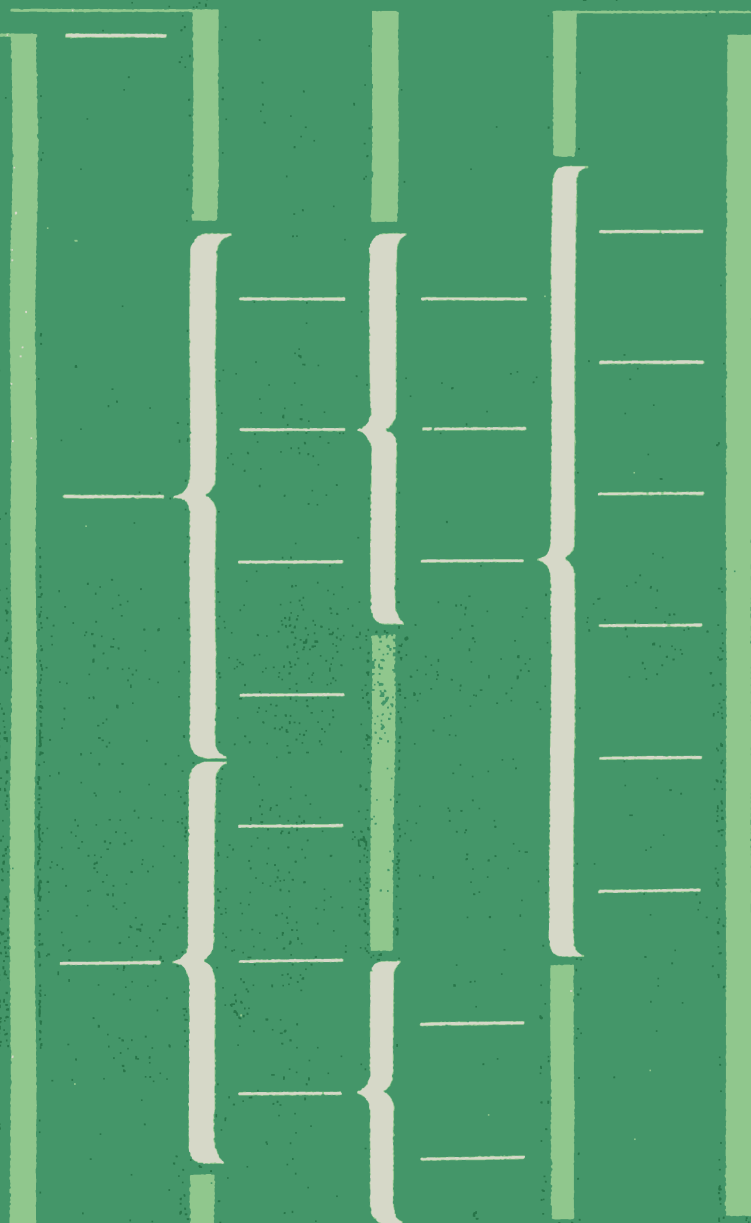


Дж.Хьюз. Дж.Мичтом

# СТРУКТУРНЫЙ ПОДХОД К ПРОГРАММИРОВАНИЮ



**Дж.Хьюз, Дж.Мичом**  
**СТРУКТУРНЫЙ ПОДХОД**  
**К ПРОГРАММИРОВАНИЮ**

---

Перевод с английского  
Э. М. КИУРУ  
и А. Л. АЛЕКСАНДРОВА  
под редакцией  
В. Ш. КАУФМАНА

ИЗДАТЕЛЬСТВО «МИР»  
МОСКВА 1980

Книга посвящена вопросам структурного подхода к программированию — новому направлению в программировании. Она содержит множество примеров, иллюстраций и рекомендаций руководителям программистских коллективов. Особую ценность имеют конкретные данные по структурному программированию на КОБОЛе, ФОРТРАНе и ПЛ/I, трансляторы с которых имеются на отечественных машинах ЕС ЭВМ.

Книга предназначена программистам и их руководителям. Она будет полезна студентам и аспирантам вычислительных специальностей университетов и институтов.

*Редакция литературы по математическим наукам*

2405000000

X  $\frac{20204-036}{041(01)-80}$  36-80

© 1977 Prentice-Hall, Inc.

© Перевод на русский язык, «Мир», 1980

## ОГЛАВЛЕНИЕ

Предисловие редактора перевода . . . . .	5
Предисловие . . . . .	7
Глава 1. НОВАЯ ДИСЦИПЛИНА ПРОГРАММИРОВАНИЯ . . . . .	9
Пролог . . . . .	9
Что такое структурное программирование? . . . . .	11
Зачем нужна эта новая дисциплина? . . . . .	12
Что такое структурный подход? . . . . .	14
Выводы для руководства . . . . .	24
Эпилог . . . . .	25
Выводы . . . . .	27
Контрольные вопросы и упражнения . . . . .	28
Глава 2. НИСХОДЯЩАЯ РАЗРАБОТКА: ПРОЕКТИРОВАНИЕ ПРОГРАММЫ . . . . .	29
Модульность . . . . .	29
Нисходящее проектирование программ . . . . .	34
Выводы для руководства . . . . .	49
Выводы . . . . .	49
Контрольные вопросы и упражнения . . . . .	50
Глава 3. НИСХОДЯЩАЯ РАЗРАБОТКА: ПЛАНИРОВАНИЕ И РЕАЛИЗАЦИЯ . . . . .	52
Планирование . . . . .	52
Реализация . . . . .	63
Выводы для руководства . . . . .	66
Выводы . . . . .	68
Контрольные вопросы и упражнения . . . . .	70
Глава 4. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ . . . . .	72
Управляющие структуры . . . . .	73
Псевдокод . . . . .	82
Выводы для руководства . . . . .	89
Контрольные вопросы и упражнения . . . . .	92
Глава 5. ПОШАГОВАЯ ДЕТАЛИЗАЦИЯ . . . . .	94
Блок-схемы . . . . .	98
Правила детализации . . . . .	99
Сегментирование . . . . .	115
Выводы . . . . .	124
Контрольные вопросы и упражнения . . . . .	128

Глава 6. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ НА СТАНДАРТ-НОМ КОБОЛе . . . . .	130
Модульность . . . . .	131
Структурирование программ на КОБОЛе . . . . .	141
О стиле программирования . . . . .	156
Как облегчить чтение программы . . . . .	163
Выводы . . . . .	168
Контрольные вопросы и упражнения . . . . .	177
Глава 7. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ НА ФОРТРАНе . . . . .	178
Организация программ на языке ФОРТРАН . . . . .	179
Структурирование программ на ФОРТРАНе . . . . .	182
О стиле программирования . . . . .	189
Как облегчить чтение программы . . . . .	192
Использование препроцессора с выходом на ФОРТРАН . . . . .	194
Выводы . . . . .	203
Контрольные вопросы и упражнения . . . . .	204
Глава 8. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ НА ПЛ/I . . . . .	206
Модульность . . . . .	206
Структурирование ПЛ/I-программ . . . . .	216
Некоторые приемы программирования на ПЛ/I . . . . .	226
Как облегчить чтение программ . . . . .	228
Пример программы . . . . .	231
Выводы . . . . .	233
Контрольные вопросы и упражнения . . . . .	240
Глава 9. СКВОЗНОЙ СТРУКТУРНЫЙ КОНТРОЛЬ . . . . .	241
Кто этим занимается? . . . . .	243
Атмосфера на контрольной сессии . . . . .	245
Как это все происходит? . . . . .	246
Выводы для руководства . . . . .	249
Выводы . . . . .	252
Контрольные вопросы . . . . .	252
Глава 10. ТЕСТИРОВАНИЕ . . . . .	254
Что и как нужно тестировать? . . . . .	255
Разработка плана тестирования . . . . .	263
Прогон тестов . . . . .	265
Выводы для руководства . . . . .	265
Выводы . . . . .	267
Контрольные вопросы и упражнения . . . . .	268
Глоссарий . . . . .	269
Список литературы . . . . .	274

## ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Традиционная технология программирования складывалась в условиях, когда основным потребителем программ были научные учреждения, вычислительные ресурсы были ограничены, а проблемы сопровождения программ были по существу неизвестны. Основным критерием качества создаваемой программы считалась ее узко понимаемая эффективность, и мало кого интересовали отрицательные последствия тех ухищрений, которые применялись для ее достижения.

К настоящему времени ситуация коренным образом изменилась. Потребителем программ стало по существу все народное хозяйство, и узким местом стали не столько вычислительные, сколько человеческие ресурсы, необходимые для создания и сопровождения крупных систем программ. Стало ясно, что разумная технология программирования должна обеспечить снижение общих трудозатрат с учетом всего жизненного цикла программы — от замысла до завершения эксплуатации. Она должна учитывать, что сопровождение крупных комплексов программ стоит намного дороже их создания, да и в процессе создания приходится не столько писать программы, сколько читать их. В этих условиях важнейшими критериями качества становятся понятность, надежность и удобство сопровождения программ, а также возможность точно планировать их производство.

Неудовлетворенность традиционной технологией и постепенное осознание новых критериев заставило крупнейших производителей программ интенсивно искать новые технологические принципы. Наиболее впечатляющих успехов в этом деле добилась корпорация ИБМ.

Когда в начале семидесятых годов появились первые сообщения о том, что существует технология программирования, позволяющая создавать достаточно крупные программы практически без ошибок и при этом укладываться в запланированные сроки и бюджет, эти сообщения были восприняты с большим недоверием. Ведь большинство программистов привыкли к тому, что отладка, а тем более комплексная отладка — весьма капризный и по существу непредсказуемый процесс, обычно нарушающий все разумные оценки по срокам и затратам,

И тем не менее сообщения оказались в целом верными — речь шла о первых применениях разработанных фирмой ИБМ так называемых Усовершенствованных методов программирования. Они оказали революционизирующее влияние практически на все стороны деятельности по созданию программ, соединив в себе лучшие элементы личной технологии классных программистов и достижения теории программирования с индустриальным характером современного производства программ при тщательном учете изменившихся требований к ним,

Перед вами учебное пособие по одной из важнейших компонент Усовершенствованных методов — структурному подходу к программированию (сущностью Усовершенствованных методов можно познакомиться по книге Ф. Я. Дзержинского и А. И. Тер-Саакова <sup>1)</sup>). Авторы не претендуют на изложение теоретических основ структурного подхода. Их цель — научить рядового программиста пользоваться этим подходом в повседневной практической деятельности. Особую ценность придают книге конкретные рекомендации по структурному программированию на КОБОЛе, ФОРТРАНе и ПЛ/1, трансляторы с которых имеются на отечественных машинах, а также по организации нисходящей разработки, пошаговой детализации, сквозному структурному контролю и тестированию. Правда, иногда авторы слишком увлекаются упрощением, не сообщая читателю о возможных трудностях при попытках использовать их рекомендации «в лоб». Это касается, в частности, нисходящей реализации и выделения наборов тестов. Существенно фундаментальнее аналогичные вопросы изложены в монографии Г. Майерса <sup>2)</sup>. Для советского читателя может оказаться трудным разобраться досконально в некоторых примерах из американской экономики, тем более, что авторы рассматривают лишь фрагменты, а не полные программы. Пугаться этого не следует — как правило, для понимания существа излагаемой технологии достаточно общего представления о назначении рассматриваемого фрагмента.

Книга Дж. Хьюз и Дж. Минчота написана на основе богатого опыта, накопленного ИБМ по внедрению и использованию структурного подхода, она будет полезна как программистам, желающим улучшить методы своей работы, так и руководителям программистских коллективов.

Предисловие авторов и главы 1, 5, 8, 9, 10 переведены Э. М. Киуру, главы 2, 3, 4, 6, 7 — А. Л. Александровым. При переводе экономических терминов большую помощь оказал С. А. Овсейчик.

*В. Ш. Кауфман*

---

<sup>1)</sup> Дзержинский Ф. Я., Тер-Сааков А. И. Технология программирования — структурный подход. — М.: ЦНИАтоминформ, 1978.

<sup>2)</sup> Майерс Г. Надежность программного обеспечения. — М.: Мир, 1980.

## ПРЕДИСЛОВИЕ

Одни называют появление *структурного программирования* революцией, другие говорят, что это дело будущего. Некоторым оно кажется модой, причудой, которая пройдет со временем. Как бы то ни было, ясно одно: после изучения идей структурного программирования и соответствующих приемов вы, несомненно, измените свой стиль решения задач с помощью ЭВМ. В этой книге понятие структурирования распространяется с собственно программы на весь процесс программирования.

Заглавие *Структурный подход к программированию* выбрано с целью подчеркнуть необходимость общего подхода к программированию — единого метода, который требует определенной дисциплины на всех стадиях программистского проекта: спецификации, проектирования, собственно программирования (кодирования) и тестирования. Можно надеяться, что при более формализованном способе получения программ программирование станет больше походить на науку.

Цель настоящей книги — изложить основные идеи и эффективные методы, присущие структурному подходу. Для программиста она содержит достаточно информации, чтобы он мог сам применять предлагаемые методы на практике. Администратору же знакомство с книгой позволит уверенно управлять теми, кто использует структурные методы, или контролировать их деятельность.

Книга предназначена для тех, кто знаком с основными понятиями программирования. Так как это не общее введение в программирование и не введение в программирование на конкретном языке, то было бы разумным использовать книгу в курсе, которому предшествуют один или несколько вводных курсов программирования. Профессионалы (любого уровня) в области обработки данных также извлекут пользу от применения излагаемой методики. Программисты и разработчики могли бы использовать новые идеи, излагаемые в книге, в своей повседневной работе. Руководители проектов и администраторы, усвоив излагаемый в книге подход, могли бы способствовать его внедрению, а также контролировать тех, кто его применяет.

Доскональное знакомство с программированием не требуется — важно понимание специфики *процесса* программирования. Разделы о проектировании программ, нисходящей разработке, структурном программировании и тестировании изложены так, что для их понимания не нужно знать конкретный язык программирования. Однако, если вас интересует вполне определенный язык программирования, вы можете выбрать в соответствии с вашими склонностями главу 6, 7 или 8, где говорится о КОБОЛе, ФОРТРАНе и ПЛИ,

---

<sup>1)</sup> Из популярной в 30-е годы песенки. — *Прим. перев.*



Основная терминология по обработке данных предполагается известной, однако чтобы быть уверенным в том, что и читатель, и авторы понимают термины одинаково, в конце книги помещен глоссарий. Новые термины, возникшие в результате развития излагаемой новой дисциплины, определяются там, где они впервые встречаются в тексте. Если вы администратор, то именно для вас предназначен материал из разделов «Выводы для руководства» соответствующих глав, специально посвященный проблемам руководства. Читатель, далекий от административной деятельности, может пропустить эти разделы без ущерба, однако, возможно, и его они заинтересуют, так как позволят взглянуть на свое дело с иных позиций.

Глава 1 содержит введение в предмет, а также краткую характеристику обсуждаемых в книге направлений и ее структуру. В главе 2 рассматривается применение *нисходящей разработки* на этапе проектирования программы, в главе 3 — применение этого метода на этапах планирования и реализации. В главе 4 описаны основные структурные конструкции — следование, развилка и повторение, а также некоторые вспомогательные структуры, причем изложение не связано с конкретным языком программирования. В главе 5 объясняется, как разрабатывать структурированную программу методом, называемым *пошаговой детализацией*. Главы 6—8 написаны по одной и той же схеме. Каждая из них посвящена применению принципов структурного программирования к конкретному языку. Если вам необходимо овладеть структурным программированием на столь детальном уровне, то следует выбрать для изучения лишь одну из этих глав, посвященную соответствующему языку. В главе 9 речь идет о методе, помогающем убедиться в правильности и надежности программы, — о *сквозном структурном контроле*. Наконец, в главе 10 рассмотрены некоторые дисциплинирующие рекомендации, касающиеся планирования и прогона тестов. В конце каждой главы подводятся итоги и предлагается ряд контрольных вопросов и упражнений.

Мы хотим выразить нашу признательность редактору издательства «Прентис-Холл» Карлу Карлстрому за подбор рецензентов, которые подвергли вашу рукопись весьма квалифицированной критике. Мы хотим также поблагодарить еще трех лиц, чьи технические замечания были наиболее полезны: Кейта Уолтиера из группы по поддержке Усовершенствованных методов программирования отделения фирмы ИБМ в Голландии, Билла Берли, декана факультета обработки данных колледжа Ситрус (Азуса, Калифорния), и Теда Тенни, руководителя обучения в корпорации «Локхид» (Саннивели, Калифорния),

Джоан К. Хьюз  
Джей И. Мичтом

# Новая дисциплина программирования

*“Нахождение глубинной простоты в запутанном клубке сущностей — это и есть творчество в программировании”.*

Х. Д. Милс <sup>1)</sup>

## Пролог

Не все события, изложенные ниже, действительно имели место при создании конкретной программы. Скорее, эта история (или сказка, если хотите) — вольная импровизация на тему реальных ситуаций, возникающих во многих проектах, связанных с обработкой данных.

### *Притча о лишней записи*

Дан, специалист по обработке данных, встречается с потенциальным пользователем вычислительной системы, чтобы выяснить его потребности. В то время как пользователь в самых общих чертах излагает свои требования к обработке информации, Дан уже придумывает хитроумные способы, с помощью которых он собирается запрограммировать решение задачи, и даже кодирует в уме некоторые фрагменты программы. Как только позволяют правила хорошего тона, он заканчивает беседу и спешит к столу начать работу.

Проектирование завершается в кратчайшие сроки, и два программиста — Марсия и Верн — приступают к кодированию. Вначале они работают над наиболее детально продуманными частями. Находясь в одном помещении, они тем не менее редко интересуются делами друг друга. Они знают по опыту, что отладка занимает массу времени. Поэтому они стараются как можно быстрее закончить кодирование, чтобы иметь достаточно времени для отладки. Каждая программа создается сразу на бланке для кодирования, затем немедленно перфорируется. Верн сам идет к перфоратору, чтобы отперфорировать некоторые тесты и *ведущую программу*, которую он написал для проверки своей под-

---

<sup>1)</sup> H. D. Mills, “New Discipline Wins Programmer Approval”, THINK, IBM Corp., March 1973. Reprinted by permission from THINK Magazine, published by IBM, Copyright 1973 by International Business Machines Corporation.

программы. Программные *спецификации* независимо интерпретируются Верном и Марсией с небольшими вариациями (как позднее выяснится, критическими). К тому же и Дан вносит небольшое изменение, о котором сообщает Марсии, но не сообщает Верну, которого не оказалось на месте в тот момент, когда зашел Дан.

Марсия и Верн продолжают кодировать и отлаживать, исправляя все ошибки, какие только способны найти. На вопрос о состоянии какой-либо программы они отвечают: «Осталось совсем немного ошибок». После того как большинство программ написано, их ответ на вопрос о результатах таков: «Готово почти на 90%». Этот ответ не меняется неделя за неделей.

Уже к концу стадии программирования обнаруживается, что в основной записи необходимо дополнительное трехсимвольное поле. Общая длина записи уже определена, поэтому Верн подробно ее изучает и обнаруживает в разных местах записи три неиспользованных одиночных символа. Он решает использовать их в своей программе, чтобы собрать необходимое трехсимвольное поле. Полная программа оказывается слишком большой для оперативной памяти, поэтому она делится расположенными в произвольно выбранных местах специальными командами, с тем чтобы можно было последовательно вызывать порции программы в процессе ее выполнения.

Готовые части *объединяются* для комплексной отладки. На это выделяется одна неделя. Сдача проекта уже задерживается на пять недель; выделяется сверхурочное время, включая третью смену и выходные. Назначаются дополнительно люди, чтобы оказать хоть какую-то помощь. Некоторые записи по необъяснимой причине оказываются записанными в файле дважды подряд. Делается «заплата» к программе вывода, чтобы сохранять предыдущую запись, сравнивать ее с очередной записью и обходить оператор WRITE, если они совпадают. Уверенность в правильности выхода так мала, что делается еще одно добавление, чтобы последовательно проверять выходные записи, как только они попадают в файл.

Обнаруживаются два крупных изъяна в проекте. Для устранения одного из них решают переписать некоторые из программ. Другая проблема требует настолько больших изменений в программе, что обновляется документация с целью ограничить возможности системы.

Наконец, система готова, но пользователь недоволен, так как она не делает того, что он имел в виду. Более того, изменилась окружающая обстановка, необходимы модификации в программах. Под давлением сроков документация не была завершена и блок-схемы не приведены в соответствие с отлаженной программой. Несмотря на бесчисленные заплатки, даже иногда

заплатки на заплатках, Верн и Марсия горды своей системой, так как она полна трюков, демонстрирующих их программистский талант.

Масса времени уходит на сопровождение и модификацию системы. Наконец, решено заняться другими задачами, и все начинается сначала.

Мораль этой истории такова: неадекватное планирование, слабое использование дисциплинирующей, четко определенной технологии ведет к огромным расходам при разработке, сопровождении и модификации программ.

### Что такое структурное программирование?

Сводится ли структурное программирование просто к программированию по строгим канонам? Или это написание подпрограмм (модулей) и объединение их в программу? Наука ли это или искусство? Тот ли это способ, которым издавна пользовались некоторые высококвалифицированные программисты? Может быть, это просто новое наименование для старых идей и процедур? Это программирование «без GOTO»? Всегда ли плохо программировать *неструктурно*? И наконец, что такое *программирование* вообще?

*Программирование* можно было бы определить как «проектирование, кодирование и тестирование программы». Добавляя слово «структурное», мы могли бы сказать, что *структурное программирование* — это «проектирование, написание и тестирование программы в соответствии с *заранее определенной дисциплиной*». Мак-Кракен, однако, сказал: «Немногие рискнули бы дать определение. Не ясно, существует ли простое определение вообще»<sup>1)</sup>. Когда Кнут попросил Хоара дать краткое определение структурного программирования, Хоар ответил, что это — «систематическое использование абстракции для управления массой деталей и способ документирования, который помогает проектировать программу»<sup>2)</sup>.

В структурном программировании важны *форма* и *дисциплина*. Подобно поэту, который сочиняет поэму, так или иначе соблюдая размер и рифму, программист создает программу из базовых логических структур. Мы упоминаем о них здесь, хотя они определяются позже (в этой же главе), так как некоторые склон-

---

<sup>1)</sup> Daniel D. McCracken, "Revolution in programming", Datamation, December 1973, p. 50. Reprinted with the permission of DATAMATION © Copyright 1973 by Technical Publishing Company, Greenwich, Connecticut 06830.

<sup>2)</sup> Donald E. Knuth, "Structured Programming with go to Statements", Computing Surveys, Vol. 6, No. 4, December 1974, p. 292, Copyright 1974, Association for Computing Machinery, Inc., reprinted by permission.

ны отождествлять структурное программирование с таким *программированием*, при котором используются именно эти базовые логические структуры. Пока расхождение во мнениях довольно значительно, и, вероятно, понадобится некоторое время для выработки согласованного определения.

### Зачем нужна эта новая дисциплина?

Раньше хорошими программистами считали тех, кто писал весьма хитроумные программы, которые занимали минимум основной памяти и выполнялись за кратчайшее время. Это было

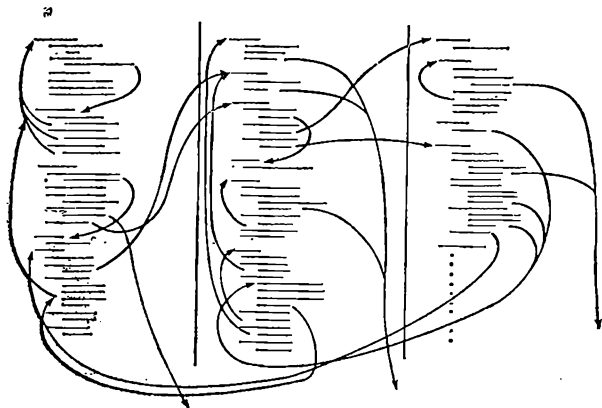


Рис. 1.1. Блюдо спагетти.

естественно, потому что в «старое доброе время» основная память была более дорогой (и, следовательно, размер ее был ограничен), а машины были намного медленнее, чем сегодня. Результатом хитроумного кодирования оказывались программы, которые было трудно (если не совершенно невозможно) понять другим лицам. Программисты зачастую сами признавали, что свою собственную программу они с трудом понимают уже через полгода, а то и через месяц. Фактически каждая написанная программа впоследствии модифицируется при изменении внешних требований или для ликвидации ошибок. Изменения, вносимые в запутанную программу, еще больше усложняют ее понимание впоследствии. Подобные программы получили название *BS-программ*<sup>1)</sup>, так как при попытке разобраться в них необходимо нарисовать нечто, напоминающее блюдо спагетти (рис. 1.1).

Поскольку затраты на содержание персонала растут быстрее, чем производительность, большинство администраторов обеспо-

<sup>1)</sup> Аббревиатура от "Bowl of Spaghetti" (блюдо спагетти).—Прим. перев.

коены тем, как увеличить производительность труда. Некоторые достижения в этой области имеются, но еще многое предстоит сделать. Большая часть времени программистов уходит на нахождение и исправление логических ошибок, а также на модификацию существующих программ в связи с изменениями спецификаций. Значительный рост продуктивности программирования возможен только за счет существенного увеличения надежности программ и облегчения их модификации.

Одна из целей структурного программирования — избавиться от плохой структуры программ. Другая цель — создавать программы, которые можно было бы понимать, сопровождать и модифицировать без участия авторов. По словам многих администраторов, стоимость сопровождения и модификации программ обычно в 3—5 раз больше, чем стоимость их создания. Этот коэффициент не всегда именно таков, но основная тенденция ясна — стоимость эксплуатации программ очень высока и продолжает расти. Сделать программу более понятной, легкой для отладки и сопровождения означает увеличить эффективность процесса обработки данных. Это могло бы уменьшить процент затрат на программное обеспечение и персонал.

Один из наиболее влиятельных сторонников структурного программирования — профессор Э. В. Дейкстра. Он утверждает: «... Я думаю, мы научились столь многому, что в течение ближайших лет программирование может превратиться в деятельность, во многом отличающуюся от того, что имеется сегодня, — настолько отличающуюся, что мы должны очень хорошо подготовиться к ожидающему нас шоку». Он верит, что «семидесятые годы завершатся тем, что мы окажемся способны проектировать и реализовывать такие системы, которые в настоящее время требуют напряжения всех наших способностей, причем расходы на них будут составлять лишь небольшой процент в человеко-годах от их сегодняшней стоимости, и, кроме того, эти системы будут фактически свободны от ошибок»<sup>1)</sup>. Итак, по мнению Дейкстры, на наших глазах происходят коренные изменения в методах программирования.

Другой авторитет в этой области, доктор Харлан Д. Милс, руководящий в фирме ИБМ изучением процесса программирования, сравнивает некоторых сегодняшних программистов-практиков с такими игроками в гольф, которые никогда не считают своих ударов. «Мы, вероятно, посмеялись бы над игроком, который поздравляет себя, загнав мяч во все 18 лунок, но не учитывает, сколько ударов ему для этого потребовалось. Но точно

---

<sup>1)</sup> Edsger W. Dijkstra, "The Humble Programmer", 1972 A. M. Turing Award Lecture, Communications of the ACM, Vol. 15, No. 10, October 1972, p. 863. Copyright 1972, Association for Computing Machinery, Inc., reprinted by permission.

так же относились программисты в прошлом к своей работе. Возможно, они никогда и не спрашивали себя, скольких отладок потребовало создание программы и не слишком ли много места занимает она в памяти машины» <sup>1)</sup>).

### Что такое структурный подход?

Основываясь на соображениях, подобных сформулированным Дейкстрой и Милсом, мы изложим подход, который назовем *структурным подходом к программированию*. Состоит он из трех частей:

- Нисходящая разработка
- Структурное программирование
- Сквозной структурный контроль

Несколько иначе комбинируя ключевые слова, можно было бы сказать, что в этой книге речь идет о *нисходящем структурном программировании*. Поговорим коротко о каждой из трех составляющих структурного подхода, чтобы ввести используемую ниже терминологию. Более подробно каждая из этих составляющих рассмотрена в специально для этого выделенных главах.

#### *Нисходящая разработка*

Традиционно проектирование прикладной системы делается сверху вниз, а программирование — снизу вверх. Рис. 1.2 иллюстрирует это на задаче управления запасами. Каждый прямоугольник мог бы представлять в этой задаче один программный модуль или подпрограмму. При *восходящем* подходе вначале кодируются подпрограммы самого нижнего уровня. Таким образом, программа, которая создает ОТЧЕТ О СОСТОЯНИИ ЗАПАСОВ, могла бы быть сразу же отдана одному из программистов для кодирования. Кодировать и отлаживать модуль ВЫПИСАТЬ ЗАКАЗЫ НА ПОСТАВКУ стал бы, вероятно, другой программист, пока третий делает модули РАСХОД, ПРИХОД и ПРОЧИЕ ОПЕРАЦИЙ.

Когда программы самого нижнего уровня кодируются перед программами более высокого уровня, могут понадобиться отладочные программы для проверки этих модулей. Обычно отладочные программы передают модулю некоторые данные и получают результаты его работы. Затем они могут проверить и (или) напечатать полученные данные, чтобы программист мог прове-

---

<sup>1)</sup> H. D. Mills, "New Discipline Wins Programmer Approval", THINK, IBM Corp., March 1973. Reprinted by permission from THINK Magazine, published by IBM, Copyright 1973 by International Business Machines Corporation,

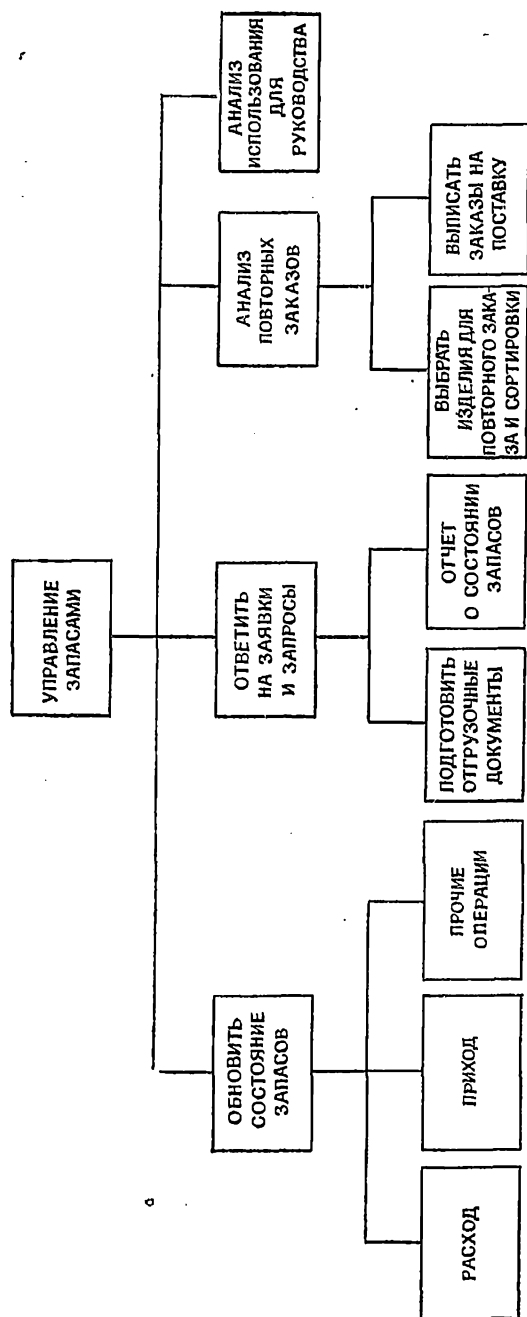


Рис. 1.2. Задача управления запасами.



речь промежуточные результаты. Например, чтобы проверить модуль РАСХОД, нужно написать отладочную программу, которая посылала бы этому модулю *записи с исходными данными*. Эта программа должна моделировать до некоторой степени ту самую среду, в которой будет работать законченная программа РАСХОД, т. е. она должна делать кое-что из того, что и (еще не написанная) программа ОБНОВИТЬ СОСТОЯНИЕ ЗАПАСОВ. Хотя отладочные программы относительно несложны, на их написание уходит время. Обычно это *чисто технологические программы*. Поскольку они не входят в окончательную программу, то затрачиваемые на них средства относятся к накладным расходам.\*

Итак, пусть при традиционном восходящем подходе, которым мы занимаемся, изготовлены и проверены программы самого низкого уровня. Теперь можно писать программы следующего уровня. В процессе их проверки потребуются, возможно, комбинировать их с уже написанными модулями более низкого уровня. Каждый следующий уровень помогает удостовериться в правильности предположений, сделанных в программах нижних уровней и выявить несоответствия или упущения в них. Модули все более высокого уровня проверяются до тех пор, пока не будет изготовлена вся программа. Основная трудность при восходящем подходе в том, что каждый модуль может правильно работать со своей отладочной программой, заставляя программиста думать, что все в полном порядке. Когда же подходит время выполнить все модули вместе, ничего не работает! Как такое может случиться? Объяснить это можно по-разному. Некоторые отладочные программы могли быть написаны на относительно ранней стадии разработки, а спецификации для остальных частей программы могли тем временем измениться, или спецификации могли интерпретироваться по-разному разными людьми, писавшими модули для одной и той же программы, или некоторые особенности полной программы оказалось трудно или даже невозможно моделировать в отладочных программах.

Другая проблема при восходящем подходе и стремлении поскорее перейти к кодированию состоит в том, что трудности имеют тенденцию концентрироваться вокруг заключительной фазы разработки. Например, в разделе АНАЛИЗ ПОВТОРНЫХ ЗАКАЗОВ (рис. 1.2) обнаруживается, что включения в инвентарную запись только количества невыполненных заказов недостаточно. Желательно также включать дату заказа на поставку, чтобы можно было послать дополнительное напоминание, в случае если заказ на поставку не был выполнен в течение уже довольно длительного периода. Это изменение сказывается на программе ОТЧЕТ О СОСТОЯНИИ ЗАПАСОВ (уже закодированной и проверенной) и довольно тонко влияет на модуль

**ПРОЧИЕ ОПЕРАЦИИ** (также закодированный и отлаженный). Кроме этого, возникает вопрос, принадлежит ли модуль ПОДГОТОВИТЬ ОТГРУЗОЧНЫЕ ДОКУМЕНТЫ этой системе или пакету расчетов с покупателями. Чтобы исправить подобные ошибки проектирования, требуется или сделать заплатки в готовых подпрограммах, или перепрограммировать всю задачу, или изменить проект, чтобы сделать его соответствующим гото-

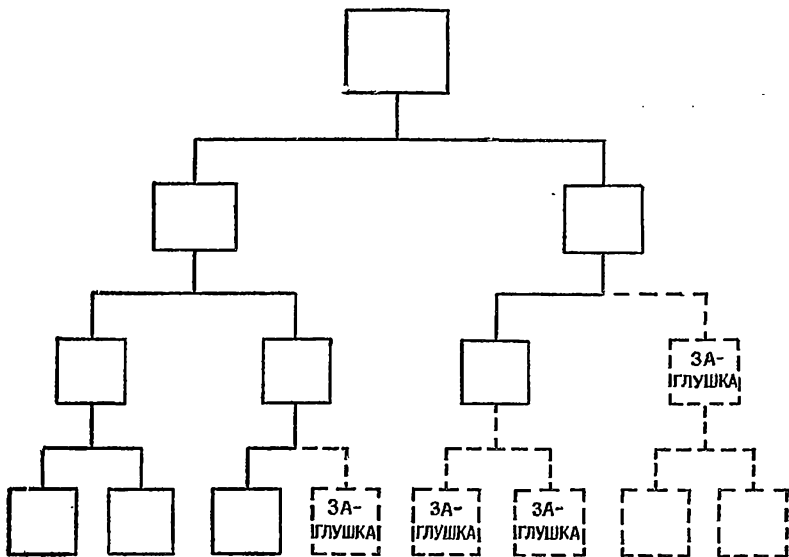


Рис. 1.3. Нисходящая разработка с применением заглушек.

вой программе. Эта так называемая *фаза интеграции* проекта может привести к *перепроектированию, перекодированию, перетестированию и переходу к новому перепроектированию*. В конце концов принимается решение считать программу готовым продуктом, поскольку заказчик не может позволить переносить все время срок сдачи проекта.

При *нисходящей разработке* и проектирование, и программирование ведутся сверху вниз. При этом соответствие функциональных спецификаций системы и составляющих ее программ проверяется до перехода на более низкий уровень спецификаций. Нисходящая разработка помогает избежать проблем, характерных для восходящего подхода. Она концентрирует внимание на частично законченной программе, заставляя исследовать ее «вдоль и поперек», так как интеграция и тестирование идут непрерывно в течение всей разработки. Так как модули разрабатываются сверху вниз, то вместо программ нижнего уровня должны использоваться *заглушки* (рис. 1.3). Про-

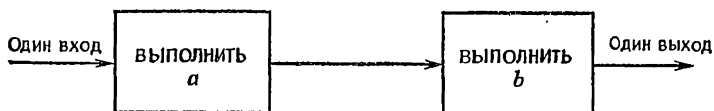
граммная заглушка требуется только для того, чтобы позволить программе верхнего уровня быть выполненной и проверенной, поэтому ее изготовление совсем несложно. Она может не содержать ничего, кроме входного и выходного операторов и сообщения о своем выполнении. Полезно, но не обязательно программировать в заглушке также подготовку данных, нужных модулям верхнего уровня. Заглушки содержат некоторые чисто технологические компоненты, но в конечном итоге их объем меньше, чем в отладочных программах; заглушки обычно устроены проще.

При использовании метода нисходящей разработки детальное проектирование программы, кодирование, проверку и документирование можно делать параллельно. Используя этот подход, можно в некоторых случаях уже в первый день работы над проектом получить первые выдачи. Подчеркнем, однако, что подробное кодирование откладывается на более поздние сроки. О том, как разрабатывать и реализовывать программу с помощью нисходящего подхода, рассказывается в гл. 2 и 3.

### *Структурное программирование*

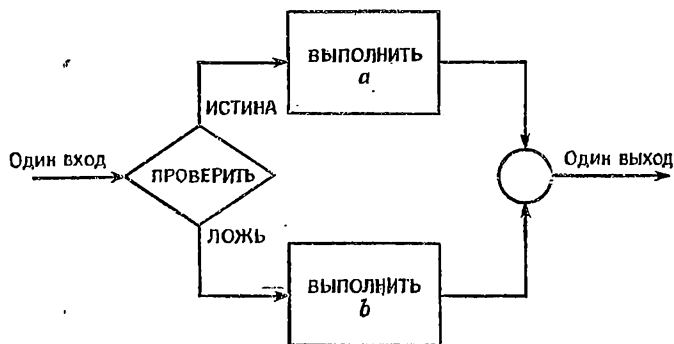
Логическая структура программы может быть выражена комбинацией трех базовых структур. Здесь эти структуры вводятся для того, чтобы дать общее представление о них, предваряющее подробные описания в гл. 4. В применяемых для этого диаграммах *прямоугольник* изображает оператор или функцию (например, оператор READ или подпрограмму извлечения квадратного корня), *ромб* изображает проверку, а *кружок* — слияние двух или более путей.

**Следование.** Это самая важная из структур; она означает, что два действия должны быть выполнены друг за другом.



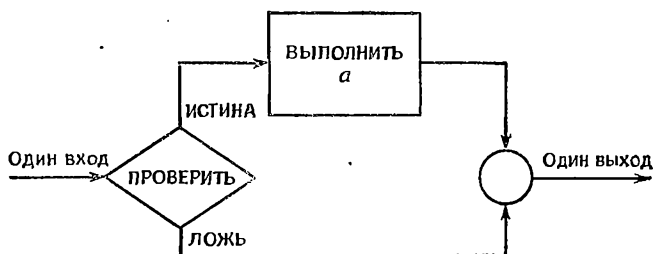
Эти прямоугольники могут представлять как один-единственный оператор типа READ или PUT, так и множество операторов, необходимых для выполнения сложных вычислений.

**Развилка.** Эта структура, называемая также «ЕСЛИ-ТО-ИНАЧЕ», обеспечивает выбор между двумя альтернативами. Делается проверка и затем выбирается один из путей.

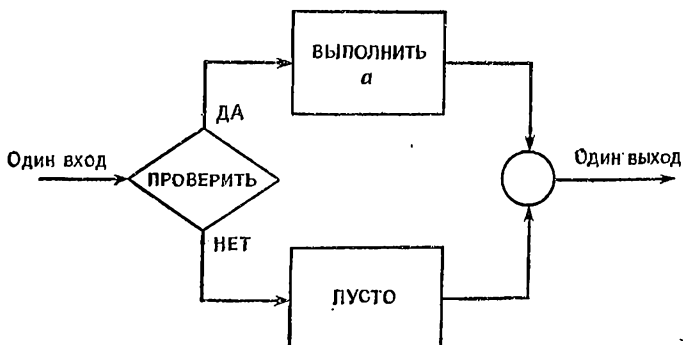


Каждый из путей (альтернатива ТО, альтернатива ИНАЧЕ) ведет к общей точке слияния, так что обработка продолжается независимо от того, какой путь был выбран. Выбираемые пути могут помечаться метками истина/ложь, да/нет, вкл/выкл и т. п.

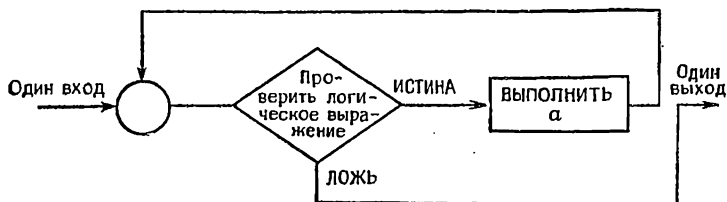
Может оказаться, что для одного из результатов проверки ничего предпринимать не надо. В этом случае можно применять только один обрабатывающий блок.



Или можно указывать блок без действий и помечать его *пусто*.

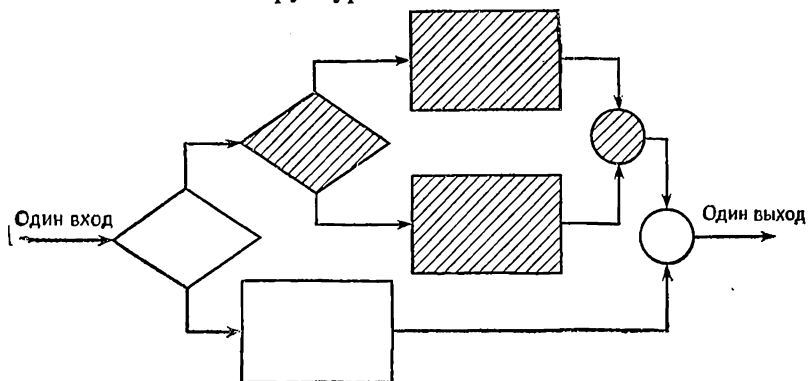


**Цикл или повторение.** Эта структура предусматривает повторное выполнение или циклическую деятельность, необходимую для большинства вычислительных программ. Она изображается следующим образом:



Управление через точку слияния попадает на проверку. Здесь вычисляется *логическое выражение*; если оно *истинно*, то изображенная *обработка* выполняется и выражение вычисляется снова. Такая итеративная деятельность продолжается до тех пор, пока проверяемое выражение истинно. Как только оно становится *ложным*, управление покидает эту структуру. Так как выражение, управляющее циклом, проверяется *в самом начале* (до выполнения какой-либо обработки), то проверяемое условие может сразу оказаться ложным, и в этом случае операторы обработки могут вообще не выполняться. Операторы, представленные блоком *обработки*, должны изменять управляющую переменную, влияющую на проверку — иначе программа зациклится. Эта структура повторения могла бы включать много операторов (возможно, даже большую программу). Она называется **ЦИКЛ-ПОКА** и означает *повторять* (т.е. выполнять операторы обрабатывающего блока), *пока* указанное логическое выражение остается истинным.

Эти три структуры могут комбинироваться одна с другой, как того требует программа. Таким образом, где бы на диаграмме ни появился отдельный прямоугольник, он может быть заменен любой из базовых структур.



Таким образом, эти строительные блоки могут комбинироваться сколь угодно разнообразно для выражения логики любой программы. Хотя на наших рисунках программа выполняется слева направо, эти структуры можно нарисовать и так, что основное направление будет сверху вниз.

Используя только что определенные структуры, возможно (в зависимости от используемого языка программирования) писать программы без операторов GOTO. По этой причине структурное программирование иногда определяется как программирование *без GOTO*. Однако это слишком узкое определение, непригодное для некоторых языков программирования. Суть в том, что ликвидация операторов GOTO — лишь побочный продукт выражения логики программ с помощью только перечисленных выше структур.

Конечно, бесконтрольное использование операторов GOTO (например, такое, как на рис. 1.1) должно быть исключено. Однако дискуссии о том, следует ли полностью исключить операторы GOTO, продолжаются. Внимание к этому вопросу привлек Дейкстра. В своем письме редактору журнала Communications of the ACM<sup>1)</sup> он писал:

*«На протяжении многих лет я очень хорошо знал, что квалификация программистов — убывающая функция от плотности операторов GOTO в создаваемых ими программах. Но лишь совсем недавно я обнаружил, почему использование оператора GOTO имеет такие губительные последствия. Я пришел к убеждению, что этот оператор должен быть исключен из всех языков программирования высокого уровня (кроме, возможно, машинного кода)».*

Кнут, однако, указал несколько особых ситуаций, когда оператор GOTO может быть полезен<sup>2)</sup>.

Структурное программирование — это не просто программирование без GOTO. Это дисциплина программирования, которая объединяет несколько способов создания ясной, легкой для понимания программы. Вполне возможно писать структурированные программы, содержащие операторы GOTO, равно как и неструктурированные программы, не содержащие ни одного GOTO.

---

<sup>1)</sup> Edsger W. Dijkstra, "GO TO Statement Considered Harmful", Communications of the ACM, Vol. 11, No. 3, March 1968, p. 147. Copyright 1968, Association for Computing Machinery, Inc., Reprinted by permission.

<sup>2)</sup> Donald E. Knuth, "Structured Programming with go to Statements", Computing Surveys, Vol. 6, No. 4, December 1974. Copyright 1974, Association for Computing Machinery, Inc.,

Еще один результат использования только базовых структур состоит в том, что поток управления в программах будет направлен *сверху вниз*. Это означает, что составляющие программу операторы будут выполняться в том порядке, в котором они появляются в тексте программы. Конечно, некоторые фрагменты будут пропускаться (как в случае ЕСЛИ-ТО-ИНАЧЕ), а некоторые будут повторяться (как в случае ЦИКЛ-ПОКА), но все разветвления будут направлены только вперед. Это облегчает чтение и модификацию программы.

### « Сквозной структурный контроль

По-видимому, каждое десятилетие предоставляет руководителям новые возможности делать более эффективной их работу по управлению людьми и проектами. В 50-е годы это был *мозговой штурм* — метод, в соответствии с которым люди специально собирались вместе, чтобы решить определенную задачу. Основная идея мозгового штурма состоит в том, чтобы высказывать даже совершенно фантастические предположения без их осуждения или оценки до тех пор, пока предложения не иссякнут. И сегодня мозговой штурм — очень полезный метод при проектировании сверху вниз.

В 60-е годы возникло так называемое МВО (Management By Objectives — управление целями). Для достижения какой-то цели требуется не только управление людьми и проектом, но и точное определение самих целей. Такие цели могли бы, например, быть распределены по четырем группам, как показано в табл. 1.1.

Таблица 1,1

#### Точно определенные цели

Задание	Оцениваемое улучшение	Стоимость	Срок достижения цели
Переделать систему управления кредитами	Свести период возврата к 2,4 месяца	В пределах 5000 долларов на программирование	1 июля

И сегодня остается важной такая формулировка целей, чтобы было возможно *проверить факт их достижения*. Из общих целей, определенных в начале проектирования, следуют частные цели каждой фазы разработки.

В текущем десятилетии подобной новинкой может стать *сквозной структурный контроль*. Его основная идея — регуляр-

ные встречи исполнителей с целью взаимной проверки работы как на стадии разработки, так и на стадии программирования.

Сквозной структурный контроль — это общее название, предложенное фирмой ИБМ для серии проверок, проводимых с различными целями и на разных фазах цикла разработки программ. Основная цель — добиться того, чтобы каждая достаточно значительная доля затраченных усилий была внимательно изучена компетентными людьми, к тому же непосредственно заинтересованными в успехе. Сквозной контроль используется и для проверки спецификаций вместе с пользователем, и для проверки проекта, чтобы быть уверенным в том, что он соответствует спецификациям, и чтобы убедиться в правильности плана тестирования и самих тестов. При этом программисты проверяют программы друг друга *до* их перфорации.

Чтобы понять механизм структурного контроля, предположим, что некоторый программист работает над частью детального проекта программы.

1. По завершении своей части работы программист организует так называемую контрольную сессию. Он приглашает четырех своих коллег, чтобы они проверили, правильно ли он организовал *сопряжение* по данным между своим модулем и другими модулями проектируемой системы.

2. Когда встреча назначена, участникам сообщается о цели сессии. Кроме этого, они получают рабочие материалы (например, список спецификаций, допущений, схемы взаимодействия), чтобы изучить их до сессии. Требуется, чтобы ее участники были знакомы со всеми материалами. *Этот контроль структурный в том смысле, что все его участники знают, в чем состоит задача и какова их роль в ее выполнении.*

3. На каждую сессию назначается председатель. Им может быть сам программист или любой другой участник сессии, но ни в коем случае не *администратор*. Руководство не присутствует на сессии, чтобы она не служила основанием для оценки персонала.

4. Во время сессии ошибки обнаруживаются, записываются, но не исправляются. Один из присутствующих играет роль секретаря и готовит *список дефектов* (т. е. ошибок или упущений). Этот список будет затем разослан всем участникам сессии для рассмотрения.

5. В зависимости от числа и типа обнаруженных дефектов сессия может быть повторена, пока дефекты не перестанут обнаруживаться. Только в этом случае программа передается на следующую стадию разработки.

Таким образом, сквозной структурный контроль — это средство анализа проекта и обнаружения логических ошибок на воз-



можно более ранней стадии проектирования программы. Это позволяет уменьшить число ошибок кодирования. Сессии способны положительно повлиять как на группу проектировщиков, так и на изучающих опыт этой группы. При этом предъявлять для изучения результаты своей работы должны все проектировщики, от самых старших до самых младших.

Итак, структурный контроль — это прежде всего продуктивно работающие сессии, которые не только следят за продвижением вперед, но и сами вносят вклад в это продвижение. Организационные и психологические аспекты сквозного структурного контроля описаны в гл. 9.

### Выводы для руководства

**Нисходящая разработка.** Когда проект системы и разработка делаются сверху вниз, руководство получает возможность лучше управлять ходом событий. При традиционном подходе, когда кодирование и тестирование велись снизу вверх, руководство никогда не знало, на какой же стадии находится проект. Даже если становилось известно, что определенные модули прошли сепаратную отладку, руководство не могло предвидеть тех ошибок, которые несомненно возникали при окончательной сборке программы.

Часто такая сборка программных модулей в одну *систему* начинается только на заключительной стадии проекта. Если при этом обнаруживаются новые ошибки, то сроки завершения проекта обязательно срываются. Дело в том, что когда используется восходящий подход, руководитель может контролировать деятельность программистов еженедельно, но все равно не будет знать даже с точностью до месяца, насколько она близка к завершению. Нисходящий подход дает возможность получать надежные и легко проверяемые результаты с самого начала работ и на всех последующих стадиях производственного цикла. Другими словами, система может постоянно поддерживаться в действующем состоянии и ее можно непрерывно проверять по ходу ее создания.

**Структурное программирование.** Программист, использующий при проектировании и кодировании рассмотренные в этой главе базовые структуры, выигрывает втрое: программа становится *понятней*, повышается ее *надежность* и облегчается ее *сопровождение*. При овладении структурным подходом становится возможным программировать почти без ошибок. Необходимость в детальных блок-схемах для объяснения программ уменьшается или вообще отпадает. Сокращается время на документирование, так как сами программы становятся документами. Наконец,

уменьшаются затраты на содержание персонала, так как возрастает производительность.

Структурный подход предполагает стиль работы, который поощряет или даже заставляет программистов по возможности откладывать детализацию. Каждая стадия разработки должна выполняться очень тщательно, кроме, возможно, самых ранних. К сожалению, в области программирования имеется тенденция поскорее перейти к кодированию, что объясняется тремя причинами:

1. Для многих программистов это наиболее интересная и захватывающая часть их работы.

2. Некоторые администраторы и руководители проектов считают, что сотрудники работают *только* тогда, когда они пишут конкретные команды.

3. Этапы кодирования (например, создание модуля ОТЧЕТ О СОСТОЯНИИ ЗАПАСОВ) хорошо определены, тогда как точные методы или какие-либо правила обдумывания проекта в целом и общего взаимодействия модулей отсутствуют.

И тем не менее, по словам Х. Милса, очень важно «настойчиво сопротивляться кодированию» для того, чтобы стадия разработки завершилась успешно.

**Структурный контроль.** Структурный контроль необходим для того, чтобы обнаружить и исправить ошибки как можно раньше, пока стоимость исправления ошибок минимальна, а их последствия наименее значительны. Это не *оценка состояния проекта* в традиционном смысле, так как руководство не участвует непосредственно в контрольных сессиях. Оно может, конечно, узнавать о продвижении от руководителя группы программистов. С точки зрения *личного роста* участникам структурного контроля предоставляется удобный случай поучиться новым методам и отшлифовать свое мастерство в процессе взаимного контроля.

## Эпилог

А теперь мы хотим рассказать о том, как был реализован структурный подход при работе над проектом в одной из фирм по производству одежды в Северной Каролине. В отличие от нашей первой истории, отразившей типичные неурядицы многих различных проектов, рассказанное ниже имело место при работе над вполне конкретным проектом.

Проект был поручен бригаде из трех человек. Один из членов бригады выполнял роль *бригадира*. Одна из его обязанностей

заключалась в том, чтобы контролировать работу двух других членов бригады и информировать администрацию о состоянии проекта. Все трое были и аналитиками, и программистами и при работе над проектом выступали в обеих ролях.

После первых встреч с пользователем, для которого писалась программа, бригада собралась, чтобы обсудить задачу, прояснить требования и вскрыть недостатки в спецификациях. Затем они снова встретились с пользователем, детально изложив ему свое понимание требований. На этой *контрольной сессии* спецификации были обсуждены и согласованы. Теперь бригада была готова начать проектирование программы.

Проектирование было итеративной процедурой. Вначале определялись и внимательно проверялись модули самого высокого уровня. Затем методом сверху вниз определялись модули уровнем ниже. Когда структурный контроль обнаруживал недостатки или упущения в спецификациях, проект исправлялся. Когда возникала твердая уверенность, что все правильно, модули самого высокого уровня кодировались и тестировались по принципу сверху вниз. При необходимости создавались заглушки. Первоначально проверить весь проект означало просто выполнить только управляющие модули самого высокого уровня. Когда же было завершено детальное проектирование нижних модулей, кодирование продолжалось сверху вниз, затрагивая модули, ранее заменявшиеся заглушками. Все модули кодировались с помощью базовых структур, введенных в этой главе. Это заставляло программиста внимательно продумать каждый модуль до его кодирования. Члены группы были откровенны между собой и без предвзятости относились к своей работе при перекрестном изучении текстов с целью обнаружить логические ошибки или ошибки проектирования.

Как только был готов очередной модуль, он немедленно подключался к уже работающей программе и проверялся в совместной работе с другими модулями. И все модули работали правильно с первого запуска! Не оказалось логических ошибок и в полной программе, которая состояла из 10 модулей и 7200 строк готового текста. Окончательная проверка состояла из единственного прогона, во время которого программа правильно сработала уже как единое целое. Не было и ошибок во взаимодействии модулей.

Проект был завершен досрочно (за 95% отведенного времени) и без сверхурочной работы. (Предыдущие проекты в этой организации ориентировались на традиционные методы и обычно выбывались из графика.) Таким образом, дополнительные усилия для создания сразу правильной программы не затянули сроки реализации.

Среднее число строк на языке высокого уровня, отлаженных

одним программистом за день, учитывая весь срок работы над проектом, равнялось 99. Если же считать только этап программирования, то это число строк равнялось 212. Общая производительность труда была примерно в четыре раза выше достигнутой в предыдущих проектах в этой же организации. Полученная программа была модульной и понятной. Дополнительные преимущества дала простота сопровождения. Подобные результаты были получены и в других организациях, где был применен нисходящий структурный подход.

## Выводы

Итак, предлагается бороться с ошибками тремя способами: 1) уменьшать число возможных ошибок за счет применения только допустимых структур (следование, развилка, повторение), 2) обнаруживать ошибки как можно раньше и 3) принимать меры к тому, чтобы нахождение и исправление ошибок, найденных позднее, было как можно проще и дешевле. Конечно, чем меньше ошибок, тем лучше. Имеются некоторые данные об относительной стоимости нахождения и исправления ошибок в зависимости от времени их обнаружения. Раннее обнаружение намного дешевле — в 10, 20 и даже 30 раз!

Структурное программирование «способно привести к радикальным переменам в программировании. Наиболее очевидная выгода — это рост производительности и уменьшение процента ошибок. Программирование постепенно превращается из ремесла в науку. Аналогичное влияние оказало в свое время на специалистов по аппаратуре открытие, что всякую логическую сеть можно построить из нескольких базовых операций типа «И» и «ИЛИ». Программирование сейчас находится на такой же стадии зрелости»<sup>1)</sup>.

«Если структурное программирование будет принято в качестве стандарта, то станет необходимым писать все программы в соответствии с этим стандартом. Но даже если такой стандарт не будет принят, программисты должны быть знакомы с его основными идеями, чтобы это помогало им в их стремлении писать более эффективные и понятные программы»<sup>2)</sup>.

---

<sup>1)</sup> Daniel D. McCracken, "Revolution in Programming", *Datamation*, December 1973, pp. 50—52. Reprinted with the permission of DATAMATION® Copyright 1973 by Technical Publishing Company, Greenwich, Connecticut 06830.

<sup>2)</sup> John M. Morris, "Structured Programming", *Pattern Analysis and Recognition Corp.*, Rome, N. Y., Tech. Memo # 73—20, 1973, p. 2.

**Контрольные вопросы и упражнения**

1. Дайте определение структурному программированию.
2. Назовите две главные цели структурного программирования.
3. Является ли структурное программирование программированием без GOTO? Объясните, почему.
4. Каковы характеристики нисходящего подхода?
5. Какого рода проблемы вы видите при внедрении структурного контроля в организации, где работа программистов никогда не изучалась их коллегами или руководителями проекта?
6. Назовите как можно больше характеристик структурированной программы.
7. Рассмотрите одну из своих прежних программ и оцените ясность ее логики. Как бы вы улучшили эту программу, если бы имели возможность ее переработать?

## Нисходящая разработка: проектирование программы

### *Законы Мэрфи:*

*Все сложнее, чем кажется.  
Все тянется дольше, чем можно ожидать.  
Все оказывается дороже, чем планировалось.  
Если что-то может испортиться, оно обязательно  
портится.*

*Комментарий Каллагана к законам Мэрфи:  
Мэрфи был оптимистом.*

### Модульность

Растущий интерес к структурному программированию может быть как проявлением постепенного осознания правоты Мэрфи, так и наступлением определенной зрелости в вычислительном деле. Структурное программирование возникло именно из-за того, что «все сложно, тянется дольше и стоит дороже, чем ожидалось». Нисходящая разработка призвана уменьшить сложность программы и дать возможность закончить ее вовремя. Если «что-то портится», то предлагаются средства, которые помогают обнаружить такие места как можно раньше и оставить достаточно времени на их устранение.

Нисходящая разработка может применяться на всех фазах проектирования системы, включая как проектирование программ этой системы, так и проектирование модулей для этих программ.

На рис. 2.1 показана система расчета заработной платы, включающая программы еженедельных выплат, регистрации выплат, контроля записей, модификации файла при добавлениях (прием на работу), исключениях (увольнение) и изменениях (изменение оклада, удержаний и т. д.), местных и штатных ежеквартальных отчетов и подготовки формы W-2. На этом рисунке верхний прямоугольник не обозначает никакой программы. Это просто имя системы, но можно считать и так, что этот прямоугольник обозначает инструкцию для исполнения программы и всю остальную документацию.

Каждая из программ этой системы решает свою часть большой задачи. Некоторые из этих программ довольно сложны, поскольку

должны уметь многое делать. Например, программа еженедельных выплат делает как минимум следующее:

- получает информацию о рабочем времени сотрудника и проверяет ее правильность,
- готовит отчетные данные на случай ревизии,
- вычисляет очередную выплату,
- обновляет основной файл сотрудников,
- вычисляет суммарный заработок от начала года по сегодняшний день и обновляет соответствующие файлы.

Подобные программы легче проектировать и реализовать, если их в свою очередь разделять на модули. Тогда программа, решающая большую задачу, состоит из одного или нескольких

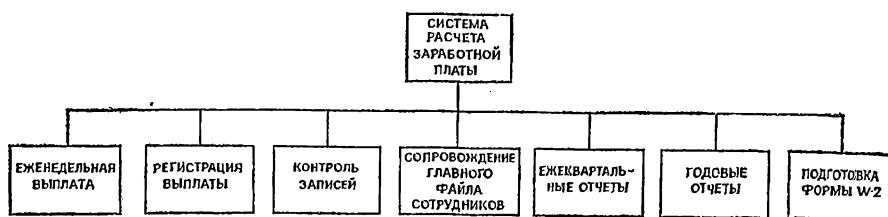


Рис. 2.1. Система расчета заработной платы.

модулей и является подсистемой. *Модуль* — это последовательность логически связанных фрагментов, оформленных как отдельная часть программы.

Понятие модульности не ново. Модульное программирование возникло еще в начале 60-х годов. Оно характеризуется следующими преимуществами:

1. Большую программу могут писать одновременно несколько исполнителей — это позволяет раньше окончить работу.
2. Можно создавать библиотеки наиболее употребительных подпрограмм.
3. Становится проще процедура загрузки в оперативную память большой программы, требующей сегментации.
4. Возникает много естественных контрольных точек для наблюдения за продвижением проекта.
5. Облегчается более полное тестирование.
6. Проще проектирование и последующие изменения программы.

Из этих шести преимуществ последние три особо существенны для организаций, которым трудно вовремя создавать хорошо проверенные программы. Важность последнего пункта особенно возросла в связи с тем, что стоимость сопровождения и модифи-

кации программ составляет значительную часть общих расходов на обработку данных.

Наряду с этими преимуществами имеются и некоторые недостатки, которые могут привести к возрастанию стоимости программы.

1. Может увеличиться время исполнения программы.
2. Может возрасти размер требуемой памяти.
3. Может увеличиться время компиляции и загрузки.
4. Проблемы организации межмодульного взаимодействия могут оказаться довольно сложными.

Для современных компиляторов и операционных систем эти недостатки невелики и обычно окупаются сокращением стоимости разработки и сопровождения.

### *Свойства модуля*

В этой главе речь идет о том, как проектировать программу — т. е. как *разбивать* ее на составляющие модули — нисходящим способом. Однако прежде, чем об этом говорить, нужно дать читателю четкое представление о том, что такое модуль. Начнем с перечисления некоторых его желательных свойств.

1. Модуль возникает в результате сепаратной компиляции (или является частью результата *совместной компиляции*). Он может активизироваться операционной системой или быть подпрограммой, вызываемой другим модулем.

2. На внутренность модуля можно ссылаться с помощью имени, называемого именем модуля.

3. Модуль должен возвращать управление тому, кто его *вызвал*.

4. Модуль может обращаться к другим модулям.

5. Модуль должен иметь один *вход* и один *выход*. Иногда программа с несколькими входами может оказаться короче и занимать меньше места в памяти. Однако изучение опыта организаций, применяющих модульное программирование, показало, что пользователи предпочитают иметь несколько похожих модулей, но не использовать несколько входов или выходов в одном модуле. Объясняют они это тем, что единственность входа и выхода гарантирует замкнутость модуля и упрощает сопровождение программ <sup>1)</sup>.

6. Модуль сравнительно невелик. В обзоре, упомянутом в предыдущем абзаце, отмечается, что модули содержат от 20 до

---

<sup>1)</sup> John Rhodes, "Tackle software with modular programming", reprinted from COMPUTER DECISIONS, July 1975, Copyright 1975, Hayden Publishing Company.



2000 предложений. Однако модули, состоящие из более чем нескольких сотен строк, встречаются сравнительно редко. Обычно в них менее 200 строк или даже менее одной-двух страниц исполняемого текста на входном языке. «Использование небольших модулей имеет определенные преимущества. Обнаружено, что небольшие модули позволяют строить такие программы, которые легче изменять, такие модули чаще используются, они облегчают оценку и управление разработкой, легче и качественнее тестируются, их можно рекомендовать и достаточно опытным, и неопытным программистам. С другой стороны, небольшие модули дольше проектируются, требуют большего числа связей, медленнее работают, состоят все вместе из большего числа предложений исходного текста, требуют большей документации, их написание может быть менее приятным для программиста»<sup>1)</sup>).

7. Модуль не должен сохранять историю своих вызовов для управления своим функционированием.

8. Модуль обладает единственной функцией: *это вполне определенное преобразование исходных данных в результат, осуществляемое в процессе исполнения данного модуля*. В этом должны состоять все изменения, происходящие от момента входа в модуль до момента завершения его работы. В идеале каждый модуль должен реализовать только *одну* функцию, причем *целиком*. Концепция *один модуль — одна функция* служит ключом к хорошо спроектированным программам. Другими словами, модуль — это элемент программы, выполняющий самостоятельную задачу. Его функция может быть выражена одной фразой. Вот примеры таких функций:

Редактировать запрос

Загрузить главный файл

Вычислить недельный заработок

Вычислить среднее число перемещений за девять месяцев

Обратить матрицу

Напечатать отчет о состоянии запасов

Таким образом, при проектировании программы нужно сначала определить необходимый набор функций, а затем разработать модули программы. Например, программа **ОБНОВИТЬ ГЛАВНЫЙ ФАЙЛ** в рассматриваемой системе могла бы состоять из следующих основных модулей.

1. Модуль *чтения* и распечатки входной информации.

2. Модуль *изменения* существующей записи главного файла (изменение оклада, удержаний и др.).

---

<sup>1)</sup> Там же,

3. Модуль *добавления* записи в главный файл (прием на работу).

4. Модуль *перемещения* записи из активного главного файла в неактивный файл (увольнение).

Кроме того, мог бы потребоваться также модуль *начальных действий* (установка индикаторов, открытие файлов, присваивание начальных значений переменным). На рис. 2.2 приведены функции каждого модуля, а также связь между этими модулями. В этом примере головной модуль, определенный общей функцией программы, активизирует по мере необходимости другие



Рис. 2.2. Программа, разделенная на модули.

модули. Здесь предполагается, что головной модуль работает в условиях, когда выполнены начальные требования для программы Сопровождение файла сотрудников. Конечно, каждый из изображенных на рисунке модулей также может требовать специфических начальных условий. Важно понять, что на этом этапе проектирования не нужно думать о том, *как* выполняет свою функцию каждый модуль. Мы не занимаемся пока логикой программы. Важно лишь то, что *вызывающий* модуль (т. е. головной модуль на рис. 2.2) рассматривает *вызываемый* модуль (т. е. ДОБАВЛЕНИЯ) как «черный ящик». *Черный ящик* характеризуется только своим именем и результатами работы. Его можно использовать, ничего не зная о его устройстве.

На рис. 2.2 каждый модуль выполняет единственную функцию. Например, головной модуль вводит главные записи и порождает обновленные записи. Конечно, этот модуль должен обращаться за помощью к другим модулям.

Модульность, основанная на точном соответствии функциям, особенно выгодна тем, что позволяет получать модули, применимые где угодно. Например, модуль ЧИТАТЬ ЗАПРОС можно использовать в других системах кадрового учета, поскольку

здесь обрабатываются записи того же формата. Такая модульность хороша еще и тем, что модули легче проверять. Поскольку функция — это определенное преобразование входной информации в выходную, то вход и выход известны уже тогда, когда становится ясно, что именно такой модуль необходим в данной программе. После его изготовления остается только убедиться, что для заданного набора входных данных получаются именно требуемые результаты.

### *Ограничение сложности модуля*

Если логика реализации отдельных функций становится очень запутанной, сложность модуля растет. Но модули применяются именно затем, чтобы ограничить сложность. Конечно, требование, чтобы модуль реализовал только одну функцию, — лишь первый шаг в этом направлении. Другие способы ограничить сложность модуля состоят в том, чтобы ограничить

- 1) количество предложений в тексте модуля,
- 2) размер оперативной памяти,
- 3) число ветвлений в программе,
- 4) число возможных путей через модуль,
- 5) время разработки модуля.

Может применяться одно или сразу несколько из этих ограничений. Стоит отметить, что само по себе ограничение лишь размеров программы не позволяет отделить длинные, но простые фрагменты от коротких, но запутанных. Другие проблемы, связанные с модульностью, касаются перекрытия в памяти программ и данных, использования виртуальной памяти и особенностей языка программирования.

### **Нисходящее проектирование программ**

*Нисходящее проектирование программ* основано на идее уровней абстракции, которые становятся уровнями модулей в создаваемой программе. Фрост определяет *абстрагирование* как процесс «обобщения, при котором внимание концентрируется на сходстве явлений и предметов, и они объединяются в группы на основе этого сходства, давая тем самым нужную абстракцию» <sup>1)</sup>. Например, абстракция *готовые счета* полезна для тех, кто хочет

---

<sup>1)</sup> David Frost, "Psychology and Program Design", Datamation, May 1975, p. 137. Reprinted with the permission of DATAMATION © Copyright 1975 by Technical Publishing Company, Greenwich, Connecticut 06830.

работать без использования таких понятий, как накладные, чеки, платежи или списки покупателей. Термины «накладные» и др. являются более низкими уровнями абстракции.

### Схемы иерархии

Уровни абстракции определяют уровни модулей в программе. На этапе проектирования строится *схема иерархии*, изображающая эти уровни. По внешнему виду она напоминает организационную схему. Их логическое сходство усиливается тем, что схема иерархии отражает функции и *взаимодействие* модулей. Каждый прямоугольник в ней изображает функцию или модуль.

От блок-схемы схема иерархии отличается тем, что не показывает логику принятия решения или точный порядок исполнения. Например, блок-схема на рис. 2.3 описывает часть системы расчета заработной платы. В ней отражены правила начисления удержаний и последовательность их расчета. С другой стороны, схема иерархии, приведенная на рис. 2.4, описывает лишь функции и их взаимосвязь в программе. Проявлена группировка функций, отсутствующая на блок-схеме. Каждое удержание выделено в самостоятельный модуль, и все они являются детализацией более общего ведущего модуля **ВЫЧИСЛИТЬ УДЕРЖАНИЯ**. Блок-схема показывает *процедуру*, схема иерархии — *функцию*. Схема иерархии позволяет программисту сначала сконцентрировать внимание на определении того, *что* нужно сделать в программе, а лишь затем решать, *как* это нужно

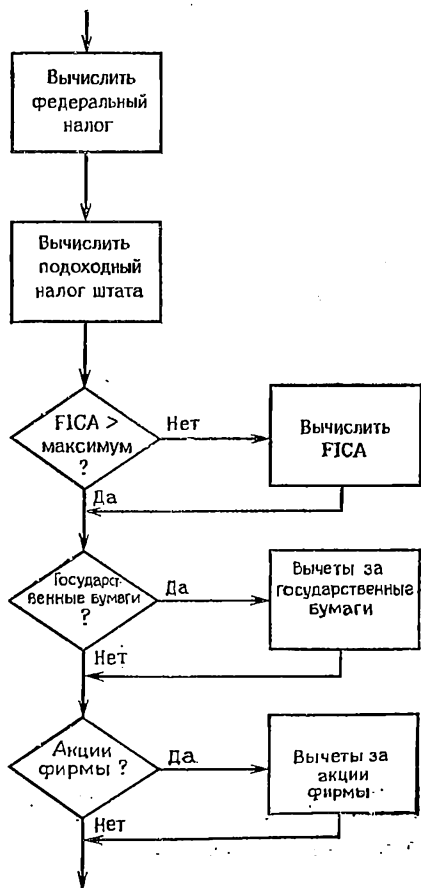


Рис. 2.3. Блок-схема начисления удержаний.



Рис. 2.4. Схема иерархии начисления удержаний.

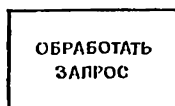
делать. Она хороша также тем, что явно группирует взаимосвязанные функции — это ключ к проектированию хороших программ.

### *Разработка схемы иерархии*

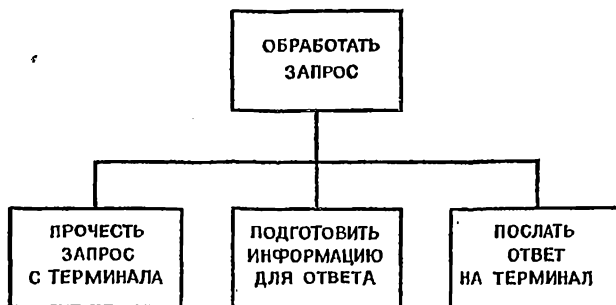
Чтобы создать схему иерархии, т. е. спроектировать структуру модулей, следует начать с вершины и идти вниз. (Отсюда и термин «нисходящее проектирование».) Нужно, чтобы один модуль в программе управлял выполнением остальных модулей, подобно тому как президент компании управляет служащими фирмы.

Рассмотрим, например, диалоговую банковскую систему, в которой с терминала вводятся запросы о состоянии счетов вкладчиков. Примерами могут быть запросы о текущем остатке, наибольшем и наименьшем остатках за месяц, размере последнего вклада.

В схеме иерархии для программы обработки этих запросов прежде всего могла бы появиться самая общая функция.



Затем появляются модули, необходимые для идентификации и детализации общей функции. Могли бы, скажем, возникнуть следующие три дополнительные функции:



Здесь все модули *передают* информацию в главную управляющую программу ОБРАБОТАТЬ ЗАПРОС. Эта программа активизирует подчиненные модули, проверяет их результаты, принимает решения и осуществляет управление.

Подобный подход хорош для небольших или простых программ. Однако с возрастанием сложности программы растет и сложность главного модуля. В программе, где одна подпрограмма управляет сотней модулей, главный модуль будет настолько сложным, что его трудно будет отлаживать и изменять. Выход здесь такой же, как и в случае с управлением компанией. Если ее численность возросла до ста служащих, то президенту становится трудно управлять ими всеми непосредственно. Скорее всего часть работ по управлению и принятию решений будет возложена на вице-президентов. В программе также появляются подпрограммы второго уровня, подчиненные главной подпрограмме. Эти подпрограммы будут осуществлять некоторые функции управления и принятия решений. В этом случае структура модуля становится иерархической.

Президент компании может распределять обязанности между вице-президентами многими способами. Одним из способов разделения обязанностей между тремя людьми может быть такой:

1-й вице-президент управляет всеми служащими, чьи фамилии начинаются с букв от А до I;

2-й вице-президент управляет всеми служащими, чьи фамилии начинаются с букв от J до R;

3-й вице-президент управляет всеми служащими, чьи фамилии начинаются с букв от S до Z.

Более вероятно, однако, что президент распределит обязанности в соответствии с тремя основными функциями компании — сбыт, производство и исследования и разработки. Подобным образом и в программе подпрограммы второго уровня — это ее основные функциональные подразделения. Эти модули затем также могут быть разделены на подчиненные модули.

Возвращаясь к нашему примеру, приведем расширение модуля ПОДГОТОВИТЬ на рис. 2.5. (Имена модулей приведены вне прямоугольников.)

Схемы иерархии не показывают потока данных, порядка исполнения или моментов и частоты активизации каждого модуля. Расположение модулей на заданном уровне не определяет порядок их исполнения. Обычно люди стремятся мыслить «слева направо», и нет ничего плохого в расположении прямоуголь-

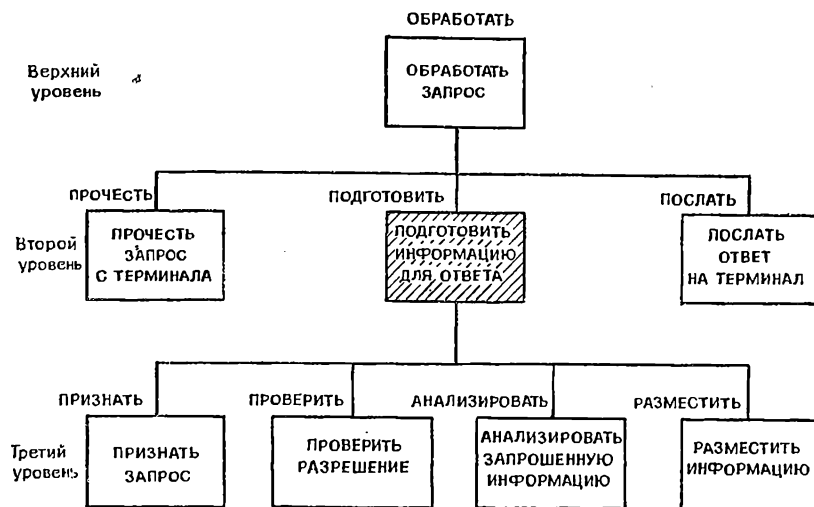


Рис. 2.5. Пример схемы иерархии.

ников именно так. Во всяком случае в этом примере незачем было умышленно располагать их в другом порядке. Однако, читая схему иерархии, не следует *предполагать*, что модули всегда будут исполняться слева направо. Управление частотой и порядком выполнения скрыто внутри прямоугольников и не показано на схеме.

Предположим, например, что при чтении запроса с терминала обнаружена ошибка. Следующим будет активизирован не модуль ПОДГОТОВИТЬ, а модуль ПОСЛАТЬ, чтобы сообщить пользователю о допущенной ошибке. В этом случае средний модуль ПОДГОТОВИТЬ вообще не будет выполняться.

Однако если линии не показывают порядок выполнения, то что же они показывают? Они показывают *подчиненность* модулей. Каждый модуль активизируется вышестоящим и, закончив свою работу, *возвращает* управление вызвавшему модулю. Таким образом, вызываемая подпрограмма подчинена вышестоящему модулю и подчиняет себе нижестоящие модули. Продолжая

аналогию со схемой организации, можно сказать, что она получает «приказы» от «хозяина» и возвращает ему результат своей работы. При этом подпрограмма может до своего завершения активизировать один или несколько подчиненных модулей.

### *Вертикальное управление*

Передачи управления происходят лишь по вертикальным линиям, соединяющим модули в схеме иерархии. Это означает, что любой модуль может активизировать подчиненный модуль более низкого уровня и получить управление после завершения его работы. Такое вертикальное управление происходит по следующим правилам:

1. Модуль должен возратить управление тому, кто его *вызвал*. Единственным исключением из этого правила может быть только обнаружение неисправимых ошибок, требующее немедленного завершения программы.

2. Модуль может вызывать другой модуль уровнем ниже; он не может вызывать модуль своего уровня или выше. (Однако он может вызывать сам себя — это случай рекурсивного программирования.) Подобные связи упрощают межмодульный обмен данными. Однако иногда возникает потребность активизировать модуль, расположенный несколькими уровнями ниже. Это разрешено, но в таком случае модуль должен быть указан в схеме несколько раз на соответствующих уровнях. Такие модули следует специально отмечать на схеме иерархии. Например, можно провести дополнительные вертикальные черточки.



Конечно, программируется такой модуль только один раз.

3. Принятие основных решений нужно выносить на максимально высокий уровень. Обычно основные решения принимает головной модуль (на первом уровне). Этот головной модуль служит кратким «конспектом» всей программы.

4. Модуль низшего уровня не должен принимать решения за модули высшего уровня. Для иллюстрации обратимся снова к аналогии со схемой организации. Вряд ли мы стали бы думать, что служащий будет принимать решения, которым будут следовать его управляющий, управляющий его управляющего или другие подразделения. Скорее он получит указания своего управляющего и доложит, может ли он успешно их выполнить. Таким образом, модуль не должен производить действий, непо-



средственно изменяющих порядок работы программы. Он не должен, например, активизировать подпрограмму своего (или более высокого) уровня. Он не должен определять, какая подпрограмма должна выполняться следующей, передавая явный адрес (т. е. используя конструкцию ИЗМЕНИТЬ языка КОБОЛ, оператор перехода по предписанию ФОРТРАНа или переменные типа метка в ПЛ/1) в программу уровнем выше или того же самого уровня. Независимо от того, что произошло при его выполнении, модуль должен всегда возвращать управление в активизировавший его модуль (т. е. в точку, откуда он был вызван). Модуль низшего уровня может передавать модулю высшего уровня характеристику полученных результатов или исследуемых условий. На основании этих данных модуль высшего уровня решает, что делать. Так, если модуль обнаруживает ошибку ввода, он передает сведения о ней в модуль высшего уровня, который и определяет, какой модуль (модули) вызвать.

*Вертикальное* управление обладает рядом преимуществ.

1. Логика программы становится понятнее.
2. При чтении головного модуля проявляется общая логика всей программы.
3. Программу проще изменять и пополнять.
4. Программирование и проверка вначале модулей высшего уровня, а затем низшего позволяет быстрее обнаруживать логические ошибки.

Модули нижнего уровня нужно детализировать только после определения всех подфункций модулей высшего уровня. Например, модуль ПОДГОТОВИТЬ на рис. 2.5 нужно *делить* только после выявления всех подфункций модуля ОБРАБОТАТЬ. Это помогает свести к минимуму пропуски или неполный анализ в функциях высшего уровня.

Каждый уровень прямоугольников в схеме иерархии представляет, как уже говорилось, некоторые модули. Например, несмотря на то что модуль ПОДГОТОВИТЬ был изменен за счет добавления четырех модулей более низкого уровня, он по-прежнему остался модулем, который будет в конце концов представлен некоторым фрагментом программы. В самом минимальном варианте этот модуль будет состоять лишь из активизаций соответствующих модулей третьего уровня.

### *Оценка схемы иерархии*

Разработчик схемы иерархии не может знать, представляет ли каждый прямоугольник достаточно обозримый фрагмент программы. Могут быть выделены такие подфункции, которые

либо настолько велики, что требуют дальнейшего деления, либо настолько малы, что неразумно их реализовывать в виде модуля. Вообще говоря, при разбиении на модули лучше сделать чуть больше требуемого, чем недостаточно продвинуться, так как объединять части функции совсем просто. Однако пропуск необходимых подфункций скорректировать позже очень трудно. Но даже такой пропуск не столь серьезен, как плохо определенные функции, составные функции и функции, оставшиеся рассредоточенными <sup>1)</sup>.

Для иллюстрации составных и рассредоточенных функций рассмотрим программирование алгоритма редактирования данных — элементов записей. Такое редактирование должно включать в себя проверку правильности содержимого цифрового или литерного поля, границ числовых полей, наличия специальных литер или вставку литер в выходные поля. Примером *составной функции* может служить модуль, редактирующий записи двух совершенно разных типов. Примером *рассредоточенной функции* может служить выделение двух модулей, в каждом из которых реализована только часть функции редактирования.

Аргументы следует передавать по возможности явно, а не через общую память. Это проясняет их использование в модуле и заставляет иметь не слишком много аргументов.

Чрезмерное количество передаваемых модулю *аргументов* (данных) может само по себе указывать на необходимость *разделения* функции. Если аргументов много, попытайтесь разделить модуль так, чтобы функция каждой части упростилась. При этом основная цель — сократить число аргументов или общих данных, которыми обмениваются модули.

Такая модульная структура, в которой число общих данных невелико, данные передаются явно, а управление подчиняется указанным в предыдущем пункте правилам, обладает тем преимуществом, что становится проще тестирование каждого модуля. Это достигается за счет того, что результаты работы модуля становятся легче предсказуемыми.

Хейни <sup>2)</sup> отмечает, что разработчик обязан сделать все возможное для уменьшения взаимозависимости модулей, потому что, когда у них много общих данных, любое изменение становится проблемой. Он рассказывает, как в изготовленную его сотрудником операционную систему задумали внести 296 *содержательных* изменений, в результате чего потребовалось из-

---

<sup>1)</sup> По поводу «рассредоточенных» функций см. Фуксман А. Л. Технологические аспекты создания программных систем.— М.: Статистика, 1979.— *Прим. ред.*

<sup>2)</sup> F. M. Haney, "Module Connection Analysis", Proceedings of the 1972 FJCC, AFIPS Press, pp. 173—180.

менять почти 3000 мест в системе <sup>1)</sup>). Таким образом, при сильной зависимости модулей попытка внести даже небольшое количество изменений может привести к самому неожиданному результату. Решению этой потенциальной проблемы может способствовать такое проектирование функций, при котором сводится к минимуму число общих данных. Именно поэтому полезно стремиться передавать данные через список аргументов и разрешать непосредственно взаимодействовать только модулям, распо-

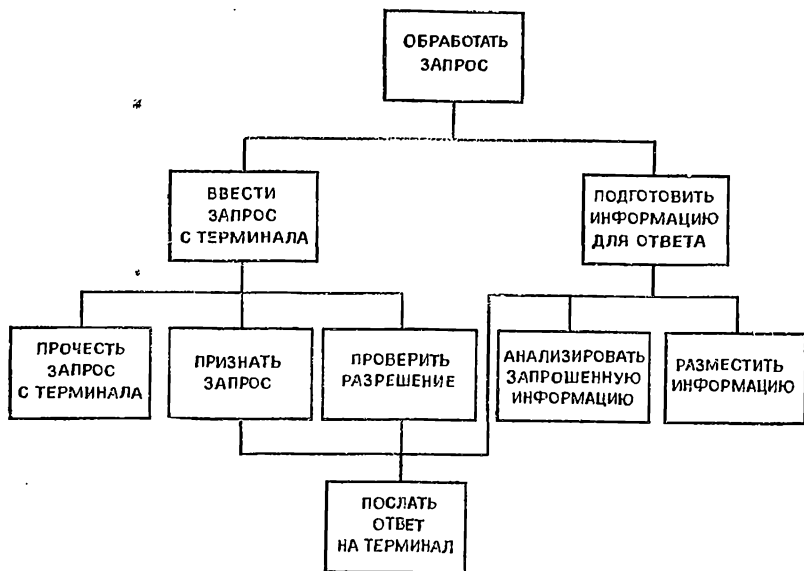


Рис. 2.6. Пересмотренная схема иерархии.

женным в схеме иерархии непосредственно один над (или под) другим.

Четкое разделение функций — это ключ к хорошим схемам иерархии, однако этого не всегда можно добиться. Тогда необходимо идти на компромисс и делать реальный вариант схемы иерархии максимально близким к идеальному.

Процесс разделения на подфункции каждого последующего уровня схемы иерархии заканчивается, когда все модули полностью спроектированы и дальнейшее разделение может привести лишь к рассредоточению функций.

У проблемы разбиения на модули не существует единственного решения. Например, схема иерархии, приведенная на рис. 2.5, может быть построена и так, как на рис. 2.6. Заметим, что

<sup>1)</sup> Richard G. Canning, "The Search for Software Reliability", EDP Analyzer, Vol, 12, No, 5, May 1974.

модуль ПОСЛАТЬ помещен на четвертый уровень, поскольку он активизируется несколькими модулями. Вместо передвижения модуля в схеме иерархии иногда легче его размножить. Вполне допустимо вторично выделить в качестве модуля ту же функцию.

### Пример

В качестве примера применения рекомендаций и правил построения схемы иерархии рассмотрим задачу обновления последовательного файла, используемого для управления запа-

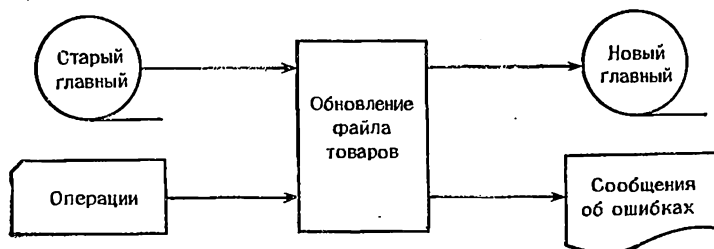


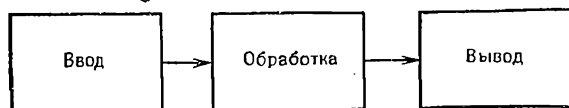
Рис. 2.7. Файлы данных в задаче управления запасами.

сами. Сведения об отдельных операциях (запросах) объединены и рассортированы в соответствии с номерами видов товаров. Имеются четыре типа операций, которым соответствуют следующие коды в записи об операции:

- 1 = Добавление в главный файл записи о новом виде товаров
- 2 = Приход (добавление новой партии товаров)
- 3 = Расход (продажа или вывоз со склада в связи с порчей и т. д.)
- 4 = Исключение существующей записи (товар снимается с производства)

*Главный файл товаров* — это последовательный файл, в котором хранятся упорядоченные записи. Файлы данных для этой задачи изображены на рис. 2.7. Старый главный файл и файл операций нужно сравнить и произвести обновление или слияние (в случае добавлений), в результате чего образуется новый главный файл. Кроме того, должны выдаваться сообщения о найденных программой ошибках.

С чего начать решение этой задачи? Следует рассмотреть три основные функции любой программы:



Первой должна рассматриваться функция вывода — *вывод определяет ввод*. Существует три типа выходных данных: записи нового главного файла, сообщения об ошибках и заключительные сообщения.

В этой задаче вывод не является *одной* функцией, поскольку первые два элемента порождаются при обработке, а последний — при завершении. Таким образом, вывод не может быть

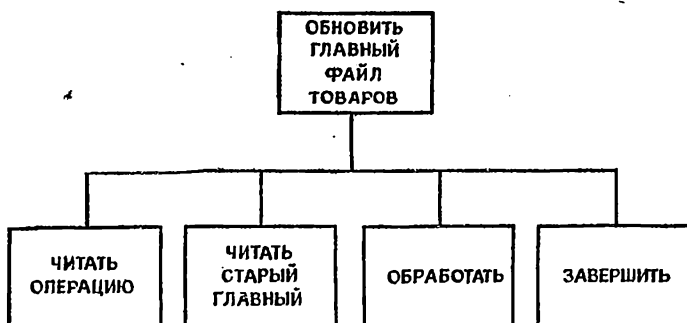


Рис. 2.8. Схема иерархии управления запасами — первая попытка.

одним модулем — иначе не получается четкого разделения функций.

Размышляя над функцией ввода, спросите себя, что должно вводиться. (Записи об операциях и записи старого главного файла.)

Первый вариант схемы иерархии показан на рис. 2.8. Головной модуль изображает главную функцию программы: ОБНОВИТЬ ГЛАВНЫЙ ФАЙЛ ТОВАРОВ. На втором уровне функция ввода изображается двумя модулями, ЧИТАТЬ ОПЕРАЦИЮ и ЧИТАТЬ СТАРЫЙ ГЛАВНЫЙ. Функция ЗАВЕРШИТЬ добавлена для закрытия файлов, выдачи заключительных сообщений и т. д.

Оправдано ли с точки зрения объема программы выделение ЧИТАТЬ ОПЕРАЦИЮ и ЧИТАТЬ СТАРЫЙ ГЛАВНЫЙ в отдельные модули? Быть может и нет, но сейчас не время думать о размерах модулей. Все внимание нужно сосредоточить на функциях и их иерархии. Небольшие модули при необходимости можно будет позже устранить, включив их как подфункции в другие модули.

На рис. 2.8 больше всего нуждается в развитии и разъяснении модуль ОБРАБОТАТЬ. Что же именно должно быть обработано? Операции в их взаимосвязи с главным файлом. Можно попытаться решать эту задачу, начав с проектирования модуля,

Таблица 2.1

**Разработка проверки соотношения между ключом операций  
и ключом главного файла**

Условие	Действие
ключ операции = ключ главного	1. Для ДОБАВИТЬ — напечатать сообщения об ошибке 2. Для ИЗМЕНИТЬ (приход или расход) или ИСКЛЮЧИТЬ — обновить или исключить запись главного файла
ключ операции > ключ главного	1. Скопировать запись старого главного файла в новый главный файл
ключ операции < ключ главного	1. Для ДОБАВИТЬ — дополнить новый главный файл 2. Для ИЗМЕНИТЬ или ИСКЛЮЧИТЬ — напечатать сообщение об ошибке

проверяющего соотношение между ключом <sup>1)</sup> операции и ключом записи главного файла. На основе этого анализа должны выполняться те или иные действия, определяемые табл. 2.1.

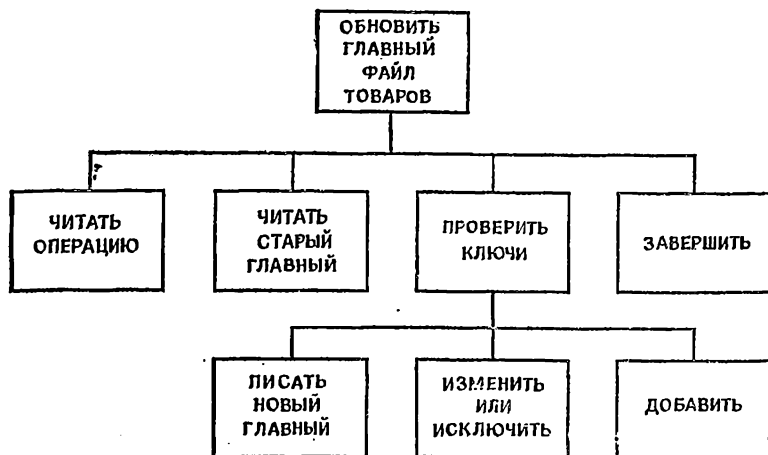


Рис. 2.9. Схема иерархии управления запасами — вариант 1.

На рис. 2.9 показано, как это можно отразить в схеме иерархии. Вполне ли удовлетворительно полученное решение? Нет, не вполне. Во-первых, некоторые вновь созданные модули третьего уровня объединяют несколько функций (например, модуль

<sup>1)</sup> Здесь *ключ* — это номер вида товаров. — Прим. ред.

ИЗМЕНИТЬ или ИСКЛЮЧИТЬ). Во-вторых, не учтено, что до проверки ключей необходимо принять некоторое решение более высокого уровня. Напомним, что основные решения должны приниматься на возможно более высоком уровне. В данном случае такое решение зависит от проверки кода операции. С нее надо начинать потому, что тогда программа естественно разделится на модули в соответствии с определяемыми этой проверкой функциями. Таким образом, ДОБАВИТЬ, ИЗМЕ-

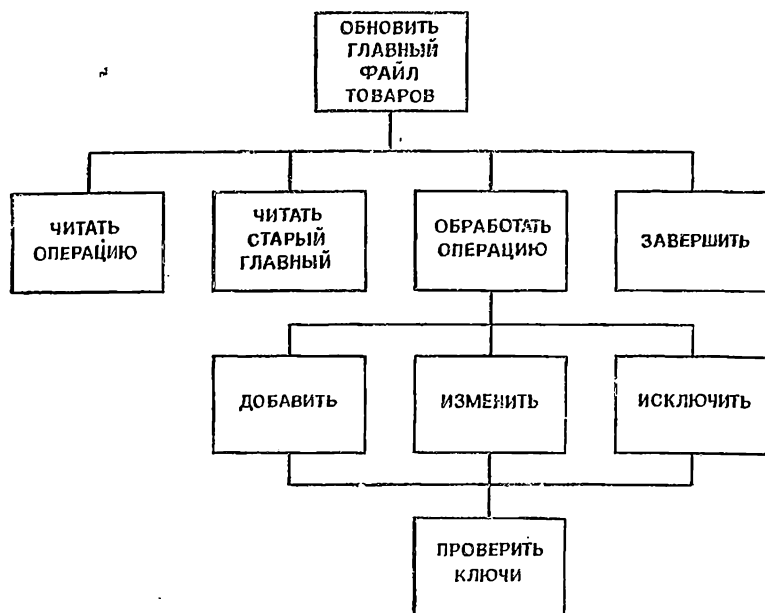


Рис. 2.10. Схема иерархии управления запасами — вариант 2.

НИТЬ и т. д. должны быть отдельными модулями. Это позволяет легко управлять последовательностью применения этих функций и расширять набор допустимых типов операций.

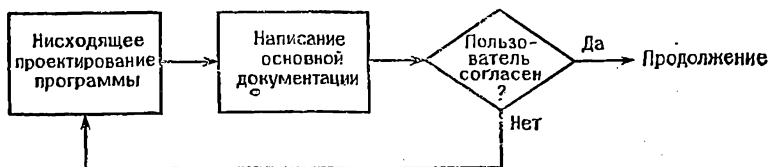
Результат показан на рис. 2.10, где на втором уровне есть модуль ОБРАБОТАТЬ ОПЕРАЦИЮ (включающий проверку кода операции). Развитие этого модуля приведено на третьем уровне. Здесь функции четко разделены, а управление простое и ясное. Модуль ПРОВЕРИТЬ КЛЮЧИ теперь на четвертом уровне. Однако он стал многократно используемым, причем результаты его работы в различных модулях интерпретируются по-разному. Этот модуль вызывается модулями ДОБАВИТЬ, ИЗМЕНИТЬ и ИСКЛЮЧИТЬ и передает в вызывающий модуль в качестве результата *индикатор*, указывающий, совпадают

ли ключи в записях операции и в главном файле. Конечно, *да* или *нет* имеют разный смысл в каждом из модулей третьего уровня. Так, например, для модуля ДОБАВИТЬ *да* означает ошибку, а для модуля ИСКЛЮЧИТЬ в этом случае все правильно.

Есть еще одна функция, не показанная на этой схеме — функция *копирования старого главного файла* в новый главный файл, если файл операций заканчивается до завершения просмотра старого главного файла. Такое копирование относится к завершающей функции, поскольку должно быть сделано лишь в конце программы. На рис. 2.11 она представлена модулем, который вызывается из модуля ЗАВЕРШИТЬ.

Разработав функции программы и их иерархию, можно перейти к этапам *планирования и реализации* (т. е. кодированию и тестированию) программы. Случается, что на этапе реализации, когда начинается подробное программирование, модули в этой схеме могут объединяться или разделяться на модули меньшего размера. Пусть, например, при программировании обнаружилось, что необходимо поместить на нижний уровень модули ЧИТАТЬ СТАРЫЙ ГЛАВНЫЙ и ПИСАТЬ НОВЫЙ ГЛАВНЫЙ, поскольку они могут активизироваться только модулями более высокого уровня. На рис. 2.12 показано новое место этих модулей, причем подчеркнуто, что встречаются они более одного раза (ниже модулей ПРОВЕРИТЬ КЛЮЧИ и КОПИРОВАТЬ СТАРЫЙ ГЛАВНЫЙ). На самом деле эти модули будут реализованы, конечно, только один раз, поскольку одна из задач схемы иерархии — показать подчиненность модулей; именно эта подчиненность и отражена с помощью повторного изображения модулей.

После завершения схем иерархии рекомендуется проверить их полноту и точность. Затем схемы должен просмотреть тот, кто будет пользоваться готовой программой (т. е. нужен структурный контроль). В некоторых организациях на этом же этапе готовятся документы, определяющие способ работы с программой. Эти документы предназначены для пользователя и также должны быть им просмотрены, чтобы достичь взаимопонимания до начала дальнейших разработок. Всю рассмотренную процедуру можно иллюстрировать следующим образом:





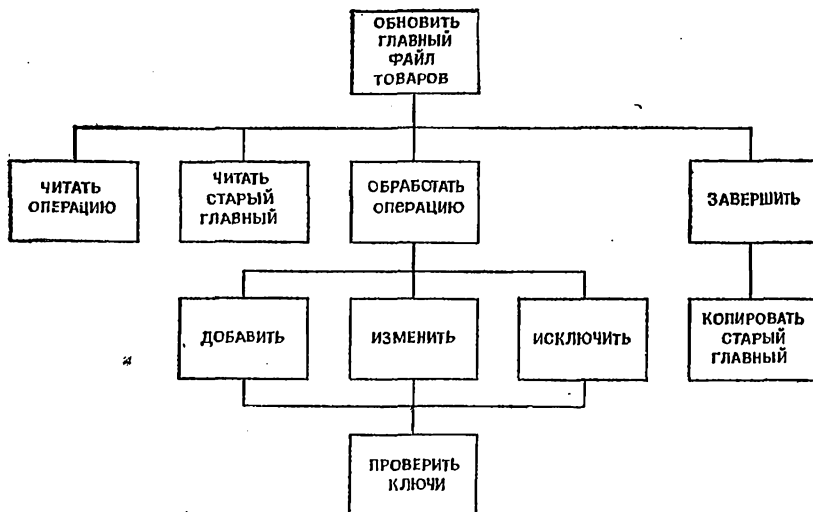


Рис. 2, 11. Схема иерархии управления запасами — вариант 3.

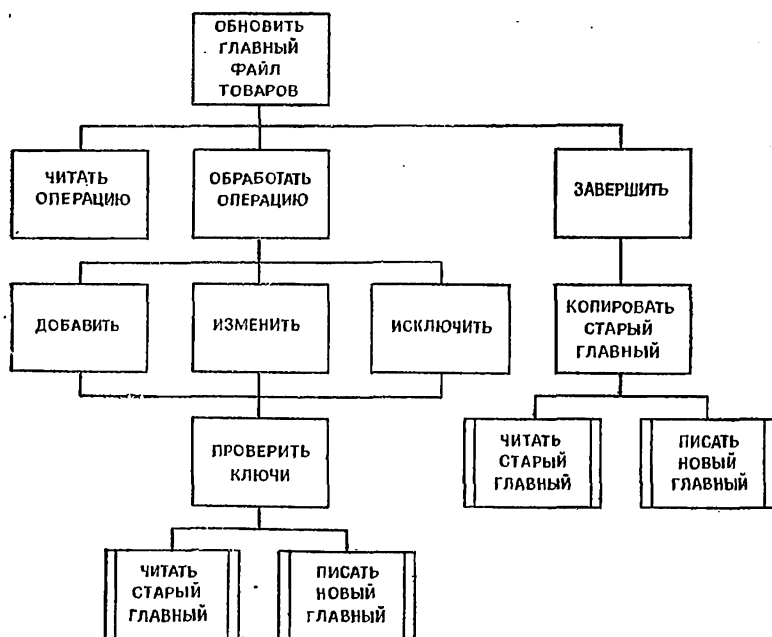


Рис. 2,12. Схема иерархии управления запасами — вариант 4.

Она легко помогает убедиться в том, что *требования и спецификации* понимаются обеими сторонами и достигнуто согласие по поводу желаемых результатов. Конец этапа проектирования характеризуется следующими условиями: 1) известно число модулей, 2) связь (функции и их иерархия) между модулями изображены схемами иерархии, 3) вся выполненная работа проверена пользователями на точность и полноту.

### Выводы для руководства

Успех нисходящей разработки зависит от умения руководства пользоваться этим инструментом. Без административной поддержки этот метод никогда не будет полностью освоен и внедрен в практику работы всех программистов организации. Но кроме умелого руководства нужны учебные программы и готовность следовать принятым в организации стандартам.

Для многих профессионалов переходный период будет не легким. Люди склонны работать старыми привычными методами. Новые идеи требуют переосмысливания и в некоторых случаях решительных изменений стиля работы. Даже если люди сами верят в новые идеи, им ведь еще нужно преодолеть сопротивление остальных.

Другая проблема в том, что большинство людей предпочитают иметь дело с хорошо определенными заданиями. Получив перечень дел, многие стремятся сначала выполнить те из них, которые относительно просты, не требуют больших затрат времени, и, что самое важное, те, про которые хорошо известно, как их сделать. Так, сотрудник может тратить кучу времени на заполнение формуляров, написание записок или отчетов, отвечать на телефонные звонки, читать или приводить в порядок оборудование. Он будет делать любое из этих дел вместо того, чтобы взяться за большую, сложную задачу, решение которой неочевидно, даже если эта сложная задача более важна.

В данном случае необходимо отложить такое хорошо определенное дело, как собственно программирование, и взяться за менее очевидную задачу — выявление модулей и определение их сопряжений.

### Выводы

В этой главе вводится одно из новых средств проектирования программ — схема иерархии. Разложение программы на составляющие ее модули требует умения переходить от общего к частному. Сначала определяются модули в соответствии с их функциями. При этом нужно руководствоваться принципом *один модуль — одна функция*. Затем модули объединяются в схему иерархии, основанную на понятии уровня абстракции. Так,

функция головного модуля — это конспект всей программы. Каждый последующий уровень определяемых модулей соответствует пониманию программистом уровней абстракции исследуемой задачи. Кнут <sup>1)</sup> соотносит талант к программированию и уровни абстракции следующим образом:

«Я давно чувствовал, что талант к программированию в основном заключается в способности легко переключиться от микроскопического к макроскопическому взгляду на вещи, т. е. без затруднений менять уровни абстракции. Я сообщил об этом Дейкстре <sup>2)</sup>, и он ответил <sup>3)</sup> блестящим анализом ситуации: «Я был неправ, когда утверждал, что наличие или введение «различных уровней абстракции» позволяет думать одновременно лишь об одном уровне, совершенно игнорируя остальные. Это неверно.

Вы пытаетесь упорядочить свои мысли, т. е. стремитесь организовать дело так, чтобы можно было сконцентрировать на некоторой части, скажем, 90 % ваших умственных усилий, в то время как остальное временно оказывается где-то на окраинах поля вашего внутреннего зрения. Но это не то, что подразумевается под словами «совершенно игнорируя»: вы позволяете себе временно опускать подробности, однако общие представления обо всем процессе продолжают играть важную роль. Вам остается следить за маленькими красными лампочками, которые временами начинают мигать в уголках ваших глаз».

Кроме функций и уровней абстракции схемы иерархии показывают подчиненность (или «кто кого активизирует»).

Наконец, при нисходящей разработке необходимо четкое понимание назначения программы. Необходима постоянная тесная связь с заказчиками. Эта связь должна осуществляться до, в течение и после этапа проектирования программы. Эффективным методом укрепления такой связи после этапа проектирования служит написание руководства по использованию программы. Это руководство и связанные с ним документы просматриваются пользователем до начала следующего этапа разработки программы.

### Контрольные вопросы и упражнения

1. Перечислите различия между системой, программой и модулем.
2. Приведите все характеристики модуля, которые вы помните.
3. Определите функцию.

---

<sup>1)</sup> Donald E. Knuth, "Structured Programming with go to Statements", Computing Surveys, Vol. 6, No. 4, December 1974, p. 292. Copyright 1974, Association for Computing Machinery, Inc., reprinted by permission.

<sup>2)</sup> Donald E. Knuth, "A Review of "Structured Programming", Stanford Computer Science Department Report STAN-CS-73-371. Stanford University, Stanford, Calif., June 1973, 25 pp.

<sup>3)</sup> Edsger W. Dijkstra, personal communication, January 13, 1973.

4. Каковы преимущества модуляризации, основанной на *функциях*? Каковы преимущества *вертикального* управления работой модуля?

5. Зачем ограничивать сложность модуля?

6. Каковы характеристики (достоинства и недостатки) различных методов ограничения сложности модуля?

7. Что вы думаете о правильности следующих утверждений:

а) Читая схему иерархии, следует предполагать, что модули выполняются сверху вниз и слева направо.

б) Если на втором уровне требуется вызвать модуль четвертого уровня, это не допускается, поскольку модуль может связываться лишь с модулем, расположенным непосредственно над (или под) ним.

в) Если подчиненный модуль обнаружил (при выполнении) ошибку, он может сам предпринять попытку ее исправить, изменив нормальную последовательность исполнения других модулей программы.

8. Приведите набор правил проектирования отдельных модулей.

9. Перерисуйте схему иерархии с рис. 2.12, чтобы модуль ИЗМЕНИТЬ был разделен на модули ПРИХОД и РАСХОД.

10. Нарисуйте схему иерархии для вашей жизни. Другими словами, решите, какова ваша главная цель. Это будет головной прямоугольник. Затем решите, что должно быть сделано для ее достижения. Продолжите деление на функции более низкого уровня.

11. Нарисуйте схему иерархии для функциональной организации автомобиля. Убедитесь, что она именно функциональная, а не процедурная, т. е. обратите внимание на действия, требующиеся для работы автомобиля, а не на порядок их выполнения.

12. Файл записей о служащих обрабатывается следующим образом: для каждого подразделения определяется математическое ожидание и дисперсия заработной платы и стажа работы. Вычисляется также математическое ожидание и дисперсия этих же показателей для всех служащих компании. Нарисуйте схему иерархии такой программы,

## Глава 3

# Нисходящая разработка: планирование и реализация

Нисходящую разработку можно рассматривать как процесс, состоящий из трех шагов:



Каждый шаг — это определенный этап жизненного цикла системы. Схема иерархии представляет собой ключевой документ, разрабатываемый на этапе проектирования. Это также и рабочий документ для следующего этапа — планирования. Важность планирования стала настолько очевидной, что теперь можно выделить его как отдельную и весьма специфическую деятельность в процессе разработки программы.

*Зачем планировать?* Когда есть план, все лучше организовано, люди работают эффективнее и продукт получается лучшего качества.

*Что планировать?* Оценки необходимого личного времени разработчиков и затрат машинного времени делаются в начале жизненного цикла системы и затем переосматриваются *плановиком* по мере перехода к следующим фазам проекта. Это обычная задача управления, но администратор, вероятно, захочет получить первоначальные оценки требуемого человеческого и машинного времени для выполнения конкретного проекта от самого программиста.

Принципы определения этих двух ресурсов — и личного, и машинного времени — будут рассмотрены ниже в разделе «Выводы для руководства». Кроме ресурсов, следует планировать также и порядок программирования и тестирования модулей.

## Планирование

### *Планирование порядка разработки модулей*

До начала программирования модулей необходимо сравнить различные последовательности программирования и тестирования модулей. Конечно, все эти последовательности должны

укладываться в рамки нисходящего подхода, но и в таких рамках возможно несколько вариантов. Например, сначала запрограммировать все модули одного уровня, а затем перейти к следующему, либо программировать все модули на одной ветви (т. е. от самого верхнего до самого нижнего модуля) схемы иерархии. Выбирать нужно последовательность, которая обеспечивает наиболее полное тестирование модулей, позволяя как можно раньше решить главные проблемы. Наилучшая последовательность разработки модулей получается комбинацией двух подходов: иерархического и операционного. Мы рассмотрим отдельно каждый из них, а затем покажем, как следует объединять эти два метода, чтобы получить оптимальную последовательность кодирования и тестирования. Последнее также необходимо планировать. Здесь мы совсем немного поговорим о тестировании и его влиянии на порядок разработки модулей, более подробно этот вопрос рассмотрен в гл. 10.

**Иерархический подход.** При этом подходе порядок программирования и тестирования модулей определяется их расположением в схеме иерархии. Сначала программируются и тестируются все модули одного уровня, после чего происходит переход на уровень ниже. На рис. 3.1 этот подход иллюстрируется на примере обработки запросов (операций), о котором уже шла речь в гл. 2. Числа над прямоугольниками указывают порядок разработки модулей. В этом поуровневом методе сначала программируется и тестируется головной модуль с применением заглушек для модулей второго уровня. Каждая заглушка — это упрощенная схема будущего модуля, содержащая все необходимое для того, чтобы было возможно пропустить полный тест модуля более высокого уровня.

После тестирования головного модуля заглушки второго уровня заменяются соответствующими модулями и тестируются с применением заглушек вместо модулей третьего уровня. Однако иерархический подход не определяет порядка создания модулей в пределах уровня. Например, на рис. 3.1 на третьем уровне можно принять не порядок «слева направо», а, скажем, АНАЛИЗИРОВАТЬ, ПРОВЕРИТЬ, РАЗМЕСТИТЬ. Наилучший порядок определяется удобством тестирования. Так, приведенная выше последовательность могла бы быть выбрана именно потому, что модуль ПРОВЕРИТЬ использует данные, создаваемые или изменяемые модулем АНАЛИЗИРОВАТЬ. Поскольку ПРОВЕРИТЬ использует результаты выхода АНАЛИЗИРОВАТЬ, первый из этих модулей должен разрабатываться после второго. Необходимость использовать в модуле данные, подготовленные или измененные другим модулем, называется *зависимостью по данным*.

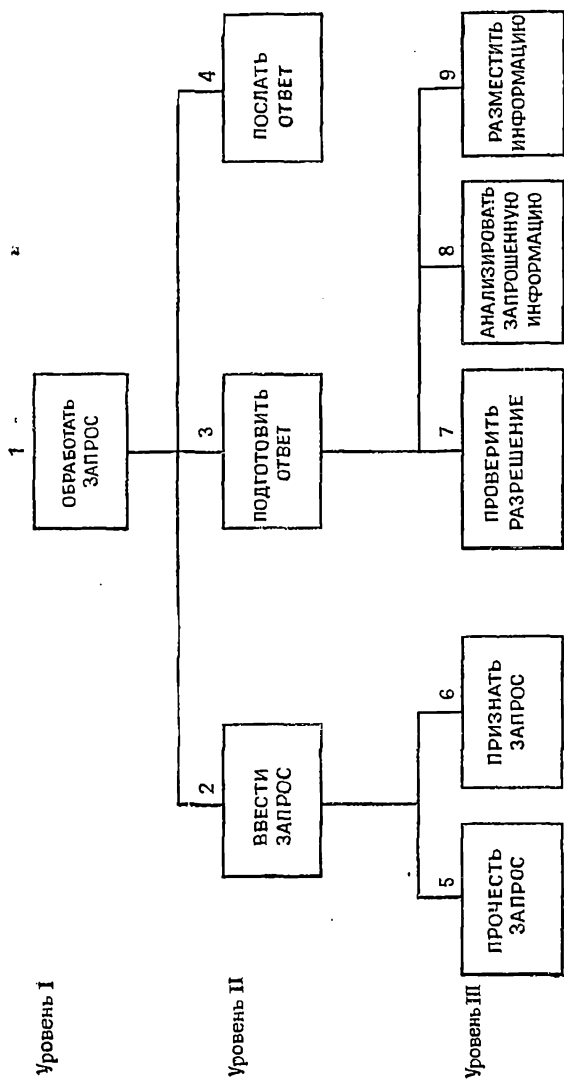


Рис. 3.1. Порядок разработки программы обработки запросов — иерархический подход.

Зависимость по данным может существовать и между модулями разных уровней. Например, ПОДГОТОВИТЬ может зависеть от данных, вводимых модулем ПРОЧЕСТЬ. Если используется поуровневый метод, то модуль ПОДГОТОВИТЬ нельзя тестировать полностью — он пишется раньше модуля ПРОЧЕСТЬ. Одно из возможных решений — переработать схему иерархии так, чтобы модули ПРОЧЕСТЬ и ПОДГОТОВИТЬ находились на одном уровне. Такая организация может, однако, оказаться нелогичной или получится схема иерархии с неоправданно большим количеством модулей на одном уровне. Другое решение — написать заглушку для ПРОЧЕСТЬ, содержащую необходимые операторы ввода, а затем программировать и тестировать модуль ПОДГОТОВИТЬ с этой заглушкой. Если, однако, последняя оказывается сложной, то, вероятно, лучше запрограммировать модуль целиком, чем тратить время на сложную заглушку. Таким образом, здесь может оказаться желательным отступить от строгого иерархического подхода.

Такое отступление *не* является нарушением принципа «сверху вниз». Нисходящая разработка означает, что модули верхнего уровня пишутся и объединяются ранее подчиненных *им* модулей. Из рис. 3.1 видно, что модуль ВВЕСТИ пишется ранее подчиненных ему модулей. Если одна ветвь схемы иерархии целиком разрабатывается ранее модулей высшего уровня (из других ветвей), то мы повышаем качество тестирования и, скорее всего, сокращаем затраты на программирование.

Используя иерархический подход, не нужно забывать о двух вещах:

1. Зависимость по данным, или обмен данными между модулями, может затруднить программирование и тестирование при поуровневом методе.
2. Слепое следование поуровневому подходу приводит к реализации основной массы модулей в конце проекта. Это может затруднить распределение машинных и человеческих ресурсов.

**Операционный подход.** При этом подходе модули разрабатываются в порядке их выполнения при запуске готовой программы (рис. 3.2). Точный порядок разработки модулей определяется *плановиком*, который мысленно выполняет программу. Однако порядок исполнения обычно меняется с изменением входных данных. Таким образом, как и для иерархического подхода, здесь остается свобода при выборе последовательности программирования и тестирования.

Одно из преимуществ операционного подхода в том, что минимизируются трудности, вызванные зависимостью по данным. Поскольку модули обычно добавляются в порядке исполнения, то сначала будут, как правило, писаться модули, порождающие



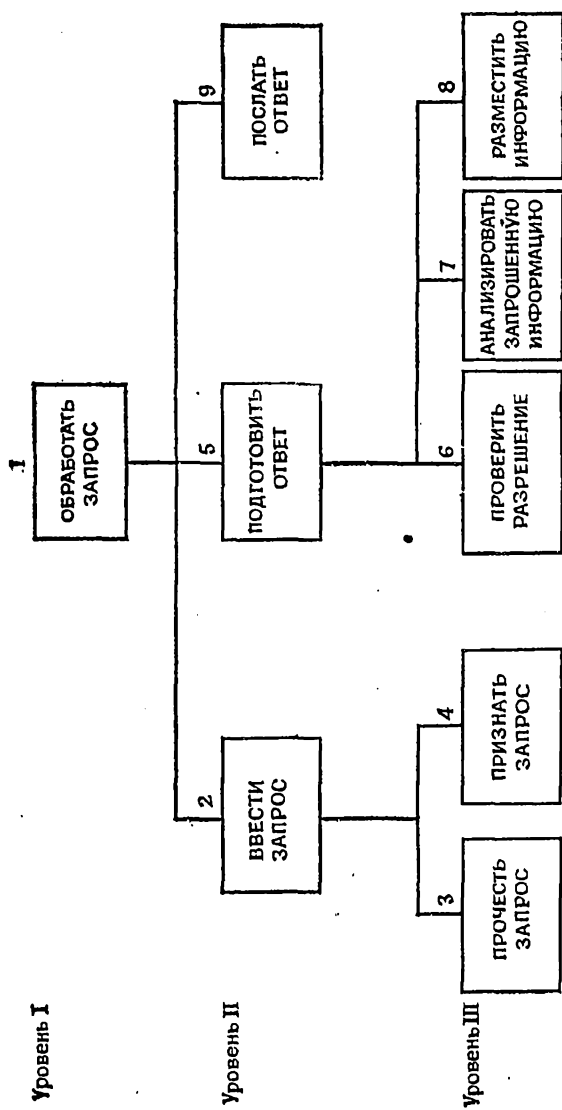


Рис. 3.2. Порядок разработки программы обработки запросов — операционный подход.

данные для последующих модулей. Недостаток этого подхода в том, что при такой последовательности программирования и тестирования разработчики модулей «выстраиваются в очередь». Так, программист, пишущий модуль ПОСЛАТЬ на рис. 3.2, должен ждать окончания работы всех остальных программистов. Из-за этого могут возникнуть проблемы с расписанием работ и своевременной реализацией всех модулей, находящихся на *критическом пути*.

Второй источник трудностей связан с техническими аспектами операционного подхода. Например, на рис. 3.2 предполагается, что весь вывод в программе осуществляет модуль ПОСЛАТЬ. Если строго следовать операционному подходу, этот модуль будет писаться последним.

Таким образом, при разработке всех остальных модулей отладочная печать будет осуществляться заглушкой. Здесь снова требуется большая программа-заглушка, пропадающая зря при замене ее на соответствующий готовый модуль. Есть несколько способов решения этой задачи:

1. Перерисовать схему иерархии.
2. Применить специальные подпрограммы отладочного вывода, написанные заранее и приспособленные для использования без значительных переделок.
3. Программировать в заглушке простые операторы вывода (такие, как PUT DATA в ПЛИ).
4. Отступить от строго операционного подхода.

Первый способ может потребовать много времени и привести к низкому качеству результата. Второй и третий способы не всегда достижимы, чаще всего используется четвертый. В нашем случае это означает, что модуль ПОСЛАТЬ будет программироваться раньше, возможно, непосредственно после модуля ОБРАБОТАТЬ. Дополнительный выигрыш состоит в том, что пользователь может раньше ознакомиться с примером работы программы. (Когда пользователь видит выдачу, он, как правило, восклицает: «Это совсем не то, что нужно!» Если ситуация именно такова, то обнаружить это следует как можно быстрее.)

Другой случай отступления от операционного подхода связан с вспомогательными программами. Они выполняют такие действия, как закрытие файлов, печать итогов и сообщений об окончании работы. Предположим, что в нашей системе есть другой модуль, расположенный на том же уровне, что и ПОСЛАТЬ, но выполняющийся позже. Его функция — закрыть файлы и т. д. Для беспрепятственного тестирования других модулей, использующих файлы, его следует написать вне очереди.

**Комбинированный подход.** Наилучший подход при разработке модулей — комбинация иерархического и операционного подходов. Здесь плановик взвешивает достоинства каждого подхода, определяя конкретную последовательность разработки модулей. Конечно, для одной и той же схемы иерархии возможны различные последовательности. Однако, чтобы не выйти за рамки нисходящей технологии, перед программированием каждого модуля необходимо проверять следующие условия:

1. Должен существовать путь управления к этому модулю, т. е. такая цепочка модулей, уже вошедшая в работающую программу, через которую управление может быть передано новому модулю.
2. Должны быть доступны значения всех данных, требуемых от модулей (или их заглушек), создающих или изменяющих эти данные.

Определяя порядок разработки модулей, следует учитывать четыре фактора: 1) зависимость по данным, 2) доступные ресурсы, 3) требование обеспечить выдачу результатов модулей, проверяемых раньше, 4) необходимость прежде всего обеспечить готовность вспомогательных модулей. Пятый фактор — это *сложность*.

Соблазнительно заниматься сначала простыми модулями, чтобы больше о них не думать. Однако обычно следует начинать со сложных модулей. С течением времени они проще не становятся, а их позднее программирование может привести к двум проблемам:

1. Они могут оказаться еще сложнее и потребовать для разработки больше времени.
2. Причиной сложности может быть то, на что не обратили должного внимания или неадекватно определили. Раннее программирование сложного модуля обнаружит это. Так, на рис. 3.2 сложным окажутся, вероятно, модули АНАЛИЗИРОВАТЬ и РАЗМЕСТИТЬ — их нужно писать раньше.

Шестой фактор — это *обработка исключительных ситуаций*, связанных с неправильными данными. Модули, обрабатывающие правильные данные, должны программироваться и тестироваться раньше модулей, имеющих дело с неправильными данными. Так, модули ПРИЗНАТЬ и ПРОВЕРИТЬ следует программировать и тестировать позже, поскольку они разрабатываются для обнаружения ошибок во входных данных.

Не существует общего правила учета этих факторов в конкретной программе. Более того, на практике они часто противоречат друг другу. Каждый случай нужно рассматривать инди-

видуально, внимательно оценивая все факторы. Вот почему нисходящее планирование требует времени и способностей.

Один из способов установить приемлемый порядок разработки — мысленно выполнить программу с учетом максимального количества подробностей. Этим должны заниматься по крайней мере два человека. Один объясняет другому (или другим), что происходит на каждом этапе выполнения программы и что должен делать каждый модуль (т. е. функцию модуля). Вообще говоря, лучше начинать с операционного подхода, чем с иерархического, поскольку первый подход позволяет убедиться, что и критерий управления, и критерий доступности данных выполняются для каждого модуля.

Процесс мысленного выполнения программы может выявить недостатки в проектировании модульной структуры или в документации на модуль. (С каждым модулем должно быть связано описание его зависимостей по данным.) При ошибке в проектировании следует пересмотреть схему иерархии, при неполной документации или неправильных спецификациях необходимо исправить недочеты до перехода к следующим этапам разработки модуля.

Чтобы вообще что-нибудь проверить, необходимо запрограммировать и выполнить нечто большее, чем только головной модуль. Эта совокупность модулей образует *ядро* программы. Ядро следует проверить, используя при необходимости заглущки. Для оставшихся модулей порядок должен определяться заботой о полноте тестирования и минимизации дополнительной работы.

На рис. 3.3 показан возможный порядок разработки, использующий комбинированный подход. При выборе именно такого порядка учитывались перечисленные выше факторы. Так, модуль ПОСЛАТЬ реализуется самым первым, чтобы обеспечить вывод. Модули ПОДГОТОВИТЬ, АНАЛИЗИРОВАТЬ и РАЗМЕСТИТЬ реализуются вслед за ним — они наиболее сложные. Модуль РАЗМЕСТИТЬ следует за АНАЛИЗИРОВАТЬ, поскольку использует создаваемые этим модулем данные. Последними реализуются ПРИЗНАТЬ и ПРОВЕРИТЬ, поскольку они простые и обрабатывают ошибки при вводе.

**Отклонения от нисходящей последовательности.** В некоторых случаях нисходящую последовательность реализовывать неразумно. Эти ситуации перечислены в табл. 3.1 вместе с возможными альтернативными решениями.

Из этой таблицы мы видим, что для первого затруднения нисходящее решение состоит в том, чтобы перерисовать схему иерархии. Здесь могут возникнуть два препятствия: во-первых, исходная схема может быть более логичной и понятной, во-вторых, при переделке схемы возникает тенденция собрать на вто-

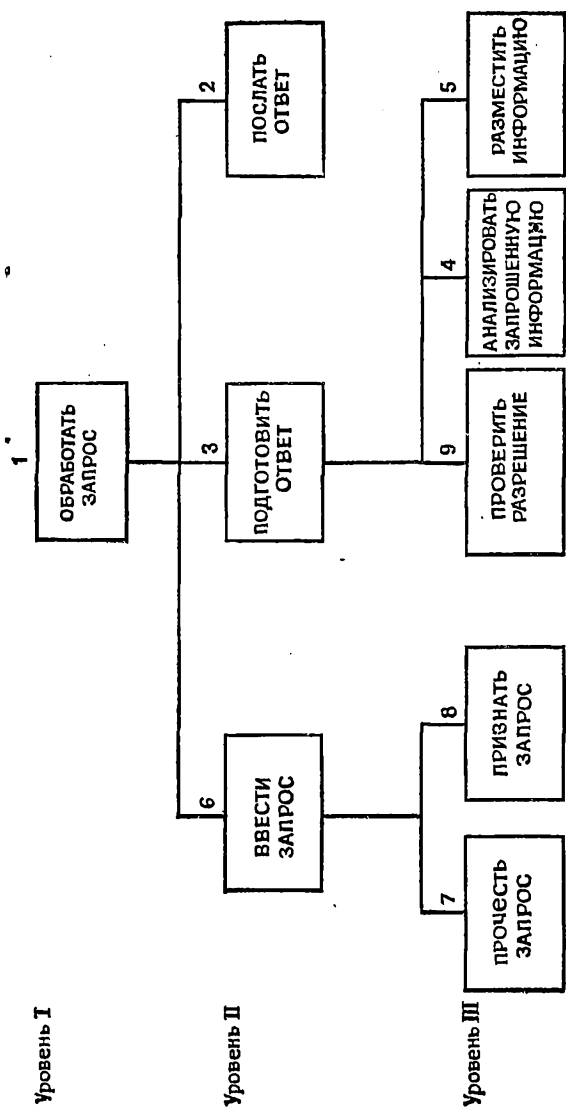


Рис. 3.3. Порядок разработки программы обработки запросов — комбинированный подход.

Таблица 3.1

## Возможные отклонения от нисходящей последовательности

Затруднение	Решение, нарушающее нисходящую последовательность	Нисходящее решение
I. Модуль нижнего уровня активизируется несколькими модулями более высокого уровня	Программировать и тестировать раньше, поместить в библиотеку подпрограмм или библиотеку исходных текстов	Перерисовать схему иерархии так, чтобы этот модуль оказался на более высоком уровне
II. Спецификация модуля постоянно изменяется	Отложить программирование и тестирование	1. Программировать модуль, предполагая, что изменения незначительные 2. Перерисовать схему иерархии
III. В начале проекта имеется больше людских ресурсов, чем в конце	1. Программировать параллельно, даже если не закончены вызывающие модули 2. Договориться с руководством о перераспределении рабочей силы	1. Не использовать в начале дополнительную рабочую силу, хотя проект будет делаться дольше 2. Договориться с руководством о перераспределении рабочей силы

ром уровне слишком много модулей. В этих случаях решение не использовать метод «сверху вниз» может быть предпочтительнее. При втором затруднении переходить к программированию, не дожидаясь фиксации спецификаций, оказывается выгодным потому, что это заставляет все-таки определить спецификации раньше, так что остается время при необходимости уточнить их. При третьем затруднении строгий нисходящий подход противоречит ситуации. Программист должен быть уверен, что руководство понимает сложившуюся ситуацию и осознает опасность задержки и дополнительных затрат.

*Планирование тестов*

Вместе с планированием порядка, в котором следует разрабатывать модули, необходимо планировать также и разработку тестов для всей программы. Прежде всего нужно определить типы тестов. Например, один тест может предназначаться для работы программы без учета ошибочных ситуаций. Здесь можно использовать набор правильных данных. Для первоначальной проверки модуля бывает достаточен набор из 3—5 записей данных. Затем следует использовать набор из 100—250 записей. Разумеется, для заключительного тестирования необходимо взять полный набор реальных данных.

Другой тест может содержать данные, не являющиеся типичными для разрабатываемой программы. Еще один тест может содержать неверные данные (например, отрицательные вместо положительных или числовые вместо буквенных). Когда, например, нужно работать с файлами, содержащими записи с *ключами*, следует иметь тесты с одинаковыми, несуществующими и неправильными ключами, а также с ключами неправильной длины. Для каждого планируемого теста следует заранее определить ожидаемый результат.

Зачем готовить тесты до начала программирования? В пользу такого решения можно привести несколько доводов:

1. Устраняется возможность подгонки тестов к уже написанной программе. Уменьшается возможность внесения одних и тех же упущений или ошибок и в программу, и в тестовые данные.
2. Лучше понимаются ограничения на входные данные, поскольку сопряжение многих модулей зависит от того, что содержится в данных.
3. Это заставляет принимать решения и вносить ясность на более ранних этапах разработки.
4. С самого начала обращается внимание на представление о завершенной программе. Готовя тесты вначале, программист должен представить себе окончательный результат, получаемый программой.
5. Обеспечивается непрерывность этапов создания программы: сначала определяются входные данные, затем они используются при разработке, затем выполняется готовая программа, давая ожидаемые результаты.
6. Тесты, подготовленные до программирования, лучше проверяют программу.

Иногда необходимо изменить или пополнить набор тестов. Это не означает отхода от ранее определенных тестов и пересмотра этих тестов по мере продвижения проекта. Речь идет просто о ликвидации упущений и исправлении ошибок в ранее подготовленном полном наборе тестов.

Подготовить законченный набор тестовых данных перед началом программирования можно не всегда. Например, в большом проекте на сбор всех данных, необходимых для тестирования, могут потребоваться недели. Этой работой занимаются обычно лишь несколько человек. Что делать остальным? Разумное распределение людских ресурсов таково: программировать головные модули одновременно с разработкой тестов, предполагая, что тесты будут готовы к первому запуску модуля. Хотя здесь мы отступаем от принципа «сначала планирование, потом

программирование», это необходимо для оптимального использования ресурсов.

В некоторых организациях используются или создаются программы генерации наборов тестовых данных. Такие *генераторы тестов* используют объявления данных (т. е. РАЗДЕЛ ДАННЫХ в КОБОЛе или DECLARE в ПЛ/1) тестируемой программы. При таком подходе сначала следует писать объявления данных. Таким образом, частично программирование предшествует созданию тестов. Однако при этом необязательно писать «выполняемую» часть программы.

**Кто готовит тесты?** За подготовку тестов должны отвечать опытные и компетентные люди, знакомые с областью применения программы. Следует всегда привлекать к этой работе и пользователя программы, поскольку он часто может указать ситуации, неизвестные программистам. Кроме того, он может помочь также и при определении ограничений на входные данные. К этому делу могут привлекаться также системный аналит и руководитель проекта. Другими словами, тесты должны готовиться не тем человеком, который пишет программу, чтобы исключить взаимовлияние этих процессов. Это, конечно, не означает, что программист, разрабатывая программу, совсем не должен думать о возможных тестах.

## Реализация

*Этап реализации* включает фактическую подготовку тестов, создание и использование нужных текстов на *языке управления заданиями* (JCL) и, наконец, программирование и тестирование ядра и других модулей или заглушек.

**Подготовка тестов.** Подготовка тестов включает знакомство с исходными документами, содержащими спецификации элементов данных, и нанесение данных на карты, ленту, диск или непосредственный ввод данных в машину с помощью диалогового терминала.

**Язык управления заданиями (JCL).** Следующий шаг — определить и запрограммировать полный текст на JCL, нужный для выполнения готовой программы. Этот текст будет использован при программировании и тестировании для создания той операционной обстановки, в которой выполняется программа. (Здесь снова внимание привлекается к готовой и уже выполняющейся программе.) Для тестирования подготовленного текста на JCL необходимо выполнить пустую программу — заглушку головного модуля. Преимущество предварительной подготовки полного текста на JCL состоит в том, что это заставляет вникать



в детали выполнения программы на ранней стадии разработки. Эти детали могут касаться спецификаций нужных файлов, размеров требуемой вторичной памяти, количества и размера внутренних буферов и т. п. Ранняя подготовка текста на JCL ускоряет программирование, помогая избежать упущений или недосмотров при определении данных, взаимодействия и связей модулей.

**Ядро.** Ядро может состоять из одного или нескольких модулей самых верхних уровней. Головной модуль должен быть частью ядра. Поскольку он уже был написан как заглушка для проверки корректности текста на JCL, логично расширять первым именно его. Если в ядро должны быть включены и другие модули, то программист должен заранее определить порядок их включения.

**Заглушки.** Чтобы проверять программу нисходящим способом, необходимо писать заглушки для тех еще не законченных модулей\* следующего уровня, которые активизируются имеющимися модулями. Это позволяет трассировать программу. Содержимое заглушек может быть разным. Заглушка может быть *пустой* и содержать только команды связи и немедленного возврата. Например:

ПЛ/I

STUB:
PROCEDURE;
END;

КОБОЛ

STUB.
EXIT.

Небольшое расширение позволяет заглушке печатать сообщение, содержащее ее имя, отмечая этим активизацию заглушки. Такое простое средство обеспечивает трассировку программы. С помощью вывода сообщений можно убедиться, что порядок и число активизаций модулей правильны. Некоторые компиляторы позволяют автоматически выдавать трассировку. В этом случае желательно воспользоваться этой возможностью, а не вставлять печатать сообщений в заглушку.

Заглушка может понадобиться для подготовки или модификации данных, используемых другими модулями программы. Например, модуль нижнего уровня может читать сообщение с терминала и проверять правильность его типа и формата. Вначале он может быть реализован как заглушка с сообщением в виде пустой записи. Эта запись передается в вызывающую подпрограмму. Обработка пустой записи осуществляется модулем более высокого уровня, он может, в частности, определить правильность переданной записи. Позднее заглушка будет рас-

ширена до фактического чтения сообщения с терминала и выполнения необходимых проверок.

В качестве другого примера рассмотрим модуль, читающий список передаваемых ему элементов данных и сортирующий их в соответствии с лексикографическим порядком с учетом только первых десяти литер. Его можно заменить заглушкой, игнорирующей пересылаемый ей список ввода и возвращающей список, содержащий заранее отсортированные элементы. Или рассмотрим модуль, читающий входные записи и обновляющий уже существующий файл. Для каждой входной записи последовательно проверяется попадание элементов в заданные диапазоны. Заменяющая его заглушка могла бы читать небольшой файл, в котором все записи содержат данные, уже лежащие в соответствующих диапазонах. Это примеры заглушек, выполняющих часть фактических функций по обработке данных, выполняемых готовым модулем. Такую часть можно реализовать программой всего в несколько строк, которые могут остаться, а могут и исчезнуть при расширении заглушки до полного модуля.

Мы уже установили, что модули низкого уровня не должны управлять работой модулей высокого уровня. Точнее говоря, они могут возвращать в вызывающий модуль указания о том, что именно произошло при их выполнении. Таким образом, подпрограммы связываются передаваемыми элементами данных или индикаторами, так что функции модулей по обработке этих элементов разделены. Когда некоторый модуль заменяется заглушкой, указанную связь необходимо учитывать. Некоторые заглушки приходится писать так, чтобы установка выдаваемых ими индикаторов обеспечивала правильную работу зависящей от них подпрограммы.

В качестве примера рассмотрим программу, модули которой читают и обрабатывают записи некоторого файла. Вызывающая подпрограмма может содержать цикл для повторной активизации *модуля чтения* до получения индикатора, свидетельствующего об окончании файла. Если такой индикатор не устанавливается в заглушке модуля чтения, вызывающая программа будет продолжать активизации бесконечно. Поэтому заглушка должна программироваться так, чтобы, прочитав несколько тестовых записей, установить индикатор конца файла. Таким образом, цикл рано или поздно закончится, причем будут проверены как активизирующая подпрограмма, так и подпрограмма, завершающая обработку файла.

Таким же образом следует отражать в заглушке любую ошибку или необычную ситуацию, которую может обнаружить модуль. Эти ситуации нужно имитировать, передавая только признаки их возникновения. Такой тип заглушек позволяет полностью

протестировать верхние модули. (Когда функции четко разделены, число индикаторов, передаваемых в вызывающий модуль, должно быть сравнительно невелико.)

### **Выводы для руководства**

Нисходящая разработка программы или системы позволяет лучше маневрировать ресурсами, чем традиционные методы проектирования. В последнем случае с проектом связано одно и то же количество исполнителей с начала до конца работы. Это приводит к их неэффективному использованию. Часто в конце проекта требуется огромное количество машинного времени для второго и третьего просмотра всех этапов разработки проекта. При нисходящем проектировании человеческие и машинные ресурсы распределяются совершенно по другим принципам.

**Распределение рабочей силы.** Фронт работы в проекте расширяется по мере его развития. В начале реализации необходимо лишь несколько человек для написания модулей высшего уровня. При программировании большего числа модулей следующих уровней к работе над проектом привлекаются новые люди. Преимущество нисходящей разработки проявляется в том, что она позволяет прогнозировать увеличение численности исполнителей.

Независимость модулей позволяет делить их между несколькими программистами и одновременно разрабатывать разные части проекта. Харлан Милс предлагает считать, что «разумный» объем работы для одного программиста равен 2000 готовых предложений. Таким образом, одному программисту можно поручить несколько больших модулей (например, одну ветвь) в схеме иерархии или, скажем, 20 небольших модулей.

При нисходящей разработке большая часть общих затрат времени уходит на планирование, а меньшая — на реализацию. Даже для большого проекта увеличение времени на планирование означает дополнительную затрату времени лишь несколькими людьми. Нисходящее планирование требует не менее двух человек: один несет ответственность за план целиком и еще один или несколько человек отвечают за выбор тестов и планирование тестирования.

**Распределение машинного времени.** При восходящей разработке в конце проекта объем работы на машине значительно увеличивается. На рис. 3.4 приведены типичные примеры использования машины при двух подходах. Нисходящий подход применялся при разработке проекта «Скайлэб», где затраты ма-

шинного времени оставались постоянными с девятого по двадцать четвертый (последний) месяцы работы над проектом <sup>1)</sup>).

Суммарные затраты машинного времени при нисходящей разработке обычно не больше, довольно часто они меньше. Сначала может показаться, что требуется дополнительное ма-

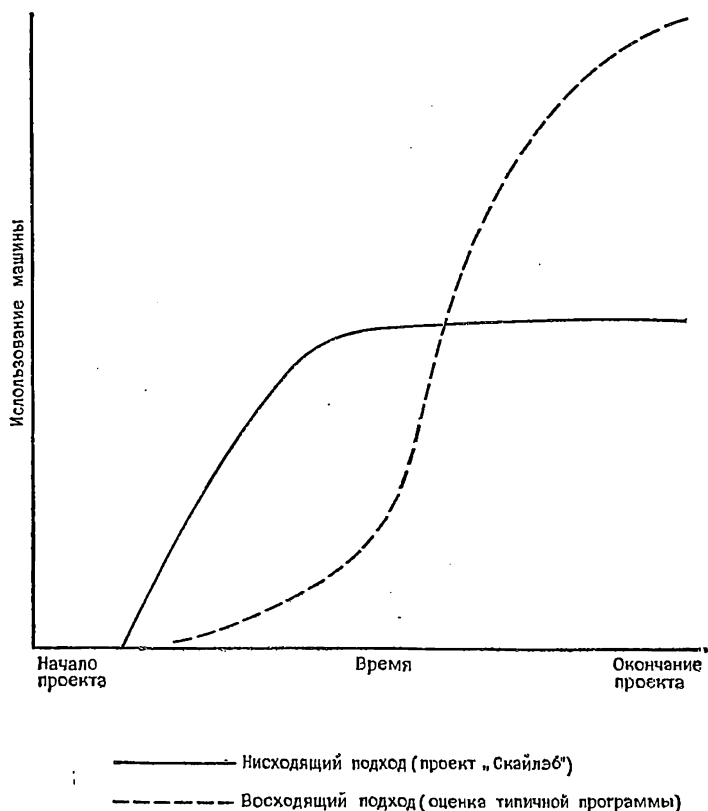


Рис. 3.4. Сравнение использования машинного времени.

шинное время, поскольку (1) период тестирования в значительной степени перекрывается с этапом реализации или (2) тестирование одного модуля на верхнем уровне иерархии требует исполнения многих других модулей. Однако это увеличение затрат машинного времени на ранних стадиях проекта компен-

<sup>1)</sup> F. T. Baker and H. D. Mills, «Chief Programmer Teams», Datamation, December 1973. Reprinted with the permission of DATAMATION® Copyright 1973 by Technical Publishing Company, Greenwich, Connecticut 06830.

сируется значительным сокращением потребностей в машине на более поздних стадиях. Таким образом, здесь нет обычного пика в конце проекта.

Раннее тестирование проверяет правильность программы, особенно подпрограмм высшего уровня, отражающих глобальные решения и осуществляющих общее управление. Нисходящее проектирование, планирования и реализация — это подход, требующий хорошей подготовки, понимания, времени и опыта. Однако выигрыш от своевременного (или даже досрочного) завершения проекта, от высокой надежности полученной программы, уменьшения стоимости ее эксплуатации и сопровождения в несколько раз превосходит первоначальные затраты на нисходящую разработку.

## Выводы

Суть нисходящей разработки в том, что реализации должно предшествовать тщательное планирование. Однако очень трудно не только заставить себя отложить программирование ради планирования, но и трудно планировать по-настоящему. В самом начале к тому же отсутствуют знание и опыт нисходящего планирования. Первые попытки будут успешнее, если предварительно изучить уже использованные методы и экспертные оценки, но все-таки планирование скорее всего останется трудной и требующей немало времени задачей.

Нисходящая разработка требует более полного проектирования модулей и анализа их зависимости по данным. Планирование заглушек и процедур тестирования требует большого внимания к деталям, чтобы убедиться в реальности и полноте выбранного набора тестов. Это особенно справедливо для достаточно сложных программ, где много модулей, сопряжений и зависимостей по данным. Именно в этих случаях планирование особенно важно. Неадекватное планирование может привести к перепроектированию, перепрограммированию и перетестированию, что потребует дополнительного времени и расходов.

На рис. 3.5 показаны все основные этапы планирования и реализации при разработке программы. На стадии планирования устанавливается порядок разработки и тестирования модулей. Этот порядок может не следовать строго ни уровням схемы иерархии, ни порядку исполнения — это обычно комбинация иерархического и операционного подходов. На этом этапе особенно важно глубокое знание межмодульных взаимосвязей. Затрата времени и сил на план работы по программированию и тестированию обычно окупается отсутствием больших задержек и рас-

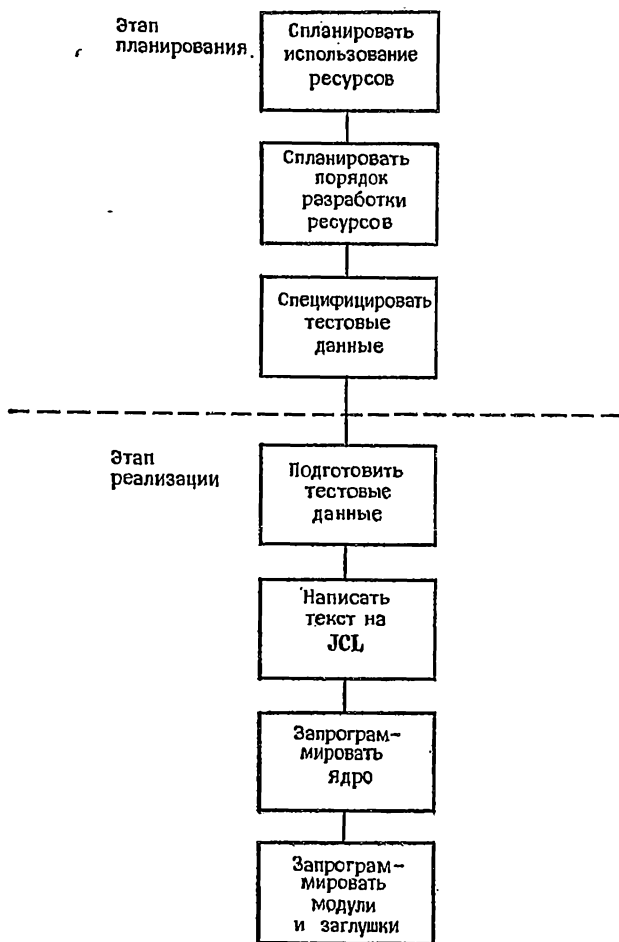


Рис. 3,5. Шаги нисходящего проектирования.

ходов, вызванных неадекватным проектированием или тестированием. Эффективное использование ресурсов и подготовку тестов также нужно планировать до начала программирования.

На этапе реализации самая важная задача — получить законченную программу. Поэтому до начала программирования ядра необходимо подготовить тесты и полный текст на языке управления заданиями.

Сначала нужно создать ядро, а затем постепенно добавлять модули, расширяющие его возможности. При тестировании этих

модулей потребуются заглушки вместо еще не написанных модулей более низкого уровня. Сложность заглушки колеблется от простого оператора возврата до заготовки будущего модуля, порождающей данные для тестирования вышестоящего модуля. Модули нужно добавлять по одному, возникающие несоответствия обнаруживать и исправлять до перехода на следующий уровень. Когда заглушка расширяется до модуля, возникает потребность в новых заглушках уровнем ниже и процесс повторяется. И все время повторяется тестирование. Каждый новый раздел программы нужно тестировать вместе с ранее подключенными и выполнять в обстановке, максимально приближенной к реальной.

График нисходящей разработки должен указывать, когда будет завершена реализация функций или модулей. Ожидаемый результат можно формулировать, например, так: «К 15 апреля мы должны запустить обновление файлов. Оно будет состоять из модулей А, D и Е». Следует избегать формулировок типа: «К 15 апреля проект должен быть завершен на 30%». Должно быть явно указано, когда будут готовы и включены в программу конкретные функции (конкретные модули). Таким образом, в тот момент, когда демонстрируется работа модуля с запланированными тестами, можно однозначно определить, выполнен график или нет.

Нисходящая разработка позволяет работать над проектами планомерным, прогнозируемым образом, позволяет упростить межмодульные связи и проблемы объединения модулей, а ресурсы распределить более равномерно.

### Контрольные вопросы и упражнения

1. Нисходящая разработка — это процесс, состоящий из трех шагов. Каких?
2. Каковы две проблемы, возникающие в связи с использованием строго иерархического подхода при планировании порядка разработки модулей?
3. Какие типы модулей (функций) вызывают отклонения от строго иерархического подхода при планировании порядка разработки модулей?
4. Какие два условия должны быть выполнены перед программированием модуля в рамках нисходящей разработки?
5. Кто при нисходящей разработке принимает решение об отходе от нисходящей последовательности — руководство или программист? Почему?
6. Почему тесты нужно готовить до начала программирования модуля?
7. Кто готовит тесты? Почему?
8. Что такое ядро программы?
9. Как обычно распределяются исполнители при нисходящей разработке?
10. Сравните потребности в машинном времени при традиционном подходе и нисходящей разработке.
11. Рассмотрите порядок разработки, показанный на рис. 3.3. Перечислите требуемые заглушки для каждого модуля и действия каждой заглушки.

(например, создать такие-то данные, установить индикатор, напечатать сообщение).

12. Рассмотрите схему иерархии на рис. 2.12. Перечислите модули в порядке их программирования и тестирования. Обоснуйте место каждого модуля в этой последовательности и перечислите требуемые заглушки.

13. Определите разумно полный, на ваш взгляд, набор тестовых данных для следующей записи (N — цифра, A — буква, X — буква или цифра):

ITEM NO.	DESCRIPTION	QTY	PRICE
NNNNAA	XXXXXXXXXXXXXXXXXXXXX	NNNN	NNN.NN



## Структурное программирование

*“GOTO — слово из четырех букв”<sup>1)</sup>*

Доказано, что программу для решения любой логической задачи можно составить только из структур *следование, развилка и повторение*. Этот результат установлен Боймом и Якопини<sup>2)</sup>. Доказательство программисту знать не обязательно, если нет особого желания. Такое же в сущности доказательство провел и Миллс<sup>3)</sup>. Главная его идея — преобразовать каждую часть программы в одну из трех основных структур или их комбинацию так, чтобы неструктурированная часть программы уменьшилась. После достаточного числа таких преобразований оставшаяся неструктурированной часть либо исчезнет, либо станет ненужной. Доказывается, что в результате получится программа, эквивалентная исходной и использующая лишь упоминавшиеся основные структуры. Такое доказательство может навести на мысль, что для получения структурной программы надо сначала написать неструктурную, а затем *преобразовать* ее. Так, однако, делать не следует.

Упомянувшееся доказательство непосредственно применимо лишь к простым программам. *Простая программа*:

- 1) содержит единственный вход,
- 2) содержит единственный выход,
- 3) не содержит бесполезных (недостижимых) фрагментов,
- 4) не содержит бесконечных циклов.

В этом смысле основные структуры являются простыми программами. В настоящей главе мы подробно опишем как основные, так и ряд дополнительных структур, а также расскажем

---

<sup>1)</sup> Перефразированное: «Употребление слов из четырех букв типа “go to” может иногда быть уместным даже в самом лучшем обществе». Donald E. Knuth, “Structured Programming with go to statements”, Computing Surveys, Vol. 6, No. 4, December 1974.

<sup>2)</sup> Corrado Bohm and Guiseppe Jacopini, “Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules”, Communications of the ACM, Vol. 9, May 1966, pp. 366—371.

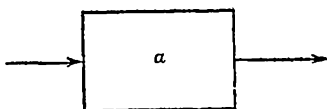
<sup>3)</sup> Harlan D. Mills, Mathematical Foundations for Structured Programming, IBM, Form No. FSC 72—6012, February 1972.

о псевдокоде для их записи. (*Псевдокод* — это средство изображения логики программы, которое можно применять вместо блок-схемы.) Эта глава может представлять особый интерес для начинающих — в ней идет речь о стиле и приемах структурного программирования.

## Управляющие структуры

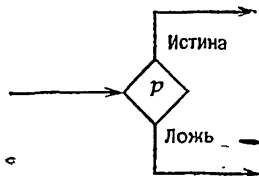
При описании этих структур используются специальные обозначения для обработки, проверки и слияния, а также соединительные линии.

**Обозначение обработки.** Действие, которое необходимо выполнить, обозначается прямоугольником, в который входит и из которого выходит ровно одна линия управления. Этот прямоугольник называется *узлом обработки*, или *функциональным узлом*.



Действие  $a$ , указываемое в этом прямоугольнике, может быть отдельным оператором, вызовом с возвратом некоторой подпрограммы, другой управляющей структурой или несколькими управляющими структурами, образующими подпрограмму или подрутину<sup>1)</sup>.

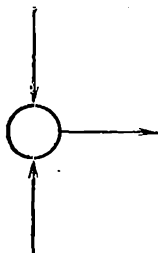
**Обозначение проверки.** Операция проверки обозначается символом, который называется также *предикатным узлом*. Он представляет собой ромб, в который входит одна линия управления, а выходит две. Помещенный внутри ромба предикат  $p$  обозначает условие, которое следует проверять. В результате проверки выбирается один из выходов (но не оба сразу).



---

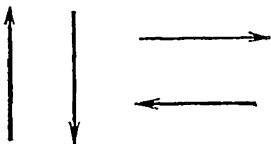
<sup>1)</sup> То есть подпрограмму, используемую несколькими модулями. — *Прим. ред.*

**Обозначение слияния.** Символ слияния — это кружок, где соединяются пути управления.



В этом узле ничего не делается. Это просто соединение, как правило, с двумя входами и одним выходом. Обычно внутри узла слияния ничего не пишется.

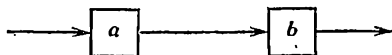
**Соединительные линии.** Соединительные линии изображают передачи управления от одного из перечисленных выше обозначений другому в направлении, указанном стрелкой:



Для изображения любой из основных управляющих структур достаточно этих четырех обозначений.

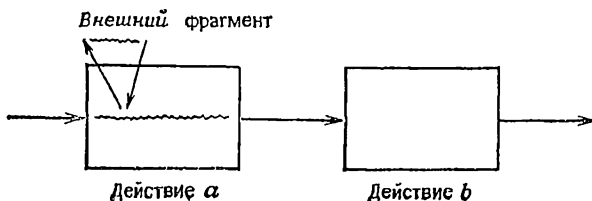
### *Следование*

Следование указывает, что управление передается от одного обозначения обработки следующему.



Так как любое обозначение обработки можно заменять следованием (т. е. двумя обозначениями обработки), то, повторяя эту замену, можно изобразить последовательное исполнение любого количества функций.

Для *полного* процесса может понадобиться *внешний* фрагмент. Это изображается так:

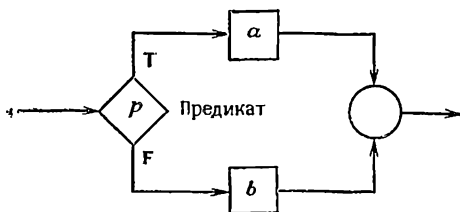


Внешний фрагмент изображает подпрограмму или подрутину. Ей может передаваться управление только до момента возврата к очередному шагу следования. Цельность такого следования обеспечивается тем, что каждая подпрограмма или подрутина сама должна быть *простой программой*.

Внешний фрагмент может быть, например, группой операторов некоторого языка программирования (таких, как параграфы в КОБОЛе или внутренние процедуры в ПЛ/I), которой передается управление с последующим возвратом. Таким образом, следование — это такая структура, в которой ряд операций выполняется в линейном порядке (с возможными отклонениями, но с обязательными возвратами к основному порядку исполнения).

### Развилка

Эта структура, ЕСЛИ-ТО-ИНАЧЕ, служит для выбора одной из двух альтернатив. Из приведенной диаграммы видно, что сначала вычисляется логическое выражение  $p$  — оно может быть либо отдельной переменной, принимающей значения вида «истина — ложь», «да — нет», «включено — выключено» и т. п., либо комбинацией таких переменных, объединенных в выражение.



Например:

$X = Y$   
 $A < B$  ИЛИ  $C < D$   
 $B$  И  $C$   
 $I = 100$   
 .

Другими словами, указанное выражение должно быть приводимо к условию типа «истина — ложь». Если выражение истинно, то выполняется действие  $a$ , если ложно — действие  $b$ .

Две линии управления, выходящие из прямоугольников, встречаются в узле слияния, из него исходит только один путь. Таким образом, читатель блок-схемы или программы может быть уверенным, что управление от входа структуры обязательно попадет к ее выходу, но остается вопрос, какая именно альтернатива будет избрана.

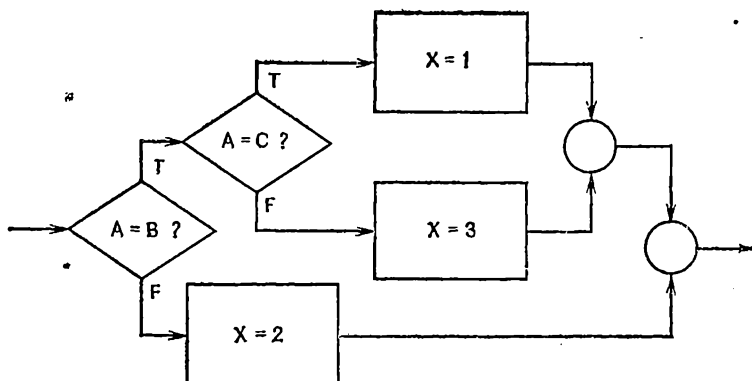
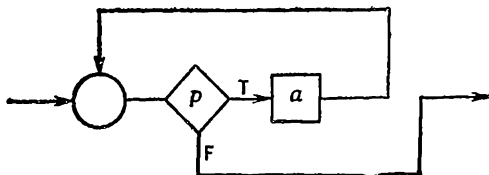


Рис. 4.1. Вложенные структуры ЕСЛИ-ТО-ИНАЧЕ.

Как и для следования, действия могут содержать столько операторов, сколько требуется. Одно или оба действия можно заменять другими структурами. Так, внутри структуры ЕСЛИ-ТО-ИНАЧЕ можно снова употреблять структуру ЕСЛИ-ТО-ИНАЧЕ (рис. 4.1). Глубина вложенности не должна быть слишком большой, иначе программу будет трудно читать.

### Повторение — ЦИКЛ-ПОКА

Эта структура служит для изображения повторений или циклов, которые нужны почти во всех программах. Выглядит она так:



В этой структуре управление проходит через узел слияния к обозначению проверки. Здесь происходит вычисление логического выражения  $p$ . Если оно истинно (Т), то выполняется действие  $a$  и снова вычисляется  $p$ . Если оно ложно (F), то  $a$  не

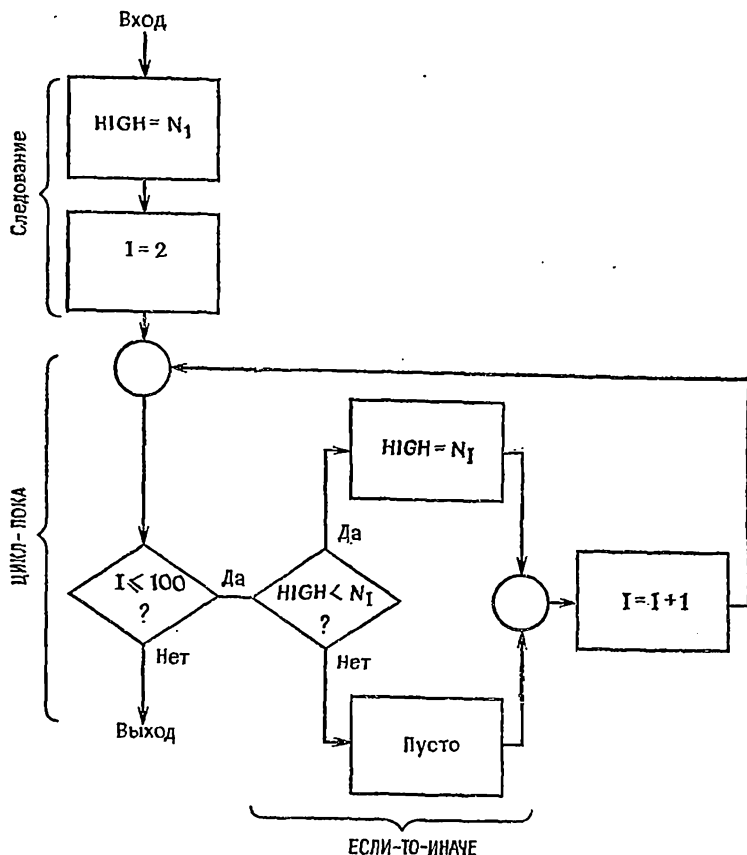


Рис. 4.2. Структурная блок-схема нахождения максимального значения в таблице из 100 чисел.

выполняется. Поскольку выражение  $p$  вычисляется до начала выполнения действия, может случиться, что это действие не будет выполняться ни разу. Если  $p$  содержит управляющую переменную, начальное значение которой было присвоено до начала цикла, то при выполнении действия  $a$  значение этой переменной должно изменяться. Это совершенно необходимо — иначе программа заикнется, что противоречит одному из ограничений простой программы. Действие  $a$  может содержать следования,

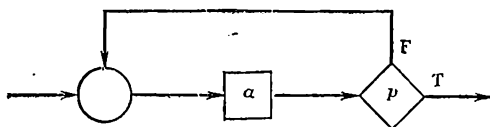
развилки, другие повторения, организованные произвольным допустимым образом.

На рис. 4.2 приведена структурная блок-схема нахождения максимального значения в таблице из 100 чисел. На ней показаны все три основные структуры в действии. Заметим, что наименования конкретных структур в структурные блок-схемы не включаются. Каждая из таких структур является *простой программой*, поскольку имеет один вход, один выход, и управление может передаваться от входа к выходу через любое из входящих в структуру действий.

Хотя перечисленных структур достаточно для изображения любой программы, мы предложим еще несколько структур, облегчающих программирование без ущерба для ясности программы. Это структуры ЦИКЛ-ДО и ВЫБОР.

### Повторение — ЦИКЛ-ДО

ЦИКЛ-ДО — это другой тип цикла. Он изображается так:

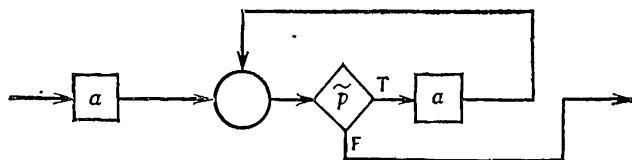


Структура ЦИКЛ-ДО предоставляет в сущности те же возможности, что и ЦИКЛ-ПОКА, за исключением двух моментов.

1. В структуре ЦИКЛ-ДО проверка производится *после* выполнения действия *a*, а в структуре ЦИКЛ-ПОКА — *перед* его выполнением. Таким образом, ЦИКЛ-ДО независимо от значения *p* будет выполнен хотя бы один раз.

2. ЦИКЛ-ДО завершается, когда *p* становится истинным, а ЦИКЛ-ПОКА — когда *p* становится ложным. Другими словами, «цикл выполняется *до* истинности условия».

Структура ЦИКЛ-ДО эквивалентна такому следованию из действия *a* и ЦИКЛ-ПОКА:

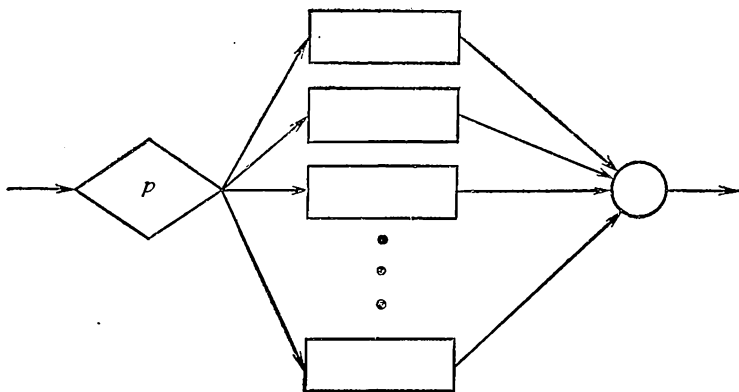


Здесь дубль действия  $a$  соединяется со структурой ЦИКЛ-ПОКА. Поскольку выход из этой структуры происходит при невыполнении условия, необходимо проверять отрицание выражения  $p$  (т. е.  $\neg p$ ). Таким образом, структуру ЦИКЛ-ДО можно преобразовать в эквивалентную структуру, состоящую просто из следования и повторения ЦИКЛ-ПОКА.

Другой способ добиться эффекта, аналогичного повторению ЦИКЛ-ДО — это воспользоваться ЦИКЛ-ПОКА, но сделать  $p$  истинным перед входом в ЦИКЛ-ПОКА. Это обеспечит исполнение действия  $a$  по крайней мере один раз.

### Выбор

Структура ВЫБОР — это обобщение развилки. Она полезна, когда желательно с помощью одной проверки выбрать одну из нескольких альтернатив. Изобразить эту структуру можно так:



Здесь, как и в ЕСЛИ-ТО-ИНАЧЕ, пути после разных действий сходятся в одной точке. Управление от точки входа всегда попадает в точку выхода, выбор пути полностью определяется проверкой.

Структура ВЫБОР эквивалентна структуре с вложенными ЕСЛИ-ТО-ИНАЧЕ, изображенной на рис. 4.3.

Здесь в начале структуры ВЫБОР помещено обозначение проверки — по аналогии со структурой ЕСЛИ-ТО-ИНАЧЕ.



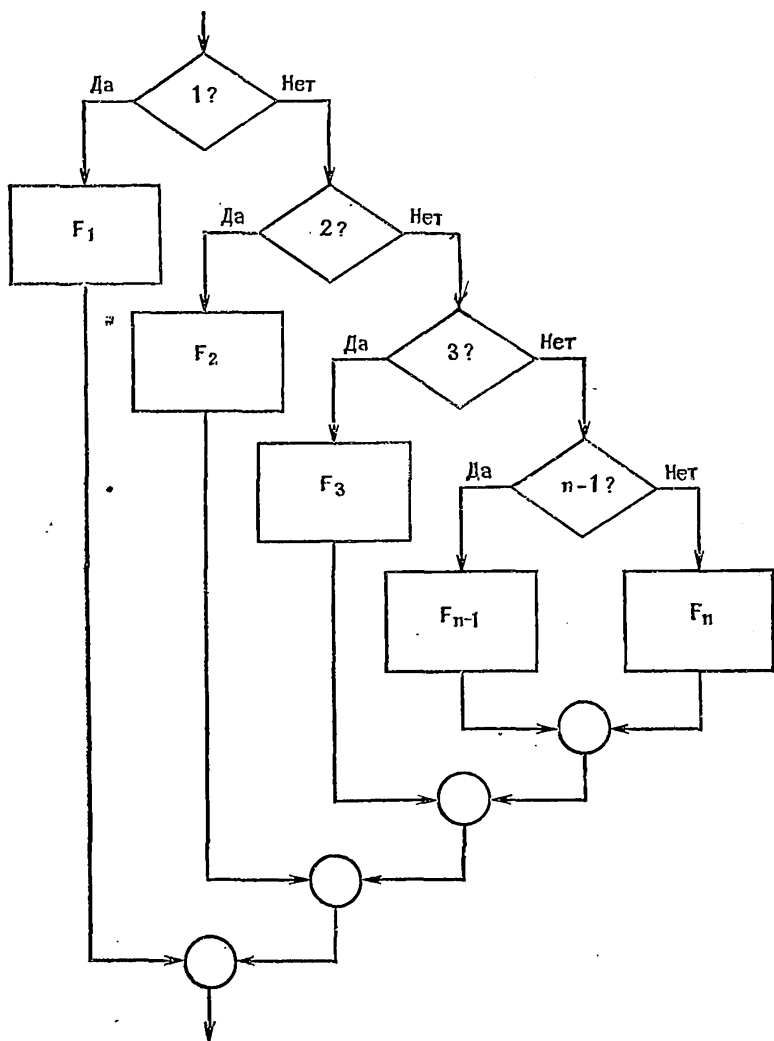
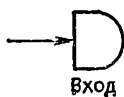
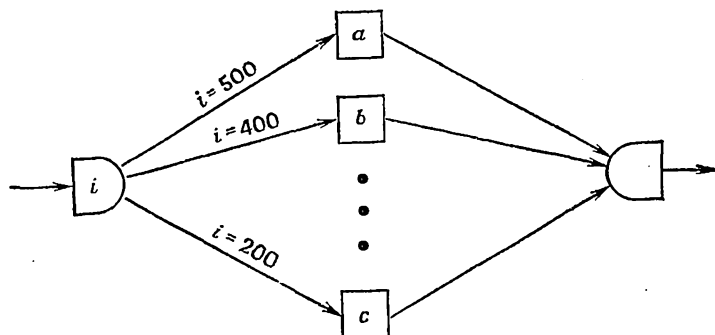


Рис. 4.3. Эквивалент структуры ВЫБОР.

Если желательно начало и конец структуры ВЫБОР отметить особым образом, то вводятся два новых обозначения:



Теперь структура выглядит так:



Какие именно из этих обозначений использовать, определяется стандартами конкретной организации.

В табл. 4.1 перечислены описанные структуры и указано их наличие в языках КОБОЛ, ФОРТРАН и ПЛ/I.

Таблица 4.1

Структуры, имеющиеся в КОБОЛе, ФОРТРАНе и ПЛ/I

Структура	КОБОЛ	ФОРТРАН IV	ПЛ/I
Следование	Есть	Есть	Есть
ЕСЛИ-ТО	Есть	Есть, в случае ТО допустим только один оператор	Есть
ЕСЛИ-ТО-ИНАЧЕ	Есть	*	Есть
ЦИКЛ-ПОКА	Есть, но следу- ет изменить про- верку в ВЫПОЛНИТЬ... ПОКА	*	Есть, кроме того, DO $i=m$ TO $n$ ; эк- вивалентно ЦИКЛ-ПОКА.
ЦИКЛ-ДО	Легко имитиру- ется	*	Легко имитирует- ся
ВЫБОР	*	(В циклах можно проверить лишь условие типа про- грессии. ЦИКЛ-ДО позво- ляет проверять любое условие.) *	

\* Для имитации требуется более одного оператора.

## Псевдокод

Для изображения управляющих структур до сих пор применялись блок-схемы. На рис. 4.2 показано, как их объединить в *структурную блок-схему*. Однако в структурном программировании можно обойтись и без подробных блок-схем. (Особенно приятно будет это услышать программистам, рисующим блок-схемы для документирования программы *после* ее программирования и отладки.)

При этом, конечно, по-прежнему необходимо описать логику программы до начала ее программирования. Но вместо детальных блок-схем служит так называемый *псевдокод*. Он занимает промежуточное положение между естественным и машинным языком. Псевдокод позволяет формально изображать логику программы, не заботясь при этом о синтаксических тонкостях конкретного языка программирования. На язык программирования псевдокод похож тем, что на нем можно выражать такие специфические операции, как:

Читать следующий счет

Добавить дневную прибыль к недельной прибыли

Он, однако, отличается от языка программирования следующими двумя особенностями.

1. Он может использоваться для выражения и более сложных или менее определенных операций. Например:

Найти подходящую запись

2. Он не ограничен никакими формальными синтаксическими правилами. Требуется только употреблять явно перечисленные управляющие структуры и соблюдать правила *ступенчатой* записи, облегчающие чтение.

Псевдокод применяется при проектировании программы *до* начала ее кодирования. Он применяется для того, чтобы подробно объяснить, как программа будет работать. Поэтому там, где используется структурный контроль, другой программист может проверить правильность программы на псевдокоде до начала ее кодирования. Одно из преимуществ псевдокода в том, что изменять текст на нем легче, чем программу, полностью подготовленную для ввода в машину.

Когда программа полностью изготовлена и проверена, то, если она совсем простая, текст на псевдокоде можно просто выбросить, если окончательную программу столь же легко читать, как и псевдокод. Для более сложных программ текст на псевдокоде следует рассматривать как часть программной документации, вместо блок-схемы или чего-нибудь подобного. (Ко-

нечно, очередным изменениям в программе должны предшествовать соответствующие изменения в тексте на псевдокоде.)

### *Изображение управляющих структур на псевдокоде*

Единого или формального определения псевдокода не существует. Вообще говоря, программист может использовать все, что ему нужно. Однако нужно придерживаться следующих рекомендаций:

**Общая.** Используйте только такие обороты, которые приводят к структурной программе. Не употребляйте структуры, явно запрещенные в вашей организации (например ЦИКЛ-ДО).

**Следование.** Записывайте каждое предложение на отдельной строке. Например:

Читать следующую запись прибыли  
Умножить цену на количество, получая величину прибыли  
Вычислить корни квадратного уравнения

**ЕСЛИ-ТО-ИНАЧЕ.** Поскольку в языках программирования высокого уровня имеется условный оператор, в псевдокоде разумно также использовать соответствующую конструкцию. Например:

ЕСЛИ величина допустима  
Добавить к общей сумме  
ИНАЧЕ  
Напечатать сообщение об ошибке  
Отбросить оставшуюся часть записи  
ВСЕ-ЕСЛИ

Слова ЕСЛИ и ИНАЧЕ нужно располагать друг под другом. Так же можно писать и ТО. Назначение ВСЕ-ЕСЛИ — явно обозначить конец структуры. Это ключевое слово облегчает понимание программы.

В рамках одной структуры слова ЕСЛИ, ИНАЧЕ и ВСЕ-ЕСЛИ следует записывать, начиная с одной и той же позиции. Выполняемые после проверки действия записываются на отдельных строках со сдвигом на строго определенное число позиций вправо от соответствующего ключевого слова. При отсутствии объемлющих ЕСЛИ слово ИНАЧЕ можно опускать, если в этом случае ничего не нужно делать (пустой ИНАЧЕ). Для вложенных ЕСЛИ пустой ИНАЧЕ необходим. Вложенные ЕСЛИ записываются со сдвигом вправо, как показано ниже:

ЕСЛИ отрицательное сальдо на счете  
Вызвать программу отрицательного сальдо

ИНАЧЕ

ЕСЛИ нулевое сальдо

Вызвать программу нулевого сальдо

ИНАЧЕ

Вызвать программу положительного сальдо

Добавить сальдо к общей сумме

ВСЕ-ЕСЛИ

ВСЕ-ЕСЛИ

**ЦИКЛ-ПОКА.** Употребляйте ключевое слово, соответствующее языку программирования, который вы затем собираетесь применять (см. табл. 4. 2).

Таблица 4.2

Примеры ключевых слов псевдокода

Язык	Пример псевдокода ЦИКЛ-ПОКА
КОБОЛ ФОРТРАН ПЛ/I	ВЫПОЛНИТЬ ДО минуты > 60 ЦИКЛ минуты = 1, 60 ЦИКЛ-ПОКА минуты ≤ 60

Иногда в КОБОЛе вместо ВЫПОЛНИТЬ (PERFORM) используется слово СДЕЛАТЬ (DO) — оно выразительнее, короче и привычнее. Могут конкурировать также ЦИКЛ (LOOP) и ПОВТОРИТЬ (REPEAT). Конец цикла рекомендуется указывать явно словами ВСЕ-ЦИКЛ или ВСЕ-ПОВТОРИТЬ. Они помогают понять, какие операторы включены в цикл, и напоминают для языков ФОРТРАН и ПЛ/I, что вместо них должны появиться соответствующие операторы завершения цикла. Например:

Читать первую запись

ЦИКЛ-ПОКА есть еще записи

Вызвать проверку записи

ЕСЛИ порядок

Вызвать обработку записи

Вызвать печать

ВСЕ-ЕСЛИ

Читать следующую запись

ВСЕ-ЦИКЛ

ЦИКЛ и ВСЕ-ЦИКЛ записываются друг под другом. Предложения тела цикла записываются на отдельных строках со сдвигом.

**ЦИКЛ-ДО.** Это необязательная структура. Вот пример ее использования:

ЦИКЛ-ДО число месяцев больше 9  
Вызвать экспоненциальное сглаживание  
Вызвать программу вывода  
ВСЕ-ЦИКЛ

**ВЫБОР.** Поскольку эта структура — обобщение ЕСЛИ-ТО-ИНАЧЕ, ее можно выразить несколькими вложенными ЕСЛИ, в которых все ИНАЧЕ выровнены по первому ЕСЛИ. Завершает такую структуру единственный ВСЕ-ЕСЛИ. Например:

ЕСЛИ код покупателя = 100  
Вызвать программу специального счета  
ИНАЧЕ ЕСЛИ код покупателя = 200  
Вызвать программу ежемесячного счета  
ИНАЧЕ ЕСЛИ код покупателя = 300  
Вызвать программу револьверного счета  
ИНАЧЕ  
Вызвать программу обработки ошибочного кода покупателя  
ВСЕ-ЕСЛИ

Другой вариант таков: начало структуры — слово ВЫБОР <sup>1)</sup>. За ними должна следовать переменная, по значению которой выбирается одна из альтернатив. Затем со сдвигом относительно слова ВЫБОР перечисляются все возможные случаи, соответствующие значениям этой переменной. Конец структуры указывается словом ВСЕ-ВЫБОР. Например:

ВЫБОР код покупателя  
СЛУЧАЙ 100  
Вызвать программу специального счета  
СЛУЧАЙ 200  
Вызвать программу ежемесячного счета  
СЛУЧАЙ 300  
Вызвать программу револьверного счета  
СЛУЧАЙ ни один из перечисленных  
Вызвать программу обработки ошибочного кода покупателя  
ВСЕ-ВЫБОР

### *Пример*

В предыдущих главах подчеркивалось, что очень важно облегчить чтение программ. В настоящей главе были введены управляющие структуры и средства их изображения на псевдо-

<sup>1)</sup> В оригинале CASESTART или CASEENTRY.— *Прим. ред.*

коде. Рассмотрим простую задачу, решение которой показывает связь удобочитаемости и структурирования.

Пусть требуется управлять температурой на разных этажах большого здания в течение заданного периода времени. Это делается для уменьшения расхода энергии при эксплуатации здания. Конечно, для этого требуется *управляющая ЭВМ*, позволяющая снимать показания температуры и запоминать их для последующей обработки. Вначале программа подсчитывает количество случаев, когда температура (измеряемая каждую минуту в течение часа) оказывается меньше  $18^{\circ}$  или больше  $26^{\circ}$ . Рассмотрим критически приведенное ниже решение этой задачи на псевдокоде, обращая внимание на его правильность и легкость восприятия.

ЦИКЛ 60 раз

Читать темп

ЕСЛИ темп  $< 18$  увеличить счетчик1

ЕСЛИ темп  $> 26$  увеличить счетчик2

ВСЕ-ЦИКЛ

Напечатать счетчик1 и счетчик2

С точки зрения легкости восприятия ошибки здесь следующие: 1) нет ступенчатой записи, 2) немнемонические названия данных, 3) отсутствует ВСЕ-ЕСЛИ, отмечающий конец ЕСЛИ-ТО-ИНАЧЕ. Имеются два замечания и по поводу правильности: нет присваивания нуля счетчику1 и счетчику2, программа будет неверно работать, если во входном файле меньше 60 значений.

Эти наблюдения приводят к следующим рекомендациям.

1. Для проявления логики программы и выделения ее структур используйте отступы.

2. Используйте mnemonic названия данных. Слова *счетчик1* и *счетчик2* слишком нейтральны. Если бы рассматриваемый фрагмент был частью большой программы, эти названия почти ничего не говорили бы о своей роли в программе.

3. Используйте в тексте на псевдокоде длинные имена данных, даже если язык программирования не позволяет этого. Так, ТЕМП менее наглядно, чем ТЕМПЕРАТУРА, поскольку может означать также и ТЕМП, и ТЕМПЕРАМЕНТ. При кодировании длинные имена, в случае необходимости, можно укоротить <sup>1)</sup>).

4. Для пояснения структуры ЕСЛИ-ТО-ИНАЧЕ используйте ВСЕ-ЕСЛИ. Это особенно существенно для вложенных ЕСЛИ.

5. Стремитесь к простоте. Например, простые ЕСЛИ легче читать, чем вложенные. В предыдущем примере использовались

---

<sup>1)</sup> При этом следует учитывать возможную коллизию имен.— *Прим. перев.*

два простых ЕСЛИ. Оба они выполняются при каждом повторении цикла. Однако логически это не всегда обязательно, поскольку если первый счетчик увеличился (т. е. температура меньше 18°), то не надо проверять, превышает ли температура 26°. Второй ЕСЛИ можно обойти с помощью вложенного ЕСЛИ. При этом программа работала бы быстрее. Если вы хотите быть уверены, что логика правильная, и не хотите жертвовать легкостью восприятия, программируйте сначала понятно, а только потом эффективно.

6. *Проверьте условие окончания файла.* В нашем примере предполагается, что всегда обрабатывается 60 записей. Если их меньше, то программа будет работать неправильно — не предусмотрено распознавание окончания файла <sup>1)</sup>.

При структурном подходе к программированию необходимо заново разработать типичные приемы программирования часто встречающихся ситуаций. Одна из таких задач — как описать обработку условия окончания файла с помощью лишь основных структур. Обычно для изменения нормального порядка выполнения программы использовались операторы перехода. Например:

```
Цикл:  Читать следующую запись
        ЕСЛИ конец файла ПЕРЕЙТИ К Кф
        Обработать запись
        ПЕРЕЙТИ К Цикл
Кф:    Закрывать файлы
        Закончить работу
```

Этот текст не структурирован, поскольку здесь есть операторы перехода как вперед, так и назад. Структурное решение состоит в использовании конструкции ЦИКЛ-ПОКА:

```
Есть необработанные записи = да
ЦИКЛ-ПОКА Есть необработанные записи
```

```
·
·
·
```

```
ВСЕ-ЦИКЛ
```

*Есть необработанные записи* — это индикатор, которому следует придать начальное значение (например, ВКЛЮЧЕН, ДА, ИСТИНА). Когда встречается конец файла, индикатору присваивается соответственно ВЫКЛЮЧЕН, НЕТ или ЛОЖЬ, что вызывает выход из структуры ЦИКЛ-ПОКА. Как это фактически делается, зависит от используемого языка программи-

---

<sup>1)</sup> В таких языках программирования, как ФОРТРАН, необходимо в конце данных поместить специальную отметку (например, 9999) и проверять на совпадение с ней каждую вводимую запись.



рования. На псевдокоде мы будем указывать *после* оператора чтения правило изменения индикатора при появлении конца файла. Например:

```
Есть необработанные записи = да
ЦИКЛ-ПОКА Есть необработанные записи
  Читать запись
    КОНЕЦ-ФАЙЛА Есть необработанные записи = нет
  .
  .
  .
ВСЕ-ЦИКЛ
```

Сразу после оператора чтения в предыдущем примере необходимо *проверить* индикатор конца файла. Это объясняется тем, что за оператором чтения следует обработка введенной записи. Если же встретился конец файла, следует избрать другой путь (не выполнять обработку записи). После оператора чтения может использоваться структура ЕСЛИ-ТО-ИНАЧЕ, определяющая, нужно ли выполнять *обработку* или *завершающие действия*, такие, как печать окончательных результатов и закрытие файлов. Например:

```
Есть необработанные записи = да
ЦИКЛ-ПОКА Есть необработанные записи
  Читать запись
    КОНЕЦ-ФАЙЛА Есть необработанные записи = нет
    ЕСЛИ Есть необработанные записи
      Обработать запись
    ИНАЧЕ
      Выполнить действия по концу файла
    ВСЕ-ЦИКЛ
ВСЕ-ЦИКЛ
```

Можно и по-другому реализовать обработку окончания файла — вынести эти действия из структуры ЦИКЛ-ПОКА.

```
Есть необработанные записи = да
ЦИКЛ-ПОКА Есть необработанные записи
  Читать запись
    КОНЕЦ-ФАЙЛА Есть необработанные записи = нет
    ЕСЛИ Есть необработанные записи
      Обработать запись
    ВСЕ-ЕСЛИ
  ВСЕ-ЦИКЛ
  Выполнить действия по концу файла
```

Структурный метод чтения и обработки записей может показаться неизящным, а в некоторых случаях и более сложным,

чем при неструктурном подходе (по сравнению с примером, где использовались операторы перехода). Однако в таких неструктурных программах обнаруживается много однотипных ошибок. Либо метка, которая должна указывать начало цикла, помечает (с самого начала или после модификации программы) не тот оператор, либо оператор перехода передает управление не на начало, а в середину тела цикла. Ни одна из подобных ошибок не возникает при структурном программировании.

```

Выше=0
Нижe=0
Есть показания=да
ЦИКЛ-ПОКА Есть показания
  Читать температуру
  КОНЕЦ-ФАЙЛА Есть показания=нет
ЕСЛИ Есть показания=да
  ЕСЛИ температура < 18
    Увеличить Нижe
  ИНАЧЕ
    ЕСЛИ температура > 26
      Увеличить Выше
  ВСЕ-ЕСЛИ
ВСЕ-ЕСЛИ
ВСЕ-ЕСЛИ
ВСЕ-ЦИКЛ
Напечатать результаты
  
```

Рис. 4.4. Псевдокод — программа анализа температуры.

На рис. 4.4 снова приведена программа анализа температуры, с отступами, mnemonicическими именами данных, ВСЕ-ЕСЛИ, присваиванием начальных значений и “структурной” обработкой конца файла.

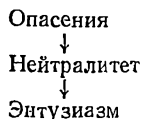
## Выводы для руководства

### *Восприятие программиста*

Интересный парадокс состоит в том, что квалифицированные программисты принимают структурное программирование быстрее, чем программисты среднего уровня, но последние получают более очевидную выгоду от его использования. Одно из объяснений этого заключается в том, что структурное программирование расширяет и усовершенствует общие принципы, которыми квалифицированные программисты руководствовались уже давно. Некоторые даже заявляют, что в структурном программировании нет ничего нового — они именно так всегда и работали. Программисту же среднего уровня структурное программи-

рование особенно полезно потому, что позволяет быстро достичь такой техники программирования, которая обычно появляется лишь после нескольких лет работы.

Реакция на структурное программирование у некоторых, если не у всех, программистов может быть изображена в виде следующих шагов:



Если начальная реакция — это *опасения*, то ее можно объяснить поздним началом этапа программирования, что наводит на мысль о возможном срыве сроков при структурном программировании. По мере знакомства с сутью дела программист обычно становится *нейтральным*. Через несколько месяцев, поняв преимущество новой технологии, все обычно становятся *энтузиастами*.

Поэтому руководителю не следует форсировать быстрое внедрение структурного программирования. На изучение новой технологии необходимо отвести, скажем, три месяца. В течение этого срока программисты используют структурное программирование в порядке эксперимента. К концу срока на основании их мнения принимается решение, вводить ли структурное программирование как стандарт для всей организации. Когда изменения делаются без особого нажима, более вероятна ответная готовность воспринять новые методы.

### *Эффективность*

Когда программист пишет свою первую структурную программу, производительность у него может снизиться. Это снижение, однако, временное. Производительность вернется к прежним показателям, а затем возрастет.

Сравнения структурных и неструктурных программ не показали сколь-нибудь значительного различия в эффективности. Программы сравнимы и по числу предложений исходного языка, и по размерам и времени исполнения выходной программы. Отклонения бывают и в одну, и в другую сторону. Различия больше касаются способностей программиста, а не используемой им технологии. Вероятно, структурный подход помогает не создавать очень неэффективные программы. Если после структурирования увеличивается объем и время работы программ, то потеря эффективности обычно окупается дешевизной последующего сопровождения. Это особенно верно тогда, когда стоимость исполнения программы сравнима с затратами на содержание программистов, занимающихся экономией времени исполнения.

## Обучение

Обучить правилам структурного программирования несложно — времени требует их освоение. Таким образом, метод подготовки квалифицированных структурных программистов состоит в том, чтобы на практике познакомить их с возможными трудностями. При этом обнаруживается немало «подводных камней», преодоление которых повышает уровень опытности программистов. Лучше всего выбрать небольшую группу высококвалифицированных программистов и предоставить им возможности и время экспериментировать, анализировать, обмениваться идеями и вырабатывать экспертные оценки. Затем они могут давать советы остальному персоналу. Другой подход — использовать *взаимопомощь*, при которой два программиста, изучая структурное программирование, могут контролировать работу друг друга. Эта идея может быть обобщена и на большее число людей — три или более человек могут систематически анализировать работу друг друга, как это делается при структурном контроле.

В идеале структурное программирование следует включать в программу обучения программиста любого уровня. Те, кто уже научился программировать неструктурно, должны изучать новую технологию, чтобы избавиться от вредных программистских привычек — это создает дополнительные потребности в хороших учебных программах. Другими словами, необходимы две учебные программы: одна — чтобы *учить* новичков, другая — чтобы *переучивать* опытных программистов.

## Стандарты

Чтобы структурное программирование было эффективным, необходимо иметь набор стандартов, полностью определяющих характер его использования в организации. Конкретные рекомендации по структурному программированию могут меняться от одной организации к другой, но внутри одного подразделения единообразие необходимо. Стандарты должны обеспечивать легкость восприятия и модификации программ.

Выбор стандартов потребует накопления определенного опыта. Необходимо время, чтобы собрать и осмыслить возможные реакции программистов. Когда же стандарты будут приняты, то они должны оказаться практичными, а программисты должны понимать их значение.

Один из способов достичь этого — испробовать структурное программирование сначала на пробном проекте. *Пробный проект* — это не эксперимент, а настоящий проект, выполнение которого тщательно исследуется и контролируется. Занятые

в нем люди должны быть заинтересованы в проверке новых методов, внимательно следить за его ходом, с тем чтобы дать оценку и рекомендовать изменения. Пробный проект должен быть небольшим, управлять им нужно с особым вниманием, чтобы выявить преимущества и оценить используемые методы. Все это позволит определить работоспособный набор стандартов. Остальные группы программистов могут изучить его, испытать и сформулировать свою точку зрения. В результате стандарты становятся более реальными и практичными.

Способность подразделения принять предлагаемые стандарты определяется следующими факторами:

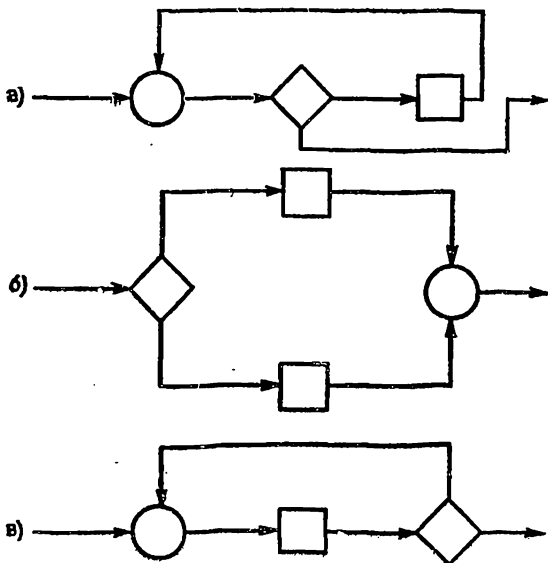
1. Если люди сами участвуют в разработке стандартов, их желание следовать этим стандартам возрастает.

2. Поскольку структурные программы читать легче, чем неструктурные, нетрудно просмотреть их и найти отклонения от стандартов.

3. Если используется метод структурного контроля, то одно сознание, что коллега будет читать программу, заставляет программиста строго следовать стандартам.

### Контрольные вопросы и упражнения

1. Определение *простой программы* состоит из четырех пунктов. Каких?
2. Определите типы приведенных ниже структур и пометьте альтернативные пути там, где это необходимо.



3. При каких условиях возникает потребность в структуре ВЫБОР?
4. Что неверно с точки зрения логики и структурирования в следующей программе, вычисляющей *среднее арифметическое* для  $n$  вводимых величин?

```
Цикл:  Читать А
        ЕСЛИ А < 9999
            Увеличить счетчик
            Добавить А к сумме
            ПЕРЕЙТИ К Цикл
        ИНАЧЕ
            Вычислить среднее (сумма/счетчик)
            Напечатать результаты
        ВСЕ-ЦИКЛ
```

5. Нарисуйте основные управляющие структуры для каждого из следующих шагов. Укажите действия, выполняемые в каждом прямоугольнике. (Пунктам а, б и в должны соответствовать отдельные структуры.)

а) Читать три значения А, В и С, а затем вызвать подпрограмму ПРОВЕРИТЬ-ДАННЫЕ,

б) если А больше, чем 0, добавить А к В, иначе если А меньше или равно 0, добавить С к В.

в) Напечатать А, В и С.

6. По заданным ниже шагам программы нарисуйте структуры, необходимые для решения задачи. Укажите действие программы в каждом прямоугольнике. Соедините каждую структуру со следующей, получив тем самым блок-схему программы.

а) Читать запись.

б) Вызвать подпрограмму по имени ПРОВЕРИТЬ-ДАННЫЕ.

в) Если входные данные правильные (т. е. если ДА), то вычислить СУММА (ЦЕНА-ТОВАРА \* КОЛИЧЕСТВО) и напечатать результаты; иначе, если не ДА, напечатать сообщение об ошибке,

г) Закончить работу,

## Пошаговая детализация

*«Программирование — зеркало разума».*

Джеральд Вейнберг <sup>1)</sup>

29

В прошлом программированию учили редко. Синтаксису языков программирования, конечно, учили, но едва ли учили тому, как создавать программу. До самого последнего времени о проектировании или организации программ было написано совсем немного. Опытные программисты в конце концов вырабатывали свой собственный метод. Но он оставался на уровне интуиции и не был хорошо определенным или документированным. В настоящее время некоторые из этих методов уточнены и зафиксированы с тем, чтобы другие могли ими пользоваться.

Один из таких методов — *пошаговая детализация*. Каждый шаг этого процесса включает в себя разложение функции модуля на подфункции. В конечном итоге эти подфункции превращаются в шаги нужной программы. Этот процесс подобен нисходящему проектированию программы, при котором схема иерархии используется как средство разложения программы на составляющие ее модули. Схема иерархии показывает, конечно, функции и их подчинение, но она не проявляет внутреннюю логику каждого модуля. Пошаговая детализация применяется для декомпозиции функции каждого модуля в соответствии с внутренней логикой, необходимой для выполнения модулем этой функции.

Пошаговая детализация основана на регулярном и довольно своеобразном использовании псевдокода. Рассмотрим, например, библиотечную информационную систему, изображенную на рис. 5.1. На нем показаны три основные программы наряду с некоторыми возможными модулями для каждой из программ. Обратите внимание на напоминания, которые должны быть посланы тем читателям, кто не вернул взятые книги вовремя. Модуль, выполняющий эту функцию, называется ПРОВЕРИТЬ УСЛОВИЕ НАПОМИНАНИЯ. Предположим, что в нашем распоряжении

---

<sup>1)</sup> Gerald Weinberg, «Primer on Programming», THINK, October/November 1974, p. 21. Reprinted by permission from THINK Magazine, published by IBM, copyright 1974 by International Business Machines Corporation.

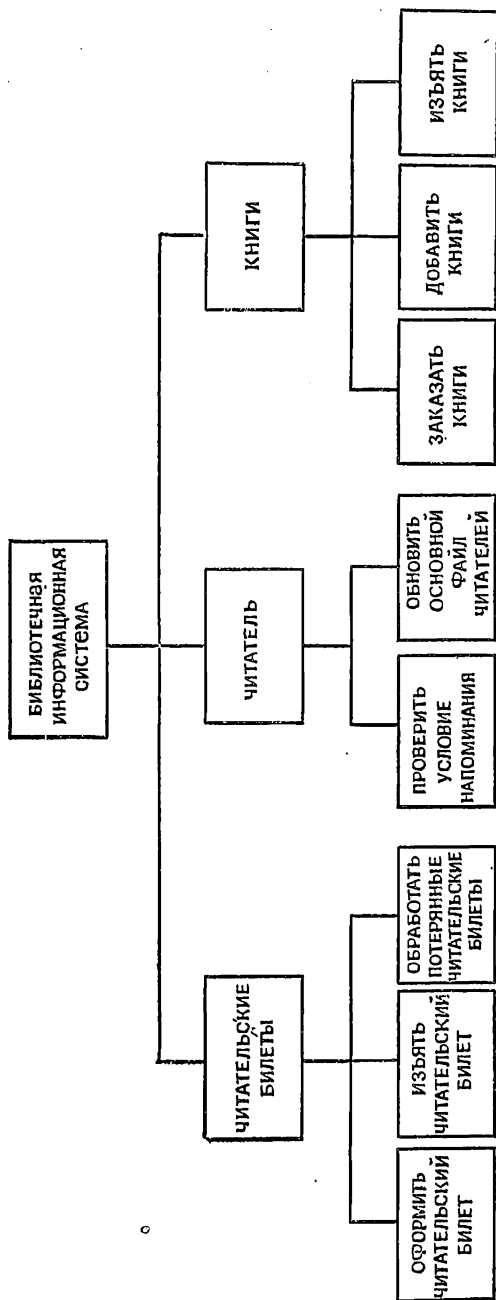


Рис. 5.1. Программы и их модули в библиотечной информационной системе.



имеется файл или база данных читателей. Этот файл регулярно просматривается, чтобы обнаружить нарушителей. Работу этого модуля можно было бы разбить на два основных шага:

Взять запись о следующем читателе

ЕСЛИ какие-то книги задержаны

Приготовить напоминание

Пошаговая детализация — это *итеративный* процесс, в котором при каждом проходе по модулю рассматриваются все более подробные детали. Первый проход порождает указанные выше предложения на псевдокоде. Второй проход заключается в том, что берется одно из этих предложений и расширяется, например, так:

Приготовить напоминание	{	ЕСЛИ предыдущее напоминание = 0
		Напечатать штрафную квитанцию
		ИНАЧЕ ЕСЛИ предыдущее напоминание = 1
		ИНАЧЕ
		Напечатать счет за потерянные книги
		Ограничить право пользования читательским билетом

Теперь, по-видимому, ясно, как возник термин *пошаговая детализация*. Первый шаг связан с очень общим предложением. Разложение первого общего шага в последовательность шагов второго или более низкого уровня заставляет более точно определить логику модуля — это и есть *детализация предыдущей формулировки задачи*. Таким образом, хорошо видно, как модуль расширяется по мере добавления и уточнения деталей. Это похоже на проектирование здания — вначале определяется общий замысел, затем проектируются этажи, комнаты и, наконец, решается вопрос об обоях, мебели и деталях конкретных комнат. Конечно, можно было бы спроектировать одну комнату полностью до начала проектирования следующей и проектировать второй этаж только тогда, когда полностью завершен первый, но это, вероятно, приведет к ошибкам и противоречиям в документации. Подобным же образом, модули должны вначале целиком проектироваться на достаточно общем уровне, а затем следует возвращаться, чтобы с каждой итерацией определять все новые и новые детали.

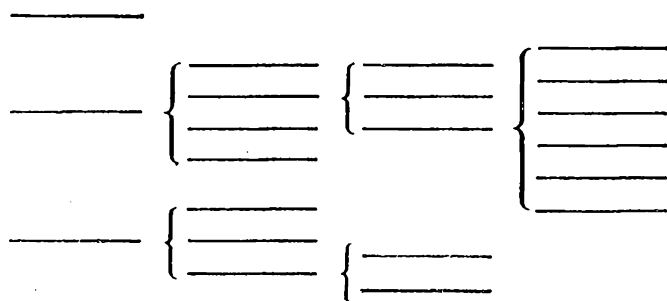
Продолжая заниматься пошаговой детализацией модуля, рассылающего напоминания, можно было бы разложить подфункцию *Напечатать штрафную квитанцию* следующим образом:

Напечатать штраф- ную квитанцию	{	Вычислить величину штрафа
		Установить предыдущее напоми- ние = 1
		Напечатать фамилию и адрес читателя
		Напечатать названия задержанных книг и штрафные квитанции

На следующем уровне детализации результаты этого шага можно разложить снова:

Вычислить величину штрафа	{	Вычислить число дней задержки
		Умножить число дней на штраф за день

Удобно изображать расширения следующим образом: справа от уточняемой строки нарисовать фигурную скобку, а за ней поместить более детальные строки. По мере дальнейших расширений рисуются новые скобки справа от нужной строки. Так, серия уточнений могла бы выглядеть следующим образом:



Если программа большая или сложная, то уровней уточнения будет много. Можно, конечно, использовать более широкий лист бумаги или доску и расширять текст по горизонтали, как только что было показано. А можно поступить иначе: располагать каждую требующую дальнейшего уточнения строку псевдокода на новом листе бумаги.

При использовании псевдокода для пошаговой детализации на каждом новом проходе логика работы модуля описывается все точнее. Первоначальная версия может быть очень общей и близкой к исходным спецификациям, в то время как поздние версии могут оказаться намного ближе к реальной программе. Когда каждое предложение псевдокода непосредственно переводится в один или два предложения языка программирования, можно считать, что процесс пошаговой детализации завершен.

Таким образом, предложения псевдокода на самом первом уровне детализации еще очень общие. Например,

Обработать запрос  
Вычислить еженедельную выплату  
Подсчитать причитающуюся сумму

На промежуточном уровне предложения псевдокода обозначают некоторые куски модуля, но еще недостаточно подробно проработанные для непосредственного программирования:

Обновить основную запись  
Найти все делители целого числа  
Отредактировать входную запись

На самом нижнем уровне предложения уже могут быть непосредственно переведены в предложения языка программирования:

Увеличить счетчик записей  
Напечатать сообщение об ошибке  
Добавить валовой доход к текущему валовому доходу  
Вычислить среднеквадратичное отклонение

### Блок-схемы

Блок-схемами также можно пользоваться как средством пошаговой детализации. Получается по существу та же самая итеративная процедура, но вместо псевдокода будут использоваться блок-схемы. Чем лучше воспользоваться для пошаговой детализации — псевдокодом или блок-схемами? Выбирайте, что вам больше нравится. В этой главе будет рассказано о применении и того, и другого средства. Возможно, сначала блок-схемы покажутся более привычными и поэтому более удобными в употреблении. Однако все же предпочитают пользоваться псевдокодом, поскольку при вычерчивании блок-схем часто возникают проблемы, связанные, например, с ограниченным размером листа бумаги.

Когда вместо псевдокода используют блок-схемы, следуют тем же принципам расширения и структурирования логики модуля. Однако как только подготовлена очередная детализация, представляющие ее блоки должны быть поставлены на место расширяемого элемента блок-схемы. Следовательно, необходимо каким-то образом отражать такую замену. Действуя «в лоб», можно просто перерисовывать всю блок-схему после каждого прохода. Лучше, однако, рисовать каждый раз заново общую блок-схему программы, чтобы расширение очередного блока полностью умещалось на одной странице.

Давайте посмотрим, как будет выглядеть наш пример с библиотекой, если для детализации применять блок-схемы. Рис. 5.2 показывает результат первого прохода при детализации логики выбранного модуля. Это нужно изобразить на первом листе бумаги. Затем на отдельном листе бумаги изображается расширение блока Приготовить напоминание (рис. 5.3). Следующее расширение блока — напечатать штрафные квитанции — изображается на третьем листе бумаги; это показано на рис. 5.4.

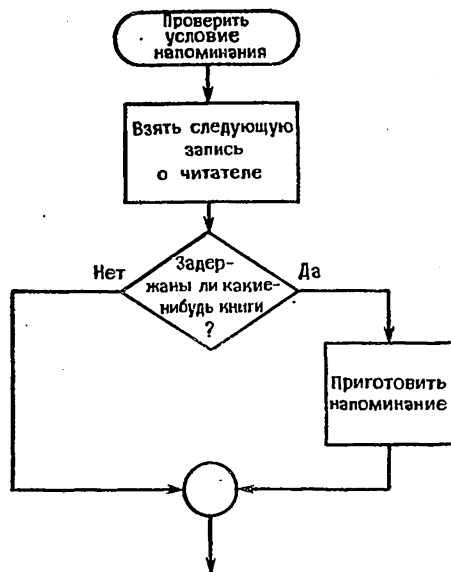


Рис. 5.2. Пошаговая детализация — первый проход для модуля подготовки напоминаний.

Если нужно, можно продолжить расширения. Такая итеративная процедура продолжается до тех пор, пока каждый блок в полученных блок-схемах можно будет непосредственно записать на подходящем языке программирования. Полученные листы с блок-схемами могут как войти, так и не войти в окончательную документацию.

### Правила детализации

Имеется несколько общих принципов, которых следует придерживаться независимо от того, применяете ли вы псевдокод или блок-схемы для итеративного процесса расширения и детализации.

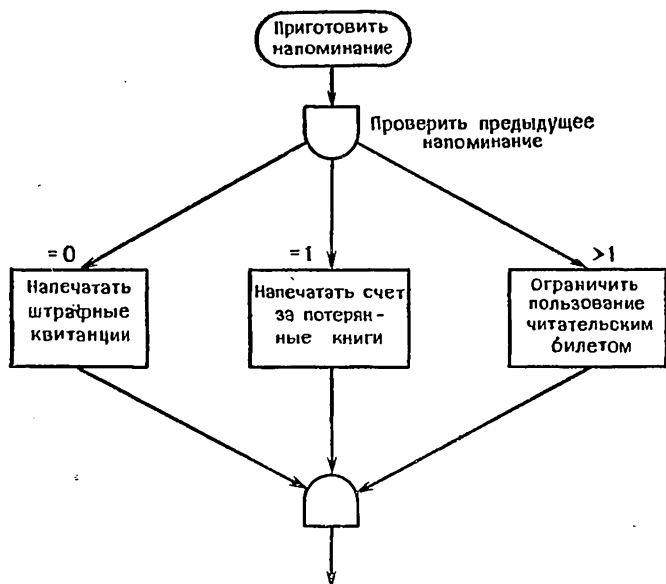


Рис. 5.3. Пошаговая детализация — второй проход для модуля подготовки напоминаний.

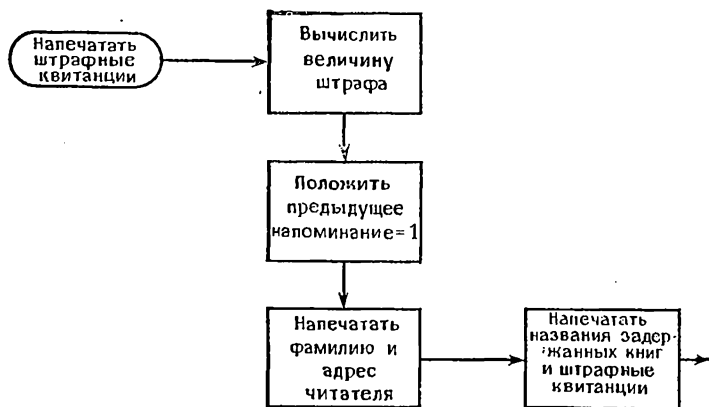


Рис. 5.4. Пошаговая детализация — третий проход для модуля подготовки напоминаний.

1. *Не спешите заниматься мелочами.* За один раз делайте только небольшие расширения. Не вдавайтесь в детали слишком рано, концентрируйте внимание прежде всего на самом существенном.

2. *Тщательно взвешивайте принимаемые решения.* С каждым расширением связаны как довольно очевидные решения, так и достаточно коварные. Не торопитесь, задавайте себе массу вопросов. Пытайтесь понять последствия того, что вы делаете: Что получится, если сделать именно так? Не выбрал ли я путь, в котором позднее раскаюсь? Есть ли другой способ сделать то же самое? Что хорошего он может дать? Рассмотрены ли все разумные варианты?

3. *Внимательно следите за данными.* По мере детализации обязательно понадобится вводить промежуточные данные. Некоторые из ваших решений будут влиять на объем требуемой памяти, типы данных и их употребление. Составьте список необходимых элементов данных и на каждом проходе расширяйте нужные определения данных. Это может существенно повлиять на принимаемое решение, вплоть до его замены.

4. *Будьте готовы отменить ранее принятые решения.* Бывают моменты, когда возникают непредвиденные проблемы. Если это каким-то образом связано с ранее принятым решением, попытайтесь реализовать другой вариант на одном или нескольких уровнях детализации.

### *Пример пошаговой детализации*

Итак, общий подход к пошаговой детализации изложен. Но чтобы лучше понять, как применяются только что перечисленные принципы, рассмотрим пример достаточно сложной программы. Она была выбрана именно потому, что наглядно демонстрирует значение пошаговой детализации как весьма мощного средства программирования. Пример связан с обработкой данных о загрязнении воздуха.

Предположим, что измерения степени загрязнения воздуха производились каждую минуту в течение 24-часового периода около дымовых труб большого завода. Мера загрязнения — число частиц, деленное на миллион (parts per million — PPM). Обычно эта величина находится в диапазоне от 10 000 до 90 000 PPM (если учитывать только неискаженные или исправленные данные). Модуль обработки информации должен делать следующее:

1. Вычислить среднюю величину загрязнения для каждого часа из всего 24-часового периода.

2. Зафиксировать число нарушений за час. Нарушением считается такая ситуация, когда в течение пяти минут подряд

величина загрязнения превосходит 100 000 PPM. (Таким образом, если эта величина превосходит 100 000 PPM в течение десяти минут, то фиксируются *два* нарушения; максимальное число нарушений в час, следовательно, равно 12.) Нарушение может оказаться на границе часов; в этом случае его относят к тому часу, во время которого оно заканчивается.

3. Напечатать отчет, в котором для каждого часа из 24-часового периода будет указана средняя величина загрязнения в PPM и число нарушений в течение этого часа. Печатать нужно в следующем формате:

ЧАС	ЗНАЧЕНИЕ (PPM)	ЧИСЛО НАРУШЕНИЙ
01	50 000	1
02	60 000	2
03	75 000	1
04	100 500	12
05	98 000	4
06	76 000	3
07	70 000	1
.	.	.
.	.	.
24	45,000	0

Мы покажем технику пошаговой детализации и для случая блок-схем, и для случая псевдокода. В практической деятельности нужно, конечно, пользоваться только одним из этих средств. Детализация начинается с единственного предложения:

Обработать данные о загрязнении за 24 часа

Затем можно добавить два более общих предложения, являющихся типичными для большинства программ:

Подготовить  
Завершить

Эта первая стадия детализации изображена на рис. 5.5, где указаны три основных шага. Вполне логично продолжить детализацию, занявшись предложением

Обработать данные за 24 часа

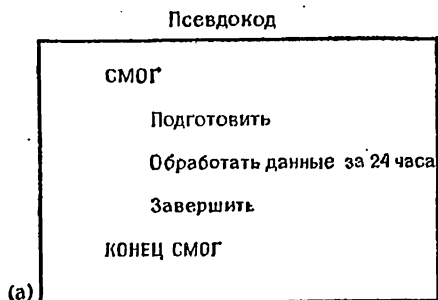
(было бы неправильно начать детализацию с первого предложения, так как только после достаточно детальной проработки второго предложения станет ясно, какие именно элементы следует подготавливать).

Так как процесс обработки данных часто включает повторяющиеся действия, то естественно задать следующий общий вопрос: «Каков наибольший (самый внешний) цикл в нашей программе?» Нам нужно обрабатывать данные за каждый из

24 часов. Поэтому программа обработки должна повторяться 24 раза. При каждом ее выполнении в процессе обработки данных, относящихся к одному часу, можно выделить следующие три шага: 1) прочесть данные этого часа, 2) обработать данные этого часа, 3) напечатать результаты этого часа. Заметим, что здесь мы следуем правилу: «Не спешите заняться мелочами». Поэтому объем одноразовой детализации невелик. Все это так, но теперь нам нужно критически оценить то, что сделано. На каждом шаге детализации нужно изучать намечаемые расширения, чтобы увидеть, какова истинная сущность принятых проектировочных решений и каковы возможные их последствия.

Какие решения мы уже приняли в нашем примере? Первое состоит в том, что все данные для каждого часа должны быть введены до начала какой-либо их обработки. Каковы последствия этого решения? Необходимо выделить память для 60 элементов данных. При программировании понадобится индексировать адреса конкретных элементов данных. Каковы возможные альтернативы? Можно вводить один замер и обрабатывать его до ввода

следующего замера. В этом случае оператор READ должен выполняться 60 раз, чтобы прочесть данные, относящиеся к каждому часу. Каковы последствия этого варианта? Элементы последующих или предыдущих данных становятся недоступными для такой программы. Допустимо ли это или такие элементы



Структурированная блок-схема

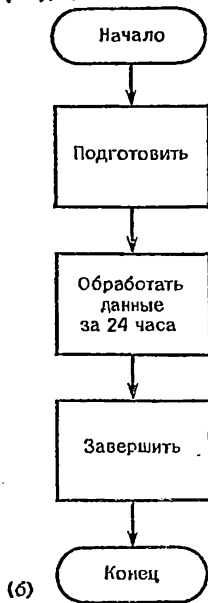


Рис. 5.5. Первая стадия детализации.



могут понадобиться позднее? Еще одна альтернатива — прочитать сразу все данные, относящиеся ко всему 24-часовому периоду. Имеются и другие решения, которые нужно оценить (например, печатать ли результаты в конце каждого часа или в конце 24-часового периода). Мы пытаемся продемонстрировать тот тип рассуждений, с помощью которых нужно взвешивать последствия самых разнообразных альтернатив. Бывает и так, что все рассматриваемые решения оказываются неправильными.

В это время нужно начать заниматься списком элементов данных. Во-первых, необходимы 60 порций памяти для чтения данных за каждый час; мы назовем эту область памяти PPMVALUE. Должны быть напечатаны два результата, MEAN (среднее) и VIOLATIONS (нарушения). Так как мы решили печатать результаты каждого часа, нам достаточно для этого только одной порции памяти. Понадобится также счетчик повторений цикла, который мы назовем HOUR. Кроме этого, чтобы выявить, какие именно элементы данных необходимы, нужно рассмотреть определенные характеристики этих элементов. В частности, еще на стадии разработки нужно определить границы значений, которые могут иметь данные. Максимальный размер элемента данных нужно знать для того, чтобы выделить достаточную память. *Список данных* для нашей задачи на рассматриваемой стадии детализации приведен в табл. 5.1. Когда потребуются новые элементы данных, они будут добавлены к этому списку. По мере того как атрибуты данных (предел, начальное значение) становятся явными, они также будут добавляться в эту таблицу.

Таблица 5.1

Список данных после второй стадии детализации

Имя данного	Описание	Границы	Подготовка
PPMVALUE	Величина загрязнения за минуту в частицах на миллион, 60 элементов	0—1 000 000	
MEAN	Среднее прочитанных PPMVALUE за каждый час	0—1 000 000	
VIOLATIONS	Число нарушений за час	0—12	
HOUR	Счетчик для цикла по часам	0—24	

Хотя мы еще ни для каких данных не определили требований на подготовку, уже есть о чем подумать. Например, нужно ли открывать файлы? В одних языках программирования не нужно, в других требуется. Если файлы нужно открывать, то это один из подготовительных шагов. Так как результаты печатаются в

цикле, то печать заголовка должна делаться вне цикла — и это тоже шаг подготовки.

Еще один подготовительный шаг — установить счетчик цикла (HOUR) равным 1. С другой стороны, оператор цикла в ФОРТРАНе или ПЛ/I или ВЫПОЛНИТЬ в КОБОЛе автоматически производят инициализацию, проверку и изменение переменной цикла. Таким образом, если предполагается пользоваться этими языками, то такой шаг на подготовительной стадии не нужен. Заккрытие файлов, если оно требуется, должно быть указано на шаге *завершить*. На рис. 5.6 показаны текущие результаты нашей детализации.

Теперь рассмотрим три указанных ниже шага основного цикла:

Читать данные этого часа  
Обработать данные этого часа  
Напечатать результаты этого часа

Первый шаг не требует дальнейшего расширения, так как эта операция может быть легко представлена оператором языка программирования.

Подробности (а именно, точный синтаксис, конкретное имя файла) могут пока не рассматриваться, так как они никак не влияют на остальные части программы. Иными словами, нужно рассмотреть в первую очередь те шаги, расширение которых необходимо для дальнейшей детализации программы. Поэтому вполне логично выбрать для расширения шаг

Обработать данные этого часа

С обработкой данных каждого часа связано 60 входных значений. Потребуется еще одна структура повторения, чтобы выполнить серию операторов 60 раз. Как и для любого цикла, мы должны определить *подготовительный* шаг, выполняемый до цикла, и *завершающий* шаг, следующий за ним. Осуществляя детализацию, вы можете обнаружить, что вам эти шаги не нужны — в случае, например, когда предполагается от псевдокода или структурированных блок-схем переходить к некоторым языкам программирования высокого уровня. На рис. 5.7 и 5.8 показана третья стадия детализации для нашей задачи. Мы ввели еще один элемент данных, MINUTES (счетчик для минут). Мы добавляем его к нашему списку данных, но его не требуется подготавливать по той же самой причине, что и счетчик часов (HOUR). Текущее состояние списка данных показано в табл. 5.2.

Следующий шаг, который должен быть расширен, это предложение

Обработать данные этой минуты

Структурированная блок-схема

Псевдокод

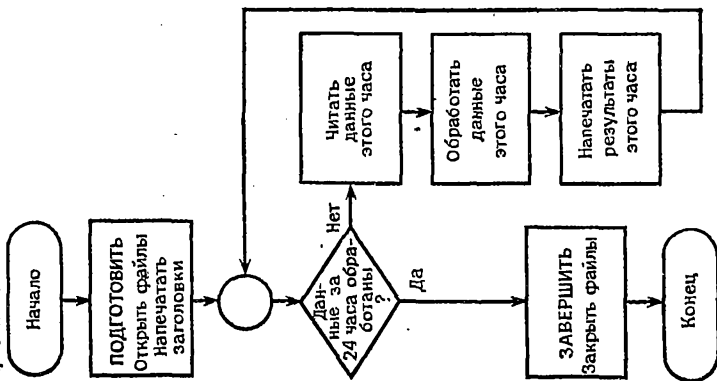
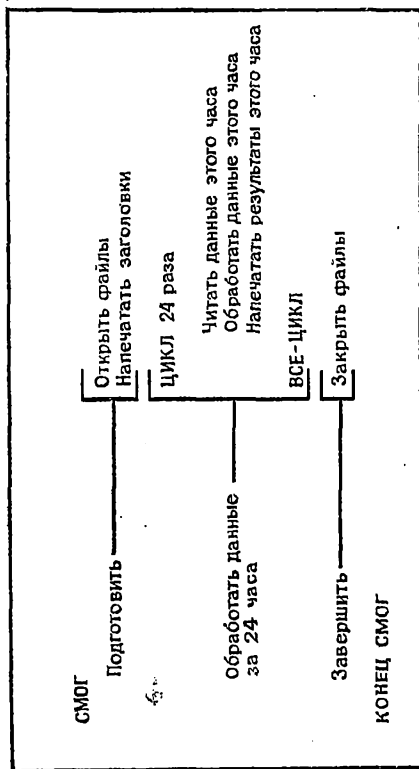


Рис. 5,6, Вторая стадия детализации.

Таблица 5.2

## Список данных после третьей стадии детализации

Имя данных	Описание	Границы	Подготовка
PPMVALUE	Величина загрязнения за минуту в частицах на миллион, 60 элементов	0—1 000 000	Не требуется Не требуется
MEAN	Среднее прочитанных PPMVALUE за каждый час	0—1 000 000	
VIOLATIONS	Число нарушений за час	0—12	
HOUR	Счетчик для цикла по часам	0—24	
MINUTES	Счетчик для цикла по минутам	0—60	

После расширения оно превращается в две последовательные операции, которые должны быть выполнены для каждого замера: 1) накопить сумму, которая готовится для вычисления

СМОГ

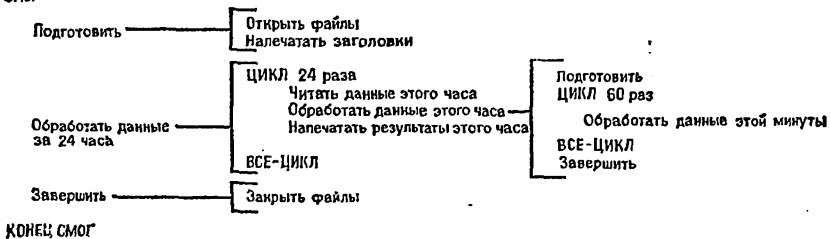


Рис. 5.7. Третья стадия детализации — псевдокод.

средней величины загрязнения за час, и 2) проверить, не превосходит ли прочитанное значение допустимого максимума (ищем нарушение).

Шаг накопления не потребует дальнейшего расширения, но зато понадобится дополнительный элемент данных. Этот элемент — накопитель, который мы назовем SUM (в нем суммируется все прочитанное за час).

Теперь мы должны задать себе вопрос, когда следует присваивать SUM начальное нулевое значение, когда изменять и когда снова обнулять. Начальное значение должно быть присвоено до начала «минутного» цикла, а изменять его нужно внутри этого цикла. (Следующее обнуление наступит при новом исполнении начального присваивания.) В конце этого цикла величина SUM должна быть поделена на 60, чтобы получить среднее, т. е. во внутренний цикл должен быть добавлен завер-

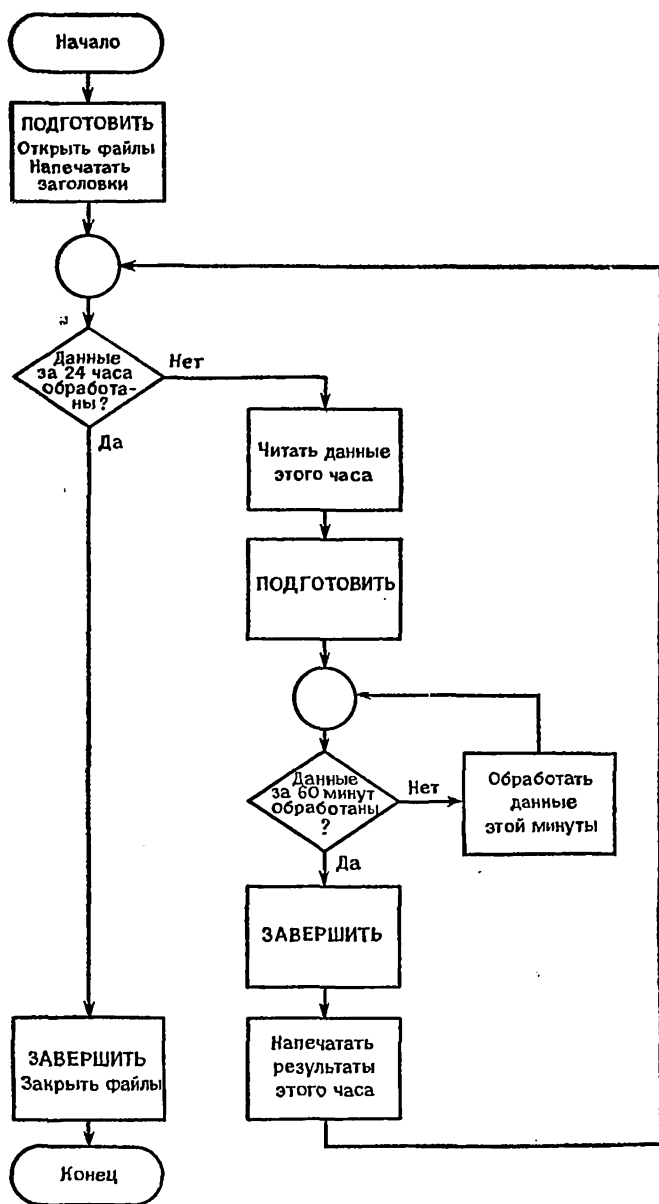


Рис. 5.8. Третья стадия детализации — структурированная блок-схема.

шающий шаг. Шаг *Проверить на нарушение* потребует дальнейшего расширения, но это будет сделано на следующем проходе. На рис. 5.9 и 5.10 показана четвертая стадия детализации. SUM добавляется к списку данных, который приведен в табл. 5.3.

Таблица 5.3

Список данных после четвертой стадии детализации

Имя данных	Описание	Границы	Подготовка
PPMVALUE	Величина загрязнения за минуту в частицах на миллион, 60 элементов	0—1 000 000	Не требуется
MEAN	Среднее прочитанных PPMVALUE за каждый час	0—1 000 000	
VIOLATIONS HOUR	Число нарушений за час	0—12	
	Счетчик для цикла по часам	0—24	
MINUTES	Счетчик для цикла по минутам	0—60	
SUM	Сумма прочитанных PPMVALUE за час	0—60 000 000	Установить равным нулю в начале каждого часа

Теперь расширим шаг

Проверить на нарушение

В соответствии с исходными требованиями нарушение должно фиксироваться тогда, когда величина загрязнения превосходит 100 000 в течение пяти минут *подряд*. Если в этот отрезок времени попадает граница между часами, то нарушение приписывается более позднему из них.

Частью рассматриваемого расширения должна быть проверка, не превосходит ли прочитанное значение 100 000. Если превосходит, то нужно проверять следующие замеры, чтобы узнать, имеется ли пять подобных значений подряд. Для этого понадобится второй счетчик. Первый (VIOLATIONS) считает, сколько раз за час наблюдаются пять превышений подряд, а второй нужен как раз для того, чтобы считать превышения. Назовем его INFRASCTIONS. Когда он станет равным пяти, то это и будет указывать на очередное нарушение.

Так как мы ввели новый элемент данных, то по его поводу нужно задать себе ряд вопросов. Нужно ли устанавливать его начальное значение? При обнаружении превышения необходимо увеличить текущее значение этого счетчика. Если это самое

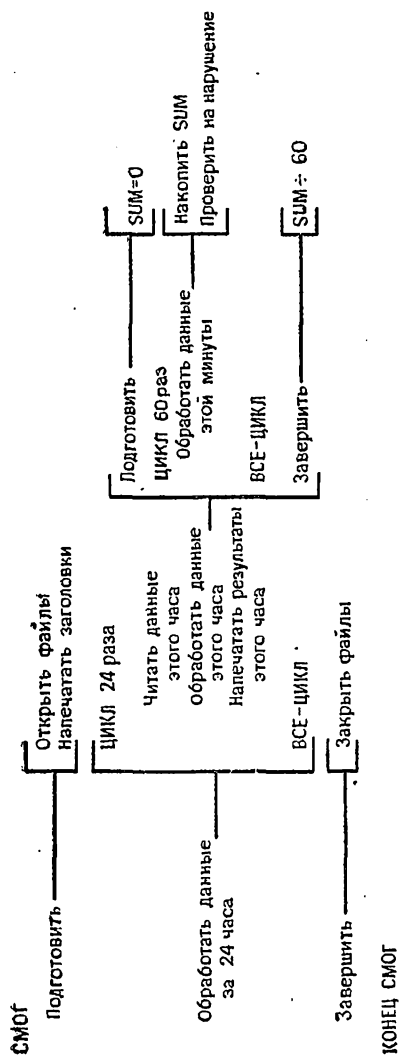


Рис. 5.9. Четвертая стадия детализации — псевдокод.

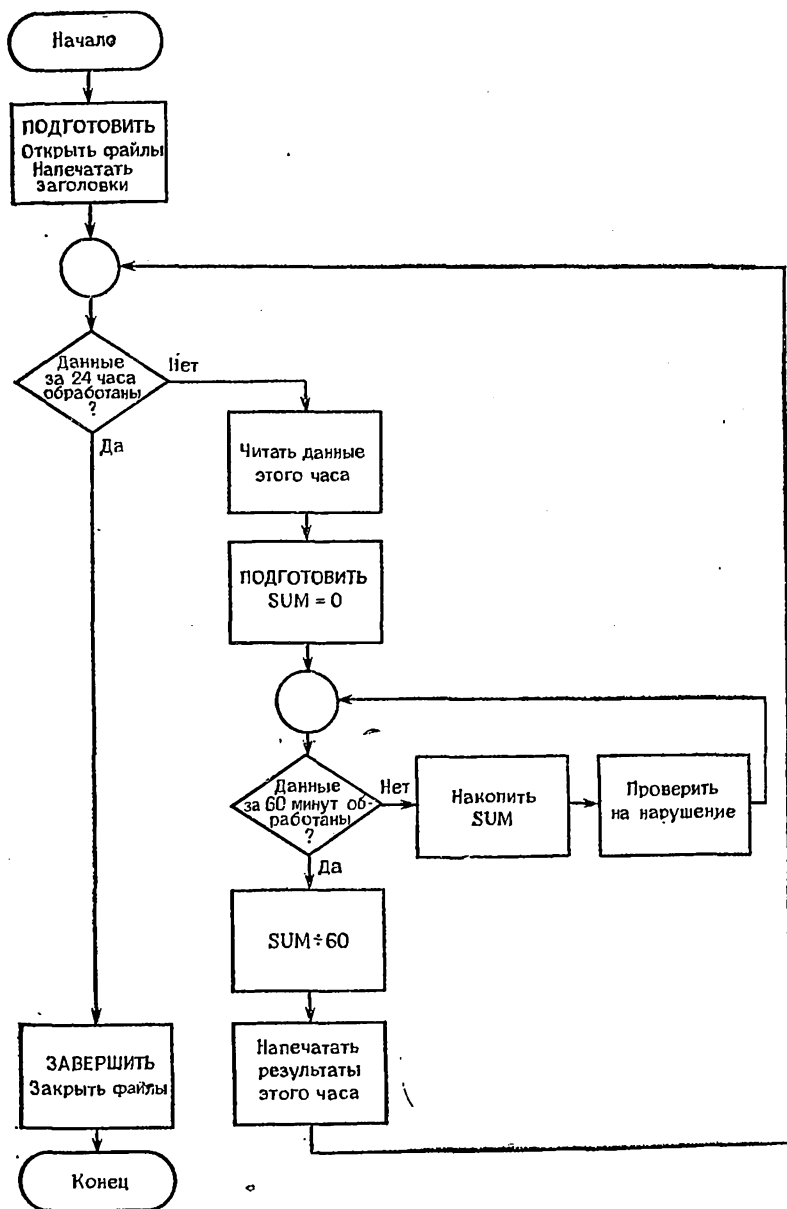


Рис. 5.10. Четвертая стадия детализации — структурированная блок-схема.



первое превышение, то нужно быть уверенным в том, что текущее значение счетчика равно нулю. Когда же нужно обнулять счетчик? Ясно, что это необходимо сделать до обработки данных за первый час, но нужно ли его обнулять перед данными за каждый час? Конечно, нет, так как следующие подряд превышения могут оказаться на границе двух смежных часов. Таким образом, этот счетчик следует обнулять только в начале модуля и после того, как наберется пять превышений подряд. Но это уже мелкая деталь, которую вполне можно отложить на более позднюю стадию детализации. Сейчас же мы можем просто сказать, что если проверка завершилась успешно, то следует «обработать превышение».

Что нужно делать, если очередное прочитанное значение не превышает 100 000? Это прерывает серию идущих подряд превышений, и, следовательно, нужно обнулить INFRACCTIONS. На рис. 5.11 и 5.12 показана пятая стадия детализации.

Нам осталось заняться предложением

#### Обработать превышение

Прежде всего нужно увеличить на единицу текущее значение счетчика INFRACCTIONS. Если этот счетчик стал равным пяти, то следует увеличить на единицу счетчик VIOLATIONS и обнулить счетчик INFRACCTIONS. Если значение счетчика INFRACCTIONS еще не достигло пяти, то больше ничего делать не нужно. Итак, мы завершили детализацию логики программы.

Теперь нужно проверить, полон ли наш список элементов данных. Мы должны также убедиться в том, что подготовка, обновление и проверка элементов данных сделаны в самых подходящих местах. Окончательный список данных приведен в табл. 5.4.

Шестьдесят замеров PPMVALUE обновляются для каждого часа, так что этот элемент данных в подготовке не нуждается. MEAN вычисляется в конце каждого часа и не требует дополнительного внимания. Подготовка HOUR, MINUTES, SUM и INFRACCTIONS уже рассматривалась, а вот VIOLATIONS еще нет. Нужно ли и этот счетчик обнулять в самом начале программы, как это сделано для INFRACCTIONS? Конечно, нет. В отличие от INFRACCTIONS его необходимо обнулять в начале каждого часа, так как мы подсчитываем число нарушений в час, и все нарушения, заканчивающиеся в течение определенного часа, приписываются именно ему. Таким образом, VIOLATIONS обнуляется на подготовительном шаге внутри «часового» цикла и поэтому ничего не требует от подготовительного этапа всей программы.

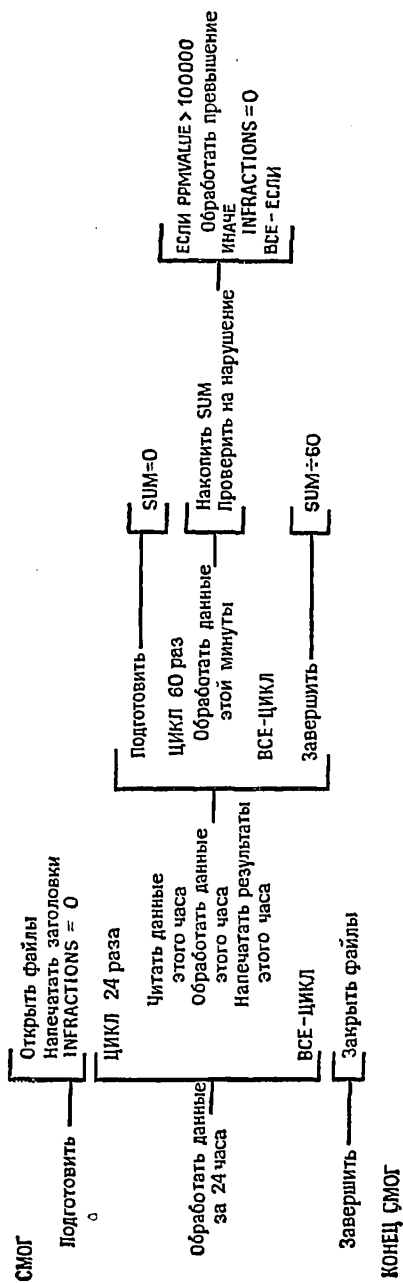


Рис. 5.11. Пятая стадия детализации — псевдокод.

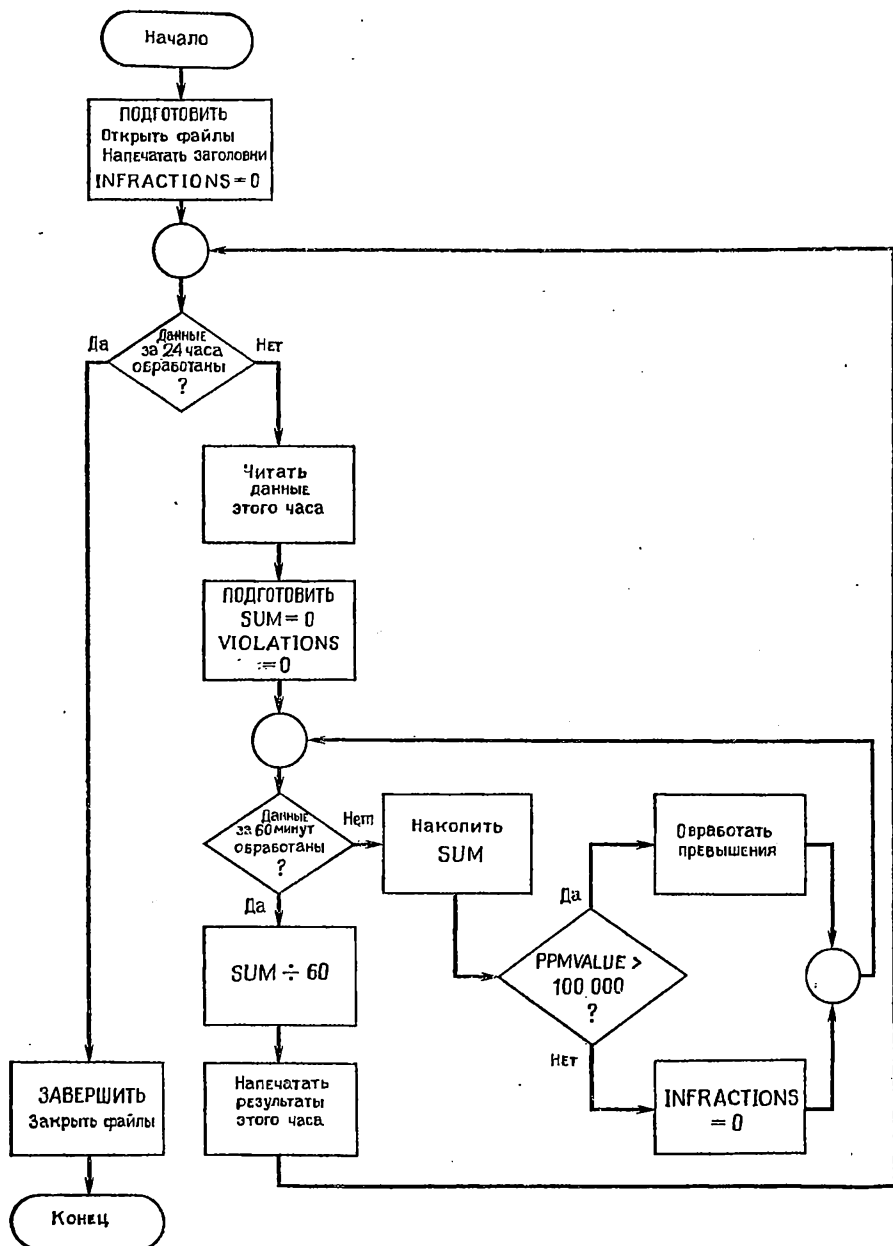


Рис. 5,12. Пятая стадия детализации — структурированная блок-схема,

Таблица 5.4

## Окончательный список данных

Имя данных	Описание	Границы	Подготовка
PPMVALUE	Величина загрязнения за минуту в частицах на миллион, 60 элементов	0—1 000 000	Не требуется
MEAN	Среднее прочитанных PPMVALUE за каждый час	0—1 000 000	Не требуется
VIOLATIONS	Число нарушений за час	0—12	Установить равным нулю в начале каждого часа
HOUR	Счетчик для цикла по часам	0—24	Не требуется
MINUTES	Счетчик для цикла по минутам	0—60	Не требуется
SUM	Сумма прочитанных PPMVALUE за час	0—60 000 000	Установить равным нулю в начале каждого часа
INFRACTIONS	Число подряд прочитанных PPMVALUE, больших 100 000	0—5	Установить равным нулю в начале программы

Итак, мы полностью завершили детализацию и получили окончательную структуру программы, показанную на рис. 5.13 и 5.14. Теперь можно было бы перевести ее на выбранный язык программирования. Если применялся псевдокод, то, как уже говорилось, может оказаться нужным переписать полученное решение на псевдокоде, подставляя соответствующее расширение вместо каждой уточняемой строки. Такой текст будет очень близок к используемому языку программирования. Например, в ПЛИ и ФОРТРАНе повторение будет часто представлено циклом DO. Так как они могут быть вложенными, то вся структура может быть последовательно записана, как показано на рис. 5.15.

В КОБОЛе повторение было бы представлено оператором ВЫПОЛНИТЬ и потребовались бы отдельные параграфы для тела цикла, как показано на рис. 5.16.

## Сегментирование

Модули, которые чересчур велики (100—200 выполняемых операторов), могут быть разбиты на *сегменты*. Сегмент — это и логическая, и физическая часть модуля. Логически, это подфункция функции модуля. Физически сегмент ограничивается

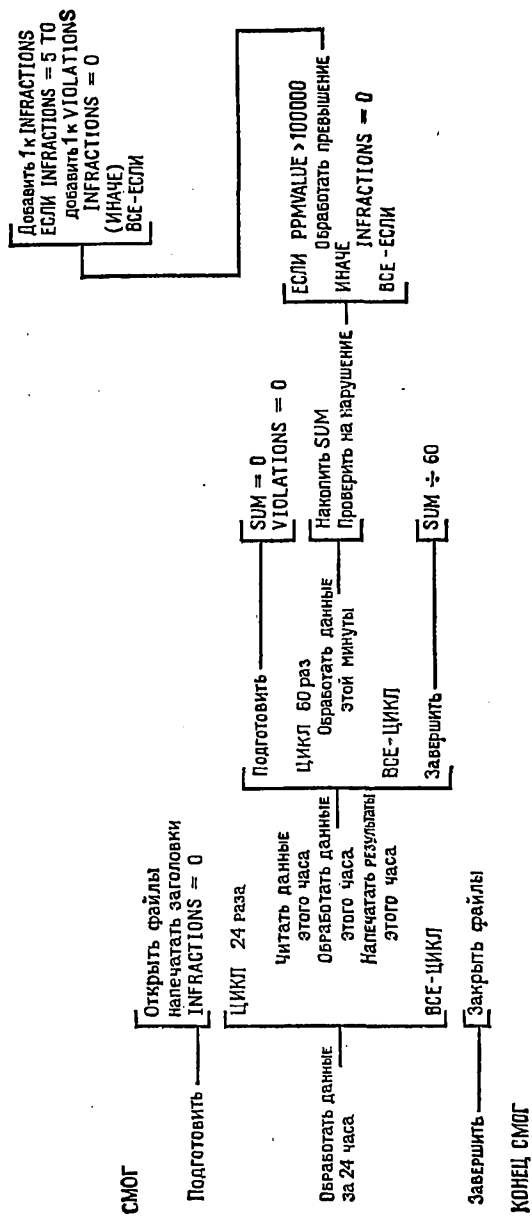


Рис. 5.13. Шестая стадия детализации — псевдокод.

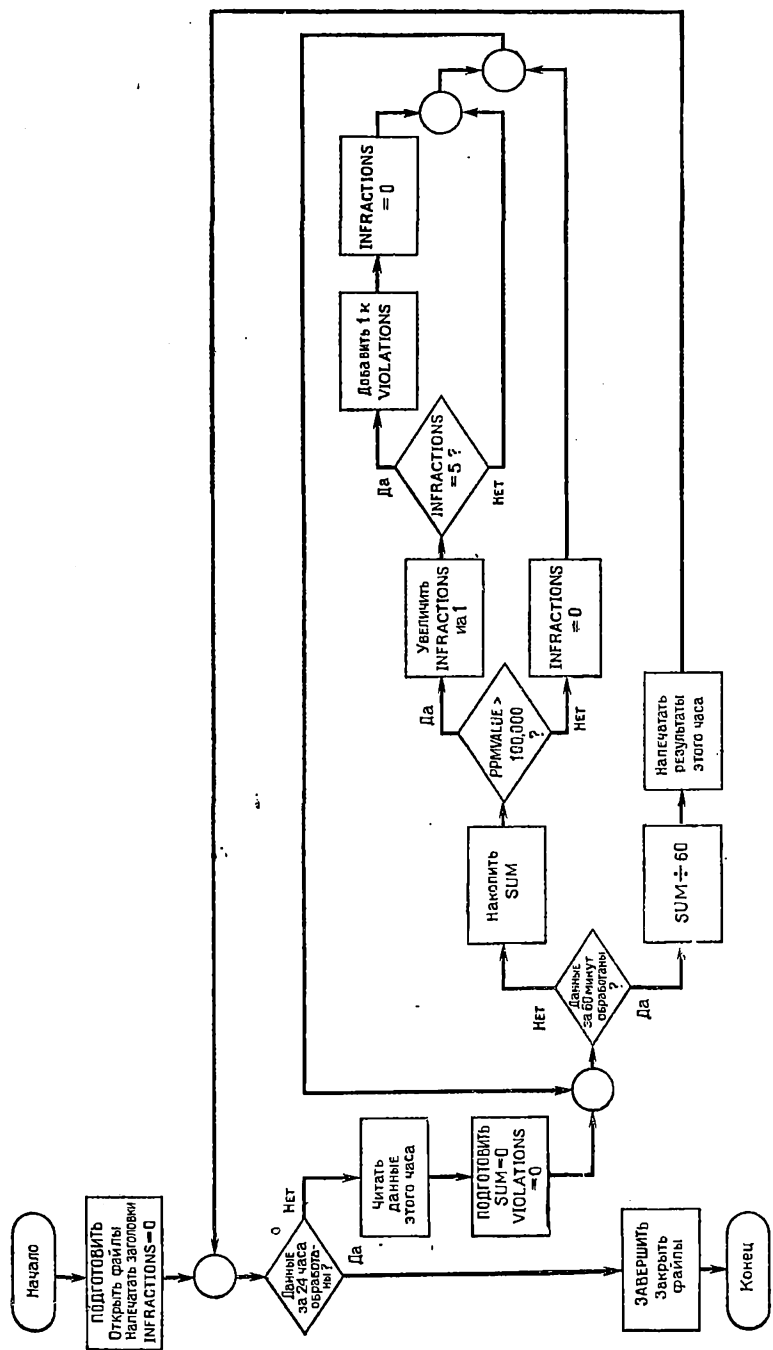


Рис. 5.14. Шестая стадия детализации — структурированная блок-схема.

числом строк исходного текста, которое помещается на одной странице печатающего устройства (50—60 строк).

Сегментирование — это способ разумной *организации листинга программы*. Этот способ характерен тем, что логика управления концентрируется в сегментах более высокого уровня, а менее важные детали переносятся в сегменты более низкого уровня. Такой подход должен упростить программирование и

```

СМОГ
    НАПЕЧАТАТЬ ЗАГОЛОВКИ
    INFRACTIONS = 0
    ЦИКЛ 24 РАЗА
        ЧИТАТЬ ДАННЫЕ ЭТОГО ЧАСА
        SUM = 0
        VIOLATIONS = 0
        ЦИКЛ 60 РАЗ
            ДОБАВИТЬ ВЕЛИЧИНУ К SUM
            ЕСЛИ ВЕЛИЧИНА > 100 000 TO
                ДОБАВИТЬ 1 К INFRACTIONS
            ЕСЛИ INFRACTIONS = 5 TO
                ДОБАВИТЬ 1 К VIOLATIONS
            INFRACTIONS = 0
            ( ИНАЧЕ )
            ВСЕ-ЕСЛИ
        ИНАЧЕ
            INFRACTIONS = 0
        ВСЕ-ЕСЛИ
    ВСЕ-ЦИКЛ
    MEAN = SUM ÷ 60
    НАПЕЧАТАТЬ РЕЗУЛЬТАТЫ ЭТОГО ЧАСА
    ВСЕ-ЦИКЛ
    КОНЕЦ СМОГ
    
```

Рис. 5.15. Текст на псевдокоде для ПЛ/1 или ФОРТРАНа после вставок.

```

СМОГ
    ОТКРЫТЬ ФАЙЛЫ
    НАПЕЧАТАТЬ ЗАГОЛОВКИ
    INFRACTIONS = 0
    ВЫПОЛНИТЬ ЧАСОВОЙ ЦИКЛ 24 РАЗА
    ЗАКРЫТЬ ФАЙЛЫ
    КОНЕЦ СМОГ

ЧАСОВОЙ ЦИКЛ
    ЧИТАТЬ ДАННЫЕ ЭТОГО ЧАСА
    SUM = 0
    VIOLATIONS = 0
    ВЫПОЛНИТЬ МИНУТНЫЙ ЦИКЛ 60 РАЗ
    MEAN = SUM ÷ 60
    НАПЕЧАТАТЬ РЕЗУЛЬТАТЫ ЭТОГО ЧАСА
    КОНЕЦ ЧАСОВОГО ЦИКЛА

МИНУТНЫЙ ЦИКЛ
    ДОБАВИТЬ ВЕЛИЧИНУ К SUM
    ЕСЛИ ВЕЛИЧИНА > 100 000
        ДОБАВИТЬ 1 К INFRACTIONS
    ЕСЛИ INFRACTIONS = 5
        ДОБАВИТЬ 1 К VIOLATIONS
    INFRACTIONS = 0
    ИНАЧЕ
        НИЧЕГО
    ВСЕ-ЕСЛИ
    ИНАЧЕ
        INFRACTIONS = 0
    ВСЕ-ЕСЛИ
    КОНЕЦ МИНУТНОГО ЦИКЛА
    
```

Рис. 5.16. Текст на псевдокоде для КОБОЛа после вставок.

тестирование модуля, облегчить чтение и понимание, а также упростить внесение изменений, которые могут потребоваться в будущем.

В процессе пошаговой детализации выделять сегменты совсем нетрудно. При очередном расширении модуля будет сразу ясно, помещается ли его полный текст на одной странице. Если не помещается и вы захотите разделить модуль, выберите такую порцию псевдокода (или блок-схемы), которая может быть оформлена как самостоятельный сегмент. Чем же она характеризуется? Это должна быть *подфункция, которая требует дальнейшего расширения*. Например, если функция полного модуля — напечатать отчет, то один из его сегментов может обрабатывать пере-

полнение печатной страницы, размещая заголовок как в начале первой страницы, так и на всех страницах-продолжениях. Другим подходящим кандидатом на выделение в сегмент может быть часть программы, которая активизируется из нескольких точек модуля.

Вполне возможно, что после новых расширений такой сегмент сам станет больше страницы. В этом случае из него могут быть выделены сегменты более низкого уровня. Таким способом модуль размещается на отдельных страницах, причем последующие страницы содержат все больше подробностей. Если рассмотреть самый верхний сегмент, то можно увидеть общий замысел и главный цикл обработки. Последующие страницы уточняют это общее представление о модуле.

Каждый сегмент должен быть *простой программой* и возвращать управление своему «старшему» сегменту. Это позволяет читать и понимать любой сегмент без изучения сегментов более низких уровней.

Сегментам можно передавать управление тремя способами:

1. Подходящими активизирующими предложениями (например, ВЫПОЛНИТЬ в КОБОЛе или CALL в ПЛ/I).

2. Средствами *текстовой библиотеки*, если она применяется для разработки сегментов (например, КОПИРОВАТЬ в КОБОЛе или %INCLUDE в ПЛ/I). В этом случае можно считать, что после компиляции модуля сегменты будут расположены в логической последовательности, поэтому управление будет попадать к ним и покидать их естественным путем. Если это возможно, то при распечатке и в этом случае нужно располагать сегменты на отдельных страницах, сохраняя основную идею постраничной организации сегментов.

3. Передавать и возвращать управление с помощью оператора перехода (GOTO). Этот вариант несомненно хуже двух предыдущих и *не* рекомендуется. Он не только затрудняет понимание, но и препятствует тому, чтобы сегменты активизировались из нескольких точек модуля, так как возврат из вызванного сегмента должен быть к предложению, непосредственно следующему за «вызвавшим» этот сегмент оператором перехода.

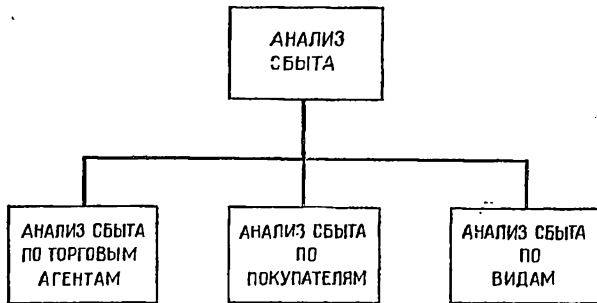
Допустимы и ситуации, когда сегмент может быть больше одной печатной страницы. Например, подпрограмму с большим числом операторов, которые всегда выполняются последовательно, лучше записать целиком, чем произвольно сегментировать. Например, выходной модуль, который размещает длинный список элементов на буфере вывода и заканчивается операторами печати, будет более понятным, если его оставить в виде естественной непрерывной последовательности операторов.



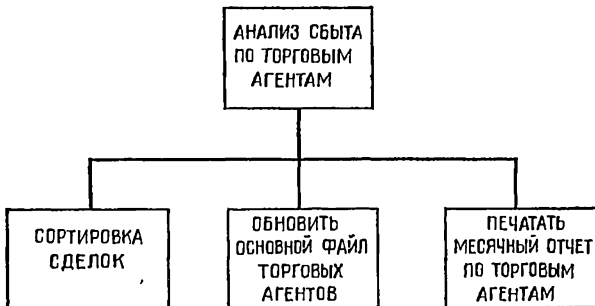
Желательно гибко управлять верхним пределом допустимой длины сегмента, чтобы не мешать возможным добавлениям. Если сегмент занял всю страницу, а позднее потребовалось добавить к нему еще несколько строк, то, вероятно, будет лучше оставить основную структуру модуля без изменений и позволить сегменту занять первые несколько строк следующей страницы. Это лучше, чем искусственно создавать дополнительные сегменты только для того, чтобы избежать выхода за пределы страниц.

### *Пример сегментирования*

Сегментирование — это проектирование сверху вниз на уровне собственно программирования, и поэтому им лучше всего заниматься на стадии пошаговой детализации. Чтобы проиллюстрировать процесс сегментирования, рассмотрим программу анализа сбыта с такими четырьмя модулями:



Самый левый модуль второго уровня мог бы быть разделен на следующие модули:



Модуль, который мы будем разбивать на сегменты, называется ОБНОВИТЬ ОСНОВНОЙ ФАЙЛ ТОРГОВЫХ АГЕН-

ТОВ. Этот основной файл содержит записи следующего формата:

Запись о торговом агенте

Номер торгового агента	Имя, торгового агента	Общий сбыт за месяц	Возвраты за месяц	Объем сбыта за текущий год	Возвраты за текущий год
------------------------------	-----------------------------	------------------------	----------------------	-------------------------------	-------------------------------

Обновляться этот файл будет по сделкам, которые упорядочены по возрастанию номера агента. Записи сделок имеют следующий формат:

Запись о сделке

Код	Номер покупателя	Номер торгового агента	Номер счета	Дата счета	Номер Товара	Количес- тво	Цена	Описание
-----	---------------------	------------------------------	----------------	---------------	-----------------	-----------------	------	----------

Объем сбыта за месяц должен вычисляться для каждого агента на основе записей об отдельных сделках. *Общий объем* должен вычисляться по входным записям для каждого агента и заноситься в поле ОБЩИЙ СБЫТ ЗА МЕСЯЦ (заштрихованное поле в записи о торговом агенте). Что касается возвратов, то будем считать, что они обрабатываются другим модулем.

Может быть и больше одной записи о сделках для одного агента. Сортировка этих сделок по агентам приведет к тому, что записи каждого агента будут собраны вместе. Логика нашего модуля должна принимать в расчет изменение номера агента. Так как записи для каждого агента собраны вместе, то всякий раз, когда изменяется прочитанный номер агента (т. е. возникает скачок в последовательности вводимых номеров), это означает, что настало время записать *общий объем* сбыта по предыдущему агенту и начать вычислять новый общий объем для следующего агента.

Чтобы обрабатывать такие скачки, нужны две программные переменные.

Старый номер торгового  
агента

"1234"

Этот пример  
иллюстрирует момент скачка,  
так как  
старый и новый номера  
различаются

Новый номер торгового  
агента

"1235"

Каждый раз, когда  
читается новая запись  
о сделке,  
номер агента  
помещается сюда

Эти переменные должны сравниваться, чтобы определить момент скачка в последовательности номеров агентов. После каж-

дого сравнения *новый номер агента* нужно присваивать переменной *старый номер агента*.

При первом исполнении программы — когда читается первая запись о сделке — нужно просто скопировать новый номер агента в старый номер агента без сравнения этих переменных. Из-за этого становится нужным индикатор первого исполнения. Изложенные предпосылки позволяют нам перейти непосредственно к изучению программы нашего модуля, изображенной на рис. 5.17.

Она начинается с подготовительных шагов, за которыми следует основной цикл чтения записей о сделках и либо обработка этих сделок, либо обработка окончания файла.

Раздел, имеющий дело с обработкой сделок, содержит три последовательных условных предложения, проверяющих соотношение между старым и новым номерами агентов. Эти проверки обнаруживают скачки в последовательности номеров агентов, а также ошибки в упорядочении файла сделок.

Когда *новый номер агента* больше, чем *старый номер агента*, то нужно обновить основную запись для ранее прочитанного агента, добавив вычисленный общий объем сбыта. Если обновляемый основной файл находится на устройстве прямого доступа и его записи доступны по ключу, для этого требуется такой фрагмент программы:

Читать основную запись агента

При ошибке в ключе положить запись найдена = нет  
ЕСЛИ запись найдена = да

Переслать общий объем в основную запись агента

Писать основную запись агента

ИНАЧЕ

Печатать сообщение об ошибке

ВСЕ-ЕСЛИ

Последний раздел в модуле на рис. 5.17 начинает работать, если обнаруживается конец файла. Предположим пока, что этот фрагмент имеет следующий вид (так он в действительности и выглядел, когда мы его разрабатывали). Все ли в нем правильно?

Завершить.

Обновить последнюю запись агента с учетом нового общего объема

Напечатать сообщение о конце работы

Закрывать файлы

Закончить работу.

Нет, не все. Когда достигается конец файла, необходимо обновить сведения о последнем агенте, поместив в запись о нем накопленный общий объем. Но в одном случае наша программа не сработала бы (выдав, скорее всего, непонятное сообщение о

Общий объем сбыта = 0  
Еще сделки = да  
Запись найдена = да  
Первый проход по модулю = да  
Открыть файлы

ЦИКЛ-ПЮКА еще сделки

    Читать сделку

        В КОНЦЕ ФАЙЛА положить еще сделки = нет

ЕСЛИ еще сделки

    Обработать сделку

ИНАЧЕ

    Завершить

ВСЕ-ЕСЛИ

ВСЕ-ЦИКЛ

Выйти из программы

Обработать сделку.

    ЕСЛИ первый проход по модулю = да

        Поместить новый номер агента в старый номер агента

        Первый проход по модулю = нет

ВСЕ-ЕСЛИ

ЕСЛИ новый номер агента = старый номер агента

    Вычислить объем

    Добавить объем к общему объему сбыта

ВСЕ-ЕСЛИ

ЕСЛИ новый номер агента > старый номер агента

    Читать основную запись агента

        При ошибке в ключе положить запись найдена = нет

ЕСЛИ запись найдена = да

    Переслать общий объем в основную запись агента

    Писать основную запись агента

ИНАЧЕ

    Печатать сообщение об ошибке

ВСЕ-ЕСЛИ

    Писать новый номер агента в старый номер агента

    Вычислить новый объем и переслать в общий объем сбыта

ВСЕ-ЕСЛИ

ЕСЛИ новый номер агента < старый номер агента

    Писать сообщение об ошибке в последовательности сделок

ВСЕ-ЕСЛИ

Завершить.

    Обновить последнюю запись агента с учетом нового общего объема  
    сбыта

    Напечатать сообщение о конце работы

    Закрывать файлы

Закончить работу.

Рис. 5.17. Псевдокод обновления месячного сбыта торгового агента.

какой-нибудь «странный» ошибке) — если бы файл сделок оказался пустым, т. е. не содержал бы ни одной записи. Это очевидная ошибка, но с ней приходится время от времени встречаться. Конечно, если бы файл сделок оказался пустым, то отсутствовал бы и общий объем, который нужно записывать в

последнюю основную запись. Поэтому здесь можно воспользоваться переменной «*первое исполнение*» как признаком того, нужно или не нужно заносить общий объем в последнюю основную запись. Получаем следующий фрагмент:

Завершить.

ЕСЛИ первое исполнение = нет

Обновить последнюю основную запись агента с учетом нового общего объема

Напечатать сообщение о нормальном конце работы

ИНАЧЕ

Напечатать сообщение об ошибочном отсутствии записей о сделках

ВСЕ-ЕСЛИ

Закреть файлы

Закончить работу.

Теперь нужно было бы расширить обновление записи тем самым способом, о котором рассказывалось выше.

А теперь посмотрим на модуль в целом, чтобы понять, какую его часть разумно оформить в виде отдельного сегмента. *Подготовить и главный цикл* могли бы образовать сегмент самого верхнего уровня. Логично выделить также в качестве сегментов *обработать сделку* и *завершить*. Если важна эффективность программы, то обновление записи агента тоже могло бы быть маленьким сегментом, так как в двух разных местах нашего модуля требуются одинаковые действия: один раз в сегменте *обработать сделку* и один раз в сегменте *завершить*.

На рис. 5.18 изображена схема иерархии для нашего модуля из четырех сегментов. В отличие от проектирования программы методом сверху вниз, где схема иерархии рисуется до начала программирования, схема сегментов рисуется после появления текста на псевдокоде. Схема иерархии сегментов показывает подфункции модуля и их зависимость (т. е. «кто кого вызывает»).

На рис. 5.19 показан исправленный вариант нашего модуля. У каждого сегмента теперь есть имя, сегменты расположены с учетом порядка исполнения, чтобы модуль было легче читать. Внутри сегмента *обработать сделку* вложенные условные предложения заменили три последовательных сравнения нового и старого номеров агентов.

## Выводы

Вы, вероятно, захотите использовать пошаговую детализацию для каждого модуля, который вам понадобится написать. В некоторых редких случаях вы можете написать окончательную

программу и без пошаговой детализации. Например, модуль, который просто готовит данные для вывода на печать, может содержать много предложений, размещающих эти данные в поле вывода и печатающих их. Но эти действия относительно простые

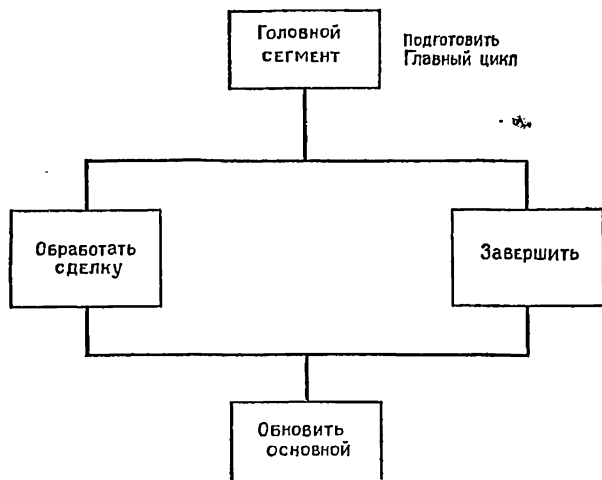


Рис. 5.18. Сегментированный модуль обновления месячного сбыта торгового агента.

и не требуют никакой многоуровневой детализации. Однако большинство модулей, которые вам потребуется написать, будут скорее всего из той категории, для которой пошаговая детализация действительно помогает создавать правильные и хорошо построенные программы.

Пошаговая детализация — это метод, при котором наиболее важные элементы рассматриваются в первую очередь. Затем в общий замысел естественно вовлекаются подчиненные элементы. По мере детализации программа расширяется и в ее основную структуру вставляются новые фрагменты. Каждый шаг этого процесса нужно внимательно изучать, за один раз делать только небольшие расширения. Если на каждой стадии детализации включать много расширений, то внимание будет слишком распылено; поэтому не торопитесь заниматься подробностями.

Перед каждой новой итерацией нужно определить, какую именно часть следует детализировать и в чем будет состоять детализация. Так как некоторые решения могут сильно влиять на окончательное строение программы, то все решения нужно принимать только после тщательного обдумывания и взвешива-

Головной сегмент.

Общий объем сбыта = 0  
Еще сделки = да  
Запись найдена = да  
Первый проход по модулю = да  
Открыть файлы

ЦИКЛ-ПОКА еще сделки

Читать сделку

В КОНЦЕ ФАЙЛА положить еще сделки = нет

ЕСЛИ еще сделки

Обработать сделку

ИНАЧЕ

Завершить

ВСЕ-ЕСЛИ

ВСЕ-ЦИКЛ

Выйти из программы

Обработать сделку.

ЕСЛИ первый проход по модулю = да

Поместить новый номер агента в старый номер агента

Первый проход по модулю = нет

ВСЕ-ЕСЛИ

ЕСЛИ новый номер агента = старый номер агента

Вычислить объем

Добавить объем к общему объему сбыта

ИНАЧЕ

ЕСЛИ новый номер агента > старый номер агента

Обновить основную запись о сбыте

Переслать новый номер агента в старый номер агента

Вычислить новый объем и переслать в общий объем сбыта

ИНАЧЕ

Написать сообщение об ошибке в последовательности сделок

ВСЕ-ЕСЛИ

ВСЕ-ЕСЛИ

Обновить основную запись о сбыте.

Читать основную запись агента

При ошибке в ключе положить запись найдена = нет

ЕСЛИ найдена запись = да

Переслать общий объем сбыта в основную запись агента

Переписать основную запись агента

ИНАЧЕ

Напечатать сообщение об ошибке

ВСЕ-ЕСЛИ

Завершить.

ЕСЛИ первый проход по программе = нет

Обновить основную запись о сбыте

Напечатать сообщение о нормальном конце работы

ИНАЧЕ

Напечатать сообщение об ошибочном отсутствии записей о сделках

ВСЕ-ЕСЛИ

Закрыть файлы

Закончить работу.

Рис. 5, 19, Сегментированный псевдокод для обновления месячного сбыта торгового агента,

ния альтернатив. Часто будет найдено несколько решений, вполне корректных, но различающихся по эффективности, размерам программы или легкости понимания. В трудных случаях или в особо ответственных программах процессе детализации нужно выполнить дважды. Второй результат обычно бывает лучше первого.

Для каждого предпринимаемого расширения нужно выявить, каковы возможные последствия такого решения. Выявленные последствия каждого шага нужно сравнить с последствиями альтернативных решений. В некоторых случаях показавшаяся вполне очевидной детализация может иметь далеко идущие последствия и сильно повлиять на другие части модуля. Может случиться и так, что, несмотря на все попытки предотвратить подобную ситуацию, нежелательный эффект будет обнаружен лишь на самых последних стадиях процесса детализации. В этом случае необходимо вернуться к тому решению, которое привело к обнаруженной проблеме, отказываясь тем самым от результатов нескольких уровней детализации. Впрочем, это еще относительно несложное и недорогое изменение. Намного дороже обходятся изменения *после* того, как модуль запрограммирован.

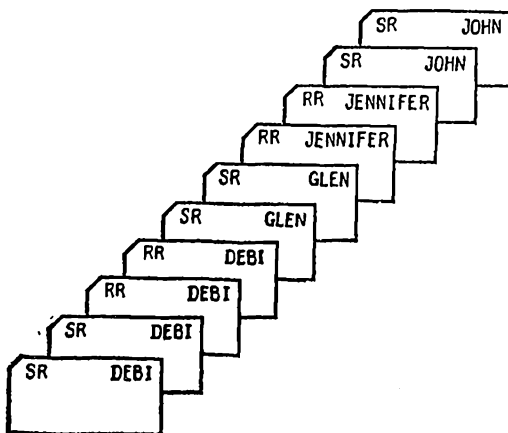
При каждой итерации принимаются решения и по поводу данных. Термин *данные* используется здесь не только по отношению к основному входу и выходу, но и по отношению к данным, определяемым внутри модуля (счетчики, переключатели, рабочие области и т. д.). Форматы записей должны быть зафиксированы до начала процесса детализации. В процессе детализации проясняются потребности в дополнительных элементах данных. Каждый такой элемент нужно определить, т. е. решить, когда и где ему нужно присвоить начальное значение, где он используется (проверяется, накапливается, пересылается) и где восстанавливается его начальное значение. Таким образом, и данные, и логика модуля должны проявляться параллельно.

Нужно ли модули сегментировать, зависит от правил, принятых в вашей организации. Иногда они требуют, чтобы модуль был не больше 50—60 строк; тогда без сегментирования не обойтись. По своей сути сегментирование — это проектирование программ сверху вниз на заключительных стадиях создания модулей. Сегмент самого верхнего уровня может содержать главный цикл программы, а также подготовительные шаги. Сегменты нижних уровней реализуют логические подфункции общей функции рассматриваемого модуля. Если модуль создавался методом пошаговой детализации, то совсем нетрудно выделить те фрагменты, которые логично оформить в качестве отдельных сегментов.



## Контрольные вопросы и упражнения

1. Объясните, что такое пошаговая детализация.
2. Как вы узнаете, что продвинулись уже достаточно далеко в процессе пошаговой детализации?
3. Каких принципов вы должны придерживаться в процессе расширения и детализации?
4. Каковы преимущества псевдокода перед блок-схемами при пошаговой детализации?
  - а) преимущества блок-схем:
  - б) преимущества псевдокода:
5. Нужно ли применять пошаговую детализацию для всех программ, которые вам потребуется разрабатывать? Почему?
6. Какого типа информация должна быть включена в *список данных*?
7. В примере с загрязнением воздуха было сделано предположение, что имеются все данные за 24 часа. Проверка на конец файла не делалась. Как бы вы проделали пошаговую детализацию для рассматриваемой задачи без названного предположения? Другими словами, включите проверку того случая, когда файл кончается раньше, чем будут обработаны данные за все 24 часа. Если это обнаружится, нужно только выдать сообщение об ошибке.
8. Станет ли яснее фрагмент, показанный на рис. 5.17 и сравнивающий новый номер агента со старым в сегменте Обработать сделку, если заменить вложенные условные предложения последовательными? Каковы преимущества вложенных условных предложений? Каковы преимущества последовательных?
9. Измените текст, изображенный на рис. 5.19, так, чтобы включить проверку Код=SR, которая устанавливает, что введена действительно запись о сделке, касающейся продажи. Если в поле Код ошибка, то нужно игнорировать введенную запись и напечатать соответствующее сообщение об ошибке.
10. Дополните изображенный на рис. 5.18 текст на псевдокоде сегментами, обрабатывающими *возвраты за текущий месяц*. При этом нужно считать, что возвраты, если они есть, для каждого торгового агента будут сгруппированы вместе с записями о его продажах и будут непосредственно следовать за ними. На иллюстрации SR обозначает Запись о продаже (Sales Record), и RR — Запись о возврате (Returns Record):



Заметим, что для одних агентов могут оказаться только лишь *записи продаж* или только *записи возвратов*, а для других — и те, и другие.

11. Напишите на псевдокоде модуль, генерирующий отчет по информации, содержащейся в файле торговых агентов. Каждая запись в этом файле содержит такие поля:

Номер торгового агента	Имя торгового агента	Объем сбыта за месяц	Возвраты за месяц	Объем сбыта за текущий год	Возвраты за текущий год
------------------------------	----------------------------	----------------------------	----------------------	----------------------------------	-------------------------------

Отчет должен иметь следующий вид:

Имя	Номер	Объем сбыта	Доход	Объем за текущий год	Доход за текущий год
XXXXXXXXXX	XXXX	XXXXX.XX	XXXXX.XX	XXXXXX.XX	XXXXXX.XX
XXXXXXXXXX	XXXX	XXXXX.XX	XXXXX.XX	XXXXXX.XX	XXXXXX.XX
XXXXXXXXXX	XXXX	XXXXX.XX	XXXXX.XX	XXXXXX.XX	XXXXXX.XX
Итого		XXXXXX.XX	XXXXXX.XX	XXXXXX.XX	XXXXXX.XX

Чистый доход вычисляется как *объем продаж текущего месяца* минус *возвраты этого месяца*. Чистый доход за текущий год вычисляется как *общий объем продаж за текущий год* минус *общий объем возвратов за текущий год*.

Файл торговых агентов ведется так, что в конце месяца сведения за текущий месяц складываются с соответствующими сведениями за текущий год, а месячные поля обнуляются.

## Структурное программирование на стандартном КОБОЛе

Уже успешно реализовано много больших структурных программ на КОБОЛе, подтверждая тем самым, что структурный подход на базе КОБОЛа не только возможен, но и удобен. Следование, развилка и повторение присутствуют в самом языке, а выбор можно легко промоделировать, причем так, что его будет легко и писать, и читать. Модули можно компилировать раздельно, а данные передавать явно как аргументы. Сегменты в модуле доступны с помощью предложения ВЫПОЛНИТЬ (PERFORM). Существуют, однако, ограничения на использование предложений ЕСЛИ и ВЫПОЛНИТЬ, неоправданно затрудняющие написание и чтение некоторых структурных программ. Чтобы преодолеть это неудобство, предложено много изменений, добавлений и расширений языка как в литературе, так и в качестве официальных предложений групп пользователей КОБОЛа <sup>1)</sup>.

В середине 60-х годов, когда появился КОБОЛ, компиляторы не были способны создавать особо эффективные программы. Это положило начало легендам о нежелательности некоторых конструкций (например, ВЫПОЛНИТЬ или вложенных ЕСЛИ). К тому же в это время основное внимание уделялось эффективности работы не человека, а ЭВМ. В результате основной целью запрета или разрешения использовать те или иные конструкции языка было достижение максимальной скорости выполнения программы или минимизация требуемой оперативной памяти. Теперь же основное внимание уделяется производительности человека, пишущего и сопровождающего программы. В соответствии с этой тенденцией совсем новые цели влияют на отбор предпочтительных конструкций языка, и эти цели — легкость чтения и сопровождения программы.

В этой главе будут рассмотрены некоторые методы разработки структурных программ на КОБОЛе и рекомендованы типовые приемы программирования, которые могут служить основой или частью набора стандартных приемов, принятых в конкретной организации. Кроме многочисленных мелких примеров, будет приведена полная структурная программа на КОБОЛе.

---

<sup>1)</sup> Например, симпозиум CODASYL по структурному программированию на КОБОЛе, Los Angeles, Calif., April 1975.

## Модульность

### Организация модуля

На рис. 6.1 приведена схема иерархии модулей в пакете экономических программ. Каждый прямоугольник обозначает отдельный модуль в этом пакете. В идеале каждый модуль в программе на КОБОЛе должен был бы допускать отдельную компиляцию. Главный модуль мог бы передавать управление подчиненным модулям (подпрограммам) с помощью предложения **ВЫЗВАТЬ (CALL)**. Данные могли бы передаваться подпрограммам с помощью конструкции **ИСПОЛЬЗУЯ (USING)**, где указываются элементы данных, используемых в вызываемой подпрограмме <sup>1)</sup>.

На это можно возразить, что потребуются дополнительные накладные расходы на программирование, компиляцию и выполнение связывающих действий. Однако преимущества отдельной компиляции перевешивают ее недостатки. Программу становится легче проверять и изменять, легче организовать библиотеку наиболее часто используемых модулей.

Модули на рис. 6.1 разрабатывались сверху вниз, слева направо, причем до начала разработки следующей ветви целиком завершалась предыдущая. Предположим, что завершен самый левый путь, и мы хотим приступить к программированию модуля (заштрихованного на рис. 6.1), подготовливающего регистр счетов (рис. 6.2). Объем этого модуля, как и всех других, должен быть ограничен, чтобы с ним было легче работать.

В некоторых организациях, использующих методы структурного программирования на КОБОЛе, считается разумным ограничивать объем модуля одной страницей печатающего устройства. В других организациях допускают модули большего размера, разделяя их на сегменты по 50—60 строк. (Обычно в сегменте 25—40 строк.) Сегментирование, о котором идет речь в этой главе — это не то же самое, что модуль **СЕГМЕНТИРОВАНИЕ (SEGMENTATION)** в КОБОЛе, который позволяет осуществлять динамическое перекрытие указанных пользователем подпрограмм.

### Как сегментировать модуль

Предположим, что модуль **СОСТАВИТЬ РЕГИСТР СЧЕТОВ** уже был написан на псевдокоде. Оценив его объем, можно прийти к выводу, что модуль будет занимать более 60 строк исходного текста и его нужно сегментировать. Среди элементов записей,

---

<sup>1)</sup> К сожалению, некоторые компиляторы не позволяют связывать подпрограммы.

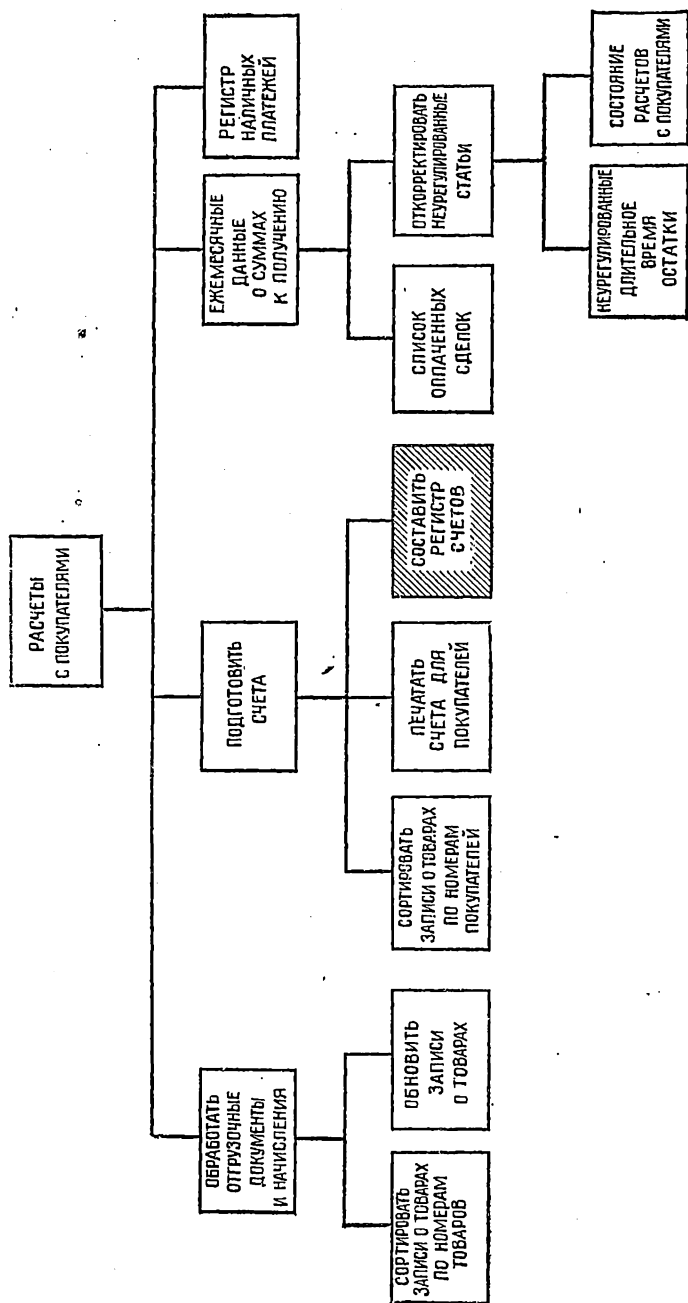


Рис. 6.1. Модуль в программе расчетов с покупателями

РЕГИСТР СЧЕТОВ					
Номер счета	Номер покупателя	Имя покупателя	Стоимость товара	Налог	Общая стоимость
16799	14753	HOBBYRAMA	52.50	2.61	54.81
16800	14758	HOFF ELEC CO	86.12	4.30	90.42
16801	14764	HOLBRO PAINT CO	92.20	4.61	96.81
16802	14770	HOLLAND CO	80.15	4.01	84.16
16803	14775	HOVER CO	912.12	45.61	957.73
16804	14780	HOWELL CO	200.10	10.00	210.10
16805	14787	HOWKEN UTILITIES	315.00	15.75	330.75
16806	14790	HUGEL MOTORS	930.30	46.50	
16807	14792	IRVING TIRE CO.	1,950.00		
16808	14795	IVY BARN INC.			
16809	14800	JACKSON SALES			
			00	2.02	672.42
			1,050.10	40.00	840.10
			280.50	92.51	1,142.61
			555.10	14.03	294.53
			106.66	27.76	582.86
			1,133.00	5.33	111.99
				56.65	1,189.65
16843	15970	ZACHARY BROS.			
16844	15972	ZEPHYR ELEC CO			
Всего			61,564.67	3,078.23	64,642.90

Рис. 6.2. Образец регистра счетов.

которые обрабатывает этот модуль, имеются, в частности, следующие поля:

Номер покупателя	Номер торгового агента	Номер счета	Номер товара	Объем закупок	Цена	Описание
------------------	------------------------	-------------	--------------	---------------	------	----------

Сегментировать лучше всего на этапе пошаговой детализации. После нескольких проходов становится видно, будет ли модуль занимать более страницы текста. Предположим, что после этого этапа текст нашего модуля выглядит так:

```
Присвоить начальные значения переменным
Конец задания = нет
ЦИКЛ-ПОКА есть элементы записи
  Читать элемент записи
  В конце задания конец задания = да
  ЕСЛИ есть элементы записи
    Проверить данные
    ЕСЛИ данные в порядке
      ВЫПОЛНИТЬ обработку элемента
    ИНАЧЕ
      ВЫПОЛНИТЬ ошибку в данных
  ВСЕ-ЕСЛИ
ИНАЧЕ
  ВЫПОЛНИТЬ окончательный итог
ВСЕ-ЕСЛИ
ВСЕ-ЦИКЛ
```

Этот вариант псевдокода несколько отличается от варианта, применявшегося в предыдущих главах. Это естественно, ведь формального определения псевдокода нет — это просто способ записи, близкий к конкретному языку, используемому при программировании. Приведенный выше текст можно теперь перевести на КОБОЛ и рассматривать как *головной сегмент* модуля. Головной сегмент определяет устройство оставшейся части модуля. Итак, модуль сегментируется тем же иерархическим способом, как и вся система или программа.

На рис. 6.3 приведена схема иерархии сегментов модуля, готовящего регистр счетов. Подчиненные сегменты можно оформить как параграфы и активизировать их с помощью **ВЫПОЛНИТЬ**. Кроме того, сегментами могут быть подпрограммы или фрагменты программ, хранящиеся в текстовой библиотеке программ КОБОЛА. В этом случае сегмент включается в программу предложением **КОПИРОВАТЬ (COPY)**. Использование модуля

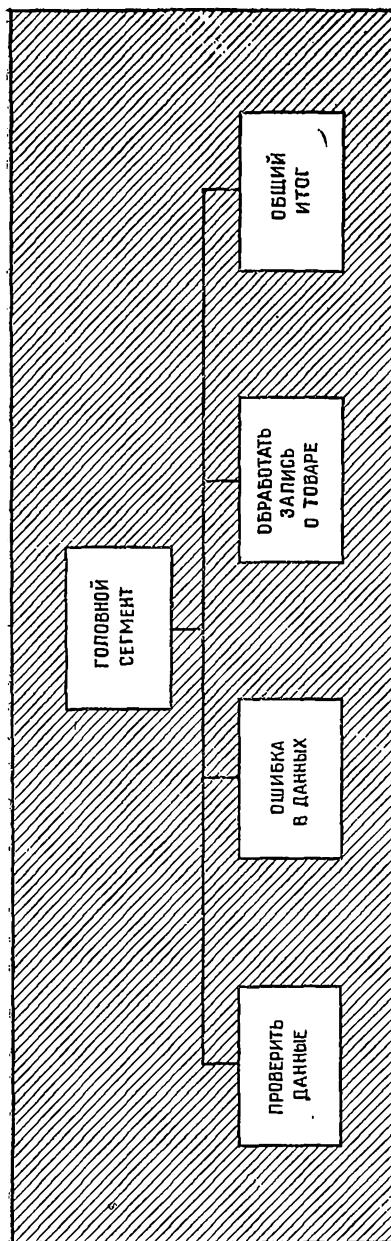


Рис. 6.3. Сегменты модуля, готовящего регистр счетов.



БИБЛИОТЕКА (LIBRARY) позволяет уменьшить объем программирования и облегчает сегментирование.

Техника разработки сегментов, хранящихся в текстовой библиотеке, такова: пусть написан некоторый сегмент, использующий подчиненный сегмент. Этот подчиненный сегмент может первоначально быть запрограммирован как заглушка (содержащая лишь предложения, необходимые для правильной работы вызывающего сегмента) и помещен в текстовую библиотеку. После тестирования сегмента верхнего уровня заглушка преобразуется в настоящий сегмент. Предложение КОПИРОВАТЬ должно стоять в вызывающем сегменте сразу после имени процедуры. Заметим, что в стандарте КОБОЛа запрещается копировать текст, в свою очередь содержащий предложение КОПИРОВАТЬ.

До начала программирования этого модуля необходимо подготовить тестовые данные и соответствующий текст на языке управления заданиями. Первым программируется головной сегмент. Конечно, для проверки сегментов более высокого уровня необходимо запрограммировать все подчиненные сегменты — либо как заглушки, либо сразу как настоящие сегменты. В нашем примере (обработка регистра счетов) все сегменты (кроме головного) можно сначала реализовать как заглушки. После тестирования головного сегмента программируются и тестируются остальные сегменты. Наилучший метод планирования порядка программирования и тестирования подчиненных сегментов — это комбинация иерархического и операционного подходов. Например, в нашем случае этот порядок может быть таким: 1) Головной сегмент; 2) Обработать запись; 3) Общий итог; 4) Проверить данные и 5) Ошибка в данных.

Однако такой порядок может оказаться не лучшим для расположения параграфов в окончательном тексте программы. Легче читать программу, в которой параграфы расположены в соответствии с порядком их исполнения; таким образом, по окончании разработки модуля может оказаться нужным переставить параграфы.

Хотя сегменты — это, как правило, параграфы, не каждый параграф является сегментом. Другими словами, можно выделить параграф, руководствуясь не только сегментированием модуля. Вот причины, из-за которых возникают дополнительные параграфы:

- синтаксические ограничения КОБОЛа, например оформление тела цикла;
- стремление к ясности;
- необходимость выделить повторно используемую часть программы.

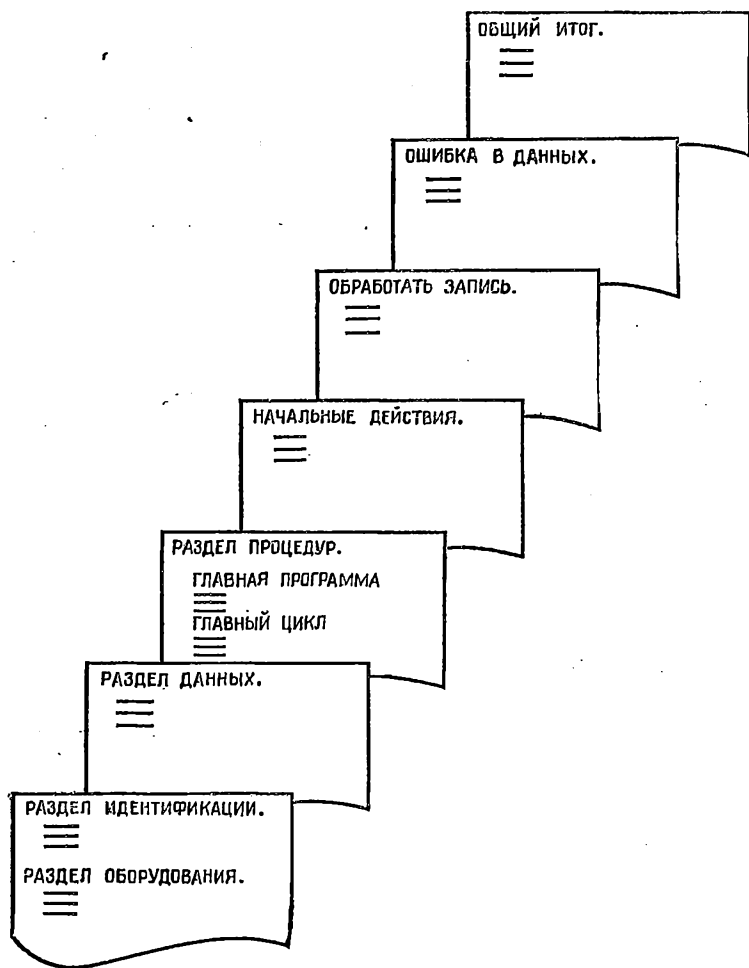


Рис. 6.4. Пример сегментированного модуля КОБОЛа.

Параграфы лучше всего помещать на той же странице, что и содержащий их сегмент. Если параграф сам величиной со страницу, нужно выделять его в отдельный сегмент.

На рис. 6.4 показана организация сегментов модуля на КОБОЛе, готовящего регистр счетов. Как уже говорилось, сегментация — это по существу то же самое, что и нисходящая разработка. Сначала идет *основной*, или *головной*, сегмент, который определяет основные этапы работы всего модуля. В этом сегменте обычно описываются следующие шаги:

— *подготовка*: открытие файлов, присваивание начальных значений переменным;

— *главный цикл*: обычно многократно повторяется и заканчивается по условию исчерпания входного файла;

— *завершение*: общие итоги, завершающее сообщение, закрытие файлов.

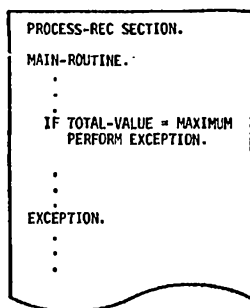
Например, головной сегмент в нашем примере мог бы выглядеть так:

```
"      PROCEDURE DIVISION.  
      MAIN-PROGRAM.  
      PERFORM INITIALIZATION.  
      PERFORM MAIN-LOOP UNTIL EOJ=YES.  
      PERFORM FINAL-TOTALS.  
      STOP RUN.  
  
      MAIN-LOOP.  
      READ ITEM INTO ITEM-REC  
      AT END MOVE 'YES' TO EOJ.  
      IF EOJ=NO  
      PERFORM VALIDATE-DATA  
      IF DATA=OK  
      PERFORM PRDCESS-ITEM  
      ELSE  
      PERFORM DATA-ERROR.
```

Должен ли сегмент быть секцией? Преимущества оформления сегмента как секции и использования ее с помощью ВЫПОЛНИТЬ заключаются в объединении отдельных параграфов и тем самым в лучшем соответствии нисходящему подходу. Однако к недостаткам такого метода относятся сложность понимания программы и трудность сопровождения. Каждый, кто читает программы на КОБОЛе, хотел бы точно знать, будет ли встретившийся параграф активизирован из некоторого сегмента более высокого уровня с последующим возвратом в вызвавший его сегмент или же никогда не будет подобным образом «начат» или «закончен». Тогда при анализе фрагмента не надо искать по всей программе, где этот параграф активизируется и каким способом. Если параграф выполняется иногда отдельно, а иногда как часть секции, то трудно понять его взаимосвязь с работой соседних параграфов. Кроме того, если такой параграф вставлен между не связанными с ним или, наоборот, находится вне естественной последовательности, то это может сильно затруднить отладку.

Если же параграф, входящий в некоторую секцию, активизирует с помощью ВЫПОЛНИТЬ другой параграф этой секции, то возникают дополнительные трудности. Если выполняется

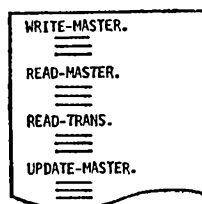
вся секция, то второй параграф будет выполнен лишний раз. Например:



Когда выполняется секция PROCESS-REC, параграф EXCEPTION будет выполнен после завершения параграфа MAIN-ROUTINE.

Чтобы избежать этого, не следует применять ВЫПОЛНИТЬ к целой секции. Однако имя секции можно употреблять для объединения параграфов при документировании, сегментировании или потому, что в некоторых предложениях, например СОРТИРОВАТЬ (SORT), имя секции требуется по синтаксису.

Аналогичное правило справедливо и для конструкции ВЫПОЛНИТЬ... ПО (PERFORM THRU). Если она используется для выполнения различных последовательностей параграфов, то возникают те же трудности, о которых только что говорилось. Рассмотрим, например, страницу исходного текста:



Когда используется ВЫПОЛНИТЬ... ПО, необходимо проанализировать одну или несколько соседних страниц исходного текста только для того, чтобы определить, как именно вызываются параграфы. В нашем случае их можно вызывать несколькими способами:

PERFORM READ-MASTER THRU  
UPDATE-MASTER.

PERFORM WRITE-MASTER THRU  
UPDATE-MASTER.

PERFORM WRITE-MASTER  
THRU READ-MASTER.

Тот, кто сопровождает эту программу и пожелает внести изменения в параграф READ-MASTER, должен будет также учитывать влияние этих изменений на остальную часть программы. Другими словами, изменения могут иметь совершенно неожиданный эффект для других параграфов.

Существуют, однако, две ситуации, в которых допустима конструкция ВЫПОЛНИТЬ... ПО. Одна из них — когда после ПО указывается параграф, содержащий только ВЫЙТИ (EXIT). Это может быть полезным, когда желательно явно выделить выход из параграфа. Но тогда между активизируемым параграфом и параграфом ВЫЙТИ не следует располагать других параграфов. Второе исключение — такое моделирование структуры ВЫБОР, при котором используется конструкция ВЫПОЛНИТЬ... ПО. Подробно об этом говорится в разделе, посвященном реализации этой структуры в КОБОЛе.

Явная активизация каждого параграфа полезна также тем, что их можно как угодно переупорядочивать, добиваясь ясности программы. Кроме того, может оказаться желательным группировать параграфы, обрабатывающие исключительные ситуации, отдельно от основных. Это не только облегчает понимание программы, но и обладает преимуществами при использовании виртуальной памяти. Таким образом, в хорошей легко понимаемой программе все параграфы должны активизироваться только с помощью ВЫПОЛНИТЬ, ни одна секция не должна активизироваться как целое, и за двумя упоминавшимися выше исключениями не должна использоваться конструкция ВЫПОЛНИТЬ... ПО.

Каждый сегмент и модуль должен быть *простой программой*. Применительно к КОБОЛу это означает следующее:

1. *Имеется один вход — начало страницы.* Таким образом, даже если используемый компилятор КОБОЛа и воспринимает предложение ВХОД (ENTRY), его не следует употреблять для указания нескольких входов.

2. *Имеется один выход — обычно это последнее предложение первого параграфа на странице.* В каждом вызываемом модуле конструкция ВЫЙТИ ИЗ ПРОГРАММЫ (EXIT PROGRAM) может встречаться только один раз. Если к сегменту применялся ВЫПОЛНИТЬ, то его выход — это просто конец параграфа. Допустимо и указание (после ПО) параграфа с ВЫЙТИ.

3. *Этот выход (возврат) осуществляется всегда в активизирующий модуль.* Единственным исключением из этого правила может быть обнаружение неисправимых ошибок и немедленное завершение программы. Например, если неисправимая ошибка была обнаружена в подчиненном сегменте или модуле, то более эффективным или удобным может быть не передача индикатора наверх по иерархии, а немедленное завершение программы. В этом случае в сегменте может использоваться конструкция ОСТАНОВИТЬ РАБОТУ (STOP RUN) или запуск программы аварийного завершения.

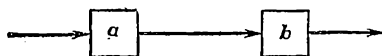
Итак, подведем итоги обсуждения принципов сегментирования программы на КОБОЛе: 1) программа конструируется из модулей; 2) каждый модуль компилируется как подпрограмма (головной — как главная программа); 3) если модуль длиннее 50—60 строк, он делится на сегменты длиной в одну страницу; 4) внутри сегментов могут потребоваться параграфы.

## Структурирование программ на КОБОЛе

Структурирование опирается на три основные структуры, а также некоторые дополнительные и состоит в комбинировании из них заданной функции или подфункции. Ниже рассмотрены эти структуры и примеры их использования в КОБОЛе.

### Следование

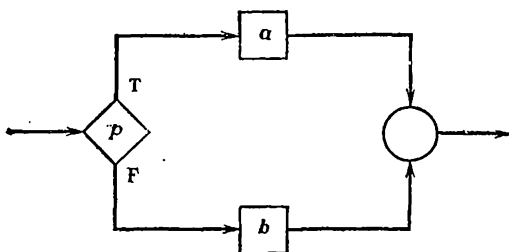
Следование состоит из двух прямоугольников — действий, причем управление передается от предыдущего прямоугольника последующему. Такой прямоугольник может представлять любое предложение КОБОЛа, не нарушающее последовательного исполнения.



Предложения ВЫПОЛНИТЬ и ВЫЗВАТЬ вполне допустимы, поскольку их можно считать сокращением активизируемого фрагмента программы. Например, если некоторый параграф был заменен соответствующим ВЫПОЛНИТЬ, последовательный порядок исполнения очевиден. Если он выносится из текста для удобства или с целью многократного повторения, логика программы не изменяется. Пока не произойдет возврат управления следующему предложению, будут исполняться только подчиненные фрагменты.

### Развилка

#### Развилка



реализуется с помощью предложения ЕСЛИ (IF). Если в каждой из двух возникающих ситуаций нужно выполнить определенные действия, то альтернатива ИНАЧЕ (ELSE) присутствует.

```

IF STOCK-ON-HAND EQUAL ZERO
  PERFORM STOCK-BALANCE-ZERO
ELSE
  MOVE STOCK-ON-HAND TO QUANTITY-WORK.
  
```

Если для ситуации «ложь» никаких действий выполнять не нужно, ИНАЧЕ можно опустить. Например:

```

IF STOCK-ON-HAND LESS THAN REORDER-QUANTITY
  PERFORM PURCHASE-ORDER.
  
```

Иногда для указания конца развилки в качестве *комментария* используется конструкция псевдокода ВСЕ-ЕСЛИ (ENDIF). Чтобы применить эту идею, достаточно поместить в колонку 7 звездочку и вставить ВСЕ-ЕСЛИ как комментарий. Например:

```

      IF AUTHORIZATION-CODE NOT NUMERIC
      '  PERFORM AUTHORIZATION-ERROR-ROUTINE
        MOVE 'NO' TO PROCESS-RECORD.
*  ENDIF

```

ВСЕ-ЕСЛИ — это не слово КОБОЛа и, следовательно, не может употребляться для сообщения компилятору о конце предложения ЕСЛИ — для этого используется точка.

Иногда программист считает удобным выразить логику программы с помощью вложенных предложений ЕСЛИ. Для многих программистов это означает что-нибудь вроде

```

IF NOT A OR B IF C THEN GO TO Q ELSE IF D GO TO R ELSE GO
TO S ELSE GO TO T.

```

Если то же самое написать с использованием правил структурирования и соглашений о формате, получаем следующее:

```

1      IF PAYMENT-LATE OR NOT COMPLETE
2          IF FIRST-TIME
3              PERFORM GENTLE-REMINDER
4          ELSE
5              IF PREVIOUS-LETTER-SENT
6                  PERFORM THREATENING-LETTER
7              ELSE
8                  PERFORM SECOND-REMINDER
9          ELSE
10             PERFORM THANK-YOU.

```

Форматные правила структурного программирования не допускают *сплошной* записи программы, показанной на первом примере вложенных ЕСЛИ. Каждое новое предложение начинается с новой строки и каждый уровень вложенности записывается с отступом относительно предыдущего уровня. Это проясняет вложенность развилки и показывает соответствие между наборами проверок и фрагментами программ. ИНАЧЕ выравнивается по соответствующему ЕСЛИ. Пары ЕСЛИ — ИНАЧЕ обнаруживаются по следующему правилу: нужно сначала сопоставить самое внутреннее ИНАЧЕ непосредственно предшествующему еще не связанному ЕСЛИ, а затем двигаться вверх, пока все ИНАЧЕ не будут перебраны, после чего перейти к следующему самому внутреннему ИНАЧЕ и повторить процесс. Так, в предыдущем примере ИНАЧЕ на строке 7 соответствует ЕСЛИ на строке 5, ИНАЧЕ на строке 4 — ЕСЛИ на строке 2, а ИНАЧЕ на строке 9 не может соответствовать ЕСЛИ на строках 2 и 5 — пары к ним уже подобраны. Следовательно, ИНАЧЕ на строке 9 соответствует ЕСЛИ на строке 1.



Вот несколько соглашений, упрощающих использование вложенных ЕСЛИ:

1) *Имена проверяемых условий должны быть mnemonicескими.* Такие имена, как НАЙДЕН-ГЛАВНЫЙ-ОБРАЗЕЦ или ЕСТЬ-РАЗРЕШЕНИЕ, несут больше информации, чем такое имя, как ПЕРЕКЛ-1. Если используется предложение ВЫПОЛНИТЬ, то имена активизируемых параграфов также должны быть mnemonicескими.

2) *Сложные проверки должны быть сведены к минимуму.* Одно И или ИЛИ легко понять, но их комбинация затрудняет чтение. Когда попадается предложение ЕСЛИ А ИЛИ В И С (здесь нарушено и первое правило), то читателю надо отгадать, что имеется в виду — А ИЛИ (В И С) или (А ИЛИ В) И С. Если очень нужно написать подобное условие, используйте скобки для указания порядка операций, даже если они и необязательны.

3) *Использование НЕ затрудняет понимание условий, особенно сложных.* Его по возможности нужно избегать. Если приходится использовать НЕ со сложным условием, помогите читателю программы расстановкой скобок или изменением порядка компонент.

Так

ЕСЛИ НЕ А ИЛИ В

следует записать

ЕСЛИ В ИЛИ НЕ А

4. *Избегайте подразумеваемых объектов и операций.* Например, трудно запомнить, что в

$A \text{ НЕ } < B \text{ И } > C$

после И подразумевается НЕ, а в

$A \text{ НЕ } B \text{ И } C$

оно не подразумевается. Программируйте ясно. Например:

$A \text{ НЕ } < B \text{ И } A \text{ НЕ } > C$

5. *Глубина вложенных ЕСЛИ должна быть ограничена.* Один-два уровня проследить легко, больше четырех становится трудно читать. Отступы каждого следующего ЕСЛИ помогут ограничить глубину вложенности, поскольку слишком «глубокие» предложения придется писать у правого края бланка.

6. *Размещайте всю вложенную структуру на одной странице.* Если необходимо, используйте предложение ВЫПОЛНИТЬ. Это позволяет читателю увидеть всю структуру целиком на одной странице. Кроме того, при таком способе сближаются соответствующие ЕСЛИ и ИНАЧЕ.

В прошлом некоторые пользователи КОБОЛа избегали вложенных ЕСЛИ из-за трудностей чтения и отладки. Однако при соблюдении приведенных шести правил эта конструкция предпочтительнее других для выражения нужного смысла. Если программист раздумывает над тем, пользоваться ли ему вложенными ЕСЛИ или нет, то он может попытаться написать оба варианта программы, а затем сравнить легкость их восприятия. Например,

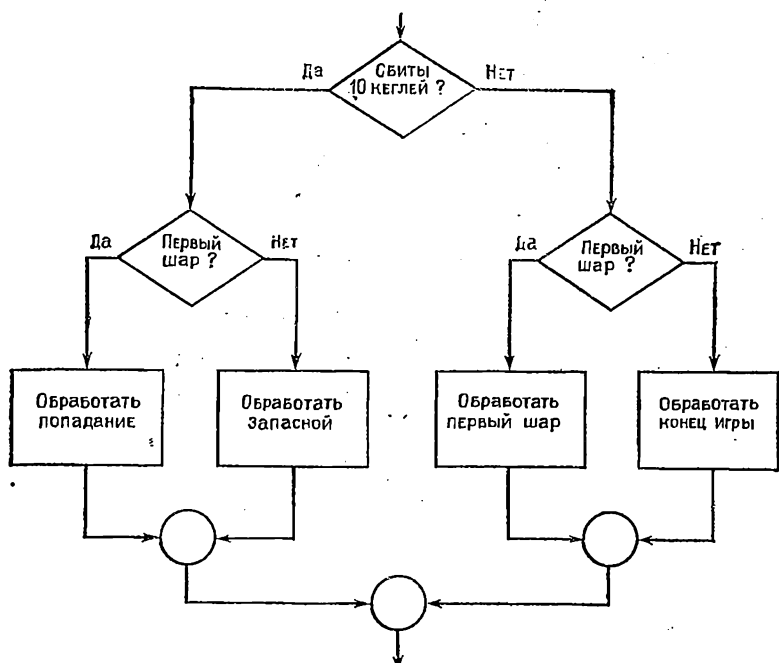


Рис. 6.5. Логическая диаграмма счета при игре в кегли.

на рис. 6.5 показана логика программы подсчета очков при игре в кегли. При использовании вложенных ЕСЛИ можно получить:

```

IF TOTAL-PINS-DOWN = 10
  IF BALL-NUMBER = 1
    PERFORM PROCESS-STRIKE
  ELSE
    PERFORM PROCESS-SPARE
ELSE
  IF BALL-NUMBER = 1
    PERFORM PROCESS-FIRST-BALL
  ELSE
    PERFORM PROCESS-END-OF-FRAME
  
```

Ниже вложенные ЕСЛИ вынесены и активизируются предложением ВЫПОЛНИТЬ

```

    IF TOTAL-PINS-DOWN = 10
        PERFORM TEST-FIRST-BALL-1
    ELSE
        PERFORM TEST-FIRST-BALL-2.
*
*
*
TEST-FIRST-BALL-1.
    IF BALL-NUMBER = 1
        PERFORM PROCESS-STRIKE
    ELSE
        PERFORM PROCESS-SPARE.

TEST-FIRST-BALL-2.
    IF BALL-NUMBER = 1
        PERFORM PROCESS-FIRST-BALL
    ELSE
        PERFORM PROCESS-END-OF-FRAME.

```

Наконец, вот решение без вложенных ЕСЛИ:

```

IF TOTAL-PINS-DOWN = 10 AND BALL-NUMBER = 1
    PERFORM PROCESS-STRIKE.
IF TOTAL-PINS-DOWN = 10 AND BALL-NUMBER NOT = 1
    PERFORM PROCESS-SPARE.
IF TOTAL-PINS-DOWN NOT = 10 AND BALL-NUMBER = 1
    PERFORM PROCESS-FIRST-BALL.
IF TOTAL-PINS-DOWN NOT = 10 AND BALL-NUMBER NOT = 1
    PERFORM PROCESS-END-OF-FRAME.

```

Если ветвь одного из вложенных ЕСЛИ не определяет никаких действий, то ее можно опустить. Однако по синтаксису КОБОЛА и в этом случае что-то нужно написать. Для этого годится конструкция СЛЕДУЮЩЕЕ ПРЕДЛОЖЕНИЕ (NEXT SENTENCE), отсылающая читателя к предложению, следующему за точкой.

```

IF DATE-FIELD NUMERIC
    IF DATE-FIELD < 79001
        MOVE ' DATE TOO SMALL' TO ERROR-MESSAGE
        WRITE PRINT-LINE
    ELSE
        NEXT SENTENCE
ELSE
    MOVE ' DATE NOT NUMERIC' TO ERROR-MESSAGE
    WRITE PRINT-LINE.

```

В этом примере мы не хотим проверять величину элемента DATE-FIELD, пока не убедимся, что она числовая. Опустить ИНАЧЕ СЛЕДУЮЩЕЕ ПРЕДЛОЖЕНИЕ нельзя — разрушится логика программы и будет печататься сообщение об ошибке для правильных данных. Избегайте использования конст-

рукции СЛЕДУЮЩЕЕ ПРЕДЛОЖЕНИЕ на ветви, соответствующей случаю «истина». Лучше изменить проверку условия так, чтобы на ветви «истина» выполнялись некоторые действия, а на ветви «ложь» — СЛЕДУЮЩЕЕ ПРЕДЛОЖЕНИЕ.

Для вложенных ЕСЛИ используется только одна точка, обозначающая конец всего предложения. КОБОЛ не разрешает использовать точку для завершения промежуточного ЕСЛИ. Например, рисунку 6.6 могла, казалось бы, соответствовать следующая программа:

```

1  IF VALID-TRANSACTION = 'Y'
2    PERFORM PROCESS-TRANSACTION
3  ELSE
4    IF ERROR-COUNT = 1
5      MOVE ' ERROR IN TRANSACTION ' TO ERROR-MESSAGE ,
6    ELSE
7      MOVE ' MULTIPLE ERRORS ' TO ERROR-MESSAGE
8    WRITE PRINT-LINE.
```

Однако эта программа неправильна, поскольку предложение на строке 8 будет выполняться только для случая «ложь» в предшествующем ЕСЛИ, а не в любом случае, как следует из рис. 6.6.

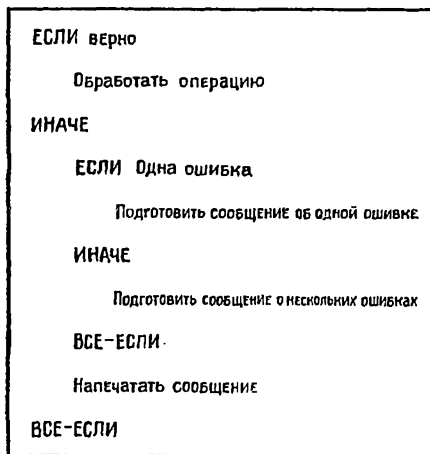


Рис. 6.6. Задача с вложенными ЕСЛИ.

Может показаться, что достаточно лишь поставить точку в конце строки 7 и указать тем самым, что закончено ЕСЛИ строки 4. Однако, так как в предложении ЕСЛИ может быть только одна точка, то будет закончено и ЕСЛИ строки 1. Предложение ПИСАТЬ (WRITE) будет вынесено из ЕСЛИ и будет выполняться всегда, включая случай правильных входных дан-

ных. (Такая ошибка особенно опасна и ее трудно обнаруживать, так как она содержится в формально правильной с точки зрения КОБОЛа программе.)

Задача может быть решена либо повторением предложения ПИСАТЬ в обеих ветвях, либо активизацией внутреннего ЕСЛИ с помощью ВЫПОЛНИТЬ. Эти два варианта приведены на рис. 6.7. В некоторых случаях задачу можно решить, поместив пред-

```

IF VALID-TRANSACTION = 'Y'
    PERFORM PROCESS-TRANSACTION
ELSE
    IF ERROR-COUNT = 1
        MOVE ' ERROR IN TRANSACTION ' TO ERROR-MESSAGE
        WRITE PRINT-LINE
    ELSE
        MOVE ' MULTIPLE ERRORS ' TO ERROR-MESSAGE
        WRITE PRINT-LINE.

```

```

IF VALID-TRANSACTION = 'Y'
    PERFORM PROCESS-TRANSACTION
ELSE
    PERFORM MULTIPLE-ERROR-TEST
    WRITE PRINT-LINE.
*
*
*
MULTIPLE-ERROR-TEST.
    IF ERROR-COUNT = 1
        MOVE ' ERROR IN TRANSACTION ' TO ERROR-MESSAGE
    ELSE
        MOVE ' MULTIPLE ERRORS ' TO ERROR-MESSAGE.

```

Рис. 6,7. Два решения на КОБОЛе задачи из рис. 6,6.

ложение ПИСАТЬ *перед* вторым предложением ЕСЛИ. В данном случае это невозможно — оно должно выполняться *после* второго ЕСЛИ.

Проблема, связанная с внутренней точкой, не ограничивается предложениями ЕСЛИ, а возникает также и в других условных конструкциях. Так, предложение ЧИТАТЬ (READ) — условное, поскольку необязательные конструкции В КОНЦЕ (AT END) или ПРИ ОШИБКЕ КЛЮЧА (INVALID KEY) предваряют предложения, выполняемые лишь при определенных условиях. Такие необязательные конструкции, как ПРИ ПЕРЕПОЛНЕНИИ (ON SIZE ERROR), также приводят к условным предложениям. Вот список условных предложений стандартного КОБОЛа:

Предложение	Конструкция
ЧИТАТЬ (READ), ИСКАТЬ (SEARCH), ВЕРНУТЬ (RETURN)	В КОНЦЕ (AT END)

ПЕРЕЙТИ К (GOTO)	В ЗАВИСИМОСТИ ОТ (DEPENDING ON)
СЛОЖИТЬ (ADD), ОТНЯТЬ (SUBTRACT), УМНОЖИТЬ (MULTIPLY), РАЗДЕЛИТЬ (DIVIDE), ВЫЧИСЛИТЬ (COMPUTE)	ПРИ ПЕРЕПОЛНЕНИИ (ON SIZE ERROR)
ЧИТАТЬ (READ), ПИСАТЬ (WRITE)	ПРИ ОШИБКЕ КЛЮЧА (INVALID KEY)
ИСКАТЬ (SEARCH)	КОГДА (WHEN)
ПИСАТЬ (WRITE)	В КОНЦЕ СТРАНИЦЫ (AT END-OF-PAGE)
ЕСЛИ (IF)	
ВЫПОЛНИТЬ (PERFORM)	ДО (UNTIL)

Рассмотрим, например, следующую программу на КОБОЛе <sup>1)</sup>:

```

READ MASTER-FILE
  INVALID KEY MULTIPLY OLD-KEY BY PRIME-GIVING NEW-KEY
    ON SIZE ERROR
      DIVIDE NEW-KEY BY 2 GIVING NEW-KEY
    MOVE 'Y' TO BAD-KEY.

```

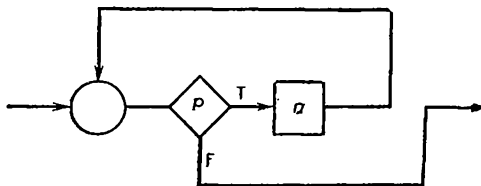
Нет простого способа указать, что ПЕРЕМЕСТИТЬ (MOVE) является частью конструкции ПРИ ОШИБКЕ КЛЮЧА, но не входит в конструкцию ПРИ ПЕРЕПОЛНЕНИИ. Здесь необходимо использовать одно из двух решений, приведенных ранее для случая вложенных ЕСЛИ.

### *Повторение*

Структура ЦИКЛ-ПОКА. Эта структура позволяет строить циклы, в которых некоторое действие повторяется до тех пор, пока остается истинным указанное в ней условие. Значение этого условия проверяется до выполнения предложений внутри этой структуры; следовательно, цикл может не выполняться ни разу. Графически эта структура изображается так:

---

<sup>1)</sup> Строго говоря, эта программа неправильная, так как в конструкции ПРИ ОШИБКЕ КЛЮЧА допустимо только простое повелительное предложение, а УМНОЖИТЬ с конструкцией ПРИ ПЕРЕПОЛНЕНИИ — условное. Однако многие компиляторы воспринимают и такую программу.



Для реализации повторения на языке КОБОЛ используется предложение ВЫПОЛНИТЬ с конструкциями ДО (UNTIL) или РАЗ (TIMES). Может пригодиться также и конструкция МЕНЯЯ (VARYING).

В конструкции ЦИКЛ-ПОКА указанное действие должно повторяться, пока проверяемый предикат истинен, и закончиться, когда он становится ложным. Таким образом, указываемое после ДО условие должно быть отрицанием того предиката, который указан на соответствующей блок-схеме или в тексте на псевдокоде. Например, если действие должно повторяться, пока выключен индикатор конца файла, то следует написать предложение КОБОЛа такого вида:

```
PERFORM PROCESS-ROUTINE UNTIL MASTER-FILE-EOF=YES..
```

Еще пример: предложение псевдокода

### ЦИКЛ 24 РАЗА

реализуется, скажем, таким предложением КОБОЛа:

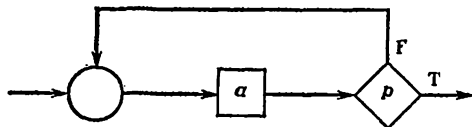
```
PERFORM CALCULATION-ROUTINE VARYING COUNTER FROM 1 BY 1
UNTIL COUNTER GREATER THAN 24.
```

Правила псевдокода задают повторение, пока счетчик меньше либо равен 24, а предложение КОБОЛа определяет повторение до тех пор, пока счетчик не станет больше 24.

Если, как часто бывает, известно, на каком языке программирования будет писаться программа, то блок-схема или псевдокод могут разрабатываться с учетом этой информации. Так, в некоторых организациях допускается версия псевдокода, близкого к КОБОЛу. В такой версии повторение может, например, изображаться «вынесенной» конструкцией ВЫПОЛНИТЬ название параграфа ДО, а не с помощью «охватывающей» ЦИКЛ-ПОКА.

**Структура ЦИКЛ-ДО.** Если требуется выполнить цикл по крайней мере один раз, используется конструкция ЦИКЛ-ДО.

Вот ее основная структура:



В предложении ВЫПОЛНИТЬ...ДО проверка делается до выполнения обработки. Это согласуется с семантикой структуры ЦИКЛ-ПОКА, но может оказаться препятствием для реализации ЦИКЛ-ДО. В последнем случае можно воспользоваться одним из двух способов:

1. Явно активизировать параграф с помощью ВЫПОЛНИТЬ без повторения перед его активизацией с помощью конструкции ВЫПОЛНИТЬ...ДО. Например:

```
PERFORM EDIT-ROUTINE.  
PERFORM EDIT-ROUTINE UNTIL EDIT-ERROR-NO.
```

2. Установить индикатор, управляющий нужным ВЫПОЛНИТЬ, непосредственно перед этим предложением. В активизируемом параграфе этот индикатор должен быть установлен так, как нужно для управления последующими повторениями.

```
MOVE 'N' TO TRANSACTION-FILE-EOF.  
PERFORM MAIN-ROUTINE UNTIL TRANSACTION-FILE-EOF=YES.
```

Такое оформление обращения к MAIN-ROUTINE удобно, если в ней содержится предложение ЧИТАТЬ и она должна выполняться по крайней мере один раз.

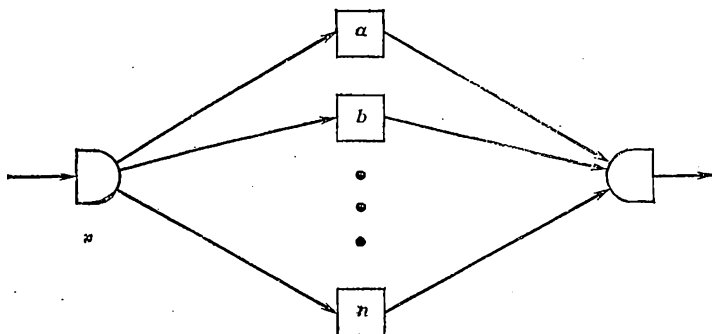
Вообще говоря, предложения ВЫПОЛНИТЬ могут быть вложенными (когда параграф, активизируемый ВЫПОЛНИТЬ, также содержит предложение ВЫПОЛНИТЬ, активизирующее подчиненный параграф). Здесь может содержаться и повторение, управляемое некоторым условием, а также многократно вложенные повторения. Как и в случае вложенных ЕСЛИ, с повышением уровня вложенности программу становится труднее понимать.

### *Дополнительные структуры*

**Выбор.** Развилка предназначена для того, чтобы в зависимости от результата проверки выполнить одну из двух альтернатив. Однако часто необходимо рассматривать более чем две



альтернативы. Для этого используется обобщение развилки — выбор.



Такая структура может быть реализована на КОБОЛе с помощью повторяющихся ЕСЛИ, указывающих различные пути для каждого результата. Эти предложения могут быть, а могут и не быть вложенными. Например, рассмотрим программу, которая обрабатывает различными способами сделки с покупателями. Тип покупателя определяется полем входной записи, называемым CUSTOMER-TYPE:

```
IF CUSTOMER-TYPE EQUAL TO 'C'
  PERFORM CASH-ACCOUNT
ELSE IF CUSTOMER-TYPE EQUAL TO 'T'
  PERFORM THIRTY-DAY-ACCOUNT
ELSE IF CUSTOMER-TYPE EQUAL TO 'R'
  PERFORM REVOLVING-ACCOUNT
ELSE IF CUSTOMER-TYPE EQUAL TO 'S'
  PERFORM SPECIAL-ACCOUNT.
```

Управление передается первому ЕСЛИ и вызывает выполнение одного из указанных параграфов. По окончании этого параграфа управление передается предложению, следующему за заключительной точкой. Правила формата для таких вложенных ЕСЛИ отличаются от отступов при использовании упоминавшихся ранее вложенных ЕСЛИ. Так как все ИНАЧЕ — альтернативы первому ЕСЛИ, то именно это и отражает принятый здесь порядок отступов. Такие правила позволяют записывать большое число альтернатив и при этом не выйти за счет отступов за правый край бланка. Можно и вообще не писать ИНАЧЕ, а просто ограничиться последовательностью ЕСЛИ, проверяющих каждую альтернативу.

- Другой способ реализации выбора — использование конструкции ПЕРЕЙТИ К ... В ЗАВИСИМОСТИ ОТ ... (GOTO DEPENDING ON). Идеальным было бы такое предложение,

в котором выбор исполняемого параграфа определяется указанной в этом предложении переменной. Это вполне проявило бы логику выбора и потребовало бы лишь одного предложения. Однако в КОБОЛе такого предложения нет — приходится создавать требуемые функции с помощью предложения ПЕРЕЙТИ К. Поэтому когда требуется выбор, для его моделирования оказывается полезным ВЫПОЛНИТЬ... ПО. (Выбор — это как раз исключение из правила, запрещающего применять ВЫПОЛНИТЬ... ПО.) Первый параграф, активизируемый внешним ВЫПОЛНИТЬ... ПО, начинается с ПЕРЕЙТИ К... В ЗАВИСИМОСТИ ОТ.... Это предложение передает управление одному из выписанных следом друг за другом параграфов, каждый из которых заканчивается переходом к параграфу ВЫЙТИ, завершающему эту серию параграфов. Таким образом, управление попадает от начального ПЕРЕЙТИ К заключительному параграфу с единственным предложением ВЫЙТИ; имя этого параграфа и указывается во внешнем предложении ВЫПОЛНИТЬ... ПО.

На рис. 6.8 показан упоминавшийся выше пример, но вместо поля CUSTOMER-TYPE используется переменная CUSTOMER-CODE. Тело выбора вынесено из программы и активизируется с помощью ВЫПОЛНИТЬ... ПО — этим обеспечивается цельность и понятность конструкции. Все пути сходятся в параграфе ВЫЙТИ, включая и тот случай, когда значение переменной неправильное и происходит вызов процедуры обработки ошибок.

Часто желательно объяснить, что именно делают вызываемые ВЫПОЛНИТЬ параграфы. Тогда комментарии следует располагать на отдельной странице. Серию параграфов-альтернатив также нужно размещать на одной странице. Если нужно, в этих параграфах для сокращения следует пользоваться предложениями ВЫПОЛНИТЬ. Поскольку в ПЕРЕЙТИ К... В ЗАВИСИМОСТИ ОТ... значения переменной должны быть последовательными, эту конструкцию можно использовать лишь при упорядоченности этих значений или после их преобразования к требуемому виду.

Какой же способ следует применять — вложенные ЕСЛИ или ПЕРЕЙТИ К... В ЗАВИСИМОСТИ ОТ...? Преимущества первого способа в том, что проверяемые значения не обязаны быть последовательными целыми числами, исходная программа обычно короче, не используются операторы перехода. Если же альтернатив много и проверяемые значения последовательные, лучше использовать второй способ, который в этом случае дает более короткую и более понятную программу, поскольку для активизации соответствующего фрагмента требуется единственное предложение. Конечно, второй способ полезен и тогда, когда компилятор ограничивает глубину вложенных ЕСЛИ.

```

***   .   BASED ON THE TYPE OF ACCOUNT, THE FOLLOWING PERFORM
***       WILL CAUSE THE TRANSACTION TO BE PROCESSED
***       BY THE APPROPRIATE ROUTINE.
        PERFORM ACCOUNT-PROCESSING THRU ACCOUNT-PROCESSING-EXIT.

```

```

ACCOUNT-PROCESSING.
    GO TO CASH-ACCOUNT
        THIRTY-DAY-ACCOUNT
        REVOLVING-ACCOUNT
        SPECIAL-ACCOUNT
            DEPENDING ON CUSTOMER-CODE.
    PERFORM INVALID-CODE-ROUTINE.
    GO TO ACCOUNT-PROCESSING-EXIT.

CASH-ACCOUNT.
*       .
*       .
*       .
    GO TO ACCOUNT-PROCESSING-EXIT.

REVOLVING-ACCOUNT.
*       .
*       .
*       .
    GO TO ACCOUNT-PROCESSING-EXIT.

SPECIAL-ACCOUNT.
*       .
*       .
*       .
    GO TO ACCOUNT-PROCESSING-EXIT.

THIRTY-DAY-ACCOUNT.
*       .
*       .
*       .
    GO TO ACCOUNT-PROCESSING-EXIT.

ACCOUNT-PROCESSING-EXIT.
EXIT.

```

Рис. 6.8. Пример реализации структуры ВЫБОР с помощью ПЕРЕЙТИ К... В ЗАВИСИМОСТИ ОТ...

**Поиск.** Поскольку просмотр таблиц или поиск в таблицах встречается в программировании очень часто, была предложена структура ПОИСК. При поиске в таблице *ключ поиска* сравнивается с *ключом таблицы*. (Обычно подразумевается точное совпадение, но это не всегда так. Можно искать первый элемент, превосходящий ключ поиска.) *Табличная функция* — это содержательная обработка записи или других данных, когда элемент *найден*. Если ни один из ключей таблицы не совпал с ключом поиска, возникает ошибочная ситуация. Таким образом,

имеются два способа завершения поиска: *нормальный выход*, когда требуемый элемент найден, и *выход по ошибке*, когда требуемый элемент не найден.

Структура, обеспечивающая *последовательный* и (или) *двоичный поиск*, приведена на рис. 6.9. Вначале управление находится в вершине этой структуры. Затем определяется элемент таблицы, который следует сравнить с ключом поиска. При этом может увеличиваться значение индекса при последовательном

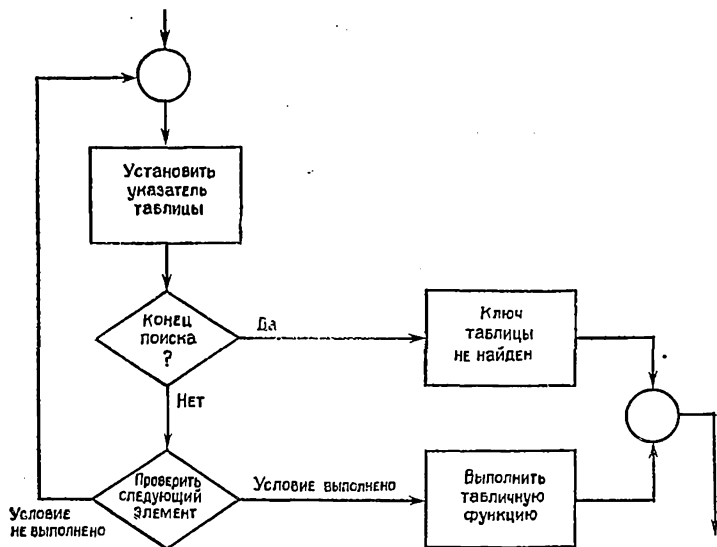


Рис. 6.9. Структура ПОИСК.

поиске или вычисляться верхние и нижние границы при двоичном поиске. После этого проверяется, остались ли еще в таблице непросмотренные элементы. Если нет, то ключ таблицы не был найден — и следует выход по ошибке. Вторая проверка сравнивает ключ таблицы с ключом поиска. Если проверяемое условие (например, ключ таблицы = ключ поиска) *выполнено*, то активизируется требуемый процесс (т. е. ВЫПОЛНИТЬ ТАБЛИЧНУЮ ФУНКЦИЮ). Если же условие *не выполнено*, то цикл продолжается. Структура имеет один вход и один выход. Цикл сравнения может повторяться много раз, но выполнено будет только одно из двух действий.

Поиск реализуется с помощью глагола ИСКАТЬ (SEARCH) в языке КОБОЛ. Этот глагол может быть употреблен либо с дополнительной конструкцией ВСЕ (ALL) (случай двоичного поиска), либо без нее (случай последовательного поиска). В обоих

случаях для условий *найден* и *не найден* определяются разные действия. Они могут быть реализованы либо одним предложением, либо, что более употребительно, активизацией «вынесенного» параграфа с помощью **ВЫПОЛНИТЬ**. Примером структуры поиска может служить

```
SET DEPT-INDEX TO 1.  
SEARCH DEPT-NO-TABLE  
  AT END  
    PERFORM DEPT-NOT-FOUND  
  WHEN DEPT-NO-TABLE (DEPT-INDEX) = INPUT-DEPT-NO  
    MOVE 'Y' TO DEPT-VALID.
```

## О стиле программирования

### *Предложения КОБОЛа*

Следует осторожно пользоваться перечисленными ниже предложениями, поскольку их употребление, вообще говоря, может противоречить принципам структурного программирования.

**ВХОД.** Предложение **ВХОД** определяет входную точку в подпрограмме на КОБОЛе. Оно может встречаться не более одного раза и не должно использоваться для определения дополнительных входных точек.

**ОСТАНОВИТЬ РАБОТУ.** Поскольку это предложение останавливает программу, оно должно встречаться только однажды в логическом конце программы. Единственным исключением, о котором говорилось выше, может быть использование этого предложения для аварийного завершения программы (в подчиненном модуле).

**ИЗМЕНИТЬ.** Армстронг утверждает, что глагол **ИЗМЕНИТЬ** (**ALTER**) — это одна из тех возможностей языка, которые следует исключить из руководств по КОБОЛу<sup>1)</sup>. Этот глагол затрудняет не только отладку, но и чтение программ. Читая программу, содержащую **ИЗМЕНИТЬ**, нельзя доверять тому, что читаешь. Армстронг предлагает следующие способы замены этого глагола: 1) переупорядочить программу; 2) использовать **ВЫПОЛНИТЬ**; 3) ввести индикатор (флажок) и использовать предложение **ЕСЛИ**.

---

<sup>1)</sup> Russell M. Armstrong, "Modular Programming in COBOL", John Wiley and Sons, 1973, p. 121.

**ПЕРЕЙТИ К . . . В ЗАВИСИМОСТИ ОТ . . . .** Это предложение может использоваться только для моделирования выбора (когда это делается не посредством вложенных ЕСЛИ).

**ПЕРЕЙТИ К . . . .** Необходимо запретить свободно пользоваться этим предложением. Обычно считается допустимым применять его только для моделирования базовых управляющих структур. Так, реализация выбора с помощью предложений **ПЕРЕЙТИ К . . .** структурна — все пути приводят в общую точку слияния в этой структуре. Аналогично конструкция **СЛЕДУЮЩЕЕ ПРЕДЛОЖЕНИЕ** иногда рассматривается как тонко замаскированное **ПЕРЕЙТИ К**, поскольку тоже передает управление. Однако здесь происходит передача управления лишь вперед к концу текущей структуры и, чтобы ее понять, не нужно проследивать логику программы или переворачивать страницы. Следовательно, ее употребление оправдано, если только нельзя переписать предложение ЕСЛИ так, чтобы избавиться от этой конструкции.

Более спорно использование конструкций **ПЕРЕЙТИ К** параграфу **ВЫЙТИ**. Некоторые считают это простой и желательной конструкцией в следующем случае: при выполнении некоторого параграфа проверяется условие, определяющее пропуск оставшейся части параграфа. Это можно запрограммировать проверкой указанного условия и, если оно истинно, продолжением выполнения. Таким образом, оставшаяся часть параграфа есть компонента предложения ЕСЛИ. При этом могут возникнуть проблемы, связанные с ограничением на вложенность ЕСЛИ, обсуждавшиеся выше, и программа может стать менее понятной. Простое предложение

#### **ЕСЛИ ОШИБКИ ПЕРЕЙТИ К П-ВЫЙТИ.**

не должно вызвать путаницу и может рассматриваться как «допустимое». Однако, если таких предложений в параграфе несколько, программа *все-таки* будет менее понятной. Мы рекомендуем следующий подход:

1. Сначала запрограммировать *без* **ПЕРЕЙТИ К**. Результат может оказаться лучше, чем ожидалось. Если это так, остановитесь на нем.

2. Если это не так, запрограммировать с **ПЕРЕЙТИ К**. Результат может оказаться не лучше, чем в первом случае. Если это так, выбросьте его.

3. Если это не так, и вы убеждены, что существенное улучшение привнесено **ПЕРЕЙТИ К**, остановитесь на полученном решении. В организациях, стандарты которых не позволяют следовать этому способу, необходимо получить специальное разрешение руководства.

## Индикаторы

Структурное программирование заставляет более интенсивно использовать *индикаторы*, или *переключатели*. Приведем некоторые правила их применения.

1. *Стремитесь уменьшить число индикаторов.* Для этого нужно добиваться, чтобы сам вызываемый параграф содержал реакции на возникающие разнообразные ситуации, а не передавал бы сразу управление наверх. Например, в приведенной ниже программе индикатор используется для сигнализации об ошибке, обнаруженной в программе редактирования. По окончании программы индикатор проверяется, и при необходимости активизируется программа обработки ошибок.

```
      MOVE 'N' TO EDIT-ERROR.  
      PERFORM EDIT-ROUTINE.  
      IF EDIT-ERROR=YES  
        PERFORM ERROR-ROUTINE.  
*  
*  
*  
*  
      EDIT-ROUTINE.  
      IF CUSTOMER-CODE NOT NUMERIC  
        MOVE 'Y' TO EDIT-ERROR.
```

А вот другой способ организации программы редактирования:

```
      PERFORM EDIT-ROUTINE.  
*  
*  
*  
*  
      EDIT-ROUTINE.  
      IF CUSTOMER-CODE NOT NUMERIC  
        PERFORM ERROR-ROUTINE.
```

Здесь при обнаружении ошибки немедленно, без использования индикатора, активизируется соответствующая процедура. Итак, мы встраиваем в этот модуль обработку особых ситуаций. Если программа для этой обработки ошибки занимает слишком много места и ее неудобно помещать непосредственно в модуль, можно использовать **ВЫПОЛНИТЬ** или **ВЫЗВАТЬ**. При этом становится легче следить за нормальным порядком выполнения. Кроме того, если выполняемая программа использует виртуальную память, то это позволяет редко выполняемые фрагменты поместить на отдельную страницу.

2. *Каждый индикатор должен служить только для одной цели.* Использование одного и того же индикатора для нескольких целей затрудняет чтение программы и является источником ошибок. Это особенно сильно сказывается при модификациях программы, когда использование индикаторов изменяется (со-

знательно или случайно). Правило *единственного назначения* индикатора должно соблюдаться даже и в тех случаях, когда индикатор используется в первом смысле в начале программы, а во втором — где-нибудь в конце. Даже в этом случае возможность путаницы или возникновения ошибки — вполне достаточное основание для того, чтобы применить разные индикаторы.

3. *Используйте имя, подчеркивающее назначение индикатора.* (Это еще один довод в пользу второго правила.) Из-за этого имена индикаторов становятся длинными. Выписывать их может показаться скучным, но это ведь дело нескольких секунд. Сравните это время с минутами, потраченными при чтении на отгадывание (возможно, неудачное) назначения сокращенных имен.

Кроме того, что имя индикатора должно быть понятным, присваиваемые ему значения тоже должны что-то говорить читателю. Значения *нуль* и *единица* не всегда содержательно понятны. Лучше использовать одну-две буквы, чтобы было ясно, что делается. Так, некоторые используют «Y» и «N» для «ДА» («YES») и «НЕТ» («NO») или «OK» и «NG» для «Хорошо» («Okay») и «Нехорошо» («No Good») и т. д. Важна мнемоничность в использовании букв. Например, использование «H» для обозначения в одном месте «НЕТ», а в другом — «НА» приводит к путанице. Для повышения мнемоничности индикаторов можно использовать уровень 88, но при этом нужно употреблять содержательные названия. Один из методов — создавать имя для каждого элемента уровня 88 путем добавления к имени уровня 77 значения, присваиваемого индикатору. Например:

77	MASTER-FILE-EOF	PIC X.	
88	MASTER-FILE-EOF-YES		VALUE 'Y'.
88	MASTER-FILE-EOF-NO		VALUE 'N'.
77	TRANSACTION-MASTER-MATCH	PIC XX.	
88	TRANSACTION-MASTER-MATCH-OK		VALUE 'OK'.
88	TRANSACTION-MASTER-MATCH-NG		VALUE 'NG'.
77	OLD-MASTER-EOF	PIC X.	
88	OLD-MASTER-EOF-Y		VALUE 'Y'.
88	OLD-MASTER-EOF-N		VALUE 'N'.

Другой метод — использовать именованные константы и присваивать их значения индикатору. Например:

77	MASTER-FILE-EOF-YES	PIC X	VALUE 'Y'.
----	---------------------	-------	------------



## А в разделе процедур

MOVE MASTER-FILE-EOF=YES TO MASTER-FILE-EOF.

### *Типичные приемы структурного программирования*

Рассмотрим теперь примеры, показывающие, как пользоваться структурами в типичных ситуациях. Сначала займемся обработкой конца файла. Традиционно для этого применяется конструкция В КОНЦЕ предложения ЧИТАТЬ, в которой указан переход к программе обработки конца файла. При структурном программировании это следует переписать так, чтобы управление передавалось только вперед и параграф, содержащий ЧИТАТЬ, имел только один выход. Таким образом, даже при окончании файла управление должно передаваться следующему предложению. Поэтому нужен индикатор конца файла.

Кроме того, само предложение ЧИТАТЬ скорее всего работает в цикле, повторяющемся для каждой записи файла. Повторение обычно обеспечивалось предложением ПЕРЕЙТИ К. Находясь в конце цикла обработки, оно передавало управление к его началу. В структурном программировании необходимое повторение осуществляется предложением ВЫПОЛНИТЬ с указанием имени повторяемого параграфа. Это предложение делает цикл явным. Например:

```
MOVE 'N' TO MASTER-FILE-EOF.
PERFORM MAIN-PROCESS-ROUTINE UNTIL MASTER-FILE-EOF=YES.
*
*   .
*   .
*   .
MAIN-PROCESS-ROUTINE.
  READ MASTER-FILE AT END MOVE 'Y' TO MASTER-FILE-EOF.
  IF MASTER-FILE-EOF=NO
    PERFORM PROCESS-ROUTINE.
```

Для управления циклом вводится индикатор конца файла и ему придается начальное значение NO. Основной цикл выполняется до тех пор, пока индикатор не изменится на YES с помощью конструкции в КОНЦЕ предложения ЧИТАТЬ. Однако даже в этот момент управление передается следующему за ЧИТАТЬ предложению. Логически обработка не должна продолжаться, если встретился конец файла, поэтому необходимо до ее выполнения проверить состояние индикатора конца файла.

Подчеркнем, что MAIN-ROUTINE активизировалась повторно, пока индикатор был равен «N». Когда встречается конец файла, значение MASTER-FILE-EOF меняется на «Y». Для проверки индикатора необходимо предложение ЕСЛИ, чтобы не выполнять параграф PROCESS-ROUTINE. Сама MAIN-ROU-

TINE больше выполняться не будет — индикатор равен «Y». Обработка конца файла программируется вслед за предложением PERFORM MAIN-ROUTINE. Таким образом, читателю явно показана логика работы программы и условия выполнения отдельных предложений. Дополнительные затраты машинного времени на структуризацию связаны только с выполнением лишнего предложения ЕСЛИ.

Есть и другой способ обработки конца файла, при котором можно обойтись без этого лишнего ЕСЛИ. Он заключается в чтении одной записи еще при осуществлении начальных действий. Тогда основной цикл состоит из обработки прочитанной записи, за которой следует программа чтения следующей записи. Цикл по-прежнему управляется индикатором конца файла. Поскольку предложение ЧИТАТЬ завершает цикл, а конструкция В КОНЦЕ «включает» индикатор, он будет проверяться до обработки очередной записи в предложении ВЫПОЛНИТЬ... ДО. Например:

```

MOVE 'N' TO MASTER-FILE-EOF.
PERFORM READ-MASTER.
PERFORM MAIN-PROCESS-ROUTINE UNTIL MASTER-FILE-EOF=YES.
*
*
*

MAIN-PROCESS-ROUTINE.
*
*
*
PERFORM READ-MASTER.

READ-MASTER.
READ MASTER-FILE AT END MOVE 'Y' TO MASTER-FILE-EOF.

```

Этот вариант структурирован, но то, что в основном цикле обработка *предшествует* чтению записи, может несколько смущать читателя. Но этот вариант работает и для случая пустого файла. Если при выполнении первого ЧИТАТЬ возникает конец файла, то, поскольку проверка в ВЫПОЛНИТЬ... ДО стоит *перед* первым выполнением параграфа MAIN-PROCESS-ROUTINE, он не будет выполняться.

Другая общая ситуация возникает в том случае, когда необходимость в некоторых действиях зависит от результатов предыдущей операции и некоторый индикатор управляет тем, нужно или нет выполнять эти действия. В этих случаях индикатор устанавливается при возникновении соответствующей ситуации (например, обнаружена ошибка, запрещающая вызов).

Позднее, когда пишется программа вывода, в ней нужно проверить, разрешен ли вывод. Например:

```
MOVE 'OK' TO RECORD-STATUS.  
PERFORM VALIDATION-ROUTINE.
```

```
OUTPUT-ROUTINE.  
IF RECORD-STATUS=OK  
  PERFORM WRITE-ROUTINE.
```

Здесь при возникновении ошибки в VALIDATION-ROUTINE меняется значение RECORD-STATUS, который затем проверяется в OUTPUT-ROUTINE. Для структурного программирования это стандартный прием.

### *Присваивание начальных значений*

Индикаторам и другим переменным начальные значения следует присваивать в разделе процедур, а не в разделе данных с помощью конструкции ЗНАЧЕНИЕ (VALUE). Делается это затем, чтобы облегчить чтение (когда читаешь раздел процедур, не нужно заглядывать в раздел данных) и обеспечить возможность повторного выполнения загруженной программы. Недостатки — в дополнительных машинных командах, реализующих начальные действия.

С другой стороны, константы следует объявлять в разделе данных с помощью конструкции ЗНАЧЕНИЕ. Это лучше присваивания фактическим константам их значений в процессе счета. Мнемонические имена констант лучше проявляют смысл предложений. Кроме того, становится легче изменять при необходимости значение константы. Таким образом, вместо того, чтобы писать

```
MULTIPLY SALARY BY .14 GIVING TAX.
```

мы определим элемент MINIMUM-TAX-RATE (МИНИМАЛЬНЫЙ-НАЛОГОВЫЙ-ПРОЦЕНТ), придадим ему значение .14 и напомним

```
MULTIPLY SALARY BY MINIMUM-TAX-RATE GIVING TAX.
```

Кроме того, если налог снизится (ха-ха!), надо будет изменить только этот элемент раздела данных.

## Как облегчить чтение программы

Другой аспект структурного программирования — удобство чтения программы. Написанные по форматным правилам программы легче читать и изменять.

### *Разметка страниц*

Для размещения сегментов и разделов на отдельных страницах следует, если это только возможно, применять предложение **ВЫБРОСИТЬ (EJECT)**. В разделе данных все описания записей, занимающие в исходном тексте более одной страницы, должны начинаться с новой страницы. Несколько мелких описаний записей могут располагаться на одной странице.

Поскольку параграфы должны активизироваться извне, каждый параграф становится более самостоятельным и его конец лучше выделять. Для разделения параграфов, размещенных на одной странице, используется предложение **SKIP** или пустая строка. Две-три такие строки между параграфами делают программу более обзримой.

Хотя бывают случаи, когда порядок выполнения параграфов меняется или неизвестен, по возможности они должны располагаться в порядке их исполнения. Так лучше всего и для чтения и работы с виртуальной памятью. Не следует вводить лишние имена параграфов.

### *Комментарии*

В последние годы ощущается сильное разочарование в комментариях и их роли для читателя, особенно в программах на КОБОЛе. Во многих случаях комментарии просто повторяют программу. Например, в следующем фрагменте

```
***      COMPUTE TOTAL = ON-HAND + RECEIPTS.  
          RECEIPTS ARE ADDED TO AMOUNT ON HAND.
```

комментарии ничего не дают для понимания программы. На самом деле в некоторых случаях они вредят делу — ведь необходимо отвлекаться от чтения собственно программы. К тому же комментарии, противоречащие программе, вызывают путаницу и трату времени на определение того, где ошибка — в программе или в комментариях. Эти расхождения обычно объясняются тем, что программа менялась при отладке или модификации, а комментарии оставались прежними.

Иногда комментарии непонятны — когда их создавали, мало заботились об их качестве. С комментариями связана и другая

проблема — иногда возникает тенденция очень запутанно программировать в надежде на то, что комментарии все объяснят. Это может повысить вероятность появления совершенно непонятных программ.

Лучше всего сделать программу настолько ясной и однозначной, чтобы разъяснения не требовались. Структурное программирование *сокращает* потребность в комментариях. Комментарии должны содержать информацию, которую *нельзя* почерпнуть в самой программе. Такие комментарии обязательно нужно позже проверить, чтобы иметь уверенность в их соответствии программе, возможно изменившейся в процессе отладки. Комментарии в разделе идентификации должны объяснить читателю общее назначение модуля и способ его работы. Это может занять несколько страниц. Перед началом важных сегментов также должны быть помещены вводные комментарии. Если они по своей природе достаточно общего характера, они остаются справедливыми и для последующих модификаций программы.

### *Имена данных*

Поскольку имена данных в КОБОЛе могут быть длиной до 30 литер, надо выбирать имена, несущие как можно больше информации об элементах данных. Следует избегать произвольно выбранных сокращений и аббревиатур. Такие сокращения, как EOF для конца файла (end-of-file) или YTD для текущего года (year-to-date), общеприняты и однозначны. Однако сокращение ИНВ-НО неоднозначно (что это — *инвентарный номер* или *инвестиционный номер*?). Сокращений типа СВУР для «сверхурочный» следует избегать, поскольку даже его автор может легко забыть, что оно обозначает. Каждый, кто впервые сталкивается с такими сокращениями, никогда сразу не поймет их смысла. Один из разумных методов — создание набора допустимых сокращений и аббревиатур в рамках данной организации.

Другая идея — использовать для логически связанных объектов похожие имена данных. Для указания не только *связи*, но и *назначения* похожих имен данных могут использоваться префиксы и суффиксы. Например, если есть элемент вычислительных данных, скажем

### ЧИСЛО-СОТР

и надо его обобщить, то добавляется префикс. Например:

### ОБЩЕЕ-ЧИСЛО-СОТР

Если приведенные элементы данных встречаются также в формате, предназначенном для печати, то в их именах можно доба-

вить и суффикс:

## ЧИСЛО-СОТР-ПЕЧАТЬ ОБЩЕЕ-ЧИСЛО-СОТР-ПЕЧАТЬ

(Или, предполагая, что допускаются сокращения Ч для ЧИСЛО и П для суффикса ПЕЧАТЬ, обозначить последний объект через ОБЩЕЕ-Ч-СОТР-П.)

Тот же самый подход можно применять и к именам параграфов. Кроме того, на одной строке должно располагаться начало только одного параграфа, чтобы было легко выделять их и отделять от предшествующих параграфов. При создании имен параграфов лучше всего ввести префикс — номер параграфа. Это может быть либо последовательный номер, либо указание страницы, где расположен параграф. Польза такого метода проявляется при использовании ВЫПОЛНИТЬ, ссылающегося на параграфы из других страниц. Шаг номеров может быть равным 10 или 100, чтобы облегчить последующую вставку новых параграфов. Первые две цифры могут представлять страницу, на которой расположен параграф, а остальные — место на странице. Конечно, если параграфы перемещаются, то имена, так же как и номера ссылок, необходимо менять для сохранения смысла такой системы нумерации.

### *Отступы*

Отступы и выравнивания в тексте программы проясняют ее логику. В разделе процедур логически связанные предложения должны быть физически объединены одинаковой величиной отступа. Ниже приведены некоторые соглашения о формате программы. Важны не столько сами эти соглашения, сколько требование, чтобы этот набор соглашений был законом для всех программистов данной организации.

*Стандартный отступ* должен быть определен так, чтобы эти соглашения легко выражались через него. Вполне подходящий стандартный отступ — две-четыре позиции (колонки).

Чтобы различать типы заголовков, следует использовать разные колонки. Так, имена разделов могут начинаться с колонки 8, имена секций — с колонки 9, имена параграфов — с колонки 10.

**Раздел идентификации.** В этом разделе должно содержаться имя программы. Кроме этого, как уже говорилось, должна присутствовать секция «ЗАМЕЧАНИЯ (REMARKS), описывающая модуль в целом.

**Раздел оборудования.** Параграф УПРАВЛЕНИЕ-МАССИВАМИ (FILE-CONTROL), как и другие параграфы, должен рас-

полагаться с колонки 10. Каждое предложение **ДЛЯ** (**SELECT**) должно начинаться с новой строки и располагаться с колонки 14. Конструкция **ПРЕДНАЗНАЧИТЬ** (**ASSIGN**) должна быть на той же самой строке, что и **ДЛЯ**. Если используются другие конструкции, такие, как **ДОСТУП** (**ACCESS**) или **СОХРАНИТЬ** (**RESERVE**), то они должны начинаться с новой строки; таким образом, они будут располагаться с колонки 16. Аналогично параграф **УПРАВЛЕНИЕ-ВВОДОМ-ВЫВОДОМ** (**I-O-CONTROL**) должен начинаться с колонки 10. Используемые в нем конструкции должны начинаться с новой строки и располагаться с колонки 12.

**Раздел данных.** Описания массивов должны начинаться в колонке 10 с литер **ОМ** (**FD**). Затем следуют два пробела и имя массива (с колонки 14). Каждый пункт, относящийся к этому массиву, должен начинаться с новой строки и располагаться на один отступ правее, чем имя файла. Таким образом, если отступ равен четырем колонкам, такие пункты будут располагаться с колонки 18. Если же они не помещаются на одной строке, то для следующих строк необходимо сделать два дополнительных отступа.

Элементы уровней 01 и 77 должны располагаться с колонки 10 — там помещается номер уровня, затем два пробела и, наконец, имя данного (с колонки 14). Элементы более низкого уровня располагаются на один отступ правее. Если отступ равен четырем, то номер следующего уровня будет располагаться под именем предыдущего.

Программу легче читать, если выровнены все конструкции **ШАБЛОН** (**PICTURE**). Их следует сдвигать направо, скажем до колонки 40, с тем, чтобы **ШАБЛОН** не перекрывался именами данных низкого уровня. Тогда без труда все конструкции **ШАБЛОН** для любого уровня будут выровнены. Если же уровень так глубок или имя настолько длинное, что **ШАБЛОН** нельзя начать с колонки 40, то следует перейти к колонке 40 следующей строки. Другие конструкции, такие, как **ИСПОЛЬЗУЯ** или **ЗНАЧЕНИЕ**, должны располагаться с колонки 52. Если необходимо продолжение, то оно должно располагаться с колонки 44.

**Раздел процедур.** Каждое предложение следует начинать с новой строки. Чтобы программу легче было читать и изменять, предложения могут занимать отдельную строку или несколько строк, даже если они довольно короткие. Если предложение необходимо продолжить, то это продолжение начинается на два отступа правее. Предложение лучше прерывать, учитывая его логику, а не просто у правого края бланка. Например,

```
IF MASTER-FILE-EOF = 'Y'
  OR TRANSACTION-CUSTOMER-NUMBER = HIGH-VALUES
```

легче читать и изменять, чем

```
IF MASTER-FILE-EOF = 'Y' OR TRANSACTION-CUSTOMER-NUMBER =
  HIGH-VALUES
```

Предложения, соответствующие случаю «истина» в предложении ЕСЛИ, сдвигаются на один отступ вправо.

```
IF LINE-COUNT = 50
  PERFORM HEADING-ROUTINE
  MOVE 0 TO LINE-COUNT.
```

Для вложенных ЕСЛИ предложения также сдвигаются на один отступ вправо, ИНАЧЕ выравнивается по ЕСЛИ, а предложения после ИНАЧЕ также сдвигаются.

Предложения, использующие несколько имен данных или имен массивов, записываются так, что каждое имя располагается на отдельной строке, и все они выровнены. Например, ОТКРЫТЬ (OPEN), ПЕРЕМЕСТИТЬ (MOVE) и ЗАКРЫТЬ (CLOSE), использующие список имен, могут выглядеть так:

```
OPEN INPUT MASTER-FILE-CUSTOMER
      TRANSACTION-FILE
      OUTPUT UPDATED-REPORT.
```

```
MOVE ZERO TO TRANSACTIONS-PROCESSED
      INVALID-TRANSACTION
      TOTAL-TRANSACTION-RECORDS.
```

```
CLOSE MASTER-FILE-CUSTOMER
      TRANSACTION-FILE.
```

Аналогично, когда в выборе употребляется ПЕРЕЙТИ К... В ЗАВИСИМОСТИ ОТ, имена параграфов записываются на отдельных строках и выравниваются, как показано ниже:

```
GO TO NORMAL-PROCESSING-ROUTINE
      EXCEPTION-PROCESSING-ROUTINE
      ERROR-PROCESSING-ROUTINE
      DEPENDING ON EDIT-CODE.
```

Конструкции В КОНЦЕ или ПРИ ОШИБКЕ КЛЮЧА в предложении ЧИТАТЬ записываются с двумя отступами относительно ЧИТАТЬ. Аналогично, конструкции ПРОДВИГАЯ (ADVANCING), КОНЕЦ-СТРАНИЦЫ (END-OF-PAGE) или ПРИ ОШИБКЕ КЛЮЧА в предложении ПИСАТЬ также записываются с двумя отступами.



## *Пример*

На рис. 6.10, расположенном на шести следующих страницах, приведена программа на КОБОЛе, иллюстрирующая некоторые из рассмотренных принципов. Эта программа обновляет последовательный файл, содержащий записи о товарах. Обрабатываются следующие четыре типа запросов (операций):

- добавить новую запись в главный файл;
- исключить запись из главного файла;
- обновить существующую запись для расхода;
- обновить существующую запись для прихода.

Можно обновить и только что добавленную запись или исключить только что обновленную. Записи главного файла, над которыми операции не производились, копируются в новый файл.

Хотя пример взят из реальной практики, пришлось сделать некоторые упрощения с тем, чтобы приспособить его к тексту книги. Например, здесь нет редактирования или проверки упорядоченности данных. Обычно программа проверяет, что запросы упорядочены и обновления предшествуют добавлениям. Обычно требуются и другие проверки, которые удлинители бы многие из подпрограмм настолько, что сравнительно небольшие параграфы превратились бы в сегменты размером со страницу. Кроме того, некоторые подпрограммы на самом деле такие большие и сложные, что их необходимо оформлять как отдельные модули (например, проверка данных).

Функции и их взаимосвязи изображены на схеме иерархии (рис. 6.11). Выбранная структура облегчает отладку и модификацию. Например, эта программа вначале была написана так, что исключение записей не предусматривалось. Позднее, когда потребовалось добавить такую возможность, все изменения свелись к тому, что добавили параграф, обрабатывающий операции исключения, проверку кода этой операции в параграфе основной обработки и предложение ВЫПОЛНИТЬ (эту операцию). Рассмотрим также изменения, требующиеся для замены последовательного доступа к файлу произвольным. Для этого нужно лишь, кроме изменений в параграфе чтения главного файла, изменить параграф, сравнивающий ключи записей.

## **Выводы**

### *Сегментирование*

1. Модулями должны быть отдельно компилируемые подпрограммы, а передача данных должна осуществляться с помощью конструкции ИСПОЛЬЗУЯ.

IDENTIFICATION DIVISION.

PROGRAM-ID. UPDATE.

REMARKS:

THIS PROGRAM UPDATES AN INVENTORY FILE FROM A TRANSACTION  
FILE AND PRODUCES A NEW INVENTORY FILE. EXCEPTIONS ARE  
PRINTED FOR TRANSACTIONS WHICH ARE IN ERROR.

THE FOLLOWING TRANSACTION CODES ARE PERMISSABLE

- 1 ADDS A NEW ITEM TO THE FILE.
- 2 INDICATES RECEIPTS AND UPDATES THE QUANTITY ON HAND,  
TOTAL RECEIPTS FOR THAT ITEM, QUANTITY STILL ON ORDER,  
TOTAL COST OF THE ITEMS ON HAND AND AVERAGE COST PER ITEM
- 3 INDICATES ITEMS HAVE BEEN ISSUED, AND UPDATES THE  
QUANTITY ON HAND, TOTAL QUANTITY ISSUED, TOTAL COST OF  
ITEMS ON HAND, AND DATE OF LAST ISSUE.
- 4 DELETES AN ITEM FROM THE FILE.

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT EXCEPTIONS	ASSIGN TO UT-S-OUT.
SELECT NEWINV	ASSIGN TO DA-3330-S-NEWMAS.
SELECT OLDINV	ASSIGN TO DA-3330-S-OLDMAS.
SELECT TRANS	ASSIGN TO UT-S-TRANS.

DATA DIVISION.

FILE SECTION.

FD EXCEPTIONS

RECORD CONTAINS 80 CHARACTERS

RECORDING MODE IS F

LABEL RECORD IS OMITTED

DATA RECORD IS PRINT-LINE.

01 PRINT-LINE PIC X(80).

FD NEWINV

BLOCK CONTAINS 0 RECORDS

RECORD CONTAINS 63 CHARACTERS

RECORDING MODE IS F

LABEL RECORD IS STANDARD

DATA RECORD IS NEW-MASTER.

01 NEW-MASTER PIC X(63).

FD OLDINV

BLOCK CONTAINS 0 RECORDS

RECORD CONTAINS 63 CHARACTERS

RECORDING MODE IS F

LABEL RECORD IS STANDARD

DATA RECORD IS OLD-MASTER.

01 OLD-MASTER PIC X(63).

FD TRANS

BLOCK CONTAINS 0 RECORDS

RECORD CONTAINS 80 CHARACTERS

RECORDING MODE IS F

LABEL RECORD IS STANDARD

DATA RECORD IS TRANSACT.

01 TRANSACT PIC X(80).

# WORKING-STORAGE SECTION.

77	FIRST-TIME	PIC X.	
88	FIRST-TIME-YES		VALUE 'Y'.
88	FIRST-TIME-NO		VALUE 'N'.
77	NEW-RECORD-ADDED	PIC X.	
88	NEW-RECORD-ADDED-YES		VALUE 'Y'.
88	NEW-RECORD-ADDED-NO		VALUE 'N'.
77	OLD-MASTER-EOF	PIC X.	
88	OLD-MASTER-EOF-YES		VALUE 'Y'.
88	OLD-MASTER-EOF-NO		VALUE 'N'.
77	TRANS-EOF	PIC X.	
88	TRANS-EOF-YES		VALUE 'Y'.
88	TRANS-EOF-NO		VALUE 'N'.
77	TRANS-MASTER-MATCH	PIC X.	
88	TRANS-MASTER-MATCH-YES		VALUE 'Y'.
88	TRANS-MASTER-MATCH-NO		VALUE 'N'.
77	DUPLICATE-RECORD	PIC X(80)	VALUE
	' ATTEMPT TO ADD DUPLICATE RECORD.'		
77	INVALID-CODE	PIC X(80)	VALUE
	' INVALID CODE.'		
77	NO-MASTER	PIC X(80)	VALUE
	' NO MASTER RECORD FOR '		
77	TRANS-EOF-MESSAGE	PIC X(80)	VALUE
	' END OF FILE ON TRANSACTION FILE.'		
77	TRANSACTIONS-PROCESSED	PIC 999.	
01	FINAL-MESSAGE.		
05	TOTAL	PIC ZZZZ9.	
05	FILLER	PIC X(75)	VALUE
	' RECORDS PROCESSED.'		
01	NEW-MASTER-REC.		
02	ACTIVITY-CODE	PIC 9.	
02	NEW-MASTER-KEY	PIC X(6).	
02	DESCRIPTION	PIC X(20).	
02	QTY-ON-HAND	PIC 99999	COMP.
02	TOTAL-ISSUES	PIC 99999	COMP.
02	TOTAL-RECEIPTS	PIC 99999	COMP.
02	QTY-ON-ORDER	PIC 99999	COMP.
02	ORDER-POINT	PIC 999	COMP.
02	REORDER-QTY	PIC 99999	COMP.
02	TOTAL-COST	PIC 9(5)V99	COMP.
02	AVERAGE-UNIT-COST	PIC 9999V999	COMP.
02	DATE-OF-LAST-ISSUE	PIC X(6).	
01	OLD-MASTER-REC	PIC X(63).	

01	TRANSACTION.			
	05	TRANS-CODE	PIC 9.	
		88 ADD-TRANS		VALUE 1.
		88 RECEIPT-TRANS		VALUE 2.
		88 ISSUES-TRANS		VALUE 3.
		88 DELETE-TRANS		VALUE 4.
	05	TRANS-KEY	PIC X(6).	
	05	FILLER	PIC X(73).	
01	ADDITION REDEFINES TRANSACTION.			
	05	FILLER	PIC X(7).	
	05	DESCRIPTION	PIC X(20).	
	05	QTY-ON-HAND	PIC 99999.	
	05	TOTAL-ISSUES	PIC 99999.	
	05	TOTAL-RECEIPTS	PIC 99999.	
	05	QTY-ON-ORDER	PIC 99999.	
	05	ORDER-POINT	PIC 999.	
	05	REORDER-QTY	PIC 99999.	
	05	TOTAL-COST	PIC 99999V99.	
	05	AVERAGE-UNIT-COST	PIC 9999V999.	
	05	DATE-OF-LAST-ISSUE	PIC X(6).	
	05	FILLER	PIC X(5).	
01	RECEIPTS REDEFINES TRANSACTION.			
	05	FILLER	PIC X(7).	
	05	QTY-RECEIVED	PIC 99999.	
	05	TOTAL-COST	PIC 99999V99.	
	05	FILLER	PIC X(61).	
01	ISSUES REDEFINES TRANSACTION.			
	05	FILLER	PIC X(7).	
	05	QTY-ISSUED	PIC 99999.	
	05	CUSTOMER-NUMBER	PIC X(6).	
	05	DATE-ISSUED	PIC X(6).	
	05	FILLER	PIC X(56).	

```

PROCEDURE DIVISION.
10-INVENTORY-UPDATE.
***  INITIALIZATION
      OPEN INPUT OLDINV
           TRANS
      OUTPUT NEWINV
           EXCEPTIONS.
      MOVE 'N' TO OLD-MASTER-EOF
           TRANS-EOF
           NEW-RECORD-ADDED.
      MOVE 'Y' TO FIRST-TIME.
      MOVE LOW-VALUES TO NEW-MASTER-KEY.
      MOVE 0 TO TRANSACTIONS-PROCESSED.

***  MAIN LOOP.
      PERFORM 12-MAIN-PROCESS UNTIL TRANS-EOF=YES.
      PERFORM 32-COPY-NEXT-MASTER UNTIL OLD-MASTER-EOF=YES.

***  END OF JOB PROCESSING
      MOVE TRANSACTIONS-PROCESSED TO TOTAL.
      WRITE PRINT-LINE FROM FINAL-MESSAGE AFTER ADVANCING 3.
      CLOSE TRANS
           OLDINV
           NEWINV
           EXCEPTIONS.

      STOP RUN.

12-MAIN-PROCESS.
      PERFORM 20-READ-TRANS.
      IF TRANS-EOF=NO
           IF ADD-TRANS
               PERFORM 22-ADD-RECORD
           ELSE IF RECEIPT-TRANS OR ISSUES-TRANS
               PERFORM 24-UPDATE-RECORD
           ELSE IF DELETE-TRANS
               PERFORM 26-DELETE-RECORD
           ELSE
               MOVE INVALID-CODE TO PRINT-LINE
               PERFORM 44-MESSAGE-WRITE.

```

20-READ-TRANS.

READ TRANS INTO TRANSACTION  
AT END PERFORM 21-TRANS-EOF.  
IF TRANS-EOF-NO  
ADD 1 TO TRANSACTIONS-PROCESSED.

21-TRANS-EOF.

WRITE PRINT-LINE FROM TRANS-EOF-MESSAGE AFTER ADVANCING 2.  
MOVE 'Y' TO TRANS-EOF.  
IF NEW-RECORD-ADDED-YES  
WRITE NEW-MASTER FROM NEW-MASTER-REC  
MOVE 'N' TO NEW-RECORD-ADDED.

22-ADD-RECORD.

MOVE ' ' TO TRANS-MASTER-MATCH.  
\*\*\* ROUTINE EXECUTED UNTIL MATCH FOUND OR HIGHER KEY FOUND  
PERFORM 30-TEST-KEYS-FOR-MATCH UNTIL TRANS-MASTER-MATCH-YES  
OR TRANS-MASTER-MATCH-NO.  
IF TRANS-MASTER-MATCH-NO  
MOVE CORRESPONDING ADDITION TO NEW-MASTER-REC  
MOVE TRANS-KEY TO NEW-MASTER-KEY  
MOVE 'Y' TO NEW-RECORD-ADDED  
ELSE  
MOVE DUPLICATE-RECORD TO PRINT-LINE  
PERFORM 44-MESSAGE-WRITE.

24-UPDATE-RECORD.

MOVE ' ' TO TRANS-MASTER-MATCH.  
\*\*\* ROUTINE EXECUTED UNTIL MATCH FOUND OR HIGHER KEY FOUND  
PERFORM 30-TEST-KEYS-FOR-MATCH UNTIL TRANS-MASTER-MATCH-YES  
OR TRANS-MASTER-MATCH-NO.  
IF TRANS-MASTER-MATCH-NO  
MOVE NO-MASTER TO PRINT-LINE  
PERFORM 44-MESSAGE-WRITE  
ELSE  
IF RECEIPT-TRANS  
PERFORM 40-PROCESS-RECEIPTS  
ELSE IF ISSUES-TRANS  
PERFORM 42-PROCESS-ISSUES.

26-DELETE-RECORD.

MOVE ' ' TO TRANS-MASTER-MATCH.  
\*\*\* ROUTINE EXECUTED UNTIL MATCH FOUND OR HIGHER KEY FOUND  
PERFORM 30-TEST-KEYS-FOR-MATCH UNTIL TRANS-MASTER-MATCH-YES  
OR TRANS-MASTER-MATCH-NO.  
IF TRANS-MASTER-MATCH-NO  
MOVE NO-MASTER TO PRINT-LINE  
PERFORM 44-MESSAGE-WRITE  
ELSE  
IF NEW-RECORD-ADDED-YES  
MOVE 'N' TO NEW-RECORD-ADDED  
MOVE OLD-MASTER-REC TO NEW-MASTER-REC  
ELSE  
PERFORM 34-READ-MASTER.

### 30-TEST-KEYS-FOR-MATCH.

```
IF TRANS-KEY LESS NEW-MASTER-KEY
  MOVE 'N' TO TRANS-MASTER-MATCH
ELSE IF TRANS-KEY EQUAL NEW-MASTER-KEY
  MOVE 'Y' TO TRANS-MASTER-MATCH
ELSE IF TRANS-KEY GREATER NEW-MASTER-KEY
  IF OLD-MASTER-EOF-YES
    MOVE 'N' TO TRANS-MASTER-MATCH
  IF NEW-RECORD-ADDED-YES
    WRITE NEW-MASTER FROM NEW-MASTER-REC
    MOVE 'N' TO NEW-RECORD-ADDED
  ELSE
    NEXT SENTENCE
ELSE
  PERFORM 32-COPY-NEXT-MASTER.
```

### 32-COPY-NEXT-MASTER.

```
IF FIRST-TIME-YES
  PERFORM 34-READ-MASTER
  MOVE 'N' TO FIRST-TIME
ELSE
  WRITE NEW-MASTER FROM NEW-MASTER-REC
  IF NEW-RECORD-ADDED-YES
    MOVE 'N' TO NEW-RECORD-ADDED
    MOVE OLD-MASTER-REC TO NEW-MASTER-REC
  ELSE
    PERFORM 34-READ-MASTER.
```

### 34-READ-MASTER.

```
*** MASTER FILE IS READ INTO TWO WORKING STORAGE AREAS
*** THIS ALLOWS A POSSIBLE NEW RECORD TO BE BUILT UP IN
*** ONE AREA, WHILE THE OTHER KEEPS THE NEXT MASTER.
READ OLDINV INTO OLD-MASTER-REC
  AT END MOVE 'Y' TO OLD-MASTER-EOF
  MOVE 'N' TO TRANS-MASTER-MATCH.
MOVE OLD-MASTER-REC TO NEW-MASTER-REC.
```

### 40-PROCESS-RECEIPTS.

```
ADD QTY-RECEIVED TO QTY-ON-HAND IN NEW-MASTER-REC.
ADD QTY-RECEIVED TO TOTAL-RECEIPTS IN NEW-MASTER-REC.
SUBTRACT QTY-RECEIVED FROM QTY-ON-ORDER IN NEW-MASTER-REC.
ADD TOTAL-COST IN RECEIPTS TO TOTAL-COST IN NEW-MASTER-REC.
DIVIDE TOTAL-COST IN NEW-MASTER-REC
  BY QTY-ON-HAND IN NEW-MASTER-REC
  GIVING AVERAGE-UNIT-COST IN NEW-MASTER-REC.
```

### 42-PROCESS-ISSUES.

```
SUBTRACT QTY-ISSUED FROM QTY-ON-HAND IN NEW-MASTER-REC.
ADD QTY-ISSUED TO TOTAL-ISSUES IN NEW-MASTER-REC.
COMPUTE TOTAL-COST IN NEW-MASTER-REC =
  TOTAL-COST IN NEW-MASTER-REC -
  (QTY-ISSUED * AVERAGE-UNIT-COST IN NEW-MASTER-REC).
MOVE DATE-ISSUED TO DATE-OF-LAST-ISSUE IN NEW-MASTER-REC.
```

### 44-MESSAGE-WRITE.

```
WRITE PRINT-LINE AFTER ADVANCING 2.
WRITE PRINT-LINE FROM TRANSACTION AFTER ADVANCING 1.
```

Рис. 6.10. Пример программы на КОБОЛе.

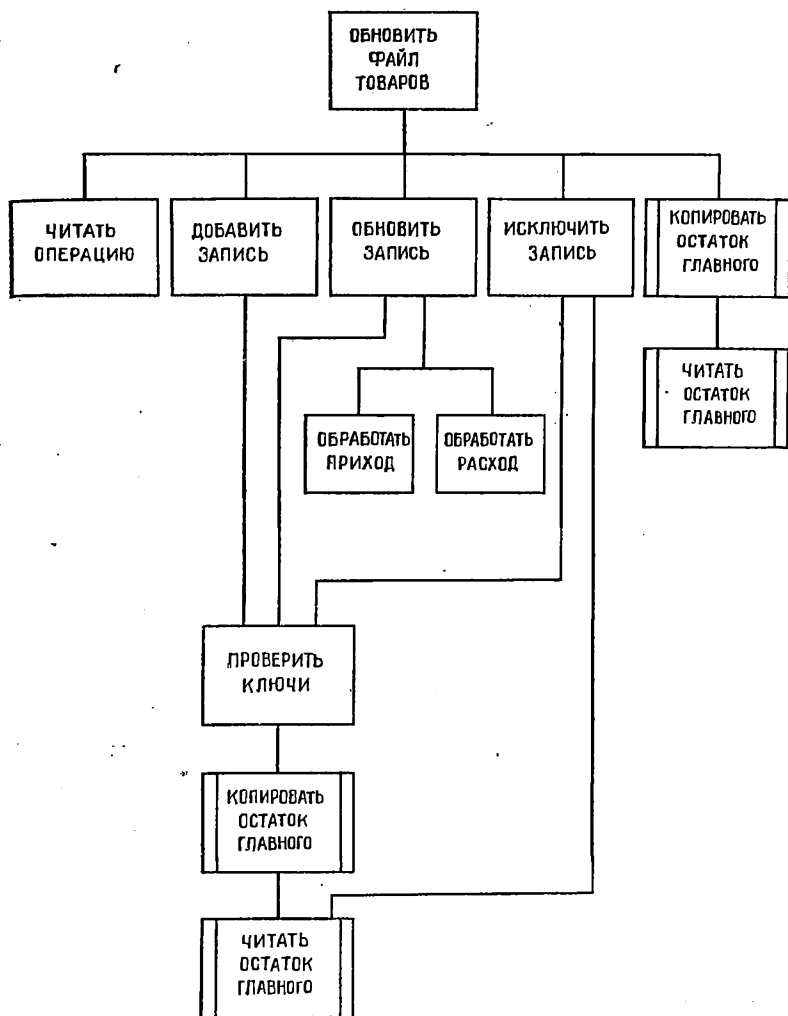


Рис. 6.11. Схема иерархии для примера обновления файла товаров.

2. Каждый сегмент должен занимать не более одной страницы.
3. Сегменты должны программироваться и тестироваться нисходящим образом, т. е. сначала программируется головной сегмент и тестируется с заглушками вместо остальных сегментов.
4. Все параграфы должны активизироваться явно — никаких «проходов сквозь» параграфы.
5. Секции нельзя активизировать с помощью ВЫПОЛ-



НИТЬ или ВЫПОЛНИТЬ... ПО (кроме двух упоминавшихся случаев).

6. Все подчиненные параграфы должны возвращать управление вызвавшему их параграфу.

### *Структурирование*

Программа должна исполняться «сверху вниз» (т. е. управление должно передаваться только вперед по тексту программы). Для реализации развилки используется ЕСЛИ, с ИНАЧЕ или без него. При использовании вложенных ЕСЛИ необходимо иметь в виду следующее: глубина вложенности должна быть ограниченной, отрицания, сложные и неявные объекты и операции следует по возможности исключать или хотя бы ограничивать, вся вложенная конструкция должна размещаться на одной странице, следует использовать ВЫПОЛНИТЬ вместо ПЕРЕЙТИ К, для индикаторов и их значений следует употреблять mnemonicические имена.

Для повторения используется ВЫПОЛНИТЬ... ДО, для выбора — вложенные ЕСЛИ или ПЕРЕЙТИ К ... В ЗАВИСИМОСТИ ОТ..., для поиска — предложение ИСКАТЬ.

### *О стиле программирования*

Исключите или сильно ограничьте употребление следующих предложений: ВХОД, ОСТАНОВИТЬ РАБОТУ, ИЗМЕНИТЬ, ПЕРЕЙТИ К ... В ЗАВИСИМОСТИ ОТ ..., ПЕРЕЙТИ К. Следует внимательно относиться к использованию индикаторов: используйте каждый индикатор только для одной цели, употребляйте mnemonicические имена и mnemonicические значения, используйте как можно меньше индикаторов.

Присваивать начальные значения переменным и индикаторам следует в разделе процедур, а указывать значения констант — в разделе данных.

### *Как облегчить чтение*

Начинать разделы и сегменты следует с новой страницы. Параграфы нужно разделять пустыми строками. Комментарии нужно помещать в начале модуля и, в случае необходимости, в начале сегментов и параграфов. Применяйте имена данных, говорящие сами за себя. Для проявления логики программы используйте отступы в разделе процедур. Не пользуйтесь литеральными константами.

## Контрольные вопросы и упражнения

1. Каковы доводы в пользу разделения модуля на сегменты размером со страницу? Доводы против такого разделения?
2. Какие проблемы возникают в параграфе, который иногда активизируется с помощью ВЫПОЛНИТЬ, а иногда как часть секции?
3. В каких случаях допустим ВЫПОЛНИТЬ... ПО?
4. Следующая программа на псевдокоде выдает справки на основе файла сотрудников. Справки касаются даты поступления на работу и/или текущей должности. Переведите этот псевдокод в программу на КОБОЛЕ.

Читать справку

ЕСЛИ справка правильная

ЕСЛИ запрос даты

выдать дату поступления

ВСЕ-ЕСЛИ

ЕСЛИ запрос должности

выдать должность

ВСЕ-ЕСЛИ

ВСЕ-ЕСЛИ

5. Какой, с вашей точки зрения, набор ограничений сделает вложенные ЕСЛИ легче воспринимаемым?
6. Что вы используете для реализации выбора — ПЕРЕЙТИ К... В ЗАВИСИМОСТИ ОТ... или ЕСЛИ? Почему?
7. Измените программу на рис. 6.10 для произвольного порядка обработки главного файла.
8. Измените программу на рис. 6.10 введением проверки упорядоченности файла запросов.
9. Измените программу на рис. 6.10 введением двух входных файлов запросов. В одном содержатся добавления и исключения, в другом — обновления (приход и расход). Оба файла упорядочены по возрастанию.
10. Предположим, что в вашей организации нельзя работать с модулями, содержащими больше 50 операторов КОБОЛа. Перепрограммируйте задачу на рис. 6.10 в соответствии с этим требованием. Начните с подготовки пересмотренной схемы иерархии.

## Структурное программирование на ФОРТРАНе

*«Для структурного программирования можно создать и лучшие языки, но инвестиции промышленности в ФОРТРАН способны еще некоторое время продлевать его жизнь».*

Тед Тенни <sup>1)</sup>

Говоря о ФОРТРАНе и структурном программировании, легче всего перечислить множество факторов, по которым ФОРТРАН не очень хорошо согласуется с этой новой технологией. Однако это значит начать с отрицательных сторон языка, все еще широко используемого для многих научных и даже экономических приложений. Давайте лучше перечислим ряд проблем, возникающих при попытке структурно программировать на ФОРТРАНе, и укажем их возможные решения.

**Проблема.** При записи структурной программы на ФОРТРАНе метки и операторы перехода должны использоваться очень часто. В некоторых случаях это может затруднить чтение программы.

**Решение.** Сначала написать программу на псевдокоде, используя подходящие ключевые слова (ЕСЛИ-ТО-ИНАЧЕ, ЦИКЛ-ПОКА, ЦИКЛ-ДО и т. д.). Затем механически преобразовать ее в последовательность предложений ФОРТРАНа, оставляя текст на псевдокоде в качестве предваряющего комментария. При таком способе читающий программу может до изучения текста на ФОРТРАНе ознакомиться с текстом на псевдокоде.

**Проблема.** Сегментация модулей на ФОРТРАНе настолько хитрая и расточительная по времени, что становится неясным, как достичь целей структурного программирования.

**Решение.** Определять программу в терминах маленьких модулей. Если при кодировке модуля его *операторы* занимают

---

<sup>1)</sup> Ted Tenny, "Structured Programming in FORTRAN", Datamation, July 1974, p. 110. Reprinted with the permission of DATAMATION © Copyright 1974 by Technical Publishing Company, Greenwich, Connecticut 06830.

более страницы, разделить его на два или более модулей (а не на два сегмента, как это было бы сделано в КОБОЛе или ПЛ/1). Конечно, каждый модуль транслируется отдельно — как головной модуль, являющийся главной программой, так и подчиненные модули — подпрограммы или функций. Преимущество раздельной трансляции в том, что все части программы (маленькие модули) хорошо изолированы друг от друга.

**Проблема.** Внутри условного логического оператора можно указывать лишь единственный оператор (выполняемый при истинности условия). Часто, однако, в этом случае требуется выполнить группу операторов. Ограничен и оператор цикла. Его заголовок позволяет проверять лишь значение управляющей переменной.

**Решение.** Использовать препроцессор с выходом на ФОРТРАН. Это позволяет расширять синтаксис ФОРТРАНа, добавляя конструкции ЕСЛИ-ТО-ИНАЧЕ, ЦИКЛ-ПОКА и др., начинающиеся именно этими ключевыми словами. Препроцессор заменяет эти предложения предложениями ФОРТРАНа, реализующими указанные структуры. Эти заменяющие предложения затем транслируются совместно с остальными предложениями, написанными непосредственно на ФОРТРАНе, в результате чего и появляется окончательная выходная программа. (Когда используется препроцессор, нет необходимости предварять модуль комментариями на псевдокоде.) Препроцессор — это промежуточное средство, наиболее легкий способ дать возможность писать структурные программы на языке ФОРТРАН. Существуют и другие методы, такие, как расширение синтаксиса языка или использование аппарата макросов. В этой области ожидается большая активность, обусловленная инвестициями промышленности в ФОРТРАН.

В данной главе речь пойдет об организации, структуре и формате программ на ФОРТРАНе. Будут также рассмотрены общие принципы использования препроцессоров. Понимание этих принципов облегчит изучение конкретного доступного вам препроцессора (их производят как разработчики ЭВМ, так и создатели программного обеспечения).

### **Организация программ на языке ФОРТРАН**

Разработка структурных программ производится за несколько шагов. Это, кратко говоря, 1) нисходящая разработка схемы иерархии; 2) пошаговая детализация модуля с использованием псевдокода и 3) кодирование на некотором языке.

Перед началом кодирования модуля необходимо подготовить тесты и операторы языка управления заданиями. Кроме того,

необходимо подготовить все требуемые заглушки для этого модуля, чтобы было в наличии все, что нужно для тестирования. Головной модуль кодируется первым. Он является «конспектом» всей программы и обычно содержит только операторы, необходимые для активизации или вызова подчиненных модулей или подпрограмм, например:

```
CALL INPUT (ROP,PIPTK,T,HVAL,AVAL)
IF ( CODE.EQ.1 ) CALL PIPE(ROP,PIPTK,HVAL,AVAL,DU)
CALL COSTS (FIXED,STEAM,TOTAL)
RETURN
END
```

При планировании той последовательности, в которой модули будут кодироваться и проверяться, следует применять *комбинированный подход*, описанный в гл. 2 и 3.

Посмотрев на программу на ФОРТРАНе, нельзя сразу сказать, хорошо ли она структурирована. Проявлению структуры способствуют комментарии, пробелы и отступы. Но даже и в этом случае может оказаться, что некоторые структурные программы не слишком легко читать. (Однако такие преимущества структурных программ, как легкость сопровождения и малое число ошибок, компенсируют этот недостаток.)

На рис. 7.1 показана организация текста программы на языке ФОРТРАН. В самом начале одну-две страницы занимают ком-

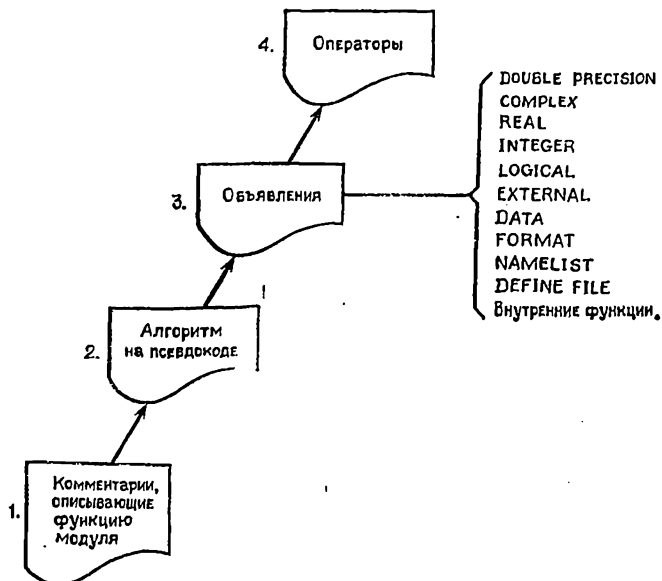


Рис. 7.1. Шаблон для исходного текста программы на ФОРТРАНе.

ментарии, описывающие модуль и его назначение (функцию).

Затем, если только не используется препроцессор, должен следовать текст на псевдокоде — для программирующего на ФОРТРАНе это особенно полезный инструмент. Здесь впервые полностью описывается логическая структура модуля. Затем структуры псевдокода нужно вручную перевести на ФОРТРАН. В некотором смысле вы выполняете при этом роль препроцессора или компилятора. Этот шаг нужно рассматривать как чисто механический — здесь нельзя *улучшать* программу. Если позднее возникает потребность ее изменить, то начинать следует с текста на псевдокоде. Сначала он изменяется в соответствии с дополнительными требованиями к программе, а затем соответственно модифицируется текст на языке ФОРТРАНа.

Непосредственно вслед за текстом на псевдокоде следуют описания. ФОРТРАН предписывает определенную упорядоченность описаний. Для единообразия и удобства восприятия описания следует располагать в следующем порядке:

1. Программный модуль, не являющийся головным, следует начинать с заголовка процедуры (SUBROUTINE или FUNCTION) или заголовка спецификации блока данных (BLOCK DATA).

2. Затем могут следовать объявления общих объектов (COMMON), но их использовать не рекомендуется. Использование общих объектов исторически возникло как средство экономии памяти при работе на ЭВМ второго поколения. Те, кто использовал общие объекты в своих программах, помнят, какие трудности возникали при отладке таких программ (не говоря уже о так и не обнаруженных ошибках). Эта конструкция препятствует изоляции подпрограмм, и поэтому ею пользоваться не рекомендуется. (Единственным исключением из этого правила может служить разработка специальных систем управления процессами или обработки событий, где возникает необходимость связи компоненты системы с ее ядром с помощью объявления общих объектов.)

3. Затем могут следовать объявления типа

DOUBLE PRECISION  
COMPLEX  
REAL  
INTEGER  
LOGICAL

(В некоторых компиляторах ФОРТРАНа допустимы не все эти объявления.) Объявления типа должны содержать все используемые переменные и должны включать информацию о размерности массивов. При таком подходе отпадает необходимость

в объявлениях массивов (DIMENSION) и неявных типов данных (IMPLICIT). Дополнительное преимущество этого — *раскрепощение* первой литеры в имени переменной, что позволяет выбирать mnemonicические имена данных произвольным образом.

При использовании в структурной программе наглядных и mnemonicических имен данных потребность в комментариях минимальна. Однако, поскольку имя переменной не может содержать более пяти (или шести) литер, часто трудно придумать для данных имена, очевидным образом связанные с их назначением. Поэтому могут потребоваться дополнительные комментарии, проявляющие сущность таких имен. Лучше всего располагать их в программе рядом с объявлениями типа.

4. Объявления эквивалентности (EQUIVALENCE) было бы логичным располагать за объявлениями типов. Однако по соображениям, приведенным в п. 2, их лучше вообще не использовать.

5. Следующими располагаются объявления внешних имен (EXTERNAL).

6. Затем следуют объявления начальных данных (DATA).

7. Объявления, связанные с вводом-выводом, должны быть собраны вместе:

объявления формата (FORMAT)

предложение NAMELIST

предложение DEFINE FILE

8. Наконец, раздел объявлений завершают объявления внутренних функций.

Возвращаясь к рис. 7.1, мы видим, что четвертая часть исходной программы состоит из операторов. В других языках высокого уровня сегментация модуля возможна, в ФОРТРАНе нет. Таким образом, лучше всего писать маленькие модули (не превышающие одной страницы текста), а затем объединять их при совместной компиляции. Предварительный анализ применения этого подхода к программам на ФОРТРАНе показал, что накладные расходы по вызову отдельно транслируемых модулей не сказываются существенно ни на объеме памяти, ни на времени работы программы. Возможно, структурные программы оказываются в этом смысле более эффективными потому, что уменьшается возможность неудачного разбиения на модули.

## Структурирование программ на ФОРТРАНе

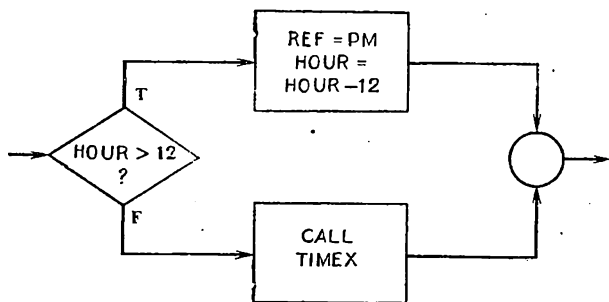
Структурирование — это представление заданной функции или подфункции с помощью трех основных и некоторых необязательных структур.

### Следование

Следование состоит из двух обрабатывающих блоков, причем управление передается от предыдущего блока к последующему. Блок может представлять либо отдельный оператор, например чтения (READ) или записи (WRITE), либо группу операторов, реализующих заданную подфункцию.

### Развилка

Конструкция ЕСЛИ-ТО-ИНАЧЕ задает проверку для определения одного из двух действий — преемников. Пусть нужно проверить значение переменной с именем HOUR.



Если для реализации приведенного примера используется условный логический оператор, то альтернатива *истина* программируется непосредственно после него. Тогда необходимо альтернативу *ложь* реализовать так:

```
IF (HOUR.LE.12) GOTO 1010
  REF = PM
  HOUR = HOUR - 12
  GOTO 1020
1010 CONTINUE
C ELSE
  CALL TIMEX (HOUR,REF)
1020 CONTINUE
C ENDIF
C
END
```

Операторы этих двух альтернатив пишутся с отступом. Операторы продолжения (CONTINUE) обязательно завершают каждую альтернативу и пишутся с той же колонки, что и условный оператор. Поскольку в условном логическом операторе фигурирует оператор перехода, позволяющий обойти совокупность операторов, если проверяемое условие истинно, необходимо



программировать *отрицание* исходного условия. Например, чтобы проверить  $A = B$  и вызвать соответствующие подчиненные модули, используется следующий фрагмент:

```

                IF (A.NE.B) GOTO 1030
                CALL EQUAL
                GOTO 1040
1030 CONTINUE
C      ELSE
        CALL NOTEQ
1040 CONTINUE
C      ENDF

```

Можно условие не менять, а автоматически предварять его конструкцией. NOT. . Например:

```

                IF (.NOT. (A.EQ.B)) GOTO 1030
                CALL EQUAL
                GOTO 1040
1030 CONTINUE
C      ELSE
        CALL NOTEQ
1040 CONTINUE
C      ENDF

```

Тогда при чтении .NOT. нужно игнорировать, помня, конечно, что альтернатива «истина» *непосредственно следует* за условным оператором. Выберите тот метод, который вам больше нравится, но затем всегда следуйте именно ему.

Если альтернатива ИНАЧЕ (ELSE) не нужна, то задается только первая альтернатива. Если это один оператор, то и вся развилка выражается одним оператором. Например:

```

IF (HOUR.LE.12) CALL TIMEX (HOUR,REF)

```

Если для изображения альтернативы требуется два или более операторов, то используется условный оператор с оператором перехода (GOTO). Например:

```

                IF (HOUR.GT.12) GOTO 1060
                REF = PM
                HOUR = HOUR - 12
1060 CONTINUE
C      ENDF

```

Оператор продолжения отмечает конец альтернативы ТО. В начале его употребление может показаться обременительным и ненужным. Хотя этот оператор был введен в языке для завершения циклов, его можно использовать в программе где угодно, причем это не влияет на порождаемую выходную программу.

Его использование в структуре ЕСЛИ-ТО-ИНАЧЕ вполне уместно. Оно учитывает возможность будущих модификаций программы. Употребление метки в операторе продолжения делает эту метку частью структуры. Если структура позднее будет меняться, то будет необходимо изменить только те операторы, которые необходимо менять содержательно. Кроме того, препроцессор также обычно порождает программу для ЕСЛИ-ТО-ИНАЧЕ, используя операторы продолжения для отделения альтернатив ТО и ИНАЧЕ и для завершения структуры.

Условный оператор может быть вложен в другой условный оператор<sup>1)</sup>. Однако удобство чтения таких вложенных операторов в ФОРТРАНе резко падает. Если вложенные условные операторы все же используются, глубина вложенности не должна превышать трех.

Некоторые эксперты по ФОРТРАНу считают условный арифметический оператор «программистским динозавром», оставшимся со времен становления вычислительной техники. Если компилятор позволяет использовать условный логический оператор, то употреблять условный арифметический оператор не следует. Приведенный выше пример можно запрограммировать и с помощью условного арифметического оператора:

```
      IF (HOUR - 12) 1020,1020,1010
1010      REF = PM
          HOUR = HOUR - 12
          GOTO 1030
1020      CONTINUE
C        ELSE
          CALL TIMEX (HOUR,REF)
1030      CONTINUE
C        ENDIF
```

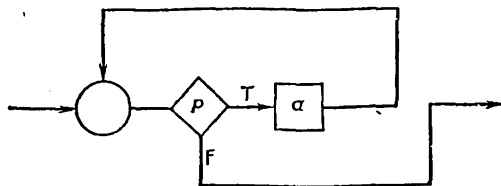
Основной довод против такого оператора в том, что трудно определить, какая метка оператора соответствует случаю «истина», а какая — случаю «ложь». Кроме того, использование условного арифметического оператора для логических проверок в структурных программах неестественно — это затрудняет чтение программ.

### *Повторение*

Конструкция ЦИКЛ-ПОКА позволяет организовать цикл. Некоторое действие повторяется до тех пор, пока остается истинным условие, заданное предикатом *p*. Этот предикат проверяется перед каждым выполнением этого действия, включая и первое. Ниже приведена блок-схема конструкции ЦИКЛ-ПОКА.

---

<sup>1)</sup> В стандарте ФОРТРАНа это запрещено.— *Прим. перев.*



Например, чтобы изобразить

ЦИКЛ-ПОКА  $I \leq 10$

где последующий фрагмент должен выполняться десять раз, можно применять оператор цикла ФОРТРАНА. Другой способ состоит в том, чтобы употреблять *условный логический оператор*.  
Общий вид конструкции таков:

```

1033 IF (.NOT.(p)) GOTO 1036
      Программа для действия a
      GOTO 1033
1036 CONTINUE
  
```

Операторы внутри конструкции ЦИКЛ-ПОКА пишутся с отступом. Оператор продолжения завершает эту конструкцию и пишется на одном уровне с условным оператором. Поскольку в условном логическом операторе фигурирует оператор перехода, обходящий совокупность последующих операторов, если проверяемое условие истинно, необходимо программировать *отрицание* исходного условия. Например:

```

      I = 1
1040 IF (.NOT.(I.LE.LIMIT)) GOTO 1050
      TOTAL = TOTAL + TEMPS (I)
      I = I + 1
      GOTO 1040
1050 CONTINUE
  
```

Для большей наглядности можно использовать обратную операцию отношения и опустить .NOT. . Так, приведенный выше условный оператор мог бы иметь вид:

```

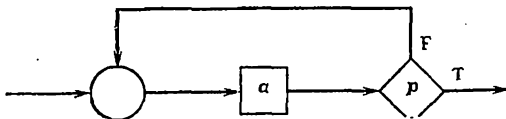
      IF (I.GT.LIMIT) GOTO 1050
  
```

Структура ЦИКЛ-ДО позволяет организовывать такие же циклы, что и ЦИКЛ-ПОКА. Она отличается от последней двумя особенностями:

1. Проверка, управляющая циклом, производится *после* выполнения тела цикла. Таким образом, тело цикла будет выполняться по меньшей мере один раз.

2. ЦИКЛ-ДО оканчивается, когда проверяемое условие становится истинным (а ЦИКЛ-ПОКА — когда ложным).

Структура ЦИКЛ-ДО такова



К моменту выхода этой книги стандарт ФОРТРАНа не определяет, похож ли оператор цикла ФОРТРАНа на ЦИКЛ-ПОКА или ЦИКЛ-ДО. Например, в данном конкретном случае (т. е. когда LIMIT меньше начального значения переменной цикла)

```

LIMIT = 0
DO 10 I = 1, LIMIT
  A(I) = 20.0
10 CONTINUE
  
```

стандарт 1966 года не определяет результат выполнения цикла как присваивание A(I) значения 20.0. Стандарт лишь устанавливает следующее (разд. 7.1.2.8): «Значение, представленное начальным параметром, должно быть меньше или равно значению, представленному конечным параметром». Другими словами, результат предыдущего примера неопределен. Большинство (но не все) реализаций ФОРТРАНа выполняют тело оператора цикла *перед* проверкой условия — таким образом, логика соответствует ЦИКЛ-ДО. Общий вид цикла таков:

DO 1052 (сведения о переменной цикла)

Программа для действия a

1052 CONTINUE

Программа для действия a должна входить в тело цикла целиком или вызывать подчиненные фрагменты с возвратом. Так, в примере

```

DO 55 I = 1, N
  READ (INP, 902) RADIUS, HEIGHT
  VOLUME = 3.14159 * RADIUS **2 * HEIGHT / 231.
  CALL PRINT (CODE, VOLUME)
55 CONTINUE
  
```

подфункция печати представлена вызовом подчиненного модуля. В более общем случае, когда не годятся счетчики, структура

ЦИКЛ-ДО реализуется на ФОРТРАНе с помощью условного логического оператора:

1057 CONTINUE

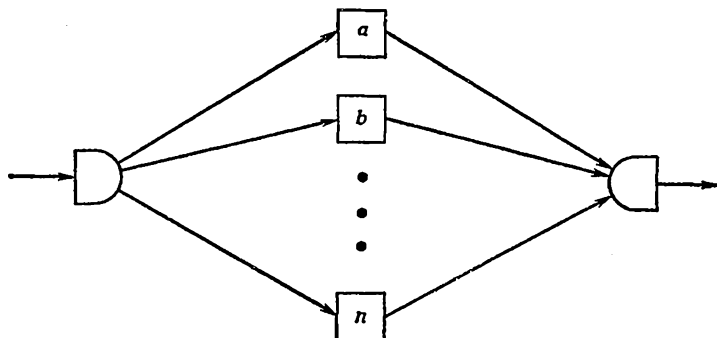
Программа для действия *a*  
IF (.NOT.(p)) GOTO 1057

Структура ЦИКЛ-ДО начинается с оператора продолжения. Например, ниже показан цикл, повторяющийся до тех пор, пока не исчерпается вводная последовательность или место для ее размещения в памяти.

```
LOGICAL DONE
DONE = .FALSE.
READ (INP,901) I,N
1060 CONTINUE
READ (INP,902) ARRAY(I)
I = I + 1
IF (I.GT.ARSIZE .OR.
  'A I.GT.N) DONE = .TRUE.
IF (.NOT.DONE) GOTO 1060
```

### Выбор

Структура ВЫБОР используется тогда, когда с помощью одной проверки нужно выбрать только одну из нескольких альтернатив. Она может быть изображена как



Заметим, что все пути через разные блоки сходятся в одной точке.

Эта структура лучше всего выражается *вычисляемым оператором перехода*. Для иллюстрации предположим, что требуется выбрать одно из четырех действий, основываясь на значении переменной ICODE. Если ICODE=1, то выбирается оператор с меткой 2001, если ICODE=2, то с меткой 2002, и т. д.

На рис. 7.2 показано, как в общем случае можно программировать выбор. Если ICODE больше 4 или меньше 1, то выпол-

няется диагностическая программа, начинающаяся с метки 2005. Структура **ВЫБОР** начинается с *вычисляемого оператора перехода* и заканчивается оператором продолжения с меткой 2010 (который в свою очередь служит началом следующей структуры в программе). Все фрагменты из структуры выбора нужно располагать непосредственно за вычисляемым оператором перехода.

```

      IF (ICODE.GT.4 .OR.
A      ICODE.LT.1) ICODE=5
C
C      GO TO (2001,2002,2003,2004,2005),ICODE      CASE START
2001  CONTINUE
C
C      .
C      .      SOURCE STATEMENTS GO HERE
C      .      FOR CODE = 1
C      .
C      GO TO 2010
C
2002  CONTINUE
C
C      .
C      .      SOURCE STATEMENTS GO HERE
C      .      FOR CODE = 2
C      .
C      GO TO 2010
C
2003  CONTINUE
C
C      .
C      .      SOURCE STATEMENTS GO HERE
C      .      FOR CODE = 3
C      .
C      GO TO 2010
C
2004  CONTINUE
C
C      .
C      .      SOURCE STATEMENTS GO HERE
C      .      FOR CODE = 4
C      .
C      GO TO 2010
C
2005  CONTINUE
C
C      .
C      .      SOURCE STATEMENTS GO HERE
C      .      FOR ERROR
C      .
2010  CONTINUE
C
                                           CASE END

```

Рис. 7.2. Моделирование структуры **ВЫБОР**.

### О стиле программирования

Структурный подход заставляет переоценить ряд *типичных* приемов программирования, которыми пользовались, как говорится, с *незапамятных времен*. Например, писались такие программы на ФОРТРАНе, где количество вводимых значений задавалось явно:

```
READ (ICARD, 0100) NCOUNT, (MATRIX(I), I=1,NCOUNT)
```

```

      READ (INPUT,904) I,J,A,B,C
      IF (I.EQ.9999) GOTO 1090
C PROCESS RECORD
C
C
C
      GO TO 1100
1090 CONTINUE
C HANDLE END-OF-FILE
C
C
C
      1100 CONTINUE
C      ENDIF

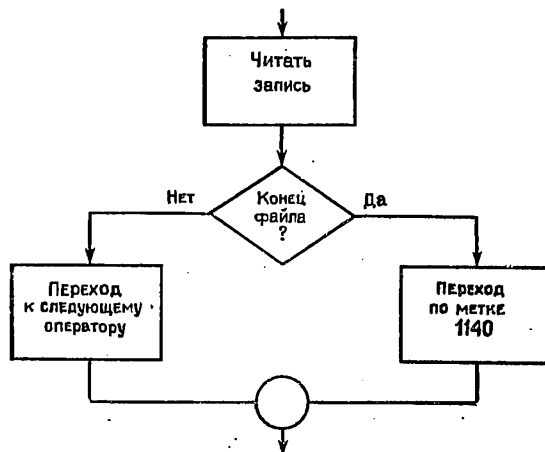
```

```

LOGICAL MORE
MORE = .TRUE.
      C
      C
      C
1020  IF (.NOT.MORE) GO TO 1130
СЛЕДОВАНИЕ — READ (INPUT,0090) I,J,A,B,C
      IF (I.EQ.9999) GO TO 1090
      PROCESS RECCRD
      C
      C
      C
      C
      GO TO 1100
1090  CONTINUE
      C
      ELSE
      HANDLE END-OF-FILE
      MORE = .FALSE.
1100  CONTINUE
      C
      ENDF
      GO TO 1020
1130  CONTINUE

```

Параметр END указывает номер оператора, которому будет передано управление при исчерпании файла. Такой оператор ввода можно изобразить следующей схемой:



На ФОРТРАНе это можно записать так:

```

      READ (INPUT,0010,END=1140) A,B,C
C PROCESS RECORD
C      .
C      .
C      .
C      GO TO 1150
1140 CONTINUE
C
C HANDLE END-OF-FILE
C      .
C      .
C      .
1150 CONTINUE
  
```

Для ввода совокупности записей эту структуру надо вложить в структуру ЦИКЛ-ПОКА. Например:

```

      LOGICAL MORE
      DATA MORE/.TRUE./
C      .
C      .
C      .
1130 IF ( .NOT. MORE ) GO TO 1160
      READ (INPUT,0010,END=1140) A,B,C
C PROCESS RECORD
C      .
C      .
C      .
C      GO TO 1150
1140 CONTINUE
C      PROCESS END-OF-FILE
C      MORE = .FALSE.
C      .
C      .
C      .
1150 CONTINUE
      GO TO 1130
C
C      ENDDO
1160 CONTINUE
  
```

Цикл-пока

Следование/если-то-иначе



Перечисленные ниже операторы ФОРТРАНа лучше вовсе не использовать, либо использовать с осторожностью — их применение может противоречить принципам структурного программирования.

1. *Объявление входа (ENTRY)*. Это предложение определяет дополнительные входы в подпрограмме. Его употребление противоречит концепции *простой программы* (один вход и один выход).

2. *Оператор останова (STOP)*. Поскольку этот оператор завершает выполнение всей программы, он может использоваться только в головном модуле. В противном случае нарушается правило возврата в модуль верхнего уровня. Исключение из этого правила возможно только тогда, когда подчиненный модуль обнаруживает неисправимую ошибку и вынужден завершить выполнение всей программы.

3. *Оператор возврата (RETURN)*. В соответствии с концепцией простой программы в подпрограмме может встречаться только один оператор возврата.

4. *Оператор перехода (GOTO)*. Бесконтрольное использование оператора перехода должно быть запрещено. Оператор перехода может использоваться лишь *внутри* управляющих структур (для их моделирования). Все остальные варианты противоречат правилам структурного программирования.

5. *Условный арифметический оператор (IF)*. Как уже говорилось, этот оператор лучше по возможности не использовать.

### Как облегчить чтение программы

Программу удобнее читать, если ее формат удовлетворяет определенным правилам. Эти правила проявляют логику программы за счет отступов при печати текста программы. Различные структуры должны быть расположены так, чтобы логические связи программы были отражены физическим расположением этих структур в тексте.

### Формат страниц

К сожалению, многие компиляторы ФОРТРАНа не позволяют (с помощью ключевых слов или специальных управляющих литер) управлять вставкой пробелов в листинг. Если используемый компилятор не способен автоматически вставлять пустые строки или пропускать страницы при печати листинга, это необходимо делать «вручную». Для того чтобы очередной раздел программы начинался с новой страницы, необходимо вставить пустые строки — комментарии. Конечно, лучше всего это

делать после полного тестирования модуля. При отладке можно каждый раздел (например, объявление типа, объявления формата и др.) начинать с карты-комментария, содержащей сплошные тире, с тем, чтобы отделить его от других частей модуля.

### *Комментарии*

Хорошо написанная структурная программа в значительной степени сама по себе способна служить документом. Комментарии, лишь *повторяющие* то, что уже ясно из программы, мало что добавляют к ее пониманию. Если программа меняется, а комментарии остаются прежними, возникает дополнительная опасность путаницы. В теле программы рекомендуется использовать комментарии умеренно. Если наличие комментария может вызвать путаницу, то его лучше убрать.

### *Номера предложений*

Программу будет легче читать, если при нумерации предложений придерживаться определенной системы. Вообще говоря, номера предложений должны возрастать при чтении программы сверху вниз. Каждый очередной номер предложения должен выбираться с шагом 5 или 10, с тем чтобы при добавлении новых предложений старые номера оставались прежними. Номера объявлений формата должны быть из диапазона 901—999.

### *Отступы*

*Отступ* — это число позиций, на которое передвигается пишущая машинка или перфоратор, если нажать клавишу табулирования или пропуска. В этом тексте один отступ — это две позиции, однако в некоторых организациях стандартный отступ больше.

*Начальной позицией* называется самая левая колонка, с которой могут начинаться предложения (исключая номер предложения). Начальная позиция может перемещаться вправо и влево в зависимости от того, какое предложение записывается. Когда встречается условный оператор или оператор цикла, колонка, в которой он расположен, становится начальной позицией. Конец условного оператора и тела цикла (например, оператор продолжения) вызывает возврат начальной позиции в прежнее состояние или сдвиг влево на один отступ. Сам оператор продолжения должен помещаться точно под соответствующим условным оператором или оператором цикла.

„ *Второй уровень* располагается на один отступ (т. е. на два пробела) правее начальной позиции. Таким образом, предло-

жение, находящееся внутри тела цикла, располагается на втором уровне, равно как и предложение, следующее за условным оператором.

### *Строки-продолжения*

Компиляторы ФОРТРАНа позволяют записывать предложения на нескольких (от 6 до 19) строках-продолжениях. Колонка с признаком продолжения может использоваться для нумерации этих строк. Это можно сделать, помещая в колонку 6 литеры в алфавитном порядке, например от A до S.

Если допустимое число строк-продолжений меньше 10, то можно использовать и цифры от 1 до 9. Вообще говоря, вторая и последующие строки предложения должны предваряться двумя-тремя отступами (четырьмя-шестью пробелами в нашем случае). Нужно стараться, чтобы ни в одной строке-продолжении информация не располагалась левее, чем в первой строке. Если же это невозможно (как в случае текстового поля в объявлении формата, превышающем число допустимых колонок текущей строки), следует попытаться облегчить чтение за счет выделения такого предложения пустыми строками-комментариями.

### **Использование препроцессора с выходом на ФОРТРАН**

Назначение этого раздела — не научить, как применять конкретный препроцессор, а скорее объяснить в общих чертах, как препроцессоры работают и как их используют. Однако наши примеры будут относиться к разработке программ с помощью конкретного препроцессора ФОРТРАНа для структурных программ<sup>1)</sup>.

Обычные предложения ФОРТРАНа препроцессоры передают на выход в том же порядке и без изменений. Повторение и развилка представлены на входе препроцессора такими предложениями, как

ЕСЛИ-ТО . . . . .	ВСЕ-ЕСЛИ
ЕСЛИ-ТО-ИНАЧЕ . . . . .	ВСЕ-ЕСЛИ
ЦИКЛ-ПОКА . . . . .	ВСЕ-ЦИКЛ
ЦИКЛ-ДО . . . . .	ВСЕ-ЦИКЛ

С помощью этих предложений легче писать структурные программы, чем «блюдо спагетти». Их можно перемешивать с любыми предложениями ФОРТРАНа, однако рекомендуется их употреблять исключительно для изображения управляю-

---

<sup>1)</sup> IBM Program Number 5798-CDW, Reference Document NO. SB21-1614, Data Processing Division, White Plains, N. Y. 10604.

щих структур (т. е. вместо условного логического оператора, оператора перехода и т. д.).

На рис. 7.3 показано взаимодействие препроцессора и компилятора, а также характер выводимой информации по каждой программе. Основная идея в том, что препроцессор переводит те предложения, которые он «понимает», на ФОРТРАН. Результат его работы может затем поступить на вход обычного компилятора с языка ФОРТРАН. Как правило, пользователь не должен беспокоиться о ступенчатой записи структурных предложений — этим обычно занимается препроцессор. Таким



Рис. 7.3. Связь препроцессора и компилятора языка ФОРТРАН.

образом, все предложения могут начинаться с седьмой колонки или могут быть сдвинуты по желанию программиста. После препроцессора работает компилятор. Для обработки структурной программы на ФОРТРАНЕ используется стандартный язык управления заданиями. С его помощью можно вызывать сначала препроцессор, а затем — компилятор ФОРТРАНа.

### *Метки предложений*

Препроцессор ФОРТРАНа порождает предложения ФОРТРАНа, помеченные, если нужно, скажем, начиная с 00100 с шагом 10. Это верно как для главной программы, так и для подпрограммы. Программисту не следует использовать эти метки для других предложений в структурной программе (таких, как объявления формата). Отсюда следует, что метки, кончающиеся на 1, 2, ..., 9, можно использовать в программе где угодно (например,  $DO\ 225\ I = 1, 100$ ). Если метка какого-либо предложения дублируется препроцессором, то при компиляции с ФОРТРАНа обнаруживается ошибка.

### *Предложения препроцессора*

**ЕСЛИ-ТО-ИНАЧЕ.** Для препроцессора ИБМ ключевое слово IF (ЕСЛИ), за которым следует пробел, означает начало развилки. Ключевые слова THEN (ТО) и ELSE (ИНАЧЕ) располагаются каждое на отдельной строке и могут начинаться с седьмой колонки или с любой другой, поскольку препроцессор сам создает ступенчатый текст. Например, можно написать так

```
IF (A.GT.B)
  THEN
    HOLD = A
    A = B
    B = HOLD
  ELSE
    CALL GCD(A,B,Q,R)
  END IF
```

и получить следующую ступенчатую выдачу:

```
IF (A.GT.B)
  THEN
    HOLD = A
    A = B
    B = HOLD
  ELSE
    CALL GCD(A,B,Q,R)
  END IF
```

Предложение END IF (ВСЕ-ЕСЛИ) обозначает конец конструкций ЕСЛИ-ТО и ЕСЛИ-ТО-ИНАЧЕ и также записывается на отдельной строке. Все структуры, начинающиеся с ЕСЛИ, заканчиваются предложением ВСЕ-ЕСЛИ.

**Повторение.** Используются ключевые слова DO (ЦИКЛ) и WHILE (ПОКА), за которыми следует логическое выражение в скобках:

```
DO WHILE (INPUT.LT.9999)
  C
  C
  C
END DO
```

Завершается эта конструкция предложением END DO (ВСЕ-ЦИКЛ). Предложение DO UNTIL (ЦИКЛ-ДО) записывается аналогично:

```
DO UNTIL (INPUT.GE.9999)
  C
  C
  C
END DO
```

## Примеры

Предлагаются две простые структурные программы, одна из которых иллюстрирует применение препроцессора.

### Сортировка методом пузырька

При сортировке методом «пузырька» числа в массиве меняются местами, если две соседние величины располагаются не в правильной (в нашем случае — возрастающей) последовательности. (Основная идея состоит в том, что наименьшие значения *всплывают* наверх.) При этом методе производится просмотр массива до тех пор, пока все нужные пары элементов массива не будут сравнены и, возможно, переставлены. После первого просмотра массива наибольшее значение будет находиться в элементе массива с наибольшим номером (например, если сортируется массив из десяти элементов, то наибольшее значение окажется в десятом элементе).

Разрабатывая алгоритм решения этой задачи на псевдокоде нисходящим методом, мы можем начать со следующих предложений:

Читать элементы массива  
Распечатать элементы массива  
Сортировать по возрастанию  
Напечатать отсортированный массив

Детализируя это основное решение, мы видим, что необходимо расписать предложение *сортировать*:

$K = \text{число элементов массива} - 1$

ЦИКЛ-ПОКА  $K \geq 1$

    Переместить наибольшее целое в последний элемент массива

    Вычесть 1 из  $K$

ВСЕ-ЦИКЛ

Теперь распишем *переместить*:

    Присвоить 1 указателю пары ( $J$ )

    ЦИКЛ-ПОКА  $J \leq K$

        Сравнить очередную пару и переставить, если нужно

        Добавить 1 к  $J$

    ВСЕ-ЦИКЛ

Наконец, распишем *сравнить и переставить*:

        ЕСЛИ элемент массива( $J$ ) > элемент массива ( $J + 1$ )

            Врем = элемент массива ( $J$ )

            Элемент массива( $J$ ) = элемент массива( $J + 1$ )

Элемент массива  $(J + 1) = \text{Врем}$   
ВСЕ-ЕСЛИ

Объединяя эти последовательные расширения, получаем окончательное решение задачи на псевдокоде:

Читать элементы массива

Распечатать элементы массива

$K = \text{число элементов массива} - 1$

ЦИКЛ-ПОКА  $K \geq 1$

Присвоить 1 указателю пары (J)

ЦИКЛ-ПОКА  $J \leq K$

ЕСЛИ элемент массива (J) > элемент массива (J+1)

Врем = элемент массива (J)

Элемент массива (J) = элемент массива (J + 1)

Элемент массива (J + 1) = Врем

ВСЕ-ЕСЛИ

Добавить 1 к J

ВСЕ-ЦИКЛ

Вычесть 1 из K

ВСЕ-ЦИКЛ

Напечатать отсортированный массив

На рис. 7.4 приведен ступенчатый текст этой программы. Это пример выдачи препроцессора ИБМ<sup>1)</sup>. На рис. 7.5 приведена соответствующая выдача компилятора ИБМ, работающего после препроцессора.

### *Измерения загрязнения воздуха*

Обратимся к задаче измерения загрязнения воздуха из гл. 5, где речь шла о пошаговой детализации. В этой задаче измерения производились около дымовой трубы большого завода каждую минуту в течение 24 часов. Мера загрязнения — число частиц, деленное на миллион (PPM). Нормальная величина загрязнения находится в диапазоне от 10 000 до 90 000 PPM. Программа делает следующее:

1. Вычисляет среднюю величину загрязнения для каждого из 24 часов.

2. Подсчитывает число нарушений в час. Нарушением считается ситуация, когда величина загрязнения в течение пяти минут подряд принимает значение свыше 100 000 PPM. Таким образом, если в течение десяти минут суммарное значение загрязнения выше 100 000 PPM, это трактуется как два нарушения; максимальное число нарушений в час может

---

<sup>1)</sup> Reprinted by permission from IBM. FORTRAN Preprocessor for Structured Programming, Form No. SB21— 1614, IBM, White Plains, New York, 1975. Courtesy of International Business Machine Corporation.

```

C      ****
C      *** BUBBLE SORT ***
C      ****
      DIMENSION IX(25)
      READ (1,5) (IX(I),I=1,10)
5     FORMAT (10I3)
      WRITE (3,6)
6     FORMAT (1H1,'BUBBLE SORT')
      WRITE (3,7)
7     FORMAT(1H0,'THIS IS THE LIST OF UNSORTED INTEGERS')
      WRITE (3,8) (IX(I),I=1,10)
8     FORMAT (1H ,10I6)
      N=10
      K=N-1
      DO WHILE (K.GE.1)
        J=1
        DO WHILE (J.LE.K)
          IF (IX(J).GT.IX(J+1))
            THEN
              IY=IX(J)
              IX(J)=IX(J+1)
              IX(J+1)=IY
            END IF
          J=J+1
        END DO
        K=K-1
      END DO
      WRITE (3,9)
9     FORMAT (1H0,'THIS IS THE SORTED LIST OF INTEGERS')
      WRITE (3,8) (IX(I),I=1,10)
      END

```

Рис. 7.4. Форматный листинг, распечатанный препроцессором.

```

C      ****
C      *** BUBBLE SORT ***
C      ****
      DIMENSION IX(25)
      READ (1,5) (IX(I),I=1,10)
5     FORMAT (10I3)
      WRITE (3,6)
6     FORMAT (1H1,'BUBBLE SORT')
      WRITE (3,7)
7     FORMAT(1H0,'THIS IS THE LIST OF UNSORTED INTEGERS')
      WRITE (3,8) (IX(I),I=1,10)
8     FORMAT (1H ,10I6)
      N=10
      K=N-1
00100 IF(.NOT.(K.GE.1)) GO TO 00110
        J=1
00120 IF(.NOT.(J.LE.K)) GO TO 00130
          IF(.NOT.(IX(J).GT.IX(J+1))) GO TO 00140
            IY=IX(J)
            IX(J)=IX(J+1)
            IX(J+1)=IY
          00140 CONTINUE
            J=J+1
            GO TO 00120
        00130 CONTINUE
          K=K-1
          GO TO 00100
      00110 CONTINUE
        WRITE (3,9)
        9     FORMAT (1H0,'THIS IS THE SORTED LIST OF INTEGERS')
        WRITE (3,8) (IX(I),I=1,10)
        END

```

Рис. 7.5. Листинг, распечатанный компилятором ФОРТРАНа.



```

C
C
C
C PROGRAM INPUT
C   POLLUTANT READINGS IN PARTS PER MILLION, MEASURED ONCE PER
C   MINUTE FOR 24 HOURS
C
C
C PROGRAM PROCESSES
C   AVERAGE THE READINGS PER HOUR
C   TEST FOR VIOLATIONS ( 5 CONSECUTIVE READINGS OVER 100000)
C
C
C PROGRAM OUTPUTS
C   REPORT SHDWING RESULTS FOR EACH HOUR
C   A. MEAN OF READINGS
C   B. NUMBER OF VIOLATIONS THIS HOUR
C
C -----PSEUDO-CODE SOLUTION-----
C   PRINT HEADINGS
C   INFRACTIONS = 0
C   LOOP 24 TIMES
C     READ DATA THIS HOUR (60 INPUT VALUES)
C     SUM = 0
C     VIOLATIONS = 0
C     LOOP 60 TIMES
C       ADD VALUE TO SUM
C       IF VALUE > 100000 THEN
C         ADD 1 TO INFRACTIONS
C       IF INFRACTIONS = 5 THEN
C         ADD 1 TO VIOLATIONS
C       INFRACTIONS = 0
C     ENDIF
C   ENDIF
C   INFRACTIONS = 0
C   ENDIF
C   ENDLOOP
C   MEAN = SUM/60
C   PRINT RESULTS THIS HOUR
C   ENDLOOP
C   END SMOG
C -----
C
C
C
C
C
C

```

Рис. 7.6. Решение на псевдокоде задачи об измерении загрязнения воздуха.

быть, следовательно, равно 12.) Нарушение на границе часов относится к тому часу, во время которого оно заканчивается.

3. Выводит отчет, содержащий время в часах, среднюю величину загрязнения и число нарушений в течение этого часа.

Программа, записанная на ФОРТРАНе, разделена на три части: решение задачи на псевдокоде, объявления и операторы. На рис. 7.6 приведено полученное в гл. 5 решение задачи на псевдокоде. Объявления приведены на рис. 7.7, операторы — на рис. 7.8.



Подробности способа представления программы на языке ФОРТРАН лучше всего определять с учетом особенностей данной организации. Если в ней еще не приняты конкретные стандарты, можно взять за основу наш. Например, типы данных и границы массивов нужно указывать в объявлениях типа. Каждая строка должна содержать одно имя переменной. Объявление типа может занимать несколько строк.

```

C
1000 WRITE (OUTPUT,0010)
      INFRAC = 0
      DO 1500 HOUR = 1,24
      * READ (INPUT,0020,END=1600) (PPMVAL(1), 1=1,60)
        SUM = 0.0
        VIOLAT = 0
        DO 1300 MINUTE = 1,60
          SUM = SUM + PPMVAL(MINUTE)
          IF (PPMVAL(MINUTE) .GT. PPMMAX) GO TO 1100
          PPMVAL LESS OR EQUAL TO PPMMAX
C          INFRAC = 0
          GO TO 1200
        1100 CONTINUE
C          ELSE PPMVAL GREATER THAN PPMMAX
          INFRAC = INFRAC + 1
          IF (INFRAC .LT. INFMAX) GO TO 1120
C          INFRAC IS EQUAL TO INFMAX
          INFRAC = 0
          VIOLAT = VIOLAT + 1
        1120 CONTINUE
C        ENDOF
      1200 CONTINUE
C      ENDOF
    1300 CONTINUE
C    ENDDO
      MEAN = SUM/60.0
      WRITE (OUTPUT,0030) HOUR,MEAN,VIOLAT
    1500 CONTINUE
C  ENDDO
    1600 CONTINUE
      RETURN
      END

```

Рис. 7.8. Операторы из решения на ФОРТРАНе задачи об измерении загрязнения воздуха.

допуская до 20 имен в одном объявлении. (Вот почему в строках-продолжениях мы употребляем буквы, а не цифры.) В объявлениях типа содержится вся информация о массивах — объявление массивов становится ненужным. На рис. 7.7 ключевые слова (INTEGER, REAL) начинаются с колонки 8. Имя переменной начинается с колонки 24. В строках-продолжениях запятая предшествует имени данного и помещается в колонке 23. Каковы преимущества такого формата при программировании? При отладке? При сопровождении программы?

Только одно имя должно указываться и в каждой строке объявления начальных данных. Если нужно придать начальные значения в одном объявлении нескольким данным, то используются строки-продолжения. Для наглядности используйте пробелы, особенно при указании начальных значений массива, когда для полного определения всех его элементов используются строки-продолжения.

Рис. 7.7 показывает также, как представлять объявления формата. Насколько возможно, код преобразования (например, X или T) и связанный с ним описатель поля должны располагаться вместе — каждая пара на своей строке. Если компилятор воспринимает код преобразования T, то управлять расположением данных при выводе на печать лучше с его помощью. Лучше также использовать литералы вместо формата текстового поля (например, 'MEAN (PPM)' вместо 9HMEAN (PPM)).

Пустые строки-комментарии используются для того, чтобы операторы начинались с новой страницы. Они вставлены уже после проверки программы, и на рис. 7.7, 7.8 можно видеть, как операторы разместились на печатной странице.

Имена данных должны быть мнемоническими настолько, насколько позволяет ограничение на их длину. Даже индексные переменные (MINUTE и HOUR) предпочтительнее обычных употребляющихся «I», «J» и т. д.

В операторах ввода и вывода для идентификации набора данных (в некоторых реализациях — логического номера устройства) нужно использовать имя переменной типа целый. Константы в этом случае хуже как с точки зрения наглядности, так и легкости сопровождения программы. Чтобы было легче читать элементы списков, нужно применять пробелы и строки-продолжения.

## Выводы

### Сегментация/Модульность

1. Сегментами должны быть маленькие модули, транслируемые отдельно.
2. Текст модуля должен занимать не более одной страницы.
3. Модули должны программироваться и тестироваться нисходящим методом, т. е. первым должен программироваться и тестироваться головной модуль, вообще говоря, с заглушками вместо подчиненных модулей.
4. Каждый модуль должен быть *простой программой*.
5. Модули со сложной логикой должны предваряться одним-двумя абзацами комментариев.

## *Структурирование*

1. Расположение фрагментов программы должно соответствовать порядку ее выполнения (кроме явных структур-повторений).

2. Для изображения развилки нужно использовать условный оператор.

3. Глубина вложенности условных операторов не должна превышать трех.

4. Повторение нужно представлять либо оператором цикла, либо комбинацией условного оператора и оператора перехода.

5. Выбор должен быть реализован вычисляемым оператором перехода.

## *Удобство восприятия*

1. Если возможно, используйте препроцессор.

2. Логические части программы разделяйте пустыми строками.

3. Вставляйте комментарии только по мере необходимости.

4. Все переменные, за исключением управляющих переменных в операторах цикла, должны иметь мнемонические имена.

5. Предложения внутри ЕСЛИ-ТО-ИНАЧЕ и тела цикла записывайте с отступами — это проясняет логику программы и упрощает ее изменения.

## *О стиле программирования*

1. Используйте очень осторожно или совсем не используйте операторы перехода, останова, возврата, выхода.

2. Используйте каждый индикатор только с одной определенной целью.

3. Используйте имена для констант, определяя их значения с помощью объявления начальных данных.

4. Начальные значения переменным и индикаторам присваивайте операторами, а не объявлениями начальных данных.

## *Контрольные вопросы и упражнения*

1. В каких случаях можно оправдать затраты времени и сил на запись на бланке и перфорацию текста программы на псевдокоде?

2. Следует ли в свете идей структурного программирования исключить из ФОРТРАНа объявления эквивалентности и общих объектов?

3. Определите стандартный формат программирования объявлений типа. Расширьте этот стандарт для объявлений начальных данных и формата.

4. Каковы достоинства и недостатки использования оператора продолжения для отделения альтернатив ТО и ИНАЧЕ, а также для указания конца развилки?

5. Если используемый компилятор воспринимает условные логические операторы, имеет ли смысл использовать условные арифметические операторы? Почему?

6. Перечислите некоторые из приемов программирования (или *стандартных* методов, применяющихся уже многие годы), где опытный программист на ФОРТРАНе должен «держать ухо востро», если желает получить программу, содержащую мало ошибок и легко изменяемую?

7. Какие предложения ФОРТРАНа не следует использовать или следует употреблять особенно осторожно?

8. Каковы достоинства и недостатки препроцессора ФОРТРАНа?

9. Возьмите программу, изображенную на рис. 7.5, и перепрограммируйте ее с учетом указаний этой главы (мнемонические имена данных, использование объявлений типа, а не объявлений массивов, определение значений переменных, изображающих логические номера устройств ввода-вывода, в объявлении начальных данных). Кроме того, измените программу так, чтобы проходы по массиву прекращались, если при очередном проходе не было произведено ни одной перестановки. (Это означает, что элементы массива уже упорядочены.) Преимущество этого изменения (его можно реализовать с помощью индикатора) состоит в том, что время сортировки, вообще говоря, уменьшается. Если позволяет компилятор, используйте логическую переменную (т. е. типа LOGICAL).

10. Напишите на языке ФОРТРАН программу порождения всех простых чисел от 1 до 1000 и печати их «в строчку».

## Структурное программирование на ПЛ/I

ПЛ/I (обратите внимание — римская цифра I) — язык программирования, пригодный как для научных, так и для планово-экономических задач. Он обладает гибкостью, которая до него была присуща только языку ассемблера, и дает возможность пользоваться преимуществами современных достижений в архитектуре ЭВМ. Благодаря богатству выразительных средств ПЛ/I используется в самых разнообразных приложениях. Его синтаксис позволяет легко программировать базовые структуры — следование, развилку (ЕСЛИ-ТО-ИНАЧЕ) и повторение (ЦИКЛ-ПОКА). Дополнительные структуры — ЦИКЛ-ДО и ВЫБОР — можно моделировать так, что их будет легко и программировать, и понимать.

Предполагается, что читатель этой главы знаком с ПЛ/I. В ней говорится о трех аспектах структурного программирования: модульности, структурности и ясности.

### Модульность

#### *Модульная организация*

На рис. 8.1 представлена схема иерархии некоторых модулей в программе, обслуживающей расчеты с покупателями. Каждый прямоугольник в схеме представляет некоторый модуль этой программы. Каждый из этих модулей — *внешняя* процедура, которая может содержать любое количество *внутренних* процедур. Модули разрабатывались нисходящим методом, причем каждая ветвь этой схемы почти наверняка была полностью завершена до начала детального проектирования, программирования и тестирования следующих ветвей. Предположим, что самая левая ветвь на схеме иерархии завершена и, двигаясь вниз по схеме, мы оказались на стадии программирования модуля (заштрихован на рис. 8.1), который составляет *регистр счетов*. Общий вид этого регистра показан на рис. 8.2.

В этом модуле, как и во всех других, количество операторов должно быть не больше допустимого в соответствующей организации. В большинстве организаций, применяющих ПЛ/I и методы структурного программирования, считают разумным ограничивать количество операторов до 50—60 строк входного текста с тем, чтобы все они помещались на одной

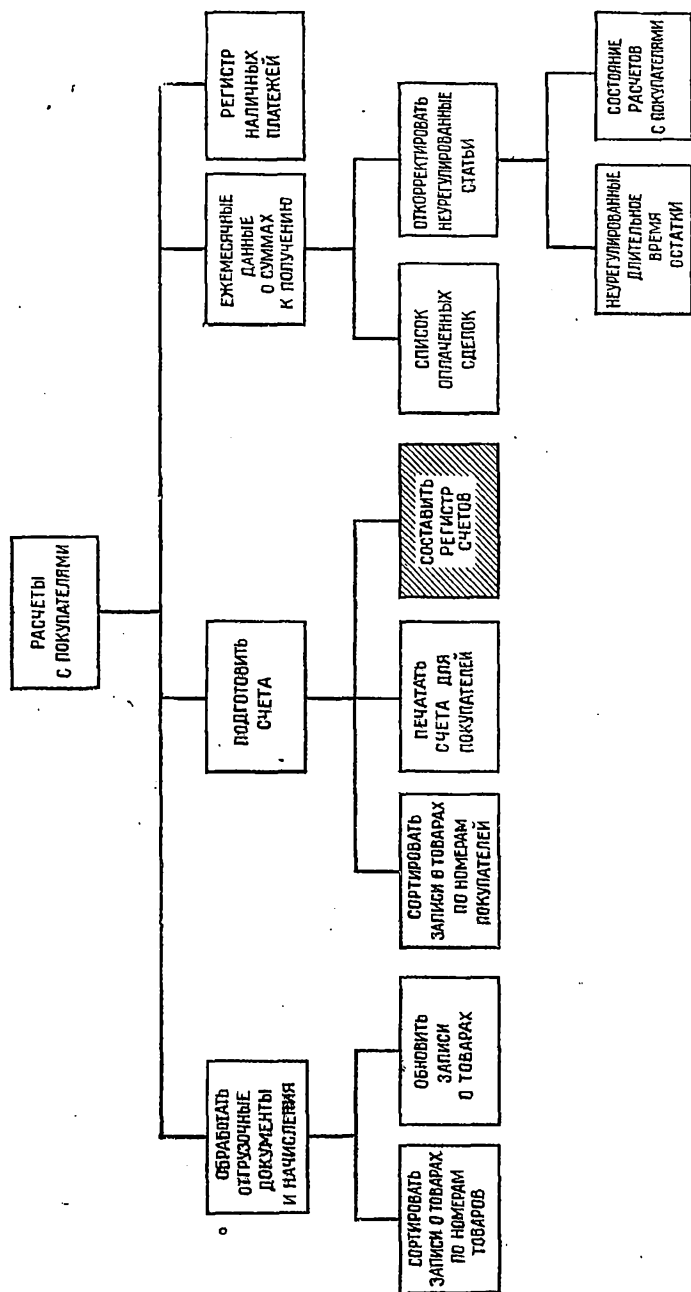


Рис. 8.1. Модули в программе расчетов с покупателями.



РЕГИСТР СЧЕТОВ					
Номер счета	Номер покупателя	Имя покупателя	Стоимость товара	Налог	Общая стоимость
16799	14753	HOBYRANA	52.50	2.61	54.81
16800	14758	HOFF ELEC CO	86.12	4.30	90.42
16801	14764	HOLBRO PAINT CO	92.20	4.61	96.81
16802	14770	HOLLAND CO	80.15	4.01	84.16
16803	14775	HOVER CO	912.12	45.61	957.73
16804	14780	HOWELL CO	200.10	10.00	210.10
16805	14787	HOWKEN UTILITIES	315.00	15.75	330.75
16806	14790	HUGEL MOTORS	930.30	46.65	
16807	14792	IRVING TIRE CO.	1,950.00		
16808	14795	IVY BARN INC.			
16809	15800	JACKSON SALES			
			00	2.02	672.42
			1,050.10	40.00	840.10
			280.50	92.51	1,142.61
			555.10	14.03	294.53
			106.66	27.76	582.86
			1,133.00	5.33	111.99
				56.65	1,189.65
16843	15970	ZACHARY BROS.			
16844	15972	ZEPHYR ELEC CO			
		Всего	61,564.67	3,078.23	64,642.90

Рис. 8.2. Образец регистра счетов.

странице печатающего устройства. В некоторых организациях можно создавать большие модули и подразделять их на сегменты по 50—60 строк в каждом. (Во многих случаях число строк в сегменте будет на самом деле порядка 25—40.)

### Как сегментировать модуль

Предположим, что модуль СОСТАВИТЬ РЕГИСТР СЧЕТОВ уже написан на псевдокоде. При внимательном изучении этого текста обнаруживается, что на ПЛ/І модуль займет более 60 строк. Значит, этот модуль нужно сегментировать. В записях о товарах, которые будут обрабатываться этим модулем, содержатся, в частности, такие поля:

Номер покупателя	Номер торгового агента	Номер счета	Номер товара	Размер закупки	Цена	Описание
------------------	------------------------	-------------	--------------	----------------	------	----------

Сегментированием лучше всего заниматься на этапе пошаговой детализации. После нескольких итераций станет ясно, уместится ли модуль на одной странице. Пусть в нашем примере в процессе пошаговой детализации возник фрагмент

ПРИ конце файла

Есть записи о товарах = нет

Есть записи о товарах = да

ЦИКЛ-ПОКА есть записи о товарах

Читать запись о товаре

ЕСЛИ есть записи о товарах

Проверить данные

ЕСЛИ данные в порядке

Вызвать обработку записи о товаре

ИНАЧЕ

Вызвать обработку ошибки в данных

ВСЕ-ЕСЛИ

ИНАЧЕ

Вызвать подведение итога

ВСЕ-ЕСЛИ

ВСЕ-ЦИКЛ

Такой способ обработки конца файла слегка отличается от рассмотренного нами раньше. Это естественно, так как псевдокод не определен строго, и просто нужно писать так, чтобы быть как можно ближе к выбранному языку программирования. Поэтому естественно включить в наш текст реакцию на прерывание по концу файла, хотя в других языках, отличных от ПЛ/І, ее может и не быть. Этот текст можно теперь перевести на ПЛ/І и образовать *головной сегмент* в нашем модуле.

Головной сегмент — это план всего модуля. Когда он готов, можно сегментировать модуль тем же самым иерархическим способом, как создается вся система или программа.

Рис. 8.3 показывает иерархию сегментов нашего модуля. Сегменты нижнего уровня могут быть внутренними процедурами, которые вызываются с помощью оператора CALL. Предположим, однако, что сегмент *Проверить данные* заготовлен заранее и помещен в текстовую библиотеку. В таком случае этот сегмент будет подключен с помощью % INCLUDE, и управление попадет к нему естественным путем без какого-либо оператора передачи управления.

До начала программирования рассматриваемого модуля должны быть полностью готовы его тесты и соответствующее задание на языке управления заданиями. Головной сегмент нужно программировать первым. Конечно, все сегменты, нужные для проверки, должны быть заранее запрограммированы как заглушки или сразу целиком, если они очень простые или относительно короткие. В нашем модуле все эти сегменты (отличные от головного) разумно вначале изготовить в виде заглушек. Это дало бы возможность проверить головной сегмент, а затем можно было бы уже заняться программированием и проверкой других сегментов.

Комбинация иерархического и операционного подходов — наилучший метод *планирования* той последовательности, в которой программируются и тестируются сегменты нижнего уровня. Эта последовательность может быть, например, такой: 1) Головной сегмент (INVREG), 2) Обработать запись о товаре (PROCESS ITEM), 3) Общий итог (FINAL TOTAL), 4) Проверить данные (VALDATA), 5) Ошибка в данных (DATA ERROR).

При оформлении текста модуля желательно позаботиться о том, чтобы каждый сегмент помещался на отдельной странице. (В некоторых случаях на одной печатной странице можно помещать два или три маленьких сегмента с промежутками между ними.) Большинство компиляторов ПЛ/И предоставляют следующие средства, позволяющие управлять оформлением листинга: 1) операторы периода компиляции %PAGE и %SKIP(*n*), где *n* — число пропускаемых строк, или 2) колонка 1 строки исходного текста (предложения ПЛ/И в таком случае должны помещаться в колонках 2—72), где можно указать управляющую литеру. Например «1» означает переход к новой странице, пробел — пропуск одной строки и т. д. Обычно при этом на языке управления заданиями или в предложении PROCESS нужно указать, какие именно из средств управления листингом будут использоваться. Другой способ достичь той же цели — просто вставлять в нужных местах пустые карты.

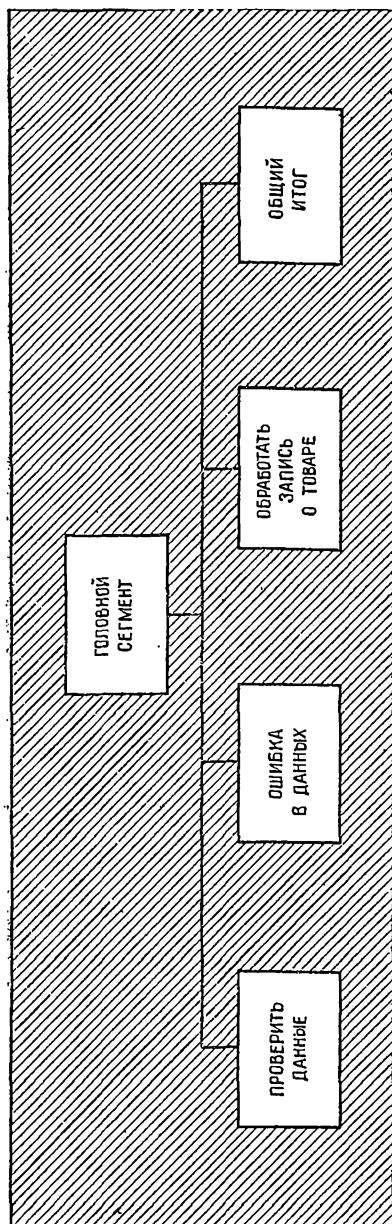


Рис. 8.3. Сегменты модуля, готовящего регистр счетов.

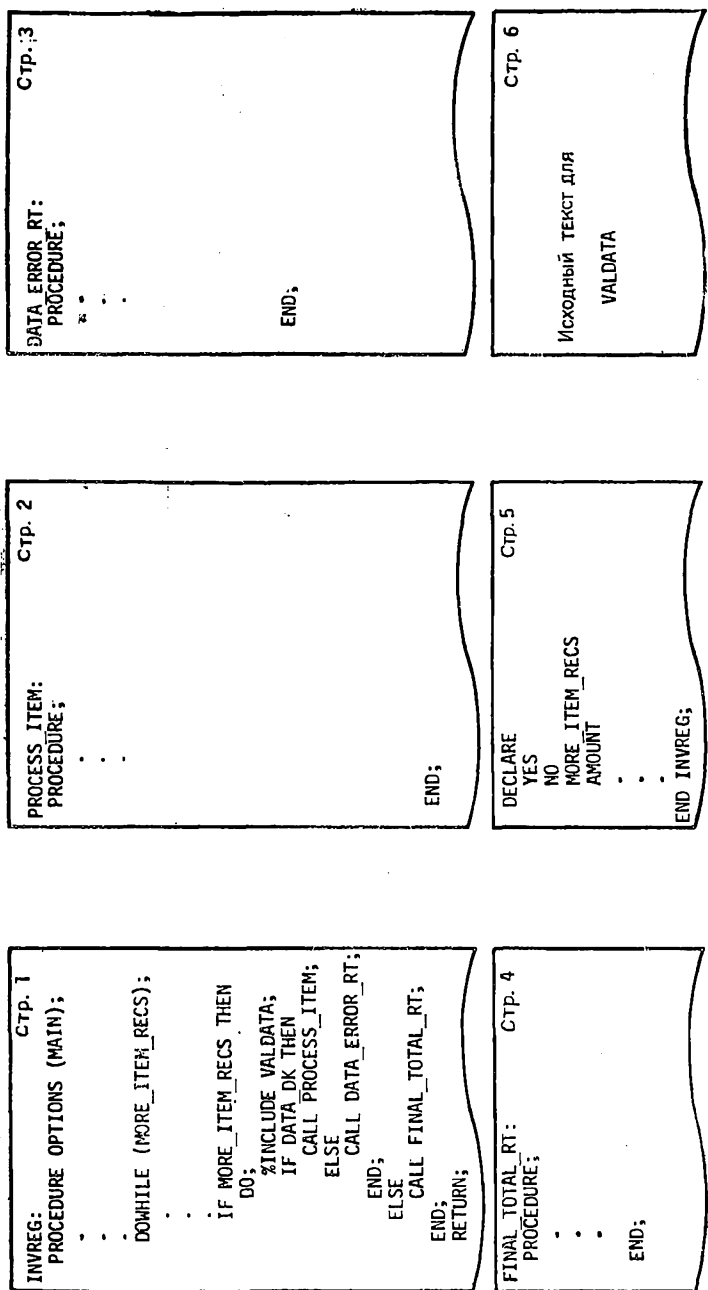


Рис. 8.4. Примеры сегментированных модулей на ПЛ/1.

Если в нашем модуле каждый сегмент разместить на отдельной странице, то его листинг выглядел бы примерно так, как показано на рис. 8.4. (Некоторые компиляторы ПЛ/1 выдают два вида листингов — в одном случае библиотечные тексты размещаются в конце, в другом случае эти тексты помещаются прямо в то место, где будут исполняться.) В нашем примере предложения DECLARE сгруппированы и помещены в конце программы.

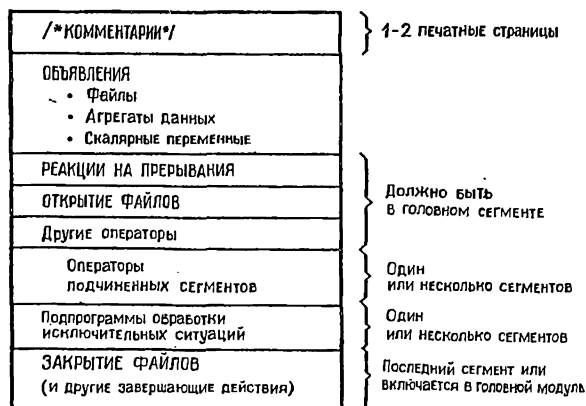


Рис. 8.5. Последовательность фрагментов в модуле на ПЛ/1.

Каждый сегмент должен быть *простой программой* с такими свойствами:

1. Вход один и расположен в начале страницы.
2. Выход также один. Это должно быть последнее предложение на странице, и оно должно обеспечивать возврат в вызвавший сегмент.
3. Поток управления направлен на каждой странице сверху вниз.
4. Логически связанные фрагменты располагаются рядом.

На рис. 8.5 показан вариант расположения фрагментов модуля. Листинг начинается с одной-двух страниц комментариев, описывающих сам модуль и выбранный метод решения задачи. (Если угодно, можно в качестве разъяснений перед программой поместить ее текст на псевдокоде.) Когда в структурированной программе используются мнемонические имена данных, необходимость в пояснениях существенно уменьшается. Однако перед особенно сложными сегментами комментарии полезны.

Предложения DECLARE можно собрать или в начале программы (перед выполняемой частью программы) или в самом ее конце. Трудно дать более точные правила размещения объ-

явлений, так как иногда программисту могут понадобиться средства динамического распределения памяти ПЛ/И. А это означает, что DECLARE могут появляться и внутри модуля (в его внутренних процедурах) или в начале блоков.

Управление может попасть в некоторый сегмент четырьмя способами:

*С помощью оператора CALL:* этот способ применяется наиболее часто. В случае сегментов операторы CALL всегда относятся к внутренним процедурам.

```
IF MORE_TRANSACTIONS THEN
  CALL PROCESS_TRANSACTION;
ELSE
  CALL WRAP_UP;
```

*С помощью указателя функций:* это другой способ вызова подпрограмм в ПЛ/И. Часто сегменты, вызываемые таким способом, реализуют конкретную расчетную функцию. Например:

```
TAX_AMT = COMPUTE_TAX(MERCHANDISE_AMT);
```

*С помощью препроцессора:* есть несколько ограничений на тип исходных предложений, которые могут располагаться в текстовой библиотеке. Они зависят от применяемого компилятора. При структурном программировании этот способ наиболее часто применяется для вставки текста непосредственно в то место, где он будет исполняться (как в случае сегмента ПРОВЕРИТЬ ДАННЫЕ в модуле РЕГИСТР СЧЕТОВ). Приведем еще один пример:

```
IF NEW_ITEM_NUMBER = SAVED_ITEM_NUMBER THEN
  %INCLUDE COMPUTE;
ELSE
  IF NEW_ITEM_NUMBER > SAVED_ITEM_NUMBER THEN
    %INCLUDE OUTREC;
  ELSE
    %INCLUDE SEQERR;
```

Одно из преимуществ препроцессора состоит в том, что вставленный текст будет выполняться подряд (CALL или указатель функции не потребуются). Таким образом, на объектном уровне с такими сегментами не связано никаких накладных расходов, поскольку отсутствуют какие-либо операторы связи с подпрограммой.

*С помощью оператора GOTO:* при условии, что общее правило возврата в точку вызова остается в силе, а программа

остается простой и структурированной, она вполне может содержать и операторы GOTO. (Неприемлемо лишь бесконтрольное употребление этого оператора.) Передачи управления сегменту с помощью GOTO по сравнению с CALL необязательно уменьшают накладные расходы. В некоторых компиляторах обращение с помощью CALL без аргументов к внутренним подпрограммам без объявлений обходится дешевле, чем реализация с помощью меточной переменной с GOTO. Поэтому нужно сначала выяснить, каковы в этом смысле характеристики применяемого компилятора, а лишь затем принимать решение об использовании GOTO при передаче управления внутрь сегмента и возврате из него. Если вы все-таки решите применять GOTO, вначале запрограммируйте сегменты как процедурные блоки и вызывайте их с помощью CALL. И только после того, как программа проверена, можно ее оптимизировать, применив GOTO. Для этого операторы CALL заменяются GOTO, удаляются заголовки внутренних процедур и соответствующие END или RETURN заменяются GOTO.

Выход из области DO по GOTO и возврат в эту область вообще не допустимы в ПЛ/И. Поэтому может случиться и так, что заменить CALL на GOTO окажется невозможным.

Способ входа в сегмент определяет и способ выхода (RETURN, END, GOTO). Самое важное, чтобы управление возвращалось к вызвавшему сегменту. Исключением из этого правила может быть только случай, когда обнаружена неисправимая ошибка или работу нужно немедленно прекратить. Если, например, неисправимая ошибка обнаружена в сегменте низкого уровня, то может оказаться неразумным передавать соответствующий индикатор вверх по всей иерархии сегментов и лучше немедленно остановить исполнение в том же сегменте низкого уровня.

Итак, можно сделать вывод, что самое важное — постраничная организация логики модуля. Сегментировать нужно выполняемые предложения ПЛ/И. Невыполняемые предложения, подобные DECLARE, группируются вместе и могут как делиться, так и не делиться на отдельные страницы.

Модули и/или сегменты могут включать тексты, скопированные из текстовой библиотеки с помощью операции препроцессора %INCLUDE, если она допустима для используемого компилятора. Вызов с помощью CALL внутренних процедур рассматривается как вызов сегментов, а вызов этим же оператором внешних процедур (отдельно скомпилированных подпрограмм) считается обращением к модулям более низкого уровня. Конечно, вся выполняемая часть программы — это комбинация базовых структур и некоторых дополнительных структур, допустимых в вашей организации.

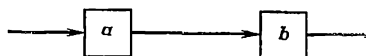


## Структурирование ПЛ/I-программ

Структурирование состоит в том, что берутся три базовые структуры (и несколько дополнительных) и заданная функция или подфункция представляется в виде их комбинации. Эти структуры кратко рассматриваются ниже вместе с примерами их реализации на ПЛ/I.

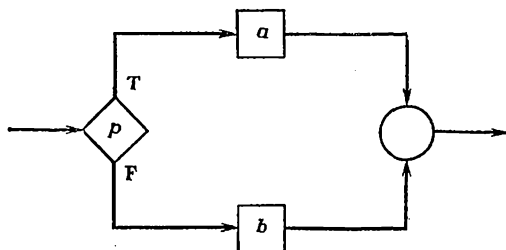
### Следование

Эта структура состоит из двух обрабатывающих блоков, причем управление передается от одного блока другому в соответствии с их расположением в этой структуре. Блок может представлять отдельный оператор, например READ, или несколько операторов, например группу DO, указатель внутренней процедуры или блок в смысле языка ПЛ/I. Эта структура изображается следующим образом:



### Развилка

Конструкция ЕСЛИ-ТО-ИНАЧЕ должна быть знакома тем, кто пользуется ПЛ/I. В ней указывается проверка и две альтернативы, определяющие, какой из двух функциональных блоков или вариантов обработки нужно выполнить.



На языке ПЛ/I эта конструкция выражается структурным условным предложением IF. В нем после THEN и ELSE можно указать одно произвольное предложение (кроме GOTO). Например:

```
IF BALANCE - PAYMENT - CREDITS = 0 THEN  
  CALL ZERO_BALANCE_RT;  
ELSE  
  CALL NON_ZERO_BALANCE_RT;
```

Можно указать, конечно, и группу DO:

```
IF SUCCESSFUL THEN
  DO:
    A=B;
    C=D;
  END;
ELSE
  DO:
    A=X;
    C=Y;
  END;
```

Альтернативу ELSE можно опускать:

```
IF QTY_ON_HAND < RE_ORDER_QTY THEN
  CALL WRITE_PURCHASE_ORDER_RT;
```

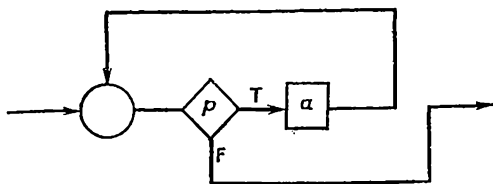
Когда требуются вложенные условные предложения, очень важно использовать мнемонические имена данных и выбирать как можно более простые условия. Если глубина вложенности превосходит 3, то читать программу становится трудно. Вложенные условные предложения нужно записывать с отступами, чтобы проявить их структуру.

```
IF DEGREE = BACHELORS THEN
  IF GRADE_PT_AVER >= 3.5 THEN
    PUT LIST ('HIRE'||NAME);
  ELSE
    PUT LIST ('CONSIDER'||NAME);
ELSE
  IF COMPARABLE_WORK_EXP THEN
    PUT LIST ('HIRE'||NAME);
  ELSE
    PUT LIST ('DO NOT HIRE'||NAME);
```

Ключевое слово ELSE нужно размещать, начиная с той же колонки, что и соответствующее IF. Вложенные условные предложения нужно всегда размещать целиком на одной странице.

### *Повторение*

**ЦИКЛ-ПОКА.** Эта конструкция служит основным средством изображения циклов. Она указывает, что некоторое действие должно повторяться до тех пор, пока остается истинным указанное условие. Это условие всегда проверяется до выполнения тела цикла. Следовательно, иногда тело может и ни разу не выполниться. Повторение изображается следующим образом:



Вот пример на ПЛ/1:

```

I = 1;
DO WHILE (I <= NO_DAYS);
  TOTAL = TOTAL + TEMPERATURES(I);
  I = I + 1;
END;

```

В теле цикла этого типа нужно всегда обеспечивать изменение значения того выражения, которое входит в условие, чтобы в конце концов это выражение стало ложным. Например, из только что приведенного примера нельзя исключить оператор

```

I = I + 1;

```

так как иначе программа заиклилась бы.

Предыдущий пример можно записать и с таким заголовком цикла:

```

DO I = 1 BY 1 WHILE (I <= NO_DAYS);

```

В этом случае индекс I автоматически увеличивается после каждого шага цикла. Если не записано условие окончания цикла, как, например, в таком заголовке

```

DO I = 1 BY 1

```

то управляющая переменная будет расти до тех пор, пока не возникнет ситуация переполнения (FIXEDOVERFLOW или OVERFLOW).

Вот еще одна форма цикла, которая может довольно часто встречаться в структурированных программах:

```

DO WHILE (MORE_TRANSACTIONS);
/* . . . */
END;

```

эта группа DO будет повторно выполняться до тех пор, пока значение MORE\_TRANSACTIONS-истина.

И другие формы циклов также описываются структурой ЦИКЛ-ПОКА. Так, заголовки

```
DO I = 1 TO 100 BY 1;
DO J = -10 TO +10;
DO K = 10 TO 20, 30 TO 40, 50 TO 60;
DO L = 1,8,9,11,6,13;
```

относятся к вариантам реализации этой структуры, потому что условие окончания цикла проверяется до выполнения его тела. Можно, конечно, указывать сразу и шаг, и условие ПОКА:

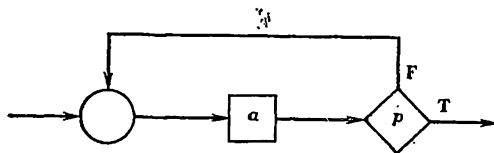
```
DO I = 1 TO 10 WHILE(P),
DO J = 1 TO 10 WHILE(A=B);
```

**ЦИКЛ-ДО.** Как уже говорилось ранее, структура ЦИКЛ-ДО по существу обеспечивает те же возможности изображать циклы, что и ЦИКЛ-ПОКА. Отличается она от ЦИКЛ-ПОКА в двух отношениях:

1. Условие окончания цикла проверяется *после* выполнения тела цикла. Поэтому тело цикла всегда будет выполняться хотя бы раз.

2. ЦИКЛ-ДО оканчивается, когда проверяемое условие становится истинным (ЦИКЛ-ПОКА кончается, когда проверяемое условие становится ложным).

Вот эта структура:



Заголовок цикла вида

**ЦИКЛ-ПОКА** ( $\neg$  выражение)

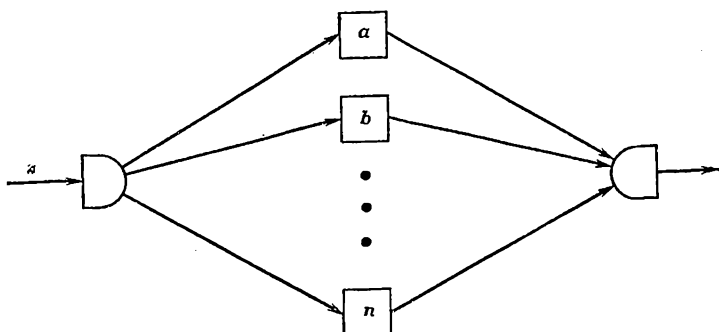
указывает, что тело цикла должно выполняться, пока «выражение» остается *не истинным*. Поэтому такой заголовок удобен для моделирования структуры ЦИКЛ-ДО. Например:

```

DECLARE DONE BIT(1);
DONE = NO;
DO WHILE ( $\neg$ DONE);
/*
  .
  .
  .
  IF RESULT < 0 THEN
    DONE = YES;
  END;
*/
```

## Выбор

Эта структура применяется тогда, когда по результатам проверки нужно выбрать одну из многих альтернатив. Это может быть изображено следующим образом:



Все пути через различные обрабатывающие блоки сходятся затем в одной и той же точке. Так как в ПЛ/И нет предложения CASE, то эту структуру нужно моделировать.

Один из способов представления выбора на ПЛ/И — несколько вложенных условий. Пусть, например, три цифры номера покупателя указывают тип расчетов, которые соответствуют этому покупателю. Основываясь на таком понимании этого номера, нужно вызвать соответствующую программу:

```
CASE=SUBSTR(CUSTOMER_NO,4,3);  
IF CASE = 110 THEN  
  CALL SHIP_PREPAID;  
ELSE IF CASE = 120 THEN  
  CALL SHIP_FOB;  
ELSE IF CASE = 130 THEN  
  CALL SHIP_COD;  
ELSE CALL SPECIAL_HANDLING;
```

Для небольшого количества альтернатив предпочтителен именно такой метод выражения структуры ВЫБОР. Но иногда нужно выбирать из 50, 75 или даже 100 альтернатив. В такой ситуации переход по меточной переменной — единственный практически приемлемый способ управления выбором альтернатив. Допустим для простоты, что нужно выбрать одну из четырех функций по значению переменной CODE. Если CODE = 1, то выполняется группа предложений, помеченная меткой CASE 1; если CODE = 2, то выполняется фрагмент программы, начинающейся с метки CASE 2 и т. д., если CODE больше 4 или меньше 1, то это ошибка и вызывается диагностическая программа.

```

DECLARE
CASE(0:4) LABEL INITIAL(ERROR,CASE1,CASE2,CASE3,CASE4),
CODE      FIXED BINARY(15);

```

Рис. 8.6. Объявление данных для структуры ВЫБОР.

На рис. 8.6 показаны подготовительные шаги, необходимые для реализации такого выбора. Объявляется массив с границами от 0 до 4 с метками соответствующих программ для CODE = 0, CODE = 1 и т. д. в качестве начальных значений. Вот как выглядит массив меток:

CASE (0)	ERROR
(1)	CASE 1
(2)	CASE 2
(3)	CASE 3
(4)	CASE 4

На рис. 8.7 показан общий способ программирования такой структуры. Структура ВЫБОР начинается с оператора GOTO, а метка END\_CASE указывает точку выхода из этой структуры. Между GOTO и меткой, отмечающей конец выбора, расположены фрагменты для каждой из альтернатив. Чтобы программа была более эффективной, переменная CODE объявлена как FIXED BINARY. Ее значение доставляется предложением GET LIST. Условное предложение IF проверяет, попадает ли это значение в диапазон границ массива и, если не попадает, устанавливает CODE равным 0, обеспечивая тем самым переход на диагностическую программу.

```

GET FILE (SYSIN) LIST(CODE);
IF CODE<0 | CODE>4 THEN
  CODE =0;
  GO TO CASE(CODE);

ERROR:

/* CODE FOR ERROR ROUTINE*/

GO TO END_CASE;
CASE1:

/* PROGRAMMING FOR CODE '1' */

GO TO END_CASE;
CASE2:

/* PROGRAMMING FOR CODE '2' */

GO TO END_CASE;
CASE3:

/* PROGRAMMING FOR CODE '3' */

GO TO END_CASE;
CASE4:

/* PROGRAMMING FOR CODE '4' */

GO TO END_CASE;
END_CASE:;

```

Рис. 8.7. Структура ВЫБОР, реализованная средствами ПЛ/1.

В только что описанном способе реализации выбора мы тщательно избегали использования GOTO везде, где это

возможно. Постоянно идет полемика между теми сторонниками структурного программирования, которые настаивают на том, что GOTO нужно избегать любой ценой, и теми, кто, подобно Кнуту<sup>1)</sup>, считает, что GOTO вполне оправдывает себя в некоторых особых ситуациях. Кнут приводит в качестве примера конструкцию выхода из сложной структуры по некоторому *событию*, в которой вполне оправдан переход на точку выхода по ошибке. Хотя в связи с этим он специально о прерываниях не говорит, но и прерывания можно было бы рассматривать как конструкцию выхода по событию. Если вы принадлежите к тем, кто больше склоняется к использованию GOTO, то, вероятно, вы вполне благосклонно отнесетесь и к следующим двум вариантам реализации проверки CODE в нашем примере:

```
IF CODE<0 | CODE>4 THEN
  GO TO ERROR;
ELSE
  GO TO CASE(CODE);

ON SUBSCRIPTRANGE
  GO TO ERROR;
(SUBRG): GO TO CASE(CODE);
```

Еще один способ реализации структуры ВЫБОР — объявить массив входов и в качестве начальных значений его элементов указать имена внутренних или внешних процедур. Тогда нужная процедура могла бы быть вызвана, скажем, так:

```
CALL TABLE(I CODE)
```

### Поиск

Так как в программировании часто приходится иметь дело с просмотром таблиц и поиском в таблицах, то имеет смысл ввести специальную структуру ПОИСК. При поиске в таблице *ключ поиска* сравнивается с *ключом таблицы*. (Обычно сравнивается на равенство, но не обязательно. Можно, например, искать самый первый элемент таблицы с ключом, *превышающим* ключ поиска.) *Табличная функция* — это та обработка, которая производится над найденной записью или данным какого-либо другого типа. Если для заданного ключа поиска не нашлось подходящего аргумента таблицы, то считается, что имеет место ошибка. Таким образом, завершить операцию поиска можно двумя способами: *нормальный выход*, когда

<sup>1)</sup> Donald E. Knuth, "Structured Programming with go to Statements", Computing Surveys, Vol. 6, No. 4, December 1974, p. 292. Copyright 1974, Association for Computing Machinery, Inc., reprinted by permission.

подходящий элемент найден, и *выход по ошибке*, когда подходящий элемент не найден.

Структура ПОИСК, с помощью которой можно реализовать как последовательный, так и двоичный поиск, показана на рис. 8.8. Сначала управление попадает на вершину этой структуры. Затем фиксируется тот элемент таблицы, который будет сравниваться с ключом поиска. Для этого может понадобиться

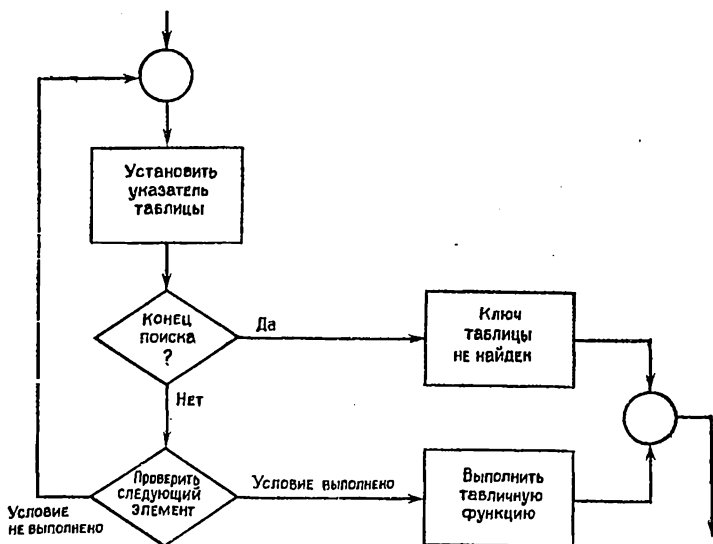


Рис. 8.8. Структура ПОИСК.

увеличить значение указателя при последовательном поиске или вычислить среднее верхней и нижней границ при двоичном поиске. Затем проверяется, остались ли в таблице элементы, которые можно сравнивать. Если же нет, то ключ таблицы не найден и нужен выход по ошибке.

Вторая проверка сопоставляет ключ таблицы с ключом поиска. Если сопоставление закончилось удачно (например, ключ таблицы равен ключу поиска), то нужно выполнить табличную функцию. Если сопоставление закончилось неудачно, то цикл продолжается. Структура ПОИСК имеет один вход и один выход. Управление после возможных ветвлений обязательно попадает на этот единственный выход. Цикл, включающий сопоставления, может выполняться много раз, но из двух обрабатывающих блоков будет работать только один, притом единственный раз. Если структура ПОИСК применяется для последовательного поиска, то на ПЛ/І может быть предложен



следующий вариант ее реализации, использующий GOTO внутри этой структуры. (LIMIT — число элементов в рассматриваемой таблице):

```
I = 0;
SEARCH_AGAIN:
I = I + 1;
IF I > LIMIT THEN
    CALL ARG_NOT_FOUND;
ELSE
    IF SEARCH_ARG = TABLE(I) THEN
        CALL PROCESS_TABLE_FUNCTION;
    ELSE
        GO TO SEARCH_AGAIN;
```

Имеется, конечно, много разнообразных алгоритмов поиска в таблицах. Вот, например, один из них:

```
DO I = 1 TO LIMIT WHILE(SEARCH_ARG != TABLE(I));
END;
IF I <= LIMIT THEN
    CALL PROCESS_TABLE_FUNCTION;
ELSE
    CALL ARG_NOT_FOUND;
```

Однако, разбирающий этот алгоритм должен потратить некоторые усилия, чтобы понять назначение пустого цикла (явных операторов в теле цикла DO нет). Это хитрый алгоритм и, согласно Кнуту, «почти всегда далеко не лучший способ поиска в массиве»<sup>1)</sup>.

Алгоритм последовательного поиска (если не применять структуру ПОИСК) можно было бы структурировать следующим образом:

```
ARG_FOUND = NO;
DO I = 1 TO LIMIT WHILE(~ARG_FOUND);
    IF SEARCH_ARG = TABLE(I) THEN
        DO;
            CALL PROCESS_TABLE_FUNCTION;
            ARG_FOUND = YES;
        END;
    END;
END;
IF ~ARG_FOUND THEN
    CALL ARG_NOT_FOUND;
```

Чтобы использовать индикатор ARG\_FOUND как показано в этом фрагменте, переменные YES и NO должны иметь значения 1 и 0 соответственно.

Структуру ПОИСК можно использовать также и для двоичного поиска. При двоичном поиске ключи таблицы должны быть

---

<sup>1)</sup> Donald E. Knuth, "Structured Programming with go to Statements", Computing Surveys, Vol. 6, No. 4, December 1974, p. 292. Copyright 1974, Association for Computing Machinery, Inc., reprinted by permission.

расположены в возрастающей (или убывающей) последовательности. Этот метод поиска содержит сопоставление ключа поиска с табличным ключом, выбранным из ее середины. К примеру, если у нас 500 табличных ключей, то ключ поиска вначале сравнивается с 250-м ключом таблицы. Если ключ поиска больше, чем ключ таблицы, то искомый элемент таблицы должен находиться во второй половине таблицы, так как таблица отсортирована. Этот метод позволяет за счет одного сравнения изъять из области поиска сразу половину таблицы.

Двоичный поиск обязан своим названием тому, что в этом методе каждая оставшаяся часть таблицы делится пополам и ключ поиска сравнивается с ключом таблицы до тех пор, пока либо сравнение закончится успешно, либо делить станет нечего. В этом методе особенно поражает то, что уже после двух сравнений три четверти таблицы будут вне области поиска.

Вот вариант двоичного поиска, построенный на базе структуры ПОИСК:

```

LOW = 1;
HIGH = LIMIT;                                /* LIMIT = TABLE SIZE */
SEARCH_AGAIN:
  MID = (LOW + HIGH) / 2;
  IF HIGH <= LOW THEN
    CALL ARG_NOT_FOUND;
  ELSE
    IF SEARCH_ARG = TABLE(MID) THEN
      CALL PROCESS_TABLE_FUNCTION;
    ELSE
      DO;
        IF SEARCH_ARG > TABLE(MID) THEN
          LOW = MID + 1;
        ELSE
          HIGH = MID - 1;
        GO TO SEARCH_AGAIN;
      END;
    END;

```

Чтобы реализовать двоичный поиск без этой структуры, можно применить ЦИКЛ-ПОКА. Кроме того, нужен индикатор, чтобы управлять этим циклом. Например:

```

LOW = 1;
HIGH = LIMIT;
OK_TO_SEARCH = YES;
DO WHILE (OK_TO_SEARCH);
  MID = (LOW + HIGH) / 2;
  IF HIGH < LOW THEN
    CALL ARG_NOT_FOUND;
  ELSE
    IF SEARCH_ARG = TABLE(MID) THEN
      CALL PROCESS_TABLE_FUNCTION;
    ELSE
      IF SEARCH_ARG > TABLE(MID) THEN
        LOW = MID + 1;
      ELSE
        HIGH = MID - 1;
      END;
    END;
  END;

```

Заметим, что обе внутренние процедуры (ARG\_NOT\_FOUND и PROCESS\_TABLE\_FUNCTION) должны устанавливать индикатор OK\_TO\_SEARCH в положение «выключено», т. е. присваивать ему NO.

Определение структуры ПОИСК вызвано, вероятно, тем фактом, что в КОБОЛе имеется глагол ИСКАТЬ. Окажется ли эта структура жизнеспособной для ПЛ/I (т. е. улучшает ли она наглядность программ, ясность и т. д.), это нужно решать в каждой организации по-своему.

### • Некоторые приемы программирования на ПЛ/I

Структурный подход заставляет по-новому взглянуть на некоторые *типичные* приемы программирования на ПЛ/I. Это касается, в частности, конструкции ON. Ее использование может нарушить правила структурного программирования. Например:

```
ON ENDFILE(MASTER) BEGIN:
/*  "
  "
  "
GO TO EOJ;
END;
```

В этом неструктурированном примере применен оператор GOTO потому, что ситуация ENDFILE возникает *после* оператора READ или GET. Если оператор GOTO убрать, то END будет действовать как RETURN, возвращая управление оператору, следующему сразу за READ или GET, вызвавшим прерывание. Обычно следующие за READ или GET операторы обрабатывают только что прочитанную запись, но этого не требуется делать, когда обнаруживается конец файла.

В некоторых организациях разрешается использовать в конструкциях ON такой оператор GOTO, который передает управление на конец процедуры или на программу обработки ошибок. Поэтому в одних организациях приведенный выше пример будет допустимым, а в других нет. Вообще говоря, принцип возврата в точку вызова не должен нарушаться даже при описании реакций на прерывания. Вот пример структурной реализации реакции на конец файла:

```

ON ENDFILE(TAPE)
  MORE_TRANSACTIONS = NO;
  MORE_TRANSACTIONS = YES;
DO WHILE (MORE_TRANSACTIONS);
  READ FILE(TAPE) INTO(TAPE_AREA);
  IF MORE_TRANSACTIONS THEN
    DO;

    /* PROCESS RECORD */

  END;
ELSE
  DO;

  /* HANDLE END OF FILE */

END;
END;

```

Следующий пример подобен предыдущему. Однако здесь вводится индикатор с противоположным смыслом и обработка конца файла происходит *за пределами* цикла.

```

ON ENDFILE(TAPE)
  TAPE_EOF = YES;
  TAPE_EOF = NO;
DO WHILE(TAPE_EOF = NO);
  READ FILE(TAPE) INTO(TAPE_AREA);
  /* EOF TURNS INDICATOR ON */
  IF TAPE_EOF = YES THEN
    DO;

    /* PROCESS RECORD */

  END;
END;
/* HANDLE END OF FILE */

```

Наконец, приведем еще один пример, в котором имеется два оператора READ; один читает первую запись в файле, другой — все остальные:

```

ON ENDFILE(ONHAND)
  MORE_ONHAND_RECS = NO;
  MORE_ONHAND_RECS = YES;
  READ FILE(ONHAND) INTO(STOCK_RECORD);
DO WHILE(MORE_ONHAND_RECS);

  /* PROCESS RECORD */

  READ FILE(ONHAND) INTO(STOCK_RECORD);
END;
/* HANDLE END OF FILE */

```

Следующие операторы ПЛ/И лучше не использовать совсем или использовать очень ограниченно:

1. *ENTRY*: этот оператор определяет дополнительный вход в подпрограмму. Если соблюдать требование, чтобы подпрограмма была простой (один вход, один выход), то использовать его нельзя.

2. *STOP* и *EXIT*. Этих операторов следует избегать, поскольку иначе будет задано несколько выходов. Так как *STOP* и *EXIT* заканчивают выполнение всей программы, они не должны появляться нигде, кроме головного сегмента; в противном случае будет нарушено правило возврата в точку вызова. *EXIT* или *STOP* в сегменте низшего уровня допустимы только тогда, когда в таком сегменте или модуле обнаружена неисправимая ошибка и нужно немедленно закончить работу программы.

3. *RETURN*: этот оператор, вообще говоря, не нужен, так как оператор *END*, соответствующий *PROCEDURE* или *BEGIN*, играет ту же роль. Однако если в модуле имеется небольшой исполняемый фрагмент (т. е. головной сегмент), за которым следуют несколько страниц внутренних процедур, то в конце этого фрагмента можно поставить *RETURN*, чтобы подчеркнуть тот факт, что исполнение модуля кончается фактически на той же странице, где он начинается. Важно, что модуль должен иметь только *один* выход.

### Как облегчить чтение программ

Наглядность при изображении логики программы достигается за счет отступов. Читать программу легче, когда управляющие структуры расположены так, что логические связи соответствуют физическому размещению фрагментов текста.

Точных правил расположения текстов на ПЛ/И не существует. Некоторые компиляторы ПЛ/И обеспечивают автоматическое размещение исходного текста по правилам структурного программирования, если задать соответствующие указания; другие на это не способны. Однако можно дать некоторые рекомендации на этот счет, хотя они и могут быть разными в различных организациях, применяющих в настоящее время структурное программирование.

*Отступ* — это стандартный размер сдвига начала строки, который измеряется числом пробелов, оставляемых пишущей машинкой или перфоратором при нажатии кнопки табулирования или пропуска. В этой книге принят отступ, равный двум, хотя и отступ в три позиции также повышает наглядность текста.

*Начальная позиция* — это самая левая позиция, с которой могут печататься символы. Обычно это вторая позиция. (В некоторых организациях из соображений экономии места указывают, что операторы ПЛ/И могут начинаться с первой позиции.) Начальная позиция перемещается то влево, то вправо в зависимо-

сти от записываемого в строке оператора. Когда встречается IF ... THEN, IF ... THEN DO, DO или BEGIN, то позиция, с которой они располагаются, становится новой начальной позицией. END или конец условного предложения сдвигает начальную позицию обратно, т. е. влево на один отступ.

Структурное программирование уменьшает необходимость в обширных комментариях. Если комментарии появляются на отдельной строке, то они должны начинаться с самой левой позиции, в которой разрешено печатать текст. Комментарий, который появляется на той же самой строке, что и предложение ПЛ/И, должен начинаться с той колонки, которая определена правилами конкретной организации.

Так как при структурном программировании уменьшается необходимость в операторе GOTO, то метки используются не часто. В ПЛ/И они могут идентифицировать сегмент или имя процедуры. Кроме того, они используются при моделировании структур ВЫБОР и ПОИСК. Метки должны располагаться на отдельной строке, начиная с самой первой начальной позиции.

### Общие рекомендации

Приведем несколько общих рекомендаций, которые следует выполнять при программировании на ПЛ/И. Эти рекомендации могут служить базой для разработки личных правил или стандартов организации.

1. Располагайте каждое предложение на отдельной строке. Если оно в строке не помещается, продолжайте его на следующей строке с *двумя* или более отступами.

2. Самое первое предложение в блоке, в группе DO, в конструкции ON или IF нужно располагать на один отступ правее ключевого слова. Например:

<pre> DN ENDFILE(CARD)   CARD_EOF = NO;         </pre>	<pre> ON CONVERSION   BEGIN;     ONCHAR = '0!';     ERR_CT = ERR_CT + 1;   END;         </pre>	<pre> DO;   X = 1;   Y = 2; END;         </pre>
--	--	---

3. Те предложения, которых правила отступов не касаются, располагаются с той же позиции, что и предложение над ним:

```

TOTAL = SUM(PRICE_TABLE);
AVERAGE = TOTAL / NO_ELEMENTS;

```

4. Слово PROCEDURE внешней процедуры располагается с одним отступом от самой левой позиции. Например:

```
SALES;  
  PROCEDURE OPTIONS(MAIN);
```

5. Внутренние процедуры сдвигаются правее на один отступ внутри объемлющей их процедуры.

6. Слово END должно располагаться с той же позиции, что и соответствующее ему ключевое слово. Не закрывайте с помощью одного END сразу несколько блоков или групп DO. Каждое слово PROCEDURE, BEGIN и DO должно иметь свой собственный END.

7. В условном предложении первая альтернатива (но не слово THEN) записывается на отдельной строке. Ключевое слово ELSE пишется на отдельной строке, в точности под соответствующим IF. Вторая альтернатива записывается на следующей строке.

```
IF MATCHING_RECORD THEN  
  CALL UPDATE_MASTER;  
ELSE  
  CALL READ_TRANSACTION;
```

Если группа DO входит в альтернативу условного предложения, то для наглядности DO и END располагаются на отдельных строчках.

8. Все элементы данных нужно объявлять явно. И хотя отсутствие атрибутов данных допустимо в ПЛ/1, это оказывается одним из источников трудностей при чтении и отладке. Поэтому, чтобы избежать потенциальных проблем, связанных с ошибками данных, в некоторых организациях требуют в обязательном порядке объявлять все элементы данных полностью.

9. Открывайте файлы явно. В большинстве реализаций ПЛ/1 файлы открываются *автоматически* при выполнении первого обращения к ним с помощью READ или GET. Однако, когда файлы открыты и закрыты *явно*, программу будет легче читать.

```
OPEN  
  FILE (INFILE),  
  FILE (SYSPRINT),  
  FILE (MASTER);
```

Все или не все файлы открывать *одним* оператором OPEN — диктуется особенностями программы и требованиями используемой операционной системы. Например, файлы, связанные с обработкой особых ситуаций, вероятно, не должны открываться до

тех пор, пока при исполнении программы не появится в них необходимость.

10. Применяемые ключевые слова нужно расшифровывать, если текст должен служить частью выходной программной документации. Допустимы только общепринятые сокращения — PIC, DEC, CHAR, но более экзотические (такие, как STRZ) должны быть расшифрованы. Удобно применять такой способ. В собственно программе использовать обычные для ПЛ/И сокращения, а затем написать указания для препроцессора, которые и обеспечат расшифровку этих сокращений. Например:

```
%DECLARE (CHAR,DEC,DCL,PIC,PROC,VAR) CHARACTER;  
%CHAR = 'CHARACTER';  
%DEC = 'DECIMAL';  
%DCL = 'DECLARE';  
/* ETC */
```

### Пример программы

Представленный здесь пример программы на ПЛ/И относится все к той же задаче управления запасами, о которой шла речь в гл. 2, посвященной нисходящему проектированию. На рис. 8.9 представлена схема иерархии этой программы — более подробная детализация схемы, приведенной в конце гл. 2.

Текст программы на рис. 8.10 должен быть тщательно оценен с точки зрения правильности решений, принятых на самых различных уровнях. Легко ли читать этот текст? Дает ли головной сегмент общее представление о целом модуле? Говорят ли что-либо читателю имена данных? Проявляет ли логику программы расположение ее фрагментов? Можно ли ответить на такой вопрос: допускает ли эта программа, чтобы непосредственно за запросом ДОБАВИТЬ следовали запросы ОБНОВИТЬ, касающиеся только что добавленной записи?

Что еще можно сказать о легкости сопровождения и модификации? Например, можно ли без особых хлопот изменить модуль, реализующий проверку упорядоченности и правильности данных? Если да, то сколько времени потребуется затратить на подготовку полностью отлаженной новой версии программы?

Что касается соглашений о размещении текста в этой программе, то кое-кого может заинтересовать, почему, например, на рис. 8.10б точка с запятой располагается на отдельной строке. Это разумно потому, что точка с запятой отмечает конец DECLARE совершенно таким же образом, как END завершает DO. Поэтому точка с запятой располагается на отдельной строке и с той же самой позиции, что и соответствующее ключевое слово DECLARE. Кроме того, если нужно что-то добавить или убрать,



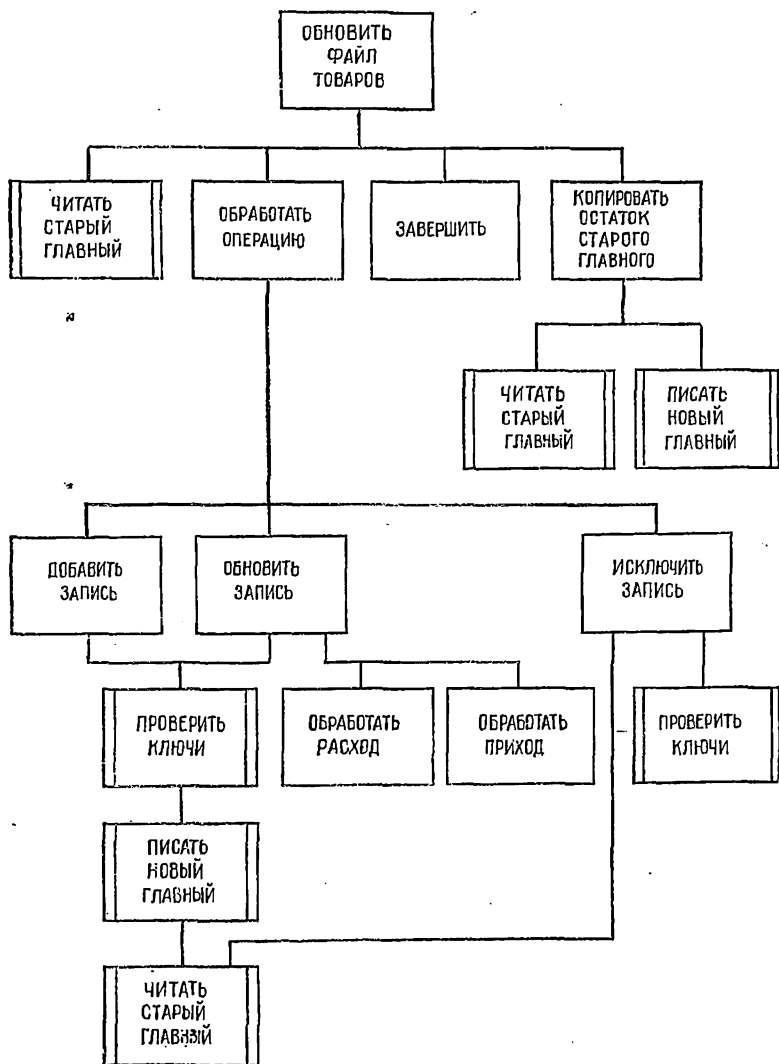


Рис. 8.9, Схема иерархии для программы обновления файла товаров.

то это можно сделать без того, чтобы перебивать какие-нибудь идентификаторы.

Обратите внимание, что на рис. 8.10в атрибут типа **LIKE** помещен сразу же за той структурой, на которую он ссылается. В некоторых организациях применение таких атрибутов не рекомендуют, ссылаясь на то, что программу становится труднее чи-

```

/* SEQUENTIAL INVENTORY FILE UPDATE PROGRAM */
UPDATE:
  PROCEDURE OPTIONS(MAIN);
/*****
/*
/* THIS PROGRAM READS A SEQUENTIAL MASTER INVENTORY FILE AND UPDATES
/* IT FROM A TRANSACTION FILE. TRANSACTIONS ARE ASSUMED TO BE IN
/* COLLATING SEQUENCE BY THE RECORD'S 'PART NUMBER.' A NEW MASTER
/* INVENTORY FILE WILL BE CREATED. EXCEPTION MESSAGES ARE PRINTED
/* FOR TRANSACTIONS WHICH ARE IN ERROR. NOTE THAT THIS PROGRAM DOES
/* NOT CHECK THE TRANSACTION FILE AS IT IS PRESUMED TO HAVE
/* BEEN CHECKED FOR ASCENDING SEQUENCE IN AN EARLIER EDIT RUN.
/*
/* THE FOLLOWING TRANSACTION CODES ARE PERMISSABLE:
/*
/* 1 ADD A NEW INVENTORY RECORD TO THE MASTER FILE
/* 2 RECEIPTS RECORD: UPDATES THE FOLLOWING FIELDS:
/*                      QUANTITY ON HAND
/*                      TOTAL RECEIPTS FOR THAT ITEM
/*                      QUANTITY STILL ON ORDER
/*                      TOTAL COST OF ITEMS ON HAND
/*                      AVERAGE COST PER ITEM
/* 3 ISSUES RECORD:   UPDATES THE FOLLOWING FIELDS:
/*                      QUANTITY ON HAND
/*                      TOTAL QUANTITY ISSUED
/*                      TOTAL COST OF ITEMS ON HAND
/*                      DATE OF LAST ISSUE
/* 4 DELETE AN ITEM FROM THE MASTER FILE
/*
*****/

```

Рис. 8.10а. Пример программы на ПЛ/1. Части б-ж следуют ниже.

тять. Это верно, если специфицировать атрибут LIKE на внутреннем уровне одной структуры и к тому же сослаться на промежуточный уровень другой структуры:

```

DCL 1 CARD_REC,
    2 KEY_FLD CHAR (16),
    2 BASIC_REC LIKE TAPE_REC. EXT 1,
    2 REST CHAR (8);

```

На рис. 8.10г показаны два сегмента: головной сегмент и сегмент ОБРАБОТАТЬ ОПЕРАЦИЮ (PROCESS TRANSACTION). Сегменты следующего уровня ДОБАВИТЬ (ADD), ОБНОВИТЬ (UPDATE), ИСКЛЮЧИТЬ (DELETE) расположены на рис. 8.10д. Каждая следующая страница листинга сообщает о программе все больше подробностей. Расположение (или порядок) отдельных сегментов на этом листинге в целом позволяет изучать программу сверху вниз (или вглубь, если иметь в виду листинг, сложенный «книжечкой»).

## Выводы

Модули должны представлять собой отдельно транслируемые подпрограммы. Сегменты по объему не должны превосходить страницы листинга (исключая DECLARE). Каждый сегмент

```

DECLARE
  EXCPTN
  FILE
  OUTPUT
  ENV ( V BLKSIZE(141))
  PRINT
  STREAM,
  NEWINV
  FILE
  OUTPUT
  RECORD
  SEQUENTIAL,
  OLDINV
  FILE
  INPUT
  RECORD
  SEQUENTIAL,
  TRANS
  FILE
  INPUT
  RECORD
  SEQUENTIAL

}
DECLARE
  1 FLAGS,
    2 YES BIT(1) INIT('1'B),
    2 NO BIT(1) INIT('0'B),
    2 NEW_RECORD_ADDED BIT(1),
    2 MORE_OLD_MASTER_RECS BIT(1),
    2 MORE_TRANSACTIONS BIT(1),
    2 TRANS_MASTER_MATCH BIT(1),
  1 TRANSACTIONS,
    2 ADD CHAR(1) INIT('1'),
    2 RECEIPT CHAR(1) INIT('2'),
    2 ISSUE CHAR(1) INIT('3'),
    2 DELETE CHAR(1) INIT('4'),
  1 COUNTERS,
    2 TRANSACTIONS_PROCESSED FIXED DEC(3)

}

```

Рис. 8.106. Пример программы на ПЛ/I.

```

DECLARE
1 NEW_MASTER_REC,
2 ACTIVITY_CODE          PIC'9',
2 KEY                    CHAR(6),
2 DESCRIPTION            CHAR(20),
2 QTY_ON_HAND            FIXED DEC(5),
2 TOTAL_ISSUES           FIXED DEC(5),
2 TOTAL_RECEIPTS         FIXED DEC(5),
2 QTY_ON_ORDER           FIXED DEC(5),
2 ORDER_POINT            FIXED DEC(3),
2 REORDER_QTY            FIXED DEC(5),
2 TOTAL_COST             FIXED DEC(7,2),
2 AVERAGE_UNIT_COST     FIXED DEC(7,3),
2 DATE_OF_LAST_ISSUE     CHAR(6) .
1 OLD_MASTER_REC LIKE NEW_MASTER_REC
;
DECLARE
TRANSACTION_RECORD      CHAR(80),
1 TRANSACTION_REC DEFINED TRANSACTION_RECORD,
2 TRANS_CODE            CHAR(1),
2 TRANS_KEY             CHAR(6),
2 TRANSACTION           CHAR(73),
1 RECEIPTS DEFINED TRANSACTION_RECORD,
2 CODE                 CHAR(1),
2 KEY                  CHAR(6),
2 QTY_RECEIVED          PIC'(5)9',
2 TOTAL_COST            PIC'(5)9V99',
1 ISSUES DEFINED TRANSACTION_RECORD,
2 CODE                 CHAR(1),
2 KEY                  CHAR(6),
2 QTY_ISSUED            PIC'(5)9',
2 CUSTOMER_NUMBER       CHAR(6),
2 DATE_ISSUED           CHAR(6),
1 ADDITION DEFINED TRANSACTION_RECORD,
2 CODE                 CHAR(1),
2 KEY                  CHAR(6),
2 DESCRIPTION           CHAR(20),
2 QTY_ON_HAND           PIC'(5)9',
2 TOTAL_ISSUES          PIC'(5)9',
2 TOTAL_RECEIPTS        PIC'(5)9',
2 QTY_ON_ORDER          PIC'(5)9',
2 ORDER_POINT           PIC'(3)9',
2 REORDER_QTY           PIC'(5)9',
2 TOTAL_COST            PIC'(5)9V99',
2 AVERAGE_UNIT_COST     PIC'(4)9V99',
2 DATE_OF_LAST_ISSUE    CHAR(6)

```

;

Рис. 8. 10в.

```

6 TOP_SEGMENT:
    ON ENDFILE(OLDINV)
      MORE_OLD_MASTER_RECS = NO;
7    ON ENDFILE(TRANS)
      MORE_TRANSACTIONS = NO;
8    OPEN
      FILE(EXCPTN),
      FILE(NEWINV),
      FILE(OLDINV),
      FILE(TRANS);

9      NEW_RECORD_ADDED = NO;
10     TRANSACTIONS_PROCESSED = 0;
11     MORE_OLD_MASTER_RECS = YES;
12     MORE_TRANSACTIONS = YES;
13     CALL READ_OLD_MASTER;
14     IF MORE_OLD_MASTER_RECS = NO THEN
        PUT FILE(EXCPTN) SKIP LIST
          ('MASTER FILE HAS NO RECORDS');

15     DO WHILE(MORE_TRANSACTIONS);
16       READ FILE(TRANS) INTO (TRANSACTION_RECORD);
17       IF MORE_TRANSACTIONS THEN
          CALL PROCESS_TRANSACTION;
18       ELSE
          IF MORE_OLD_MASTER_RECS THEN
              CALL COPY_REST_OF_OLD_MASTER;
19     END;
20     CALL WRAP_UP;
21     RETURN;

22 PROCESS_TRANSACTION:
    PROCEDURE;
23     TRANSACTIONS_PROCESSED = TRANSACTIONS_PROCESSED + 1;
24     IF TRANS_CODE = ADD THEN
        CALL ADD_RECORD;
25     ELSE IF TRANS_CODE = RECEIPT | TRANS_CODE = ISSUE THEN
        CALL UPDATE_RECORD;
26     ELSE IF TRANS_CODE = DELETE THEN
        CALL DELETE_RECORD;
27     ELSE
        PUT FILE(EXCPTN) SKIP LIST
          ('INVALID TRANSACTION CODE', TRANSACTION_REC.
            'ABOVE TRANSACTION BYPASSED');
28     END;

```

Рис. 8.10г. Пример программы.

```

29 ADD_RECORD:
    PROCEDURE:
30     CALL TEST_KEYS_FOR_MATCH;
31     IF -TRANS_MASTER_MATCH THEN
        DO;
32         NEW_MASTER_REC = ADDITION;
33         NEW_RECORD_ADDED = YES;
34     END;
35     ELSE
        PUT FILE(EXCPTN)SKIP LIST
        ('ATTEMPT TO ADD DUPLICATE RECORD',TRANSACTION_REC);
36     END;

37 UPDATE_RECORD:
    PROCEDURE:
38     CALL TEST_KEYS_FOR_MATCH;
39     IF -TRANS_MASTER_MATCH THEN
        PUT FILE(EXCPTN)SKIP LIST
        ('ATTEMPT TO UPDATE RECORD NOT IN FILE',TRANSACTION_REC);
40     ELSE
        IF TRANS_CODE = RECEIPT THEN
            CALL PROCESS_RECEIPTS;
41     ELSE
        CALL PROCESS_ISSUES;
42     END;

43 DELETE_RECORD:
    PROCEDURE:
44     CALL TEST_KEYS_FOR_MATCH;
45     IF -TRANS_MASTER_MATCH THEN
        PUT FILE(EXCPTN)SKIP LIST
        ('ATTEMPT TO DELETE RECORD NOT IN FILE',TRANSACTION_REC);
46     ELSE
        DO;
47         IF NEW_RECORD_ADDED THEN
            DO;
48             NEW_MASTER_REC = OLD_MASTER_REC;
49             NEW_RECORD_ADDED = NO;
50         END;
51         ELSE
            CALL READ_OLD_MASTER;
52     END;
53 END;

```

```

54 TEST_KEYS_FOR_MATCH:
    PROCEDURE;
55     DO WHILE (TRANS_KEY > NEW_MASTER_REC.KEY)
56     IF MORE_OLD_MASTER_RECS THEN
57         CALL WRITE_NEW_MASTER;
58     ELSE
59         DO;
60             NEW_MASTER_REC.KEY = HIGH(6);
61             IF NEW_RECDDRO_ADDED THEN
62                 # DO;
63                     WRITE FILE(NEWINV)FROM(NEW_MASTER_REC);
64                     NEW_RECORD_ADDED = NO;
65                 END;
66             END;
67             IF TRANS_KEY = NEW_MASTER_REC.KEY THEN
68                 TRANS_MASTER_MATCH = YES;
69             ELSE
70                 TRANS_MASTER_MATCH = NO;
71             ENO;

68 READ_OLD_MASTER:
    /* MASTER FILE IS READ INTO 2 WORKING STORAGE AREAS TO ALLOW A NEW
       RECORD TO BE BUILT UP IN 1 AREA WHILE THE OTHER AREA HOLDS THE */
    .NEXT MASTER.
    PROCEDURE;
69     READ FILE(OLDINV)INTO(OLD_MASTER_REC);
70     IF MORE_OLD_MASTER_RECS THEN
71         NEW_MASTER_REC = OLD_MASTER_REC;
72     END;

72 WRITE_NEW_MASTER:
    PROCEDURE;
73     WRITE FILE(NEWINV)FROM(NEW_MASTER_REC);
74     IF NEW_RECORD_ADDED THEN
75         DO;
76             NEW_MASTER_REC = OLD_MASTER_REC;
77             NEW_RECORD_ADDED = NO;
78         END;
79     ELSE
80         CALL READ_OLD_MASTER;
81     END;

```

Рис. 8,10е. Пример программы на ПЛ/1.

```

80  PROCESS_RECEIPTS:
    PROCEDURE;
81      NEW_MASTER_REC.QTY_ON_HAND =
        NEW_MASTER_REC.QTY_ON_HAND + QTY_RECEIVED;
82      NEW_MASTER_REC.TOTAL_RECEIPTS =
        NEW_MASTER_REC.TOTAL_RECEIPTS + QTY_RECEIVED;
83      NEW_MASTER_REC.QTY_ON_ORDER =
        NEW_MASTER_REC.QTY_ON_ORDER - QTY_RECEIVED;
84      NEW_MASTER_REC.TOTAL_COST =
        NEW_MASTER_REC.TOTAL_COST + RECEIPTS.TOTAL_COST;
85      NEW_MASTER_REC.AVERAGE_UNIT_COST =
        NEW_MASTER_REC.TOTAL_COST / NEW_MASTER_REC.QTY_ON_HAND;
86  END;

87  PROCESS_ISSUES:
    PROCEDURE;
88      NEW_MASTER_REC.QTY_ON_HAND =
        NEW_MASTER_REC.QTY_ON_HAND - QTY_ISSUED;
89      NEW_MASTER_REC.TOTAL_ISSUES =
        NEW_MASTER_REC.TOTAL_ISSUES + QTY_ISSUED;
90      NEW_MASTER_REC.TOTAL_COST = NEW_MASTER_REC.TOTAL_COST -
        (QTY_ISSUED * NEW_MASTER_REC.AVERAGE_UNIT_COST);
91      NEW_MASTER_REC.DATE_OF_LAST_ISSUE = DATE_ISSUED;
92  END;

93  COPY_REST_OF_OLD_MASTER:
    PROCEDURE;
94      CALL WRITE_NEW_MASTER;
95      DO WHILE(MORE_OLD_MASTER_RECS);
96          WRITE FILE(NEWINV)FROM(NEW_MASTER_REC);
97          READ FILE(OLDINV)INTO(OLD_MASTER_REC);
98          NEW_MASTER_REC = OLD_MASTER_REC;
99      END;
100  END;

101  WRAP_UP:
    PROCEDURE;
102      IF NEW_RECORD_ADDED THEN
103          WRITE FILE(NEWINV)FROM(NEW_MASTER_REC);
104          PUT FILE(EXCPTN)SKIP(3)LIST
            ('NUMBER OF TRANSACTIONS PROCESSED:',
             TRANSACTIONS_PROCESSED);
105      CLOSE
106          FILE(EXCPTN),
          FILE(NEWINV),
          FILE(OLDINV),
          FILE(TRANS);
107  END;
108  END;
/* END OF PROGRAM */

```

Рис. 8.10ж.



должен быть простой программой. Перед сегментами с достаточно сложной логикой нужно располагать комментарии. В общем случае передавать управление сегменту следует с помощью CALL (или указателя функции) или можно средствами препроцессора вставлять сегмент непосредственно внутрь текста.

Для вложенных условных предложений глубина вложенности должна быть ограничена: все предложение должно уместиться на одной странице. Повторения нужно реализовывать посредством DO (с WHILE или без него). ВЫБОР обычно следует реализовывать оператором GOTO с массивом меток или CALL с массивом входов.

Используйте пустые строки, чтобы выделить логически связанные группы предложений. Включайте комментарии в текст только тогда, когда это необходимо. Используйте mnemonic имена данных. Проявляйте логику программы с помощью отступов внутри процедур, блоков и групп DO. Программу будет легче читать и изменять. Каждому ключевому слову PROCEDURE, BEGIN и DO должен соответствовать собственный END.

Используйте каждый индикатор только для одной-единственной цели. Вводите имена для констант и присваивайте им начальные значения в предложении DECLARE. Переменным и индикаторам присваивайте начальные значения в исполняемой части процедуры.

### Контрольные вопросы и упражнения

1. Стали бы вы использовать оператор GOTO для передачи управления внутри и из сегмента? Почему?

2. Каким рекомендациям нужно следовать при использовании индикаторов? Расскажите о назначениях, названиях и инициализации индикаторов.

3. Напишите набор требований, при выполнении которых, как вам кажется, вложенные условные предложения было бы нетрудно читать и воспринимать.

4. Какие операторы и ключевые слова ПЛ/И нужно применять с большой осторожностью?

5. Сформулируйте требования к программированию модулей.

6. Модифицируйте программу на рис. 8.10 так, чтобы она проверяла упорядоченность файла запросов.

7. Модифицируйте программу на рис. 8.10 так, чтобы она обрабатывала два входных файла запросов. Один файл содержит добавления и исключения, а другой обновляемые записи (приходы и расходы). Оба файла отсортированы в возрастающей последовательности.

8. Предположим, что в организации, где вы работаете, не допускаются модули объемом более 50 предложений на языке ПЛ/И (кроме DECLARE). Перепишите программу на рис. 8.10 так, чтобы она удовлетворяла этому требованию. Начните с исправления схемы иерархии,

## Сквозной структурный контроль

*«Если я видел дальше, то потому, что стоял на плечах гигантов».*

Исаак Ньютон

Классическое средство, применяемое руководством для наблюдения за продвижением проекта в области обработки данных, — это *экспертная оценка проекта*. Такие оценки делаются регулярно на протяжении всей работы над проектом, чтобы обнаружить опасности раньше, чем они смогут достигнуть угрожающих размеров, и пока еще остается время на их устранение. С точки зрения руководства, при этом следует беспокоиться прежде всего о том, как проект укладывается в график, нужны ли дополнительные ресурсы, уложится ли проект в рамки запланированного бюджета. Естественно, руководство заинтересовано также в том, чтобы завершённый проект отвечал целям, ради достижения которых велась работа.

Сквозной структурный контроль — это новое средство, которое в конечном итоге очень облегчает управление проектом, но является по существу не управленческим, а *техническим*, используемым людьми, непосредственно занятыми в проекте. Термин *структурный* подчеркивает то обстоятельство, что этот контроль становится естественной составной частью рабочего цикла и проводится заранее предопределённым и продуманным во всех тонкостях способом. Термин *сквозной* указывает на способ проверки — все контролируемые элементы мысленно выполняются шаг за шагом. Сквозной контроль придуман для того, чтобы обнаруживать ошибки в принятых решениях и при этом создать такую атмосферу, при которой каждый, и особенно сам проектировщик, стремился бы найти эти ошибки как можно раньше.

На рис. 9.1 показаны *этапы* разработки. Площадь под кривой представляет затраченные средства (т. е. люди, финансы и т. п.) для проекта в области систем обработки данных. Сквозной контроль осуществляется на всех этапах разработки — начинается как можно раньше и продолжается до самого этапа реализации включительно. Частые контрольные сессии, на которых рассматривается относительно небольшая порция материала, предпоч-

тительнее длинных, но редких встреч. Преимущество частых сессий в том, что легче управлять объемом контролируемого материала и большинство недостатков ликвидируется раньше — в результате уменьшается опасность того, что должна будет переделываться уже почти завершенная работа.

В табл. 9.1 показаны те элементы, которые могут проверяться при использовании структурного контроля на различных стадиях разработки.

Одна из первых сессий, которая должна быть организована еще на *этапе определения требований*, — это проверка исходных спецификаций будущей системы. На этой сессии нужно проверить

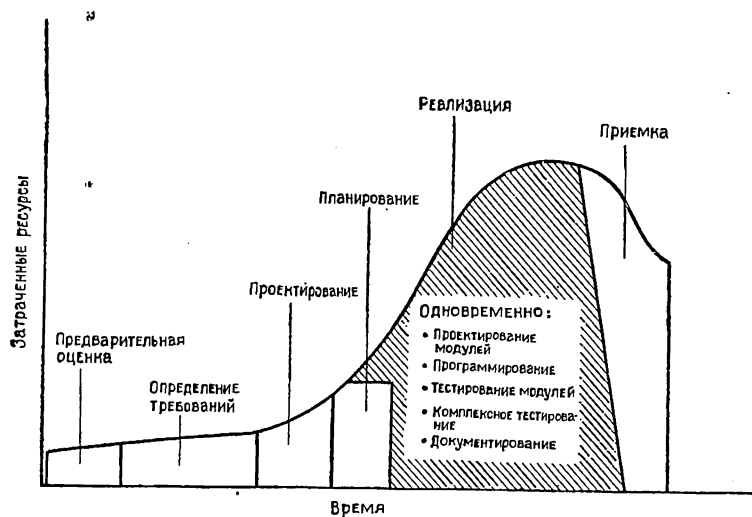


Рис. 9.1. Поэтапная разработка (определения этапов см. в глоссарии).

полноту спецификаций, а также убедиться в том, что они поняты теми проектировщиками, которые будут работать над этим проектом. На *этапе проектирования* следует проверять, как система будет работать на конкретных входных данных, чтобы убедиться, что это согласуется со спецификациями. На *этапе планирования* контролируется запланированный порядок разработки модулей и план тестирования. Особенно внимательно нужно проверять критерии отбора подлежащих тестированию ситуаций и соответствующие тесты, чтобы убедиться в соответствии подготовленных тестов заранее составленному плану тестирования. На *этапе реализации* проверяется во всех подробностях устройство программы, взаимодействие и наличие модулей, а также документация. Наконец, на *этапе приемки* готовый продукт снова полностью проверяется.

Таблица 9.1

## Элементы, которые должны проверяться на контрольных сессиях

Этапы проекта	Что проверяется	Наиболее вероятные недостатки
Определение требований	Планы проекта Сроки Сфера применения проекта	Различного рода упущения Нереальные сроки Недоступность основных пользователей
Начало проектирования	Спецификация системы Постановка задачи	Неполные спецификации Нечеткая постановка
Проектирование	Функциональные спецификации программы Проектирование программы (схемы иерархии) Обработка особых случаев Проверки	Неполные спецификации Передача управления не по вертикали, плохо определены функции Недостаточно детально проработана Неполное соответствие требованиям
Планирование	План порядка разработки модулей План порядка тестирования модулей	Не полностью проявлена зависимость по данным Неполное соответствие требованиям
Реализация	Тесты Детальное проектирование программы Взаимодействие модулей Документация (для пользователя) Псевдокод или собственно программа	Недостаточно полный набор Отклонения от общего проекта Незафиксированные предположения Неясности, ненужные подробности Логические ошибки
Приемка	Окончательная документация Окончательный продукт (соответствие результатов спецификациям)	Не установлен критерий пригодности документации Изменения в спецификациях

## Кто этим занимается?

Создатель подлежащего контролю материала сам подбирает контролеров. (Он также сам назначает дату и время каждой сессии.) Администратор <sup>1)</sup> обычно не приглашается, так как одно из основных условий эффективности контрольных сессий — открытая и доброжелательная атмосфера. Структурный контроль не должен стать средством оценки исполнителя, а присутствие администратора может сделать его таковым. Иногда возникают

<sup>1)</sup> Администратор здесь — это лицо, в ведении которого находится зарплата, продвижение и увольнение сотрудников.

такие взаимоотношения администратора и сотрудника, что последний не ощущает неудобств от того, что его ошибки становятся известны администратору. Но даже при полном взаимном уважении администратор обычно не может избавиться от непроизвольной оценки сотрудника или сотрудник чувствует себя неудобно, если его ошибки или упущения становятся известными администратору.

Наоборот, руководитель программистской бригады или всего проекта будет идеальным кандидатом на роль контролера, так как такой руководитель отвечает за техническую сторону проекта. Более того, желательно, чтобы руководитель бригады участвовал почти в каждой контрольной сессии. Ведь руководитель бригады — один из самых квалифицированных экспертов по внутренним проблемам проекта. Он, конечно, обсудит с администратором и ход работ, и возникшие проблемы, в том числе и связанные с теми исполнителями, чью работу нужно улучшить. Таким образом, администратор будет в курсе всех дел и без того, чтобы лично присутствовать на контрольной сессии, где обсуждаются сугубо технические вопросы.

Более других заинтересован в выявлении неясных вопросов и ошибок такой контролер, который непосредственно заинтересован в успехе анализируемого проекта. Следовательно, среди контролеров в бригадных проектах должны быть члены бригады. Если имеется специальная группа, занимающаяся тестированием, то члены этой группы также должны быть заинтересованы в том, чтобы устранить ошибки, так как их дело — официально подтвердить готовность программного продукта. Очень важно, чтобы члены тестовой группы участвовали в контрольных сессиях уже на самых ранних стадиях проекта.

Кроме этого, обычно бывает так, что в каждой организации имеется несколько сотрудников, которые славятся своей высокой технической компетентностью и/или способностью быстро находить ошибки. Как правило, эти люди гордятся такой репутацией и обычно готовы помочь любому, кто придет к ним за помощью. Разработчик должен знать этих людей и обязательно пригласить одного или нескольких из них.

На ранних стадиях проекта определяются требования к системе в целом. В этот период очень важно, чтобы на контрольных сессиях присутствовал пользователь конечного продукта. Сессии не исключают других встреч и контактов с пользователем; это дополнительное средство улучшения взаимопонимания между группой программистов-разработчиков и представителями организации-пользователя. При таком подходе могут быть подготовлены более удачные спецификации. У пользователя возникает более высокий уровень доверия к качеству разработки, чем в том случае, когда с ним обращаются по принципу «объясни

ему, что ему нужно» — ведь теперь пользователь сам принимает участие в выработке требований к будущей системе. Поэтому раннее привлечение пользователя к контрольным сессиям уменьшает вероятность того, что пользователь будет недоволен готовой системой или потребует крупных переделок после ее завершения.

Кроме того, на самых ранних стадиях проекта разработчик должен пригласить на контрольные сессии людей, которые будут обеспечивать *функционирование* системы. Возникающие в связи с функционированием специфические требования лучше всего знакомы именно этому персоналу, поэтому его представители будут самыми квалифицированными контролерами разработки с операционной точки зрения.

В контрольных сессиях обычно участвуют от трех до шести человек. В небольших организациях может оказаться невозможным собрать много людей. (В шутку было предложено, чтобы «кустарь-одиночка» проводил контрольную сессию перед зеркалом.) В небольших организациях в сессиях могут участвовать и два специалиста, если их всего два в подразделении. Фактически в маленькой фирме президент или вице-президент часто непосредственно занимаются проблемами обработки данных. В этом случае один из них мог бы участвовать в контрольных сессиях.

Руководитель бригады, члены бригады, группа тестирования, представители операторов, представители пользователя и особо квалифицированные эксперты — это все кандидаты в контролеры. Из них разработчик должен выбрать такой состав, который обеспечит разнообразие точек зрения и мнений, чтобы гарантировать высокое качество контроля.

### Атмосфера на контрольной сессии

Предположим, что вы участвуете в сессии в роли автора обсуждаемого материала. Если вы настроены на спокойное восприятие критики, а не на защиту любой ценой, вы облегчите вашим оппонентам их задачу — помочь вам создать лучший продукт. Если сами контролеры чувствуют себя призванными решать проблемы, а не создавать трудности, то сессия будет полезной и плодотворной. Что можно сделать, чтобы способствовать созданию такой атмосферы? Нужно руководствоваться как минимум следующими пятью правилами:

1. *Контролировать работу каждого, независимо от старшинства.* При таком подходе нет повода придавать особое значение тому, чья именно работа контролируется. (Если планируется контролировать только некоторые работы, то их авторы могут расценить это как недоверие к ним и начать доказывать необоснованность такого недоверия вместо того, чтобы думать о насущ-

ных задачах и решать именно их.) Если вы — руководитель бригады, то сделайте так, чтобы вначале проверялась ваша работа. Ваша готовность подаст пример другим.

2. *Помните, что вас не будут оценивать по числу или серьезности ошибок, найденных ВО ВРЕМЯ сессии.* Представьте, что вас попросили выполнить некоторый вариант квалификационного теста. При этом составитель теста совершенно не стремится оценить ваши знания, его задача — убедиться в пригодности теста, поэтому его интересуют только те ошибки, которые допускаются при исполнении этого теста. По окончании работы вы должны сдать результаты *без* указания своего имени. После этого вам сообщают правильные ответы с тем, чтобы вы могли разобраться в своих ошибках. При этом никому другому не интересно, как выполнили этот тест *именно вы*. Как вы вообще относитесь к выполнению такого рода тестов — чувствуете себя спокойно или испытываете напряжение, рассматриваете ли вы это как возможность личного развития или просто как возможность узнать что-то новое? Во время контрольной сессии вы должны забыть, что рассматривается именно *ваша* работа, все, что вам нужно сделать — это убедиться в ее пригодности.

3. *Помните, что вас будут оценивать по числу или серьезности ошибок, найденных ПОСЛЕ того, как завершились все сессии.* В некоторых организациях ведут учет таких ошибок, и, если их окажется слишком много, автор может быть привлечен к ответственности. И если не будет найдено удовлетворительного объяснения (сложность проблемы, новизна применения), то замеченные ошибки могут повлиять на вашу профессиональную репутацию. И поэтому, если вы дорожите своей репутацией, вы будете относиться к контрольной сессии весьма серьезно.

4. *Контролеры должны заниматься выявлением ошибок, а не их исправлением.* Когда ошибка обнаружена, то предполагается, что вы исправите ее позже. Контролеры не должны делать конкретных изменений в вашей работе. Важно не обсуждать, какая именно из нескольких альтернатив является лучшим решением. Единственная цель контролеров — помочь вам найти ошибки, а вы сами должны решить, как лучше их исправить.

5. *Нужно заниматься основными проблемами, а не мелкими ошибками.* Раннее обнаружение основных трудностей сохраняет время и силы для их последующего преодоления. Тривиальные ошибки, например синтаксические, и без контрольных сессий обнаруживаются достаточно быстро и легко исправляются.

### Как это все происходит?

Когда вы считаете, что готовы к контрольной сессии, выберите подходящее время и пригласите ваших контролеров. Однако это вовсе не означает, что сессию можно начинать без специальной

подготовки. Еще в самом начале работы вами совместно с администратором должен был быть составлен общий план, определяющий даты конкретных контрольных сессий. Старайтесь, чтобы эти даты были как можно реальнее.

После того как контролеры приглашены и ясно, что они могут присутствовать, раздайте им копии контролируемых материалов за несколько дней до начала сессии. Это могут быть описания спецификаций, блок-схемы, проекты форм документов, описания записей и файлов, структура базы данных, схемы иерархии, тексты на псевдокоде или даже листинги программ. Другими словами, большая часть контролируемого материала позднее станет документацией по данному проекту. Изучая материал к сессии, контролеры могут заметить мелкие или типографские ошибки в этих документах, которые вернутся к вам после сессии. Таким путем не только контролируется сама работа, но и отшлифовывается документация до полной завершенности, точности и ясности. Поэтому сквозной контроль создает условия для своевременной выработки документации на тех стадиях разработки, когда это может быть сделано наилучшим образом.

На сессии один из контролеров выступает в роли секретаря и отмечает все возникающие спорные вопросы. Составляется *список обнаруженных дефектов* (ошибок) и раздается вам и всем контролерам. Копия не посылается администратору, но должна быть послана руководителю бригады, если он не мог присутствовать на сессии.

Другой контролер выступает в роли *председателя* — его дело способствовать созданию благоприятной рабочей атмосферы. Председатель должен быть очень внимательным к множеству тонкостей — к характеру, темпу представления материала и т. п. Умелый председатель знает, когда кончать дискуссию и как ее вести так, чтобы никого не раздражать. Он не должен позволить сессии переродиться в *спор* или попытку найти лучшие решения.

Сессия должна начинаться с определения ее *целей*, которые должны быть *проверяемыми*. Например, «цель этого совещания — *прогнать* данные через программные модули и *найти* логические ошибки или ошибки в сопряжениях». Сравните это с предложением: «цель этого совещания — понять, как взаимодействуют программные модули». (Как узнать, что контролер *понял*?)

Контролеры приходят на сессию со своими замечаниями. Если есть какие-то вопросы или проблемы, касающиеся документов, то они получают приоритет и обсуждаются в самом начале сессии. Руководствуясь этими замечаниями, вы даете краткое пояснение, если это необходимо. (Если контролеры не знакомы с проектом в целом, то можно рассказать и обо всей системе. Ваше пояснение должно затрагивать только концептуальный уровень и содержать только основную информацию, необходимую для пользы дела.)



Теперь следует самая главная часть сессии — тот период, когда работа изучается подробно. Вы объясняете, для чего предназначена каждая деталь вашего проекта или программы, почему она необходима и как она будет работать. И так шаг за шагом. Вероятно, у вас были случаи, когда казалось невозможным найти ошибку в программе и вы обращались к кому-то за помощью. Для того чтобы объяснить, что делает ваша программа и в чем суть проблемы, вы начинали с самого начала и «шли» по программе, описывая каждую ее часть. Затем внезапно вы сами находили ошибку, благодарили слушателя и возвращались к своему столу. Эта ошибка была найдена в процессе объяснения программы без единого слова со стороны вашего помощника. В точности такая же ситуация часто возникает на контрольных сессиях. Механика контроля такова, что проявляются ошибки, которые зачастую вы обнаруживаете первыми. Если же это не так, то обычно это делает один из контролеров.

Один из наиболее эффективных способов находить ошибки — «прокрутить» программу с конкретными данными. Примеры таких данных можно выбрать среди тестов, которые будут использоваться при проверке на машине, когда программа заработает. Программа мысленно прокручивается вами, а контролеры по ходу вашего рассказа задают вопросы. Особенно популярны вопросы типа «А что, если...?» Атмосфера такая, как будто группа изучает решение задачи, а не должна подтвердить, что оно правильно. Когда выявляются спорные вопросы, требующие особого внимания, они фиксируются секретарем перед тем, как председатель предложит группе перейти к другому вопросу. Ваши контролеры должны искать и упущения, и неправильные распределения функций, и отклонения от спецификаций. Они должны искать возможные ошибки и в сопряжениях, и в обработке исключительных ситуаций.

Должен ли контролер указывать на *значительную* неэффективность вашей программы? Имеются две школы, которые по-разному отвечают на этот вопрос: одна считает самым важным установить, что программа будет работать правильно, т. е. будет решать поставленную задачу. Тратить время на поиски лучшего способа сделать то же самое (особенно если его правильность также нужно будет проверять) может оказаться чересчур расточительным. Другая точка зрения состоит в том, что такие поиски — хорошая возможность для каждого присутствующего научиться лучшим методам. Советы такого рода могут быть полезными, если пользоваться ими разумно. Вероятно, самое лучшее — это компромисс, учитывающий выделенное выше слово «значительную». Если экономия может быть существенной, то разговор о стиле программы и т. п. может оказаться полезным. Однако председатель должен ограничить дискуссию, возникающую по

поводу незначительной экономии места или времени, а также в сомнительных случаях.

Если контролеры отмечают мелкие ошибки или недовольны стилем документов, то председатель может попросить их *письменно* изложить свои замечания. Это позволяет не отвлекать внимание участников сессии, улучшает психологическую атмосферу и сохраняет для вас всю информацию о сути дела. Если список, составленный секретарем, по количеству или серьезности ошибок выходит за заранее установленные рамки, то может быть запланирована повторная сессия. В противном случае вы просто после сессии исправляете свой материал.

Формально никаких мер контроля за фактическим внесением изменений нет, однако вы можете захотеть переслать вашим контролерам отчет о сделанных исправлениях, чтобы они знали о текущем состоянии дел. Это тем более необходимо, если их работа связана с вашей.

В некоторых организациях накапливается общий список ошибок всех разработчиков. Такой список служит двум целям:

1. Он помогает выявлять *типичные* трудности. На его основе могут быть составлены методические пособия для повышения квалификации разработчиков в конкретных областях.

2. Он может служить руководством для будущих контролеров. Зная типичные ошибки, они смогут лучше управлять своим вниманием.

Контрольная сессия должна занимать приблизительно полтора-два часа. За меньшее время невозможно глубоко вникнуть в суть материала, а более двух часов не смогут выделить те контролеры, которым трудно распоряжаться своим временем.

Рис. 9.2 иллюстрирует работу контрольной сессии.

### Выводы для руководства

Сквозной структурный контроль может в значительной степени способствовать успеху проекта. Однако, чтобы обеспечить этот успех, необходимо суметь избежать нескольких ловушек:

1. *Использования результатов структурного контроля как средства оценки сотрудника.* Если контролеры и разработчики думают, что администрация использует сессии как средство оценки, то они могут подсознательно надеяться на взаимную поддержку и не стремиться находить ошибки в работе друг друга.

2. *Выделения недостаточного времени для структурного контроля.* Сквозной контроль отнимает много времени, особенно в начальной стадии разработки. Это время должно быть явно включено как в общий план, так и в индивидуальный план каж-

дого разработчика. Оценивая индивидуальные затраты на структурный контроль, нужно исходить из предположения, что разработчику необходимо от 2 до 4 часов, чтобы подготовиться к сессии, занимающей 2 часа. Кроме того, контролер тратит время не только тогда, когда присутствует на сессии, но и предварительно знакомясь с материалом. В некоторых проектах, следовательно, контрольные сессии могут потребовать многих человеко-недель.

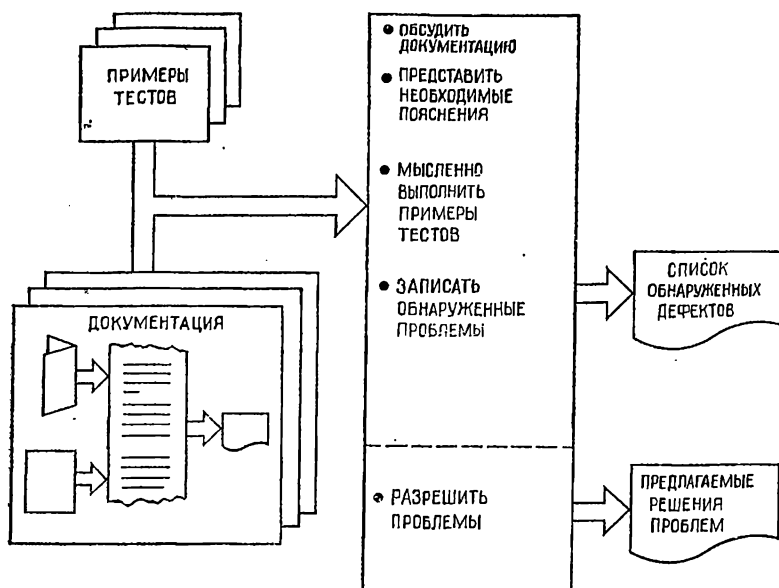


Рис. 9.2. Схема работы контрольной сессии.

Бесполезная трата времени? Вовсе нет! В проектах, использующих сквозной структурный контроль (и другие элементы структурной технологии программирования), программисты не только укладывались в запланированные сроки завершения работы, но и избегали переделок в проекте на его заключительной стадии, а это типичное явление при традиционном подходе к проектированию. Участие в контрольных сессиях должно быть явно запланированной частью работы каждого сотрудника. Важно, чтобы разработчики не относились к контролю как к внеплановой трате своего времени. Если контрольная сессия сокращается, чтобы сэкономить время, то обнаруживаются только простые, очевидные ошибки.

3. Излишнего беспокойства о том, что контрольные сессии помогают плохому разработчику, несостоятельность которого иначе проявилась бы раньше. Если контролеры находят за сотрудника

ошибки, то делает ли он вообще свое дело или бригада делает за него всю работу? На самом деле плохому сотруднику сквозной структурный контроль дает не так уж много, так как именно он должен внести все изменения.

Сквозной структурный контроль позволяет обнаруживать ошибки проектирования и логические ошибки на ранних стадиях разработки. Сессии становятся важными событиями, существенно повышающими продуктивность.

Задача контролеров состоит в том, чтобы хорошо подготовиться, глубоко исследовать и четко сформулировать проблемы. Если контролеры — члены одной бригады, то их общая забота — проявить скрытые связи между тем, что они сами разрабатывают, и тем, что сейчас контролируется. В результате станут явными двусмысленности, потребуются новые уточнения и разъяснения. И наконец, администратор должен рассматривать сквозной структурный контроль просто как эффективное средство общения разработчиков.

Сквозной контроль может заставить и разработчика, и контролера стремиться повышать уровень своей квалификации. Естественно пытаться еще до сессии сделать работу получше, в надежде, что в таком случае в ней будет обнаружено меньше ошибок. Таким образом, контроль способствует улучшению качества программы даже тогда, когда во время сессий найдено мало ошибок или их вовсе не обнаружено. Контролеры обычно испытывают удовлетворение, сознавая свою роль в улучшении продукта. А автор контролируемых материалов видит пользу такой сессии в том, что пройдена еще одна важная веха на пути к завершению проекта.

Для вас как администратора контрольная сессия — удобный показатель продвижения проекта. Знать, что проведена контрольная сессия — это намного лучше, чем просто спрашивать разработчика о состоянии его дел.

Сквозной структурный контроль должен комбинироваться с другими методами, упоминавшимися ранее. Так, при проектировании сверху вниз нужно контролировать правильность выделения модулей верхнего уровня, прежде чем разрабатывать модули более низкого уровня. Простота и ясность программ достигаются легче, если сквозной структурный контроль применять совместно со структурным программированием. В этом случае контроль служит и для того, чтобы удостовериться в соблюдении основных требований структурного программирования.

Необходимо, чтобы ваши сотрудники знали, какое значение вы придаете сквозному контролю, специально планируя время для контрольных сессий. Для успеха сессий очень важно благоприятное отношение к ним со стороны всех участников, го-

товность к открытому обсуждению, а не к борьбе, и вера в то, что вы не собираетесь оценивать их на основании того, что происходит *во время* сессии. Может потребоваться несколько контрольных сессий, прежде чем люди поверят в их пользу и усвоят истинные цели сквозного контроля. По этой причине первые сессии могут оказаться менее успешными, чем последующие.

### Выводы

Опыт использования сквозного структурного контроля очень обнадеживает. Несомненно, существует много способов модифицировать его для применения к другим типам проектов. Применение структурного контроля позволяет достичь сразу всех (или по крайней мере некоторых) из указанных ниже результатов:

1. Появляется возможность обнаружить ошибки в системе как можно раньше, когда цена их исправления минимальна.
2. Улучшается общее качество продукта.
3. У сотрудников чаще возникает стимул повышать уровень своей квалификации.
4. Разработчики приобретают большую уверенность в высоком качестве своей работы.
5. Разработчики получают большее удовлетворение от своей работы.
6. Улучшается психологическая атмосфера в бригаде.
7. Молодые сотрудники получают удобную возможность обучиться новым методам работы.
8. Опытные сотрудники часто еще более повышают уровень своего мастерства.
9. Новые участники проекта с помощью контрольных сессий могут обучиться быстрее и лучше, чем с помощью других методов.
10. Руководству легче следить за продвижением проекта и оценивать его завершенность.

### Контрольные вопросы

1. Назовите хотя бы два отличия сквозного структурного контроля от обычной экспертной оценки проекта.
2. Назовите как минимум два отличия структурного контроля от обращения к друзьям за помощью в случае, когда вы сталкиваетесь с какой-либо трудностью.
3. Предположим, что вы — разработчик и готовитесь к контрольной сессии. Перечислите категории людей, которых бы вы пригласили и в каждом случае укажите причину.
4. Предположим, что вы — разработчик и готовитесь к первой контрольной сессии. Назовите, что могут сделать руководитель, коллеги и вы сами, чтобы вы были открытым и доброжелательным во время сессии.

5. Что бы вы рекомендовали в качестве процедуры, гарантирующей исправление выявленных на сессии ошибок.

6. Спланируйте контрольную сессию. Укажите то, что должно быть выполнено до сессии, каковы должны быть демонстрируемые материалы, как следует распределить два часа отведенного времени.

7. Укажите два довода за и два против присутствия администратора на сессии.

8. Назовите по меньшей мере три возможных полезных применения списка дефектов, обнаруженных во время сессии,

## Тестирование

*“Неудовлетворенность качеством своей работы вызывает горечь, заглушающую сладостное чувство от сознания того, что она выполнена в срок”.<sup>1)</sup>*

*Тестирование* — это процесс, посредством которого проверяется правильность программы. Эта деятельность носит *позитивный* характер, ее цель — показать, что программа правильна и действительно работает в соответствии с проектными спецификациями. *Отладка* — процесс исправления ошибок в программе. Эта деятельность носит *негативный* характер в том смысле, что она полностью направлена на избавление от обнаруженных ошибок. Тестирование закончено, когда осуществлены все требуемые проверки на соответствие спецификациям. Отладка прекращается, когда исправлены все известные ошибки. Однако отладка — это процесс, который прекращается только временно, так как последующие прогоны программы могут выявить другие ошибки, тогда процесс отладки возобновляется (вспомним слова Марка Твена о том, как бросить курить: «Это легко; я делал это много раз»).

Так как отладка может многократно возобновляться, то ее в отличие от тестирования нельзя заранее планировать. Лучшее, что может быть сделано, — это прикинуть, как именно отлаживать, и перечислить «что искать». А вот тестирование может и должно планироваться. Эту работу можно очертить достаточно ясно, указав, как и что именно следует проверять. Тестирование может быть спланировано так, чтобы быть выполненным за вполне определенный период времени.

Отладка не может начаться, пока не кончилась разработка, так как для нее нужна готовая к исполнению программа. А тестирование можно начинать на самых ранних стадиях разработки. Конечно, тесты сами по себе должны прогоняться ближе к концу проекта, но решения о том, что тестировать, как тестировать и с какими данными, могут и должны быть приняты еще до начала программирования.

---

<sup>1)</sup> C. Weagle, “Visit to a Programming Center”, THINK, October/November 1974, p. 31. Reprinted by permission from THINK. Copyright 1974 by International Business Machines Corporation.

## Что и как нужно тестировать?

Проверять следует и спецификации, и конкретные модули и межмодульные связи. Об этом и пойдет речь в этом разделе.

### Тестирование спецификаций

Хорошо определенные спецификации образуют каркас, на основе которого планируется и осуществляется *тестирование* программы. Спецификации должны быть определены достаточно четко, чтобы можно было удостовериться, что программа им действительно удовлетворяет. Поэтому прежде всего нужно проверить на полноту и ясность сами *спецификации*. Один из подходов к анализу спецификаций состоит в использовании так называемого метода *ситуация-эффект*. Под ситуацией здесь понимается *условие* или комбинация условий, которые могут возникнуть. А *эффект* — это есть видимый результат или действие, которое должно быть предпринято (например, изменить запись, напечатать сообщение). Прекрасно, если спецификации записаны в такой форме, но если нет, то необходимо встретиться с пользователем (и проектировщиком, если пользователь не может написать такие спецификации). В результате этой встречи (которая может быть, а может и не быть организована в форме контрольной сессии) должен быть составлен список соответствий ситуация-эффект, описывающий требования решаемой прикладной задачи. Пример такого списка приведен в табл. 10.1.

Таблица 10.1

Спецификации программы в терминах соответствия ситуация-эффект

Условие (ситуация)	Действие (эффект)
1. Операции не упорядочены	Напечатать сообщение об ошибке и окончить работу
2. Неправильный код операции	Напечатать сообщение об ошибке и игнорировать операцию
3. Для операции ДОБАВИТЬ уже существует запись в основном файле	Напечатать сообщение об ошибке (попытке повторно записать) и игнорировать операцию
4. Операция обновления следует за операцией ДОБАВИТЬ, при- чем обновить требуется именно ту запись, для которой выпол- нено условие 3	Напечатать сообщение об ошибке и игнорировать <i>все</i> операции обнов- ления, относящиеся к этой записи, для которой была проигнорирована операция ДОБАВИТЬ
5. И т. д.	



Если обнаруживается ситуация, для которой не указан эффект, то фиксируется упущение в спецификациях. Может оказаться и так, что указан эффект, а условие его возникновения отсутствует. Рассмотрим, например, следующий фрагмент спецификации для программы обновления файла запасов.

*Операции образуют упорядоченную числовую последовательность и могут содержать добавления, изменения или исключения записей главного файла. Каждая операция должна быть отредактирована согласно определенному критерию, и должны выдаваться соответствующие сообщения об ошибках. Добавления, касающиеся существующей записи, а также обновления и исключения, для которых нет главной записи, должны приводить к печати диагностических сообщений.*

Что нужно делать, если операции не упорядочены? Должна ли программа проверять это? Если это обнаружено, то нужно ли кончать работу? Или текущая операция игнорируется? Что делать с предыдущей операцией? Все это требует прояснения до того, как разработка будет продолжена.

Рассмотрим далее комбинации условий, которые связаны «И» или «ИЛИ». Например, каков должен быть эффект, если операция добавления обладает ошибками в формате и касается существующей записи? Должны ли быть указаны обе ошибки? Если одна, то какая? Ответы на такие вопросы могут существенно повлиять на проектирование и реализацию программы. Поэтому, если возможно, должны быть рассмотрены все комбинации условий. Однако в некоторых программах имеется так много возможных комбинаций условий, что проверять их все непрактично. Тогда нужно выделить наиболее важные комбинации и те, которые возникнут с наибольшей вероятностью. Некоторые комбинации, конечно, логически невозможны. Например, *условие окончания файла и операции не упорядочены* не могут возникнуть одновременно, поэтому такую комбинацию проверять не нужно.

### *Тестирование отдельных модулей*

Одно из преимуществ модульного проектирования состоит в том, что оно облегчает тестирование. Когда новый модуль готов к проверке, он добавляется способом сверху вниз, как сказано в гл. 3. Методы ограничения сложности модуля, о которых шла речь в гл. 2, не только облегчают модификацию программы, но и упрощают тестирование модулей. Планируя проверку модулей-функций, вы можете пожелать составить для них *список функциональных пар* (список случаев). Функциональная пара (случай) — это определенный тип результата работы модуля, соответствующий конкретной комбинации элементов входных данных.

Таблица 10.2 представляет собой пример списка случаев для обновления последовательного файла, фигурирующего в качест-

ве примера в главах по КОБОЛу и ПЛ/I. Такой список затем становится основой для определения необходимых тестов. Эта работа должна, вероятно, начинаться на стадии проектирования, чтобы подготовленные образцы данных могли использоваться на контрольных сессиях для проверки правильности проектирования. Конечно, как было отмечено в гл. 3, реальные тесты должны быть подготовлены до начала программирования модуля. Каждый элемент в списке случаев превращается в один из конкретных тестов. Эти случаи должны быть повторены в различных последовательностях. Например, тесты должны содержать как минимум два рядом стоящих повторения одной и той же ошибки. Если это возможно практически, то следует перебрать все сочетания типов ошибок. Этот список тестов нужно дополнить и рассмотрением следующих ситуаций:

1. *Первая запись в файле.* Как минимум один тест необходим для каждого из возможных типов первой записи. Причем нужно иметь тесты и для правильной, и для неправильной записи.

Таблица 10,2

Пример списка случаев

Случай	Условия			Действие
	Тип операции	Операция правильна	Совпадает с главной	Эффект
1	Добавить	Да	Нет	Запись добавляется
2	Добавить	Нет	Нет	Сообщение об ошибке #5
3	Добавить	Да	Да	Сообщение об ошибке #3
4	Добавить	Нет	Да	Сообщение об ошибке #5 и #3
5	Исключить	Да	Да	Запись исключается
6	Исключить	Нет	Да	Сообщение об ошибке #7
	.			
	.			

2. *Последняя запись.* Для каждого файла должен быть как минимум один тест, когда этот файл кончается *первым*. Кроме этого, каждый вводной файл должен иметь и правильные, и ошибочные записи в качестве последней записи этого файла.

Для этих ситуаций нужно создать много наборов тестов. Таблица 10.3 — пример некоторых возможных наборов таких тестов для нашей программы обновления файла.

И другие ситуации могут потребовать специального рассмотрения. К ним относятся:

## Наборы тестов

Таблица 10.3

Набор	Ошибки	Первая запись	Последняя запись	Первым кончается файл
A	Код операции	Правильная	Правильная	Операций
B	То, что выше, плюс ДОБАВИТЬ запись повторно	Неправильная	Неправильная	Оба
C	То, что выше, плюс ОБНОВИТЬ и не- совпадение ключей	Неправильная	Правильная	Операций
D	То, что выше, плюс ОШИБКИ в формате операции	Правильная	Неправильная	Операций
E	Все типы	Правильная	Неправильная	Главный

1. *Тесты большого объема.* Важно моделировать внешнюю среду будущего продукта как можно точнее. Для тестов большого объема нужно использовать реальные данные из той сферы деятельности, для которой разрабатывается прикладная программа.

2. *Многотомные тесты.* Часто большие файлы занимают несколько томов (например, один ленточный файл занимает несколько бобин или один файл прямого доступа занимает несколько пакетов дисков). Чтобы проверить работу с данными из каждого тома многотомного файла, нужно перепробовать все варианты, когда *первая* и *последняя* обрабатываемые записи находятся в каждом из томов такого файла.

Для каждого набора тестов нужно запланировать определенное время для прогона и оценки результатов. Поэтому число таких наборов не должно быть настолько большим, чтобы их обработкой стало невозможно управлять.

После того как определены наборы тестов, должен быть составлен *главный список тестов*. Образец такого списка приведен в табл. 10.4. Главный список включает все случаи тестов, за возможным исключением реальных или исторически сложившихся. Для каждого конкретного набора тестов указывается однозначно идентифицирующий его номер (или имя) и связанный с этим элементом случай (из списка случаев) вместе с типом операции (если таковой уместен) и дополнительными комментариями. Это может быть довольно кропотливой и длительной работой, но главный список тестов — очень полезное средство как для планирования и выполнения теста, так и для оценки его результатов.

Кроме запланированных тестов, нужно использовать и реальные данные из соответствующей прикладной области. Если эта

Таблица 10,4

## Пример главного списка тестов

Набор тестов					Тип операции	Номер	Случай	Комментарии
A	B	C	D	E				
✓			✓	✓	Добавить	015237	1	До первой главной записи
	✓		✓	✓	Добавить	016440	3	Повторная запись
✓	✓		✓	✓	Добавить	016441	1	
		✓	✓	✓	Исключить	016441	6	Уничтожить только что добавленную
✓	✓	✓	✓	✓	Исключить	016924	5	
		✓	✓	✓	Добавить	016925	2	
	✓	✓	✓	✓	Добавить	033101	3	Повторная запись
✓	✓	✓	✓	✓	Изменить	040485	8	
					⋮	⋮		

область до сих пор была связана с *ручным трудом*, то может оказаться желательным получить необходимые данные из реальных источников документов. Если же готовится новая программа вместо уже действующей, то реальные входные данные для старой программы могут быть непосредственно использованы для проверки новой программы. Другими словами, нужно всеми средствами стараться создать реальную операционную обстановку при тестировании программ.

Чтобы генерировать запланированные тесты, можно воспользоваться *программой-генератором тестов*. Это программа общего назначения, в которой можно с помощью спецификаций или управляющих карт указать тот тип записей данных, которые должны быть сгенерированы.

### *Тестирование межмодульных связей*

Как только мы начнем интересоваться данными, с помощью которых можно проверить, правильно ли работает программа, мы будем вынуждены посмотреть на нее «со стороны». Имеются, конечно, ситуации и эффекты, сведенные ранее в список случаев. Но межмодульные связи также должны специально проверяться, потому что большинство модулей передают данные и получают их от других модулей. В общем случае мы будем различать следующие две категории передаваемых данных:

1. Внешние данные (операции, основные записи).
2. Внутренние данные (индикаторы, счетчики, чье состояние определяется внутренней логикой модулей).

Планируя проверку межмодульных связей, полезно составить *список вход/выход (В/В)* для каждого модуля программы. Вход и выход здесь совсем не обязательно связывать с операторами чтения и записи внутри модуля. *Входом* может быть любой элемент данных, *переданный* модулю извне. Это может быть и полная запись, и часть записи, и индикатор, и счетчик. *Выход* содержит элементы, *выдаваемые* модулем. Это не обязательно элементы результата работы всей программы, так как они могут стать входными данными для других модулей. Одни и те же переменные могут быть и входными, и выходными элементами данных, если рассматриваемый модуль изменяет их значение. Пример списка В/В приведен в табл. 10.5. Этот список включает как внешние, так и внутренние данные в том смысле, как они были определены выше.

*Таблица 10.5*

**Пример списка В/В для программных модулей**

<i>Имя модуля</i>	<i>Вход</i>	<i>Выход</i>
ДОБАВИТЬ	Операция ДОБАВИТЬ Индикатор СОВПАДАЕТ С ГЛАВНОЙ	Новая главная запись Индикатор ДОБАВЛЕНА С НОВАЯ ЗАПИСЬ
ОБНОВИТЬ	Операция РАСХОД Операция ПРИХОД Индикатор СОВПАДАЕТ С ГЛАВНОЙ	Новая главная запись
ИСКЛЮЧИТЬ	Операция ИСКЛЮЧИТЬ Индикатор СОВПАДАЕТ С ГЛАВНОЙ	Ничего
ЗАВЕРШИТЬ и т. д.	Ничего  . . .	Сообщение о конце работы  . . .

Этот список полезен тем, что он помогает лучше понять, что должен делать каждый модуль. (Напомним, что функция модуля — это, преобразование входа в выход.) Если для некоторых модулей список содержит слишком много элементов, то это может указывать на слабость проекта.

Так как список В/В определяет сопряжение (передачу данных) между модулями, то этот список становится полезным руководством при разработке заглушек. При нисходящем тестировании модулей заглушки нужно программировать так, чтобы они могли принимать входные данные в качестве *аргументов* и возвращать требуемые выходные данные. При организации подготовки аргументов очень полезна *матрица аргументов* (табл. 10.6).

В этой матрице указаны имена всех модулей. Что касается элементов данных, то в матрицу следует включить буквально все, которые, как вы полагаете, будут передаваться от модуля к модулю. Те из них, которые на самом деле не потребуются, можно легко вычеркнуть позже. Каждый элемент данных, который воспринимает или выдает некоторый модуль, нужно отнести к категориям I, M и U, где

I означает, что этот элемент данных первоначально определяется этим модулем (т. е. либо вводится в программу из внешней среды, либо определяется внутри модуля с помощью инициализации);

M означает, что элемент данных может быть модифицирован этим модулем;

U означает, что значение рассматриваемого элемента данных используется этим модулем.

Для некоторых модулей в табл. 10.6 некоторые элементы данных попадают сразу в несколько категорий, например ЗАКАЗАННОЕ КОЛИЧЕСТВО, который и *модифицируется* и *используется* модулем ОБРАБОТАТЬ ПРИХОД. Проверяя такие записи в матрице, можно обнаружить упущения или противоречия. Например, можно обнаружить, что некоторый элемент данных нигде не инициализируется. Если данные поступают в модуль, но никогда в нем не используются, это тоже будет замечено. Если сопряжения спроектированы не слишком сложными (т. е. передается небольшое число элементов данных), то такого рода анализ практически вполне осуществим.

Однако главное назначение матрицы аргументов — служить основой для разработки плана тестирования по методу сверху вниз. Так как матрица показывает место инициализации данных и их зависимость, то она оказывается очень полезной при планировании проверки модулей и определении требований к заглушкам. Подробнее об этом будет рассказано в следующем разделе.

Таблица 10.6

Пример матрицы аргументов (I-данные готовятся;  
U-данные используются; M-данные изменяются).

		главный	читать операцию	проверить ключи	читать главный	добавить	исключить	обработать приход	...
Индикаторы	первый проход	I		U-M					
	новая запись	I		U-M		M	U-M		
	конец главного файла	I		U	M				
	конец файла опера- ций	I-U	M						
	совпадает с глав- ной	I-U		M		U	U		
Главная запись	ключ товара			U	I	U			
	описание				I				
	количество имею- щееся в наличии				I			U	
	общий расход				I				
	общий приход				I			U	
	заказанное коли- чество				I			U-M	
	точка заказа				I				
	повторно заказан- ное количество				I				
	общая стоимость				I			U-M	
	средняя стоимость				I			M	
	дата				I				
Операция	тип операции	U	I						
	.								
	.								
	.								

## Разработка плана тестирования

Составляя план тестирования, укажите для каждого модуля, какие именно тесты нужно выполнить. Для некоторых модулей, особенно для тех, которые находятся на вершине схемы иерархии, может потребоваться несколько наборов тестов. Для многих модулей нижнего уровня будет вполне достаточно *одного набора тестов*. Если возможно, присоединяйте модули по одному и проверяйте их также по одному.

Бывают и исключения из этого правила. Например, для двух модулей нижнего уровня, не связанных друг с другом и находящихся на разных ветвях схемы иерархии, можно скомбинировать тестовые данные так, чтобы один тест проверял сразу оба модуля. Другое исключение — когда присоединяется модуль, требующий нетривиальной заглушки. Проблема в том, что заглушка сложна и ее программирование может потребовать таких же усилий, как и программирование полного модуля. В этом случае лучше вместо заглушки сразу делать полный модуль и тестировать вместе оба модуля. Последнее исключение — в случае модуля (модулей) самого верхнего уровня, когда может понадобиться несколько модулей нижнего уровня только для того, чтобы хоть что-нибудь заработало.

Когда решено, какой именно тест будет проверять каждый модуль, следующий шаг — выяснить, какие заглушки нужны каждому модулю. Матрица аргументов (табл. 10.6) поможет вам сделать это. Модулю нужна заглушка для каждого элемента данных, отнесенного к категориям *M* и *U* в соответствующем этому модулю столбце. Например, для модуля ОБРАБОТАТЬ ПРИХОД в табл. 10.6 элемент данных ЗАКАЗАННОЕ КОЛИЧЕСТВО отмечен и как *U*, и как *M*. Этот элемент готовится модулем ЧИТАТЬ ГЛАВНЫЙ. Поэтому, чтобы выполнить модуль ОБРАБОТАТЬ ПРИХОД, модуль ЧИТАТЬ ГЛАВНЫЙ должен уже работать или как модуль, или как заглушка.

Каждый модуль нижнего уровня, который программируется вначале как заглушка и содержит *I*- или *M*-элементы, должен, конечно, поставлять эти данные. Данные категории *I* можно читать с внешнего файла или просто запастись в заглушке в качестве констант.

Если же элементы данных относятся к категории *M*, т. е. должны быть модифицированы в заглушке, то она может вернуть в вызвавший ее модуль некоторые заранее приготовленные результаты, запасенные в ней в качестве констант. Например, модуль ДОБАВИТЬ относит индикатор «СОВПАДАЕТ С ГЛАВНОЙ» к категории *U*, а «ПРОВЕРИТЬ КЛЮЧИ» относит этот же индикатор к категории *M*. Таким образом, чтобы проверить



ДОБАВИТЬ, заглушка для модуля ПРОВЕРИТЬ КЛЮЧИ должна возвращать оба возможных значения индикатора.

Передачу индикаторов между модулями и заглушками реализовать довольно легко. Если нужно передавать несколько индикаторов, старайтесь проверить все возможные комбинации. Если индикаторы двоичные, то число комбинаций равно  $2^n$ , где  $n$  — число индикаторов.

Следующий шаг при планировании проверок модулей — вычисление ожидаемых результатов после прогона тестов. Это лучше, чем брать получаемые после прогона тестов выдачи, и, не зная, что должно получиться, пытаться понять, правильно ли работал модуль. Ожидаемые результаты прогона тестов должны быть определены во всех деталях и могут включить, например:

- Последовательность, в которой должны выполняться модули для правильных и ошибочных запросов

- Количество обработанных записей (для всех входных и выходных записей)

- Список модифицированных записей (до и после обновления).

- Число и тип диагностических сообщений

- Результаты выполненных программой расчетов (в качестве защиты от накопления ошибок округления и т. п.)

Кроме этих проверок, в программу должны быть, конечно, встроены содержательные проверки. Эти проверки входят в постоянную часть программы, но могут оказаться очень полезными также и в процессе тестирования.

Еще один шаг в планировании тестов — определение того, в каком случае прогон теста считается для модуля успешным. Какого типа ошибки, если они обнаружатся, вы готовы считать допустимыми? Или какой процент ошибок можно допустить? Предположим, например, что 300 наборов тестовых данных были пропущены через только что подключенный модуль транслятора с некоего языка. Два из этих тестов прошли неудачно. Скорее всего, эти тесты касаются довольно редко используемых возможностей языка. Можно ли эту проверку считать успешной и переходить к программированию и тестированию модулей нижнего уровня? На самом деле здесь, конечно, приходится решать, эксплуатировать ли программу, когда еще не окончена отладка. Но каков бы ни был критерий успешного тестирования, старайтесь зафиксировать его до начала прогона тестов на машине.

Еще один элемент, который нужно зафиксировать — это процедура исправления ошибок и повторного тестирования. Кто будет повторять тестирование? Каким способом? Нужно ли исправлять каждую ошибку отдельно или исправления будут группироваться? Если ошибка обнаружена после того, как модуль прошел тестирование, собираетесь ли вы после ее исправления

прогонять старые тесты? Если да, то как много? Какие именно? Другими словами, для каждого вопроса подобного рода нужно иметь четкий план действий и разработать процедуру его осуществления. Это должно помочь сделать процесс тестирования менее изнурительным и более упорядоченным.

Наконец, самая последняя часть плана тестирования — *приемочный тест*. С его помощью пользователь и разработчик достигают согласия в том, что созданная программа — жизнеспособный продукт, отвечающий целям, поставленным при ее проектировании. Еще до начала фазы реализации пользователь, разработчик и ответственный за тесты должны выработать и зафиксировать критерий приемки готового продукта. Откуда можно знать, чего потребует пользователь? Поэтому лучше всего заранее привлекать пользователя и к разработке плана тестирования, и к разработке тестов.

### Прогон тестов

На стадии реализации вы будете программировать и тестировать модули сверху вниз в порядке, соответствующем плану тестирования. В этом плане должно быть оценено время, которое потребуется для каждого модуля. Успешное выполнение каждого конкретного теста становится важной вехой для проекта. Результаты каждого теста нужно сравнивать с заранее запланированными. Если обнаружены расхождения, нужно найти и устранить ошибки. Результаты каждого теста нужно документировать для ссылок в будущем. После того как проверено несколько первых модулей, можно выбрать лишь некоторые из этих тестов для проверки модулей более низкого уровня. Однако время от времени нужно прогонять все предыдущие тесты, чтобы быть уверенным в правильном подключении новых модулей.

### Выводы для руководства

Если тестирование проводить так, как описано в этой главе, то это следующим образом повлияет на руководство проектом.

1. *Кривая расходов будет совсем другой.* Так как тестирование начинается на ранней стадии разработки, то значительные ресурсы потребуются раньше, чем при традиционном проектировании. Однако общая стоимость тестирования возрасти не должна. В некоторых случаях общая стоимость даже падает. Традиционная и предполагаемая кривые затрат показаны на рис. 10.1.

2. *Стоимость тестирования легче установить.* В традиционных проектах затраты на тестирование сводятся обычно только к стоимости системного теста. Затраты, связанные с переделками из-за плохого проектирования или неправильного тестирования,

обычно относились к общим затратам на разработку. Так как стоимость тестирования теперь становится более определенной, то администрация, возможно, посвятит больше времени рассмотрению статей расхода. Зато после этого можно будет достаточно точно оценить, какова стоимость тестирования.

3. *Польза тестирования становится явной к концу разработки.* В традиционных проектах прогон тестов часто превращается

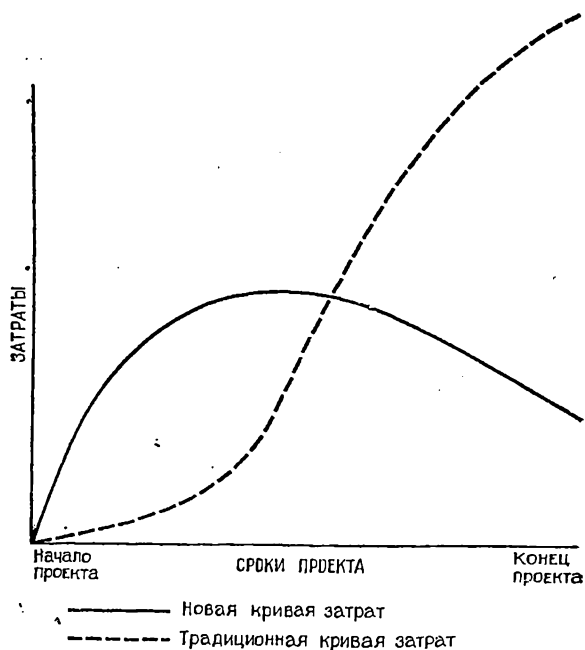


Рис. 10.1. Изменение кривой затрат на тестирование.

в хаотический многочасовой аврал, когда тратится гораздо больше времени на фиксацию ошибок, чем требуется на их предупреждение. В проектах, использующих новые усовершенствованные методы программирования, обычные трудности и отсрочки не возникают. Но наибольший выигрыш часто проявляется при использовании готового продукта, так как уменьшаются трудности и стоимость его сопровождения.

4. *Осуществление тестирования одновременно с разработкой требует дополнительного персонала.* Чтобы реализовать тестирование описанным в этой главе способом, нужны люди, полностью или частично занятые планированием и прогоном тестов. Такого рода деятельность может потребовать установления новой категории работников и планирования дополнительного обуче-

ния. Люди, занимающиеся проектированием, могут отнестись без энтузиазма к переводу их в категорию планировщиков тестов, рассматривая это как понижение. Такое отношение можно изменить, только если будет твердо установлено, что работа планировщика тестов оценивается как деятельность, от которой зависит успех всего проекта.

5. *Новый подход к тестированию может потребовать некоторых начальных затрат, связанных с его внедрением.* Как это бывает в случае внедрения любой новой идеи или метода, вначале приходится преодолевать некоторые трудности. Однако, если тестирование начинается на ранней стадии разработки, эти трудности быстро преодолеваются.

### Выводы

Тестирование может и должно планироваться до начала программирования. Это необходимо для того, чтобы проанализировать и усвоить спецификации будущей программы во всех деталях. Только тогда можно определить, что именно следует проверять. Использование анализа ситуаций-эффектов помогает прояснить спецификации.

Кроме проверки ожидаемых эффектов необходимо проверить связи между модулями. Для каждого модуля нужно составить список входных и выходных элементов данных. Это можно представить в виде матрицы аргументов, помогающей вам в дальнейшей работе.

Должен быть подготовлен план тестирования, указывающий, какие тесты должны быть пропущены для каждого модуля, нужные заглушки и ожидаемые результаты. Все это делается до начала программирования. Когда модули запрограммированы, нужно прогнать тесты и сравнить полученные результаты с ожидаемыми.

Если следовать этим рекомендациям, то потребуется меньше времени на тестирование и отладку, ошибок будет меньше и их легче будет обнаруживать и исправлять. От этого выиграют и программисты, и пользователи.

Число ошибок, обнаруженных в процессе тестирования модулей, должны быть небольшим. Это следует из всей совокупности принципов, изложенных в этой книге. Функциональное проектирование приводит к простым модулям, а в результате сквозного структурного контроля ошибки обнаруживаются до этапа машинного тестирования. Дисциплина программирования, базирующаяся на разумных управляющих структурах, приводит к более тщательно продуманной программе и, следовательно, к меньшему количеству ошибок. Строго ограниченное использование GOTO уменьшает число возможных путей через модуль — это еще один способ уменьшить сложность, облегчить

тестирование и сделать его более полным. Когда же ошибки все-таки найдены, то ясная программа позволяет легко понять их причину и место. Нисходящее тестирование более равномерно распределяет моменты обнаружения возможных ошибок, благодаря чему на исправление каждой ошибки удастся выделить больше времени.

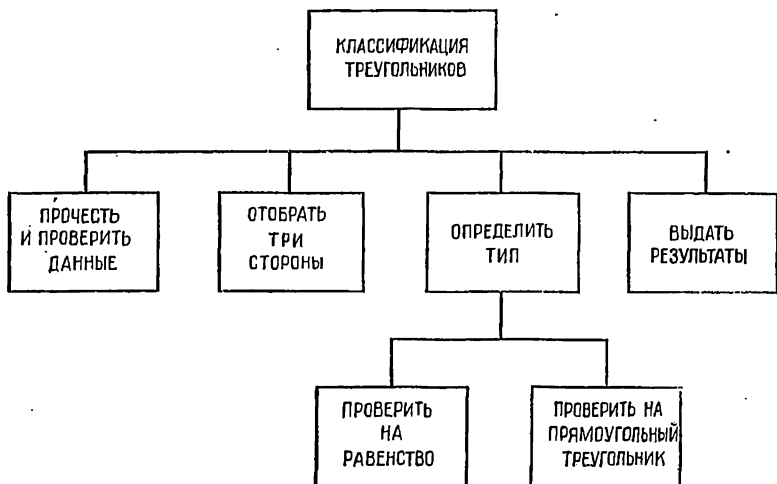
### Контрольные вопросы и упражнения

1. Перечислите как минимум три отличия тестирования от отладки.
2. Почему необходимо тратить время и усилия на анализ спецификаций?
3. Рассмотрим программу определения типа треугольника, образованного тремя сторонами. Она читает тройки чисел, интерпретируемые как длины сторон  $a$ ,  $b$  и  $c$ , и определяет тип треугольника следующим образом:

если сумма двух сторон не больше, чем третья, то это не треугольник;  
 если две стороны равны, то он равнобедренный;  
 если три стороны равны, то он равносторонний;  
 если квадрат одной стороны равен сумме квадратов других сторон, то треугольник прямоугольный.

Напишите спецификации для этой программы, используя анализ ситуаций-эффектов.

4. Составьте список случаев для программы из п. 3.
5. По каким причинам вам может понадобиться более чем один набор тестов для некоторой программы?
6. Составьте список тестов для программы из п. 3. Разделите этот тест на наборы, которые содержат различные типы правильных и неправильных данных.
7. Предположим, что для задачи из п. 3 составлена следующая схема иерархии:



Составьте список вход/выход (согласно табл. 10.5) для каждого из этих модулей.

8. Нарисуйте матрицу аргументов для программы из п. 7.

## ГЛОССАРИИ

**Абстрагирование (abstracting).** Процесс обобщения, при котором внимание сосредоточивается на сходстве объектов и на основании этого сходства объекты объединяются в группы, образуя абстракцию.

**Администратор (manager).** Сотрудник, обладающий административной и хозяйственной властью. Имеет дело с наймом, оценкой и назначением сотрудников.

**Аргумент (argument).** Элемент данных, который передается из вызывающей программы в подчиненную, где он играет роль параметра.

**Виртуальная память (virtual storage).** Адресуемое пространство, которое пользователь трактует как оперативную память. Команды и данные отображаются из него на слова оперативной памяти. Размер виртуальной памяти ограничен схемой адресации вычислительной системы (или виртуальной машины), а также объемом доступной вторичной памяти, но не фактическим объемом оперативной памяти.

**Восходящий подход (bottom-up approach).** Программирование и тестирование, начиная с модулей самого низкого уровня в схеме иерархии.

**Вызвать (invoke).** Активизировать процедуру.

**Вызов (flow of control).** Активизация одного модуля из другого с последующим возвратом.

**Вызываемая подпрограмма (called routine).** Процедура, активизируемая другой процедурой.

**Вызыватель (caller).** Модуль или сегмент, вызывающий другой модуль или сегмент.

**Выходная программа (object program).** Последовательность машинных команд, являющаяся результатом компиляции исходной программы. Объектной программой производится фактически обработка данных.

**Вычислительная организация (installation).** Конкретная вычислительная система с учетом выполняемой ею работы, а также людей, управляющих ею, работающих на ней, использующих ее для решения задач, обслуживающих ее и использующих получаемые с ее помощью результаты.

**Генератор тестов (test data generator).** Программа, результаты работы которой используются как входные данные для тестирования других программ.

**Главный (или основной) файл (master file) (определение ANS<sup>1)</sup>).** Файл, который либо сравнительно постоянен, либо считается особо важным в конкретном задании.

**Двоичный поиск (binary search).** Метод нахождения элемента в упорядоченном множестве последовательным делением пополам части этого множества, содержащей искомый элемент, до тех пор, пока часть не будет состоять только из искомого элемента.

**Дефект (exposure).** Обнаруженный при структурном контроле недостаток или упущение.

---

<sup>1)</sup> Здесь и далее ANS означает Американский национальный стандарт.— Прим. перев.

- Зависимость по данным (data dependency)*. Ситуация, при которой один модуль использует данные, создаваемые или изменяемые другим модулем.
- Заглушка (stub)*. При нисходящей разработке — программа, необходимая для тестирования модулей или сегментов более высокого уровня. Заглушки обычно невелики, требуются лишь для выдачи сообщений об ошибке или порождении данных, необходимых для указанного тестирования.
- Иерархический порядок разработки (hierarchical development order)*. Порядок написания и тестирования модулей, определяемый их уровнем в схеме иерархии. Все модули одного уровня должны быть завершены до перехода на уровень ниже.
- Индикатор (indicator)*. Переменная, изменение и проверка которой применяются для управления порядком выполнения.
- Исходное предложение (source statement)*. Предложение, входящее в текст на языке программирования.
- Ключ поиска (search argument)*. То, что в операции поиска сравнивается с элементами некоторого списка значений.
- Ключ таблицы (table argument)*. Элемент списка значений, сопоставляемый при поиске с ключом поиска.
- Константа (constant)*. Фиксированное или неизменяемое значение или элемент данных.
- Логическое выражение (logical expression)*. Комбинация переменных, констант и операций отношения, результат которой есть истинностное значение.
- Массив (array)*. Упорядоченное множество элементов данных, идентифицируемое одним именем.
- Модуль (module)*. (1) Отдельный программный объект, допускающий отдельную компиляцию, сборку с другими блоками и загрузку. (2) Функциональная компонента программы.
- Надежность (reliability)* (определение ANS). Вероятность того, что устройство или программа будет функционировать без ошибок на протяжении определенного времени или числа обращений.
- Нисходящая разработка (top-down development)*. Метод разработки программ, следующий естественному подходу к разработке систем вообще, при котором программирование начинается с наивысшего в системе уровня управления и постепенно захватывает модули все более низкого уровня. Включает нисходящее проектирование и нисходящую реализацию.
- Нисходящая реализация (top-down implementation)*. Часть нисходящей разработки, связанная с программированием и прогоном тестов.
- Нисходящее проектирование (top-down design)*. Определение всех модулей программы и их взаимосвязей «сверху вниз» вплоть до получения полной схемы иерархии. Часть нисходящей разработки.
- Обозначение проверки (decision symbol)*. Ромб, изображающий выбор из двух альтернатив. Один из трех узлов, использующихся при построении структурных блок-схем. Первый символ в конструкции ЕСЛИ-ТО-ИНАЧЕ.
- Оператор (executable statement)*. Предложение, которое определяет выполняемое программой действие, например, вычисление выражения, проверку условия, передачу управления.
- Операционный порядок (execution order)*. Порядок программирования и тестирования модулей, определяемый порядком их исполнения в готовой программе.
- Отладка (debugging)*. Процесс устранения ошибок в программе.
- Отступ (indentation unit)*. Число позиций, на которые необходимо сдвигать исходные предложения, чтобы облегчить восприятие программы. Обычно является единым в масштабах конкретной организации.
- Параметр (parameter)*. Имя в процедуре, используемое для ссылки на аргумент, передаваемый этой процедуре.
- Перебор (serial search)*. Поиск, осуществляемый последовательным сравнением каждого элемента таблицы с ключом поиска.

**Переменная (variable)** (определенные ANS). (1) Величина, которая может принимать значения из заданного множества. (2) Элемент данных, значение которого может изменяться при выполнении объектной программы.

**Планировщик (planner)**. Сотрудник, ответственный за определение сроков реализации проекта и устанавливающий порядок программирования и тестирования модулей.

**Повторение (repetition structure)**. Основная управляющая структура, в которой проверка некоторого условия определяет число повторений. Называется также циклом. ЦИКЛ-ДО и ЦИКЛ-ПОКА — два типа повторения.

**Пошаговая детализация (stepwise refinement)**. Итеративный процесс, при котором постепенно становятся известными подробности устройства программы вплоть до определения ее полной логической структуры.

**Поздняя разработка (phased-development cycle)**. Деление проекта на несколько последовательных этапов, на каждом из которых внимание сосредотачивается на определенном аспекте разработки.

**Предикатный узел (predicate node)**. Ромб, изображающий проверку. Один из трех основных символов, применяемых в структурных блок-схемах. Имеет один вход и два выхода. Представляет решение типа «истина — ложь», «да — нет», «перейти — не перейти».

**Пробный проект (pilot project)**. Проект, который тщательно изучается с точки зрения оценки нового метода, но к тому же дающий практически полезный результат.

**Программа (program)**. Объект, способный исполняться и состоящий из одного или нескольких модулей.

**Продуктивность (productivity)**. Мера полезного выхода в сравнении с затраченной ресурсами.

**Простая программа (proper program)**. Программа с одним входом и одним выходом, в которой нет тупиков и бесконечных циклов.

**Псевдокод (pseudo code)**. (1) Квазипрограмма, требующая ручного перевода на используемый язык программирования. (2) Язык, на котором пишутся такие квазипрограммы.

**Пустой «иначе» (null else)**. Альтернатива ИНАЧЕ в конструкции ЕСЛИ-ТО-ИНАЧЕ, не вызывающая каких-либо действий. Часто требуется при использовании вложенных условных предложений.

**Развилка (choice structure)**. Основная управляющая структура, в которой проверяется некоторое условие и, в зависимости от его значения, выполняется один из двух обрабатываемых блоков. Называется также ЕСЛИ-ТО-ИНАЧЕ, условным предложением или селекцией.

**Разлагать (decompose)**. Дробить на более мелкие компоненты.

**Разработчик (developer)**. Проектировщик, аналитик, программист, создатель тестов, ответственный за документацию, а также любой другой сотрудник, обеспечивающий техническую сторону проекта.

**Руководитель проекта (project leader)**. Сотрудник, координирующий действия людей, работающих над проектом, и обладающий техническими полномочиями. Не путать с администратором.

**Сборка (integration)**. Объединение и тестирование модулей для проверки правильности их совместной работы.

**Сегмент (segment)**. В структурном программировании — совокупность исходных предложений (обычно не больше 60 строк), реализующая некоторую подфункцию модуля. Сегменты обычно вызываются с помощью конструкций ВЫЗВАТЬ, ВЫПОЛНИТЬ и др., а также могут быть включены в исходный текст с помощью таких средств периода компиляции, как КОПИРОВАТЬ (в КОБОЛе) и % INCLUDE (в ПЛ/1).

**Система (system)**. Совокупность методов, программ или технических средств, объединенных регулярным способом взаимодействия в единое целое.

**Скачок (control break)**. Изменение значения поля, указывающее на окончание.



- серии записей. Обычно в этой ситуации выполняются специальные действия (например, суммирование).
- Сквозной структурный контроль** (*structured walk-through*). Официальная процедура, во время которой результаты деятельности разработчика детально изучаются с целью проверки их правильности и завершенности.
- Следование** (*sequence structure*). Основная конструкция структурного программирования, изображающая последовательное соединение функций или процессов.
- Сопровождение** (*maintenance*). Любая деятельность, направленная на исправление недостатков или последовательное улучшение программы в процессе ее эксплуатации.
- Сопровождение файла** (*file maintenance*) (определение ANS). Деятельность по поддержанию файла в соответствии с изменяющейся обстановкой путем добавления, изменения или изъятия данных.
- Сопряжение\*** (*interface*). Любая связь, которая существует между модулями. Это могут быть данные, передающиеся в одном из направлений при вызове, общие данные, а также любые предположения об устройстве одного модуля, влияющие на устройство другого.
- Спецификации** (*specifications*). Подробные сведения о том, как должна функционировать создаваемая система.
- Список вход/выход** (*input/output list*). Список, подготавливаемый для каждого модуля и показывающий, какие элементы данных поступают в этот модуль извне, какие им создаются и какие им изменяются.
- Список В/В** (*I/O list*). См. список вход/выход.
- Список данных** (*data list*). Список всех элементов данных, нужных некоторому модулю, вместе с их атрибутами. Обычно разрабатывается при пошаговой детализации.
- Структурное программирование** (*structured programming*). Проектирование, написание и тестирование программ, осуществляемые взаимосвязанным образом в соответствии с определенной дисциплиной. Метод программирования в соответствии с набором правил, облегчающих восприятие и сопровождение программ.
- Схема иерархии** (*hierarchy chart*). Графическая схема, изображающая структуру программы и применяемая при нисходящем проектировании. Схема иерархии показывает функции (или модули) программы, а также их подчиненность.
- Табличная функция** (*table function*). Запрограммированное действие, выполняемое в том случае, когда ключ поиска соответствует ключу таблицы.
- Текстовая библиотека** (*source statement library*). Средство, позволяющее хранить группы исходных предложений так, чтобы их удобно было вставлять в программы и модифицировать. Для доступа к этой библиотеке применяются операторы КОПИРОВАТЬ (в КОБОЛе) и % INCLUDE (в ПЛ/1).
- Тестирование** (*testing*). Процесс подтверждения пригодности изготовленного объекта демонстрацией его соответствия спецификациям.
- Технологическая программа** (*throw-away code*). Любая программа, использующаяся при разработке программы, но не становящаяся ее частью.
- Точка входа** (*entry point*). Точка в процедуре или подпрограмме, на которую может быть передано управление при вызове.
- Точка слияния** (*merge point*). Любая точка блок-схемы, где два или более путей сливаются в один. В структурных блок-схемах обозначается узлом слияния.
- Том** (*volume*). Съемный носитель данных, например бобина магнитной ленты или пакет магнитных дисков.
- Удобство сопровождения** (*maintainability*). Приспособленность готовой программы к изменениям.
- Узел** (*node*). Представление состояния или события с помощью символа на диаграмме. В структурном программировании означает один из трех основных

- элементов — функциональный узел, предикатный узел и узел слияния — применяемых в структурных блок-схемах.
- Узел слияния** (*collector node*). Кружок, изображающий точку слияния. Один из трех основных символов, применяемых в структурных блок-схемах. Не задает никаких действий, служит лишь точкой соединения.
- Файл** (*file*) (определение ANS). Набор связанных записей, трактуемый как единое целое.
- Функциональный узел** (*process node*). Прямоугольник, изображающий функцию или процесс. Один из трех основных символов, применяемых в структурных блок-схемах.
- Функция** (*function*). Преобразование входных данных в выходные, происходящее в результате вызова модуля.
- Цикл** (*loop structure*). См. повторение.
- Этап определения требований** (*requirements phase*). Этап, на котором подробно определяются требования к проектируемой системе. Результатом этого этапа являются системные спецификации.
- Этап планирования** (*planning phase*). Этап, на котором определяются подробности этапа реализации. Результатом этого этапа является план, определяющий порядок и способ программирования и тестирования модулей.
- Этап предварительной оценки** (*feasibility phase*). Начальный этап, в котором предварительно оцениваются требования, ожидаемая выгода и стоимость проекта, чтобы определить его жизнеспособность.
- Этап приемки** (*acceptance phase*). Этап, на котором система окончательно оформляется в согласованном с пользователем виде. Результатом этого этапа является программный продукт.
- Этап проектирования** (*design phase*). Этап, на котором завершается детальное описание системы, соответствующей выработанным ранее требованиям. В результате этого этапа получается схема иерархии, описания модулей, определения файлов и записей и т. д.
- Этап реализации** (*implementation phase*). Этап, на котором осуществляется детальное проектирование, программирование, тестирование и сборка модулей. Результатом этого этапа является готовая, проверенная система.
- Ядро** (*nucleus*). «Каркас» программы, состоящий из головного модуля и ряда заглушек и/или модулей, необходимых для его правильного выполнения.
- Язык управления заданиями** (*job control language — JCL*). Предложения, использующиеся для идентификации задания или описания его требований к операционной системе.

## СПИСОК ЛИТЕРАТУРЫ

- Armstrong Russell M. *Modular Programming in COBOL*. John Wiley and Sons, 1973.
- Aron J. D. *The Program Development Process*, Part 1. Addison-Wesley, 1974.
- Ashcroft E., Manna Z. The Translation of GO TO programs to WHILE programs. *Proceedings of IFIP Congress 1971*, Vol. I. Amsterdam, The Netherlands: North Holland Publishing Co., 1972, pp. 250—255.
- Baker F. T. Chief Programmer Team Management of Production Programming. *IBM Systems Journal*, Vol. 11, No. 1, 1972, pp. 56—73. Reprinted as G320—5320.
- Baker F. T. System Quality through Structured Programming. *Fall Joint Computer Conference*, Vol. 41, Part 1, 1972.
- Baker F. T., Mills H. D. Chief Programmer Teams, *Datamation*, December, 1973.
- Boehm B. W. Software and Its Impact: A Quantitative Assessment, *Datamation*, May 1973.
- Böhm Corrado, Jacopini Giuseppe. Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules. *Communications of Association for Computing Machinery*, Vol. 9, May 1966, pp. 366—371.
- Booth Grayce M. *Functional Analysis of Information Processing*. John Wiley and Sons, 1973.
- Brandon D. H., Gray M. *Project Control Standards*. Auerbach, 1970.
- Brown A. R., Sampson W. A. *Program Debugging*. MacDonald, 1973.
- Canning Richard G., ed. The Advent of Structured Programming. *EDP Analyzer*, Vol. 12, No. 6, June 1974.
- Canning Richard G., ed. The Search for Software Reliability. *EDP Analyzer*, Vol. 12, No. 5, May 1974.
- Constantine L. L. *Concepts in Program Design*. Paragon Press, 1967.
- Conway Richard, Gries David. An Introduction to Programming: A Structured Approach Using PL/I an PL/C. Winthrop, 1973.
- Cooper D. C. Böhm and Jacopini's Reduction of Flow Charts. *Communications of Association for Computing Machinery*, Vol. 10, No. 8, August 1967, pp. 463—473.
- Dahl O. J., Dijkstra E. W., Hoare C. A. R. *Structured Programming*. Academic Press, 1972. [Имеется перевод: Дал У., Дейкстра Э., Хоар К. Структурное программирование.— М.: Мир, 1975.]
- Dijkstra Edsger W. A Constructive Approach to the Problem of Program Correctness. *BIT*, Vol. 8, No. 3, 1968, pp. 174—186.
- Dijkstra Edsger W. Complexity Controlled by Hierarchical Ordering of Function and Variability. In *Software Engineering*, NATO Science Committee Report, eds. Peter Naur and Brian Randell, January 1969, pp. 181—185.
- Dijkstra Edsger W. GO TO Statement Considered Harmful. *Communications of Association for Computing Machinery*, Vol. 11, No. 3, March 1968, pp. 147—148.
- Dijkstra Edsger W. Notes on Structured Programming. T. H. Report 70-WSK-03. Eindhoven, Netherlands: Technological University, 1970,

- Dijkstra Edsger W. Structured Programming. In Software Engineering Techniques, NATO Science Committee, eds. J. N. Burton and B. Randell, 1969, pp. 88—94.
- Dijkstra Edsger W. The Humble Programmer. Communications of Association for Computing Machinery, Vol. 15, No. 10, October 1972, pp. 859—866.
- Dijkstra Edsger W. The Structure of «THE»—Multiprogramming System. Communications of Association for Computing Machinery, Vol. 11, No. 5, May 1968.
- Donaldson J. R. Structured Programming. Datamation, December 1973.
- Elmendorf W. R. Controlling the Functional Testing of an Operating System. IEEE Transactions on System Science and Cybernetics, Vol. SCC—5, No. 4, October 1969, pp. 284—290.
- FORTTRAN Preprocessor for Structured Programming. Form No. SB21—1614, IBM, 1975.
- Frost David. Psychology and Program Design. Datamation, May 1975, p. 137.
- Gauthier R., Pont S. Designing Systems Programs. Prentice-Hall, 1970.
- Haney F. M. Module Connection Analysis. Proceedings of the 1972 FJCC. AFIPS Press, pp. 173—180.
- HIPO—A Design Aid and Documentation Technique. Form GC20—1851, IBM.
- Hoare C. A. R. Proof for a Program; FIND. Communications of the Association for Computing Machinery, Vol. 14, No. 1, January 1971, pp. 39—45.
- Hughes Joan K. PL/I Programming. John Wiley and Sons, 1973.
- Improved Programming Technologies, Management Overview. Form GE19—5086, IBM.
- An Introduction to Structured Programming in COBOL. Form GC20—1776, IBM.
- Kernighan Brian W., Plaugher P. J. The Elements of Programming Style, McGraw-Hill, 1974.
- Knuth Donald E. The Art of Computer Programming, Volume 1: Fundamental Algorithms. Addison-Wesley, 1968. [Имеется перевод: Кнут Д. Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы.— М.: Мир, 1972.]
- Knuth Donald E. A Review of «Structured Programming». Stanford Computer Science Department Report STAN-CS-73-371, Stanford University, June 1973, 25 pp.
- Knuth Donald E. Structured Programming with go to Statements. Computing Surveys, Vol. 6, No. 4, December 1974, p. 292.
- Knuth Donald E., Floyd R. W. Notes on Avoiding GO TO Statements. Report No. SC-148, Computer Science Department, Stanford University, 1970.
- Liskov Barbara H. A Design Methodology for Reliable Software Systems. Proceedings of FJCC, December 1972.
- Maynard J. Modular Programming. Auerbach, 1972.
- McCracken Daniel D. Revolution in Programming. Datamation, December 1973.
- McCracken Daniel D., Weinberg G. M. How to Write a Readable FORTRAN Program. Datamation, Vol. 18, No. 10, October 1972, pp. 73—77.
- McGowan Clement L., III, Kelly John R. Top-Down Structured Programming Techniques. Petrocelli, 1975.
- Metzger P. W. Managing a Programming Project. Prentice-Hall, 1973.
- Miller E. F., Jr., Lindamood G. E. Structured Programming: Top-Down Approach. Datamation, December 1973.
- Mills Harlan D. Chief Programmer Teams Principles and Procedures. IBM Federal Systems Division, June 1971.
- Mills Harlan D. Mathematical Foundations for Structured Programming. IBM, Form No. FSC 72—6012, February 1972.
- Mills Harlan D. New Discipline Wins Programmer Approval, THINK, IBM Corp., March 1973.

- Mills Harlan D. On the Psychology of Quality. IBM Research, March 1972.
- Mills Harlan D. Top-Down Programming in Large Systems. In *Debugging Techniques in Large Systems*, ed. Randall Rustin, Courant Computer Science Symposium 1, MYU, 1971, pp. 41—45.
- Morris John M. Structured Programming. Pattern Analysis and Recognition Corp. Tech. Memo No. 73—20, 1973, p. 2.
- Myers Glenford J. Reliable Software Through Composite Design. Petrocelli, 1975.
- Parnas D. L. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of Association for Computing Machinery*, Vol. 5, No. 12, December 1972, pp. 1053—1058.
- Parnas D. L. A Technique for Software Module Specifications with Examples. *Communications of Association for Computing Machinery*, Vol. 14, No. 5, May 1972, pp. 330—336.
- Proceedings of the International Conference on Reliable Software, IEEE Cat, No. 75, CHO 940—7CSK, Los Angeles, 1975.
- Rhodes John. Tackle Software with Modular Programming. *Computer Decisions*, October 1973.
- Rice J. R. GO TO Statement Reconsidered. *Communications of Association for Computing Machinery*, Vol. 11, No. 8, Aug. 1968, p. 538.
- Sammes J. E. Perspective on Methods of Improving Software Development. *Software Engineering*, Vol. 1, 1970.
- Stevens W. P., Myers G. J., Constantine L. L. Structured Design. *IBM Systems Journal*, Vol. 13, No. 2, 1974.
- Stevenson Henry P., ed. *Proceedings of a Symposium on Structured Programming in COBOL*, Los Angeles, ACM, 1975.
- Stewart S. L., ed. *Concept in Quality Software Design*. U. S. Dept. of Commerce, NBS-TN-842, August 1974.
- Tenny Ted C. Structured Programming in FORTRAN. *Datamation*, July 1974.
- Van Tassel D. *Program Style, Design, Efficiency, Debugging, and Testing*. Prentice-Hall, 1974.
- Weinberg Gerald M. *Primer on Programming*. THINK, October/November 1974.
- Weinberg Gerald M. *The Psychology of Computer Programming*. Van Nostrand Reinhold Company, 1971.
- Weinberg Gerald M. *The Psychology of Computer Programming*. Van Nostrand Reinhold Company, 1971.
- Weinberg Gerald M. *The Psychology of Improved Programming Performance*. *Datamation*, April 1971.
- Wirth Niklaus. *Systematic Programming: An Introduction*. Prentice-Hall, 1973: [Имеется перевод: Вирт Н. Систематическое программирование.— М.: Мир, 1977.]
- Wirth Niklaus. Program Development by Stepwise Refinement. *Communications of Association for Computing Machinery*, Vol. 14, No. 4, April 1971, pp. 221—227.
- Wulf W. A. A Case against the GOTO. *Proceedings of the ACM Annual Conference*, Boston, August 1972, pp. 791—797. Association for Computing Machinery, 1972.
- Yourdon Edward. *Techniques of Program Structure and Design*. Prentice-Hall, 1975.