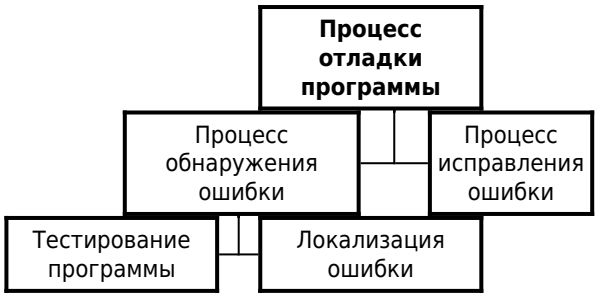
Глава VIII. Некоторые вопросы методологии отладки программ

Я услышал и забыл.
Я увидел и запомнил.
Я сделал и понял!

—Г.Клейман



VIII.1. Ошибки при программировании

Какую бы программу вы ни писали,
любая ошибка, которая может в
нее вкрасться, — вкрадется!

—Следствие первого закона Чизхолма

Будем говорить, что «в программе имеется *ошибка*, если её выполнение не оправдывает ожиданий пользователя» [50], [51].

Напомним, что при решении задач с использованием компьютера под *отладкой* программ понимается обычно один из этапов решения, во время которого с помощью компьютера происходит обнаружение и исправление ошибок, имеющихся в программе; в ходе отладки программист хочет добиться определённой степени уверенности в том, что его программа соответствует своему назначению и не делает того, для чего она не предназначена.

Все мы делаем ошибки при программировании. Даже программисты с двадцатилетним опытом работы допускают их десятками. Разница между хорошим и плохим программистом заключается не в том, что первый не делает ошибок, а скорее в том, что он делает значительно меньше *простых* ошибок.

Начинающий программист, как правило, переоценивает свои возможности и, разрабатывая программу, исходит из того, что в его программе ошибок не будет. А говоря про только что составленную программу, готов уверять, что она на 99% правильна, и её остаётся только для большей уверенности один(!) раз выполнить на компьютере с какими–нибудь(!) исходными данными. Естественно, что каждый неверный результат, каждая найденная ошибка вызывают у него изумление и считаются, *конечно*, последними. Вследствие такого подхода получение с помощью компьютера надёжных результатов по составленной программе откладывается на длительный и неопределённый срок. Только приобретя достаточный опыт, программист понимает справедливость древнего высказывания: «Человеку *свойственно ошибаться*!».

Оказывается, что практически невозможно для достаточно сложной программы быстро найти и устранить все имеющиеся в ней ошибки. Трудности программирования и отладки подчёркивает следующий популярный в среде программистов афоризм: «В *каждой* программе есть по крайней мере одна ошибка». Поэтому можно сказать, что наличие ошибок в только что разработанной программе — вполне нормальное и закономерное явление. А совсем ненормальным, из ряда вон выходящим фактом является отсутствие ошибок в программе, которая не была ещё подвергнута тщательному тестированию и отладке (конечно, речь здесь идёт о достаточно сложных программах!).

Поэтому разумно уже при разработке программы на этапах алгоритмизации и программирования *готовиться к*

обнаружению ошибок на стадии отладки и принимать профилактические меры для их предупреждения.

Ошибки можно объединить в следующие группы [50], [51]:

1. ошибки обращения к данным — использование переменных с неустановленными значениями, ошибки индексации, несоответствие структур данных;
2. ошибки описания данных — отсутствие явного описания или неполное описание данных, отсутствие или неправильное присвоение начальных значений, несогласованность инициализации переменной с её описанием;
3. ошибки вычислений — наличие в последовательных вычислениях данных недопустимых типов, несогласованность масштабов, приводящая к переполнению или потере точности, возможность деления на нуль;
4. ошибки при сравнениях — использование при операциях сравнения величин несовместимых типов, искажение смысла операций отношения (>, =, <) и логических операций (NOT, AND, OR), сравнение чисел с фиксированной и плавающей запятой;
5. ошибки в передачах управления — ошибки организации циклов, приводящие к возможности зацикливания или неправильного выполнения цикла, наличие неполного числа выходов в операторах–переключателях;
6. ошибки программного *интерфейса* — несоответствие количества, типов или размерности фактических и формальных параметров подпрограмм при их вызове, несоответствие описаний переменных требованиям на выходе модуля, несогласованность описаний глобальных переменных и их интерпретации операторами программы (*модуль* — это замкнутая программа, которую можно вызвать из любого другого модуля в программе и можно отдельно компилировать). Напомним, что программный *интерфейс* определяет совокупность допустимых процедур или операций и их параметров, список общих переменных, областей памяти или других объектов;
7. ошибки ввода–вывода — неполное или неправильное описание атрибутов файлов или оператора обращения к файлу, несогласованность размера записи и выделенной памяти, неполный контроль и регистрация операций с файлами;
8. помехозащита — отсутствие контроля входных данных, отсутствие сохранения исходных данных и возможности повторного запуска модуля при сбоях.
9. никогда не считайте, что Вы точно знаете причину ошибочного выполнения программы; очень часто в этом виновна ошибка, встретившаяся гораздо раньше (иногда её называют *отложенной* ошибкой).
10. и, наконец, ошибки могут быть также следствием неверной работы оборудования — это так называемые *аппаратные* ошибки. Если в регистр памяти компьютера на одном из этапов работы программы занесено число 12, — а при чтении из этого же регистра оно прочиталось как 11, то и дальнейшие результаты, разумеется, будут неверными. Возможен случай, когда из–за такой ошибки результат вовсе не будет получен(процесс решения задачи аварийно прекратится).

Разработаны надёжные методы борьбы с аппаратными ошибками и их последствиями — *повторные* вычисления с последующим сравнением результатов, хранение нескольких экземпляров данных с целью их защиты от искажения и т.д. Поэтому среди встречающихся на практике случаев выдачи компьютерами неверных результатов или невыдачи их вообще доля ошибок, порождённых аппаратными средствами, составляет ничтожный процент.

Так, согласно одному из определений «ПЭВМ — это вычислительная машина с надёжностью военной аппаратуры и ценой изделия бытовой электроники»[58].

Таким образом, в ошибочных ответах компьютера виноваты, как правило, люди!

Приведённая классификация полезна тем, что для каждой из перечисленных групп ошибок и для каждого типа ошибки в группе можно выделить операторы каждого конкретного языка программирования, потенциально допускающие данный тип ошибок.

Некоторые ошибки являются *систематическими*; они возникают всякий раз при выполнении программы. Если в команде на вычисление по заданной формуле вместо плюса поставить минус, то и ответ, скорее всего, будет ошибочным. Такие ошибки в программах обычно живут недолго: их быстро обнаруживают и исправляют. Приведём интересный пример систематической ошибки в программном интерфейсе.

19 июня 1985 года команда американского космоплана многоразового использования («Шаттла») должна была развернуть свой корабль так, чтобы зеркало на его борту могло отражать луч лазера, находившегося на горе высотой 10023 фута. Навигационная система пыталась развернуть «Шаттл» так, чтобы принимать луч с вершины несуществующей горы высотой в 10023 морских миль над уровнем моря. Это произошло из–за того, что один из пары взаимосвязанных компонентов программно–аппаратного комплекса передавал высоту в футах, а другой — интерпретировал эту величину в милях.

Другие ошибки носят *случайный* характер; при каждом выполнении программы они будут приводить к разным результатам, либо программа может выполняться в большинстве случаев правильно, но время от времени неожиданно давать неверный результат. Такие случайные ошибки следует стараться выявить на этапе ручной проверки, потому что при машинном выполнении программы они могут «исчезнуть» лишь для того, чтобы снова появиться позже.

В некотором смысле достаточно сложная программа напоминает карточный домик: тот факт, что домик стоит, ещё не гарантирует, что он не рассыплется в следующее мгновение. Программы опровергают опыт нашей жизни. Обычно, если что–то работает, то оно работает! Если новый стул выдержал Вас, он выдержит Вас и в следующий раз; если сошедший с конвейера автомобиль проехал один километр, он сможет проехать ещё сотни километров; если здание простояло 5 минут, то, как уверяют строители и архитекторы, оно простоит ещё сто лет.

Однако на основании того факта, что часть программы работает, ничего нельзя сказать о работоспособности остальной части программы!

Показательны в этом отношении результаты первых попыток запуска «Шаттла». Программное обеспечение этого космического корабля состояло примерно из полумиллиона программных строк, над которыми трудился большой коллектив разработчиков. Корабль не смог оторваться от Земли из–за нарушения синхронизации всех его компьютеров. Оказалось, что программная ошибка, явившаяся причиной неудачи, была внесена нечаянно при исправлении другой ошибки, обнаруженной двумя годами раньше, и могла проявляться в среднем только в одном из 67 полётов.

Одна из наиболее типичных случайных ошибок возникает, если Вы забыли инициализировать переменную, т.е. присвоить ей начальное значение. В этом случае начальное значение переменной зависит от того, какая программа (назовём её P) выполнялась на компьютере перед тем, как была загружена отлаживаемая программа. Если Ваша переменная имеет адрес α, то программа P может оставить после своего выполнения в ячейке по адресу α «все, что угодно» — код команды, значение переменной, значение адреса переменной (*мусор*, т.е. произвольное, непредсказуемое значение).

Ещё одной часто встречающейся случайной ошибкой является запись данных в массив, когда значение индекса вышло за допустимые пределы. Если, например, Вы присваиваете начальное значение элементу массива T(J), а значение индекса J должно находиться в границах от 1 до 100, но J случайно оказалось в какой–то момент больше 100 либо меньше 1, то Вы, разумеется, получите не тот результат, который хотели. Если Вы часто совершаете ошибку такого рода, вам полезно написать несколько операторов, которые будут проверять значения каждого индекса перед его использованием и фиксировать его выход из установленного диапазона. Конечно, это увеличивает время выполнения программы, но заметно ускоряет процесс отладки.

В заключение — *полезный* совет: **старайтесь вести список допущенных Вами ошибок**.

Каждый раз, когда Вы обнаруживаете ошибку, заносите её в этот список и обязательно проверяйте, чтобы она не повторилась в дальнейшем!

> Чтобы избегать *ошибок*, надо набираться *опыта*,
> чтобы набираться *опыта*, надо делать *ошибки*.
>
> *—Принцип Компетентности по Питеру*

VIII.2. Некоторые классические приёмы тестирования программ

> Самая коварная уловка дьявола состоит в том,
> чтобы убедить нас, будто его не существует.
>
> *—Ш.Бодлер*

Известно, что в процессе разработки программы работы по доказательству *(демонстрации)* правильности разрабатываемой программы равнозначны работам по её изготовлению (алгоритмизации и написанию), что можно

выразить следующей формулой:

Разработка программы = Изготовление + Доказательство.

Поэтому *программой* следовало бы называть только такую программу, которая выдаёт правильные результаты, а то, что ещё не прошло стадию доказательства правильности, является не программой, а её *полуфабрикатом*. Изготовление такого полуфабриката, конечно, является делом несравнимо более лёгким, чем разработка *настоящей* программы (особенно если программист и не думает о предстоящей отладке!).

VIII.2.1. Ручная проверка

> Если вам кажется, что ситуация улучшается, значит, вы чего–то не заметили!
>
> *—Второе следствие второго закона Чизхолма*

Отладку любой программы никогда не следует начинать с прогона программы на компьютере, т.к. экспериментально установлено, что «ручными» методами (т.е. без помощи компьютера) удаётся обнаруживать от 30 до 70% программных и аналитических ошибок из общего числа ошибок, допущенных при программировании!

Вначале обязательно проведите *ручную проверку* («desk checking»), которая есть не что иное, как тщательная проверка Вашей программы за письменным столом. При ручной проверке программы (или алгоритма) программист по тексту программы мысленно старается восстановить тот вычислительный процесс, который определяет программа, после чего сверяет его с требуемым процессом.

Самое главное, о чем всегда следует помнить, это то, что ошибки в проверяемой программе *обязательно* есть и, чем больше их будет обнаружено за *столом*, тем легче и быстрее пройдёт предстоящий этап отладки программы на компьютере. На время проверки нужно постараться «забыть» о том, что должен делать проверяемый участок программы, и «узнавать» об этом по ходу его проверки. Только после окончания проверки участка и выявления тем самым его действительных функций можно «вспомнить» о том, что он должен делать, и сравнить реальные действия программы с требуемыми.

Полезно, найдя какую–либо специфическую ошибку, отойти от последовательной проверки и сразу узнать, нет ли таких же ошибок в аналогичных, в особенности уже проверенных, местах.

Приведём примерный список вопросов, на которые отвечает программист при *ручной проверке* программы.

1. Есть ли обращения к переменным, которым не присвоены значения?
   «Присваивайте начальные значения переменным перед тем, как они будут использованы. Убедитесь в том, что переменным в программах и во внутренних циклах присваиваются соответствующие значения при каждом входе в них» [56].
2. Не выходит ли значение индекса элемента массива за границы, определённые для соответствующего измерения, при всех обращениях к массиву?
   «Проверьте, чтобы индексы при обращении к элементам массива не выходили за границы»[56].
3. Принимает ли каждый индекс целые значения при всех обращениях к массиву? Нецелые индексы не являются ошибкой для многих языков программирования, но представляют практическую опасность.
4. Все ли переменные описаны явно? Отсутствие явного описания не обязательно является ошибкой, но обычно служит источником беспокойства.
5. Есть ли переменные со сходными именами (например, VOLT и VOLTS)? Наличие сходных имён не обязательно является ошибкой, но служит признаком того, что имена могут быть перепутаны где–нибудь внутри программы.
6. Возможны ли переполнение или исчезновение порядка во время вычисления значения выражения? Это означает, что конечный результат может казаться правильным, но промежуточный результат может быть слишком большим (переполнение) или слишком малым (исчезновение порядка) для машинного представления данных.
7. Учтено ли, что делитель при делении может обратиться в нуль?
8. Может ли значение переменной выходить за пределы установленного для её типа диапазона?
9. Сравниваются ли величины различных типов?

10. Корректны ли операции сравнения? Обычно часто путают такие операции отношения, как «не меньше, чем», «не больше, чем».
11. Каждое ли логическое выражение сформулировано так, как это предполагалось? Программисты часто делают ошибки при написании логических выражений, содержащих операции NOT, AND, OR.
12. Если в программе содержится оператор–переключатель `ON k GOTO m1, m2, …, mn`, то может ли значение переменной k превысить n? Например, всегда ли k будет принимать значения 1, 2 или 3 в операторе `ON k GOTO` 10, 20, 30 ?
13. Будет ли каждый цикл в конце концов завершён? Придумайте неформальное доказательство или аргументы, подтверждающие их завершение.
    «Не используйте числа с плавающей запятой в качестве значений счётчиков. Не надейтесь, что для дробных величин с плавающей запятой справедливы известные правила арифметики — это не так»[56].
14. Будет ли программа или подпрограмма в конечном счёте завершена?
15. Возможно ли, что некоторый цикл никогда не сможет выполниться? Если это так, то является ли это оплошностью?
    «Проверьте, могут ли циклы в программе при определённых обстоятельствах выполняться нулевое число раз»[56].
16. Совпадают ли количество и тип формальных и фактических параметров используемых подпрограмм?
17. Совпадают ли единицы измерения значений соответствующих фактических и формальных параметров? Например, нет ли случаев, когда значение фактического параметра выражено в градусах, а в подпрограмме все расчёты проводятся с формальным параметром, выраженным в радианах.
18. Не изменяет ли подпрограмма значения переменной, которая используется только как входная величина?
19. Все ли файлы открыты перед их использованием?
20. Существуют ли смысловые или грамматические ошибки в тексте, выводимом программой на печать или на экран дисплея?

VIII.2.2. Ручная прокрутка. Методические указания по её проведению

> Программисту не всегда нужна ЭВМ, иногда полезнее удобное кресло и спокойная обстановка.
>
> *—А.Архангельский*

После окончания ручной проверки проведите несколько раз *ручную прокрутку* («walkthrough» — «сквозной контроль») отдельных частей Вашей программы. Иногда её называют *«сухой» прокруткой* («dry running» — «пробный прогон») в отличие от метода прокрутки, использующего компьютер. Основой *прокрутки* является имитация программистом процесса выполнения программы (алгоритма) компьютером с целью более конкретного и наглядного представления о процессе, определяемом *текстом* проверяемой программы. Прокрутка даёт возможность приблизить последовательность проверки программы к последовательности её выполнения, что позволяет проверять программу как бы в динамике её работы, проверять элементы вычислительного процесса, задаваемого проверяемой программой, а не только статичный текст программы.

Для выполнения прокрутки обычно приходится задавать какие–то конкретные исходные данные и производить над ними необходимые вычисления, используя текст программы. Для программ со сложной логикой, в которых, например, характер работы одного участка программы зависит от результатов работы других её участков,необходимо осуществлять ручную прокрутку программы для ряда специально подобранных исходных данных и параметров. Прокрутка даёт программисту возможность найти более хитрые ошибки в программе, чем при ручной проверке.

Трудность применения прокрутки — большой объем ручной работы при попытке точного моделирования работы программы. Поэтому успех применения прокрутки заключается в выборе такой необходимой степени детализации моделирования, чтобы, с одной стороны, выявить максимальное количество ошибок, а с другой — затратить на это минимальные усилия.

Приведём несколько соображений, которые могут помочь уменьшить время, затрачиваемое на прокрутку.

Прокрутку следует применять лишь для контроля логически сложных программ или *блоков* (под *блоком* будем понимать некоторую группу операторов, объединяемых по какому–либо признаку, например: арифметический блок — выполняемая последовательно группа операторов, производящих вычисления в программе, логический блок — группа операторов, управляющих последовательностью вычислений в программе).

Арифметические блоки нужно проверять обычным способом, не задаваясь конкретными исходными данными. Вычислять числовые значения нужно лишь для тех величин, от которых зависит последовательность выполнения блоков (операторов) программы, и эта последовательность является очень существенной. Поэтому во время прокрутки программы при всякой возможности, когда позволяет характер прокручиваемого блока программы, нужно переходить на обычную ручную проверку и возвращаться на режим прокрутки при начале проверки логически сложных блоков.

Исходные данные, влияющие на логику программы, должны выбираться так, чтобы минимизировать прокрутку программы. Но данные должны быть и такими, чтобы в прокрутку вовлеклось большинство ветвей программы и чтобы прокрутка отразила типичный характер ее работы.

Кроме того, в ходе прокрутки необходимо проверить работу программы и для особых случаев (например, для экстремальных значений параметров).

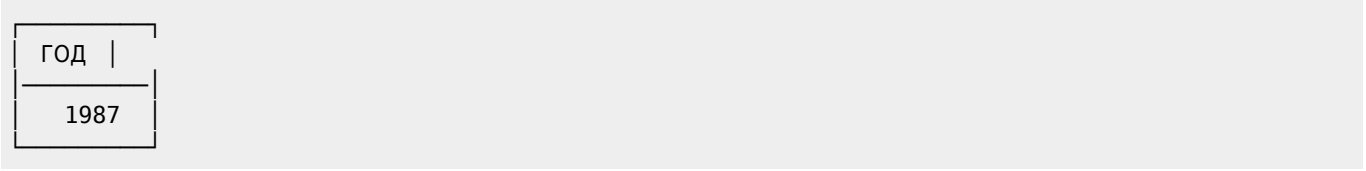
Многократные повторные прокрутки какого–либо участка программы можно не производить, если в логике его выполнения ничего не изменяется по сравнению с предыдущими проходами. Например, тело цикла можно прокрутить лишь для *первых* двух–трёх проходов (проверка входа в цикл) и для *последних* одного–двух (проверка выхода из цикла).

Прокрутка бывает необходимой и в том случае, когда программист не в состоянии вполне чётко представить себе логику проверяемой программы, особенно **если программа написана не им и нет хорошего описания**.

При первом же пробном запуске вычислительной машины МЭСМ (первая в СССР ЭВМ, 1951 г.) произошёл показательный случай. Первую программу для МЭСМ перед запуском прокрутили вручную два квалифицированных математика и получили одинаковые результаты. А машина выдала другой результат. Инженеры долго искали неисправность и не смогли её найти. Тогда академик С.А. Лебедев, главный конструктор МЭСМ, сам взялся за ручную прокрутку. Проработав всю ночь, он обнаружил, что оба математика ошиблись в одном и том же месте, а машина оказалась права!

Поговорим теперь о *методике* проведения ручной прокрутки.

Нарисуйте на листе бумаги «начальную обстановку», а затем — «исполняйте» операторы по одному, отмечая все изменения, происходящие при этом в запоминающем устройстве. Поэтому для ручной прокрутки нужно уметь изображать на бумаге начальную обстановку в Оперативном Запоминающем Устройстве и происходящие в нем изменения. Разумеется, нас интересуют не все *блоки* (участки) памяти, а только те из них, которые используются в программе. Удобнее всего рисовать блоки и их значения мелом на школьной доске: ведь для того, чтобы поместить в блок новое содержимое, нужно «уничтожить» (стереть) старое — на доске это сделать очень просто. Каждый блок можно рисовать в виде «домика», на «крыше» которого записано имя блока, а внутри размещается содержимое. Форма крыши может говорить о типе переменной.



Например, на рисунке изображён блок, имя которого ГОД, содержимое блока — 1987, тип — строковый, т.к. крыша прямоугольная.

Если доски нет, ручную прокрутку можно вести и на бумаге. При этом таблицу имён (блоков) приходится изображать немножко по–другому, потому что стирать старые значения на бумаге неудобно, лучше их зачёркивать, а рядом писать новые значения.

Ручную прокрутку лучше всего проводить *вдвоём* с приятелем. Выполняйте строку за строкой программы (алгоритма) и старайтесь независимо друг от друга обнаружить ошибки. Однако избегайте такой методики в том случае, если выяснится, что Вы испытываете искушение переложить на своего приятеля (или он на Вас) заботу о тестировании программы!

В заключение необходимо отметить, что использование прокрутки весьма полезно ещё и потому, что она содействует глубокому осознанию программистом логики составленной им программы и того реального вычислительного процесса, который ею задаётся. Ведь быстрота ориентирования в отлаживаемой программе и в выдаваемых отладочных результатах всецело зависит от способности программиста мысленно представить себе во всех деталях алгоритм рассматриваемой программы, что невозможно без глубокого и крепкого знания его в течение всего длительного времени проведения отладки.

VIII.2.3. Метод контрольных тестов

> Никогда не берите на корабль два хронометра, берите один или, если есть возможность, три, но не два.
>
> *—Наставление мореплавателям начала XIX века*

> ...путём тестирования никогда нельзя установить отсутствие ошибок в программе. Дело в том, что невозможно определённо утверждать, что при проверке найдена последняя ошибка в программе; следовательно, никогда нельзя быть уверенным, что может быть найдена и первая ошибка
>
> *—Р. Лингер, Х. Миллс, Б. Уитт*

Как бы ни была тщательно проверена и «прокручена» программа за столом, решающим этапом, устанавливающим её пригодность для работы, является контроль программы по результатам её выполнения на компьютере.

Мы рассмотрим здесь универсальный метод контроля — *метод контрольных тестов* («test» — «испытание», «проверка»).

Тестирование — это процесс исполнения программы на компьютере с целью *обнаружения* ошибок [50]. Поясним это определение.

Тестом будем называть информацию, состоящую из исходных данных, специально подобранных для отлаживаемой программы, и из соответствующих им эталонных результатов (не только окончательных, но и промежуточных), используемых в дальнейшем для контроля правильности работы программы.

Если поставить целью демонстрацию *отсутствия* ошибок, то мы подсознательно будем стремиться к этой цели, выбирая тестовые данные, на которых вероятность появления ошибки мала. В то же время, если нашей задачей станет обнаружение ошибок, то создаваемый нами тест будет обладать большей вероятностью обнаружения ошибки. Такой подход заметнее повысит качество программы.

Тестирование — процесс деструктивный (т.е. обратный созидательному, конструктивному). Именно этим и объясняется, почему многие считают его трудным. Большинство людей склонно к конструктивному процессу созидания объектов и в меньшей степени — к деструктивному процессу разделения на части. Для усиления определения тестирования проанализируем два понятия — «удачный» и «неудачный». Большинство назовёт тестовый прогон неудачным, если обнаружена ошибка и, наоборот, удачным, если он прошёл без ошибок. Из определения тестирования следует противоположное: **тестовый «прогон» будем называть *удачным*, если в процессе его выполнения обнаружена ошибка, и *неудачным*, если получен корректный результат**.

Вопрос о позиции программиста по отношению к продукту его труда связан, как это показано Вейнбергом [68], с принципами безличного программирования и когнитивного диссонанса.

Когнитивный диссонанс — это психологический принцип, который руководит действиями человека, чьи представления о себе оказались под угрозой. «Программист, который искренне считает программу продолжением своего «я», не будет пытаться найти все ошибки в ней. Напротив, он постарается показать, что программа правильна, даже если это означает не замечать ошибок, чудовищных для постороннего взгляда... Человеческий глаз имеет почти безграничную способность не видеть то, чего он видеть не желает»[68].

Спасти в такой ситуации может безличное программирование. Вместо того, чтобы быть скрытным и защищать свою программу, программист занимает противоположную позицию: он открыто приглашает других программистов читать и конструктивно критиковать её. Когда кто–то находит ошибку в его программе, программист, конечно, не должен радоваться, что ошибся; его позиция примерно такова: «О! Мы нашли ошибку в *нашей* программе! Хорошо, что мы нашли её сейчас, а не позже! Поучимся на этой ошибке, а заодно посмотрим, не найдём ли ещё!» Программист, обнаруживший ошибку в чужой программе, не кричит: «Посмотри на свою идиотскую ошибку!», а реагирует примерно так: «*Как* любопытно! Интересно, не сделал ли и я такой ошибки в написанном мною модуле?»

При использовании метода тестов **программа (или отдельный её блок) считается *правильной*, если запуск программы для выбранной системы с тестовыми исходными данными даёт правильные результаты**.

Таким образом, контроль программы сводится к тому, чтобы подобрать систему тестов, получение правильных результатов для которой гарантировало бы правильную работу программы и для остальных исходных данных из области, указанной в решаемой задаче.

Для реализации метода контрольных тестов должны быть изготовлены или заранее известны *эталонные* результаты, на основании сверки с которыми получаемых тестовых результатов, можно было бы сделать вывод о правильности работы программы на данном тесте.

Эталонные результаты для вычислительных задач можно получить, осуществляя вычисления вручную, применяя результаты, полученные ранее на другом компьютере или по другой программе, или, используя известные факты, свойства, физические законы.

Разрабатывая систему тестов, нужно стремиться к тому, чтобы успешный пропуск её на компьютере *доказывал* наличие ошибок в программе (или отдельном её блоке), хотя для многих достаточно сложных программ, особенно если над ними работает несколько программистов, можно практически говорить лишь о большей или меньшей вероятности правильности программы.
Это объясняется тем, что изготовление и пропуск *всех* тестов, необходимых для доказательства, может потребовать такого объёма работ, который затянет этап контроля на многие месяцы или годы. Поэтому при разработке системы тестов наряду с задачей всестороннего и глубокого тестирования, стоит задача минимизации количества необходимых тестовых результатов, машинного времени и усилий программиста.

В большинстве случаев при использовании метода контрольных тестов вопрос о *доказательстве отсутствия* ошибок практически можно ставить лишь для небольших блоков (модулей) программы, а для целой программы приходится ограничиваться той или иной вероятностью отсутствия ошибок в программе.
Неоднократно экспериментально установлено, что в любой сложной программе в процессе эксплуатации обнаруживаются ошибки, даже если проведено самое тщательное тестирование. Тем самым утверждается объективная реальность, заключающаяся в невозможности формализовать и обеспечить абсолютную полноту всех эталонных значений, а также провести всеобъемлющее исчерпывающее тестирование и устранить *все* ошибки в сложных программах.

Опыт показывает, что до начала тестирования число ошибок в сложных программах — порядка 1–2% от общего числа операторов в программе. Самое тщательное тестирование сложных программ позволяет получить программы с вероятностью ошибки в каждом операторе 0.0001–0.00001, т.е. несколько ошибок может остаться.

После завершения тестирования программы в течение нескольких лет эксплуатации могут быть выявлены ещё десятки ошибок!

VIII.2.3.1. Стратегия тестирования

> Вы должны радоваться, что мост разрушился, — я планировал построить ещё тринадцать по тому же проекту.
>
> *—Замечание, приписываемое Х.Брюнелю, адресованное дирекции Большой западной железной дороги*

Перечислим основные принципы тестирования [50]. Эти принципы интересны тем, что в основном они интуитивно ясны, но в то же время на них часто не обращают должного внимания.

1. Описание предполагаемых значений выходных данных или результатов должно быть необходимой частью тестового набора.

   Нарушение этого очевидного принципа представляет одну из наиболее распространённых ошибок. Ошибочные, но правдоподобные результаты могут быть признаны правильными, если результаты теста не были заранее определены.

Здесь мы сталкиваемся с явлением психологии: мы видим то, что мы хотим увидеть. Другими словами, несмотря на то, что тестирование по определению — деструктивный процесс, есть подсознательное желание видеть корректный результат. Один из способов борьбы с этим состоит в поощрении детального анализа выходных переменных заранее, ещё при разработке теста.

2. Следует избегать тестирования программы её *автором*.

   Многие, кому приходилось самому делать дома ремонт, знают, что процесс обрывания старых обоев (деструктивный процесс) не лёгок, но он просто невыносим, если не кто–то другой, а Вы сами первоначально их наклеивали. Вот так же и большинство программистов не может эффективно тестировать свои программы, потому что им трудно демонстрировать собственные ошибки.

3. Необходимо досконально изучать результаты применения каждого теста.

   Представляется достоверным, что значительная часть всех обнаруженных в конечном итоге ошибок могла быть выявлена в результате самых первых тестовых прогонов, но они были пропущены вследствие недостаточно тщательного анализа результатов первого тестового прогона.

4. Тесты для неправильных и непредусмотренных входных данных следует разрабатывать так же тщательно, как для правильных и предусмотренных.

   Вполне вероятно, что тесты, представляющие неверные и неправильные входные данные, обладают большей обнаруживающей способностью, чем тесты, соответствующие корректным входным данным.

5. Необходимо проверять не только, делает ли программа то, для чего она предназначена, но и не делает ли она то, что не должна делать.

   Обязательно проверяйте программу на нежелательные побочные эффекты.

6. Не следует выбрасывать тесты, даже если программа уже не нужна. Необходимость в использованных тестах наиболее часто возникает в интерактивных системах отладки.

   Обычно тестирующий сидит за терминалом, на лету придумывает тесты и запускает программу на выполнение.

   При такой практике работы после применения тесты пропадают. После внесения изменений или исправления ошибок необходимо повторить тестирование, тогда приходится заново изобретать тесты. Как правило, этого стараются избегать, поскольку повторное создание тестов требует значительной работы.

7. Нельзя планировать тестирование в предположении, что ошибки не будут обнаружены.

8. **Вероятность наличия необнаруженных ошибок в части программы пропорциональна числу ошибок, уже обнаруженных в этой части.**

   На первый взгляд, этот принцип лишён смысла, но тем не менее подтверждается многими программами. Например, допустим, что некоторая программа состоит из модулей A и B. К определённому сроку в модуле A обнаружено пять ошибок, а в модуле B — только одна, причём модуль A не подвергался более тщательному тестированию. Тогда из рассматриваемого принципа следует, что вероятность необнаруженных ошибок в модуле A больше, чем в модуле B. Справедливость этого принципа подтверждается ещё и тем, что для ошибок свойственно располагаться в программе в виде неких скоплений, хотя данное явление пока никем ещё не объяснено.

   Таким образом, если в какой–нибудь части программы обнаружено больше ошибок, чем в других, то на её тестирование должны быть направлены дополнительные усилия.

9. Тестирование — процесс творческий. Вполне вероятно, что для тестирования большой программы требуется больший творческий потенциал, чем для её проектирования.

   Чтобы подчеркнуть некоторые мысли, высказанные в этом разделе, приведём ещё раз три наиболее важных принципа тестирования.

Тестирование — это процесс выполнения программ на компьютере с целью *обнаружения* ошибок.

Хорошим считается тест, который имеет высокую вероятность обнаружения ещё не выявленной ошибки.

Удачным является тест, который обнаруживает ещё не выявленную ошибку.

«На закуску» рекомендуем выполнить следующий простой тест. Задача состоит в том, чтобы проверить программу, которая по трём заданным числам печатает сообщение о том, является ли треугольник со сторонами, длины которых равны данным значениям, неравносторонним, равнобедренным или равносторонним [51].

Напишите на листе бумаги набор тестов, которые, как вам кажется, будут адекватно проверять эту программу. Построив свои тесты, проанализируйте их.

Приступайте к работе...

Следующий шаг состоит в оценке эффективности Вашей проверки. Оказывается, что программу труднее написать, чем это могло показаться вначале. Были изучены различные версии данной программы, и составлен список общих ошибок. Оцените Ваш набор тестов, попытавшись с его помощью ответить на приведённые ниже вопросы. За каждый ответ «да» присуждается одно очко.

1. Составили ли Вы тест, который представляет правильный неравносторонний треугольник? (Заметим, что ответ «да» на тесты со значениями 1, 2, 3, и 2, 5, 10 не обоснован, т.к. не существует треугольников, имеющих такие стороны.)
2. Составили ли Вы тест, который представляет правильный равносторонний треугольник?
3. Составили ли Вы тест, который представляет правильный равнобедренный треугольник? (Тесты со значениями 2, 2, 4 принимать в расчёт не следует.)
4. Составили ли Вы, по крайней мере, три теста, которые представляют правильные равнобедренные треугольники, полученные как перестановки двух равных сторон треугольника (например, 3, 3, 4; 3, 4, 3, и 4, 3, 3)?
5. Составили ли Вы тест, в котором длина одной из сторон треугольника принимает нулевое значение?
6. Составили ли Вы тест, в котором длина одной из сторон треугольника принимает отрицательное значение?
7. Составили ли Вы тест, включающий три положительных целых числа, сумма двух из которых равна третьему? (Другими словами, если программа выдала сообщение о том, что числа 1, 2, 3 представляют собой стороны неравностороннего треугольника, то такая программа содержит ошибку.)
8. Составили ли Вы, по крайней мере, три теста с заданными значениями всех трёх перестановок, в которых длина одной стороны равна сумме длин двух других сторон (например, 1, 2, 3; 1, 3, 2 и 3, 1, 2)?
9. Составили ли Вы тест из трёх целых положительных чисел, таких, что сумма двух из них меньше третьего числа (т.е. 1, 2, 4 или 12, 15, 30)?
10. Составили ли Вы, по крайней мере, три теста из категории 9, в которых Вами испытаны все три перестановки (например: 1, 2, 4; 1, 4, 2 и 4, 1, 2)?
11. Составили ли Вы тест, в котором все стороны треугольника имеют длину, равную нулю (т.е. 0, 0, 0)?
12. Составили ли Вы по крайней мере, один тест, содержащий нецелые значения?
13. Составили ли Вы хотя бы один тест, содержащий неправильное число значений (например, два, а не три целых числа)?

Конечно, нет гарантий, что с помощью набора тестов, который удовлетворяет вышеперечисленным условиям, будут найдены все возможные ошибки. Но поскольку вопросы 1÷13 представляют ошибки, имевшие место в различных версиях данной программы, адекватный тест для неё должен их обнаруживать.

Отметим, что опытные профессиональные программисты набирают в среднем только 7÷8 очков из 14 возможных.

Выполненное упражнение показывает, что тестирование даже тривиальных программ, подобных приведённой, — не простая задача.

VIII.2.3.2. Тактика тестирования

> Перевести программу из хорошего состояния в отличное неизмеримо труднее, чем из плохого в удовлетворительное.
>
> *—Программистский фольклор*

Поговорим о методах тестирования.

При *неупорядоченном* тестировании («smoke test» — «грубая проверка работоспособности простым запуском», «дымовой тест») исходные данные, имитирующие внешнюю среду, случайным образом генерируются во всем диапазоне возможного изменения параметров. При этом многие значения исходных данных характеризуются малой вероятностью обнаружения ошибок и не оправдывают затраты на выполнение тестирования. Кроме того, возможно появление логически противоречивых данных. В то же время данные, наиболее важные с позиции реального использования программ и возможностей обнаружения ошибок, могут оказаться не охваченными в процессе тестирования.

Поэтому на практике последовательно применяют следующие методы тестирования: *статический*, *детерминированный* и *стохастический*.

Статическое тестирование («static check») является наиболее формализованным методом проверки корректности программ. Тестирование проводится без исполнения программы путём формального анализа текста программы на языке программирования. Операторы и операнды текста программ при этом анализируются в символьном виде, поэтому такой метод называют также *символическим* тестированием.

Наиболее трудоёмкими и детализирующими являются методы *детерминированного* тестирования. При детерминированном тестировании контролируется каждая комбинация исходных эталонных данных и соответствующая ей комбинация эталонных результатов. Разумеется, в сложных программах невозможно перебрать все комбинации исходных данных и проконтролировать результаты функционирования программы на каждой из них. В таких случаях применяется *стохастическое* тестирование, при котором исходные тестовые данные задаются множеством случайных величин с соответствующими распределениями и для сравнения полученных результатов используются также распределения случайных величин. В результате при стохастическом тестировании возможно более широкое варьирование исходных данных, хотя отдельные ошибки могут быть не обнаружены, если они мало искажают средние статистические значения или распределения.

Стохастическое тестирование применяется в основном для обнаружения ошибок, а для диагностики и локализации ошибок приходится переходить к детерминированному тестированию с использованием конкретных значений параметров из области изменения использовавшихся случайных величин.

Рассмотрим некоторые *правила* тестирования, в которых делается попытка учесть как желательность доказательства правильности контролируемой программы, так и ограниченность человеческих возможностей при проведении такого доказательства [49].

Проход участков. Каждый линейный участок программы должен быть обязательно пройден при выполнении по крайней мере, одного теста. Очевидно, что в противном случае никакой гарантии в правильности работы всей программы дать будет нельзя.

В том случае, когда выполнение некоторого участка программы меняет порядок выполнения или характер работы других участков, может потребоваться перебор всех ветвей программы, т.е. проход по всем возможным путям выполнения программы (многократная проверка требуется, в частности, для участков, содержащих переменные с индексами).

Точность проверки. Контроль арифметических блоков (как и других блоков) производится путём сверки результатов, полученных при выполнении блока, с эталонными результатами. Для арифметических результатов дополнительная сложность заключается в определении точности, с которой необходимо сверять (и, тем самым, вычислять) эталонные и тестовые результаты, с тем, чтобы можно было действительно удостовериться в правильности работы блока.

Дело в том, что величины, входящие в проверяемое арифметическое выражение, в зависимости от соотношения их значений и характера производимых над ними операций, вносят различный вклад в результат. Поэтому может оказаться, что неправильно запрограммированное выражение для некоторых тестовых значений величин, входящих в него, будет иметь *якобы* правильное значение ввиду того, что результат неправильной операции или неверно вычисленный ранее операнд выражения не окажут почти никакого влияния на тестовое (сравниваемое) значение выражения.

Например, для оператора C=A+B из того, что значение C совпало с эталонным значением, не следует, что выражение записано в программе верно, поскольку для случая, когда A>>B, замена плюса на минус не будет обнаружена, если эталонное значение C вычислено с недостаточной точностью. Кроме того, если вычисление A и B не было проверено ранее, то из правильности C нельзя сделать вывод о правильности вычисления B (для случая A>>B).

Таким образом, для того, чтобы быть уверенным в том, что правильный числовой результат, полученный на

компьютере, говорит о правильности программы, необходимо следить за промежуточными результатами вычислений, которые не должны выходить за определённый диапазон, устанавливаемый в зависимости от точности вычислений эталонных результатов. Выполнение такого требования может привести к необходимости многократной проверки выражения для различных диапазонов данных.

Минимальность вычислений.

Когда продолжительность работы контролируемой программы и, тем самым, количество вычислений и необходимых для контроля тестовых данных зависит от каких–либо параметров, то при контроле их следует выбирать такими, чтобы они минимизировали количество вычислений.

К таким параметрам, например, могут относиться :

- шаг или отрезок интегрирования;
- порядок матрицы или количество элементов вектора;
- длина символьных строк;
- точность для итерационных вычислений и т.п.

Конечно, такая инициализация не должна значительно снижать надёжность контроля. Следует заметить также, что значения исходных данных нужно выбирать такими, чтобы изготовление эталонных результатов вручную было, по возможности, облегчено. Например, данные могут быть сначала взяты целочисленными или такими, чтобы при проверке выражений некоторые их слагаемые, уже проверенные ранее, обращались в нуль.

Достоверность эталонов. Нужно обратить внимание и на достоверность процесса получения эталонных результатов. По возможности они должны вычисляться не самим программистом, а кем–то другим, с тем, чтобы одни и те же ошибки в понимании задания не проникли и в программу, и в эталонные результаты. Если тесты готовит сам программист, то эталоны нужно вычислять до получения на компьютере соответствующих результатов. В противном случае имеется опасность невольной подгонки вычисляемых значений под желаемые, полученные ранее на компьютере. В качестве эталонных результатов часто используют и данные,полученные при ручной прокрутке программы.

Планирование.

При отсутствии планового подхода тестирование обычно сводится к тому, что программист берет какие–то, можно сказать, первые попавшиеся исходные данные и пропускает программу многократно, исправляя её при обнаружении ошибок и добиваясь того, чтобы получаемые результаты походили на желаемые.

Ясно, что при этом контролируется только некоторая часть блоков и операторов, а остальные выполнятся в первый раз уже во время счета, и будут ли при этом найдены ошибки, имеющиеся в них, зависит только от случая.

При *плановом* подходе программа проверяется последовательно блок за блоком, причём если программа состоит из центрального блока, который проводит обращения к периферийным блокам, мало связанным друг с другом, то возможны следующие два основных подхода к контролю такой программы, два основных направления тестирования: от периферии к центру (*восходящее* тестирование) или, наоборот, от центра к периферии (*нисходящее* тестирование).

При первом, *восходящем* способе, применяемом обычно для небольших программ, сначала тестируют отдельные периферийные блоки, а затем переходят к тестированию центральной части, которая, разумеется, взаимодействует только с отлаженными уже блоками.

При *нисходящем* тестировании, используемом для достаточно больших программ, параллельно с контролем периферийных блоков (или даже до начала их контроля) производится и контроль центрального блока, выполняемого на компьютере совместно с имитаторами периферийных блоков, называемых *«заглушками»*. В задачу имитаторов входит моделирование работы соответствующих блоков с целью поддержать функционирование центрального блока. Обычно «заглушки» выдают простейший результат, например константу и сообщение о факте своего участия в работе. Вместо постоянной величины на наиболее поздней стадии тестирования может выдаваться и случайная величина в требуемом диапазоне.

Например, для начального контроля программы, включающей в качестве одного из своих блоков вычисление определённого интеграла, «заглушка» такого блока может возвращать константу. В свою очередь, блок интегрирования сам имеет периферийный блок вычисления значений подинтегральной функции, в качестве «заглушки» которого поначалу также может быть взята константа или простейшая функциональная зависимость.

К сожалению, часто неверно понимают функции, выполняемые «заглушками». Так, порой можно услышать, что «заглушка» должна выполнять лишь запись сообщения, устанавливающего: «Модуль подключился». В большинстве случаев эти утверждения ошибочны. Когда модуль А вызывает модуль В, А предполагает, что В выполняет некую работу, т.е. модуль А получает результаты работы модуля В. Когда же модуль В просто возвращает управление или выдаёт некоторое сообщение без передачи в А определённых осмысленных результатов, модуль А работает неверно не вследствие ошибок в самом модуле, а из–за несоответствия ему модуля–«заглушки». Более того, результат может оказаться неудовлетворительным, если «ответ» модуля–«заглушки» не меняется в зависимости от условий теста. Если «заглушка» всегда возвращает один и тот же фиксированный результат вместо конкретного значения, предполагаемого вызывающим модулем именно в этом вызове, то вызывающий модуль сработает как ошибочный (например, зациклится) или выдаст неверное выходное значение.

Следовательно, **создание модулей–«заглушек» — задача нетривиальная**.

Практически, оба этих способа редко используются в чистом виде, отдельно один от другого. Обычно ко времени, когда приступают к контролю центрального блока, какие–то простейшие периферийные блоки уже отлажены автономно, и нет необходимости моделировать их работу и разрабатывать «заглушки».

Преимуществом ранней отладки центрального блока при нисходящем тестировании является то, что программист быстро получает возможность проверить периферийные блоки в условиях, которые в необходимой степени приближены к реальным. Действительно, центральный блок, снабжённый хотя бы и простейшими функциональными возможностями, можно рассматривать как реальную среду, в которую «погружаются» отлаживаемые блоки, добавляемые к центральной части. Добавление отлаживаемых блоков удобно производить по одному для быстрейшего поиска ошибок, возникающих при стыковке с центральным блоком.
Подключение каждого нового блока к центральной части позволяет постепенно усложнять испытания, которым подвергается тестируемая программа.

Строгой, корректной процедуры подключения очередного последовательно тестируемого модуля не существует. Единственное правило, которым следует руководствоваться при выборе очередного модуля, состоит в том, что им должен быть один из модулей, вызываемых модулем, предварительно прошедшим тестирование.

Запомните, что даже если изменения вносятся только в одну подпрограмму, то повторному тестированию подлежит вся система. Этот процесс называется *тестированием с возвратом*. Проверять работу только изменённой подпрограммы недостаточно! Недостаточно полное тестирование такого рода повышает вероятность неудач.

Проведём сравнение нисходящего и восходящего тестирования [50].

Преимущества	Недостатки
Нисходящее тестирование	
1. Имеет преимущества, если ошибки, главным образом, в верхней части программы 2. Раннее формирование структуры программы позволяет провести её демонстрацию пользователю и служит моральным стимулом	1. Необходимо разрабатывать модули–«заглушки», которые часто оказываются сложнее, чем кажется вначале 2. Может оказаться трудным или невозможным создать тестовые условия 3. Сложнее оценка результатов тестирования 4. Стимулируется незавершение тестирования некоторых модулей
Восходящее тестирование	
1. Имеет преимущества, если ошибки, главным образом, в модуле нижнего уровня 2. Легче создавать тестовые примеры 3. Проще оценка результатов	1. Программа как единое целое не существует до тех пор, пока не добавлен последний модуль

Значительное повышение корректности и надёжности программ достигается применением *двойного* или N — *кратного* программирования («duplication check» — «двойной просчёт», «двойная проверка»).

При этом методе при разных алгоритмах и на разных языках программирования создаётся несколько вариантов программы. Эти варианты реализуют одни и те же функции и при определённых тестовых данных должны выдавать тождественные результаты. Различие результатов при тестировании указывает на наличие ошибок по крайней мере в одном из вариантов. Обычно при разработке вариантов программы используется один и тот же алгоритм, но программы создаются на разных языках, разных компьютерах и разными программистами. На практике применяется программирование с $N \leq 2$. Практически очень редки случаи, когда реальная программа создавалась в трёх и более вариантах.

В заключение заметим, что если исполнение теста приносит результаты, не соответствующие предполагаемым, то это означает, что

- либо модуль имеет *ошибку*;
- либо *неверны* предполагаемые результаты (ошибки в тесте).

Для устранения такого рода недоразумений нужно тщательно проверять набор тестов («тестировать» тесты).

VIII.2.3.3. Типы тестов

> О несчастном случае мой подзащитный впервые узнал лишь тогда, когда этот случай произошёл.
>
> *—Из речи адвоката*

Основным тестом мы будем называть тест, проверяющий основные функциональные возможности программы. Однако существует опасность, что после успешного окончания основного тестирования «на радостях» обычно забывают о необходимости дальнейшего, более тщательного контроля программы и отдельных её участков, да и настроиться на такой контроль становится уже психологически трудно. Поэтому помимо основного теста необходимо применить следующие типы тестов.

Вырожденный тест. Этот тест затрагивает работу отлаживаемой программы в самой минимальной степени. Обычно тест служит для проверки правильности выполнения самых внешних функций программы, например обращения к ней и выхода из неё.

Тест граничных значений, или *«стрессовый* тест»(«high–low bias checking», «twin check»). Тест проверяет работу программы для граничных значений параметров, определяющих вычислительный процесс. Часто для граничных значений параметра работа программы носит особый характер, который тем самым требует и особого контроля.

Если в качестве примера рассмотреть тестирование подпрограммы сортировки, то нужно исследовать следующие ситуации:

- сортируемый массив пуст;
- сортируемый массив содержит только один элемент;
- все элементы в сортируемом массиве одинаковы;
- массив уже отсортирован.

Л.Питер приводит следующий поучительный пример. Компьютер одной компании по страхованию автомобилей выслал проживающему в Сент–Луисе клиенту счёт на сумму 0.00 долларов. Когда же компьютер направил ему «последнее уведомление» с угрозой расторгнуть договор, этот человек обратился за помощью к своему финансовому агенту. Тот пришёл к выводу, что лучший способ уладить дело — отправить компьютеру чек на 0.00 долларов. Это было сделано, и в ответ пришло подтверждение с благодарностью и заверением, что договор остаётся в силе!

Аварийный тест. Тест проверяет реакцию программы на возникновение разного рода аварийных ситуаций в программе, в частности вызванных неправильными исходными данными, то есть проверяется диагностика, выдаваемая программой, а также окончание её работы или, может быть, попытка исправления неверных исходных данных (разработчики реальных программ знают, что пользователи подобны шаловливому ребёнку, играющему в отсутствие старших с телевизором или магнитофоном).

Поэтому в реальных программах, спроектированных с достаточной надёжностью, совокупности приказов, которые должны работать только в особых аварийных ситуациях, занимают порой более 90% общего объёма программы. Эти совокупности приказов называют иногда блоками *«защиты от дурака»* («fool proof»). Такие системы, обладая достаточной надёжностью, устойчиво функционируют даже при самых неподходящих действиях работающих с ними людей.

В журнале «Компьютерный мир» рассказывалось об одной довольно дорого обошедшейся ошибке. При вводе данных палец оператора случайно задел не ту клавишу, и автомобиль «Форд», принадлежащий одному из граждан, стал стоить не 950, а 7000950 долларов! А если обладаешь таким дорогим имуществом, нужно платить большой налог. Налог составил 290000 долларов. Когда ошибка обнаружилась, эта сумма была уже включена в бюджет города. Владелец автомобиля получил счёт на 290 тыс. долларов, но платить не стал. Причиной ошибки следует считать,

конечно, не неверно нажатую клавишу, а плохую программу, автор которой не позаботился о достаточно мощных процедурах контроля входных данных.

Процветающие фирмы, занятые разработкой программного обеспечения, специально нанимают профессионально неподготовленных людей, чтобы они поработали с вновь созданными программами. В их задачу входит за короткое время сделать столько неправильных обращений к программе, сколько пользователь не сделает и за долгий период.

Например, когда программа запрашивает цену товара, оператор набирает на клавиатуре слово «Почему?» вместо числа и т.д.

Одним из свойств хорошей программы является, как говорят специалисты, её *дружественность*. Это означает, что в случае ошибки пользователя программа выдаст на экран сообщение, направленное на оказание помощи в выполнении поставленной задачи. Это может быть подсказка, наводящий вопрос, разъяснение противоречивости или иной ошибки в требованиях пользователя.

В лучших образцах таких программ вместо сообщения «треугольника с такими сторонами не бывает» на экране выдаются тексты вида: «Вероятно, Вы ошиблись. На плоскости невозможно построить треугольник со сторонами, имеющими длины 1, 1, 100. Попытайтесь изменить значения длин сторон.»

Существуют программы, которые не только обнаруживают, но и исправляют ошибки. Например, при проектировании какого–то прибора инженер за дисплеем подбирает параметры его деталей и вводит приказ запомнить величину сопротивления 150 кОм. Тогда компьютер может ответить: «Вероятно, Вы ошиблись. К сожалению, известны только данные о выпускаемых сопротивлениях с номиналами 160 и 180 кОм. Попытайтесь изменить значение номинала сопротивления. Если вам подходит значение 180 кОм, нажмите клавишу Ввод». Программа лишь предложила один из возможных вариантов взамен явно неосуществимого. Окончательное решение осталось за пользователем.

Однако дружественность программ должна иметь чёткие границы, иначе автоматическое исправление ошибок превратится в медвежью услугу пользователю. Одно из свойств хороших программ состоит в том, что пользователь не должен при работе с ними удивляться, они не должны делать ничего неожиданного, т.к. эти неожиданности редко бывают приятными и полезными.

В связи с этим интересны рекомендации по проектированию программ ведения диалога [60], где автор вообще выступает против какого–либо очеловечивания вычислительных систем:

- «Создание ЭВМ, ведущих себя как люди, напоминает попытки строить самолёты, машущие крыльями. ...Создавать вычислительные системы, которые будут вести себя как инструменты... это означает избегать диалоговых систем, начинающихся с реплик «Привет, я Бетси 307, назови своё имя».
- Не давайте человеческих имён или признаков программам и системам.
- Не приписывайте вычислительным системам свободы воли или поведения, напоминающего живое существо.
- Если что–то происходит неправильно, не обвиняйте в этом ЭВМ или программу. Помните, что инструменты не делают ошибок, они или отказывают или ломаются».

VIII.3. Методы локализации ошибок

> Если Вы думаете, что разработка и кодирование программы — вещь трудная, то Вы ещё ничего не видели.
>
> *—Популярный афоризм*

После того как установлено, что в программе или в конкретном её блоке имеется ошибка, возникает задача её *локализации*, то есть установления точного места в программе, где она находится. Можно считать, что *процесс локализации ошибок* состоит из трёх основных компонентов [49]:

1. получение вручную или с помощью компьютера тестовых результатов;
2. анализ тестовых результатов и сверка их с эталонными;
3. выявление ошибки или формулировка предположения о характере и месте ошибки в программе.

Если ошибка найдена, то производится её исправление; в противном случае осуществляется переход к пункту 1, т.е. к получению дополнительных тестовых результатов.

В ходе поиска ошибок программист, анализируя полученные на компьютере результаты, проверяет различные предположения о характере и месте ошибки в программе, которые при этом приходят ему в голову. В случае несоответствия этих гипотез выданным результатам, программист выдвигает новые гипотезы и проверяет их или в уме, или проводя вычисления за столом, или обращаясь за новыми результатами к компьютеру.

В таком характере работы программиста можно найти нечто общее с расчётом вариантов, который осуществляет шахматист (или шашист) во время игры, когда он путём расчётов в уме ищет выигрывающий ход в позиции на шахматной доске, подвергая проверке один из заслуживающих внимания ходов за другим. Не найдя выигрывающего хода, шахматист делает какой–то, по его мнению, хороший ход, приближающий его к цели.Так и программист, не найдя ошибки путём исследования полученных тестовых результатов, делает новое предположение о месте или о характере ошибки, вставляет новую отладочную печать или изменяет программу («ход программиста»), а ЭВМ выдаёт новые тестовые результаты («ход ЭВМ»). Компьютер выступает как своеобразный партнёр, задача которого заключается в том, чтобы вскрыть ошибки в рассуждениях программиста, как бы сформулированных им в тексте отлаживаемой программы. Продолжая аналогию, можно сказать, что подобно тому, как нельзя реально надеяться выиграть партию в два–три хода, так же нельзя найти все ошибки в реальной программе за одно–два обращения к компьютеру.

Программистов, успешно проводящих поиск ошибок в программе, можно условно разделить на «аналитиков» и «экспериментаторов» [49].

Аналитики отлаживают программу, редко используя компьютер и применяя простейшие способы получения тестовых результатов на компьютере путём тщательного изучения этих результатов и на основании глубокого и чёткого представления о структуре и особенностях алгоритма отлаживаемой программы.

Экспериментаторы ищут ошибки, изощрённо используя всевозможные *отладочные средства*, быстро получая необходимые для все большей и большей локализации ошибок промежуточные результаты и легко ориентируясь в них.

Конечно, идеальным является случай, когда программист сочетает в себе способность к глубокому расчёту в уме различных вариантов работы программы и навыки работы с разнообразными отладочными средствами.

Если успех аналитического подхода к поиску ошибок зависит, видимо, от способностей и опыта программиста, то изучение и использование средств, помогающих локализации ошибок — главным образом средств получения необходимых промежуточных результатов, — доступно каждому программисту.

Имеются такие средства и в языке программирования MSX BASIC!

Под *промежуточными* результатами выполняемой программы договоримся понимать как арифметические результаты, характеризующие значения используемых величин, так и *логические* «результаты»,т.е. информацию, содержащую сведения о факте или последовательности выполнения операторов программы.

Учтите следующие принципы Г.Майерса [51].

1. *Думайте* ! Наиболее эффективный метод отладки заключается в глубоком анализе информации об ошибках. Для её эффективного проведения **специалист должен обладать способностью точно определять большинство ошибок без использования компьютера.**

2. Используйте средства отладки только как вспомогательные. Не применяйте эти средства вместо того, чтобы обдумывать задачу.

   Ясно, что такие средства как трассировка (раздел VIII.3.1.) и аварийная печать (раздел VIII.3.2.) отражают случайный подход к отладке.

   Эксперименты показали, что программисты, избегающие применения средств отладки, даже при отлаживании незнакомых им программ выполняют её лучше, чем те, кто пользуется этими средствами.

3. **Избегайте экспериментирования!**

   Пользуйтесь им только как последним средством. Наиболее общей ошибкой, которую допускают начинающие программисты, занимающиеся отладкой, является попытка решить задачу посредством внесения в программу экспериментальных изменений («Я не знаю, что неправильно, но я изменю этот оператор `IF` и посмотрю, что получится»). Этот абсолютно неверный подход не может даже рассматриваться как отладка; он основан на случайности. Экспериментирование не только уменьшает вероятность успеха, но часто и усложняет задачу, поскольку при этом в программу вносятся новые ошибки.

4. Если Вы зашли в тупик, отложите рассмотрение программы. Наше подсознание является мощным механизмом решения проблем. То, что мы часто приписываем вдохновению, оказывается всего лишь выполненной подсознанием работой по решению задачи, тогда как наша сознательная деятельность в это время связана с чем–нибудь другим, например с едой, прогулкой или просмотром кинофильма. Если Вы не можете локализовать ошибку в приемлемые сроки (предположительно за 30 минут для небольших программ и за несколько часов для больших), прекратите поиски и займитесь каким–нибудь другим делом, так как эффективность Вашей работы значительно снизится. Проблему следует «забыть» до тех пор, пока Вы либо подсознательно не найдете ее решения, либо отдохнете и будете готовы вновь рассмотреть симптомы ошибки.

И наконец, **если Вы окончательно зашли в тупик, то изложите задачу кому-нибудь ещё**.

Сделав это, Вы, вероятно, обнаружите что–то новое. Часто случается так, что просто пересказав задачу хорошему слушателю, Вы вдруг найдёте решение без какой–либо помощи с его стороны.

Далее мы рассмотрим некоторые, ставшие уже классическими, способы получения программистом промежуточных результатов, вырабатываемых отлаживаемой программой.

VIII.3.1. Трассировка

> Всякий необходимо причиняет пользу, употреблённый на своём месте. Напротив того: упражнения лучшего танцмейстера в химии неуместны; советы опытного астронома в танцах глупы.
>
> *—Козьма Прутков*

Трассировка («trace» — «след») представляет собой пошаговое выполнение программы в автоматическом режиме. Если Вы создали сложную по логике программу, и она выполняется неправильно, а по информации, выводимой на экран нельзя установить, где ошибка, то оператор

```
TRON
```

,

где TRON («TRacing ON» — «установить трассировку») — служебное слово, поможет вам произвести трассировку программы во время её выполнения.

Выполнение оператора TRON приводит к последовательному выводу номеров *всех* выполняемых в данный момент строк программы в виде

```
[n1][n2]…[nk]
```

Цепочка этих номеров составляет *след* (или *трассу*) работы программы. Между компонентами [...] располагаются значения, выводимые по оператору PRINT или вводимые с клавиатуры по оператору INPUT.

Так как номер *каждой* выполненной строки будет выведен на экран, то Вы легко увидите, по каким «ветвям» выполняется Ваша программа, поэтому анализ полученной трассы, как правило, позволяет локализовать ошибку.

Отменяется режим трассировки оператором

```
TROFF
```

,

где TROFF («TRacing OFF» — «отменить трассировку») — служебное слово.

Итак, если вам непонятна работа какого–то участка программы, то в начале участка надо поставить оператор TRON, а в конце — оператор TROFF.

Например:

```
249 TRON
250 'Начало проверяемого участка
     …
300 'Конец проверяемого участка
310 TROFF
```

Пример 1.
0831-01.bas
0831-01.bas

```
10 TRON:GOSUB 100:END
100 IF K<1 THEN K=K+1:PRINT K:GOSUB 100
110 RETURN
run                    run
[100] 1                [10][100] 1
[100][110][110]        [100][110][110]
Ok   ———▲———           Ok   ———▲———
       │                       │
     Подумайте, почему ?!
```

Пример 2.
0831-02.bas
0831-02.bas

```
10 TRON:INPUT A,B:PRINT A;B
20 IF A*B<99 THEN 100 ELSE TROFF
100 PRINT A+B;A-B
Ok
run                     run
? 5,6                   [10]? 10,10
 5  6                    10  10
[20][100] 11 -1         [20] 20  0
Ok                      Ok
```

Отметим, что команда RUN в её простейшем виде не отменяет режима TRON, так что использование оператора TRON ведёт к выполнению трассировки всех последующих программ до тех пор, пока не встретится оператор TROFF.

Единственное, что может отменить трассировку, — это выполнение оператора TROFF или нажатие кнопки RESET.

Операторы TRON и TROFF можно использовать и в режиме прямого выполнения команд.

Пример 3.
0831-03.bas
0831-03.bas

```
Ok
TRON
Ok
20 GOSUB 100:END
100 IF K<1 THEN K=K+1:PRINT K ELSE GOSUB 100
110 RETURN
run
[20][100] 1
[110]
Ok
```

Этими операторами нужно пользоваться избирательно и крайне осторожно — иначе Вы «утонете» в протоколах трассировки. Поэтому стремитесь ограничить область действия операторов TRON и TROFF в тексте Вашей программы!

VIII.3.2. Аварийная печать

Щёлкни кобылу в нос — она махнёт хвостом.

—Козьма Прутков

Под *аварийной печатью* («dump» — «дамп», «разгрузка памяти», «выдача») понимается печать значений переменных в программе в тот момент её выполнения, когда в ней возникает ошибка, препятствующая дальнейшему нормальному её выполнению; обычно после осуществления такой печати выполнение программы прекращается. Благодаря аварийной выдаче программист получает доступ к тем значениям переменных, которые они имели в момент возникновения аварийной ситуации. Изучение и сопоставление таких значений обычно позволяет программисту достаточно точно локализовать ошибку в программе, и иногда и не одну.

Пример.

```
10 PRINT 1/SIN(X)
run
Division by zero in 10
Ok
 ???     ◀— Раздумье...
print X       ◀— Аварийная печать
 0      ◀— Вот теперь все ясно!
Ok
```

—Вам известны мои методы. Примените их!

—А.Конан Дойль. Собака Баскервилей

Эффективным методом локализации ошибки для небольших программ является прослеживание в обратном порядке логики выполнения программы с целью обнаружения точки, в которой нарушена логика [51]. Другими словами, отладка начинается в точке программы, где был обнаружен (например, напечатан) некорректный результат. Для этой точки на основании полученного результата следует установить (логически вывести), какими должны быть значения переменных.

Мысленно выполняя из данной точки программу в обратном порядке и опять рассуждая примерно так: «Если в этой точке состояние программы (т.е. значения переменных) было таким, то в другой точке должно быть следующее состояние», можно достаточно быстро и точно локализовать ошибку (т.е. определить место в программе между точкой, где состояние программы соответствовало ожидаемому, и первой точкой, в которой состояние программы отличалось от ожидаемого).

VIII.3.3. Локализация с точками останова

Разделяй и властвуй!

—Людовик XI

Трассировка хороша только для коротких программ; более универсальным способом является *печать в узлах* («snapshot» — «моментальный снимок») или *локализация с точками останова*.

Точка останова — это точка в программе, на которой Вы можете временно остановить выполнение программы, просмотреть значения интересующих Вас переменных и затем продолжить выполнение.

Это осуществляется путём включения в программу в точке останова оператора, имеющего простейший синтаксис

```
STOP
```
.

При выполнении этого оператора вычисления приостанавливаются и пользователь в режиме непосредственного счета может вывести значения интересующих его переменных. Анализ их позволяет делать выводы о правильности вычислительного процесса, а следовательно, и принимать те или иные решения.

Оператор STOP выводит сообщение

«Break in ****»
*(«Остановка в строке с номером ****»)*,

где **** — номер строки, содержащей оператор STOP.

Если Вы будете внимательны, то услышите и предупредительный звонок, как и при выполнении оператора BEEP.

После останова по оператору STOP вычисления могут быть возобновлены командой CONT которая должна быть выполнена в режиме непосредственного счета.

Однако ни в коем случае не изменяйте, не добавляйте и не исключайте в этот момент строки Вашей программы!

Пример. Программа, осуществляющая вычёркивание R-й буквы из слова P$ и запись полученного после вычёркивания слова в слово Q$.
0833-01.bas
0833-01.bas

```
100 INPUT P$,R:Q$="":FOR K=1 TO LEN(P$)
130 IF K<>R THEN Q$=Q$+MID$(P$,K,1):STOP:NEXT K:PRINT Q$
run
? корова,2    ┌─► cont          ┌─► cont          ┌─► cont          ┌─►…
Break in 130 │   Break in 130  │   Break in 130 │   Break in 130 │
Ok           │   Ok           │   Ok           │   Ok           │
print Q$     │   print Q$     │   print Q$     │   print Q$     │
к      │ кр     │ кро     │ кров    │
Ok     └──────┘   Ok     └─────────┘   Ok     └──────┘   Ok     └──────┘
```

Кстати, нажатие клавиш CTRL+STOP(если, конечно, Вы не запретите подобное прерывание!) выполняет те же действия, что и ввод оператора STOP в Вашу программу, только, разумеется, программную строку, в которой произошло прерывание, вам будет трудно угадать!

Заметим, что во время прерывания Вы можете изменять значения переменных. Однако учтите, что если «что-либо»(например, оператор CLEAR) делает продолжение программы, прерванной по оператору STOP, бессмысленным, то любая попытка выполнить команду CONT приводит к сообщению

«? Can't CONTINUE»
(«Нельзя продолжить»).

Для продолжения выполнения программы в подобных случаях используйте оператор GOTO.

Кстати, аналогичное сообщение возникает и при остроумной попытке продолжения «листания» текста программы (после выполнения команды LIST и нажатия клавиш CTRL+STOP) путём выполнения команды CONT!

Итак, нами изучены средства языка MSX BASIC, применяемые при локализации с точками останова.

Перейдём теперь к рассмотрению *алгоритма* этого процесса.

Предположим, что Вы знаете как выполняется Ваша программа, т.е. какова последовательность выполняемых операторов в любой момент времени.

- 1. Выберите оператор, до которого, как Вы полагаете, программа будет выполняться правильно; установите в этом месте точку останова; выполните программу вплоть до этой точки.
- 2. Просмотрите значения интересующих Вас переменных, чтобы убедиться в том, что программа работает правильно. Если это не так, перейдите к шагу 4; в противном случае удалите только что установленную Вами

точку останова и установите новую точку останова, опять выбрав оператор, до которого, как вам кажется, программа будет выполняться правильно.

- 3. Запустите программу с только что удалённой Вами точки останова. Когда программа дойдёт до следующей точки останова, вернитесь к шагу 2.
- 4. Теперь Вы приблизительно знаете, где находится ошибка.

Если Вы не знаете, как выполняется Ваша программа, например, если в результате ошибок в программе выполняется не тот оператор, либо происходит переход не на ту точку, можно воспользоваться описанной выше процедурой отладки с некоторыми изменениями.

На шаге 1 выберите *несколько* возможных точек останова; то же сделайте на шаге 2. Кроме того, на шаге 2 удалите *все* точки останова, до которых, по Вашему убеждению, программа не дойдёт, когда Вы её запустите на шаге 3. Далее следите не только за содержимым памяти (значениями переменных), но и за тем, на какую точку останова вышла программа, чтобы убедиться, что это есть запланированная в данном случае точка останова.

Итак, локализация с точками останова используется для программ, слишком больших для отладки с помощью пошагового выполнения и трассировки. Для ещё более длинных программ используется так называемый принцип *разрыва телефонной линии*, к описанию алгоритма которого мы и приступаем.

Установите первую точку останова где–то в середине программы. Когда программа выполнится до этого места (если это произойдёт), проверьте содержимое памяти, чтобы убедиться, что программа до этого места работала правильно. Если это так, то Вы будете знать, что Ваша первая ошибка находится во второй половине программы. В противном случае она в первой половине.

Так или иначе Вы сократили область поиска ошибки до половины программы. Теперь установите точку останова приблизительно в середине этой половины, снова запустите программу и проверьте содержимое памяти. Таким образом Вы локализуете ошибку на участке в четверть программы. Продолжайте процедуру, деля программу каждый раз приблизительно пополам, пока Вы не сократите область поиска ошибки до таких размеров, при которых можно воспользоваться обычной методикой отладки с точками останова.

Если ошибки в программе приводят к неправильному порядку выполнения программных строк, предложенный метод следует, как и ранее, несколько изменить.

Может, например, получиться, что Вы установили точку останова в середине некоторого участка программы, но при выполнении программа проходит вообще мимо точки останова. Конечно, это все же локализует ошибку: она в первой половине этого участка. Однако Вы всегда можете установить несколько точек останова, как и при обычной отладке с точками останова.

VIII.3.4. Программная обработка ошибок

Don't worry, computer bugs don't byte.

—Из программистского фольклора

Кроме сбоев в работе компьютера из–за неисправностей каких–либо его узлов имеется достаточно много причин, по которым происходит прерывание вычислений по программе и при нормально работающем компьютере.Перечислим некоторые из них:

1. компьютеру «предложено» поделить на нуль;
2. при вычислениях получено число большее, чем допустимо;
3. для элемента массива получено значение индекса, не лежащее в диапазоне, указанном оператором `DIM`;
4. в программе встретилась функция пользователя `FN`, которая не описана.

Все эти и многие подобные им причины, вызывающие прерывания, имеют конкретные номера от 1 до 255, называемые *кодами* ошибок. Если какая–нибудь «ошибка» происходит, то на экран выводится соответствующее сообщение. Например:

«NEXT without FOR in 40»
(«NEXT *без* FOR в *строке* 40»).

После устранения причин, вызвавших прерывание программы, как правило, приходится запускать её заново.Однако имеется возможность «обработать» ошибку, не прекращая вычислений. Для этих целей используются операторы:

```
ON ERROR GOTO n
ON ERROR GOTO 0
```
,

где:

- ON(«на»), ERROR(«ошибка»), GOTO — служебные слова;
- n — номер программной строки.

При выполнении оператора ON ERROR GOTO n происходит назначение передачи управления на строку с номером n,но сама передача реализуется *лишь* в случае возникновения ошибки.

Оператор ON ERROR GOTO 0 отключает обработку ошибок пользователем и включает системную обработку ошибок.

Фрагмент программы, начинающийся со строки с номером n и заканчивающийся одним из операторов:

```
RESUME 0 или RESUME
RESUME NEXT
RESUME m
```

называется подпрограммой *обработки ошибок*.
Здесь:

- RESUME («продолжаю»), NEXT — служебные слова;
- m — номер программной строки.

После обработки ошибки в зависимости от значения параметра, расположенного за служебным словом RESUME, возврат в основную программу осуществляется:

- 1) либо к оператору, вызвавшему ошибку, в случае операторов

  ```
  RESUME 0
  ```

  или

  ```
  RESUME
  ```

- 2) либо к следующему за ним в случае оператора

  ```
  RESUME NEXT
  ```

  причём этот оператор может находиться в той же строке, в которой была обнаружена ошибка;

- 3) либо к программной строке с номером m в случае оператора

  ```
  RESUME m
  ```

Разумеется, старайтесь не возвращаться «внутрь» цикла, минуя заголовок, и «внутрь» другой подпрограммы!

Примеры:

- 1)
  0834-01.bas
  0834-01.bas

  ```
  10 ON ERROR GOTO 50:INPUT M
  30 IF A(M)=0 THEN ?"Ошибки нет!":GOSUB 40:END
  40 PRINT"Осуществлен переход на оператор GOSUB 40!":RETURN
  50 RESUME NEXT
  run
  ```

```
? 1
Ошибки нет!
Осуществлен переход на оператор  GOSUB 40!
Ok
run
? 30
Осуществлен переход на оператор  GOSUB 40!
Ok
```

- 2)
0834-02.bas
0834-02.bas

```
10 ON ERROR GOTO 70:INPUT W
30 IFA(W)=1: ?"4" THEN:?"5":END
           ▲      ▲
     └──────┘└── Ошибки
70 RESUME NEXT
run
? 30
45
Ok
```

Подпрограмма обработки ошибок должна заканчиваться операторами `RESUME`, `END` или `STOP`, в противном случае последует сообщение об ошибке:

«No RESUME in ∗∗∗∗»
(«*Отсутствие оператора* RESUME *в строке* ∗∗∗∗»).

При установке «ловушек» ошибок можно не ограничиваться стандартными сообщениями MSX BASIC об ошибках.

Если в Вашей программе требуется, чтобы вводимое число находилось в интервале между 0 и 1000, нарушение этого требования можно рассматривать как ошибку, которой присваивается собственный код.

Конечно, Вы заметили, что в MSX BASIC задействованы не все номера кодов ошибок. Так, коды 26÷49 и 60÷255 находятся в распоряжении программиста. (В последующих версиях MSX BASIC эти значения могут изменяться!)

Для подпрограммы обработки указанной ошибки ввода возьмём код 255. Соответствующие операторы могут иметь такой вид:

```
10 ON ERROR GOTO 1000
          …
50 IF X<0 OR X>1000 THEN ERROR 255
          …
1000 IF ERR=255 THEN PRINT"Число вне диапазона"
1010 RESUME
```

Строка 50 при обнаружении ошибки вызывает автоматический переход к подпрограмме.

Формирование собственных сообщений об ошибках приносит не много пользы. Этот способ несколько неуклюж: он требует наличия двух операторов`IF…THEN`, тогда как фактически достаточно одного. Главное его преимущество состоит в том, что появляется возможность сгруппировать в программе все сообщения об ошибках и сделать их единообразными, что облегчает процедуру пополнения списка обрабатываемых ошибок.

MSX BASIC обеспечивает одновременную обработку *только одной* ошибки. Если оператор `ON ERROR GOTO` отсылает какую-либо программу к подпрограмме обработки ошибок, в которой возникает ещё одна ошибка, эта ошибка не будет обработана, но она вызовет появление сообщения об ошибке и прекращение счета.

Отметим, что в слове ONELIN рабочей области по адресу &HF6B9 хранится ссылка на адрес первой программной строки подпрограммы обработки ошибок.

Пример 3.
0834-03.bas

0834-03.bas

```
10 ON ERROR GOTO 90
20 INPUT
30 END
90 A=A+1
100 RESUME
```

Выполним эту программу. А теперь...

```
? HEX$(PEEK(&HF6BA));HEX$(PEEK(&HF6B9))
801B  ◄── Мы получили адрес PIT, начиная с которого расположена первая строка подпрограммы обработки ошибок.
Ok
```

А теперь, просмотрев PIT при помощи «палочки-выручалочки» — оператора PEEK, получим:

Адрес	Содержимое	Комментарий
&H801B	&H25	Указатель на адрес программной строки, следующей за строкой 90
&H801C	&H80	
&H801D	&H5A=90	Номер первой программной строки подпрограммы обработки ошибок
&H801E	0	
&H801F	&H41	Код символа «A»

В подпрограммах обработки ошибок обычно используются функции:

- `ERR`, где ERR («ERRor») — служебное слово, которое возвращает код последней ошибки, и
- `ERL`, где ERL («ERror Line» — «ошибочная строка»)

— служебное слово, которое возвращает номер строки, где произошла ошибка.

Примеры:

- 4)
  0834-04.bas
  0834-04.bas

```
10 Z=1/0
run
Division by zero in 10
Ok
print err;erl
 11  10
Ok
```

- 5)
  0834-05.bas
  0834-05.bas

```
10 ON ERROR GOTO 100
20 INPUT A:Z=1/A:PRINT Z:END
100 ?"Ошибка!":? ERR;ERL:RESUME NEXT
run
?   нажата клавиша "RETURN" ⟶A=0
Ошибка!
 11  20
 0
Ok
```

Значение функции ERL хранится в слове ERRLIN области системных переменных по адресу &HF6B3

Например:

```
1000 PRINT 1/0
run
Division by zero in 1000
Ok
print peek(&HF6B3)+256*peek(&HF6B4)
 1000
Ok
```

Далее, оператором

```
ERROR α
```

,

где:

- ERROR — служебное слово;
- α — арифметическое выражение;

можно искусственно вызвать (имитировать!) ошибку с кодом, равным INT(α) (разумеется, 0≤INT(α)≤255 !).

При этом ошибка с заданным кодом возникает в том месте программы, в котором находится оператор `ERROR α`.

Заметим, что можно легко, зная код ошибки, восстановить «содержание» ошибки с помощью команды(!) `ERROR` в непосредственном режиме. Например:

```
Ok
error 73
Unprintable error
Ok

Ok
error 15
String too long
Ok

Ok
error 0
Illegal function call
Ok
```

Примеры:

- 6)
  0834-06.bas
  0834-06.bas

```
10 ON ERROR GOTO 80
20 E=10:Z=0
30 P=E/Z:PRINT "P=";P
40 Q=LOG(E-11)
50 ERROR 250
60 PRINT "Завтра":END
70 'Обработка ошибок
80 PRINT ERL,ERR
90 IF ERR=11 THEN 130
110 IF ERR=250 THEN 150
120 IF ERR=5 THEN 140
130 Z=.1:RESUME 0
140 RESUME NEXT
150 RESUME 60
run
 30            11
P= 100
 40            5
 50            250
Завтра
Ok
```

- 7)
  0834-07.bas
  0834-07.bas

```
10 ON ERROR GOTO 100
20 X=0
30 INPUT"Сколько миль до Луны";FAR
40 IF FAR<>238857.0! THEN 50
45 PRINT"Отлично!"
48 GOTO 130
50 X=X+1
55 IF X>2 THEN ERROR 200
```

```
60 GOTO 30
100 IF ERR<>200 THEN ON ERROR GOTO 0'Обработать ошибку как обычно!
110 PRINT"Уже три попытки. Хватит!"
120 RESUME 130
130 END
run
Сколько миль до Луны? 500000
Сколько миль до Луны? 10000
Сколько миль до Луны? 68999
Уже три попытки. Хватит!
Ok
```

Отметим, что обработка ошибок пользователем действительна и в непосредственном режиме: Вы можете выполнить в непосредственном режиме команду `ON ERROR GOTO`, а затем ввести команду, которая вызовет обрабатываемую ошибку, например:

```
20 PRINT 1:END
Ok
on error goto 20:max=0
 1                ▲
Ok                └── "Сомнительное" имя!
```

Если Вы не обнаружили причины появления ошибки и номер строки, в которой она произошла, то это может привести к «зацикливанию» Вашей программы, избавиться от которого вам поможет включение в подпрограмму обработки ошибок оператора:

```
IF err=X AND erl=Y THEN on error goto 0:resume next
                   ELSE X=err:Y=erl:resume next
```

Пример 8.
0834-08.bas
0834-08.bas

```
10 ON ERROR GOTO 50
20 PRINT MAX
30 END
50 PRINT 2:IF ERR=XANDERL=Y THEN ON ERROR GOTO 0:RESUME NEXT ELSE X=ERR:Y=ERL:RESUME NEXT
run
 2
Ok
print x;y
 2  20
Ok
```

Список ошибок может быть дополнен ошибками с кодами и названиями, придуманными пользователем! Для этого зарезервированы ошибки с номерами от 71 до 255 для MSX Disk BASIC (и с номерами от 60 до 255 для MSX BASIC).

Если вам потребуется ввести свои коды ошибок, то сделайте это по аналогии с фрагментом программы:

Пример 9.
0834-09.bas
0834-09.bas

```
10 ON ERROR GOTO 150
20 INPUT X
30 IF X<0 THEN ERROR 250
40 PRINT SQR(X)
50 END
150 IF ERR=250 THEN PRINT"Аргумент меньше нуля"
160 RESUME NEXT
run
? 10
 3.1622776601684
Ok
```

```
run
? -10
Аргумент меньше нуля
Ok
```

Если код ошибки, стоящей в операторе ERR0R α, не определён, то будет выдано сообщение об ошибке: «Unprintable error», и программа будет прервана.

VIII.3.5. Некоторые причины, осложняющие поиск ошибок [57]

> Следствие 17: некомпетентность не знает преград ни во времени, ни в пространстве.
>
> *—Л.Питер*

1. При отладке достаточно сложных программ, использующих большой набор обрабатываемых данных, иногда трудно вообще установить факт наличия ошибок в программе, так как программист лишь приблизительно представляет, какой результат должна давать программа. Например, интегрируя систему дифференциальных уравнений, мы, как правило, лишь качественно представляем картину изменения искомых функций и при этом не всегда можем определить количество значащих цифр полученного результата.

   Такое положение вещей заставляет иногда на стадии отладки прибегать к дублированию получения результата другим методом. В нашем примере это можно сделать, проинтегрировав систему уравнений с помощью другой программы интегрирования, и затем сравнить полученные результаты. Правда, при этом возникает вопрос: как быть в случае расхождения результатов? Ведь ошибочным может оказаться не первоначальный, а как раз контрольный результат!

2. В программе, как правило, содержится не одна, а несколько ошибок. Поэтому программист наблюдает эффект не одной ошибки, а результат взаимодействия нескольких ошибок.
3. Трудно, а иногда и просто невозможно разработать достаточно полную систему тестов, гарантирующих обнаружение всех ошибок в программе.
4. Одни и те же признаки ошибки (формы проявления ошибок) могут быть обусловлены различными причинами.
5. Некоторые ошибки не проявляются сами по себе, а лишь приводят к возникновению других ошибок (так называемые *наведённые* ошибки), которые и наблюдает программист.

   Например, обнуляя значения элементов массива и по ошибке выйдя за его границы, можно присвоить нулевое значение некоторой переменной, при использовании которой в арифметическом выражении будет зафиксировано деление на нуль, хотя вначале этой переменной было присвоено ненулевое значение и она нигде не изменялась.

6. Некоторые ошибки нельзя выявить, разбивая программу на части и отлаживая эти части по отдельности. Ошибка возникает лишь при взаимодействии этих частей. Таким образом, стратегия «разделяй и властвуй» не всегда оказывается применимой.
7. Иногда внесение каких–либо изменений в программу с целью локализации ошибки (например, промежуточная печать данных, трассировка и т.п.) приводит к исчезновению проявлявшихся до этого признаков ошибки или к их изменению. Получается своеобразный заколдованный круг, когда всякая попытка выявить ошибку лишь маскирует её, не давая никакой информации. Эта ситуация схожа с ситуацией, имеющей место в физике микромира: использование какого–либо прибора для наблюдения процесса полностью изменяет этот процесс.
8. Иногда, изменяя программу, методом проб и ошибок, можно устранить ошибку, т.е. она перестаёт как–либо проявлять себя. При этом остаётся абсолютно непонятным, в чем же она заключалась.
9. > Происходит не просто нечто более странное, чем мы предполагали:странность происходящего превышает и то, чего мы не смели предположить.
   >
   > *—Принцип Ожидаемого по Питеру*

   В ходе отладки программист нередко допускает просчёт, необоснованно принимая некоторые предположения о возможных источниках ошибок.

   Например, используя стандартную библиотечную подпрограмму,он полностью уверен в её безошибочности. Такие

неверные установки, как правило, либо заводят программиста в логический тупик, либо программист впустую тратит время, пытаясь обнаружить ошибку там, где её нет.

10. Не всегда имеет место повторяемость ошибки от запуска программы к запуску, даже если в программу и данные не вносились изменения.

    Наиболее часто это возникает при наличии в программе переменных, которым не присвоено начальное значение.В этом случае начальное значение этой переменной случайным образом зависит от содержимого соответствующей ей ячейки памяти в момент загрузки программы. В зависимости от этого значения возможно возникновение ошибочных ситуаций.

11. При разработке большого программного комплекса отдельные части программы создаются разными исполнителями, что значительно затрудняет согласование между частями.

Отладка программы — это прежде всего эксперимент, а не наблюдение за поведением программы. Различие между этими двумя понятиями удачно и точно охарактеризовал знаменитый русский физиолог И.П.Павлов: «Наблюдение собирает то, что ему предлагает природа, опыт же берёт у природы то, что хочет». И, как всякий эксперимент, отладку нужно уметь проводить. Очень важно при этом делать правильные выводы на основании данных, полученных из эксперимента. То, насколько при этом можно ошибиться, наглядно демонстрирует следующая шутливая история [57].

Некий школьник предложил интересную гипотезу: он утверждал, что органы слуха у пауков находятся на ногах, и взялся доказать это. Положив пойманного паука на стол, он крикнул: «Бегом!». Паук побежал. Мальчик ещё раз повторил свой приказ. Паук снова побежал. Затем юный экспериментатор оторвал пауку ноги и, снова положив его на стол, скомандовал: «Бегом!». Но на сей раз паук остался неподвижен. «Вот видите, — заявил торжествующий мальчик, — стоило пауку оторвать ноги, как он сразу оглох.

А «окончив» отладку, вспомните, что когда известного датского скульптора Торвальдсена (1768 или 1770–1844) спросили мнение об одной из его скульптур, он ответил: «Я не вижу в ней недостатков, из чего заключаю,что у меня хромает воображение».

VIII.4. Принципы исправления и анализа допущенных ошибок

> Программа, свободная от ошибок, есть абстрактное теоретическое понятие.
>
> *—Д.Ван Тассел*

Ясно, что процесс отладки складывается из двух этапов: определение местонахождения ошибки и последующего её исправления. Поговорим о принципах исправления ошибок по Майерсу [51].

1. **Там, где есть одна ошибка, вероятно, есть и другие.**

Другими словами, ошибки имеют тенденцию группироваться. При исправлении ошибки проверьте её непосредственное окружение: нет ли здесь каких–нибудь подозрительных симптомов.

2. **Находите ошибку, а не её симптом.**

Другим общим недостатком является устранение симптомов ошибки, а не её самой. Если предполагаемое изменение устраняет не все симптомы ошибки, то она не может быть полностью выявлена.

3. **Вероятность правильного нахождения ошибки не равна 100%**

С этим, безусловно, соглашаются, но в процессе исправления ошибки часто наблюдается иная реакция (например «да, в большинстве случаев это справедливо, но *данная* корректировка столь незначительна, что она правильна»).

Никогда нельзя предполагать, что текст, который включён в программу для исправления ошибки, правилен!

Можно утверждать, что корректировки более склонны к ошибкам, чем исходный текст программы. Подразумевается, что корректирующая программа должна тестироваться, возможно, даже более тщательно, чем исходная.

4. **Вероятность правильного нахождения ошибки уменьшается с увеличением объёма программы.**

Это утверждение формулируется по–разному. Эксперименты показали,что отношение числа неправильно найденных ошибок к числу первоначально выявленных увеличивается для больших программ. В большой программе, рассчитанной на широкое применение, каждая шестая вновь обнаруженная ошибка может быть допущена при предшествующем внесении изменений в программу.

5. **Остерегайтесь внесения новой ошибки при корректировке.**

Необходимо рассматривать не только неверные корректировки, но и те, которые кажутся верными, однако имеют нежелательный побочный эффект и таким образом приводят к новым ошибкам. Другими словами, существует вероятность не только того, что ошибка будет обнаружена неверно, но и того, что её исправление приведёт к новой ошибке. Поэтому после проведения корректировки должно быть выполнено повторное тестирование, позволяющее установить, не внесена ли новая ошибка.

> Когда кто выходит из дому, пусть поразмыслит о том, что намерен делать, а когда снова войдёт в дом, пусть поразмыслит о том, что сделал.
>
> *—Древнегреческий мыслитель Клеобул*

Укажем один старый приём исправления ошибок, заключающийся в использовании так называемых *«заплат»* («patch» — «заплата», «вставка в программу»).

Необходимость в «заплате» возникает, когда Вы хотите вставить последовательность новых операторов между двумя операторами, которые мы обозначим O1 и O2. Организация заплаты происходит при помощи оператора `GOTO` (тут мы отступаем от одного из основных принципов структурного программирования!):

```
30 O1:GOTO 1000 'Переход на операторы "заплаты"
40 O2
     …
500 END 'Окончание основной программы.
     …
1000 … 'Операторы "заплаты"
1010 GOTO 40
```

Никогда не оставляйте «заплаты» в *отлаженной* программе!

Удалить «заплату» — это значит включить операторы заплаты в основной текст программы, расположив их там, где им надлежит находиться.

Интересная невыдуманная история с «заплатой» произошла на корабле «Аполлон», облетающем Луну. Бортовая ЭВМ выдала сигнал тревоги и отказалась выполнять дальнейшие вычисления. Космонавты быстро обнаружили,что неисправен аварийный датчик, дающий неправильный отсчёт. Программист на Земле написал текст «заплаты» для программы обработки аварийных сигналов, изменяющий её так, чтобы конкретный аварийный сигнал не считался аварийным. Эта «заплата» была написана прямо в кодах, и необходимые изменения текста бортовой программы на машинном языке были продиктованы космонавтам по радио. Программа с «заплатой» благополучно довела космонавтов до Земли.

Однако учтите, что «система, состоящая из «заплаток», возникших при исправлении ошибок, редко оказывается понятнее системы, которая с самого начала не имела ошибок» (Дж.Фокс).

Качество работы каждого отдельного программиста существенно повышается, если выполняется детальный анализ обнаруженных ошибок или,по крайней мере, их подмножества. Эта задача трудная и требующая больших временных затрат, поскольку она подразумевает нечто большее, чем просто поверхностную классификацию, такую, как «X% ошибок являются ошибками в логике» или «Y% ошибок встречается в операторах IF». Тщательный анализ может включать в себя рассмотрение следующих вопросов.

1. *Когда была сделана ошибка*?

Данный вопрос является наиболее трудным, так как ответ на него требует исследования документации. Однако это и наиболее интересный вопрос. Необходимо точно определить первоначальную причину и время возникновения ошибки. Такой причиной может быть, например, неясная формулировка в постановке задачи или коррекция предшествующей ошибки.

2. *Кто сделал ошибку*?

3. *Какова причина ошибки*?

Недостаточно определить, когда и кем была сделана ошибка, нужно также выяснить, *почему* она произошла. Была ли она вызвана чьей–то неспособностью писать ясно, непониманием отдельных конструкций языка программирования, ошибкой при печатании на машинке, неверным предположением, отсутствием рассмотрения недопустимых входных данных?

4. *Как ошибка могла быть предотвращена*?

Ответ на этот вопрос наиболее ценен, так как позволяет осмыслить и количественно обосновать накапливаемый опыт проектирования.

5. *Почему ошибка* не *была обнаружена ранее*?

6. *Как ошибка могла быть определена ранее*?

Ответ на этот вопрос является другим примером полезной обратной связи. Как могут быть улучшены процессы обзора и тестирования для более раннего нахождения этого типа ошибок в будущих проектах?

7. *Как была найдена ошибка*?

При условии, что мы рассматриваем только ошибки, которые обнаружены с помощью теста, необходимо выяснить, как был написан удачный тест. Почему этот тест был удачным? Можем ли мы что–нибудь почерпнуть из него для написания других удачных тестов с целью проверки данной программы или будущих программ?

Такой анализ, конечно, является сложным процессом, но его результаты могут оказаться полезными для дальнейшего улучшения работы программиста.
Поэтому вызывает опасения тот факт, что подавляющее большинство программистов его не используют!

VIII.5. Основные понятия структурного программирования

Высокое качество программ может достигаться «безошибочным» программированием (*«пассивными»* методами) и выявлением и устранением ошибок (*«активными»* методами). Активные методы мы уже кратко описали.

Пассивные методы основываются на применении методологических и организационных правил проектирования программ, а также языков программирования высокого уровня.

VIII.5.1. Модульность программ [49]

[49]

Модульной называют программу, составляемую из таких частей — *модулей*, что их можно независимо друг от друга программировать, транслировать, отлаживать (проверять, исправлять). Предполагается, что модули имеют небольшие размеры, чётко определённые функции и, кроме того, их связи между собой максимально упрощены, в частности, предполагается,что модули имеют лишь одну точку входа (в начале модуля). Разбиение программы на модули при её написании, хотя и является весьма непростым делом, позволяет существенно облегчить в дальнейшем работу над программой на других этапах.

После того как в алгоритме выявлены мало зависимые друг от друга части, составление программы упрощается, так как при программировании каждой из этих частей почти не приходится заботиться об их взаимодействии с другими частями, что в свою очередь способствует уменьшению количества вносимых ошибок. Кроме того, малая зависимость модулей позволяет при необходимости существенно распараллелить составление программы, поручив программирование программистам разного класса, причём всегда можно найти подходящую работу и для начинающих, и для опытных программистов.

На этапе отладки независимость модулей позволяет отлаживать их в любом порядке, в частности и одновременно. Считается, что **усилия, затрачиваемые на отладку модуля, обычно пропорциональны квадрату его длины [Майерс]**, и поэтому при тестировании небольшие размеры модулей дают возможность поставить задачу о проверке

всех ветвей таких модулей, что ведёт к увеличению достоверности тестирования. Решение такой задачи является обычно недостижимым по отношению ко всей программе или крупным её блокам, когда приходится ограничиваться лишь проверкой работы всех линейных участков блока и условий. Разумеется, и наиболее трудная часть отладки — локализация ошибок, проводимая для модулей, при этом значительно упрощается и ускоряется.

В силу минимальности логических связей между модулями облегчается, конечно, и внесение исправлений в алгоритм программы, поскольку меньше приходится заботиться о том, чтобы при изменении одной части программы не испортить работу другой её части.

Учтите, что чем более мелкими требуется получать модули, тем больше трудностей возникает при проектировании и алгоритмизации программы, но тем легче будет каждый из модулей проверять и тестировать в дальнейшем. Не следует, однако, забывать и о том, что слишком большое количество мелких модулей может значительно увеличить трудоёмкость предстоящей комплексной (стыковочной) отладки.

Замечание.

Серьёзной помощью в разработке программ могут стать *библиотек и стандартных*, или *типовых*, модулей, заранее составленные автором или другими программистами. Применение при разработке ранее многократно опробованных модулей, трудность использования которых сводится только к заданию правильных аргументов, значительно ускоряет составление программы и облегчает её отладку.

VIII.5.2. Строение программ [49]

[49]

Не претендуя на полноту классификации, строение программ можно охарактеризовать одной из следующих схем:

1. *Монолитное*. Программа написана цельным куском, без выделения каких–либо отдельных независимых частей;
2. *Монолитно–модульное*. Имеется достаточно большая монолитная главная часть программы, в которой производятся основные вычисления, и из которой происходят последовательные обращения к модулям;
3. *Последовательно–модульное*. Центральная часть программы состоит из последовательно выполняемых модулей, которые в свою очередь обращаются к другим модулям;
4. *Иерархическое*. Программа состоит из модулей, связи между которыми подчиняются строгой иерархии: каждый модуль может обращаться только к модулям, которые ему непосредственно подчинены. Возврат всегда должен происходить в вызывающий модуль, даже в том случае, если в вызываемом модуле обнаруживается ошибка, препятствующая дальнейшим вычислениям (правда, не все языки программирования имеют средства для выполнения этого требования);
5. *Иерархически–хаотическое*. Иерархическая (или последовательная) подчинённость модулей нарушена дополнительными связями.
6. *Модульно–хаотическое*. Программа состоит из модулей, но связи их между собой не отвечают принципу иерархии (или последовательности).

Последовательно–модульное и иерархическое (для более сложных программ) строение, как наиболее простые по логическим связям, являются теми образцами, к которым необходимо стремиться при разработке программы. Допустимыми вариантами являются иерархически–хаотическое и, может быть, монолитно–модульное.

Помимо модульности другим свойством, которое содействует предупреждению появления в программе ошибок, является структурированность.

Обычно *структурированной* называется программа, логическая структура которой отвечает некоторым жёстко установленным требованиям.

Уже модульную программу можно иногда считать в определённой степени структурированной, поскольку от модульной программы требуется, например, чтобы она состояла только из модулей с одним входом.

VIII.5.3. Структурное программирование

> Структура (от лат. «structura» — «строение, расположение, порядок»), совокупность устойчивых связей объекта, обеспечивающих его целостность и тождественность самому себе, т.е. сохранение основных свойств при различных внешних и внутренних изменениях.
>
> *—Советский Энциклопедический Словарь*

Впервые основные идеи структурного программирования были высказаны Эдсгером Дейкстрой в 1965 году и позже опубликованы в его работе [55]. Основная задача, которую Э.Дейкстра решал, разрабатывая идеи структурного программирования, была задача доказательства правильности программы. Его внимание было сосредоточено на вопросе, «какими должны быть структуры программ, чтобы без чрезмерных усилий мы могли находить доказательство их правильности».

Это особенно важно при разработке больших программных систем. Опыт применения методов структурного программирования при разработке ряда сложных операционных систем показывает, что правильность логической структуры системы поддаётся доказательству, а сама программа допускает достаточно полное тестирование. В результате, в готовой программе встречаются только тривиальные ошибки кодирования, которые легко исправляются.

Очевидно, что уменьшение трудностей тестирования приводит к увеличению производительности труда программистов. Это следует из того, что на тестирование программы тратится от трети до половины времени её разработки. Производительность труда программиста обычно измеряется числом отлаженных операторов, которые он может написать за день. Приближенные оценки показывают, что применение методов структурного программирования позволяет увеличить это число в 5÷6 раз по сравнению с традиционными способами программирования.

Заметим между прочим, что при структурном программировании становится излишним вычерчивание *блок–схем*. Блок–схема вполне структурированной программы настолько тривиально проста, что о программе можно сказать больше по тексту, чем по блок–схеме.

Итак, структурное программирование представляет собой некоторые принципы написания программ в соответствии со строгой дисциплиной и имеет целью облегчить процесс тестирования, повысить производительность труда программистов, улучшить ясность и читабельность программы, а также повысить её эффективность.

В настоящее время вряд ли существует достаточно простое и краткое определение структурного программирования.

Например, Хоор [54] определяет структурное программирование как «систематическое использование абстракции для управления массой деталей и способ документирования, который помогает проектировать программу.»

Структурное программирование можно толковать как «проектирование, написание и тестирование программы в соответствии с заранее определённой дисциплиной» [54].

Х.Миллс, П.Лингер и Б.Уитт в книге [69] использовали такое определение:

- *структуризованная* программа — это программа, составленная из фиксированного базового множества первичных программ.
- *Первичная* программа — это простая программа, не имеющая простых подпрограмм, состоящих более чем из одного узла.
- *Простая* программа — это программа, которая:
    1. имеет один вход и один выход,
    2. для каждого узла существует путь от входа до выхода, проходящий через этот узел.

Суть дела здесь заключается в том, что если программное обеспечение строится только из первичных и простых программ, то логика и сам ход процесса её выполнения значительно проясняются благодаря структуризации. Использование таких (готовых) структур дисциплинирует разработчика программ, что в результате приводит к появлению более понятных программ, в которых, следовательно, имеется меньшее число ошибок.

Перейдём к рассмотрению теоретических оснований и методов структурного программирования.

Теоретической основой структурного программирования принято считать принципы, изложенные в классической работе Бома и Джакопини [40]. Эта работа в оригинале на итальянском языке была опубликована в 1965 г., а в

английском переводе — в 1966 г.

В соответствии с так называемой «структурной» теоремой, сформулированной и доказанной в этой работе, всякая программа может быть построена с использованием только трёх основных типов блоков [40].

1. *Функциональный блок*. Ему в языках программирования соответствуют операторы ввода и вывода или любой оператор (группа операторов) присваивания. В виде функционального блока может быть изображена любая последовательность операторов, выполняющихся один за другим, имеющая один вход и один выход.

2. *Условная конструкция*. Этот блок включает проверку некоторого логического условия (P), в зависимости от которого выполняется либо оператор S1, либо оператор S2. Приведём аналог условной конструкции на языке программирования MSX BASIC:

   ```
   IF P THEN S1 ELSE S2
   ```
   .

3. *Блок обобщённого цикла*. Этот блок обеспечивает многократное повторение выполнения оператора(ов) S, пока выполнено логическое условие P. Аналог блока обобщённого цикла на языке MSX BASIC:

   ```
   n: IF P THEN S ELSE …:GOTO n
   ```
   .

Важной особенностью всех перечисленных блоков является то, что каждый из них имеет один вход и один выход.

Кроме того, блоки S, S1, S2, входящие в состав условной конструкции или блока обобщённого цикла, сами могут быть одним из рассмотренных типов блоков, поэтому возможны конструкции, содержащие «вложенные» блоки. Однако какова бы ни была степень и глубина «вложенности», важно, что любая конструкция в конечном итоге имеет один вход и один выход. Следовательно, любой сложный блок можно рассматривать как «чёрный ящик» с одним входом и одним выходом.

При конструировании программы с использованием рассмотренных типов блоков, эти блоки образуют линейную цепочку так, что выход одного блока подсоединяется к входу следующего. Таким образом, программа имеет линейную структуру, причём порядок следования блоков соответствует порядку, в котором они выполняются. Такая структура значительно облегчает чтение и понимание программы, а также упрощает доказательство её правильности. Так как линейная цепочка блоков может быть сведена к одному блоку, то любая программа может в конечном итоге рассматриваться как единый функциональный блок с одним входом и одним выходом.

Перечислим теперь основные *принципы и методы* структурного программирования.

> Вы говорите, что я повторяюсь.
> Но я повторю.
>
> *—Т.Эллиот*

- I. **Как можно меньше переходов `GOTO`** !

  Э. Дейкстра выразил это таким образом: «Уже давно было замечено, что квалификация программистов является убывающей функцией от плотности предложений `GOTO` в создаваемых ими программах». Причина этого заключается в том, что основные конструкции структурного программирования гораздо более лаконичны и просты, чем их аналоги на неструктурном языке программирования, например MSX BASIC. Отсюда сразу следует, что программы, написанные на MSX BASIC, будут насыщены предложениями `GOTO` (написанными как явно,так и неявно!).

  Четыре предложения структурного программирования на приведённой ниже схеме в той или иной форме используются во многих языках программирования (приведены примеры конструкций языка программирования TURBO Pascal).

Предложения структурного программирования	Неформальное описание	Соответствующая последовательность операторов на языке MSX BASIC
`IF C THEN S1 ELSE S2`	Если условие истинно выполнить предложение S1; в противном случае выполнить предложение S2	`IF C THEN S1 ELSE S2`

`WHILE C DO S`	Повторить предложение S, пока условие C останется истинным (0 или более раз)	`GOTO n`
		`m: S`
		`n: IF C THEN GOTO m`

`REPEAT S UNTIL C`	Повторять последовательность S (один или более раз) до тех пор, пока условие C не станет истинным	`m: S` `IF NOT C THEN GOTO m`

`CASE K OF`
`1: S1`
`2: S2`
`…`
`m: Sm`
	Выполнить предложение Si(только если значение K=i, причём i равно либо 1, либо 2, … либо m (выбор по значению)	`ON K GOTO N1, N2, …, Nm:`
		`GOTO s`
		`N1:S1:GOTO s`
		`N2:S2:GOTO s`
		`…`
		`Nm:Sm`
		`…`
		`s:…`

Обратим Ваше внимание на то, что при программировании конструкций структурного программирования на языке MSX BASIC невозможно обойтись без оператора `GOTO`, с помощью которого осуществляется переход на ту или иную ветвь условной конструкции. Однако следует иметь в виду, что оператор `GOTO` используется только для передачи управления *внутри* конструкции, что не противоречит идеям структурного программирования.

Дейкстра продолжает: «Я пришёл к убеждению, что предложение `GOTO` должно быть устранено из всех языков программирования «высокого уровня» (т.е. отовсюду, за исключением, возможно, простых машинных кодов)».

Однако сейчас стало ясным, что программирование без оператора `GOTO` — это ещё не структурное программирование. Можно написать программу без оператора перехода, логическая структура которой тем не менее будет неудачной. И, наоборот, существуют ситуации, в которых переход является лаконичным, простым и ясным средством, в то время как другие подходы сравнительно неудачны.

Например, правила структурного программирования часто предписывают повторять одинаковые фрагменты программы в разных участках модуля, чтобы избавиться от употребления оператора `GOTO`. В этом случае «лекарство хуже болезни», т.к. дублирование резко увеличивает возможность внесения ошибок при изменении модуля в будущем.

Д.Кнут в работе [70] показал, что можно говорить о структурном программировании и при использовании оператора `GOTO`! Структурное программирование на языках Фортран или BASIC возможно, хотя с большими трудностями и некоторыми нежелательными последствиями. Так, например, Чармонмен и Ведженер [72] показали, что можно сделать программу на языке Фортран похожей на структурную!

- II. Другой метод улучшения качества программирования заключается в применении *нисходящего проектирования*, («top–downprogramming» — «программирование «сверху вниз»»).

Оператор `GOSUB` является *основным* инструментом *структурного* программирования.

В методе нисходящего проектирования Вы вначале пишете основную программу, используя оператор `GOSUB` для вызова подпрограмм, причём в качестве подпрограмм вначале Вы вводите «заглушки» вида:

```
PRINT "Вызвали подпрограмму номер …":RETURN
```

Затем, будучи уверенным в правильности логического построения основной программы, Вы детально «расписываете» каждую подпрограмму, вызывая по мере необходимости подпрограммы более низкого уровня. Этот последовательный процесс продолжается, пока программа не будет завершена и проверена.

При другом методе — *восходящем проектировании* (программировании «снизу вверх») — Вы вначале пишете подпрограммы нижнего уровня и тщательно их тестируете и отлаживаете. Далее Вы добавляете подпрограммы более высокого уровня, которые вызывают подпрограммы нижнего уровня, и так до тех пор, пока Вы не достигнете программы самого верхнего уровня. Метод проектирования «снизу вверх» пригоден при наличии больших библиотек стандартных подпрограмм.

Учтите, что иногда лучшим является гибрид двух методов. Однако в обоих случаях каждая подпрограмма должна быть небольшой, так, чтобы можно было охватить одним взглядом всю её логику (для персональных компьютеров желательно, чтобы и основная программа, и подпрограммы *целиком* помещались в пределах 20÷30 строк экрана дисплея!)

Всякий велосипедист хорошо знает, что ехать сверху вниз быстрее и удобнее, чем снизу вверх. В программировании дело обстоит примерно так же: «сверху вниз» писать программы удобнее потому, что при таком методе мы точно знаем, какие подпрограммы описывать.

Но есть у этого метода и недостаток: на верхнем уровне не всегда видно, куда спускаться, то есть как разделить решение задачи на такие части, каждую из которых было бы легко описать отдельной процедурой. У опытных программистов вырабатывается своеобразное чутье: они сразу видят, какие нужны процедуры, а новичкам иногда приходится туго.

Метод «снизу вверх», хотя и требует большого труда, бывает очень полезен на первых порах. Пусть даже половину составленных Вами подпрограмм придётся потом «выбросить», но зато Вы хорошо почувствуете, какие подпрограммы для исходной задачи необходимы. Да и отлаживать каждую написанную подпрограмму можно сразу: ведь все, что «под ней», уже описано (а обычно и отлажено). Словом, любишь кататься «сверху вниз» — люби и саночки возить (в обратном направлении). Опытные программисты иногда применяют метод «снизу вверх» для того, чтобы заранее заготовить для новой задачи набор подпрограмм, которые могут понадобиться в различных случаях. Так что «возить саночки» приходится не только новичкам!

- III. Структурное программирование до сих пор было у нас представлено как свойство или оценка *окончательного* текста программы.
  Необходимо добавить ещё один ключевой момент — методологию, или особенности мыслительного процесса, управляющего процессом получения структурной программы. Этот мыслительный процесс называется *пошаговой детализацией* и был первоначально предложен Э.Дейкстрой [73], а затем улучшен Н.Виртом [74].

  Пошаговая детализация представляет собой простой процесс, предполагающий первоначальное выражение логики программы в терминах гипотетического языка «очень высокого уровня» с последующей детализацией каждого предложения в терминах языка более низкого уровня, до тех пор, пока, наконец, не будет достигнут уровень используемого языка программирования.

  Причём на протяжении всего процесса логика выражается основными конструкциями структурного программирования.

  В методе пошаговой детализации можно выделить следующие существенные этапы [8]:

  1. На уровне 1 создаётся общее описание программы в целом. Определяются основные логические шаги, требуемые для решения задачи, даже если пока неизвестно, как их выполнить. Эти логические шаги могут отражать различные физические шаги решения или могут быть удобными групповыми именами для тех действий, выполнение которых представляется довольно смутно. Последовательности шагов, требуемых для решения задачи, записываются на обычном языке или на *псевдокоде* (см. ниже).
  2. На уровне 2 в общих терминах детализируется описание шагов, введённых на этапе 1. В детализированное описание может входить обозначение циклических структур, в то время как действия внутри циклов могут по–прежнему оставаться неясными. Таким образом, выполняются только общие эскизы сложных действий.
  3. На этом и последующих уровнях в виде последовательных итераций производятся те же действия, что описаны на этапе 2. При каждой новой итерации уточняются детали, оставшиеся неясными после предыдущих итераций, и создаются более определённые описания. По мере выполнения итераций неопределённые детали становятся все проще и проще, так что на каком–то этапе могут быть полностью описаны.
  4. Разработка завершена: в модульном виде получено описание требуемой программы. Перевод этого описания в программу на конкретном языке программирования должен быть достаточно простой задачей.

  Псевдокод включает в себя наборы фраз для написания таких групп операторов: последовательность, выбор, итерация, — дополняемых текстом на обычном языке. Псевдокод не имеет строгого определения, поэтому Вы всегда можете сконструировать свой *собственный псевдокод*, используя, например, конструкции школьного алгоритмического языка: если , пока , для , выбор , а также комментарии, формулы и словесное описание действий и процессов.

  В описание процессов могут входить и такие операторы конкретного языка программирования (например, BASIC), как `INPUT`, `PRINT`, `READ` и присваивания, но не операторы *перехода* или другие средства передачи управления, применение которых должно ограничиваться реализацией трёх указанных выше типов структур на заключительном этапе процесса проектирования.

  Концепцию псевдокода легче всего уяснить на примере.

  Пусть требуется определить наибольшее значение в некотором наборе данных и вывести эти данные, поделённые на наибольшее значение. Скажем, если данные представляют собой последовательность чисел

  ```
  4., 2.51, 10.0, -5.0, 7.5
  ```
  ,

  то вывод должен выглядеть следующим образом:

  ```
  0.40, 0.251, 1.00, -0.5, 0.75
  ```
  .

  Первый уровень разработки ясен:

Уровень 1:

1. ввести данные;
2. найти максимум введённых данных;
3. вывести результаты.

Детализация 1.1. *Ввод данных* можно детализировать на псевдокоде следующим образом:

1. определить количество чисел;
2. пока не все элементы введены, прочитать и запомнить значение элемента;
3. конец цикла.
   - Это описание можно перевести на язык MSX BASIC следующим образом:

```
100 INPUT"Укажите количество элементов";N
110 FOR I=1 TO N:INPUT A(I):NEXT I
```

Детализация 1.2. *Отыскание максимума* можно детализировать следующим образом:

1. выбрать в качестве максимума первый элемент данных;
2. пробежать все введённые значения, заменяя текущий максимум на очередное значение, если оно не превысило его.

Детализация 1.3. *Вывод результатов* можно детализировать следующим образом:

1. пока не все результаты выведены;
2. ВЫВОД значение очередного элемента, поделённое на максимум;
3. 3) конец цикла.
   - Это описание можно немедленно перевести на MSX BASIC следующим образом:

```
300 FOR I=1 TO N:PRINT A(I)/M:NEXT I
```

Уровень 2.

Он включает в себя три детализированные выше части, из которых только детализация (1.2) требует дополнительного внимания. Её можно детализировать на псевдокоде следующим образом:

1. положить M, равное первому элементу данных;
2. пока не все элементы просмотрены;
3. если M<текущий элемент, то M = текущий элемент;
4. конец цикла.
   - Это описание можно перевести на MSX BASIC следующим образом:

```
200 M=A(1)
210 FOR I=1 TO N
220   IF M<A(I) THEN M=A(I)
230 NEXT I
```

Так как приведённые выше модули используются только по одному разу и очень просты,то можно не делать из них подпрограммы, а объединить их вместе в одну программу:
0853-01.bas
 0853-01.bas

```
 10 'Пример программы, полученной из псевдокода
   20 DIM A(20)
┌──────────────────────────────────────────┐
│   100 INPUT "Введите число элементов";N  │
│   110 FOR I=1 TO N:INPUT A(I):NEXT I     │
└──────────────────────────────────────────┘

┌──────────────────────────────────────────┐
│   200    M=A(1)                          │
│   210    FOR I=1 TO N                    │
│   220       IF M<A(I) THEN M=A(I)        │
│   230    NEXT I                          │
└──────────────────────────────────────────┘

┌──────────────────────────────────────────┐
│   300 FOR I=1 TO N:PRINT A(I)/M;:NEXT I  │
└──────────────────────────────────────────┘

   500 END
   run
Введите число элементов? 5
   ? 4
```

```
? 2.51
? 10.0
? -5.0
? 7.5
 .4  .251  1 -.5  .75
Ok
```

При решении реальной задачи может потребоваться написать на псевдокоде много уровней, чтобы довести все модули до такого состояния, при котором они окажутся готовыми для программирования.

> Известно, что отладка в два раза сложнее написания программы. Поэтому, если Вы были предельно хитроумны при написании программы, то что же Вы будете делать при её отладке?
>
> *—Б.Керниган, Ф.Плоджер*

- IV. **Никаких трюков и заумного программирования!**

  Трюками мы называем необычные приёмы программирования. Трюк должен быть лаконичным и давать выигрыш в быстродействии или в объёме программы; каждый трюк несёт в себе элемент неожиданности.

  Отношение к трюкам может быть различным. Некоторые уравновешенные люди признают, что в этом что-то есть, но избегают фокусов, дабы не усложнять жизнь. Другие получают в трюках эстетическое удовольствие и восхищаются ими настолько, что забывают о назначении программы. Третьи находят прямую связь между трюками, трюкачеством и «бит-жонглёрством» и считают все это безусловно вредными привычками плохо воспитанных программистов.

  Вероятно, в каждом из этих мнений есть что-то от истины. Однако мы считаем, что к *месту* употреблённый трюк, снабжённый, когда надо, комментариями, ничего, кроме пользы, принести не может. Некоторые трюки, входя в обиход, становятся привычными и воспринимаются как нормальные приёмы программирования. В большинстве случаев чистый выигрыш от них невелик, но бывают ситуации, когда время работы программы жёстко ограничено и только трюк может спасти положение. Наконец, знакомство с трюками полезно и тем, кто их не использует, так как позволяет глубже осознать особенности компьютера и чувствовать себя свободно. Из-за своей необычности трюки, как правило, реализуются только на языках ассемблерного типа или непосредственно в машинных кодах.

  Однако никогда не используйте трюков там, где можно использовать простые методы. Заметим, что *элегантное* решение задачи — это такое решение, которое одновременно и просто и оригинально. Простые решения всегда желательны, но вопрос о том, всегда ли приемлемы оригинальные решения (трюки), остаётся открытым.

  Под *оригинальностью* решения подразумевается его неочевидность.

  Кстати, Оксфордский словарь английского языка определяет *элегантность* как «утонченное изящество; корректность; искусная простота; изысканность...»

  Возникающие при использовании трюков проблемы можно проиллюстрировать на следующем «модельном» трюке:

```
10 A=A+B:B=A-B:A=A-B  .
```

  Здесь две переменные меняются значениями без промежуточного копирования значения одной из переменной. Этот приём используется системными программистами в том случае, когда при перемене мест содержимого регистров важно сэкономить дополнительный регистр.

  Оригинальность очевидна, но *элегантным* этот приём можно считать, лишь принимая во внимание особенности конкретной задачи. Он, очевидно, не эффективен, когда используется вне рассматриваемого контекста, например для перестановки элементов многомерного массива в обычной программе, и к тому же может привести к ошибке округления, если A и B описаны как вещественные числа. Например:

```
10 INPUT A,B:A=A-B:B=A+B:A=B-A:PRINT A;B
run
? 1.5E15,1
 0  1.5E+15    ◄── Результат ужасен!
Ok
```

  Итак, никогда не используйте трюки только для того, чтобы продемонстрировать свои умственные способности!

- V. Наконец, обсудим и *организационные* приёмы.

  Почти непременным элементом методологии программирования является принцип *бригадной* организации работ в

программировании. Практическая реализация больших программных проектов требует умения и опыта многих программистов.

Почему в программировании необходима бригадная организация работ? Ответ прост: **задача может потребовать бригадной организации её решения, потому что она слишком трудна или слишком велика, или слишком разнородна**.

В 1971г. Г.Миллз предложил схему организации программистской деятельности, известную как *бригада главного программиста*. Этот подход успешно использовался при разработке и реализации нескольких крупных программных проектов.

По словам Миллза, «бригада главного программиста — это небольшой коллектив сотрудников, созданный для эффективной работы и рассматриваемый как единое целое». Бригада состоит из нескольких человек, обычно от трёх до пяти; среди них — главный программист, резервный программист, секретарь бригады и по мере необходимости другие специалисты.

Основная идея бригады заключается в том, что она работает как суперпрограммист, т.е. как один программист с очень высокой производительностью труда, что обеспечивается участием в работе всех членов бригады, действующих (благодаря внутренним связям в бригаде) с полным единомыслием.

Главный программист — это компетентный программист, доказавший свои незаурядные технические способности. Он руководит бригадой, ему непосредственно подчиняются все остальные её члены.

В обязанности главного программиста входит проектирование программы, написание самых важных её модулей и определение остальных модулей, которые программируют другие члены его бригады. Вся работа бригады находится под постоянным контролем главного программиста; он объединяет результаты в одно целое.

Резервный программист, работающий в непосредственном контакте с главным программистом и полностью посвящённый во все его решения, может в случае необходимости возглавить бригаду. Обычно резервный программист отвечает за независимую подготовку тестов для разрабатываемой программы; он может также помогать главному программисту в исследовательской деятельности, изучая альтернативные стратегии и тактики бригады.

Основная задача *секретаря* бригады — это документационное обеспечение проекта как в машинно–ориентированной форме, так и в виде, доступном для человека. Секретарь отражает текущее состояние проекта и его историю.

Каждый член бригады обязан тщательно регистрировать все те свои действия, которые изменяют положение дел в проекте, и передавать эти записи секретарю.

Подчеркнём, — основная функция секретаря заключается не столько в том, чтобы избавить программистов от бумажной работы, сколько в том, чтобы обеспечивать наглядную информацию о положении дел в проекте и о достигнутых успехах. Достоинство такой организации — в наличии источника единообразно представленной и свежей информации о ходе разработки программы.

При реализации большого проекта одной бригады главного программиста может быть недостаточно, как недостаточно одного программиста для решения многих программистских задач. Миллз предлагает организовать в таком случае иерархию бригад главного программиста, начиная с одной бригады на самом высшем уровне. Бригады следующих уровней создаются лишь после того, как бригада предыдущего (более высокого) уровня подготовила им задание. В противоположность ориентированной на управление иерархии в классической организации программистских коллективов здесь не существует разделения функций на высших уровнях иерархии: бригады главного программиста на высших уровнях проходят различные этапы и отвечают за различные виды деятельности (проектирование, кодирование, тестирование, проверка), на каждом этапе устанавливая конкретные задания для подчинённых групп.

Бригада высшего уровня завершает проектирование (в самых абстрактных терминах), кодирование (в этих же терминах) и тестирование (тоже в этих терминах!) на самых ранних этапах разработки проекта. И только когда эта бригада успешно прошла все тесты (больше похожие на доказательства), можно безбоязненно передавать задания бригадам низших уровней.Все остальное время выполнения проекта бригада высшего уровня посвящает верификации (проверке) результатов, поступающих с нижних уровней.

Однако учтите, что *пассивные* методы, способствуя значительному повышению качества программ, не могут гарантировать удовлетворения всех заданных требований к программам, а главное, не полностью предотвращают ошибки.

Поэтому *активные* методы поиска и устранения ошибок дополняют пассивные в процессе достижения заданного качества программ.

VIII.6. О стиле программирования [57]

[57]

Форме дай щедрую дань
Временем: важен в поэме
Стиль, отвечающий теме.
Стих, как монету, чекань
Строго, отчётливо, честно,
Правилу следуй упорно:
Чтобы словам было тесно,
Мыслям просторно.

—Н. Некрасов

Трудно дать исчерпывающее определение понятия »*стиль* программирования«. Попытаемся охарактеризовать его, рассмотрев с различных точек зрения. Главный тезис состоит в том, что *стиль программирования — это стиль мышления*, проявляющийся в умении переводить алгоритм решения задачи на конкретный язык программирования.

Не существует совокупности правил написания программ, следуя которым Вы могли бы создавать качественные и не содержащие ошибок программы. Стиль программирования не сводится к хорошему знанию конкретного языка программирования, знанию его возможностей и правил записи программ, хотя он все это и предполагает. Скорее эти знания помогут отличить программу, написанную в хорошем стиле от программы, написанной в плохом стиле. Многие достаточно хорошо знают какой–либо иностранный язык, чтобы без особых затруднений читать специальную или художественную литературу на этом языке. Но лишь немногие могут с такой же лёгкостью написать на иностранном языке даже небольшую статью. Таким образом, от знания языка до владения языком лежит «дистанция огромного размера». Но даже овладев языком программирования в совершенстве, мы лишь незначительно приблизимся к грамотному стилю программирования.

К сожалению, взгляд на стиль программирования как на стиль мышления ещё не всеми и не до конца осознан. Очень часто стиль программирования сводится к технологии программирования. За последние годы мы пережили несколько вспышек увлечения модными методологиями и технологиями программирования, такими например, как структурное программирование. Но в главном все они ориентируются на внешние факторы, характеризующие программу. Кстати сказать, жертвой структурного программирования чуть было не стали Фортран и BASIC с их неструктурными операторами. Но очевидно, что плохо разработанная программа, записанная по правилам структурного программирования, так и останется плохой программой. В этом плане переход от языка BASIC к таким языкам структурного программирования, как Паскаль и Ада, мало что даст.

Подводя итог сказанному, приведём некоторые рекомендации, которые, как мы надеемся, при осознанном и неформальном их использовании помогут вам выработать свой стиль программирования. Большинство рекомендаций имеют достаточно общий характер и могут быть использованы при программировании не только на BASIC, но и на многих других языках программирования. Эти рекомендации не являются исчерпывающими. Много подобных советов читатель может почерпнуть из книг по разработке программ и структурному программированию [52], [53], [54], [55].

1. **В основе алгоритма решения задачи лежит *математическая модель*. Не нужно жалеть времени на разработку и изучение свойств этой модели!**

   Это поможет лучше понять задачу и найти наиболее естественный путь её решения.

2. Максимально используйте «изобразительные» возможности языка программирования. В частности, **старайтесь располагать только *один* оператор на строке**.

   Размещение нескольких операторов на одной физической строке противоречит правилу структурного программирования, требующему сдвигать оператор по строке в соответствии с уровнем его вложенности.

3. Относитесь с должным вниманием и аккуратностью к написанию даже очень простых частей программы. Помните, что программа — это единый организм, работоспособность которого определяется качеством всех её частей, а не каких–то отдельных компонентов. Просчёт в каком–то одном месте может привести к неудаче в целом.

4. Не пытайтесь сразу написать эффективную программу. Это может привести к противоположному результату. Помните о следующем эмпирическом факте: 75% времени выполнения программы приходится на 25% её текста. Лишь тестирование первого варианта Вашей программы сможет выявить эти 25%, которые действительно требуют тщательной проработки.

Экономия на мелочах нередко приводит к проигрышу в целом.

5. **Помните об отладке с самого начала создания программы**.

   Для этого активно используйте отладочную печать, вставляя соответствующие операторы ещё при написании программы. Во всех местах, где происходит ветвление процесса вычислений, необходимо распечатывать данные, определяющие выбор варианта. В местах слияния ветвей решения следует печатать маркер этого места и информацию о том, по какой из ветвей прошло решение.

6. Не забывайте о надлежащей организации операций ввода–вывода данных. Не жалейте усилий на разработку средств, обеспечивающих наглядность представления вводимых и выводимых данных. Все вводимые данные должны распечатываться и проверяться на корректность.

   Печать данных так же, как и представление вводимых данных, должна быть содержательной, отражающей структуру данных и их интерпретацию, а не быть хаотичной грудой цифр. Используйте такие формы отображения, как таблицы и графики.

7. Если Вы хотите достигнуть определённых высот в программировании, то процесс создания программы не должен заканчиваться для Вас получением работающей программы.

   Не спешите расстаться с вашим детищем, принёсшим вам столько хлопот и мучений. Ещё раз внимательно просмотрите программу. Постарайтесь извлечь максимум пользы на будущее. Оцените идеи и методы, использованные Вами с точки зрения их применимости в других ситуациях и пытайтесь выработать шаблоны, обобщающие эти идеи и методы.

   По мере того, как число таких шаблонов будет увеличиваться,будет расти и Ваше мастерство, умение оценивать все особенности конкретной задачи и пользоваться наиболее адекватными средствами для их отражения. Сказанное не следует понимать как призыв к шаблонному мышлению в программировании. Скорее наоборот, применение того или иного шаблона требует глубокого предварительного анализа имеющейся ситуации с целью определения, какой именно шаблон можно применить в данной ситуации. И чем больше шаблонов находится в Вашем распоряжении, тем более детальным должен быть этот анализ. Результатом такого анализа может быть рождение новой оригинальной идеи или метода. Таким образом, использование шаблонов освободит Вас от «изобретения велосипеда» в тех ситуациях, когда можно добраться до цели в роскошной гоночной машине, и позволит сосредоточить основное внимание на решении новых для Вас проблем.

8. Помните, что наилучшей документацией для любой программы является сама эта программа. Поэтому программа должна содержать в явном виде исчерпывающую информацию, представленную в виде комментариев.

9. Чтобы снизить погрешность результатов при выполнении вычислений с действительными числами, следует:

   - избегать вычитания близких чисел,
   - избегать деления больших по модулю чисел на малые числа, особенно если последние имеют невысокую точность,
   - сложение (вычитание) длинной последовательности чисел начинать с меньших чисел,
   - стремиться уменьшить число операций,
   - использовать алгоритмы, для которых известны оценки ошибок,
   - не использовать сравнение на равенство действительных чисел в операторе `IF`.

10. Программисты в своих программах не всегда явно присваивают начальные значения используемым переменным, полагаясь на то, что интерпретатор присвоит им вполне определённые значения. Это может привести к неверной работе программы. Чтобы избежать такой ошибки, следует явным образом присваивать начальные значения всем используемым в программе переменным. Такие «сокращения» предназначены для дилетантов или тех, кто программирует от случая к случаю, но не программистов–профессионалов.

    Профессиональный программист должен явно определять или объявлять все переменные в самом начале программы или подпрограммы.

11. При использовании в записи идентификатора цифр помещайте их в конце идентификатора, так как цифры 0, 1, 2, 3, 4, 5 в середине идентификатора можно воспринять как буквы O, I, Z, З, Ч, S.

    Никогда не используйте переменную более чем для одной цели. Распространённый приём экономии лишнего слова памяти или лишнего оператора для описания типа состоит в повторном использовании переменной в различных условиях. Программист вполне может решить: «Я закончил работать с TIME для расчётов времени, поэтому теперь буду использовать эту переменную как промежуточную при вычислении даты». Такая практика увеличивает шансы внесения ошибки при модификации программы.

12. В тексте программы старайтесь явно не употреблять константы, за исключением, быть может, нуля и единицы. Для констант лучше вводить имена, которые и использовать в программе. Это приводит к лучшей «читабельности» программы и к уменьшению числа возможных ошибок.

13. Избегайте употреблять арифметические выражения, содержащие переменные разных типов. Пользуйтесь функциями `CINT()` и `CSNG()` для преобразования значений переменных одного типа в значения другого типа данных.

14. Предусматривайте «фразу» `ELSE` для каждой «фразы» `THEN`. В операторе условного перехода должно быть *поровну* «фраз» `THEN` и `ELSE`. Даже если не нужно ничего делать в случае «фразы» `ELSE`, следует предусмотреть пустой оператор. Это подскажет читателю, что случай «пустого» `ELSE` также рассматривается, и поможет понять последовательность действий.

15. Не пишите изменяющих себя программ.

16. Изучите и активно используйте возможности языка программирования: в результате программы становятся короче и исключаются определённые типы ошибок.

    Внимательно прочтите раздел о «подводных камнях» в руководстве по языку программирования, на котором Вы программируете.
    Экономьте время, учитесь на ошибках других!

17. Другими факторами, оказывающими влияние на организацию программ, являются ограничения на память и время выполнения. Учёт этих факторов может вступать в противоречие с требованиями методов структурного программирования при разработке программ.

    Если ограничения на память существенны, возможно, придётся пожертвовать наглядностью программы и комментариями для экономии памяти. Конечно, программу можно разделить так, чтобы результат получался поэтапно с помощью не одной, а нескольких программ. Можно также уменьшить размеры программы, ограничивая использование повторяющихся операторов или сегментов программы. Для этого может потребоваться объединить модули или сегменты программы, что в определённой степени нарушит модульную структуру программы. Кодирование нескольких операторов в одной строке, удаление необязательных пробелов в операторах, а также исключение операторов `REM` — все эти приёмы позволяют сэкономить память, но сделают программу более трудной для понимания, отладки и модернизации.

    Объем вычислений можно уменьшить различными методами. Прежде всего необходимо исключить лишние вычисления. Следует убедиться, что одно и то же значение не вычисляется в программе многократно. Вот что иногда можно увидеть в программе:

```
10 FOR I=1 TO N:FOR J=1 TO 1000
20    K=I+3
        …
80 NEXT J:NEXT I
```

    В этой программе выражение K=I+3 вычисляется 1000 раз для каждого из N значений параметра цикла I. Чтобы устранить ошибку, оператор 20 следует записать перед оператором `FOR J=1 TO 1000` (произвести *«чистку»* цикла).

    На вычисление тригонометрических и экспоненциальных функций затрачивается много машинного времени. Довольно часто мы можем пересмотреть алгоритм и обойтись без этих функций. Вообще сведение всех вычислений только к операциям сложения и вычитания *целых* чисел существенно увеличивает скорость выполнения программы.

    Уменьшение размерности массивов или их исключение также дают выигрыш во времени (например, для движущихся изображений это означает сокращение числа перемещаемых элементов рисунка и увеличение шага каждого перемещения), поскольку адрес элемента двухмерного массива вычисляется дольше, чем одномерного. Кроме того, время доступа к элементу одномерного массива больше, чем к простой переменной.

    Однако основной метод уменьшения времени выполнения состоит в кодировании программы на языке ассемблера или на машинном языке!

Тем не менее игнорируйте все предложения по повышению эффективности, пока программа не будет правильной. Худшее, что может быть сделано, — это начать беспокоиться о быстродействии программы до того, как она начнёт работать правильно. Быстродействующая, но неправильная программа бесполезна; медленнодействующая, но правильная всегда имеет некоторую ценность, а может оказаться и вполне удовлетворительной.

1. Если программа неверна, то её быстродействие не имеет значения. Убедитесь в её правильности прежде, чем Вы начнёте её «улучшать».

2. Сохраняйте программу ясной и понятной, не пытайтесь повысить её быстродействие в процессе кодирования. Преждевременная оптимизация — корень всех бед.
3. Не стремитесь к оптимизации каждого простого вычисления. Пусть транслятор делает это за Вас.
4. Беспокойтесь об алгоритме, а не о деталях программы. Помните, что структура данных может существенно повлиять на то, как будет реализован алгоритм...

—Б.Керниган, Ф.Плоджер

Вейнберг [68] рассказывает забавную историю о новой программе, которая из–за слишком большой сложности оказалась совершенно ненадёжной. Был вызван новый программист, который нашёл лучшее решение и за две недели сделал новую, надёжную версию программы. При демонстрации её работы он отметил, что его программе требуется 10 секунд на каждую перфокарту. Один из разработчиков первоначального варианта, торжествуя, заявил: «А моей программе требуется только одна секунда на перфокарту». Ответ программиста стал классическим:

«Но Ваша программа не работает. Если программа не должна работать, я могу написать такую, которой хватает одной миллисекунды на перфокарту».

VIII.7. Недостатки языка программирования BASIC [59]

[59]

Меня вы научили говорить
На вашем языке. Теперь я знаю,
Как проклинать, - спасибо и за это.
Пусть унесет чума обоих - вас
И ваш язык.

—У.Шекспир

Взирая на солнце,прищурь глаза свои
и ты смело разглядишь в нем пятна.

—Козьма Прутков

Школьный курс информатики преследует две цели:

1. *Общетеоретическая цель*.

   Овладение алгоритмическим стилем мышления, который включает в себя умение чётко сформулировать заданные условия и требуемые результаты, разбиение большой задачи на малые, поиск решения в виде последовательности действий, выбор которых может зависеть от конкретных условий. Алгоритмическое мышление в таком понимании применимо (и необходимо!) практически в любой сфере человеческой деятельности.

2. *Практическая цель*.

   Понимание принципиальных возможностей современных компьютеров, представление о круге решаемых ими задач. Умение выделить «человеческую» и «компьютерную» части конкретной задачи, построение модели и формализация «компьютерной» части, доведение её до уровня программы для ЭВМ.

Заметим, что программирование занимает и здесь подчинённое место. Главное — понять, что можно, а чего нельзя запрограммировать, и формализовать выбранную для программирования задачу.

Итак, цели поставлены, теперь надо выбрать соответствующие им средства. И тут программирование заслуженно занимает ведущее место. Именно в нем наиболее полно проявляются алгоритмические закономерности, поэтому программирование становится *основным* средством овладения алгоритмическим мышлением.

Итак, алгоритмическое мышление — цель, программирование — средство!

Возникает вопрос: на каком языке записывать составляемые в процессе обучения алгоритмы и программы? Исходя из поставленных целей, определим требования к языку программирования.

1. Язык должен быть *простым*. Главная трудность должна заключаться в составлении алгоритма, а не в его записи (это требование немедленно исключает такие языки, как Лисп, Ассемблер, Си и язык программируемого микрокалькулятора).
2. Язык должен допускать естественное представление основных алгоритмических конструкций (отпали Фортран и BASIC).
3. Язык должен быть *универсальным* т.е. допускать естественную обработку величин различных типов (Алгол и Фортран не содержат достаточно развитых средств работы со строковыми (литерными) величинами).

Не кажутся обязательными такие требования, как распространённость языка или наличие эффективной реализации. Скорее, наоборот, выбранный для массового обучения язык имеет хорошие шансы стать широко распространённым.

Таким образом, существующие языки программирования не пригодны для массового обучения. Для решения этой проблемы были созданы специальные языки: Робик и Рапира Г.А. Звенигородского, алгоритмический язык А.П. Ершова. По своим идеям,да и по внешней форме эти языки достаточно близки, главное их отличие заключается в трактовке понятия типа величины: в алгоритмическом языке тип связан с именем, в Рапире — со значением.

Отметим особенности школьного алгоритмического языка, которые делают его особенно ценным для использования при обучении.

1. Это прежде всего *русская* нотация.
2. Свободный синтаксис записи выражений, который даёт ученику возможность сосредоточиться не на форме записи алгоритма, а на его сути.
3. Алгоритмический язык обладает основными фундаментальными конструкциями, присущими структурному программированию:
    - командой условного перехода «если ..., то ... , иначе ... всё»,
    - циклами «для» и «пока», командой «выбор» .
4. Нет конструкции «идти к...» (аналог оператора GOTO) (впрочем, иногда это создаёт определённые трудности при составлении алгоритма!).

Поэтому

главной задачей, стоящей при изучении языка программирования в школе, является — показать, как фундаментальные алгоритмические конструкции (конструкции школьного алгоритмического языка) приобретают свойственную конкретному языку программирования форму, сохраняя своё содержание.

Программа преподавания информатики предоставляет учителю свободу выбора языка программирования, но большинство останавливается на BASIC и Рапире.

Именно поэтому мы поговорим о недостатках языка программирования BASIC.

1. В MSX BASIC отсутствует естественное представление алгоритмической конструкции — цикл «пока». Приведённые в школьном учебнике операторы WHILE–WEND относятся к весьма редкой версии BASIC, не встречающейся на имеющихся сегодня (1990г.) компьютерах.
2. Нет аналога вспомогательным алгоритмам. Конструкция подпрограмм не имеет таких важнейших свойств вспомогательных алгоритмов, как независимость имён, передача параметров и возможность организации библиотек. Таким образом, эта конструкция не является эффективным средством разбиения большой задачи на малые и сведения нерешённой задачи к решённым (важнейшие навыки алгоритмизации!). Других средств для этого BASIC не предлагает.

3. Выбор имён в подавляющем большинстве версий BASIC (MSX BASIC, конечно, не в счёт!) ограничен формулой «буква + цифра». Использование содержательных мнемонических имён запрещено, что очень затрудняет разработку и понимание программ, выходящих за рамки примитива.

Таким образом, BASIC может быть инструментом программирования (хотя и очень несовершенным), но он не даёт адекватных средств для развития мышления. В то же время из–за отсутствия естественного представления основных конструкций его очень неудобно использовать в качестве иллюстрации.

Особенно опасно изучение BASIC в компьютерном курсе. Слабые ученики плохо понимают язык, путаются в многочисленных ограничениях при программировании алгоритмических конструкций, теряют интерес к предмету и к компьютеру. На программирование у них уходит столько сил, что на мышление уже не остаётся. Сильные же ученики, видя, что компьютер прекрасно понимает их без «алгоритмических излишеств» типа заголовка алгоритма и продуманной структуры программы, пренебрегают этими деталями, не хотят писать на школьном алгоритмическом языке задачи, которые можно сразу сделать на BASIC. В итоге они получают навык программирования, а не мышления, и главная цель остаётся не достигнутой.

Почему же, несмотря на столь серьёзную опасность BASIC, он так популярен?

Перечислим основные аргументы, выдвигаемые его защитниками, и попытаемся их прокомментировать.

- BASIC прост в изучении. Это единственное достоинство языка как такового. Но это та самая простота, которая хуже воровства, «простота орудий каменного века». BASIC прост потому, что в нем нет сложных элементов (в первую очередь вспомогательных алгоритмов и некоторых алгоритмических конструкций), без овладения которыми поставленные цели не достигаются.
- При работе на BASIC диалог с компьютером происходит быстро и просто.
  Это достоинство не языка, а его реализации.
- BASIC имеется на многих компьютерах, он широко распространён. Однако это не достоинство, а большая беда. Фактически мы сталкиваемся здесь с осуждённым в промышленности диктатом производителя. Изготовители ЭВМ и программного обеспечения рекламируют BASIC, так как BASIC–системы проще в изготовлении и могут быть перенесены с западных компьютеров.

Итак, необходимо, чтобы поставляемые в школы компьютеры имели систему программирования не на BASIC, а на другом языке программирования высокого уровня!

VIII.8. Сравнительная характеристика языков программирования

> Карл Пятый, римский император, говаривал, что ишпанским языком с богом, французским — с друзьями, немецким — с неприятельми, италиянским — с женским полом говорить прилично. Но если бы он российскому языку был искусен, то, конечно, к тому присовокупил бы, что им со всеми оными говорить пристойно. Ибо нашёл бы в нем великолепие ишпанского, живость французского, крепость немецкого, нежность италиянского и, сверх того, богатство и сильную в изображениях краткость греческого и латинского языка...
>
> Сильное красноречие Цицероново, великолепная Виргилиева важность, Овидиево приятное витийство не теряют своего достоинства на российском языке...
>
> *—М.Ломоносов. Российская грамматика*

Идеальный язык программирования имеет следующие свойства:

- 1) он лёгок для начинающих;

- 2) он лёгок для специалистов;
- 3) он позволяет легко читать программы;
- 4) он дисциплинирует пользователей и таким образом заставляет их следовать хорошему стилю программирования;
- 5) он подходит для написания коротких программ;
- 6) он подходит для написания программ среднего размера;
- 7) он подходит для написания больших программ, содержащих десятки или сотни тысяч строк;
- 8) он подходит для доказательства правильности программ;
- 9) он лёгок для реализации на любом компьютере;
- 10) он позволяет эффективно (т.е. быстро) выполнять программы.

Невозможно разработать язык программирования, который удовлетворял бы всем этим критериям

(тем не менее люди пытаются это делать, и новые языки программирования рождаются ежедневно). На практике разработчики удачных языков сосредоточиваются на нескольких критериях.

Разработчики языка BASIC выделяют в качестве наиболее важных критериев 1 и 5.

Разработчики языка Паскаль оценивают свойство 4 как наиболее важное, и считает важными также позиции 3 и 10.

Разработчики языка Ада охватили свойство 7, и это заставило их пойти на компромиссы со многими другими желательными критериями.

Разработчики языка Си выделяют критерии 10, 6 и 2 как наиболее важные, возможно, в этой последовательности.

В таблице, составленной А.К.Поляковым и В.К.Раковым в пособии для студентов МЭИ «Программное обеспечение микропроцессорных систем» оценивается в пятибалльной системе пригодность различных языков для программирования встроенных микропроцессорных систем.

	Ассемблер	**PL/M**	**Паскаль**	**Фортран**	**BASIC**	**Ада**	**Си**
Простота изучения	3	3	4	4	5	1	3
Наличие литературы	5	3	3	4	4	2	1
Поддержка фирмами–разработчиками микропроцессорных систем	5	4	4	4	4	3	4
Универсальность (широта диапазона решаемых задач)	3	3	3	3	2	5	3
Удобство для системного программирования	5	5	4	3	1	4	5
Удобство для прикладного программирования	2	3	4	4	2	4	5
Возможность компоновки программ из независимо отлаженных модулей	1	4	2	4	1	5	4
Структурность	0	4	5	3	2	5	3
Эффективность средств реального времени	3	3	0	0	0	3	0
Развитость встроенных типов данных	1	2	4	4	2	5	4
Возможность введения новых типов данных	1	0	4	0	0	5	0
Возможность параллельного программирования	0	0	0	0	0	4	0
Простота транслятора	5	3	3	3	4	1	3
Эффективность программ	5	4	4	3	1	3	4
Эффективность средств отладки	3	3	3	3	5	3	3

В заключение три цитаты:

1. Практически невозможно научить хорошему программированию студентов, ориентированных первоначально на бейсик; как потенциальные программисты они умственно оболванены без надежды на исцеление

   — Э.Дейкстра, ASMSIGPLAN Notice, 1982, 7, P.13–15

2. ...коль уж становиться программистом, то программистом хорошим; такого программиста отличает постоянное желание стать ещё лучшим программистом, а единственно верный путь для этого — стремиться в совершенстве овладеть несколькими языками, т.е. сделаться хорошим лингвистом в программировании. Безусловно, можно доказать, что несомненный вред нанесли и наносят те, довольно хорошие программисты, которые, став слишком самодовольными или консервативными, полагают, что язык, которым они пользуются, во всех смыслах является последним словом

   — Б.Мик

3. **Предмет, достойный целого курса лекций, вы превратили в собрание *рассказиков*.**

   — А.Конан Дойль. Медные буки

Диск с примерами

Загрузить образ диска

 Открыть диск в WebMSX

---

http://sysadminmosaic.ru/msx/basic_dialogue_programming_language/008

2023-02-19 16:21