самостоятельную программистскую работу. Возьмём нашу задачу о земельных участках. Как получить значение площади участка с криволинейной стороной, если именно ради этого мы и составляли программу? Единственный выход – имитация. Следует “придумать” участок с

криволинейной стороной, вид которой задаётся некоторой формулой, и вычислить его площадь методом аналитического интегрирования. Затем выбрать отрезки разбиения, рассчитать длины оснований трапеций и, применив нашу программу, сравнить результаты. Однако ручной расчет длин оснований трапеций для нескольких вариантов теста вполне может оказаться задачей трудоёмкой, следовательно, надо написать программу. Но её тоже надо отладить! Замкнутый круг! Наконец, никто не может гарантировать, что мы не ошибемся и при аналитическом интегрировании.

Проблема получения тестовых данных настолько серьёзна, что иногда сдерживает разработку сложных систем. Например, один из аргументов противников разработки американской системы противоракетной обороны космического базирования (СОИ) состоял в том, что проверить правильность работы сложнейшей компьютерной системы управления крайне трудно. Реальный тест – запустить все ракеты потенциального противника и сбить положенное количество боеголовок – просто невозможен.

На практике разработчики сложных программных систем следующим образом решают проблему, связанную с заранее очевидной “недотестированностью” программ. Программы продаются (сдаются в эксплуатацию), а разработчики берут на себя обязательства по так называемому *сопровождению* программного продукта. Другими словами, разработчики обязуются при выявлении кем-нибудь из пользователей ошибок вносить необходимые исправления и предоставлять потребителям обновлённые версии программы.

Принципиально другой подход к выяснению корректности программы состоит в методе формального доказательства её правильности по типу доказательства теорем. Однако на пути практического применения этого подхода стоят большие трудности, и он не играет заметной роли в приложениях.

Итак, программа написана, предварительно отлажена и сдана в эксплуатацию. Неужели есть что-то дальше? Да, есть. Причем, чем лучше и серьезнее разработанная программная система, тем более долгая жизнь ждет ее в потенциале, а значит, тем длиннее будет это “дальше”.

Модификация

Крупные программные комплексы разрабатываются с расчетом на достаточно длительную эксплуатацию, измеряемую как минимум годами. В то же время динамичность компьютерной индустрии настолько велика, что даже пара лет является огромным сроком, в течение которого существенно меняется аппаратная база, требования к функциональности и качеству программных продуктов. А значит, разработанную программную систему придется модернизировать, добавлять в нее новые возможности, не предусмотренные первоначальной постановкой. Естественно этот процесс должен быть максимально быстрым и простым, что в свою очередь накладывает определенные довольно существенные требования на первоначальную реализацию. Она должны быть достаточно гибкой, чтобы необходимость модификации не приводила к полному перепрограммированию. В то же время в стремлении к универсальности нельзя заходить слишком далеко, иначе программная система станет чересчур громоздкой, а ее внутреннее устройство (как говорят, “архитектура”) запутанным и излишне сложным. В общем, необходим разумный компромисс между стремлением как можно раньше создать первую полнофункциональную версию и желанием облегчить процесс дальнейшей ее модификации. Найти этот компромисс – весьма непростая задача.

Различные приемы, помогающие в достижении указанного компромисса, мы явно или косвенно будем рассматривать далее в книге, а пока отметим в качестве иллюстрации один достаточно простой момент. В примере 1.1 параметр модели ***Rate*** мы представили как константу с

некоторым заданным значением. Решение это основано на вполне разумном предположении, что плата за кв. метр изменяется достаточно редко, и пользователю программы будет не слишком удобно вводить одно и то же число при каждом ее запуске. В то же время, если плата все-таки изменится, без участия программиста и наличия исходного текста программы обойтись не удастся. Как свести воедино эти противоречащие друг другу (на первый взгляд) требования? Решение может состоять, например, в следующем. Параметр ***Rate*** представляется в программе как переменная. В начале программы ей автоматически присваивается некоторое значение. Затем программа пытается найти и прочитать файл инициализации, в котором может храниться измененное значение параметра. Если файл найден, то ***Rate*** устанавливается равным значению, считанному из файла. Теперь при изменении платы за кв. метр пользователь программы сможет внести новое значение в файл инициализации, и программа будет функционировать правильно.

Выводы

Завершая первую главу, попытаемся подвести некоторые итоги. Мы рассмотрели – достаточно подробно для начального ознакомления – основные этапы на пути от возникновения задачи, для решения которой необходима вычислительная техника, до ее воплощения в программный код, а также обсудили некоторые моменты дальнейшей жизни получившейся программы. Отметили существенную взаимосвязанность всех этапов и общий циклический характер процесса, когда с каждого из этапов может потребоваться вернуться к одному из предыдущих для уточнения постановки задачи, адаптации модели, выбора более подходящего метода, формирования более эффективного алгоритма. Каждый из представленных этапов важен и занимает свое место в индустрии программирования. Вместе с тем, мы в нашей книге основное внимание сосредоточим на части общей технологической цепочки: “алгоритм – программа – отладка – модификация”.