Содержание:

Введение

Программирование - сравнительно молодая и быстро развивающаяся отрасль науки и техники. Опыт ведения реальных разработок и совершенствования, имеющихся программных и технических средств постоянно переосмысливается, в результате чего появляются новые методы, методологии и технологии, которые, в свою очередь, служат основой более современных средств разработки программного обеспечения.

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

Однако, несмотря на всю изменчивость и молодость компьютерного мира, уже существуют островки знания, признанные сообществом, как классические, без овладения которыми стать профессионалом невозможно.

Целью данной работы является изучение методов современного программирования.

Для реализации поставленной цели необходимо выполнить ряд задач:

- Изучить методологию и технологию программирования;

- Рассмотретть модульное программирование;

- Изучить структурное программирование;

- Рассмотреть метод объектно-ориентированного программирования и т.д.

Глава 1. Современная система и методы разработки программного обеспечения

1.1 Методология и технология программирования

Императивное программирование — это исторически первая методология программирования, которой пользовался каждый программист, программирующий на любом из «массовых» языков программирования – Basic, Pascal, C.[1] Она ориентирована на классическую фон Неймановскую модель, остававшуюся долгое время единственной аппаратной архитектурой. Методология императивного программирования характеризуется принципом последовательного изменения состояния вычислителя пошаговым образом. При этом управление изменениями полностью определено и полностью контролируемо.

Метод изменения состояний — заключается в последовательном изменении состояний. Метод поддерживается концепцией алгоритма.

Метод управления потоком исполнения — заключается в пошаговом контроле управления. Метод поддерживается концепцией потока исполнения.

Вычислительная модель. Если под вычислителем понимать современный компьютер, то его состоянием будут значения всех ячеек памяти, состояние процессора (в том числе — указатель текущей команды) и всех сопряженных устройств. Единственная структура данных — последовательность ячеек (пар «адрес» - «значение») с линейно упорядоченными адресами.

В качестве математической модели императивное программирование использует машину Тьюринга-Поста — абстрактное вычислительное устройство, предложенное на заре компьютерной эры для описания алгоритмов.

Синтаксис и семантика. Языки, поддерживающие данную вычислительную модель, являются как бы средством описания функции переходов между состояниями вычислителя. Основным их синтаксическим понятием является оператор.

Первая группа — простые операторы, у которых никакая их часть не является самостоятельным оператором (например, оператор присваивания, оператор безусловного перехода, вызова процедуры и т. п.). Вторая группа — структурные операторы, объединяющие другие операторы в новый, более крупный оператор

(например, составной оператор, операторы выбора, цикла и т. п.).[2]

Традиционное средство структурирования — подпрограмма (процедура или функция). Подпрограммы имеют параметры и локальные определения и могут быть вызваны рекурсивно. Функции возвращают значения как результат своей работы.

Если в данной методологии требуется решить некоторую задачу для того, чтобы использовать ее результаты при решении следующей задачи, то типичный подход будет таким. Сначала исполняется алгоритм, решающий первую задачу. Результаты его работы сохраняются в специальном месте памяти, которое известно следующему алгоритму, и используются им.

Программирование и отладка действительно больших программ (например, компиляторов), написанных исключительно на основе методологии императивного программирования, может затянуться на долгие годы.

1.2 Модульное программирование

Модульное программирование — это такой способ программирования, при котором вся программа разбивается на группу компонентов, называемых модулями, причем каждый из них имеет свой контролируемый размер, четкое назначение и детально проработанный интерфейс с внешней средой.[3] Единственная альтернатива модульности — монолитная программа, что, конечно, неудобно. Таким образом, наиболее интересный вопрос при изучении модульности — определение критерия разбиения на модули.

Концепции модульного программирования. В основе модульного программирования лежат три основных концепции:

Аксиома модульности Коуэна. Модуль — независимая программная единица, служащая для выполнения некоторой определенной функции программы и для связи с остальной частью программы.

Программная единица должна удовлетворять следующим условиям:

блочность организации, т. е. возможность вызвать программную единицу из блоков любой степени вложенности;

синтаксическая обособленность, т. е. выделение модуля в тексте синтаксическими элементами;

семантическая независимость, т. е. независимость от места, где программная единица вызвана;

общность данных, т. е. наличие собственных данных, сохраняющихся при каждом обращении;

полнота определения, т. е. самостоятельность программной единицы.

Сборочное программирование Цейтина. Модули — это программные кирпичи, из которых строится программа. Существуют три основные предпосылки к модульному программированию:

стремление к выделению независимой единицы программного знания. В идеальном случае всякая идея (алгоритм) должна быть оформлена в виде модуля;

потребность организационного расчленения крупных разработок;

возможность параллельного исполнения модулей (в контексте параллельного программирования).

Определения модуля и его примеры. Приведем несколько дополнительных определений модуля.

Модуль — это совокупность команд, к которым можно обратиться по имени.

Модуль — это совокупность операторов программы, имеющая граничные элементы и идентификатор (возможно агрегатный).

Функциональная спецификация модуля должна включать:

синтаксическую спецификацию его входов, которая должна позволять построить на используемом языке программирования синтаксически правильное обращение к нему;

описание семантики функций, выполняемых модулем по каждому из его входов.

Разновидности модулей. Существуют три основные разновидности модулей:

1) "Маленькие" (функциональные) модули, реализующие, как правило, одну какую-либо определенную функцию. Основным и простейшим модулем практически во

всех языках программирования является процедура или функция.

2) "Средние" (информационные) модули, реализующие, как правило, несколько операций или функций над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля.

Набор характеристик модуля предложен Майерсом [Майерс 1980]. Он состоит из следующих конструктивных характеристик:

1) размера модуля;

В модуле должно быть 7 (+/-2) конструкций (например, операторов для функций или функций для пакета). Это число берется на основе представлений психологов о среднем оперативном буфере памяти человека. Символьные образы в человеческом мозгу объединяются в "чанки" — наборы фактов и связей между ними, запоминаемые и извлекаемые как единое целое. В каждый момент времени человек может обрабатывать не более 7 чанков.[4]

Модуль (функция) не должен превышать 60 строк. В результате его можно поместить на одну страницу распечатки или легко просмотреть на экране монитора.

2) прочности (связности) модуля;

Существует гипотеза о глобальных данных, утверждающая, что глобальные данные вредны и опасны. Идея глобальных данных дискредитирует себя так же, как и идея оператора безусловного перехода goto. Локальность данных дает возможность легко читать и понимать модули, а также легко удалять их из программы.

3) сцепления модуля с другими модулями;

Сцепление (coupling) — мера относительной независимости модуля от других модулей. Независимые модули могут быть модифицированы без переделки других модулей. Чем слабее сцепление модуля, тем лучше. Рассмотрим различные типы сцепления.

Независимые модули — это идеальный случай. Модули ничего не знают друг о друге. Организовать взаимодействие таких модулей можно, зная их интерфейс и соответствующим образом перенаправив выходные данные одного модуля на вход другого. Сцепление по данным (параметрическое) — это сцепление, когда данные

передаются модулю, как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Этот вид сцепления реализуется в языках программирования при обращении к функциям (процедурам). Две разновидности этого сцепления определяются характером данным.

Сцепление по простым элементам данных.

Сцепление по структуре данных. В этом случае оба модуля должны знать о внутренней структуре данных.

4) рутинности (идемпотентность, независимость от предыдущих обращений) модуля.

В спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость, чтобы пользователи имели возможность прогнозировать поведение такого модуля.[5]

1.3 Структурное программирование

Структурное программирование (СП) возникло как вариант решения проблемы уменьшения СЛОЖНОСТИ разработки программного обеспечения.

В начале эры программирования работа программиста ничем не регламентировалась. Решаемые задачи не отличались размахом и масштабностью, использовались в основном машинно-ориентированные языки и близкие к ним язык типа Ассемблера, разрабатываемые программы редко достигали значительных размеров, не ставились жесткие ограничения на время их разработки.

По мере развития программирования появились задачи, для решения которых определялись ограниченные сроки все более сложных задач с привлечением групп программистов.

Таким образом, цель структурного программирования - повышение надежности программ, обеспечение сопровождения и модификации, облегчение и ускорение разработки.

Методология структурного императивного программирования — подход, заключающийся в задании хорошей топологии императивных программ, в том

числе отказе от использования глобальных данных и оператора безусловного перехода, разработке модулей с сильной связностью и обеспечении их независимости от других модулей.

Подход базируется на двух основных принципах:

Последовательная декомпозиция алгоритма решения задачи сверху вниз.

Использование структурного кодирования.

Напомним, что данная методология является важнейшим развитием императивной методологии.

Происхождение, история и эволюция. Создателем структурного подхода считается Эдсгер Дейкстра. Ему также принадлежит попытка (к сожалению, совершенно неприменимая для массового программирования) соединить структурное программирование с методами доказательства правильности создаваемых программ. В его разработке участвовали такие известные ученые как Х. Милс, Д.Э. Кнут, С. Хоор.[6]

Методы и концепции, лежащие в основе структурного программирования. Их три

Метод алгоритмической декомпозиции сверху вниз — заключается в пошаговой детализации постановки задачи, начиная с наиболее общей задачи. Данный метод обеспечивает хорошую структурированность. Метод поддерживается концепцией алгоритма.

Метод модульной организации частей программы — заключается в разбиении программы на специальные компоненты, называемые модулями. Метод поддерживается концепцией модуля.

Метод структурного кодирования — заключается в использовании при кодировании трех основных управляющих конструкций. Метки и оператор безусловного перехода являются трудно отслеживаемыми связями, без которых мы хотим обойтись. Метод поддерживается концепцией управления.

1.4 Метод объектно-ориентированного программирования

Метод структурного программирования оказался эффективен при написании программ «ограниченной сложности». Однако с возрастанием сложности реализуемых программных проектов и, соответственно, объема кода создаваемых программ, возможности метода структурного программирования оказались недостаточными.

Чтобы писать все более сложные программы, необходим был новый подход к программированию. В итоге были разработаны принципы Объектно-Ориентированного Программирования. ООР аккумулирует лучшие идеи, воплощённые в структурном программировании, и сочетает их с мощными новыми концепциями, которые позволяют по-новому организовывать ваши программы.[7]

Надо сказать, что теоретические основы ООП были заложены еще в 70-х годах прошлого века, но практическое их воплощение стало возможно лишь в середине 80-х, с появлением соответствующих технических средств.

Методология ООП использует метод объектной декомпозиции, согласно которому структура системы (статическая составляющая) описывается в терминах объектов и связей между ними, а поведение системы (динамическая составляющая) - в терминах обмена сообщениями между объектами. Сообщения могут быть как реакцией на события, вызываемые как внешними факторами, так и порождаемые самими объектами.

Объектно-ориентированные программы называют «программами, управляемыми от событий», в отличие от традиционных программ, называемых «программам, управляемыми от данных».

Основные методы и концепции ООП

Метод объектно-ориентированной декомпозиции – заключается в выделении объектов и связей между ними. Метод поддерживается концепциями инкапсуляции, наследования и полиморфизма.

Метод абстрактных типов данных – метод, лежащий в основе инкапсуляции. Поддерживается концепцией абстрактных типов данных.

Метод пересылки сообщений – заключается в описании поведения системы в терминах обмена сообщениями между объектами. Поддерживается концепцией сообщения.[8]

Вычислительная модель чистого ООП поддерживает только одну операцию – посылку сообщения объекту. Сообщения могут иметь параметры, являющиеся объектами. Само сообщение тоже является объектом.

Объект имеет набор обработчиков сообщений (набор методов). У объекта есть поля – персональные переменные данного объекта, значениями которых являются ссылки на другие объекты.

Синтаксис и семантика

В синтаксисе чистых объектно-ориентированных языков все может быть записано в форме посылки сообщений объектам. Класс в объектно-ориентированных языках описывает структуру и функционирование множества объектов с подобными характеристиками, атрибутами и поведением. Объект принадлежит к некоторому классу и обладает своим собственным внутренним состоянием. Методы — функциональные свойства, которые можно активизировать.

В объектно-ориентированном программировании определяют три основных свойства:

Инкапсуляция. Это сокрытие информации и комбинирование данных и функций (методов) внутри объекта.

Наследование. Построение иерархии порожденных объектов с возможностью для каждого такого объекта-наследника доступа к коду и данным всех порождающих объектов-предков. Построение иерархий является достаточно сложным делом, так как при этом приходится выполнять классифицирование.

Большинство окружающих нас объектов относится к категориям, рассмотренным в книге:

Реальные объекты – абстракции предметов, существующих в физическом мире;

Роли – абстракции цели или назначения человека, части оборудования или организации;

Инциденты – абстракции чего-то произошедшего или случившегося;

Взаимодействия – объекты, получающиеся из отношения между другими объектами.

Полиморфизм (полиморфизм включения) — присваивание действию одного имени, которое затем разделяется вверх и вниз по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, подходящим именно ему.

Глава 2. Проектирование программы

2.1 Установочные данные

Цель: написать программу для нахождения равновесной цены на рынке.

Обращаясь к курсу экономики, отметим, что зависимость спроса S и предложения D от цены товара P задаются некоторыми функциями S = S(P) и D = D(P). По мере роста цены товара спрос падает, а предложение увеличивается. Напротив, при падении цены товара происходит рост спроса и уменьшение предложения. В результате мы имеем ситуацию борьбы продавца и покупателя, цели которых противоположны.

Продавец стремиться держать цену как можно выше, покупатель – купить товар как можно дешевле. При этом при чрезмерном завышении цены товара покупатель перестает его приобретать, и, наоборот, при ее занижении продавец не хочет более торговать этим товаром. В результате работы рыночных механизмов происходит стабилизация цены товара на некотором уровне, приемлемом для всех субъектов рынка. Для нахождения этой равновесной цены необходимо решить уравнение S(P) = D(P), которое соответствует ситуации, когда весь товар распродается (спрос равен предложению).

Допустим, что вид функций S(P) и D(P) нам известен. Перенеся D(P) в левую часть, мы обнаруживаем, что задача определения соответствующей цены P* сводится к нахождению нулей некоторой известной функции f(P) = S(P) – D(P).

Из курса математики известно, что, если функция y = f(x) является *н е п р е р ы в н о й* на [a, b] и f(a) · f(b) < 0, то эта функция имеет *корень* на [a, b], т.е. такое x = x_0, что x_0∈[a, b], f(x_0) = 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* .

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

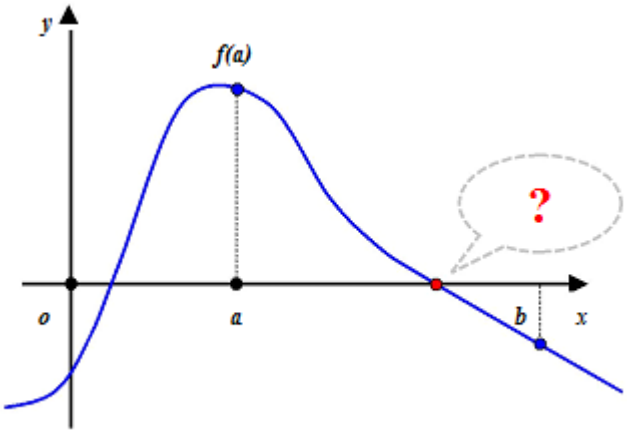

Рис. 1. Корни уравнения f(x) = 0

Для численного нахождения корней уравнения f(x) = 0 разработано немало методов. Среди них широко известными являются так называемые итерационные методы, которые строят цепочку точек x1, x2, …, xn, последовательно приближаясь к решению.

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

Рассмотрим кратко один из методов – м е т о д половинного деления.

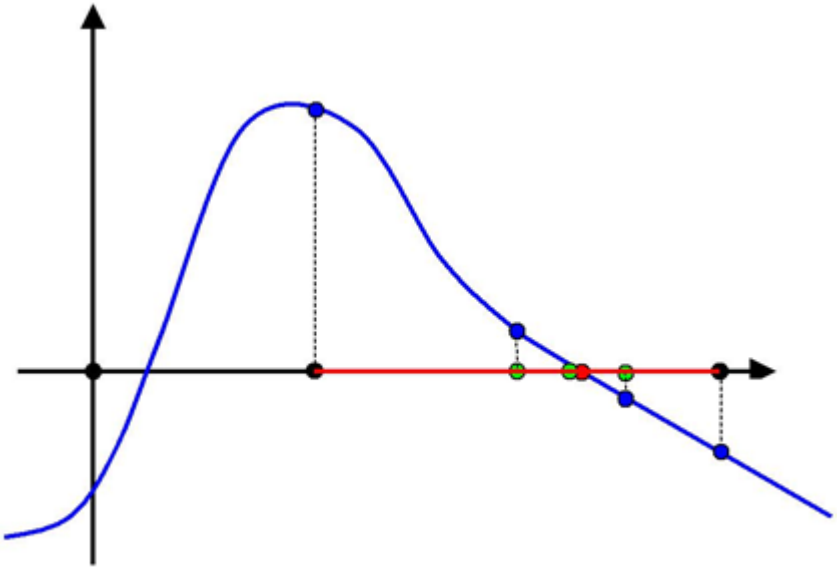

Рис. 2. Метод половинного деления

Итак, решение любой 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.

2.2 Постановка задачи

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” [40].

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.

2.3 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.

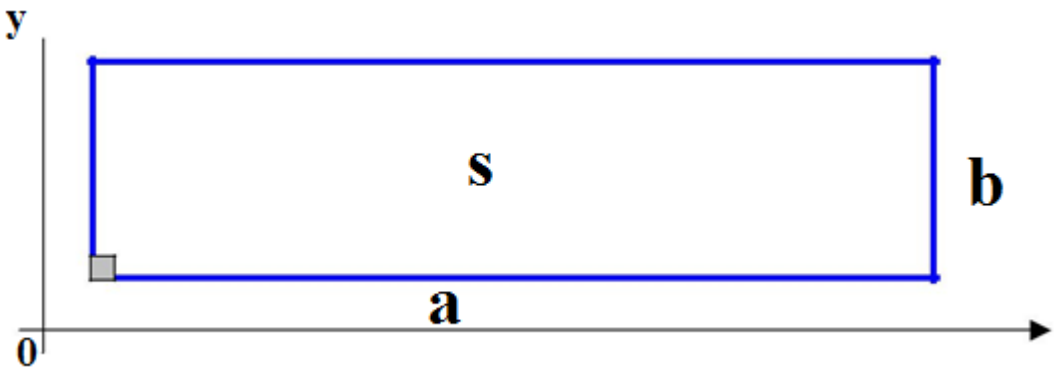

Рис. 3. Модель земельного участка прямоугольной формы

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.

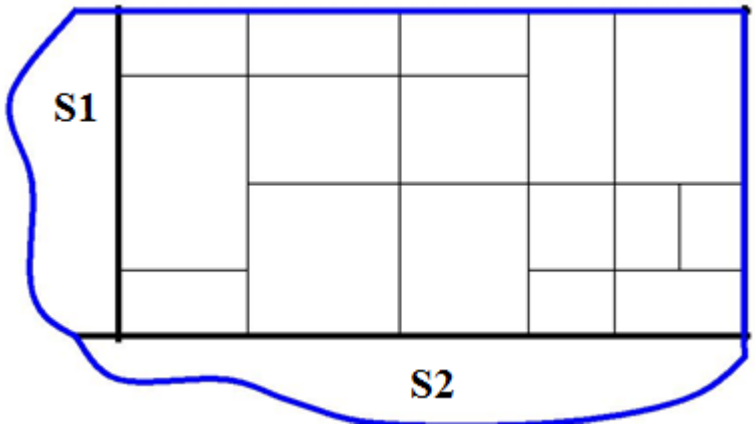

Рис. 4. Модель земельного участка общего вида

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

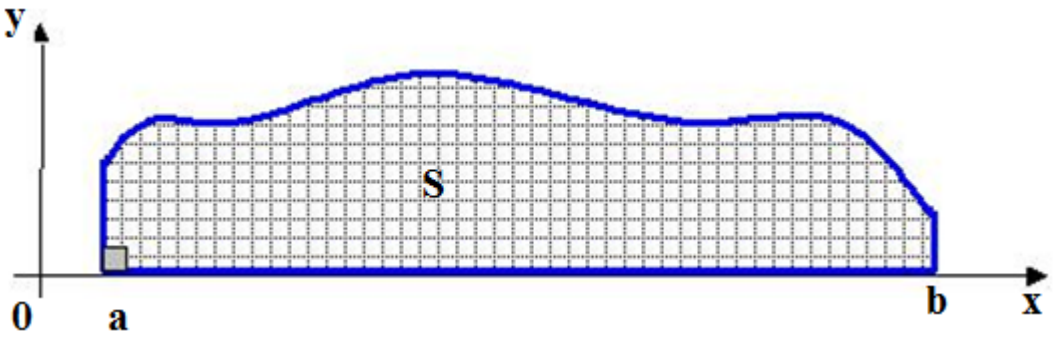

Рис. 5. Криволинейная трапеция

Внесем изменения в систему параметров.

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 .

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

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.

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. 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 втекстовом 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:

Иванов И.И.

3

40 37 50 45

7 20 10

Теперь нужно решить вопрос с представлением результатов расчетов. Например, они могут требоваться в виде справки на бумаге (то есть программа должна вывести их на принтер) 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.

Вообще, для представления алгоритма в литературе в настоящий момент чаще всего применяется следующая форма описания: словесное изложение с элементами того или иного языка программирования, как правило, Pascal. 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 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”, то есть 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.

2.4 Создание программы

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 Pascal. 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 }

Program Rent; { программа Rent (заголовок) }

{ $APPTYPE CONSOLE}

uses Printers, SysUtils; { использует стандартные библиотеки }

{ декларативная часть программы – различные объявления } { константы }

const = 20;

MaxSeg { максимальное число отрезков разбиения }

Rate = 100; { плата за 1 кв. метр в рублях }

{ типы данных } type

Coord = array [0..MaxSeg] of Real; { вектор вещественных }

{ координат из MaxSeg + 1 }

{ элементов }

{ переменные и их типы } var

AreaData: Text; { текстовый файл на диске }

y, h: Coord; { векторы длин оснований трапеций }

{ и длин отрезков разбиения }

{ (высот трапеций) }

N, i: Integer; { текущее число отрезков разбиения }

Number, { и переменная цикла (целые числа) }

{ номер участка }

Name: string; { и имя владельца (текстовые строки) }
```

Area, { площадь и }

Cost: Real; { стоимость (вещественные числа) }

{ начало исполняемой части программы } begin

AssignFile(AreaData, 'AREADATA.TXT'); { присвоение значения } { переменной типа файл }

{ конкретного }

{ имени файла на диске }

{ (связывание) }

{$I-} { директива компилятору: отключить автоматическую }

{ проверку результата операции ввода/вывода }

Reset(AreaData); { поиск и открытие файла на чтение }

if IOResult <> 0 then { если файл не найден, то вывод на дисплей }

begin 1: Не найден файл AREADATA.TXT');

Writeln('ОШИБКА

Halt { завершение работы }

end;

{$I+}

{ чтение с диска }

Readln(AreaData, Number); { номер участка } Readln(AreaData, Name); { имя владельца }

Readln(AreaData, N); { количество отрезков разбиения }

if

N > MaxSeg

then

```
{ если количество отрезков больше максимально } { допустимого, то вывод на дисплей }

begin

Writeln('ОШИБКА 2: Недопустимо большое число отрезков');

Halt { завершение работы } end;

{ чтение с диска } for i := 0 to N do

Read(AreaData, y[i]); { длины оснований трапеций }

Readln(AreaData);{ перейти на следующую строку файла }

for i := 1 to N do

Read(AreaData, h[i]); { длины высот трапеций }

{ вычисление площади участка }

{ "метод трапеций" } Area := 0;

for i := 1 to N do

Area := Area + (y[i - 1] + y[i]) * h[i]; Area := Area / 2;

Cost := Area * Rate; { арендная плата }

{ вывод на принтер } with Printer do begin

BeginDoc;

Canvas.TextOut(10, 10, 'Участок ' + Number); Canvas.TextOut(10, 210, 'Владелец ' + Name); Canvas.TextOut(10, 410, 'Площадь участка ' + FloatToStr(Area) +

' кв. м');

Canvas.TextOut(10, 610, 'Арендная плата ' + FloatToStr(Cost) + ' руб.'); EndDoc;

end;

{ Закрытие файла } CloseFile(AreaData); { конец программы }

end.
```

```
{==========================================================
}
```

Специалист, взявший на себя труд разобрать текст программы, должен заметить, что собственно расчетная ее часть занимает всего лишь пять строк из общего текста. Может быть эта ситуация вызвана простотой, решаемой нами задачи? На самом деле нет. Опыт показывает, что 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.п.). 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 о ш и б о к чудных готовит просвещенья дух”. 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.

Завершая данную 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”.

Заключение

Изначально программирование имело крайне примитивный вид и практически не имело отличий от упорядоченного бинарного кода с формализованным подходом. По сути, при зарождении сферы отличий языка программирования от компьютерного кода было немного. Очевидных и естественных удобств для программиста не существовало, он обязан был обладать знаниями числовых кодов для каждой команды машины. Даже распределение памяти для выполнения команд ложилось на специалиста.

Для упрощения обращения с ЭВМ люди стали активно разрабатывать языки, одним из первых стал Ассемблер. Для отображения переменных стали использоваться символьные наименования. Вместо числовых операций человеку достаточно знать мнемонические имена, их запоминание в разы облегчалось. Уже на этом этапе языки программирования стали более приближёнными к понятному для человека языку.

К первооткрывателям среди языков программирования относится Фортран – это сокращённое сочетание 2 слов: Formula и Translation. Создан уже в середине 50-х. До сих пор язык используется благодаря лёгкости и простоте написания, а также развитой системе библиотек для Фортран. Чаще используется для научных и инженерных подсчётов, а также активно применяется в физичке и остальных

науках, связанных с математикой.

В процессе написания данной курсовой работы была изучена история возникновения самого программирования. Также была проведена систематизация знаний о подходах и принципах создании новых языков программирования. В заключении следует отметить, что рассмотренная тема, позволяет просмотреть путь становления технологий и языков программирования и является интересной с точки зрения специалиста в области информационных технологий.

Список использованной литературы

1. Алехин, В.А. Микроконтроллеры PIC: основы программирования и моделирования в интерактивных средах MPLAB IDE, mikroC, TINA, Proteus. Практикум / В.А. Алехин. - М.: ГЛТ , 2016. - 248 с.
2. Ашарина, И.В. Основы программирования на языках С и С++: Курс лекций для высших учебных заведений / И.В. Ашарина. - М.: ГЛТ, 2016. - 208 с.
3. Ашарина, И.В. Основы программирования на языках С и С++ / И.В. Ашарина. - М.: ГЛТ, 2015. - 208 с.
4. Биллиг, В.А. Основы программирования на C#: Учебное пособие / В.А. Биллиг. - М.: Бином, 2016. - 483 с.
5. Богачев, К.Ю. Основы параллельного программирования: Учебное пособие / К.Ю. Богачев. - М.: Бином, 2014. - 342 с.
6. Богачев, К.Ю. Основы параллельного программирования / К.Ю. Богачев. - М.: Бином, 2015. - 342 с.
7. Воскобойников, Ю.Е. Основы вычислений и программирования в пакете MathCAD PRIME: Учебное пособие / Ю.Е. Воскобойников и др. - СПб.: Лань, 2016. - 224 с.
8. Гулиа, Н.В. Основы вычислений и программирования в пакете MathCAD PRIME: Учебное пособие / Н.В. Гулиа, В.Г. Клоков, С.А. Юрков. - СПб.: Лань, 2016. - 224 с.
9. Культин, Н.Б. Самоучитель. Основы программирования в Delphi 8 для MS.NET Framework / Н.Б. Культин. - СПб.: BHV, 2014. - 400 с.
10. Культин, Н.Б. Основы программирования в Delphi XE / Н.Б. Культин. - СПб.: BHV, 2016. - 416 с.
11. Культин, Н.Б. Основы программирования в Delphi 2006 для Windows / Н.Б. Культин. - СПб.: BHV, 2016. - 384 с.

12. Культин, Н.Б. Основы программирования в Turbo Delphi / Н.Б. Культин. - СПб.: BHV, 2016. - 384 с.
13. Культин, Н.Б. Основы программирования в Turbo C++ / Н.Б. Культин. - СПб.: BHV, 2016. - 464 с.
14. Кундиус, В.А. Теоретические основы разработки и реализации языков программирования / В.А. Кундиус. - М.: КноРус, 2015. - 184 с.
15. Маркин, А.В. Основы Web-программирования на PHP / А.В. Маркин. - М.: Диалог-МИФИ, 2016. - 252 с.
16. Маркин, А.В. Основы web-программирования на PHP / А.В. Маркин, С.С. Шкарин. - М.: Диалог-МИФИ, 2016. - 252 с.
17. Окулов, С.М. Основы программирования, перераб / С.М. Окулов. - М.: Бином, 2015. - 336 с.
18. Окулов, С.М. Основы программирования / С.М. Окулов. - М.: Бином. Лаборатория знаний, 2016. - 336 с.
19. Семакин, И.Г. Основы алгоритмизации и программирования: Учебник для студ. учреждений сред. проф. образования / И.Г. Семакин, А.П. Шестаков. - М.: ИЦ Академия, 2016. - 400 с.
20. Семакин, И.Г. Основы алгоритмизации и программирования. Практикум: Учебное пос. для студ. учреждений сред. проф. образования / И.Г. Семакин, А.П. Шестаков . - М.: ИЦ Академия, 2016. - 144 с.
21. Семакин, И.Г. Основы алгоритмизации и программирования: Учебник для студ. учреждений сред. проф. образования / И.Г. Семакин, А.П. Шестаков . - М.: ИЦ Академия, 2016. - 304 с.
22. Фридман, А. Основы объектно-ориентированного программирования на языке СИ++ / А. Фридман. - М.: Горячая линия -Телеком, 2015. - 234 с.
23. Черпаков, И.В. Основы программирования: Учебник и практикум для прикладного бакалавриата / И.В. Черпаков. - Люберцы: Юрайт, 2016. - 219 с.
24. Черпаков, И.В. Основы программирования: Учебник и практикум для СПО / И.В. Черпаков. - Люберцы: Юрайт, 2016. - 219 с.

1. Алехин, В.А. Микроконтроллеры PIC: основы программирования и моделирования в интерактивных средах MPLAB IDE, mikroC, TINA, Proteus. Практикум / В.А. Алехин. - М.: ГЛТ , 2016. - 248 с. ↑

2. Маркин, А.В. Основы web-программирования на PHP / А.В. Маркин, С.С. Шкарин. - М.: Диалог-МИФИ, 2016. - 252 с. ↑

3. Черпаков, И.В. Основы программирования: Учебник и практикум для СПО / И.В. Черпаков. - Люберцы: Юрайт, 2016. - 219 с. ↑

4. Семакин, И.Г. Основы алгоритмизации и программирования: Учебник для студ. учреждений сред. проф. образования / И.Г. Семакин, А.П. Шестаков . - М.: ИЦ Академия, 2016. - 304 с. ↑

5. Гулиа, Н.В. Основы вычислений и программирования в пакете MathCAD PRIME: Учебное пособие / Н.В. Гулиа, В.Г. Клоков, С.А. Юрков. - СПб.: Лань, 2016. - 224 с. ↑

6. Ашарина, И.В. Основы программирования на языках С и С++: Курс лекций для высших учебных заведений / И.В. Ашарина. - М.: ГЛТ, 2016. - 208 с. ↑

7. Маркин, А.В. Основы Web-программирования на PHP / А.В. Маркин. - М.: Диалог-МИФИ, 2016. - 252 с. ↑

8. Воскобойников, Ю.Е. Основы вычислений и программирования в пакете MathCAD PRIME: Учебное пособие / Ю.Е. Воскобойников и др. - СПб.: Лань, 2016. - 224 с. ↑