МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
Томский государственный университет систем управления и радиоэлектроники

Кафедра автоматизированных систем управления (АСУ)

В.Т. Калайда, В.В. Романенко

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методическое Пособие для студентов специальности 230105
«Программное обеспечение вычислительной техники
и автоматизированных систем»

2012

Методическое Пособие рассмотрено и рекомендовано к изданию методическим семинаром кафедры автоматизированных систем управления ТУСУР «28» июня 2012 г.

Зав. кафедрой АСУ
проф. д-р техн. наук _____А.М. Кориков

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ. ПРОБЛЕМЫ СОВРЕМЕННОГО ПРОГРАММИРОВАНИЯ

Первый этап развития программирования в первую очередь был связан с накоплением опыта в приобретении технических (стандартных) навыков написания программ. Однако, по мере развития средств вычислительной техники и накопления этих навыков, существенно изменилось положение. Возникли естественные вопросы – продолжаем ли мы делать ошибки?; является ли сам процесс написания программ правильным?; не возникла ли необходимость в создании новых методов разработки программного обеспечения?

Ответом на эти вопросы и занимается научная дисциплина – методы (технология) разработки программного обеспечения. Круг вопросов, который рассматривает данная дисциплина, связан с классическими составляющими программирования:

– написанием спецификаций;
– проектированием;
– тестированием и
– функционированием программ.

Практическую значимость данной дисциплины можно проиллюстрировать сопоставления состоянием дел по разработке технических систем и больших программных систем.

- В 1958 году в США приступили к строительству Веррацано-Нарроуского моста. По расчетам инженеров затраты на строительство определялись в размере 325 млн. $, а работы планировалось завершить в 1965 году. Это самый большой подвесной мост в мире. Работы по нему завершились в ноябре 1964 года в соответствии с проектной сметой.
- Приблизительно в это же самое время производилась разработка операционной системы OS фирмы IBM. По планам разработчиков длительность разработки составляла 5000 человеко-лет. Но, несмотря на все возможные принятые фирмой средства завершилась на 4 года позже планируемого срока.

Возникает вопрос, почему в технических системах возможен достаточно точный прогноз, тогда как при разработке программных систем он оказывается несостоятельным?

Частично на этот вопрос можно ответить следующим образом – инженеру проще предусмотреть возрастающую сложность строительства, вызванную увеличением размеров моста, чем разработчику программного обеспечения определить сложность программы большого размера.

В настоящем курсе мы будем рассматривать методы и технические приемы, введение которых позволит уменьшить стоимость и повысить надежность программ.

В общем случае *методы разработки программного обеспечения* не есть программирование, хотя программирование составляет важную его часть. Данный предмет также не сводится к проблеме изучения компиляторов и операционных систем, хотя они также играют существенную роль в рассматриваемой технологии. Точно также, проблемы электронной техники, структуры ЭВМ не являются предметом исследований, хотя и их знание в данном предмете необходимо.

Методы разработки программного обеспечения – это синтезированная дисциплина, в которой для составления алгоритмов используются *математические* методы, для оценки затрат и выбора компромиссных решений – методы *инженерных* расчетов, для определения требований к системе, учета ситуаций связанных с потерями, организации работы исполнителей и прогнозирования – методы *управления*. Все это и будет предметом наших исследований.

2. ЭТАПЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Естественно, что отдельный человек не в состоянии полностью осмыслить и построить программное обеспечение большой системы. Для управления ходом разработки больших программных систем выделяются *шесть* этапов, составляющих цикл разработки (цикл жизни) программного обеспечения:

1) анализ требований, предъявляемых к системе;
2) определение спецификаций;
3) проектирование;
4) кодирование;
5) тестирование:
   а) автономное;
   б) комплексное;
6) эксплуатация и сопровождение.

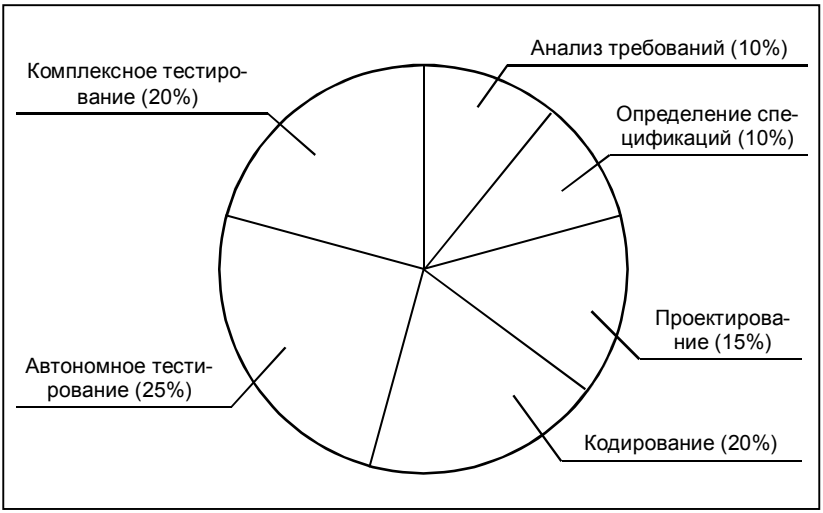


Рис. 2.1 – Распределение затрат по этапам разработки

На диаграмме (рис. 2.1) показано приблизительное распределение затрат на реализацию отдельных этапов разработки.

Рассмотрим определение каждого из этих этапов.

2.1. Анализ требований, предъявляемых к системе

На первом этапе, часто неоправданно опускаемом, определяются требования, которые позволяют получить приемлемое решение

проблемы. На этом этапе формулируется *целевое* назначение и основные *свойства* разрабатываемой программной системы.

Процесс выполнения работ и оформление результатов на этом этапе проработаны гораздо в меньшей степени, чем на других, и в общем случае не являются объектом деятельности профессиональных программистов.

Если предметом разработки является не программная система, а более сложный объект (например, система управления технологическим процессом), включающий программы только в качестве составной части, требования формируются ко всему предмету разработки. В том случае, когда разработка программного обеспечения является самоцелью, обычно используются методы составления исходных описаний. Одним из самых эффективных методов исходных описаний является метод структурного анализа, сущность которого сводится к декомпозиции исходного объекта на его составные части (рис. 2.2).

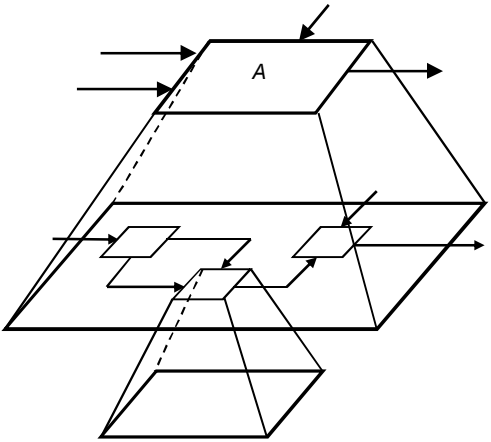


Рис. 2.2 – Схема декомпозиции системы

Таким образом, создается иерархия связанных подсистем (обязательно непересекающихся).

В общем случае, анализ требований, предъявляемых к системе, должен быть сосредоточен на интерфейсе между человеком (пользователем) и инструментом (ЭВМ). При этом для программных систем можно выделить лишь базовые требования:

- время обработки (работы) программы;
- стоимость обработки;
- вероятность ошибки;

– реакция на непредсказуемые действия оператора (защита от дурака) и др.

При декомпозиции требований следует делать различия между жесткими требованиями и требованиями, выполнение которых не является строго обязательным.

Особое внимание следует уделять пространственно-временным ограничениям и средствам системы, которые в будущем могут претерпеть изменения.

К важнейшим требованиям относятся ресурсные требования и затраты на реализацию системы.

Фактически, анализ требований завершается составлением развернутого технического задания на систему, которое в терминологии классического САПР называется аван-проектом.

2.2. Определение спецификаций

На этапе определения спецификаций осуществляется точное *описание функций*, реализуемых ЭВМ, а также задается *структура* входных и выходных данных, методы и средства их размещения. Определяются алгоритмы обработки данных.

Центральным вопросом определения спецификаций является проблема организации базы данных. При этом решается комплекс вопросов, имеющих отношение к структуре файлов, организации доступа к ним, модификации и удаления.

В случае, когда новая система создается на основе существующих, составной частью спецификаций является *схема* (алгоритм) приведения существующей базы данных к новому формату. Такое преобразование может потребовать разработку специальной программы, которая становится ненужной после ее первого и единственного использования.

Все эти вопросы должны быть отражены в функциональных спецификациях, которые представляют собой документ, являющийся основополагающим в процессе разработки системы, так как содержит конкретное *описание* последней. Чем подробней составлены спецификации, тем меньше вероятность возникновения ошибок.

В спецификациях должны быть представлены *данные* для тестирования элементов системы и системы в целом. Это требование является объективным и обязательным, т.к. на данном этапе на параметры тестирования не будет оказывать влияние конкретная реализация системы.

Так как функциональные спецификации описывают принятые решения в целом, данный документ можно использовать для началь-

ных оценок временных затрат, числа специалистов и других ресурсов, необходимых для проведения работ.

В общем случае, спецификации определяют те функции, которые должна выполнять система, не указывая, каким образом это достигается. Составление подробных алгоритмов на этом этапе преждевременно и может вызвать нежелательные осложнения.

2.3. Проектирование

На стадии проектирования разрабатываются *алгоритмы*, задаваемые спецификациями, и формируется общая *структура* вычислительной системы. При этом система разбивается (при необходимости) на составные части таким образом, чтобы ответственность за реализацию каждой составной части можно было бы возложить на одного разработчика (или группу исполнителей). При этом для каждого определенного таким образом модуля системы должны быть сформированы предъявляемые к нему требования:

- реализуемые функции;
- размеры;
- время выполнения и др.

В целом соотношения «требования – спецификация – проектирование» можно проиллюстрировать следующей схемой (рис. 2.3).

Прежде всего, Заказчик анализирует и формулирует требования реальности, которые находят отображение в требованиях, предъявляемых к программной системе. Однако ЭВМ не способна решить задачу непосредственно, реальные данные необходимо каким-то образом закодировать и ввести в ЭВМ. Подобная модель решаемой задачи представляет собой абстрактное выражение реального мира и отражается в спецификациях.

Если *спецификации* программы определены, возникает необходимость в описании процесса обработки информации, что относится к этапу *проектирования*. Поскольку программа должна использоваться для реальной задачи, то на этапе *реализации* проекта (*кодирование, тестирование*) осуществляется перевод формального проекта в выполняемую программу.

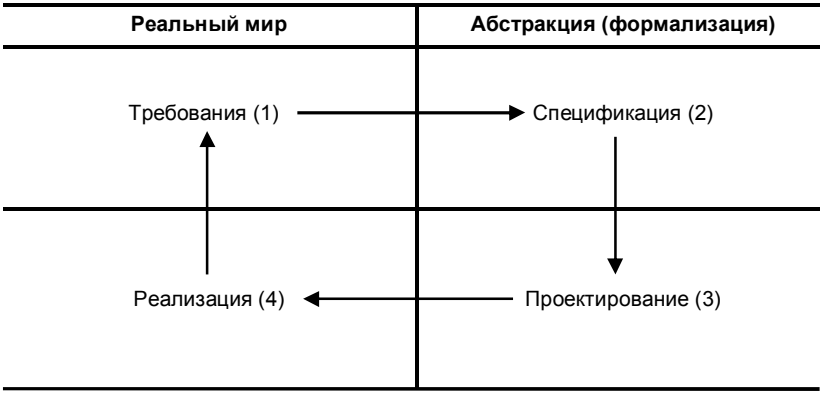


Рис. 2.3 – Схема проектирования программных систем

И, наконец, пользователь может сравнить планируемую функцию программы с реализованной функцией. Эти функции очень редко совпадают полностью. Таким образом, *сопровождение* замыкает цикл проектирования и позволяет изменить системные требования, спецификации, проекты программ и т.п.

В процессе проектирования системы, по мере выполнения спецификаций на определенные подмодули, последние представляются в виде древовидной схемы, показывающей вхождение элементов системы (рис. 2.4).

Такая схема называется базисной и она не адекватна спецификациям. Поскольку в начале этапа проектирования решение ряда функциональных задач зачастую не определено, процесс разбиения на подзадачи может быть весьма сложным. В проектировании программных систем обычной является ситуация, когда Заказчик не знает, что точно он хочет. Особенно это относится к процессам, слабо поддающимся формализации (медицина, системы военного назначения и т.д.). По мере разработки проекта Заказчик меняет спецификации. Если это происходит часто, разработка системы существенно усложняется. Выход из таких ситуаций будет нами далее проанализирован.

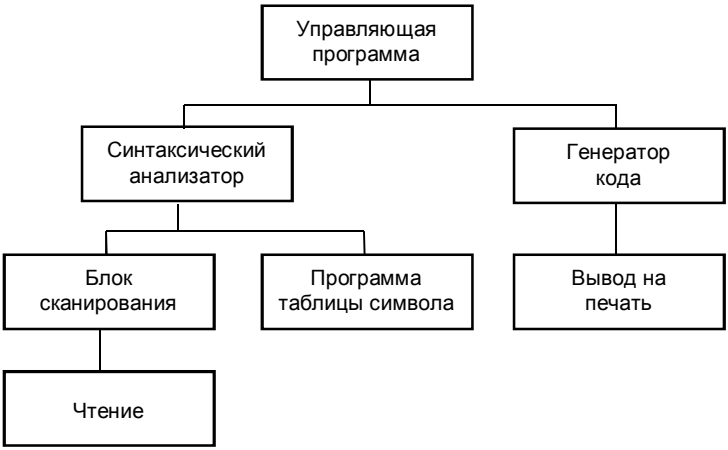


Рис. 2.4 – Структурная схема компилятора

2.4. Кодирование

Данный этап является наиболее простым, а его реализация существенно облегчается при использовании алгоритмических языков высокого уровня. Кодирование – это этап разработки программного обеспечения, доставляющий наименьшее беспокойство разработчику. По имеющимся статистическим данным 64% всех ошибок вносятся на этапах проектирования и лишь 36% на этапе кодирования. Однако эти ошибки могут быть очень дорогостоящими (**DO 4 I=1,5** и **DO 4 I=1.5**). В общем случае кодирование освоено лучше, чем другие этапы создания программ, и очень четко формализовано.

2.5. Тестирование

Этап тестирования обычно в финансовых затратах составляет половину расходов на создание системы. Плохо спланированное тестирование приводит к существенному увеличению сроков разработки системы и является основной причиной срывов графиков разработки.

В процессе тестирования используются данные, характерные для системы в рабочем состоянии, т.е. данные для тестирования выбираются случайным образом. План проведения испытаний должен быть составлен заранее, обычно на этапе проектирования.

Тестирование подразумевает три стадии:

– автономное;

– комплексное и
– системное.

При *автономном* тестировании модуль проверяется с помощью данных, подготовленных программистом. При этом программная среда модуля имитируется с помощью программ управления тестированием, содержащих фиктивные программы вместо реальных подпрограмм, с которыми имеется обращение из данного модуля (заглушки). Подобную процедуру называют программным тестированием, а программу тестирования – **UUT** (тестирующей программой). Модуль, прошедший автономное тестирование, подвергается комплексному тестированию.

В процессе *комплексного* тестирования проводится совместная проверка групп программных компонент. В результате имеем полностью проверенную систему. На данном этапе тестирование обнаруживает ошибки, пропущенные на стадии автономного тестирования. Исправление этих ошибок может составлять до ¼ от общих затрат.

Системное (или оценочное) тестирование – это завершающая стадия проверки системы, т.е. проверка системы в целом с помощью независимых тестов. Независимость тестов является главным требованием. Обычно Заказчик на стадии приемки работ настаивает на проведении собственного системного тестирования. Для случая, когда сравниваются характеристики нескольких систем (имеется альтернативная разработка), такая процедура известна как *сравнительное* тестирование.

В процессе тестирования, для определения правильности выполнения программы вводится ряд критериев:

1) каждый оператор должен быть выполнен, по крайней мере, один раз для заданного набора тестов, и программа должна выдать правильный результат;
2) каждая ветвь программы должна быть опробована, и программа при этом должна выдать правильный результат;
3) каждый путь в программе должен быть испытан хотя бы один раз с использованием набора тестовых данных, и программа должна выдать правильный результат;
4) для каждой спецификации программы необходимо располагать набором тестовых данных, позволяющих установить, что программа правильно реализует данную спецификацию.

Хотя критерии 1) и 2) кажутся схожими, в действительности они сильно разнятся. Например, арифметический оператор **IF** в Fortran:

IF (Выражение) N1, N2, N3.

Критерий 1) подразумевает, что **IF** должен быть выполнен, в то время как 2) подразумевает различные наборы данных, для того чтобы выполнились условия N1, N2, N3 (т.е. передачу на эти метки).

В общем случае не существует единого программного критерия, определяющего «хорошо проверенную» программу.

Тесно связаны с тестированием понятия «верификация» и «испытание».

Испытание системы осуществляется посредством тестирования. Цель такой проверки – показать, что система функционирует в соответствии с разработанными на нее спецификациями.

Верификация заключается в выполнении доказательств, что программа удовлетворяет своим спецификациям.

Современный процесс разработки программ не позволяет реализовать обе указанные концепции. Для ситуаций, не контролируемых тестовыми данными, система, прошедшая испытания, может дать неверные результаты. После проведения верификации система работает правильно лишь относительно первоначальных спецификаций и допущений о поведении окружающей среды; формальные доказательства правильности программ весьма сложны и слабо разработаны.

Общий процесс создания правильных программ с помощью процедур испытания и верификации называется *аттестацией.*

Различаются три вида отклонения от нормальной работы системы.

Сбой системы – это явление, связанное с нарушением системой установленных на нее спецификаций.

Данные, при обработке которых правильными алгоритмами системы происходит сбой, называются *выбросом*. Исправление выброса можно предусмотреть в программе, так что не каждый выброс может приводить к сбою.

Ошибка – это алгоритмический дефект, который создает выброс (программная ошибка).

Различают понятия «правильная» и «надежная» программа. *Правильная* программа – это та, что удовлетворяет своим спецификациям. Что касается *надежной* программы, то она не обязательно является правильной, но выдает приемлемый результат даже в том случае, когда входные данные либо условия ее использования не удовлетворяют принятым допущениям. Естественно стремление иметь «живую» (robustness) систему, т.е. систему, способную воспринимать широкий спектр входных данных при неблагоприятных условиях.

Система является *правильной*, если в системе нет ошибок, а ее внутренние данные не содержат выбросов. Система называется

надежной, если, несмотря на сбои, она продолжает удовлетворительно функционировать. Это особо видно на примере операционной системы (ОС), включающей систему обработки сбоев. При обнаружении выброса такая система прекращает работу с сохранением текущей информации и возможности продолжения работы после устранения выброса.

2.6. Эксплуатация и сопровождение

С учетом затрат на эксплуатацию и сопровождение временные затраты на разработку программной системы можно представить так (рис. 2.5):

1) анализ требований;
2) определение спецификаций;
3) проектирование;
4) кодирование;
5) автономное тестирование;
6) комплексное тестирование;
7) сопровождение.

Ни одна из вычислительных систем не остается неизменной по мере ее эксплуатации. Это объясняется несколькими причинами, среди которых можно выделить следующие:

1) Заказчик обычно не может четко сформулировать свои требования, редко бывает удовлетворен созданной системой и поэтому настаивает на внесении изменений в готовую систему;
2) могут быть обнаружены ошибки, пропущенные при тестировании;
3) могут потребоваться специальные модификации системы для частных условий функционирования, связанные с различными применениями;
4) сопровождение многочисленных компонентов системы.

Рассмотрим затраты, оказывающие наибольшее влияние на процесс разработки системы. В первую очередь следует отметить, что методы разработки, стимулирующие раннее завершение проекта, могут привести к весьма высоким затратам по сопровождению. Поэтому не следует ориентироваться на возможно ранний переход на этап кодирования. Хотя написание кодов и создает иллюзию благополучия у руководителя проекта, однако, это чревато такими последствиями, как многократное тестирование и возникновение большого числа проблем на более позднем этапе сопровождения.

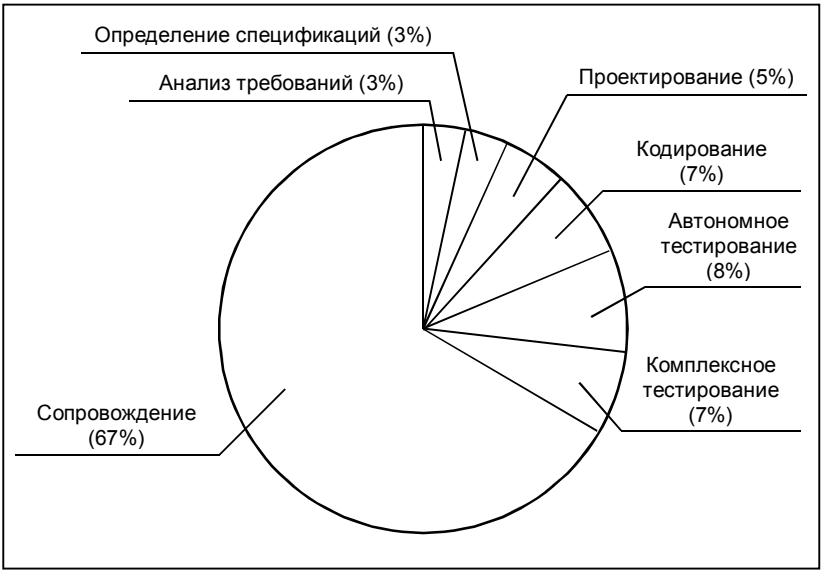


Рис. 2.5 – Временные затраты на реализацию этапов жизненного цикла программного обеспечения

Задачу сопровождения обычно трактуют как задачу отработки непомерно возрастающего числа версий системы.

Пусть некоторая система содержит компоненты **A**, **B**, **C** и установлена у потребителей **I**, **II**, **III** (рис. 2.6).

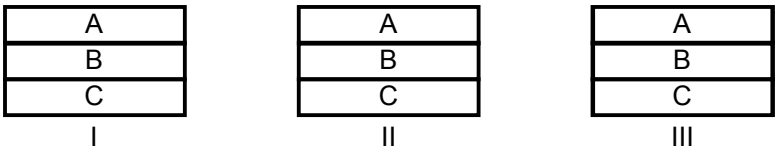


Рис. 2.6 – Исходные системы потребителей

В процессе эксплуатации системы потребитель **I** обнаружил ошибку и сообщил об этом разработчику. Последний корректирует ее и направляет исправленный модуль **A'** всем пользователям системы. Опыт применения надежного программного обеспечения показывает, что большинство потребителей ведет себя осторожно в отношении внесенных изменений. Поэтому потребители **II** и **III**, не встречаясь с

задачами, решаемыми потребителем **I**, продолжают использовать первоначальный вариант системы, поддерживая принцип «если система работает, не вмешивайся». Спустя некоторое время потребители **I** и **II** обнаружили другую ошибку в модуле **A**. Разработчик должен определить, являются ли обе обнаруженные ошибки одной и той же, поскольку использовались различные версии модуля **A**. Исправление ошибки ведет к корректировке модуля **A** и **A'**, в результате чего в эксплуатацию вводятся модули **A''** и **A'''**. Теперь функционируют уже три версии системы (рис. 2.7).

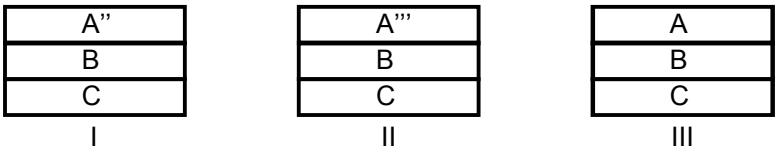


Рис. 2.7 – Система после исправления двух ошибок

Во многих случаях большая часть усилий разработчиков затрачивается на повторное обнаружение ошибок, выявленных ранее в других версиях. Чтобы исключить лавинообразное нарастание версий, системы обычно корректируются в определенные промежутки времени, называемые *периодами обновления*.

Многочисленные проблемы, возникающие на этапе сопровождения системы, должны решаться с привлечением концепции «базы данных системы». Формирование такой концепции начинается на этапе определения спецификаций. Эта база данных должна учитывать требования различных заказчиков и включать средства индикации, тестирования и устранения ошибок, применяемые для корректировки систем.

Контрольные вопросы

1) Этапы разработки программного обеспечения.
2) Анализ требований, предъявляемых к системе.
3) Жизненный цикл программного обеспечения.
4) Функциональные спецификации. Определение спецификаций.
5) Проектирование. Кодирование.
6) Тестирование: программное, системное, оценочное и сравнительное.
7) Сбой системы, выброс, ошибка. Испытания. Верификация системы.
8) Правильность и надежность программ.
9) Эксплуатация и сопровождение. Периоды обновления.

3. МЕТОДЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ КАК НАУЧНАЯ ДИСЦИПЛИНА

Из приведенного нами обзора этапов разработки программного обеспечения ясно, что каждый этап разработки оказывает влияние на другие более ранние этапы (технология «синтез – анализ»). Так, процесс написания спецификаций оказывает влияние на анализ исходных требований. На этапе проектирования вскрываются ошибки, допущенные в процессе написания спецификаций. На этапах кодирования, тестирования и эксплуатации выявляются проблемы, решить которые можно лишь на этапе проектирования. В связи со всем вышесказанным, основные цели научной дисциплины «методы разработки программного обеспечения» можно сформулировать следующим образом:

1) разработка методов управления сложными системами;
2) повышение надежности и правильности программного обеспечения;
3) развитие методов более точного прогнозирования затрат на создание программного обеспечения.

Совокупность этих проблем разделяется на методы управления разработкой и методы ведения разработки.

Методы *управления* разработкой имеют отношение к эффективной организации работы исполнителей.

Методы *проведения* разработки охватывают технические приемы работы программистов, способствующие повышению производительности их труда.

3.1. Методы управления разработкой

Руководитель проекта контролирует два основных ресурса – исполнителей и вычислительные средства. Следовательно, его основная цель – оптимизировать эти ресурсы. Успех разработки в сильной степени зависит от того, насколько руководитель следит за ходом разработки. Задержка проекта на год складывается из множества однодневных задержек.

Обычно наибольшие трудности возникают при построении интерфейса между модулями, написанными различными программистами. Поскольку количество таких интерфейсов при N исполнителях соответствует $N(N–1)/2$ и возрастает пропорционально квадрату числа исполнителей, проблема становится довольно сложной при разработке проекта группой из 4 и более человек.

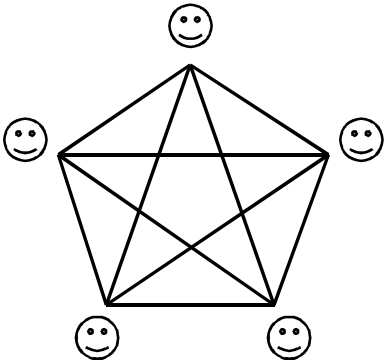


Рис. 3.1 – Для группы из 5 человек имеем 10 взаимосвязей

Пример. Программист может написать в течение года программу, включающую 5000 строк, а проектируемая система должна содержать 500000 строк, и ее разработка должна быть завершена в течение двух лет. Очевидно, что для создания такой системы достаточно пяти программистов. Однако, эти пять программистов должны взаимодействовать друг с другом, а это требует времени и в определенной степени снижает производительность труда, поскольку взаимное непонимание приводит к дополнительным затратам на тестирование (рис. 3.1).

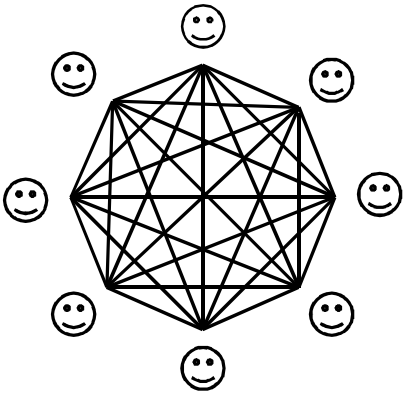


Рис. 3.2 – 8 исполнителей – 36 связей

Пусть каждый из путей взаимодействия обходится программисту в 250 строк/год. В этом случае каждый программист может соста-

вить лишь 4000 строк/год, а в течение двух лет будет составлено лишь 40000 строк. Это означает, что для написания программы из 50000 строк нужно 8 программистов, каждый из которых пишет 3250 строк/год (рис. 3.2). Для управления их работой нужен руководитель.

Однако простой подсчет строк кода не является достоверной оценкой эффективности труда программиста. Учет всех факторов, влияющих на производительность программистов, крайне сложен. Следовательно, необходимо использовать методы и приемы ограничения «коммуникационного взрыва» и увеличения производительности труда программистов.

3.1.1. Выполнение проекта

Программное обеспечение обычно разбивают на три категории:

- управляющие программы (например, операционные системы – ОС);
- системные программы (например, компиляторы);
- прикладные программы (обработка данных, счетная программа и др.).

Статистика показывает, что программист в течение года способен написать и отладить управляющую программу длиной примерно 600 строк, системную программу длиной 2000 строк и прикладную длиной 6000 строк.

Естественно, что к этим цифрам нужно относиться очень осторожно, т.к. неясно, что понимать под строкой кода. Включаются ли сюда комментарии и объявления данных? Или это генерируемая транслятором машинная команда? Как учесть программы, хранящиеся в библиотеке? Следует ли исходить из строк или операторов начального текста? В зависимости от ответов на эти вопросы количество строк кода может меняться в 2-4 раза.

Эффективность действия отдельного программиста в значительной мере зависит от типа задачи. Организация работы программиста также влияет на результаты труда. Например, в условиях дефицита времени очень мало внимания уделяется на документацию. Однако, т.к. ≈ 70% общих затрат идет на сопровождение, экономия времени на разработку документации может оказаться пагубной.

Один из методов решения поставленных проблем может состоять в учреждении должности *библиотекаря*, осуществляющего функции интерфейса между программистом и ЭВМ. Программы кодируются и передаются библиотекарю для включения их в *оперативную системную* библиотеку. Отладка модулей осуществляется программистом, однако изменение официальной копии программы в библиотеке

осуществляется только библиотекарем. Использование библиотеки еще более расширяется при наличии оперативной системы управления данными. Введение должности библиотекаря имеет еще один положительный эффект: все изменения в программных модулях системной библиотеки производит один человек, поэтому этим процессом легко управлять. Подобный подход позволяет предотвратить неоправданные изменения и побуждает программиста тщательно обдумывать каждое из них. При этом руководитель получает возможность осуществлять систематический контроль за ходом проектирования, и облегчается проверка состояния работ.

При крупномасштабных проектах для написания документации привлекаются *технические исполнители*, что позволяет освободить программистов для более квалифицированных работ.

В крупных организациях (фирма IBM) создается бригада *главного* программиста. При создании бригады исходят из того, что программисты имеют различные уровни квалификации. Организация бригады главного программиста является одним из путей уменьшения количества коммуникаций (рис. 3.3).

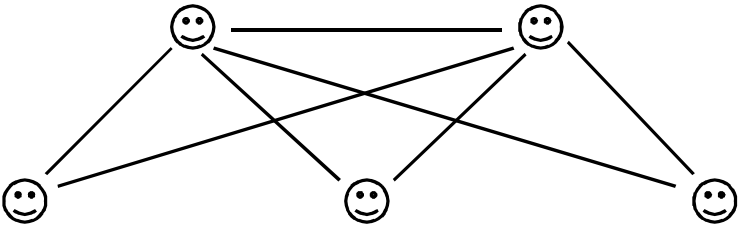


Рис. 3.3 – Снижение количества взаимосвязей при использовании бригады главного программиста

3.1.2. Методика оценки затрат

Одним из важнейших аспектов процесса разработки программного обеспечения является оценка необходимых ресурсов или требуемых затрат.

3.1.2.1 Методика инженерно-технической оценки затрат

Базовая методика оценки затрат заключается в следующем:

Шаг 1. Формируются общие требования к системе, исходя из существующего технического задания. Потенциальных разработчиков информируют о том, что ожидают от системы. Этот документ имену-

ют «списком пожеланий». Для более точного определения требуемых ресурсов проводится анализ требований.

Шаг 2. Собирается аналогичная информация, например данные о подобных системах.

Шаг 3. Отбираются основные релевантные данные.

Шаг 4. Проводится предварительная оценка.

Шаг 5. Проводится окончательная оценка системы.

Основной этап этой работы – шаг 4. При его выполнении проводятся следующие действия.

Шаг 4а. Сравнение проектируемой системы с подобными уже разработанными системами.

Шаг 4б. Разбиение системы на части и сравнение каждой из этих частей с подобными ей частями других систем.

Шаг 4в. Планирование работ и оценка затрат на каждый месяц.

Шаг 4г. Разработка соглашений, которые могут быть использованы при работе.

Отметим, что реализация шага 4а при отсутствии достаточного опыта может занять довольно продолжительный период времени. Шаг 4б требует тщательного определения понятия «часть системы». Не отличается строгим регламентом и шаг 4г, так как нет адекватного набора стандартных соглашений. Поэтому в реальной практике используются различные модификации рекомендуемого метода, базирующиеся либо на более формализованных понятиях, либо на субъективных оценках.

Метод экспертных оценок. Оценка производится исходя из личного опыта квалифицированного проектировщика (эксперта). Для многих приложений проектировщики могут прогнозировать сложность системы и оценку ресурсных затрат, даже если алгоритмы еще в явном виде не определены. Подобная оценка основывается на опыте работы проектировщика над схожей системой. Однако при этом очень велико влияние субъективных факторов, так что метод в целом не лучше, чем искусство отдельного проектировщика. Тем не менее, при отсутствии строгой теории и недостатке эмпирических знаний этот метод принимается за основу стратегии оценки затрат.

Метод алгоритмического анализа. При данном методе оценки затрат используется некоторый алгоритм. Оценка является объективной и повторяемой. Сейчас теория подобных алгоритмов развивается довольно бурно, однако пока на практике применяется лишь небольшое число подобных алгоритмов. Кроме того, алгоритм приводит к правильным результатам лишь в том случае, если используемые для

расчетов данные (спецификации) достаточно точны, что редко бывает на практике.

Пошаговый анализ. Задаются спецификации на основе пошагового анализа по нисходящему либо восходящему принципу, так что каждая определенная таким образом задача оценивается отдельно. Такой полуалгоритмический процесс может быть скомбинирован с такими методами, как метод экспертных оценок.

Закон Паркинсона. Во многих случаях для выполнения некоторой работы (задачи) затрачивается то время, которое отведено для нее, независимо от того, является ли выполнение этой работы необходимым. Каждый исполнитель вносит какой-то вклад в работу над системой и расходует определенное время. Подобный подход опирается на использование других методов, т.е. если оценка уже произведена (например, с помощью экспертов), все привлекаемые исполнители будут выполнять свою работу безотносительно ее важности и необходимости. При подобном подходе структура системы зачастую определяется составом коллектива разработчиков (если над системой работает 3 человека, то она, скорее всего, будет состоять из трех сегментов).

Затраты на завершение разработки. В некоторых случаях стоимость системы, оговариваемая при заключении договора, намеренно занижается разработчиком системы в надежде, что в последующем ее можно будет существенно увеличить за счет изменения спецификаций. После заключения договора на разработку какой-то части системы изменение спецификаций происходит по соглашению между разработчиком и заказчиком без участия других лиц. В этом случае разработчик находится в более выгодном положении, чем заказчик, поскольку на разработку системы уже затрачены значительные средства, и для заказчика нецелесообразно начинать ее заново с привлечением других специалистов. Заказчику следует проявлять осмотрительность в отношении предложений, резко отличающихся от других. Более того, он должен тщательно проводить анализ системных требований, чтобы избежать необходимости изменения спецификаций.

Психологический аспект. В некоторых случаях оценки стоимости системы различными разработчиками довольно близки. Проведенный ими алгоритмический анализ проблемы остается впечатляющим до тех пор, пока не проведен более тщательный анализ. В подобных случаях разработчики просто осведомлены о наличии средств у заказчика.

В общем случае точную оценку затрат можно дать лишь на основе опыта.

3.1.2.2. Оценка на основе распределения Рэлея

В настоящее время некоторые результаты теории надежности аппаратных средств ЭВМ начинают использовать для оценки сроков и затрат при разработке программного обеспечения. Было установлено, что зависимость суммарных затрат от времени при разработке больших систем (свыше 50 человеко-лет) хорошо отображается следующим уравнением:

$$E(t) = K\left(1 - e^{-at^2}\right),$$

где $E(t)$ – суммарные затраты к моменту времени t;

K – общая стоимость системы;

a – характеристика максимальных затрат на единичном отрезке времени.

Такая зависимость, выраженная в дифференциальной форме, отображается кривой Рэлея

$$E'(t) = 2 \cdot K \cdot a \cdot t \cdot e^{-at^2},$$

где $E'(t)$ – плотность затрат или ежегодные затраты (рис. 3.4).

Поскольку 60% затрат производится на этапе сопровождения, нет ничего удивительного, что максимум кривой близок к моменту создания системы, т.е. к тому моменту, когда традиционно считается, что работы завершены.

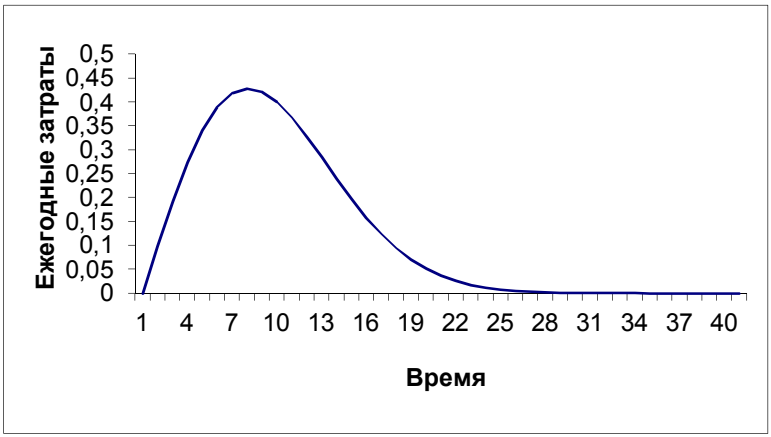


Рис. 3.4 – Отображение ежегодных затрат кривой Рэлея

Пусть $P(T > t)$ вероятность того, что в интервале $[0, t]$ событие не произошло. Тогда, в соответствии с законом Пуассона

$$P\left(T > t\right) = e^{-\lambda t}.$$

Поскольку $P(T \le t) + P(T > t) = 1$, вероятность того, что событие произошло в интервале $[0, t]$, можно представить в следующем виде:

$$P\left(T \le t\right) = 1 - P\left(T > t\right) = 1 - e^{-\lambda t}.$$

Частота событий, или скорость решения задач, выражается как производная функции распределения, т.е.

$$f\left(t\right) = \lambda e^{-\lambda t}.$$

Допустим, что если событие произошло, p есть вероятность решения задачи (вероятность получения правильного решения). Тогда получим следующие соотношения:

$$P\left(T \le t\right) = 1 - e^{-\lambda t},$$

$$f\left(t\right) = p\lambda e^{-p\lambda t}.$$

Положим, что вероятность p является функцией времени. В этом случае имеем

$$P\left(T > t\right) = e^{-\lambda \int_0^t p(\tau) d\tau},$$

$$P\left(T \le t\right) = 1 - e^{-\lambda \int_0^t p(\tau) d\tau},$$

$$f\left(t\right) = \lambda p\left(t\right) e^{-\lambda \int_0^t p(\tau) d\tau}.$$

Опыт разработки больших программных систем показывает, что зависимость вероятности правильного решения задачи от времени можно выразить в виде

$$p\left(t\right) = \alpha t.$$

При этом получим

$$P\left(T \le t\right) = 1 - e^{-\left(\lambda a t^2\right)/2}.$$

Вводя обозначение $a = (\lambda\alpha)/2$ и умножая последнюю формулу на общую стоимость системы K, получим приведенную выше формулу для суммарных затрат $E(t)$ к моменту времени t. Таким образом, для ежегодных затрат имеем:

$$E'(t) = 2at\left[K - E(t)\right].$$

Это уравнение содержит две переменные величины – t и $[K - E(t)]$. По мере приближения работ к завершению (с возрастанием t) скорость решения задач $f(t)$ увеличивается. Это происходит вследствие эффекта «обучения», поскольку по мере знакомства исполнителей с задачами работа становится более эффективной.

Противоположную тенденцию имеет выражение $[K - E(t)]$, которое определяет незавершенную работу. С приближением работ к завершению сложность системы увеличивается, вследствие чего снижается производительность труда исполнителей. Кривая Рэлея имеет два параметра – K и a. В начале работы K можно оценить, используя величину планируемых затрат, а a можно определить, исходя из состава исполнителей. Дату завершения работ определяют по достижению максимума расходов (максимум кривой Рэлея).

3.1.3. Контрольные точки

Контрольные точки указывают на моменты завершения работ; они позволяют судить о состоянии разработки системы. Контрольные точки планируются руководителем проекта с целью осуществления контроля за разработкой. Ситуацию типа «программа завершена на 90%» нельзя отнести к контрольной точке, поскольку руководитель проекта не может знать, какой объем составляют 90% программы до тех пор, пока программа не завершена.

Существует большое количество «стандартных» контрольных точек:

- выпуск функциональных спецификаций;
- завершение проектирования отдельных модулей;
- компилирование модулей без ошибок;
- успешное проведение тестирования модуля и т.д.

В техническом задании, кроме срока разработки программ, обязательно должны быть указаны контрольные точки, для того чтобы раньше выявить возможные недоработки.

Для каждой контрольной точки должны быть рассчитаны общие характеристики системы, такие как стоимость, сроки завершения, сложность.

При работе с библиотекарем должна быть разработана соответствующая форма отчетных документов по контрольным точкам.

3.1.4. Средства разработки

Компиляторы и отладочные средства известны уже достаточно давно. В настоящее время создано (и создается) ряд новых программных средств, помогающих разработке.

Среди таких средств следует особо выделить системы управления базами данных (**СУБД)**, которые помогают управлять организацией разрабатываемого программного обеспечения. Весьма удобны для контроля таблицы перекрестных ссылок, атрибутивные листинги, таблицы распределения памяти.

Одной из первых систем управления базой данных с возможностью ведения библиотеки модулей в исходном коде является разработанная в Мичиганском университете система **ISDOS**, включающая в себя язык определения задач и анализатор определения задач (**PSL/PSA**). В эту систему входит язык для описания интерфейса при проектировании программ, позволяющий осуществлять автоматическую проверку взаимосвязи программ. Схожа с указанной система **RSL** (язык определения требований), предназначенная для определения требований и интерфейсов посредством системы управления данными. Ниже эти системы будут рассмотрены подробнее.

3.1.5. Надежность

Одним из основных параметров надежности разрабатываемой программной системы является *концептуальная целостность*, т.е. единообразие стиля и простота структуры. Обычно она достигается за счет минимизации числа разработчиков. Организация бригады главного программиста также способствует концептуальной целостности системы, т.к. основная работа по проектированию в этом случае выполняется главным программистом.

Среди проектировщиков этот метод называется *интеллектуальным программированием*. Техническим документом, отражающим этот подход, является логическая структура программы (структурная схема).

Аттестация системы должна осуществляться на всех стадиях разработки. Для каждого уровня проектирования, кодирования или тестирования необходимо показать, что правильность системы сохраняется при добавлении в нее любых новых частей. Это должно быть отражено в структурной схеме. Для контроля правильности используется так называемый *контрольный анализ*. Проведение контрольного анализа периодически планируется для всех исполнителей. Для просмотра выбирается какая – то часть системы. Каждому участнику анализа выдается необходимая информация (например, ТЗ). В случае

необходимости разработчик дает пояснение. Эксперт(ы) должен сделать заключение по правильности этапа разработки. Цель контрольного анализа – обнаружение ошибок, а не их исправление. Объясняя проект другим, исполнитель может выявить нечетко сформулированную спецификацию либо незаданное условие.

Существенным является то обстоятельство, что такой анализ не ставит целью оценку работы исполнителя.

3.2. Методы проведения разработки программного обеспечения

Эти методы наиболее развиты, т.к. они относятся к области профессиональной деятельности программистов: кодирование, тестирование, отладка и др. Средства их проведения (решения) известны давно и динамически развиваются. Довольно хорошо формализованы методы определения спецификаций. Здесь уже имеется эффективная методология, хотя не все технические проблемы преодолены.

Верификация и испытания. Верификация и испытания занимают почти половину времени, отведенного на создание системы. Для уменьшения затрат, связанных с этими процессами, были разработаны средства отладки, представляющие собой специальные программы для проверки тех или иных характеристик системы. Самыми старыми и примитивными являются *дамп* и *трассировка*.

Дамп – это распечатка содержимого памяти. Однако поиск по дампу неэффективен, т.к. он выдается лишь по истечению некоторого времени после возникновения ошибки, так что причину последней установить достаточно трудно.

Трассировка – это анализ значения данных переменных после каждого выполнения оператора. В общем случае трассировка дает очень большой объем информации при минимальных пояснениях.

Анализаторы графов программы способны выявить ситуацию, когда-либо происходит обращение к неинициированной переменной, либо если после присвоения значения переменной к ней не обращаются.

Для тестирования программ используются *генераторы* тестовых данных. Проверка определенных условий в заданных точках осуществляется с помощью *согласующих компиляторов*. Для относительно простых языков разработаны системы автоматической верификации. В качестве примера средства, предназначенного для отработки этапов определения спецификаций и проектирования, можно назвать систему **PSL/PSA**.

Применительно к процессу разработки программного обеспечения термин «правильная» программа может иметь различную интер-

претацию. Мы будем пользоваться восемью основными положениями. Правильная программа:

1) не содержит синтаксических ошибок;
2) не имеет ошибок, допущенных в процессе компилирования, либо сбоев в процессе выполнения;
3) для некоторых наборов тестовых данных обеспечивает получение правильного результата;
4) для типичных наборов тестовых данных обеспечивает получение правильного результата;
5) для усложненных наборов тестовых данных обеспечивает получение правильного результата;
6) для всех возможных наборов данных, удовлетворяющих спецификации задачи, обеспечивает получение правильного результата;
7) для всех возможных наборов данных и всех вероятных условий ошибочного входа обеспечивает получение правильного результата;
8) для всех возможных наборов данных обеспечивает получение правильного результата.

Естественно, что уровень правильности **8** не всегда достижим, и, более того, не всегда необходим. Обычно для программных комплексов достаточно уровня правильности **6**.

Достижение каждого уровня правильности требует затрат, и при проектировании комплекса разработчик сам должен обосновывать необходимый уровень правильности. Одним из путей, повышающих уровень «правильности» программ, является верификация. На настоящем уровне развития методов разработки полная автоматизация верификации невозможна.

Общим для различных систем верификации является представление программы в виде графа, с каждой дугой которого соотносится некоторый предикат (утверждение) (рис. 3.5).

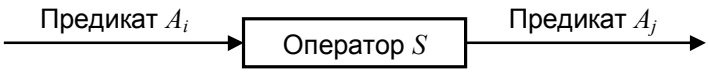


Рис. 3.5 – Связь каждого оператора программы с утверждениями A_i и A_j

Если A_i – входное утверждение, связанное с входной дугой оператора S, а A_j утверждение, связанное с исходящей дугой того же оператора, тогда доказывается правильность оператора «*если A_i истинно и оператор S выполнен, то утверждение A_j истинно*». Подобный процесс может быть повторен для каждого оператора программы. Если A_1

есть утверждение, предшествующее входному узлу графа (начальное утверждение), а A_n – утверждение на выходном узле, то «*если* A_1 *истинно и программа выполнена, то* A_n *истинно*» (рис. 3.6).

Рис. 3.6 – Определение входа и выхода программы с помощью предикатов A_1 и A_n

Подобный подход был формализован Хоаром, который сформулировал ряд аксиом, определяющих влияние каждого типа операторов в языке на утверждение. Таким образом, доказательство правильности программ сводится к доказательству теорем исчисления предикатов.

Естественно, при таком математическом подходе результаты работы программы необходимо сравнивать с ее спецификациями. Каждая программа правильна лишь по отношению к некоторому набору спецификаций. Поэтому возникают вопросы:

- эквивалентна ли спецификация предъявляемым к программе требованиям?
- и является ли программа правильной по отношению к этим спецификациям?

Это наиболее важные вопросы, которые мы будем решать при анализе правильности программ.

3.3. Развитие методов разработки программного обеспечения

Кроме традиционных хорошо известных методов разработки (проектирования) программного обеспечения в настоящее время все шире применяются методы, ориентированные на автоматизированную разработку.

3.3.1. Язык определения задач и анализатор задач

Среди таких методов следует отметить разработку Мичиганского университета **ISDOS**, в состав которой входят две базовые составляющие:

1) язык описания задач **PSL**, предназначенный для отображения функциональных требований и требований к ресурсам. Этот язык содержит набор средств объявлений, позволяю-

щих пользователю определять объекты системы, их свойства, а также соединять объекты посредством взаимосвязи;

2) анализатор определения задач **PSA** представляет собой процессор, с помощью которого осуществляется испытание предложений, написанных на языке **PSL**. Анализатор генерирует базу данных, отображающую системные требования, и осуществляет проверку последовательности и анализ полноты данных. Он также поддерживает ряд выходных документов, содержащих сведения о выборке данных, а также об ошибках, объектах, имеющихся в базе данных, взаимосвязи между ними.

Язык **PSL** имеет простой синтаксис и ориентирован на использование ключевых слов. Основными операторами этого языка являются:

PROCESS <имя>	<**имя**> определяется как новый процесс;
DISCRIPTION <текст>	<**текст**> представляет описание функции, реализуемой процессом, средствами английского языка;
SUBPARTS ARE <имя>	процесс <**имя**> связан с текущим процессом и расположен ниже его на дереве иерархии;
PART OF <имя>	процесс <**имя**> вызывает текущий процесс;
DERIVE <файл>	файл <**файл**> является выходным для текущего процесса;
USING <файл>	файл <**файл**> является входным для текущего процесса;
PROCEDURE <текст>	<**текст**> представляет описание на языке **PDL** алгоритма, реализуемого процессом.

Пример: если процесс **Y** содержит предложение **PART OF X**; то процесс **X** должен включать предложение **SUBPARTS ARE Y**;.

Система **PSL/PSA** обладает следующими характеристиками:

1) позволяет пользователю получить формализованное представление проблемы;
2) осуществляет документирование системных требований в единообразной форме;
3) способствует выявлению условий возникновения некоторых видов ошибок, обусловленных, в первую очередь, неполно-

той информации и нарушением последовательности ее ввода.

Третья составляющая часть **ISDOS** – язык проектирования программ **PDL.** Язык включает две категории структур. Имеется внешний синтаксис, построенный на основных типах операторов (**if-then-else**, **sequence** и др.). Внешний синтаксис для соединения отдельных компонент в программы. Кроме того, существует внутренний синтаксис, соответствующий конкретному применению. Внутренний синтаксис выражается предложениями на естественном языке, которые раскрываются последовательно, пока алгоритм не окажется описанным компонентами внешнего синтаксиса.

Пример:

```
Max: procedure (list);
  /* Поиск наибольшего элемента в списке */
  declare (maxmin, next) целое;
  declare list последовательность целых чисел;
  maxmin = первый элемент list;
  do while (имеются элементы в list)
    next = следующий элемент в list;
    maxmin = наибольший из next и maxmin;
  end;
  return (maxmin);
end Max;
```

Естественно, что рассматриваемая система не лишена недостатков. В первую очередь, это тот факт, что для многих спецификаций она излишне универсальна. Однако, учитывая, что эта система автоматизирована, выполняет достаточно большой объем документирования разработок, ее применение оправдано.

Тем не менее, на этапе проектирования используется ряд других подходов, некоторые из которых мы рассмотрим.

3.3.2. Система структурного анализа и проектирования SADT

SADT – это аббревиатура марки фирмы Software Technology, представляет собой ручную графическую систему, предназначенную для проектирования больших программных комплексов. Графический язык системы **SADT** – это иерархический структурированный набор диаграмм, причем каждый блок диаграммы раскрывается более детально с помощью другой диаграммы. Таким образом, структура модели представляется с все большей степенью детализации по мере разработки проекта (рис. 3.7).

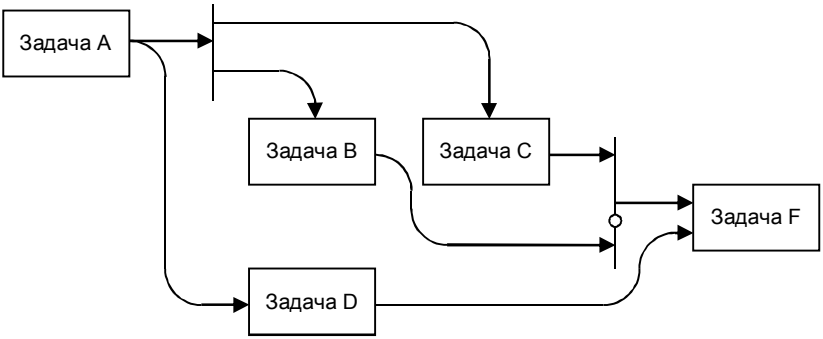


Рис. 3.7 – Структура SADT (SA – формальный язык описания взаимосвязей между компонентами системы)

Структура системы:
Задача A
(*Задача B или C*) и *Задача D*
Задача F

При использовании **SADT** каждый разработчик наделен строго определенными полномочиями.

Авторы. Разработчики, занятые изучением требований и ограничений системы и их описаний с помощью системы **SADT**.

Комментаторы. Обычно это проектировщики, анализирующие работу своих коллег (авторов) и подготавливающие замечания по ней.

Читатели. Лица, занятые анализом проектов, разрабатываемых другими специалистами, но не обязанные их комментировать.

Технический комитет. Группа опытных технических специалистов, анализирующих проект на высших уровнях его описания.

Библиотекарь проекта. Лицо, отвечающее за ведение файлов проекта.

Руководитель проекта. Лицо, несущее основную ответственность за техническую разработку проекта.

Главный аналитик. Основной консультант по использованию **SADT**. Он хорошо понимает особенности использования системы **SADT** и выдает рекомендации по ее применению.

Инструктор. Лицо, обучающее исследователей пользованию **SADT**.

К основным достоинствам **SADT** можно отнести:

1) система способствует организации коллективной работы, а также установлению соглашений относительно спецификаций на ранних этапах проектирования;
2) письменные отчеты технических комитетов позволяют проводить непрерывный контрольный анализ системы, что немаловажно при осуществлении испытаний системы;
3) эффективным средством получения специальной информации являются доклады экспертов;
4) система позволяет на ранних этапах принять основные решения высокого уровня, что создает прочный фундамент для вырабатывания последующих решений более низкого уровня;
5) использование **SADT** дает возможность осмыслить разрабатываемую систему лицам, не являющимся специалистами по программному обеспечению;
6) предусмотрены легкодоступные средства контроля хода проектирования;

Основным и главным недостатком этой системы является тот факт, что она не автоматизирована.

3.3.3. Система SREM

Система **SREM** используется для автоматизации этапа *анализа требований*, предъявляемых к программному обеспечению. Она включает в себя язык определения требований (**RSL**), посредством которого устанавливаются связи между объектами. Проверка последовательности предложений на языке **RSL** осуществляется с помощью процессора **REVS**. При использовании системы **SREM** выполняются следующие шаги.

- Трансляция. Разрабатывается система требований, включающая описатели данных и этапы их обработки.
- Декомпозиция. Разрабатываются подробные проекты.
- Распределение. С помощью процессора **REVS** моделируются отдельные аспекты проектных решений с учетом принятых допущений. В результате имеем множество требований к построению системы, генерируемых **REVS**.
- Анализ. Пользователь проверяет все требования, предъявляемые к будущей системе.

3.3.4. Методика Джексона

Разработанная М. Джексоном методика включает нисходящее проектирование, структурное программирование и структурный кон-

трольный анализ. В соответствии с этой методикой строятся иерархически структурированные программы, имеющие четыре компонента, подобные структурам управления в структурном программировании.

- Элемент. Функция, которая не может быть разбита на более простые функции.
- Последовательность. Ряд функций, реализуемых последовательно и однократно.
- Выборка. Одна из возможных последовательностей.
- Итерация. Функция, выполняемая заданное число раз, включая нулевое.

Базовая идея метода состоит в том, что структура системы должна быть идентична структуре используемых данных. Следовательно, древовидная схема организации системы должна отражать структуру данных: в противном случае проект будет неправильным.

3.4. Выводы

Проведенный обзор методов проектирования показал, что всем этим методам присущи общие черты. Хотя такие методы и способствуют получению рациональной структуры системы, тем не менее, они не исключают необходимость в творческом процессе вырабатывания основополагающих на этапах анализа требований, определения спецификаций и проектирования. Благодаря использованию этих методов становится возможным уделять больше времени профессиональной стороне разработки программного обеспечения.

Существует ряд стратегий объединения различных методов в единую методологию. Наиболее эффективную стратегию создал Боэм, который сформулировал семь принципов разработки программного обеспечения:

1) *управление разработкой на основе последовательной реализации отдельных этапов жизненного цикла программного обеспечения.* Применение этого принципа позволяет на каждом этапе принимать обоснованные решения с учетом результатов, достигнутых на предыдущих этапах, и способствует использованию контрольных точек для оценки состояния разработки проекта;
2) *выполнение испытаний.* На каждом шаге совершенствования модуля осуществляется его аттестация;
3) *осуществление строгого контроля над ходом разработки.* Вся выходная документация – проекты программ, исходные программы, инструкции пользователю и т.д. – должна быть формально утверждена. Внесение любых изменений в доку-

ментацию и библиотеки программ должно строго контролироваться и осуществляться лишь с разрешения соответствующих должностных лиц;

4) *использование всего диапазона средств структурного программирования.* По возможности желательно применять языки, позволяющие реализовать удобные структуры управления и данных. Для описания процессов желательно использовать технику пошагового совершенствования;
5) *строгий учет.* Для учета достигнутого прогресса в разработке системы необходимо использовать контрольные точки, а для проверки работы каждого исполнителя – системный журнал;
6) *использование немногочисленного штата, укомплектованного квалифицированными работниками.* Хорошие результаты работы дает использование принципа организации бригады главного программиста;
7) *Высокий уровень.* Необходимо использовать имеющиеся достижения в области технологии и разработки программного обеспечения, не забывая при этом о надежности и возможности модификации программ.

Средства управления и контроля над разработкой программного обеспечения еще требуют совершенствования, однако сам процесс управления при этом приближается к уровню, свойственному другим техническим областям.

Контрольные вопросы

1) Методы разработки программного обеспечения как научная дисциплина.
2) Организация интерфейса между модулями, написанными разными программистами.
3) Выполнение проекта. Бригада главного программиста.
4) Методика оценки затрат. Методика инженерно-технической оценки затрат.
5) Методика экспертных оценок. Метод алгоритмического анализа. Пошаговый анализ. Закон Паркинсона.
6) Затраты на завершение разработки.
7) Оценка длительности разработки на основе распределения Рэлея.
8) Контрольные точки. Средства обработки. Надежность. Концептуальная целостность.

9) Верификация и испытания. Дамп. Трассировка. Анализ графов программ.
10) «Уровни правильности» программ.
11) Методы программирования. Эффективность программ.
12) Определение спецификаций. Язык определения задач и анализатор определения задач (PSL/PSA).
13) Система структурного проектирования SADT.
14) Структурное проектирование. Методика Джексона.
15) Стратегия объединения различных методов проектирования.

4. МЕТОДЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Проведенный ранее анализ показал, что в настоящее время методы разработки программного обеспечения интенсивно развиваются. Однако далеко не все методы находят широкое практическое применение. Цель наших дальнейших исследований рассмотреть лишь те методы, которые считаются основными для разработки хороших программ.

4.1. Язык проектирования программ

При создании программ разработчики обычно используют два способа: либо составляют структурную схему программы, либо сразу пишут программу. Очевидно, что при втором способе структура программы наверняка окажется нечеткой. Однако известно, что и хорошая структурная схема может привести к созданию программы с плохой структурой. Необходима некоторая система записи, с помощью которой стало бы возможным формальное проектирование программ.

Для создания такой системы записи был разработан *язык проектирования программ* **PDL**. Этот язык строится по образу универсальных языков программирования. Однако следует отметить, что языки программирования и проектирования используются для разных целей, поэтому сравнение их недопустимо. Если языки программирования связаны с обработкой структур данных обычных программ, то языки проектирования могут включать такие «элементарные конструкции», как «решение задачи» или «обращение матрицы», или вообще не содержать выражений, заданных в явной форме. Если запись программы на языке программирования известна, то этап проектирования не обязателен.

Язык проектирования обычно состоит из двух частей:

- заданного набора операторов, построенных по образцу языков программирования;
- общего, обычно неопределяемого синтаксиса, пригодного для описания задач в данной области.

Таким образом, язык проектирования программ включает определенный *внешний синтаксис*, описывающий логику программы при проектировании. Эта логика строится при помощи управляющих структур, имеющихся в большинстве языков программирования.

С другой стороны, язык **PDL** содержит неопределенный *внутренний синтаксис*, который включает все структуры данных и процедуры по их обработке. Почти любое предложение на естественном языке можно использовать для описания преобразования данных.

if X неотрицательно
then return (квадратный корень из **X** –
действительное число);
else return (квадратный корень из **X** –
мнимое число);

В этом случае на языке **PDL** фиксируется логика программы **if** – **then** – **else**, а внутренние выражения (например, «квадратный корень») не определяются. Далее должно производиться проектирование с помощью **PDL** – программы «извлечение квадратного корня из **X**». Язык проектирования обычно называют *псевдокодом*.

Язык **PDL** включает шесть групп операторов.

Оператор выбора.

а) **if** выражение;
then оператор1;
else оператор2;

Оба оператора могут быть последовательностью операторов, входящих в группу **do** – **end**.

б) **do case** (выражение);
/индекс$_1$/ оператор1;
...
/индекс$_n$/ операторn;
else оператор$_{n+1}$;
end;

Оператор **case** используется для выбора из многих вариантов. Оператор **case** вычисляет выражение и выполняет тот оператор, у которого индекс равен значению выражения. Если никакой из индексов не равен значению выражения, то выполняется оператор **else** (если он, конечно, имеется). Как и оператор **if**, каждый из этих операторов может входить в группу **do** – **end.**

Оператор цикла.

а) **do while** (выражение);
набор операторов;
end;

Набор операторов выполняется до тех пор, пока значение выражения остается истинным.

б) **do** переменная = выраж$_1$ **to** выраж$_2$ **by** выраж$_3$;
набор операторов;

end;

При вхождении в цикл в первый раз вычисляются значения выраж$_1$, выраж$_2$ и выраж$_3$. Приращение (выраж$_3$) может быть положительным, отрицательным или опущено (по умолчанию предполагается равным +1). Цикл выполняется любое число раз.

Оператор описания данных.

declare имя атрибуты;

Имена объявляются для переменных со списками атрибутов. Атрибуты могут быть стандартными типами данных языка программирования (**real**, **float**, **integer** и др.) или структурами данных высокого уровня (рис. 4.1).

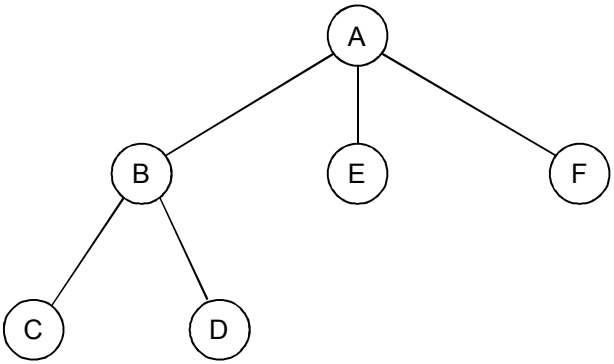


Рис. 4.1 – Структура данных в языке **PDL**

Для определения сложных структур данных используются структуры типа:

declare 1 A,
2 B,
3 C,
3 D,
2 E,
2 F;

Это соответствует структуре дерева, изображенного на рисунке. Для ссылки на элементы подобных структур используется система уточненных имен. Таким образом, на узел C можно ссылаться как на A.B.C, хотя к C можно обращаться и непосредственно, если это имя используется однозначно.

Другие операторы.

а) переменная = выражение;

б) **call** имя процедуры(список аргументов);
в) **return** (значение);
г) имя **procedure** (список параметров);
список операторов;
end;
д) **get** (список данных для ввода);
е) **put** (список данных для вывода);

Все параметры в процедуре вызываются с помощью ссылок (т.е. адреса переменных передаются в процедуру). Область действия имен – в блоке, где проведено их описание.

Оператор leave.

Оператор **leave** обеспечивает выход из цикла, организованного с помощью оператора **do**. Оператор **leave** является типом управляющего оператора перехода.

Предложения на естественном языке.

Кроме указанных пяти классов операторов, любое предложение, написанное на естественном языке можно, использовать как оператор языка **PDL**.

а) Найти наибольший элемент в массиве B;
б) **do** для всех X из {a, b, c};
в) A = первый элемент B, который больше чем C.

Для языка **PDL** разработаны специальные трансляторы. С помощью таких трансляторов можно получать документацию, которая необходима для изучения и сопровождения любой системы. Кроме того, проектирование с помощью языка, сходного с языком программирования, в значительной мере уменьшает количество ошибок.

4.2. Стратегия проектирования

4.2.1. Нисходящее проектирование и нисходящая разработка

Язык программирования является лишь средством разработки проектов. Важное место в построении правильных проектов играет методология проектирования. При разработке программ на этапе проектирования обычно используется два подхода: нисходящий и восходящий. Суть нисходящего проектирования можно объяснить следующей схемой (рис. 4.2).

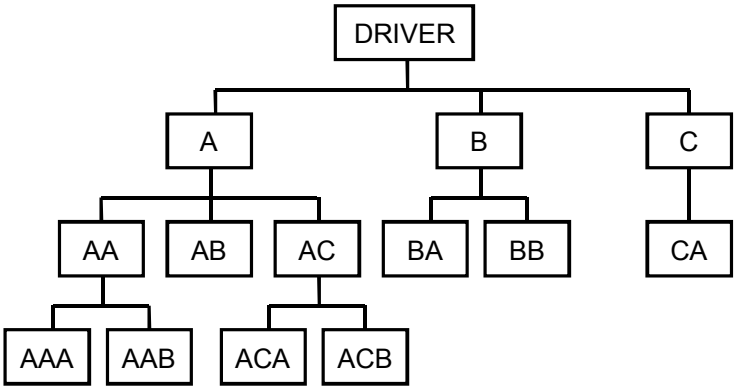


Рис. 4.2 – Пример базисной схемы

На приведенной базисной схеме каждый блок – это модуль системы. При этом вызов каждого модуля производится модулем более высокого уровня.

При нисходящем проектировании вначале проектируется управляющая программа – драйвер. Управляющий модуль может быть представлен программой на **PDL**.

```
DRIVER: procedure;
        Выполнить задачу A;
        do while (условие истинно);
          Выполнить задачу B;
        end;
        Выполнить задачу C;
      end DRIVER;
```

Затем более подробно представляются каждый из операторов псевдокода и разрабатываются другие модули. Например, если задачи A, B, C достаточно сложны, их можно оформить как отдельные процедуры. В этом случае проект драйвера можно представить следующим образом:

```
DRIVER: procedure;
        Инициировать задачу A;
        call A;
        do while (условие истинно);
          Инициировать задачу B;
          call B;
```

```
  end;
 Инициировать задачу C;
 call C;
end DRIVER;
```

Затем, таким же образом, можно определить процедуры A, B и C. Очевидно, что язык **PDL** хорошо подходит для нисходящего проектирования.

Нисходящее проектирование также называют *пошаговым совершенствованием*: программы иерархически структурируются и разбиваются путем последовательного уточнения. На каждом шаге функционирование модуля описывается с помощью ссылок на предыдущие более подробные шаги.

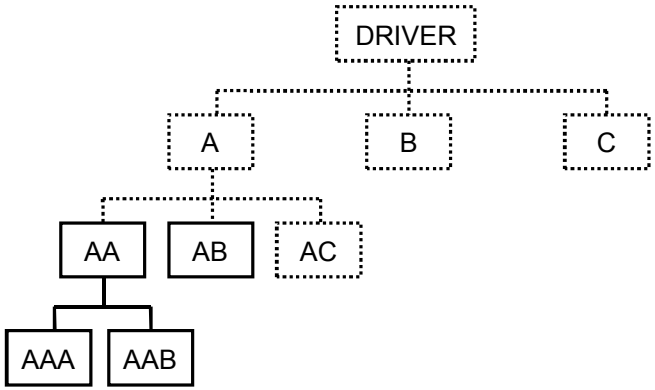


Рис. 4.3 – Восходящее кодирование и тестирование

При *восходящем проектировании* вначале проектируются программы нижнего уровня. Обычно такой подход используется при проектировании операционных систем, где самым нижним уровнем иерархии являются аппаратные средства (технология виртуальных машин). Например, один из модулей может обеспечить доступ к аппаратным средствам страничного механизма ЭВМ и предоставить виртуальную память для всех остальных модулей. Вследствие этого большинство систем реального времени проектируется снизу вверх.

На этапах кодирования и тестирования ситуация противоположная. Хотя большинство систем проектируется сверху вниз, кодирование и тестирование удобнее осуществлять снизу вверх, так как модули AAA и AAB не вызывают других компонент, их кодируют и тестируют (рис. 4.3).

Когда задача хорошо определена, пользоваться этим подходом очень удобно.

Однако, если решаемая задача не понятна или детально не определена, то тестирование снизу вверх может вызвать серьезные проблемы. Например, пользователь не может убедиться в правильности функционирования системы согласно спецификациям, пока не будет проверен модуль верхнего уровня. Однако этого нельзя сделать до тех пор, пока не будет проверена вся иерархическая структура системы, т.е. до завершения проекта. А внесение изменений на этом этапе сопряжено со значительными затратами и обходится дорого.

Чтобы избежать этого, можно использовать нисходящее кодирование. В этом случае в первую очередь проверяют модули управляющей программы, а также модули A, B, C. Пользователь системы проверяет функционирование верхнего уровня на начальном этапе разработки, поэтому сделать любые необходимые изменения в спецификациях гораздо легче.

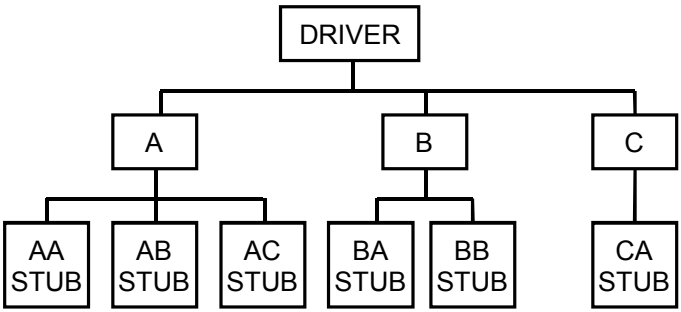


Рис. 4.4 – Нисходящее тестирование

Единственное неудобство при таком методе кодирования заключается в том, что для проверки модулей A, B, C требуются также модули AA, AB, AC, BA, BB и CA. Для этих целей служат *подыгрывающие программы – заглушки*. Это короткие программы, которые составляются специально для того, чтобы моделировать ненаписанные модули и передавать управляющим программам необходимые тестовые данные (рис. 4.4).

Подобные средства оказываются полезными, если ими пользоваться достаточно осторожно, так как корректность системы не может быть доказана, пока не убрана последняя заглушка.

Нисходящее проектирование не так просто, как это кажется на первый взгляд. Это связано с тем, что в любой программной системе имеется три вершины:

1) начало работ;
2) управляющая программа;
3) программа связи пользователя с системой.

Основные различия между этими моментами можно показать на примере компилятора. Для разработчика аппаратных средств «вершиной» системы является модуль, инициирующий работу системы. Этот модуль – основное средство интерфейса с операционной системой. С его помощью с диска считываются фрагменты программы, реализующие различные этапы компилирования, и передается управление на их выполнение. Все остальные части системы можно рассматривать как подпрограммы этого модуля.

Для системного программиста «вершиной» является управляющая программа. В компиляторе «вершиной» можно считать основной цикл анализа, который осуществляет поиск очередного анализируемого оператора (лексемы). Таким образом, логической вершиной является цикл вида:

```
do while (продолжить до конца компилирования);
  Чтение до начала следующего оператора;
  Анализ введенного оператор;
end;
```

Что касается пользователя, то, с его точки зрения, компилятор читает операторы, а затем их транслирует. Таким образом, для него «вершиной» является программа ввода данных.

От искусства и квалификации программиста зависит правильный выбор вершины для всей системы. Однако для прикладных программ всегда целесообразно «вершиной» считать *управляющую программу*.

При нисходящей разработке пользователь видит взаимодействие верхних уровней системы на начальных этапах. Изменения в этот период можно вносить относительно легко. Такими же свойствами обладает и *метод последовательной модификации* (*модернизации*). При использовании этого метода вначале проектируется и реализуется некоторый вариант системы. Пользователь очень быстро получает работающую систему. Процесс модернизации с последующим расширением функций системы продолжается до тех пор, пока не будет получена окончательная версия.

4.2.2. Структурное проектирование

Одним из эффективных методов разработки программ является метод пошагового совершенствования. Использование **PDL** хорошо

согласуется с этим методом. Программист обдумывает проект задачи все более детально, причем каждый шаг является «интеллектуально управляемой» компонентой задачи.

В начале программист представляет задачу как набор задач:

do task A;
do task B;
do task C;

Каждая из задач определяется и детализируется с помощью спецификаций. Каждую небольшую задачу можно представит в виде нескольких предложений **PDL**, входящих в некоторую процедуру. Если задача сложная, ее можно представить как отдельную процедуру.

При детализации каждой задачи можно пользоваться только операторами **PDL**. Выбор операторов этого языка не случаен – они обеспечивают управляемые конструкции проектируемых программ. При этом программы рассматриваются как функции. Любой оператор можно представить в виде $y = f(x)$. (В задачах со многими переменными x и y – векторы с соответствующим числом компонент). Таким образом, оператор присвоения

$$A = B + C \cdot D$$

можно представить в виде $F(X, Y, Z) = X + Y \cdot Z$ и заменить его следующим оператором: $A = F(X, Y, Z)$.

Аналогичным образом можно представить оператор любой степени сложности. Таким образом, каждую программу можно записать как функцию, преобразующую входные переменные в выходные.

Для формализации процесса нисходящей разработки вводится понятие простая программа. *Простая программа* определяется как программа, которую можно представить в виде структурной схемы со следующими свойствами:

1) существуют только одна входная и одна выходная дуги;
2) для каждого узла существует путь от входной дуги через этот узел к выходной дуге (рис. 4.5, 4.6).

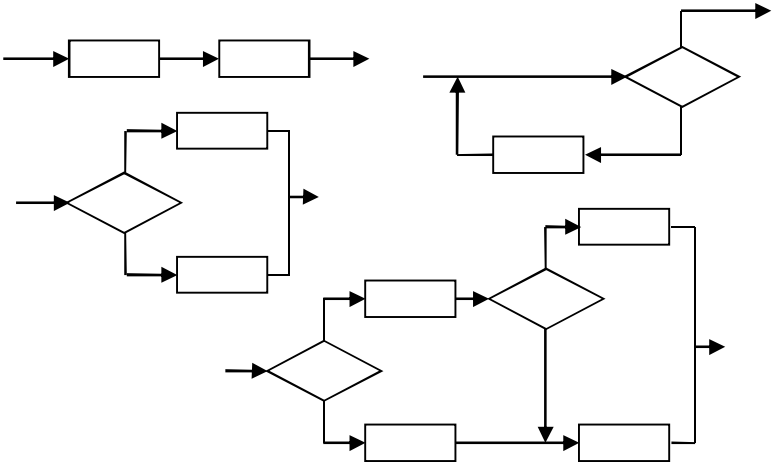


Рис. 4.5 – Примеры простых программ

Используя указанное определение, нисходящую разработку можно представить в виде следующего алгоритма:

Пусть программа представлена одним
  функциональным узлом;
do while (проектирование не окончено);
  Заменить некоторый узел простой программой;
end;

Этот алгоритм не позволяет использовать оператор **goto** и требует от программиста больших временных затрат на разработку, чем обычно. Однако для реализации метода пошагового совершенствования разработан соответствующий аппарат, ускоряющий этот процесс.

Определим *элементарную программу* как простую программу, которая не включает простых программ, состоящих более чем из одного узла.

Обычно программист, анализируя программу, изучает отдельные операторы и, разобрав группу операторов, объединяет их вместе. Такой процесс изучения программы диаметрально противоположен методу пошагового совершенствования. Некоторый функциональный узел обязательно окажется оператором присвоения, и определить его функцию относительно просто. Если небольшое количество узлов объединено в элементарные программы, то понять их функции также

относительно несложно. Конструкция **if** – **then** – **else** состоит из трех узлов (один предикат и две функции); для нее можно определить функцию, включающую в себя функции, соответствующие частям **then** и **else** оператора. Изучение сложных программ основано на объединении сведений о более мелких составных частях программы.

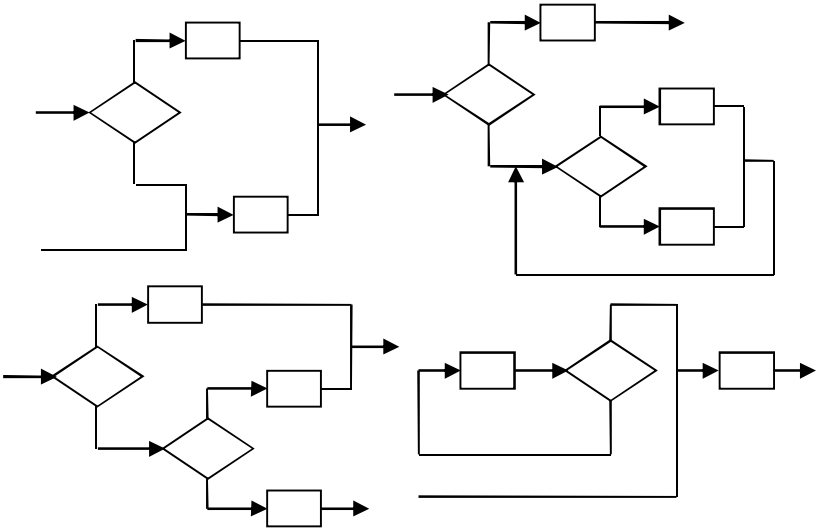


Рис. 4.6 – Примеры непростых программ

Рассмотрим программу, которая имеет несколько операторов **goto**.

В этом случае вся программа (или большая ее часть) может быть элементарной. Таким образом, разбор программы можно начать с просмотра всех узлов, поскольку непростых программ в ней не может быть.

Для структурного проектирования обычно используются следующие операторы: **if** – **then** – **else**, **while** – **do**, последовательность. Указанные управляющие структуры помогают программистам создать простые программы (функцию, которая преобразует входные данные в выходные). Пусть $f(x)$ – сегмент программы, состоящей из оператора **if** – **then** – **else** (рис. 4.7).

if $p(x)$ **then** $g(x)$ **else** $h(x)$

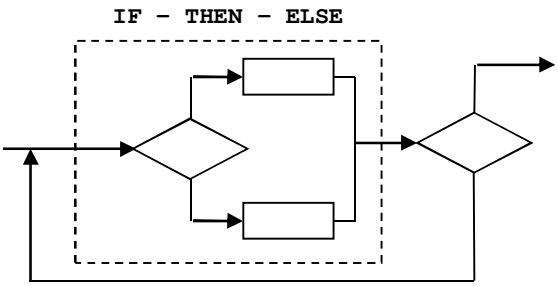


Рис. 4.7 – Разбиение на элементарные программы

Функции $g(x)$ и $h(x)$ проще, чем функция $f(x)$; таким образом, их спецификации должны быть проще. Если их спецификации известны, то функция $f(x)$ определяется следующим образом:

$$f\left(x\right) = \left(p\left(x\right) \Rightarrow g\left(x\right)\right) \cup \left(\neg p\left(x\right) \Rightarrow h\left(x\right)\right).$$

Программист может формально определить $f(x)$, зная более простые функции $g(x)$ и $h(x)$ (рис. 4.8).

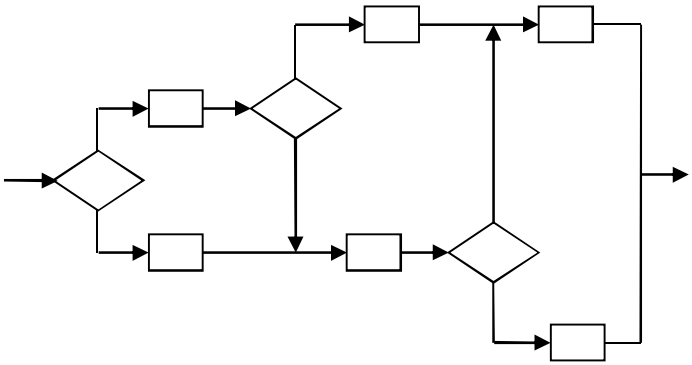


Рис. 4.8 – Сложная программа

В языке **PDL** используются элементарные подпрограммы с наименьшим числом узлов. Операторы **if** и **do while** являются минимальным набором, поскольку было доказано, что любая программная функция может быть представлена программой с указанными двумя управляющими структурами. Однако вместо них можно использовать другие конструкции, например, **repeat** – **until**.

```
repeat
  набор операторов;
until (выражение);
```

Что эквивалентно следующему:

```
набор операторов;
do while (выражение);
  набор операторов;
end;
```

Кроме того, такие же функции может реализовывать оператор **do – case** с произвольным числом узлов. Для передачи управления предназначен оператор **leave**. При его использовании передача управления принимает вид «остановить выполнение данной функции».

Заметим, что «интеллектуальная управляемость», а не «отсутствие операторов **goto**» является движущей силой процесса проектирования.

4.3. Данные

4.3.1. Обзор структур данных

Любая программа – это формальное описание решения некоторой задачи реального мира. Как часть этого решения, конкретные данные тоже должны быть формализованы таким образом, чтобы программа могла проводить вычисления. Для облегчения процесса формализации задачи в языки программирования включены наборы различных типов данных. Но так как ни один разработчик языка не сможет предвидеть всех возможных применений последнего, набор типов данных неизбежно окажется неполным.

Основным атрибутом переменной является ее *тип* или множество значений, которые может принимать переменная. Кроме того, существует набор операторов, который может оперировать с переменной данного типа.

Так как программы все более усложняются, требуются все более новые типы данных для того, чтобы моделировать задачи реального мира. Новые типы данных должны создаваться программистом на основе уже существующих.

Переменные, объявленные как *элементарные* типы данного языка, называются *скалярными* переменными, а переменные, состоящие из наборов существующих типов данных, называются *агрегативными* переменными. Из агрегативных переменных можно строить новые типы данных, для работы с которыми создаются специ-

альные операторы. Цель наших исследований – понять, по какой концепции строятся *абстрактные* (определенные пользователем) типы данных из агрегативных структур.

4.3.1.1. Массивы

Массивы – это простейшие агрегативные данные в языках программирования. Массивом называется упорядоченный набор данных одного типа

```
declare A(10) FIXED BINARY;
```

Объявлен массив A из десяти двоичных переменных с фиксированной точкой с именами A(1), A(2), A(3),. . . , A(9) A(10). Аналогично

```
declare B(5:10) FIXED BINARY;
```

объявлен массив B из шести элементов с именами B(5), B(6), . . , B(10). Массивы могут быть как одномерными, так и многомерными.

4.3.1.2. Структуры

Самой сложной разновидностью данных в языках программирования являются *структуры.* Структурой называют поименованную совокупность различных типов данных.

```
declare 1 X,
        2 Y FIXED BINARY,
        2 Z BIT(12);
```

Объявляется структура с именем X, состоящая из двоичной переменной с фиксированной точкой с именем X.Y и строки длиной 12 бит с именем X.Z.

Структуры могут использоваться для создания переменных нового типа.

4.3.1.3. Списки

Списком называют упорядоченный набор переменных одного типа. Список отличается от массива тем, что его размер обычно является переменной величиной, т.е. элементы могут добавляться в список и изыматься из него.

Список может быть объявлен как:

```
declare 1 LIST(N),
  2 DATA_ENTRIES TYPE (некоторый тип данных),
    SIZE; /* текущий размер списка */
```

Элементами списка являются DATA_ENTRIES(1), DATA_ENTRIES(2),... , DATA_ENTRIES(SIZE).

Если список может расти неограниченно, то такой способ описания не годится, поскольку SIZE может стать больше, чем N. Другой способ состоит в реализации совокупности базированных переменных.

Если список может расти неограниченно, то такой способ описания не годится, поскольку SIZE может стать больше, чем N. Другой способ состоит в реализации совокупности базированных переменных.

```
declare 1 LIST BASED,
  2 DATA_ENTRIES TYPE (некоторый тип данных);
  2 FPRT POINTER /* указатель следующей записи
                    в списке */
    LIST_HEAD POINTER; /* указатель первого
                          элемента в списке */
```

В первом случае элементами списка являются

```
LIST.DATA_ENTRIES(1),
LIST.DATA_ENTRIES(2),
LIST.DATA_ENTRIES(3),
...
LIST.DATA_ENTRIES(SIZE),
```

а во втором

```
LIST_HEAD → DATA_ENTRIES
(LIST_HEAD → FPTR) → DATA_ENTRIES
((LIST_HEAD → FPTR) → FPTR) → DATA_ENTRIES
...
```

Для работы со списками обычно используются следующие операторы (рис. 4.9а):

1) ADD – поместить новый элемент в список;
2) DELETE – удалить элемент из списка;
3) SEARCH – проверить наличие элемента в списке.

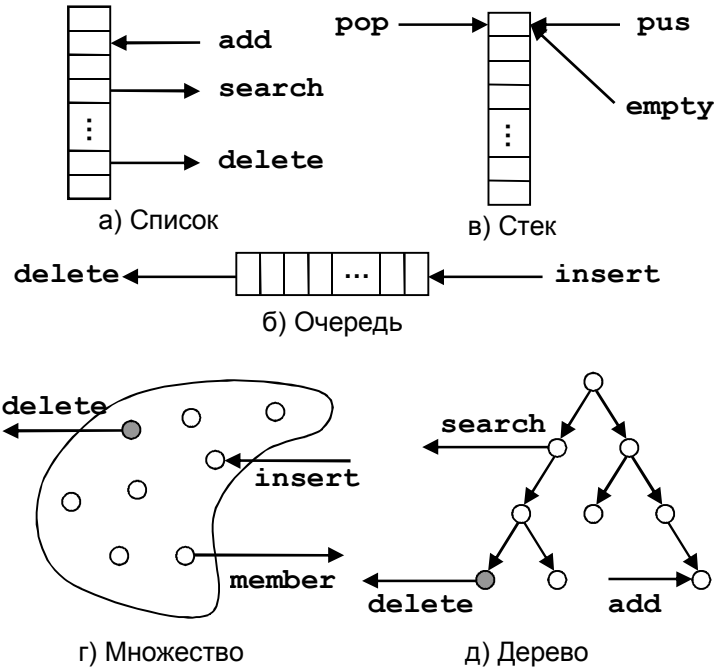


Рис. 4.9 – Агрегативные структуры и соответствующие им операторы

4.3.1.4. Очереди

Очередь – это упорядоченный список, в один конец которого элементы добавляются, а из другого изымаются (рис. 4.9б). Очередь называют списком FIFO – Fist In, Fist Out (поступивший первым обслуживается первым). Очередь может быть организована любым из рассмотренных выше способов, однако второй способ (использование указателей) более эффективен. Для обслуживания очереди необходимо две операции:

1) INSERT – добавить элемент в очередь;
2) DELETE – удалить элемент из очереди.

4.3.1.5. Стеки

Стек – это упорядоченный список, в один конец которого элементы добавляются и изымаются из этого же конца. Стек называют

списком LIFO – Last In, Fist Out (поступивший последним обслуживается первым). Аналогично очереди, стек может быть организован любым из рассмотренных выше способов, однако использование массивов более эффективно. Для работы со стеком обычно используются три операции (рис. 4.9в).

1) PUSH – поместить элемент в стек;
2) POP – извлечь элемент из стека;
3) EMPTY – функция, принимающая значение ИСТИННО, если стек не заполнен.

4.3.1.6.Множества

Множества – это совокупность переменных одного типа. Множество аналогично списку, за исключением того, что порядок расположения элементов не имеет значения. Множества обычно организуются как списки с помощью любого из двух способов.

Множества обрабатываются с использованием следующих операторов (рис. 4.9г):

1) INSERT – добавить новый элемент во множество;
2) DELETE – удалить элемент из множества;
3) MEMBER – функция, которая принимает значение ИСТИННО, если данная переменная находится во множестве.

4.3.1.7. Графы

Направленный граф – это структура, состоящая из *узлов* и *дуг,* причем каждая дуга направлена от одного узла к другому. Графы обычно организовываются с помощью базированных переменных, где дуги являются переменными типа указатель. Если из каждого узла выходит несколько дуг, то граф можно описать следующим образом:

```
declare 1 GRAPH BASED,
  2 DATA_ENTPIES TYPE (некоторый тип данных),
  2 EDGES(N) POINTER;
```

Для ненаправленных графов дугам соответствуют два направления – вперед и назад:

```
declare 1 GRAPH BASED,
  2 DATA_ENTPIES TYPE (некоторый тип данных),
  2 FORWARD_EDGES(N) POINTER,
  2 BACKWARD_EDGES(N) POINTER;
```

Операторы для работы с графами представлены на рис. 4.9д.

4.3.1.8. Деревья

Дерево – это направленный граф, обладающий следующими свойствами:

1) только один узел не имеет дуг, входящих в него (корневой узел);
2) в каждый узел входит не более одной дуги;
3) в каждый узел можно попасть из корневого узла за несколько шагов.

4.3.2. Абстрактные конструкции

В современных языках программирования основное внимание уделяется структурам данных. Управляющие операторы остались почти такими же, какими они были в первых версиях языка ALGOL. Кроме использования оператора **case** и замены оператора **goto**, обработка операторов **if**, **for** и процедур вызова претерпела незначительные изменения.

Однако, структуры данных за это время изменились в значительной степени. Ранее назначение данных состояло в представлении чисел в системе исчисления, ориентированной на ЭВМ. Поэтому в первых языках программирования использовались целочисленные и действительные типы данных, к которым применялась арифметика с фиксированной и плавающей точкой.

В более развитых языках, кроме числовых данных, включались данные, состоящие из строк символов. Кроме того, под одним именем группировались иерархически составленные структуры, состоящие из различных типов данных. Программисту представлялась возможность составлять структуры данных произвольной сложности.

Поскольку структуры данных становятся все более сложными, это сильно затрудняет процесс тестирования и сопровождение тех систем, в которых используются такие данные. Небольшие изменения одной структуры данных могут вызвать разлад в работе всей системы и превратить процесс сопровождения в сложную и дорогостоящую задачу. Кроме того, агрегативные структуры становятся все более машинно-ориентированными. Поэтому программисту приходится мыслить категориями (целочисленные, действительные, строки), а не рассматриваемой прикладной задачей.

Для того, чтобы избежать таких затруднений, в настоящее время при проектировании больших программных систем используется принцип *информационной локализованности.* Этот принцип заключается в том, что вся информация о структуре данных сосредотачивается в одном модуле. Доступ к данным осуществляется из этого модуля.

Таким образом, внесение изменения в структуру данных не сопряжено с особыми затруднениями, потому что при этом меняется только один модуль. В языках высокого уровня имена данных и представление данных тесно связаны. Если пользователь объявляет стек следующим образом:

```
declare 1 STACK,
        2 TOP FIXED,
        2 ENTRIES(100) FIXED;
```

то в модуле, который производит обращение к стеку, должна быть известна внутренняя структура последнего, т.е. число и массив с фиксированной точкой. Принцип информационной локализованности позволяет пользователю не описывать структуру данных в модуле, в котором происходит обращение к данным. Пользователю достаточно знать, что существует некоторая переменная типа STACK, и только в одном модуле, выполняющем все операции со стеком, должно быть известно представление этого стека.

Данные абстрактного типа создаются с использованием принципа информационной локализованности. Предполагается, что программа выполняет все необходимые преобразования данных. Каждый модуль в системе, в которой описываются данные, имеет вид:

```
Имя_модуля: module
   Описание структуры данных
  fun1: function
   Операторы функции
  end
  fun2: function
   Операторы функции
  end
end имя_модуля
```

Другие модули системы обращаются к этому модулю с помощью функций (fun1, fun2), и только непосредственно в модуле известна подробная структура данных.

Несмотря на это, проектирование программ с абстрактными типами данных эффективно. Можно создавать наборы данных и определять на них функции. Обращение к таким данным происходит в модуле, который описывает данные абстрактного типа. Хотя при переводе проекта программы на некоторый язык могут появиться ошибки, хороший проект является предпосылкой хорошо написанной программы.

Для создания абстрактного типа данных при проектировании язык программирования должен обеспечивать два свойства:

– возможность формирования структур данных абстрактного вида;
– возможность организации процедур обращения к таким типам данных.

В настоящее время (в зависимости от возможностей языка) используются три основных способа создания данных абстрактного типа.

4.3.2.1. Фиксированные типы данных абстрактного типа

Данный подход ориентирован на использование языков, «плохо» поддерживающих сложные структуры данных. При реализации такого подхода правильность проектирования программ зависит, главным образом, от программиста. Компилятор не в состоянии найти ошибки в использовании этих данных, так как программист определяет данные абстрактного типа как структуру данных и оформляет каждую операцию с такими данными в виде отдельной процедуры; при этом со структурами обращаются как с параметрами.

Например, для того, чтобы задать стек, программист должен добавить следующее объявление в каждую процедуру, которая содержит обращение к стеку.

```
declare 1 STACK,
        2 ENTRIES(100) TYPE(INTEGER),
        2 TOPOFSTACK TYPE(INTEGER);
```

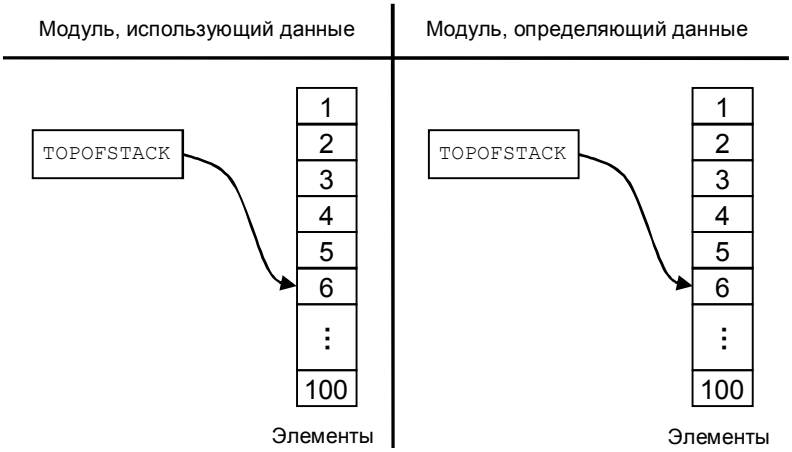


Рис. 4.12 – Фиксированные данные абстрактного типа

После этого программист должен написать процедуры для PUSH, POP и любых других функций над стеком. Если проект по-

строен по такому способу, то его перевод на язык программирования сохранит структуру данных абстрактного типа. Достоинство такого способа состоит в том, что его можно использовать для любого языка программирования (рис. 4.12).

Однако, такая организация данных имеет существенные недостатки, так как компоненты любой структуры видны во всех процедурах, которые используют эти структуры. Программист должен изменять структуры осторожно с помощью определенных операторов и не вводить кажущиеся «безвредными» операторы для непосредственного обращения к данным. Используемый тип оператора может вызвать неприятности, если каждая компонента не будет заменена соответствующим образом.

Несмотря на отмеченные недостатки, данный способ может оказаться единственно возможным для некоторых языков программирования.

4.3.2.2. Размещение указателей

Используя переменные типа указатель, можно построить конструкцию, близкую к фиксированным данным абстрактного типа. Предполагается, что указатель указывает на область памяти, а формат этой памяти определяется с помощью базированных переменных абстрактного типа. Используя данные абстрактного типа, пользователь определяет данные как переменную – указатель и вызывает процедуру для размещения данных и обращения к этой структуре. Для стека это может быть в виде:

```
declare STACK POINTER;
call STACK_INITIALIZATION(STACK);
```

В этом случае процедура STACK_INITIALIZATION должна быть вызвана явным образом для того, чтобы разместить стек в памяти (рис. 4.13).

Процедура STACK_INITIALIZATION должна иметь следующую структуру:

```
STACK_INITIALIZATION: procedure(X);
   declare X POINTER;
   declare 1 STACK BASED(X),
         /* X – указывает на стек */
           2 ENTRIES(100) FIXED,
           2 TOPOFSTACK FIXED;
  ALLOCATE STACK SET(X);
  TOPOFSTACK = 0;
```

end;

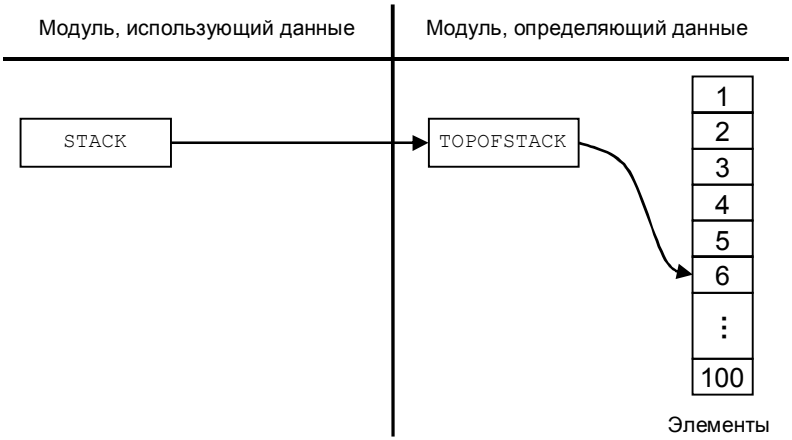


Рис. 4.13 – Размещение указателей

Только в этой процедуре известна структура данных, на которые ссылается указатель STACK, поэтому довольно сложно изменить эту структуру за пределами процедуры.

Для того, чтобы определить функции обращения к стеку, можно использовать операторы ENTRY. Таким образом, вся конструкция STACK в целом имеет следующий вид:

```
STACK: procedure;
  declare X POINTER;
  declare 1 STACK BASED(X);
          2 ENTRIES(100) FIXED,
          2 TOPOFSTACK FIXED;
  /* разместить стек */
  STACK_INITIALIZATION: ENTRY(X);
  ...
  /* поместить элемент в стек */
  PUSH: procedure(X, элемент);
  ...
       return;
  /* извлечь элемент из стека */
  POP: procedure(X, элемент);
  ...
     return;
  ...
     end;
```

Рассмотренный способ удовлетворяет основным предъявляемым требованиям: имена структур и их представления разъединены, все операции со стеком сосредоточены внутри одной процедуры (только из этой процедуры можно обращаться к данным). Однако данному способу присущ тот же недостаток, что и первому. С его помощью действительно строятся данные абстрактного типа, однако во многих современных языках не существует ограничений, которые гарантировали бы, что к этим данным никто не обращается, т.е. невозможно запретить программисту напрямую обратиться к данным, на которые ссылается указатель.

4.3.2.3. Защита данных от несанкционированного доступа

Этот способ организации данных подобен предыдущему, но содержит средства контроля для определения некорректных обращений. Для определения данных этого типа добавляется атрибут ENVIRONMENT (среда, окружение).

declare X ENVIRONMENT(STACK);

Для пользователя X – это стек, к которому можно обращаться из процедуры STACK (например, с помощью функций POP, PUSH, EMPTY) аналогично тому, как если бы X было действительное число и им можно было бы оперировать, используя операции сложения или умножения. Пользователь не знает, как составлен стек, и не имеет доступа к его внутренней структуре, так же, как и большинство программистов находятся в неведении относительно того, каким образом величина с плавающей точкой записывается в память ЭВМ. Стек объявляется компилятором как переменная – указатель, но при этом накладываются дополнительные ограничения:

1) обработка стека X происходит в процедуре FUNCTION в модуле с именем STACK;
2) описание стека X может находиться только в операторе REP (представление для переменной) в модуле STACK;
3) любые другие обращения к стеку X запрещены.

С точки зрения пользователя, данные абстрактного типа не должны отличаться от других переменных в программе. Однако имеется одно существенное различие между переменными типа FIXED и BLIPPO. Переменная типа FIXED размещается в памяти, когда начинает выполняться процедура (или блок), содержащая объявление переменной, и освобождает память, когда процедура (блок) завершится. А для переменной типа BLIPPO должна быть выделена постоянная

память. Это может привести к двусмысленной ситуации. Так как одна из целей применения данных абстрактного типа заключается в расширении типов данных для прикладных задач, различие между типами данных, генерируемых компилятором, и данных, созданных пользователями, может вызвать недоразумения. Следовательно, одной из задач проектирования с использованием данных абстрактного типа должно быть автоматическое размещение таких данных при их использовании, так чтобы не было различий в стратегии использования памяти для указанных двух типов данных.

Для того, чтобы использовать автоматическое распределение памяти, переменная X инициируется при вызове модуля STACK. По существу, компилирование объявления ENVIRONMENT эквивалентно следующей записи.

declare X POINTER INITINAL(STACK);

При первоначальном обращении к модулю STACK распределяется память для стека X и возвращается ее адрес в переменную-указатель. Модуль, в котором происходит обработка стека X, имеет вид:

```
STACK: ABSTRACTION;
REP 1 STACK[X], /* имя параметра стека*/
      2 ENTRIES(100) FIXED,
      2 TOPOFSTACK FIXED;
/* разместить стек */
/* присвоить начальные значения */
TOPOFSTACK = 0;
PUSH: function(X, элемент);
    ...
    end;
POP:  function(X, элемент);
    ...
    end;
...
end;
```

Такую процедуру легко написать на современных языках, учитывая следующее.

Начало процедуры ABSTRACTION предназначено для размещения данных абстрактного типа и присвоения им начальных значений; однако действительным распределением памяти автоматически занимается компилятор. Это делается для сохранения совместимости с элементарными типами данных, такими, как FIXED и REAL.

Слово ABSTRACTION заменяется ключевым словом PROCEDURE и добавляется атрибут RETURN(POINTER). Кроме того, используются следующие операторы:

```
declare DUMMY POINTER;
ALLOCATE datatype SET(DUMMY);
```

Эти операторы позволяют автоматически распределять память для данных абстрактного типа. Для того, чтобы обеспечить возврат из подпрограммы распределения памяти к соответствующему оператору **declare**, перед первым оператором FUNCTION помещают оператор RETURN(DUMMY). Вместо оператора REP объявляется структура типа BASED. Таким образом, транслятор транслирует следующие операторы:

```
STACK: procedure RETURNS(POINTER);
  declare 1 STACK BASED(X),
          2 ENTRIES(100) FIXED,
          2 TOPOFSTACK FIXED;
  declare DUMMY POINTER;
  ALLOCATE STACK SET(DUMMY);
  TOPOFSTACK = 0;
  return(DUMMY);
end;
```

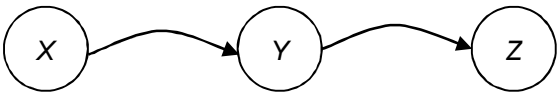
Вместо оператора FUNCTION подставляется последовательность ENTRY BEGIN. Для соответствующих операторов END добавляются операторы RETURN. Таким образом, имеем:

```
STACK: procedure RETURNS(POINTER);
  declare 1 STACK BASED(X),
          2 ENTRIES(100) FIXED,
          2 TOPOFSTACK FIXED;
  declare DUMMY POINTER;
  ALLOCATE STACK SET(DUMMY);
  TOPOFSTACK = 0;
  return(DUMMY)
  PUSH: function(X, элемент);
        ...
        return;
        end;
  POP:  function(X, элемент);
        ...
        return;
        end;
end;
```

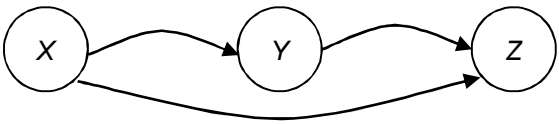
Доступ к данным. При указанном способе организации данных защитные меры приняты лишь отчасти: модуль не может получить представление любой переменной абстрактного типа, определенное в некотором другом модуле. Однако должен ли вообще данный модуль иметь доступ к определенным данным абстрактного типа? Окончательная версия программы имеет следующую структуру:

$Модуль_1$
$Модуль_2$
…
$Модуль_n$

При такой структуре программы каждый модуль может пользоваться любой функцией любого другого модуля. Однако это может оказаться нежелательным. Предположим, что модуль *X* использует абстрактный тип данных *Y*, который в свою очередь обращается к абстрактному типу данных *Z* (рис. 4.14).



X известно о *Y*;
Y известно о *Z*;
X неизвестно о *Z*
а)



X вызывает Z непосредственно
б)

Рис. 4.14 – Нарушение принципа информационной локализованности

При написании программы программист, стремясь к большей эффективности программы, записывает вызов из модуля *X* данных *Z*, тем самым неумышленно нарушает спецификацию относительно того, что о структуре данных *Z* известно только в модуле *Y*. Однако, компилятор не в состоянии проверить это нарушение, а ошибка может быть выявлена гораздо позже, например, если модуль *Y* когда-либо изменится.

В операционных системах имеется возможность защиты от некорректных действий; такую форму защиты можно добавить в языки

программирования. В операционных системах имеется *вектор возможностей*, связанный с каждым независимо выполняемым процессом. Этот вектор описывает функции, которые может выполнять данный процесс. Так как процессы начинаются и заканчиваются, когда задания пользователя соответственно поступают в систему или выходят из нее, вектор возможностей динамичен, и производится проверка правомерности операций во время работы системы.

В языках программирования аналогом такого вектора является *право доступа*. По сравнению с вектором возможностей права доступа обладают одним значительным преимуществом: проверка производится во время компиляции, а не во время выполнения программы. Программа статична, все ее процедуры известны до начала выполнения, все абстрактные типы данных известны, и проверка правильности обращения к данным может быть выполнена компилятором.

Заметим, что цель использования прав доступа заключается в ограничении доступа к абстрактным типам данных, а не в обеспечении безопасности системы. Предполагается, что программист может обращаться ко всем элементам системы, поэтому права доступа могут изменяться.

Контрольные вопросы

1) Язык проектирования программ PDL. Операторы выбора. Операторы цикла. Операторы описания данных. Операторы ввода/вывода и вызова процедур. Оператор **leave**. Предложения на естественном языке.
2) Нисходящее проектирование и нисходящая разработка.
3) Пошаговое совершенствование.
4) Восходящее проектирование.
5) Подыгрывающие программы (заглушки).
6) Структурное проектирование. Простая программа. Элементарная программа. Управляющие структуры, способы их описания.
7) Скалярные и агрегативные типы данных.
8) Массивы. Структуры. Списки. Очереди. Стеки. Множества. Графы. Деревья.
9) Абстрактные конструкции.
10) Фиксированные данные абстрактного типа.
11) Размещение указателей.
12) Защита данных от несанкционированного доступа.

5. ПРАВИЛЬНОСТЬ ПРОГРАММ

Одним из важных методов повышения эффективности проектирования программ является верификация программ или математическое доказательство того, что программа работает правильно. Для доказательства правильности программ используется аксиоматический подход, при котором применяется теория перечисления предикатов. Предполагается, что каждый оператор в программе выполняет заранее определенные действия, зависящие только от синтаксиса языка. Для двух предикатов P и Q и оператора S необходимо определить истинно ли выражение:

«Если P истинно и если выполняется оператор S, то Q истинно».

Предикат P является спецификацией правильного выполнения оператора S, а предикат Q будет истинным после выполнения оператора S и является спецификацией следующего за S оператора.

Если это утверждение распространить на все операторы программы и если P является спецификацией первого оператора, а Q истинно после окончания программы, то будет доказана правильность всей программы относительно предикатов P и Q. Эту конструкцию можно записать в следующем виде:

$$\{P\}\, S\, \{Q\},$$

где P называется предусловием истинности Q после выполнения программы S.

Доказательство правильности программы заключается в определении, является ли выражение $\{P\}S\{Q\}$ истинным относительно входных спецификаций P, выходных спецификаций Q и операторов S программы. Если $\{P\}S\{Q\}$ истинно, то это означает, что доказана правильность программы S относительно P и Q.

5.1. Аксиомы

Для целей доказательства правильности программ к правилам исчисления предикатов следует добавить правила, необходимые для выполнения последовательности операторов программы. Эти правила, называемые *правилами следствия*, формулируются следующим образом:

1) если $\{P\}S\{Q\}$ и $Q \Rightarrow R$, то $\{P\}S\{R\}$;
2) если $\{Q\}S\{R\}$ и $P \Rightarrow Q$, то $\{P\}S\{R\}$.

Первое правило заключается в следующем: «Если P – предусловие для Q и если $Q \Rightarrow R$ является теоремой исчисления предикатов, то

P – предусловие для R». Таким образом, если P истинно и выполняется S, то R (так же, как и Q) истинно.

Второе правило следствия аналогичное данному. Эти правила следствия можно представить в более формальном виде: числитель выражения является посылкой, а знаменатель – заключением:

1) $\dfrac{\{P\}\,S\,\{Q\},\ Q \Rightarrow R}{\{P\}\,S\,\{R\}}$;

2) $\dfrac{\{Q\}\,S\,\{R\},\ P \Rightarrow Q}{\{P\}\,S\,\{R\}}$.

Правила следствия используются для доказательства сложных элементов программы. Пусть программа представлена в виде структуры (рис. 5.1), где S_i операторы, а P_i предикаты, соответствующие дугам.

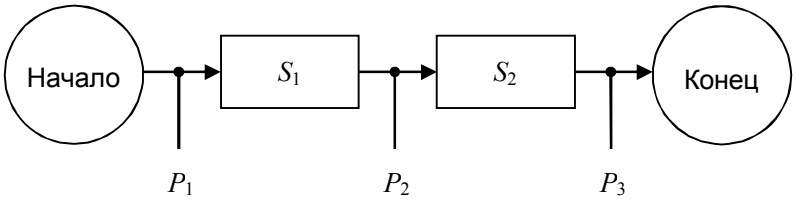


Рис. 5.1 – Структура программы

Предположим, что доказаны следующие утверждения: $\{P_1\}S_1\{P_1'\}$ и $\{P_2\}S_2\{P_2'\}$ для некоторых предикатов P_1' и P_2'. Если можно доказать, что $P_1' \Rightarrow P_2$ и $P_2' \Rightarrow P_3$, то, используя правила следствия, можно доказать, что $\{P_1\}S_1;S_2\{P_3\}$ истинно.

Следует определить, какие операторы программы являются правильными. Предполагается, что программа состоит только из последовательности операторов присвоения и операторов **if** и **do while**.

Так как только оператор присвоения может изменять значение переменных, то для определения правильности оператора присвоения нужно добавить только одну аксиому, она называется *аксиомой присвоения* и формулируется следующим образом:

$$\frac{x = \mathrm{expr}, P(x)}{\{P(\mathrm{expr})\}\ x = \mathrm{expr}\ \{P(x)\}}.$$

В соответствии с этой аксиомой устанавливается, что если P утверждение, содержащее переменную x, истинно, то P *должно быть*

истинно и до выполнения оператора присвоения, если x изменяется expr.

Например, если $P(x)$ является предикатом $x > 0$ и $x = x + 1$ – оператор присвоения, то $P(x + 1)$ является предикатом $x + 1 > 0$ или $x > -1$.

По правилам следствия, если можно доказать, что $Q \Rightarrow P(x + 1)$ для некоторого предиката Q, то справедливо утверждение $\{Q\}x = x + 1\{x > 0\}$ и предикат является предусловием для P.

Осталось определить правильные управляющие структуры. Введем следующие аксиомы.

Аксиома следования:

$$\frac{\{P\}\, S_1\, \{Q\}, \{Q\}\, S_2\, \{R\}}{\{P\}\, S_1; S_2\, \{R\}}.$$

Два оператора программы могут быть объединены. Если после выполнения S_1 предикат Q остается истинным, и Q есть предусловие для оператора S_2, то P является предусловием для операторов $S_1;S_2$.

Аксиома цикла:

$$\frac{\{P \,\&\, B\}\, S\, \{P\}}{\{P\}\, do\ while(B); and; \{\neg B \,\&\, P\}}.$$

Утверждается, что если истинность P не изменяется оператором S (т.е. является инвариантом), то P инвариантно относительно цикла, содержащего S. Кроме того, при выходе из цикла значение выражения B ложно.

Аксиома выбора:

а) $$\frac{\{P \,\&\, B\}\, S_1\, \{Q\}, \{P \,\&\, \neg B\}\, S_2\, \{Q\}}{\{P\}\, if\ B\ then\ S_1; else\ S_2\, \{Q\}};$$

б) $$\frac{\{P \,\&\, B\}\, S_1, P \,\&\, \neg B \Rightarrow Q}{\{P\}\, if\ B\ then\ S_1\, \{Q\}}.$$

Эти аксиомы устанавливают простые соотношения для оператора **if**.

5.2. Правила преобразования данных

Коммутативность:

$$x + y = y + x;$$

$$x \times y = y \times x.$$

Ассоциативность:

$$(x + y) + z = x + (y + z);$$

$$(x \times y) \times z = x \times (y \times z).$$

Дистрибутивность:

$$x \times (y + z) = (x \times y) + (x \times z).$$

Вычитание:

$$x - y + y = x.$$

Обработка констант:

$$x + 0 = x;$$

$$x \times 0 = 0;$$

$$x \times 1 = x.$$

5.3. Доказательства правильности программ

Используя приведенные выше правила и аксиомы, можно осуществлять доказательство правильности программ. Рассмотрим следующую программу:

```
1.  MULTIPLY (R,A,B);
2.  declare X;
3.  declare R;  /*   R=A*B    */
4.  declare A,B; /*   B≥0      */
5.  R=0;
6.  X=B;
7.  do while (X≠0);
8.  R=R+A;
9.  X=X-1;
10. end;
11. end MULTIPLY;
```

Входным утверждением является предикат $B \geq 0$, выходное утверждение $R = A \times B$.

Общая схема доказательства правильности программы:

Строка 5.

- $\{0 = 0 \ \& \ B \geq 0\}\ R = 0\ \{R = 0 \ \& \ B \geq 0\}$ по аксиоме присваивания;
- $\{B \geq 0\}\ R = 0\ \{R = 0 \ \& \ B \geq 0\}$ по правилу следствия в связи с тем, что истинны следующие утверждения:

$$\text{истинно} \Rightarrow 0 = 0;$$

$$B \geq 0 \Rightarrow (B \geq 0 \ \& \ \text{истинно}).$$

Строка 6.

- $\{B = B \;\&\; R = 0 \;\&\; B \geq 0\}\ X = B\ \{X = B \;\&\; R = 0 \;\&\; B \geq 0\}$ по аксиоме присвоения;
- $\{R = 0 \;\&\; B \geq 0\}\ X = B\ \{X = B \;\&\; R = 0 \;\&\; B \geq 0\}$ по аксиоме следования;
- $\{B \geq 0\}\ R = 0;\ X = B\ \{X = B \;\&\; R = 0 \;\&\; B \geq 0\}$ по аксиоме следования.

Таким образом, показано, что если $B \geq 0$ (входное предусловие), то после выполнения строки 6 выражение $\{X = B \;\&\; R = 0 \;\&\; B \geq 0\}$ истинно.

Строки 7-10.

Необходим инвариант для предиката P цикла **do while**. Этот инвариант должен описывать работу, выполняемую циклом. В нашем случае цикл вычисляет произведение; таким образом, инвариантом может быть $R = A \times (B - X)$. В соответствии с аксиомой цикла, если можно показать, что это выражение инвариантно относительно цикла и что оно истинно, когда цикл заканчивается, то возникает следующая ситуация:

- $R = A \times (B - X)$ – инвариант цикла;
- $X = 0$ – выражение внутри цикла ложно.

Таким образом, если выражение $R = A \times (B - X)\ \&\ (X = 0)$ истинно, то истинно и выражение $R = A \times (B - 0)\ \&\ (X = 0)$, а это приводит к истинности соотношения $R = A \times B$, что и требовалось доказать. Следовательно, для того, чтобы завершить доказательство, нужно, чтобы выражение $R = A \times (B - X)$ в строке 7 было истинно, и оно является инвариантом цикла.

Доказательство того, что указанное выражение есть инвариант цикла, проводится следующим образом:

Строка 8.

- $\{R + A = A \times B - A \times X + A\}\ R = R + A\ \{R = A \times B - A \times X + A\}$ по аксиоме присвоения;
- $\{R = A \times B - A \times X\}\ R = R + A\ \{R = A \times B - A \times X + A\}$ по правилам целочисленной арифметики;
- $\{R = A \times (B - X)\}\ R = R + A\ \{R = A \times B - A \times X + A\}$ по правилам дистрибутивности.

Строка 9.

- $\{R = A \times (B - (X - 1))\}\ X = X - 1\ \{R = A \times (B - X)\}$ по аксиоме присвоения;
- $\{R = A \times B - A \times (X - 1)\}\ X = X - 1\ \{R = A \times (B - X)\}$ по правилам целочисленной арифметики.

Объединение строк 8 и 9 в соответствии с аксиомой следования дает выражение $\{R = A\times(B - X)\}\ R = R + A; X = X - 1\ \{R = A\times(B - X)\}$, которое является желаемым инвариантом соотношения.

Теперь следует показать, что инвариантное выражение в строке 7 истинно, а это эквивалентно доказательству с помощью правил следования следующей теоремы: $X = B\ \&\ R = 0\ \&\ B \geq 0$ истинно в строке 6 $\Rightarrow R = A\times(B - X)$.

Остановимся на нескольких важных моментах в связи с доказательством правильности программ.

- Доказательства длинны и сложны даже для простой программы.
- Если оперировать нецелочисленными данными, то формулировка аксиом становится затруднительной. Операции со строками, плавающей точкой, доступ к базам данных вызывают серьезные осложнения при аксиоматическом подходе.
- Даже для относительно простого случая – использования целочисленных данных – существуют трудности. Например, все ЭВМ используют слова фиксированной длины для записи целых чисел. Возможно переполнение разрядной сетки, и аксиомы должны учитывать эти условия.
- Реальные языки программирования имеют встроенные структуры, которые нелегко поддаются проверке на правильность.

Несмотря на отмеченные недостатки, доказательства правильности программ достаточно широко применяются. Способы проверки правильности программ можно использовать при проектировании реальных программ. Для разрабатываемой программы должны быть определены предусловия работы. Даже если условия не доказаны формально, а просто установлены, они оказывают неоценимую помощь в понимании структуры разрабатываемой программы.

Добавление предусловий тесно связано с элементарными программами. Кроме того, предусловия являются базой для разработки тестов программ. Все предусловия могут быть закодированы как операторы **if**, и выполняться как часть теста.

Контрольные вопросы

1) Правильность программ.
2) Правила следствия.
3) Аксиома присвоения.
4) Аксиома следования.
5) Аксиома цикла.
6) Аксиома выбора.

7) Правила целочисленной арифметики – коммутативность, ассоциативность, дистрибутивность, вычитание, обработка констант.
8) Доказательство правильности программ.

6. ТЕСТИРОВАНИЕ

6.1. Психология и экономика тестирования программ

Тестирование, как объект изучения, может рассматриваться с различных, чисто технических, точек зрения. Однако наиболее важными при изучении тестирования представляются вопросы его экономики и психологии разработчика. Иными словами, достоверность тестирования программы, в первую очередь, определяется тем, кто будет ее тестировать и каков его образ мышления и уже затем определенными технологическими аспектами. Поэтому, прежде чем перейти техническим проблемам, мы остановимся на этих вопросах.

Поначалу может показаться тривиальным жизненно важный вопрос определения термина «тестирование». Необходимость обсуждения этого термина связана с тем, что большинство специалистов используют его неверно, а это, в свою очередь, приводит к плохому тестированию, Таковы, например, следующие определения: «Тестирование представляет собой процесс, демонстрирующий отсутствие ошибок в программе», «цель тестирования – показать, что программа корректно исполняет предусмотренные функции», «тестирование – это процесс, позволяющий убедиться в том, что программа выполняет свое назначение».

Эти определения описывают нечто *противоположное* тому, что следует понимать под тестированием, поэтому они неверны. Оставив на время определения, предположим, что если мы тестируем программу, то нам нужно добавить к ней некоторую новую стоимость (т. е. тестирование стоит денег и нам желательно возвратить затраченную сумму путем увеличения стоимости программы). Увеличение стоимости означает повышение качества или возрастание надежности программы. Последнее связано с обнаружением и удалением из нее ошибок. Следовательно, программа тестируется не для того, что бы показать, что она работает, а скорее наоборот – тестирование начинается с предположения, что в ней есть ошибки (это предположение справедливо практически для любой программы), а затем уже обнаруживается их максимально возможное число. Таким образом, сформулируем наиболее приемлемое определение:

Тестирование – это процесс исполнения программы с целью обнаружения ошибок.

Пока все наши рассуждения могут показаться тонкой игрой семантики, однако практикой установлено, что именно ими в значительной мере определяется успех тестирования. Дело в том, что верный выбор цели дает важный психологический эффект поскольку для че-

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

Из приведенного определения тестирования вытекает несколько следствий. Например, одно из них состоит в том, что тестирование – процесс *деструктивный* (т. е. обратный созидательному, конструктивному). Именно этим и объясняется, почему многие считают его трудным. Большинство людей склонны к конструктивному процессу созидания объектов и в меньшей степени – к деструктивному процессу разделения на части. Из определения следует также то, как нужно строить набор тестовых данных и кто должен (а кто не должен) тестировать данную программу.

Для усиления определения тестирования проанализируем два понятия «удачный» и «неудачный» и, в частности, их использование руководителями проектов при оценке результатов тестирования. Большинство руководителей проектов называют тестовый прогон неудачным, если обнаружена ошибка, и, наоборот, удачным, если он прошел без ошибок. Чаще всего это является следствием ошибочного понимания термина «тестирование», так как, по существу, слово «удачный» означает «результативный», а слово «неудачный» – «нежелательный», «нерезультативный». Но если тест не обнаружил ошибки, его выполнение связано с потерей времени и денег, и термин «удачный» никак не может быть применен к нему.

Тестовый прогон, приведший к обнаружению ошибки, нельзя назвать неудачным хотя бы потому, что, как отмечалось выше, это целесообразное вложение капитала. Отсюда следует, что в слова «удачный» и «неудачный» необходимо вкладывать смысл, обратный общепринятому. Поэтому в дальнейшем будем называть тестовый прогон удачным, если в процессе его выполнения обнаружена ошибка и «неудачным», если корректный результат.

Проведем аналогию с посещением больным врача. Если рекомендованное врачом лабораторное исследование не обнаружило причины болезни, не назовем же мы такое исследование удачным – оно неудачно: ведь деньги пациент заплатил, а он все так же болен. Если же исследование показало, что у больного язва желудка, то оно является удачным, поскольку врач может прописать необходимый курс лечения. Следовательно, медики используют эти термины в нужном

нам смысле (аналогия здесь, конечно, заключается в том, что программа, которую предстоит тестировать, подобна больному пациенту).

Определения типа «тестирование представляет собой процесс демонстрации отсутствия ошибок» порождают еще одну проблему: они ставят цель, которая не может быть достигнута ни для одной программы, даже весьма тривиальной. Результаты психологических исследований показывают, что если перед человеком ставится невыполнимая задача, то он работает хуже. Иными словами, определение тестирования как процесса обнаружения ошибок переводит его в разряд решаемых задач и, таким образом, преодолевается психологическая трудность.

Другая проблема возникает в том случае, когда для тестирования используется следующее определение:

«Тестирование – это процесс, позволяющий убедиться в том, что программа выполняет свое назначение», поскольку программа, удовлетворяющая данному определению, может содержать ошибки. Если программа не делает того, что от нее требуется, то ясно, что она содержит ошибки. Однако ошибки могут быть и тогда, когда она *делает, и что от нее не требуется*.

Подводя итог, можно сказать, что тестирование представляется деструктивным процессом попыток обнаружения ошибок в программе (наличие которых предполагается). Набор тестов, способствующий обнаружению ошибки, считается удачным. Естественно, в конечном счете, каждый с помощью тестирования хочет добиться определенной степени уверенности в том, что его программа соответствует своему назначению и не делает того, для чего она не предназначена, но лучшим средством для достижения этой цели является непосредственный поиск ошибок. Допустим, кто-то обращается к вам *с* заявлением: «Моя программа великолепна» (т.е. не содержит ошибок). Лучший способ доказать справедливость подобного утверждения – попытаться его опровергнуть, обнаружить неточности, нежели просто согласиться с тем, что программа на определенном наборе входных данных работает корректно.

6.2. Экономика тестирования

Дав такое определение тестированию, необходимо на следующем шаге рассмотреть возможность создания теста, обнаруживающего *все* ошибки программы. Покажем, что ответ будет отрицательным даже для самых тривиальных программ. В общем случае, невозможно обнаружить все ошибки программы. А это, в свою очередь, порождает

экономические проблемы, задачи, связанные с функциями человека в процессе отладки, способы построения тестов.

6.2.1. Тестирование программы как черного ящика

Одним из способов изучения поставленного вопроса является исследование стратегии тестирования, называемой стратегией черного ящика, *тестированием с управлением по данным* или *тестированием с управлением по входу-выходу*. При использовании этой стратегии программа рассматривается как черный ящик. Иными словами, такое тестирование имеет целью выяснение обстоятельств, в которых поведение программы не соответствует ее спецификации.

При таком подходе обнаружение всех ошибок в программе является критерием *исчерпывающего входного тестирования*. Последнее может быть достигнуто, если в качестве тестовых наборов использовать все возможные наборы входных данных. Поэтому для тестирования даже небольшой программы требуется бесконечное число тестов.

Если такое испытание представляется сложным, то еще сложнее создать исчерпывающий тест для большой программы. Образно говоря, число тестов можно оценить «числом, большим, чем бесконечность».

Из изложенного следует, что построение исчерпывающего входного теста невозможно. Это подтверждается двумя аргументами: во-первых, нельзя создать тест, гарантирующий отсутствие ошибок; во-вторых, разработка таких тестов противоречит экономическим требованиям. Поскольку исчерпывающее тестирование исключается, нашей целью должна стать максимизация результативности капиталовложений в тестирование (иными словами, максимизация числа ошибок, обнаруживаемых одним тестом). Для этого мы можем рассматривать внутреннюю структуру программы и делать некоторые разумные, но, конечно, не обладающие полной гарантией достоверности предположения.

6.2.2. Тестирование программы как белого ящика

Стратегия *белого ящика,* или стратегия тестирования, *управляемого логикой программы*, позволяет исследовать внутреннюю структуру программы. В этом случае тестирующий получает тестовые данные путем анализа логики программы (к сожалению, здесь часто не используется спецификация программы).

Сравним способ построения тестов при данной стратегии с исчерпывающим входным тестированием стратегии черного ящика. Непосвященному может показаться, что достаточно построить такой

набор тестов, в котором каждый оператор исполняется хотя бы один раз; нетрудно показать, что это неверно. Не вдаваясь детали, укажем лишь, что исчерпывающему входному тестированию может быть поставлено в соответствие *исчерпывающее тестирование маршрутов.* Подразумевается, что программа проверена полностью, если с помощью тестов удается осуществить выполнение этой программы по всем возможным маршрутам ее потока (графа) передач управления.

Последнее утверждение имеет два слабых пункта. Один из них состоит в том, что число не повторяющих друг друга маршрутов в программе – астрономическое. Чтобы убедиться в этом, рассмотрим представленный на рис. 2.1 граф передач управления простейшей программы. Каждая вершина или кружок обозначают участок программы, содержащий последовательность линейных операторов, которая может заканчиваться оператором ветвления. Дуги, оканчивающиеся стрелками, соответствуют передачам управления. По-видимому, граф описывает программу из 10-20 операторов, включая цикл DO, который исполняется не менее 20 раз. Внутри цикла имеется несколько операторов IF. Для того, чтобы определять число неповторяющихся маршрутов при исполнении программы, подсчитаем число неповторяющихся маршрутов из точки A в B в предположении, что все приказы взаимно независимы. Это число вычисляется как сумма $5^{20} + 5^{19} + \ldots + 5^{1} =$ 100 триллионов, где 5 – число путей внутри цикла. Приведем такой пример: если допустить, что на составление каждого теста мы тратим пять минут, то для построения набора тестов нам потребуется примерно один миллиард лет.

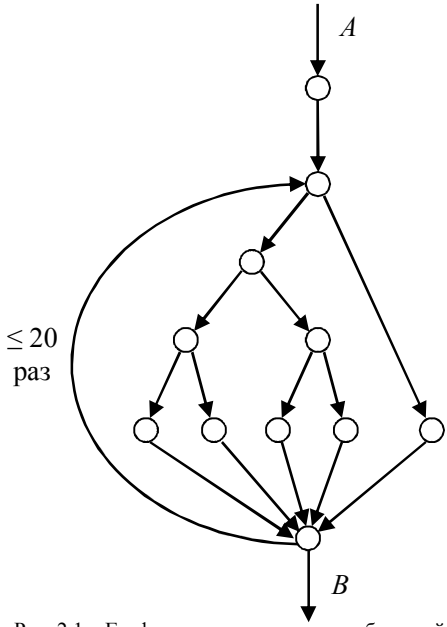


Рис. 2.1 – Граф передач управления небольшой программы

Второй слабый пункт утверждения заключается в том, что, хотя исчерпывающее тестирование маршрутов является полным тестом, и хотя каждый маршрут программы может быть проверен, сама программа может содержать ошибки. *Это* объясняется следующим образам, Во-первых, исчерпывающее тестирование маршрутов не может дать гарантии того, что программа соответствует описанию. Например, вместо требуемой программы сортировки по возрастанию случайно была написана программа сортировки по убыванию. В этом случае ценность тестирования маршрутов невелика, поскольку после тестирования в программе окажется одна ошибка, т.е., программа неверна. Во-вторых, программа может быть неверной в силу того, что пропущены некоторые маршруты. Исчерпывающее тестирование маршрутов не обнаружит их отсутствия. В-третьих, исчерпывающее тестирование маршрутов не может обнаружить ошибок, *появление которых зависит от обрабатываемых данных*. Существует множество примеров таких ошибок. Приведем один из них. Допустим, в программе необходимо выполнить сравнение двух чисел на сходимость, т. е. определить, является ли разность между двумя числами меньше

предварительно определенного числа. Может быть написано выражение

IF ((A – B) < EPSILON)…

Безусловно, оно содержит ошибку, поскольку необходимо выполнить сравнение абсолютных величин. Однако обнаружение этой ошибки зависит от значений, использованных для A и B, и ошибка не обязательно будет обнаружена просто путем исполнения каждого маршрута.

В заключение отметим, что, хотя исчерпывающее входное тестирование предпочтительнее исчерпывающего тестирования маршрутов, ни то, ни другое не могут стать полезными стратегиями, потому что оба они нереализуемы. Возможно, поэтому реальным путем, который позволит создать хорошую, но, конечно, не абсолютную стратегию, является сочетание тестирования программы и как черного, и как белого ящиков. Этот вопрос обсуждается в п. 6.4.

6.2.3. Принципы тестирования

Сформулируем основные принципы тестирования, используя главную предпосылку настоящего раздела о том, что наиболее важными в тестировании программ являются вопросы психологии. Эти принципы интересны тем, что в основном они интуитивно ясны, но, в то же время, на них часто не обращают должного внимания.

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

Нарушение этого очевидного принципа представляет одну из наиболее распространенных ошибок. Ошибочные, но правдоподобные результаты могут быть признаны правильными, если результаты теста не были заранее определены. Здесь мы сталкиваемся с явлением психологии: мы видим то, что мы хотим увидеть. Другими словами, несмотря на то, что тестирование по определению – деструктивный процесс, есть подсознательное желание видеть корректный результат. Один из способов борьбы с этим состоит в поощрении детального анализа выходных переменных заранее при разработке теста. Поэтому тест должен включать две компоненты: описание входных данных и описание точного и корректного результата, соответствующего набору входных данных.

Необходимость этого подчеркивал логик Копи: «Проблема может быть охарактеризована как факт или группа фактов, которые не имеют приемлемого объяснения, кажутся необычными или которые не удается подогнать под наши представления или предположения. Очевидно, что если *что-нибудь* подвергается сомнению, то об этом долж-

на иметься какая-то предварительная информация. Если нет предположений, то не может быть и неожиданных результатов».

Следует избегать тестирования программы ее автором.

Этот принцип следует из обсуждавшихся ранее положений, которые определяют тестирование как деструктивный процесс. После выполнения конструктивной части, при проектировании и написании программы трудно быстро (в течение одного дня) перестроиться на деструктивный образ мышления. Многие, кому приходилось самому делать дома ремонт, знают, что процесс обрывания старых обоев (деструктивный процесс) не легок, но он просто невыносим, если не кто-то другой, а вы сами первоначально их наклеивали. Вот так же и большинство программистов не могут эффективно тестировать свои программы, потому что им трудно демонстрировать собственные ошибки.

В дополнение к этой психологической проблеме следует отметить еще одну, не менее важную: программа может содержать ошибки, связанные с неверным пониманием постановки или описания задачи программистом. Тогда существует вероятность, что к тестированию программист приступит с таким же недопониманием своей задачи.

Тестирование можно уподобить работе корректора или рецензента над статьей или книгой. Многие авторы представляют себе трудности, связанные с редактированием собственной рукописи. Очевидно, что обнаружение недостатков в своей деятельности противоречит человеческой психологии.

Отсюда вовсе не следует, что программист *не может* тестировать свою программу. Здесь лишь делается вывод о том, что тестирование является более эффективным, если оно выполняется кем-либо другим. Заметим, что все наши рассуждения не относятся к отладке, т.е. к исправлению уже известных ошибок. Эта работа эффективнее выполняется самим автором программы.

Программирующая организация не должна сама тестировать разработанные ею программы.

Здесь можно привести те же аргументы, что и в предыдущем случае. Во многих смыслах проектирующая или программирующая организация подобна живому организму с его психологическими проблемами. Работа программирующей организации или ее руководителя оценивается по их способности производить программы в течение заданного времени и определенной стоимости. Одна из причин такой системы оценок состоит в том, что временные и стоимостные показатели легко измерить, но в то же время чрезвычайно трудно количественно оценить надежность программы. Именно поэтому в процессе

тестирования программирующей организации трудно быть объективной, поскольку тестирование в соответствии с данным определением может быть рассмотрено как средство уменьшения вероятности соответствия программы заданным временным и стоимостным параметрам.

Как и ранее, из изложенного не следует, что программирующая организация не может найти свои ошибки; тестирование в определенной степени может пройти успешно. Мы утверждаем здесь лишь то, что экономически более целесообразно выполнение тестирования каким-либо объективным, независимым подразделением.

Необходимо досконально изучать результата ты применения каждого теста.

По всей вероятности, это наиболее очевидный принцип, но и ему часто не уделяется должное внимание. В проведенных экспериментах многие испытуемые не смогли обнаружить определенные ошибки, хотя их признаки были совершенно явными в выходных листингах. Представляется достоверным, что значительная часть всех обнаруженных в конечном итоге ошибок могла быть выявлена в результате самых первых тестовых прогонов, но они были пропущены вследствие недостаточно тщательного анализа результатов первого тестового прогона.

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

При тестировании программ имеется естественная тенденция концентрировать внимание на правильных и предусмотренных входных условиях, а неправильным и непредусмотренным входным данным не придавать значения. Множество ошибок можно также обнаружить, если использовать программу новым, не предусмотренным ранее способом. Вполне вероятно, что тесты, представляющие неверные и неправильные входные данные, обладают большей обнаруживающей способностью, чем тесты, соответствующие корректным входным данным.

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

Это логически просто вытекает из предыдущего принципа. Необходимо проверить программу на нежелательные побочные эффекты. Например, программа расчета зарплаты, которая производит правильные платежные чеки, окажется неверной, если она произведет

лишние чеки для работающих или дважды запишет первую запись в список личного состава.

Не следует выбрасывать тесты, даже если программа уже не нужна.

Эта проблема наиболее часто возникает при использовании интерактивных систем отладки. Обычно тестирующий сидит за терминалом, на лету придумывает тесты и запускает программу на выполнение. При такой практике работы после применения тесты пропадают. После внесения изменений или исправления ошибок необходимо повторять тестирование, тогда приходится заново изобретать тесты. Как правило, этого стараются избегать, поскольку повторное создание тестов требует значительной работы. В результате повторное тестирование бывает менее тщательным, чем первоначальное, т.е. если модификация затронула функциональную часть программы и при этом была допущена ошибка, то она зачастую может остаться необнаруженной.

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

Такую ошибку обычно допускают руководители проекта, использующие неверное определение тестирования как процесса демонстрации отсутствия ошибок в программе, корректного функционирования программы.

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

Этот принцип, не согласующийся с интуитивным представлением, иллюстрируется рис. 2.2.

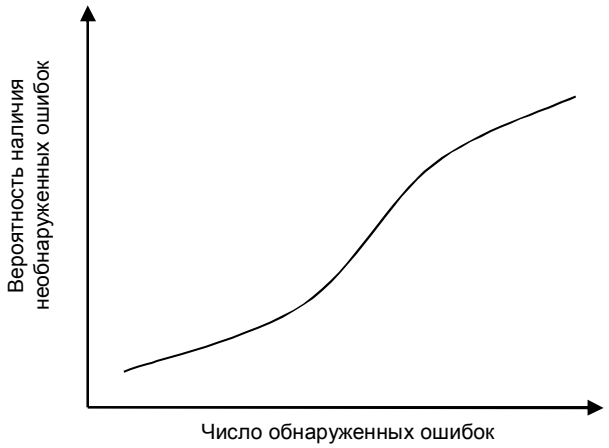


Рис. 2.2 – Неожиданное соотношение числа оставшихся и числа обнаруженных ошибок

На первый взгляд он лишен смысла, но, тем не менее, подтверждается многими программами. Например, допустим, что некоторая программа состоит из модулей А и В. К определенному сроку в модуле А обнаружено пять ошибок, а в модуле В – только одна, причем модуль А не подвергался более тщательному тестированию.

Тогда из рассматриваемого принципа следует, что вероятность необнаруженных ошибок в модуле А больше, чем в модуле В. Справедливость этого принципа подтверждается еще и тем, что для ошибок свойственно располагаться в программе в виде неких скоплении, хотя данное явление пока никем еще не объяснено. В качестве примера можно рассмотреть операционные системы IBM S/370. В одной из версий операционной системы 47% ошибок, обнаруженных пользователями, приходилось на 4% модулей системы.

Преимущество рассматриваемого принципа заключается в том, что он позволяет ввести обратную связь в процесс тестирования. Если в какой-нибудь части программы обнаружено больше ошибок, чем в других, то на ее тестирование должны быть направлены дополнительные усилия.

Тестирование – процесс творческий.

Вполне вероятно, что для тестирования большой программы требуется больший творческий потенциал, чем для ее проектирования. Выше было показано, что нельзя дать гарантию построения теста, обнаруживающего все ошибки. В дальнейшем здесь будут обсуждаться методы построения хороших наборов тестов, но применение этих методов должно быть творческим.

Чтобы подчеркнуть некоторые мысли, высказанные в настоящем разделе, приведем еще раз три наиболее важных принципа тестирования.

Тестирование – это процесс выполнения программ с целью обнаружения ошибок.

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

Удачным считается тест, который обнаруживает еще не выявленную ошибку.

6.3. Ручное тестирование

В течение многих лет большинство программистов были убеждены в том, что программы пишутся исключительно для выполнения их на машине и не предназначены для чтения человеком, а единственным способом тестирования программы является ее исполнение на

ЭВМ. Это мнение стало изменяться в начале 70-х годов, а значительной степени – благодаря книге Вейнберга «Психология программирования для ЭВМ». Вейнберг доказал, что программы должны быть удобочитаемыми и что их просмотр должен быть эффективным процессом обнаружения ошибок.

По этой причине, прежде чем перейти к обсуждению традиционных методов тестирования, основанных на применении ЭВМ, рассмотрим процесс тестирования без применения ЭВМ («ручного тестирования»), являющейся темой настоящего раздела. Эксперименты показали, что методы ручного тестирования достаточно эффективны с точки зрения нахождения ошибок, так что один или несколько из них должны использоваться в каждом программном проекте. Описанные здесь методы предназначены для периода разработки, когда программа закодирована, но тестирование на ЭВМ еще не началось. Аналогичные методы могут быть получены и применены на более ранних этапах процесса создания программ (т.е. в конце каждого этапа проектирования), но рассмотрение этого вопроса выходит за рамки данного пособия.

Следует заметить, что из-за неформальной природы методов ручного тестирования (неформальной с точки зрения других, более формальных методов, таких, как математическое доказательство корректности программ) первой реакцией часто является скептицизм, ощущение того, что простые и неформальные методы не могут быть полезными. Однако их использование показало, что они не «уводят в сторону». Скорее эти методы способствуют существенному увеличению производительности и повышению надежности программы. Во-первых, они обычно позволяют раньше обнаружить ошибки, уменьшить стоимость исправления последних и увеличить вероятность того, что корректировка произведена правильно. Во-вторых, психология программистов, по-видимому, изменяется, когда начинается тестирование на ЭВМ. Возрастает внутреннее напряжение и появляется тенденция «исправлять ошибки так быстро, как только это возможно». В результате программисты допускают больше промахов при корректировке ошибок, уже найденных во время тестирования на ЭВМ, чем при корректировке ошибок, найденных на более ранних этапах.

6.3.1. Инспекции и сквозные просмотры

Инспекции исходного текста и сквозные просмотры являются основными методами ручного тестирования. Так как эти два метода имеют много общего, они рассматриваются здесь совместно. Различия между ними обсуждаются в последующих разделах.

Инспекции и сквозные просмотры включают в себя чтение или визуальную проверку программы группой лиц. Эти методы развиты из идей Вейнберга. Оба метода предполагают некоторую подготовительную работу. Завершающим этапом является «обмен мнениями» – собрание, проводимое участниками проверки. Цель такого собрания – нахождение ошибок, но не их устранение (т.е. тестирование, а не отладка).

Инспекции и сквозные просмотры широко практикуются в настоящее время, но причины их успеха до сих пор еще недостаточно выяснены. Не удивительно, что они связаны с некоторыми принципами п. 6.2. Заметим, что данный процесс выполняется группой лиц (оптимально – три-четыре человека), лишь один из которых является автором программы. Следовательно, программа, по существу, тестируется не автором, а другими людьми, которые руководствуются изложенными выше принципами, обычно не эффективными при тестировании собственной программы. Фактически, «инспекция» и «сквозной просмотр» – просто новые названия старого метода «проверки за столом» (состоящего в том, что программист просматривает свою программу перед ее тестированием), однако они гораздо более эффективны опять-таки по той же причине: в процессе участвует не только автор программы, но и другие лица. Результатом использования этих методов, по-видимому, также является более низкая стоимость отладки (исправления ошибок), так как во время поиска ошибок обычно точно определяют их природу. Кроме того, с помощью данных методов обнаруживают группы ошибок, что позволяет в дальнейшем корректировать несколько ошибок сразу. С другой стороны, при тестировании на ЭВМ обычно выявляют только *симптомы* ошибок (например, программа не закончилась или напечатала бессмысленный результат), а сами они определяются поодиночке.

Эксперименты по применению этих методов показали, что с их помощью для типичных программ можно находить от 30 до 70% ошибок логического проектирования и кодирования (однако, эти методы не эффективны при определении ошибок проектирования «высокого уровня», например, сделанных в процессе анализа требований). Так, экспериментально установлено, что при проведении инспекций и сквозных просмотров определяются в среднем 38% общего числа ошибок в учебных программах. При использовании инспекций исходного текста в фирме IBM эффективность обнаружения ошибок составляет 80% (в данном случае имеется в виду не 80% общего числа ошибок, поскольку, как отмечалось в п. 6.2, общее число ошибок в программе никогда не известно, а 80% всех ошибок, найденных к моменту

окончания процесса тестирования). Конечно, можно критиковать эту статистику в предположении, что ручные методы тестирования позволяют находить только «легкие» ошибки (те, которые можно просто найти при тестировании на ЭВМ), а трудные, незаметные или необычные ошибки можно обнаружить только при тестировании на машине. Однако проведенные исследования показали, что подобная критика является необоснованной. Исследователи сделали также вывод о том, что при нахождении определенных типов ошибок ручное тестирование *более* эффективно, чем тестирование на ЭВМ, *в то время как для других типов ошибок верно противоположное утверждение*. Подразумевается, что инспекции, сквозные просмотры и тестирование, основанное на использовании ЭВМ, дополняют друг друга; эффективность обнаружения ошибок уменьшится, если тот или иной из этих подходов не будет использован.

Наконец, хотя эти методы весьма важны при тестировании новых программ, они представляют не меньшую ценность при тестировании модифицированных программ. Опыт показал, что в случае модификации существующих программ вносится большее число ошибок (измеряемое числом ошибок на вновь написанные операторы), чем при написании новой программы. Следовательно, модифицированные программы также должны быть подвергнуты тестированию с применением данных методов.

6.3.2. Инспекции исходного текста

Инспекции исходного текста представляют собой набор процедур и приемов обнаружения ошибок при изучении (чтении) текста группой специалистов. При рассмотрении инспекций исходного текста внимание будет сосредоточено в основных процедурах, формах выполнения и т.д.; после краткого изложения основной процедуры мы изучим существующие методы обнаружения ошибок.

Инспектирующая группа включает обычно четыре человека, один из которых выполняет функции председателя. Председатель должен быть компетентным программистом, но не автором программы; он не должен быть знаком с ее деталями. В обязанности председателя входят подготовка материалов для заседаний инспектирующей группы и составление графика их проведения, ведение заседаний, регистрация всех найденных ошибок и принятие мер по их последующему исправлению. Председателя можно сравнить с инженером отдела технического контроля. Членами группы являются автор программы, проектировщик (если он не программист) и специалист по тестированию.

Общая процедура заключается в следующем. Председатель заранее (например, за несколько дней) раздает листинг программы и проектную спецификацию остальным членам группы. Они знакомятся с материалами, предшествующими заседанию. Инспекционное заседание разбивается на две части:

1) Программиста просят рассказать о логике работы программы. Во время беседы возникают вопросы, преследующие цель обнаружения ошибки. Практика показала, что даже только чтение своей программы слушателям представляется эффективным методом обнаружения ошибок и многие ошибки находит сам программист, а не другие члены группы.
2) Программа анализируется по списку вопросов для выявления исторически сложившихся общих ошибок программирования (такой список будет рассмотрен в следующем разделе).

Председатель является ответственным за обеспечение результативности дискуссии. Ее участники должны сосредоточить свое внимание на нахождении ошибок, а не на их корректировке (корректировка ошибок выполняется программистом после инспекционного заседания).

По окончании заседания программисту передается список найденных ошибок. Если список включает много ошибок или если эти ошибки требуют внесения значительных изменений, председателем может быть принято решение о проведении после корректировки повторной инспекции программы. Список анализируется, и ошибки распределяются по категориям, что позволяет совершенствовать его с целью повышения эффективности будущих инспекций.

В большинстве примеров описания процесса инспектирования утверждается, что во время инспекционного заседания ошибки не должны корректироваться. Однако существует и другая точка зрения: вместо того, чтобы сначала сосредоточиться на основных проблемах проектирования, необходимо решить второстепенные вопросы. Два или три человека, включая разработчика программы, должны внести очевидные исправления в проект с тем, чтобы впоследствии решить главные задачи. Однако обсуждение второстепенных вопросов сконцентрирует внимание группы на частной области проектирования. Во время обсуждения наилучшего способа внесения изменений в проект кто-либо из членов группы может заметить еще одну проблему. Теперь группе придется рассматривать две проблемы по отношению к одним и тем же аспектам проектирования, объяснения будут полными и быстрыми. В течение нескольких минут целая область проекта может быть полностью исследована, и любые проблемы станут очевид-

ными. Как упоминалось выше, многие важные проблемы, возникавшие во время обзоров блок-схем, были решены в результате многократных безуспешных попыток решить вопросы, которые на первый взгляд казались «тривиальными».

Время и место проведения инспекции должны быть спланированы так, чтобы избежать любых прерываний инспекционного заседания. Его оптимальная продолжительность, по-видимому, лежит в пределах от 90 до 120 мин. Так как это заседание является экспериментом, требующим умственного напряжения, увеличение его продолжительности ведет к снижению продуктивности. Большинство инспекций происходит при скорости, равной приблизительно 150 операторам в час. При этом подразумевается, что большие программы должны рассматриваться за несколько инспекций, каждая из которых может быть связана с одним или несколькими модулями или подпрограммами.

Для того, чтобы инспекция была эффективной, должны быть установлены соответствующие отношения. Если программист воспринимает инспекцию как акт, направленный лично против него, и, следовательно, занимает оборонительную позицию, процесс инспектирования не будет эффективным. Программист должен подходить к нему с менее эгоистической позиции; он должен рассматривать инспекцию в позитивном и конструктивном свете. Объективно инспекция является процессом нахождения ошибок в программе и, таким образом, улучшает качество его работы. По этой причине, как правило, рекомендуется результаты инспекции считать конфиденциальными материалами, доступными только участникам заседания. В частности, использование результатов инспекции руководством может нанести ущерб целям этого процесса.

Процесс инспектирования, в дополнение к своему основному назначению, заключающемуся в нахождении ошибок, выполняет еще ряд полезных функций. Кроме того, что результаты инспекции позволяют программисту увидеть сделанные им ошибки и способствуют его обучению на собственных ошибках, он обычно получает возможность оценить свой стиль программирования и выбор алгоритмов и методов тестирования. Остальные участники также приобретают опыт, рассматривая ошибки и стиль программирования других программистов.

Наконец, инспекция является способом раннего выявления наиболее склонных к ошибкам частей программы, позволяющим сконцентрировать внимание на этих частях в процессе выполнения тестирования на ЭВМ (один из принципов тестирования п. 6.2).

6.3.3. Список вопросов для выявления ошибок при инспекции

Важной частью процесса инспектирования является проверка программы на наличие общих ошибок с помощью списка вопросов для выявления ошибок. Концентрация внимания в предлагаемом списке на рассмотрении стиля, а не ошибок (вопросы типа «Являются ли комментарии точными и информативными?» и «Располагаются ли операторы THEN/ELSE и DO/END по одной вертикали друг под другом?») представляется неудачной. Также неудачным представляется и наличие неопределенности в списке, уменьшающее его полезность (вопросы типа «Соответствует ли текст программы требованиям, выдвигаемым при проектировании?»). Список, приведенный в данном разделе, был составлен после многолетнего изучения ошибок программного обеспечения. В значительной мере он является независимым от языка; это означает, что большинство ошибок встречается в любом языке программирования.

6.3.3.1. Ошибки обращения к данным

1. Существуют ли обращения к переменным, значения которым не присвоены или не инициализированы? Наличие переменных с не установленными значениями – наиболее часто встречающаяся программная ошибка, она возникает при различных обстоятельствах. Для каждого обращения к единице данных (например, к переменной, элементу массива, полю в структуре) попытайтесь неформально «доказать», что ей присвоено значение в проверяемой точке.

2. Не выходит ли значение каждого из индексов за границы, определенные для соответствующего измерения при всех обращениях к массиву?

3. Принимает ли каждый индекс целые значения при всех обращениях к массиву? Нецелые индексы не обязательно являются ошибкой для всех языков программирования, но представляют практическую опасность.

4. Для всех обращений с помощью указателей или переменных-ссылок память, к которой производится обращение, уже распределена? Наличие переменных-ссылок представляет собой ошибку типа «подвешенного обращения». Она возникает в ситуациях, когда время жизни указателя больше, чем время жизни памяти, к которой производится обращение. Например, к такому результату приводит ситуация, когда указатель задает локальную переменную в теле процедуры, значение указателя присваивается выходному параметру или глобальной переменной, процедура завершается (освобождая адресуемую память), а программа затем пытается использовать значение указателя. Как и

при поиске ошибок первых трех типов, попытайтесь неформально доказать, что для каждого обращения, использующего переменную-указатель, адресуемая память существует.

5. Если одна и та же область памяти имеет несколько псевдонимов (имен) с различными атрибутами, то имеют ли значения данных в этой области корректные атрибуты при обращении по одному из этих псевдонимов? Ошибки типа некорректных атрибутов у псевдонимов могут возникнуть при использовании атрибута DEFINED или базированной памяти в PL/1, оператора EQUIVALENCE в Фортране, глагола REDEFINES в Коболе, записей с вариантами в Паскале или объединений (UNION) в языке Си. В качестве примера можно привести программу на языке Си, содержащую вещественную переменную *A* и целую переменную B. Обе величины размещены на одном и том же месте памяти (т.е. помещены в одно и то же объединение). Если программа записывает величину *A,* а обращается к переменной *B,* то, вероятно, произойдет ошибка, поскольку машина будет использовать битовое представление числа с плавающей точкой в данной области памяти как целое.

6. Отличаются ли типы или атрибуты переменных величин от тех, которые предполагались компилятором? Это может произойти в том случае, когда программа на PL/1 или Коболе считывает записи из памяти и обращается к ним как к структурам, но физическое представление записей отлично от описания структуры. Или программа на языке Си, допускающем произвольные преобразования типов, содержит переменную-указатель на структуру типа T_1, инициализированную указателем на структуру типа T_2 (не наследуемую от T_1).

7. Существуют ли явные или неявные проблемы адресации, если в машине будут использованы единицы распределения памяти, меньшие, чем единицы адресации памяти? Например, в PL/1 в системе IBM S/370 битовые строки фиксированной длины не обязательно начинаются с границ байтов, а адресами могут быть только границы байтов. Аналогично, в языке Си размер отдельных полей в структуре может задаваться в битах. Если программа вычисляет адрес битового поля и впоследствии обращается к нему по этому адресу, то может произойти ошибочное обращение к памяти. Такое же положение может сложиться при передаче подпрограмме (процедуре) битового поля в качестве аргумента.

8. Если используются указатели или переменные-ссылки, то имеет ли адресуемая память атрибуты, предполагаемые компилятором? Примером несоответствия атрибутов может служить случай, ко-

гда указатель PL/1, по которому базируется структура данных, имеет в качестве значения адрес другой структуры.

9. Если к структуре данных обращаются из нескольких процедур или подпрограмм, то определена ли эта структура одинаково в каждой процедуре?

10. Не превышены ли границы строки при индексации в ней?

11. Существуют ли какие-нибудь другие ошибки в операциях с индексацией или при обращении к массивам по индексу?

6.3.3.2. Ошибки описания данных

1. Все ли переменные описаны явно? Отсутствие явного описания не обязательно является ошибкой, но служит общим источником беспокойства. Так, если в подпрограмме на Фортране используется элемент массива и отсутствует его описание (например, в операторе DIMENSION), то обращение к массиву (например, X = A(1)), будет интерпретироваться как вызов функции. Последнее приведет к тому, что машина будет пытаться обработать массив как программу. Если отсутствует явное описание переменной во внутренней процедуре или блоке, следует ли понимать это так, что описание данной переменной совпадает с описанием во внешнем блоке?

2. Если не все атрибуты переменной явно присутствуют в описании, то понятно ли их отсутствие? Например, отсутствие атрибутов, считающееся общепринятым в PL/1, часто является источником неожиданных осложнений.

3. Если начальные значения присваиваются переменным в операторах описания, то правильно ли инициализируются эти значения? Во многих языках программирования присвоение начальных значений массивам и строкам представляется довольно сложным и, следовательно, является возможным источником ошибок.

4. Правильно ли для каждой переменной определены длина, тип и класс памяти (например, STATIC, AUTOMATIC, BASED или CONTROLLED в PL/1; AUTO, CONST, VOLATILE, REGISTER, STATIC в Си и т.д.)?

5. Согласуется ли инициализация переменной с ее типом памяти? Например, если в подпрограмме на Фортране необходимо устанавливать начальные значения переменной каждый раз при вызове подпрограммы, переменная должна быть инициализирована в операторе описания, отличном от оператора DATA. Если в PL/1 описывается инициализация величины и начальное значение необходимо устанавливать каждый раз при вызове процедуры, то для этой переменной должен быть определен класс памяти AUTOMAT1C, а не STATIC.

6. Есть ли переменные со сходными именами (например, VOLT и VOLTS)? Наличие сходных имен не обязательно является ошибкой, но служит признаком того, что имена могут быть перепутаны где-нибудь внутри программы.

6.3.3.3. Ошибки вычислений

1. Имеются ли вычисления, использующие переменные недопустимых (например, неарифметических) типов данных?

2. Существуют ли вычисления, использующие данные разного вида? Например, сложение переменной с плавающей точкой, и целой переменной. Такие случаи не обязательно являются ошибочными, но они должны быть тщательно проверены для обеспечения гарантии того, что правила преобразования, принятые в языке, понятны. Это особенно важно для языков со сложными правилами преобразования (например, для PL/1, Си). Например, следующий фрагмент программы на PL/1:

```
DECLARE A BIT(1);
A=1;
```

определяет значение A равным битовому 0, а не 1. Или в языке Си попытка присвоить переменной с плавающей точкой значение 1/2 даст значение 0 (т.к. 1 и 2 имеют целый тип).

3. Существуют ли вычисления, использующие переменные, имеющие одинаковый тип данных, но разную длину? Такой вопрос справедлив для PL/1 и возник он из этого языка. Например, в PL/1 результатом вычисления выражения 25 + 1/3 будет 5.333..., а не 25.333… Аналогично, в языке Си присутствуют множество данных одного типа, но имеющих разную длину. Например, это целые типы (CHAR, INT, LONG, SHORT, INT64 и т.д.), вещественные типы (FLOAT, DOUBLE, LONG DOUBLE) и т.д. К тому же, эти типы могут быть как знаковыми (SIGNED), так и беззнаковыми (UNSIGNED).

4. Имеет ли результирующая переменная оператора присваивания атрибуты, описывающие ее с меньшей длиной, чем в атрибутах выражения в правой части?

5. Возможны ли переполнение или потеря результата во время вычисления выражения? Это означает, что конечный результат может казаться правильным, но промежуточный результат может быть слишком большим или слишком малым для машинного представления данных.

6. Возможно ли, чтобы делитель в операторе деления был равен нулю?

7. Если величины представлены в машине в двоичной форме, получаются ли какие-нибудь результаты неточными? Так, 10×0,1 редко равно 1.0 в двоичной машине.

8. Может ли значение переменной выходить за пределы установленного для нее диапазона? Например, для операторов, присваивающих значение переменной PROB (имеющей смысл вероятности какого-либо события), может быть произведена проверка, будет ли полученное значение всегда положительным и не превышающим 1,0.

9. Верны ли предположения о порядке оценки и следования операторов для выражений, содержащих более чем один оператор?

10. Встречается ли неверное использование целой арифметики, особенно деления? Например, если I – целая величина, то выражение 2*I/2 == I зависит от того, является значение I четным или нечетным, и от того, какое действие – умножение или деление – выполняется первым.

6.3.3.4. Ошибки при сравнениях

1. Сравниваются ли в программе величины, имеющие несовместимые типы данных (например, строка символов с адресом)?

2. Сравниваются ли величины различных типов или величины различной длины? Если да, то проверьте, правильно ли интерпретируются (поняты) правила преобразования.

3. Корректны ли операторы сравнения? Программисты часто путают такие отношения, как *наибольший, наименьший, больше чем, не меньше чем, меньше или равно*.

4. Каждое ли булевское выражение сформулировано так, как это предполагалось? Программисты часто делают ошибки при написании логических выражений, содержащих операции «И», «ИЛИ», «НЕ».

5. Являются ли операнды булевских выражений булевскими? Существуют ли ошибочные объединения сравнений и булевских выражений? Они представляют другой часто встречающийся класс ошибок. Примеры нескольких типичных ошибок приведены ниже. Если величина L определена как лежащая в интервале между 2 и 10, то выражение 2<L<10 является неверным. Вместо него должно быть написано выражение (2< L)&(L< 10). Если же величина L определена как большая, чем X или Y, то выражение L>X|Y является неверным; оно должно быть записано в виде (L>X) | (L>Y). При сравнении трех чисел на равенство выражение IF (A=B=C) означает совсем другое. Например, в языке Си произойдет присвоение переменным A и B значения переменной C. А условие будет истинным, если это значение ненулевое. В случае необходимости проверить математическое отношение

X=Y=Z правильным будет выражение (X=Y)&(Y=Z). Также в языке Си следует различать булевские и битовые операторы. Например, если A = 1 и B = 2, то условие IF(A && B) будет истинно, а IF(A & B) – ложно.

6. Сравниваются ли в программе мантиссы или числа с плавающей запятой, которые представлены в машине в двоичной форме? Это является иногда источником ошибок из-за усечения младших разрядов. Или из-за неточного равенства чисел в двоичной и десятичной формах представления.

7. Верны ли предположения о порядке оценки и следовании операторов для выражений, содержащих более одного булевского оператора? Иными словами, если задано выражение (A==2)&(B==2)|(C==3), понятно ли, какая из операций выполняется первой: И или ИЛИ?

8. Влияет ли на результат выполнения программы способ, которым конкретный компилятор выполняет булевские выражения? Например, оператор

$$IF\ (X \neq O)\&((Y/X)>Z)$$

является приемлемым для некоторых компиляторов PL/1 (т.е. компиляторов, которые заканчивают проверку, как только одно из выражений оператора «И» окажется ложным), но приведет к делению на 0 при использовании других компиляторов.

6.3.3.5. Ошибки в передачах управления

1. Если в программе содержится переключатель (например, вычисляемый оператор GO TO в Фортране или его аналог ON… GOTO в Бейсике), то может ли значение индекса когда-либо превысить число возможных переходов? Например, всегда ли L будет принимать значение 1, 2 или 3 в операторе Фортрана GO TO (200, 300, 400), L или операторе Бейсика ON L GOTO 200, 300, 400?

2. Будет ли каждый цикл, в конце концов, завершен? Придумайте неформальное доказательство или аргументы, подтверждающие их завершение.

3. Будет ли программа, модуль или подпрограмма, в конечном счете, завершена?

4. Возможно ли, что из-за входных условий цикл никогда не сможет выполняться? Если это так, то является ли это оплошностью? Например, что произойдет для циклов, начинающихся операторами:

```
DO WHILE (NOTFOUND)
DO I=X TO Z
```

если первоначальное значение NOTFOUND – *ложь* или если X больше Z?

5. Для циклов, управляемых как числом итераций, так и булевским условием (например, цикл для организации поиска), какова последовательность «погружения в тело цикла»? Например, что произойдет с циклом, имеющим заголовок

DO I=1 TO TABLESIZE WHILE (NOTFOUND)

если NOTFOUND никогда не принимает значение «ложь»?

6. Существуют ли какие-нибудь ошибки «отклонения от нормы» (например, слишком большое или слишком малое число итераций)?

7. Если язык программирования содержит понятие группы операторов (например, DO-группы в PL/1, ограниченные операторами DO-END), то имеется ли явный оператор END для каждой группы, и соответствуют ли операторы END своим группам?

8. Существуют ли решения, подразумеваемые по умолчанию? Например, пусть ожидается, что входной параметр X принимает значения 1, 2 или 3. Логично ли тогда предположить, что он должен быть равен 3, если он не равен 1 или 2? Например, рассмотрим программу на языке Си:

```
switch(X)
{
   case 1: printf("1!!!"); break;
   case 2: printf("2!!!"); break;
   default: printf("3!!!");
}
```

Коль скоро это так, то является ли предположение правильным?

6.3.3.6. Ошибки интерфейса

1. Равно ли число параметров, получаемых рассматриваемым модулем, числу аргументов, передаваемых каждым из вызывающих модулей? Правилен ли порядок их следования?

2. Совпадают ли атрибуты (например, тип и размер) каждого параметра с атрибутами соответствующего ему аргумента?

3. Совпадают ли единицы измерения каждого параметра с единицами измерения соответствующих аргументов? Например, нет ли случаев, когда значение параметров выражено в градусах, а аргумента – в радианах?

4. Равно ли число аргументов, передаваемых из рассматриваемого модуля другому модулю, числу параметров, ожидаемых в вызываемом модуле?

5. Соответствуют ли атрибуты каждого аргумента, передаваемого другому модулю, атрибутам соответствующего параметра в рассматриваемом модуле?

6. Совпадают ли единицы измерения каждого аргумента, передаваемого другому модулю, с единицами измерения соответствующего параметра в рассматриваемом модуле?

7. Если вызываются встроенные функции, правильно ли заданы число, атрибуты и порядок следования аргументов?

8. Не изменяет ли подпрограмма параметр, который должен использоваться только как входная величина?

9. Если имеются глобальные переменные (например, переменные в PL/1 с атрибутом EXTERNAL или в Си с атрибутом EXTERN, переменные, указанные в операторах COMMON Фортрана), имеют ли они одинаковые определения и атрибуты во всех модулях, которые к ним обращаются?

10. Передаются ли в качестве аргументов константы? В некоторых реализациях Фортрана такие операторы, как

```
CALL SUBX (J,3)
```

являются опасными, поскольку, если подпрограмма SUBX присвоит значение второму параметру, значение константы 3 будет изменено.

6.3.3.7. Ошибки ввода-вывода

1. Являются ли правильными атрибуты файлов, описанных явно?

2. Являются ли правильными атрибуты оператора OPEN?

3. Согласуется ли спецификация формата с информацией в операторах ввода-вывода? Например, согласуется ли каждый оператор FORMAT (с точки зрения числа элементарных данных и их атрибутов) с соответствующими операторами READ и WRITE в программе, написанной на Фортране? То же самое применимо к проверке соответствия между списком данных и списком форматов в операторах ввода-вывода PL/1 и Си.

4. Равен ли размеру записи размер области памяти для ввода-вывода? Это может быть важно при блочном вводе-выводе (функции BLOCKREAD и BLOCKWRITE в Паскале, FREAD и FWRITE в Си).

5. Все ли файлы открыты перед их использованием?

6. Правильно ли обнаруживаются и трактуются признаки конца файла?

7. Правильно ли трактуются ошибочные состояния ввода-вывода?

8. Существуют ли смысловые или грамматические ошибки в тексте, выводимом программой на печать или экран дисплея?

6.3.3.8. Другие виды контроля

1. Если компилятор выдает таблицу перекрестных ссылок идентификаторов, проверьте величины, на которые в этом списке нет ссылок или есть только одна ссылка.

2. Если компилятор выдает список атрибутов, проверьте атрибуты каждой величины для обеспечения гарантии того, что в программе нет никаких неожиданных и отсутствующих атрибутов.

3. Если программа оттранслирована успешно, но компилятор выдает одно или несколько «предупреждений» или «информационных» сообщений, внимательно проверьте каждое из них.

Предупреждение свидетельствует о «подозрениях» компилятора в отношении правильности ваших действий. Все эти «подозрения» должны быть рассмотрены. В информационных сообщениях могут перечисляться неописанные переменные или конструкции языка, которые препятствуют оптимизации кода.

4. Является ли программа (или модуль) достаточно устойчивой? Иными словами, проверяет ли она правильность своих входных данных?

5. Не пропущена ли в программе какая-нибудь функция?

6.3.4. Сквозные просмотры

Сквозной просмотр, как и инспекции, представляет собой набор процедур и способов обнаружения, осуществляемых группой лиц, просматривающих текст программы. Такой просмотр имеет много общего с процессом инспектирования, но их процедуры несколько отличаются и, кроме того, здесь используются другие методы обнаружения ошибок.

Подобно инспекции, сквозной просмотр проводится как непрерывное заседание, продолжающееся один или два часа. Группа по выполнению сквозного просмотра состоит из 3-5 человек. В нее входят председатель, функции которого подобны функциям председателя в группе инспектирования, секретарь, который записывает все найденные ошибки, и специалист по тестированию. Мнения о том, кто должен быть четвертым и пятым членами группы, расходятся. Конечно, одним из них должен быть программист. Относительно пятого участника имеются следующие предположения:

1) высококвалифицированный программист;
2) эксперт по языку программирования;

3) начинающий (на точку зрения которого не влияет предыдущий опыт);
4) человек, который будет, в конечном счете, эксплуатировать программу;
5) участник какого-нибудь другого проекта;
6) кто-либо из той же группы программистов, что и автор программы.

Начальная процедура при сквозном просмотре такая же, как и при инспекции: участникам заранее, за несколько дней до заседания, раздаются материалы, позволяющие им ознакомиться с программой. Однако процедура заседания отличается от процедуры инспекционного заседания. Вместо того, чтобы просто читать текст программы или использовать список ошибок, участники заседания «исполняют роль вычислительной машины». Лицо, назначенное тестирующим, предлагает собравшимся небольшое число написанных на бумаге тестов, представляющих собой наборы входных данных (и ожидаемых выходных данных) для программ или модуля. Во время заседания каждый тест мысленно выполняется. Это означает, что тестовые данные подвергаются обработке в соответствии с логикой программы. Состояние программы (т.е. значения переменных) отслеживается на бумаге или доске.

Конечно, число тестов должно быть небольшим и они должны быть простыми по своей природе, потому что скорость выполнения программы человеком на много порядков меньше, чем у машины. Следовательно, тесты сами по себе не играют критической роли, скорее они служат средством для первоначального понимания программы и основой для вопросов программисту о логике проектирования и принятых допущениях. В большинстве сквозных просмотров при выполнении самих тестов находят меньше ошибок, чем при опросе программиста.

Как и при инспекции, мнение участников является решающим фактором. Замечания должны быть адресованы программе, а не программисту. Другими словами, ошибки не рассматриваются как слабость человека, который их совершил. Они свидетельствуют о сложности процесса создания программ и являются результатом все еще примитивной природы существующих методов программирования.

Сквозные просмотры должны протекать так же, как и описанный ранее прогресс инспектирования. Побочные эффекты, получаемые во время выполнения этого процесса (установление склонных к ошибкам частей программы и обучение на основе анализа ошибок, стиля и методов) характерны и для процесса сквозных просмотров.

6.3.5. Оценка посредством просмотра

Последний ручной процесс обзора программы не связан с ее тестированием (т.е. целью его не является нахождение ошибок). Однако описание этого процесса приводится здесь потому, что он имеет отношение к идее чтения текста.

Оценка посредством просмотра является методом оценки анонимной программы в терминах ее общего качества, ремонтопригодности, расширяемости, простоты эксплуатации и ясности. Цель данного метода – обеспечить программиста средствами самооценки. Выбирается программист, который должен выполнять обязанности администратора процесса. Администратор в свою очередь отбирает приблизительно 6-20 участников (6 – минимальное число для сохранения анонимности). Предполагается, что участники должны быть одного профиля (например, в одну группу не следует объединять программистов, использующих Кобол, и системных программистов, пишущих на Ассемблере). Каждого участника просят представить для рассмотрения две свои программы – наилучшую (с его точки зрения) и низкого качества.

Отобранные программы случайным образом «распределяются между участниками. Им дается на рассмотрение по четыре программы. Две из них являются «наилучшими», а две – «наихудшими», но рецензенту не сообщают о том, какая программа к какой группе относится. Каждый участник тратит на просмотр одной программы 30 мин и заполняет анкету для ее оценки. После просмотра всех четырех программ оценивается их относительное качество. В анкете для оценки проверяющему предлагается оценить программу по семибалльной шкале (1 означает определенное «да», 7 – определенное «нет») при ответе, например, на следующие вопросы:

- Легко ли было понять программу?
- Оказались ли результаты проектирования высокого уровня очевидными и приемлемыми?
- Оказались ли результаты проектирования низкого уровня очевидными и приемлемыми?
- Легко ли для вас модифицировать эту программу?
- Испытывали бы вы чувство удовлетворения, написав такую программу?

Проверяющего просят также дать общий комментарий и рекомендации по улучшению программы. После просмотра каждому участнику передают анонимную анкету с оценкой двух его программ. Участники получают статистическую сводку, которая содержит общую и детальную классификацию их собственных программ в сравне-

нии с полным набором программ, и анализ того, насколько оценки чужих программ совпадают с оценками тех же самых программ, данными другими проверяющими. Цель такого просмотра – дать возможность программистам самим оценить свою квалификацию. Этот способ представляется полезным как для промышленного, так и для учебного применения.

6.4. Проектирование теста

Результаты психологических исследований, обсуждавшиеся в п. 6.2, показывают, что наибольшее внимание при тестировании программ уделяется проектирование или созданию эффективных тестов. Это связано с невозможностью «полного» тестирования программы, т.е. тест для любой программы будет обязательно неполным (иными словами, тестирование не может гарантировать отсутствия всех ошибок). Стратегия проектирования заключается в том, чтобы попытаться уменьшить эту «неполноту» настолько, насколько это возможно.

Если ввести ограничения на время, стоимость, машинное время и т.п., то ключевым вопросом тестирования становится следующий:

Какое подмножество всех возможных тестов имеет наивысшую вероятность обнаружения большинства ошибок?

Изучение методологий проектирования тестов дает ответ на этот вопрос.

По-видимому, наихудшей из всех методологий является тестирование со случайными входными значениями (стохастическое) – процесс тестирования программы путем случайного выбора некоторого подмножества из всех возможных входных величин. В терминах вероятности обнаружения большинства ошибок случайно выбранный набор тестов имеет малую вероятность быть оптимальным или близким к оптимальному подмножеством. В настоящем разделе рассматривается несколько подходов, которые позволяют более разумно выбирать тестовые данные. В п. 6.2 было показано, что исчерпывающее тестирование по принципу черного или белого ящика в общем случае невозможно. Однако, при этом отмечалось, что приемлемая стратегия тестирования может обладать элементами обоих подходов. Таковой является стратегия, излагаемая в этом разделе. Можно разработать довольно полный тест, используя определенную методологию проектирования, основанную на принципе черного ящика, а затем дополнить его проверкой логики программы (т.е. с привлечением методов белого ящика).

Методологии, обсуждаемые в настоящем разделе, представлены ниже:

Черный ящик	Белый ящик
Эквивалентное разбиение	Покрытие операторов
Анализ граничных значений	Покрытие решений
Применение функциональных диаграмм	Покрытие условий
	Покрытие решений/условий
Предположение об ошибке	Комбинаторное покрытие условий

Хотя перечисленные методы будут рассматриваться здесь по отдельности, при проектировании эффективного теста программы рекомендуется использовать если не все эти, то, по крайней мере, большинство из них, так как каждый из них имеет определенные достоинства и недостатки (например, возможность обнаруживать и пропускать различные типы ошибок). Правда, эти методы весьма трудоемки, поэтому некоторые специалисты, ознакомившись с ними, могут не согласиться с данной рекомендацией. Однако следует представлять себе, что тестирование программы – чрезвычайно сложная задача. Для иллюстрации этого приведем известное изречение: «Если вы думаете, что разработка и кодирование программы – вещь трудная, то вы еще ничего не видели».

Рекомендуемая процедура заключается в том, чтобы разрабатывать тесты, используя методы черного ящика, а затем как необходимое условие – дополнительные тесты, используя методы белого ящика. Методы белого ящика, получившие более широкое распространение, обсуждаются первыми.

6.4.1. Тестирование путем покрытия логики программы

Тестирование по принципу белого ящика характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы. Как показано в п. 6.2, исчерпывающее тестирование по принципу белого ящика предполагает выполнение каждого пути в программе; но, поскольку в программе с циклами выполнение каждого пути обычно нереализуемо, то тестирование всех путей не рассматривается как перспективное.

6.4.1.1. Покрытие операторов

Если отказаться полностью от тестирования всех путей, то можно показать, что критерием покрытия является выполнение каждого оператора программы, по крайней мере, один раз. К сожалению, это слабый критерий, так как выполнение каждого оператора, по крайней

мере, один раз есть необходимое, не недостаточное условие для приемлемого тестирования по принципу белого ящика (рис. 4.1).

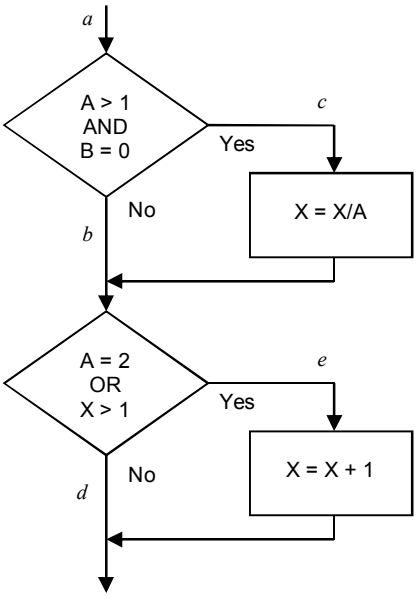


Рис. 4.1 – Алгоритм тестируемой программы

Предположим, что на рис. 4.1 представлена небольшая программа, которая должна быть протестирована. Эквивалентная программа, написанная на языке PL/1, имеет вид:

```
M: PROCEDURE (A,B,X);
  IF ((A>1) & (B=0)) THEN DO;
     X=X/A;
  END;
  IF ((A=2) | (X>1)) THEN DO;
     X=X+1;
  END;
END;
```

Можно выполнить каждый оператор, записав один-единственный тест, который реализовал бы путь *ace*. Иными словами, если бы в точке *a* были установлены значения $A = 2$, $B = 0$ и $X = 3$, каждый опе-

ратор выполнялся бы один раз (в действительности X может принимать любое значение).

К сожалению, этот критерий хуже, чем он кажется на первый взгляд. Например, пусть первое решение записано как *или,* а не как *и.* При тестировании по данному критерию эта ошибка не будет обнаружена. Пусть второе решение записано в программе как $X > 0$; эта ошибка также не будет обнаружена. Кроме того, существует путь, в котором X не изменяется (путь *abd).* Если здесь ошибка, то и она не будет обнаружена. Таким образом, критерий покрытия операторов является настолько слабым, что его обычно не используют.

6.4.1.2. Покрытие решений

Более сильный критерий покрытия логики программы известен как *покрытие решений,* или *покрытие переходов.*

Согласно данному критерию должно быть записано достаточное число тестов, такое, что каждое решение на этих тестах примет значение *истина* и *ложь*, по крайней мере, один раз. Иными словами, каждое направление перехода должно быть реализовано, по крайней мере, один раз. Примерами операторов перехода или решений являются операторы DO (или PERFORM UNTIL в Коболе, REPEAT UNTIL, WHILE в Паскале, WHILE и DO WHILE в Си), IF, многовыходные операторы GO TO.

Можно показать, что покрытие решений обычно удовлетворяет критерию покрытия операторов. Поскольку каждый оператор лежит на некотором пути, исходящем либо из оператора перехода, либо из точки входа программы, при выполнении каждого направления перехода каждый оператор должен быть выполнен. Однако существует, по крайней мере, два исключения. Первое – патологическая ситуация, когда программа не имеет решений. Второе встречается в программах или подпрограммах с несколькими точками входа; данный оператор может быть выполнен только в том случае, если выполнение программы начинается с соответствующей точки входа. Так как покрытие операторов считается необходимым условием, покрытие решений, которое представляется более сильным критерием, должно включать покрытие операторов. Следовательно, покрытие решений требует, чтобы каждое решение имело результатом значение *истина* или *ложь*, и при этом каждый оператор выполнялся бы, по крайней мере, один раз. Альтернативный и более легкий способ выражения этого требования состоит в том, чтобы каждое решение имело результатом значение *истина* или *ложь*, и что каждой точке входа должно быть передано управление при вызове программы, по крайней мере, один раз.

Изложенное выше предполагает только двузначные решения или переходы и должно быть модифицировано для программ, содержащих многозначные решения. Примерами таких программ являются:

- программы на PL/1, включающие операторы SELECT (CASE) или операторы GO TO, использующие метку-переменную;
- программы на Фортране с вычисляемыми операторами GO TO или операторами GO TO по предписанию;
- программы на Бейсике с вычисляемыми операторами ON – GOTO, ON – GOSUB.
- программы на Коболе, содержащие операторы GO TO вместе с ALTER или операторы GO TO – DEPENDING – ON.
- программы на Паскале, использующие операторы CASE.
- программы на языке Си, использующие операторы SWITCH – CASE.
- любые программы с арифметическими операторами IF.

Критерием для них является выполнение каждого возможного результата всех решений, по крайней мере, один раз и передача управления при вызове программы или подпрограммы каждой точке входа, по крайней мере, один раз.

В программе, представленной на рис. 4.1, покрытие решений может быть выполнено двумя тестами, покрывающими либо пути *ace* и *abd,* либо пути *acd* и *abe.* Если мы выбираем последнее альтернативное покрытие, то входами двух тестов являются $A = 3$, $B = 0$, $X = 3$ и $A = 2$, $B = 1$, $X = 1$.

Покрытие решений – более сильный критерий, чем покрытие операторов, но и он имеет свои недостатки. Например, путь, где X не изменяется (если выбрано первое альтернативное покрытие), будет проверен с вероятностью 50%. Если во втором решении существует ошибка (например, $X<1$ вместо $X>1$), то ошибка не будет обнаружена двумя тестами предыдущего примера.

6.4.1.3. Покрытие условий

Лучшим критерием по сравнению с предыдущим является *покрытие условий.* В этом случае записывают число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз. Поскольку, как и при покрытии решений, это покрытие не всегда приводит к выполнению каждого оператора, к критерию требуется дополнение, которое заключается в том, что каждой точке входа в программу или подпрограмму, а также ON-единицам должно быть передано управление при вызове, по крайней мере, один раз. Например, оператор цикла в языке Фортран

DO K=0 TO 50 WHILE (J+K<QUEST);

или его аналог на языке Си

for(K=0; K<=50 && J+K<QUEST; K++)

содержит два условия: *K* меньше или равно 50 и *J*+*K* меньше, чем *QUEST*. Следовательно, здесь требуются тесты для ситуаций *K*≤50, *K*>50 (т.е. выполнение последней итерации цикла), *J*+*K*<*QUEST* и *J*+*K*≥*QUEST*.

Программа рис. 4.1 имеет четыре условия: A > 1, *B* = 0, *A = 2* и X > 1. Следовательно, требуется достаточное число тестов, такое, чтобы реализовать ситуации, где A > 1, A ≤ 1, B = 0 и B ≠ 0 в точке *a* и *A = 2, A* ≠ 2 и *X > 1* в точке *b*. Тесты, удовлетворяющие критерию покрытия условий, и соответствующие им пути:

1. A = 2, B = 0, X = 4 – *ace*.
2. A = 1, B = 1, X = 1 – *abd*.

Заметим, что, хотя аналогичное число тестов для этого примера уже было создано, покрытие условий обычно лучше покрытия решений, поскольку оно *может* (но не всегда) вызвать выполнение решений в условиях, не реализуемых при перекрытии решений. Например, рассмотренные выше операторы представляют собой двузначный переход (либо выполняется тело цикла, либо выход из цикла). Если использовать тестирование решений, то достаточно выполнить цикл при изменении K от 0 до 51 *без проверки случая, когда второе условие ложно*. Однако, при критерии покрытия условий необходим тест, который реализовал бы результат «ложь» условия J+K<QUEST.

Хотя применение критерия покрытия условий на первый взгляд удовлетворяет критерию покрытия решений, это не всегда так. Если тестируется решение IF (A&B), то при критерии покрытия условий требовались бы два теста – A есть *истина,* B есть *ложь* и A есть *ложь,* B есть *истина*. Но в этом случае не выполнялось бы THEN-предложение оператора IF. Тесты критерия покрытия условий для ранее рассмотренного примера покрывают результаты всех решений, но это только случайное совпадение. Например, два альтернативных теста

1. A = 1, B = 0, X = 3
2. A = 2, B = 1, X = 1

покрывают результаты всех условий, но только два из четырех результатов решений (они оба покрывают путь *abe* и, следовательно, не выполняют результат *истина* первого решения и результат *ложь* второго решения).

6.4.1.4. Покрытие решений/условий

Очевидным следствием из этой дилеммы является критерий, названный *покрытием решений/условий.* Он требует такого достаточного набора тестов, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз, все результаты каждого решения выполнялись, по крайней мере, один раз и каждой точке входа передавалось управление, по крайней мере, один раз.

Недостатком критерия покрытия решений/условий является невозможность его применения для выполнения всех результат всех условий; часто подобное выполнение имеет место вследствие того, что определенные условия скрыты другими условиями. В качестве примера рассмотрим приведенную на рис. 4.2 схему передач управления в машинном коде программы рис. 4.1. Многоусловные решения исходной программы здесь разбиты на отдельные решения и переходы, поскольку большинство машин не имеет команд, реализующих решения с многими исходами. Наиболее полное покрытие тестами в этом случае осуществляется таким образом, чтобы выполнялись все возможные результаты каждого простого решения.

Два предыдущих теста критерия покрытия решений не выполняют этого. Они недостаточны для выполнения результата «*ложь*» решения Н и результата «*истина*» решения К. Набор тестов для критерия покрытия условий такой программы также является неполным – два теста (которые случайно удовлетворяют также и критерию покрытия решений/условий) не вызывают выполнения результата «*ложь*» решения I и результата «*истина*» решения К.

Причина этого заключается в том, что, как показано на рис. 4.2, результаты условий в выражениях *и* и *или* могут скрывать и блокировать действие других условий. Например, если условие *и* есть *ложь,* то никакое из последующих условий в выражении не будет выполнено. Аналогично, если условие *или* есть *истина,* то никакое из последующих условий не будет выполнено. Следовательно, критерии покрытия условий и покрытия решений/условий недостаточно чувствительны к ошибкам в логических выражениях.

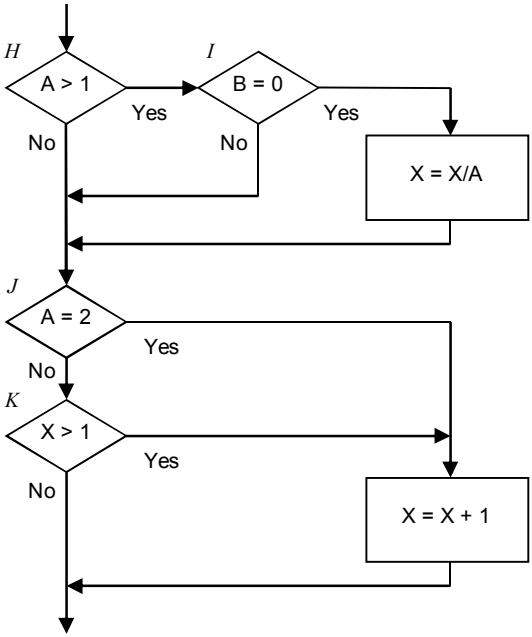


Рис. 4.2 – Машинный код программы, изображенной на рис. 4.1

6.4.1.5. Комбинаторное покрытие условий

Критерием, который решает эти и некоторые другие проблемы, является *комбинаторное покрытие условий.* Он требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении и во всех точках входа выполнялись, по крайней мере, один раз. Например, в приведенной ниже последовательности операторов существуют четыре ситуации, которые должны быть протестированы:

```
NOTFOUND = '1' B;
/* поиск в таблице */
DO I=1 TO TABSIZE WHILE (NOTFOUND);
 /* последовательность операторов, реализующая
   процедуру поиска */
END;
```

1. I≤TABSIZE и NOTFOUND есть «истина».

2. I≤TABSIZE и NOTFOUND есть «ложь» (обнаружение необходимого искомого значения до достижения конца таблицы).
3. I>TABSIZE и NOTFOUND есть «истина» (достижение конца таблицы без обнаружения искомого значения).
4. I>TABSIZE и NOTFOUND есть «ложь» (искомое значение является последней записью в таблице).

Легко видеть, что набор тестов, удовлетворяющий критерию комбинаторного покрытия условий, удовлетворяет также и критериям покрытия решений, покрытия условий и покрытия решений/условий.

По этому критерию в программе на рис. 4.1 должны быть покрыты тестами следующие восемь комбинаций:

1. $A > 1, B = 0$
2. $A > 1, B \neq 0$
3. $A \leq 1, B = 0$
4. $A \leq 1, B \neq 0$
5. $A = 2, X > 1$
6. $A = 2, X \leq 1$
7. $A \neq 2, X > 1$
8. $A \neq 2, X \leq 1$

Заметим, что комбинации 5-8 представляют собой значения второго оператора IF. Поскольку *X* может быть изменено до выполнения этого оператора, значения, необходимые для его проверки, следует восстановить исходя из логики программы с тем, чтобы найти соответствующие входные значения.

Для того, чтобы протестировать эти комбинации, необязательно использовать все восемь тестов. Фактически они могут быть покрыты четырьмя тестами. Приведем входные значения тестов и комбинации, которые они покрывают:

A=2, B=0, X=4 – покрывает 1, 5;

A=2, B=1, X=1 – покрывает 2, 6;

A=1, B=0, X=2 – покрывает 3, 7;

A=1, B=1, X=1 – покрывает 4, 8.

То, что четырем тестам соответствуют четыре различных пути на рис. 4.1, является случайным совпадением. На самом деле, представленные выше тесты не покрывают всех путей, они пропускают путь *acd*. Например, требуется восемь тестов для тестирования следующей программы:

```
IF ((X=Y) & (LENGTH(Z)=0) & EPS) THEN J=1;
ELSE I=1;
```

хотя она покрывается лишь двумя путями. В случае циклов число тестов для удовлетворения критерию комбинаторного покрытия условий обычно больше, чем число путей.

Таким образом, для программ, содержащих только одно условие на каждое решение, минимальным является критерий, набор тестов которого

1) вызывает выполнение всех результатов каждого решения, по крайней мере, один раз и
2) передает управление каждой точке входа (например, точке входа, ON-единице) по крайней мере, один раз (чтобы обеспечить выполнение каждого оператора программы, по крайней мере, один раз).

Для программ, содержащих решения, каждое из которых имеет более одного условия, минимальный критерий состоит из набора тестов, вызывающих выполнение всех возможных комбинаций результатов условий в каждом решении и передающих управление каждой точке входа программы, по крайней мере, один раз.

6.4.2. Эквивалентное разбиение

В п. 6.2 отмечалось, что хороший тест имеет приемлемую вероятность обнаружения ошибки и что исчерпывающее входное тестирование программы невозможно. Следовательно, тестирование программы ограничивается использованием небольшого подмножества всех возможных входных данных. Тогда, конечно, хотелось бы выбрать для тестирования самое подходящее подмножество (т.е. подмножество с наивысшей вероятностью обнаружения большинства ошибок).

Правильно выбранный тест этого подмножества должен обладать двумя свойствами:

а) уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования;
б) покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

Указанные свойства, несмотря на их кажущееся подобие, описывают два различных положения. Во-первых, каждый тест должен включать столько различных входных условий, сколько это возможно, с тем, чтобы минимизировать общее число необходимых тестов. Во-вторых, необходимо пытаться разбить входную область программы на конечное число *классов эквивалентности* так, чтобы можно было

предположить (конечно, не абсолютно уверенно), что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса. Иными словами, если один тест класса эквивалентности обнаруживает ошибку, то следует ожидать, что и все другие тесты этого класса эквивалентности будут обнаруживать ту же самую ошибку. Наоборот, если тест не обнаруживает ошибки, то следует ожидать, что ни один тест этого класса эквивалентности не будет обнаруживать ошибки (в том случае, когда некоторое подмножество класса эквивалентности не попадает в пределы любого другого класса эквивалентности, так как классы эквивалентности могут пересекаться).

Эти два положения составляют основу методологии тестирования по принципу черного ящика, известной как *эквивалентное разбиение.* Второе положение используется для разработки набора «интересных» условий, которые должны быть протестированы, а первое – для разработки минимального набора тестов, покрывающих эти условия.

Примером класса эквивалентности для программы о треугольнике (см. п. 6.1) является набор «трех равных чисел, имеющих целые значения, большие – нуля». Определяя этот набор как класс эквивалентности, устанавливают, что если ошибка не обнаружена некоторым тестом данного набора, то маловероятно, что она будет обнаружена другим тестом набора. Иными словами, в этом случае время тестирования лучше затратить на что-нибудь другое (на тестирование других классов эквивалентности).

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:

1) выделение классов эквивалентности и

2) построение тестов.

6.4.2.1. Выделение классов эквивалентности

Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза в спецификации) и разбиением его на две или более групп. Для проведения этой операции используют таблицу, изображенную на рис. 4.3.

Входные условия	Правильные классы эквивалентности	Неправильные классы эквивалентности

Рис. 4.3 – Форма таблицы для перечисления классов эквивалентности

Заметим, что различают два типа классов эквивалентности: правильные классы эквивалентности, представляющие правильные входные данные программы, и неправильные классы эквивалентности, представляющие все другие возможные состояния условий (т.е. ошибочные входные значения). Таким образом, придерживаются одного из принципов п. 6.2 о необходимости сосредоточивать внимание на неправильных или неожиданных условиях.

Если задаться входными или внешними условиями, то выделение классов эквивалентности представляет собой в значительной степени эвристический процесс. При этом существует ряд правил:

1. Если входное условие описывает *область* значений (например, «целое данное может принимать значения от 1 до 999»), то определяются:

   один правильный класс эквивалентности (значений от 1 до 999);

   два неправильных класса (значения целого данного <1 и значение целого данного >999).

2. Если входное условие описывает *число* значений (например, «в автомобиле могут ехать от одного до шести человек»), то определяются один правильный класс эквивалентности и два неправильных (ни одного и более шести).

3. Если входное условие описывает множество входных значений и есть основание полагать, что каждое значение программа трактует особо (например, «известны способы передвижения на АВТОБУСЕ, ГРУЗОВИКЕ, ТАКСИ, ПЕШКОМ или МОТОЦИКЛЕ»), то определяется правильный класс эквивалентности для каждого значения и один неправильный класс эквивалентности (например, «НА ПРИЦЕПЕ»).

4. Если входное условие описывает ситуацию «должно быть» (например, «первым символом идентификатора должна быть буква»), то определяется один правильный класс эквивалентности (первый символ – буква) и один неправильный (первый символ – не буква).

5. Если есть любое основание считать что, различные элементы класса эквивалентности трактуются программой неодинаково, то данный класс эквивалентности разбивается на меньшие классы эквивалентности.

Этот процесс ниже будет кратко проиллюстрирован.

6.4.2.2. Построение тестов

Второй шаг заключается в использовании классов эквивалентности для построения тестов. Этот процесс включает в себя:

1. Назначение каждому классу эквивалентности уникального номера.

2. Проектирование новых тестов, каждый из которых покрывает как можно большее число правильных непокрытых классов эквивалентности, до тех пор, пока все правильные классы эквивалентности не будут покрыты (только не общими) тестами.

3. Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор, пока все неправильные классы эквивалентности не будут покрыты тестами.

Причина покрытия неправильных классов эквивалентности индивидуальными тестами состоит в том, что определенные проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами. Например, спецификация устанавливает «тип книги при поиске (ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА, ПРОГРАММИРОВАНИЕ или ОБЩИЙ) и количество (1 – 9999)». Тогда тест

XYZ 0

отображает два ошибочных условия (неправильный тип книги и количество) и, вероятно, не будет осуществлять проверку количества, так как программа может ответить: «XYZ – НЕСУЩЕСТВУЮЩИЙ ТИП КНИГИ» и не проверять остальную часть входных данных.

Пример.

Предположим, что при разработке компилятора для подмножества языка Фортран требуется протестировать синтаксическую проверку оператора DIMENSION. Спецификация приведена ниже. В спецификации элементы, написанные латинскими буквами, обозначают синтаксические единицы, которые в реальных операторах должны быть заменены соответствующими значениями, в квадратные скобки заключены необязательные элементы, многоточие показывает, что предшествующий ему элемент может быть повторен подряд несколько раз.

Оператор DIMENSION используется для определения массивов. Форма оператора DIMENSION:

DIMENSION ad[,ad]…,

где *ad* есть описатель массива в форме

n(d[,d]…),

где *n* – символическое имя массива, а *d* – размерность массива. Символические имена могут содержать от одного до шести символов – букв или цифр, причем первой должна быть буква. Допускается использование от одной до семи размерностей в форме

[lb:]ub,

где *lb* и *ub* задают нижнюю и верхнюю границы индекса массива. Граница может быть либо константой, принимающей значения от –65534 до 65535, либо целой переменной (без индексов). Если *lb* не определена, то предполагается, что она равна единице. Значение *ub* должно быть больше или равно *lb*. Если *lb* определена, то она может иметь отрицательное, нулевое или положительное значение. Как и все операторы, оператор DIMENSION может быть продолжен на нескольких строках (конец спецификации).

Первый шаг заключается в том, чтобы идентифицировать входные условия и по ним определить классы эквивалентности (табл. 4.1). Классы эквивалентности в таблице обозначены числами.

Таблица 4.1 – Классы эквивалентности

Входные условия	Правильные классы эквивалентности	Неправильные классы эквивалентности
Число описателей массивов	один (1), больше одного (2)	ни одного (3)
Длина имени массива	1-6 (4)	0 (5), больше 6 (6)
Имя массива	имеет в своем составе буквы (7) и цифры (8)	Содержит что-то еще (9)
Имя массива начинается с буквы	да (10)	нет (11)
Число индексов	1-7 (12)	0 (13), больше 7 (14)
Верхняя граница	константа (15), целая переменная (16)	имя элемента массива (17), что-то иное (18)
Имя целой переменной	имеет в своем составе буквы (19), и цифры (20)	состоит из чего-то еще (21)
Целая переменная начинается с буквы	да (22)	нет (23)
Константа	–65534 – +65535 (24)	меньше –65534 (25), больше 65535 (26)
Нижняя граница определена	да (27), нет (28)	
Верхняя граница по отношению к нижней границе	больше (29), равна (30)	меньше (31)
Значение нижней границы	отрицательное (32), нуль (33), положительное (34)	
Нижняя граница	константа (35), целая переменная (36)	имя элемента массива (37), что-то иное (38)
Оператор расположен на нескольких строках	да (39), нет (40)	

Следующий шаг – построение теста, покрывающего один или более правильных классов эквивалентности. Например, тест

```
DIMENSION A(2)
```

покрывает классы 1, 4, 7, 10, 12, 15, 24, 28, 29 и 40. Далее определяются один или более тестов, покрывающих оставшиеся правильные классы эквивалентности. Так, тест

```
DIMENSION A 12345 (I,9,J4XXXX,65535,1,
  KLM, * 100), BBB (–65534 : 100,0 :
  1000,10 : 10,I: 65535)
```

покрывает оставшиеся классы. Перечислим неправильные классы эквивалентности и соответствующие им тесты:

```
(3)     DIMENSION
(5)     DIMENSION   (10)
(6)     DIMENSION   A234567(2)
(9)     DIMENSION   A.I(2)
(11)    DIMENSION   1A(10)
(13)    DIMENSION   B
(14)    DIMENSION   B (4,4,4,4,4,4,4,4)
(17)    DIMENSION   B(4,A(2))
(18)    DIMENSION   B(4,,7)
(21)    DIMENSION   C(I.,10)
(23)    DIMENSION   C(10,1J)
(25)    DIMENSION   D(–65535:1)
(26)    DIMENSION   D(65536)
(31)    DIMENSION   D(4:3)
(37)    DIMENSION   D(A(2):4)
(38)    DIMENSION   D.:4)
```

Эти классы эквивалентности покрываются 18 тестами.

Хотя эквивалентное разбиение значительно лучше случайного выбора тестов, оно все же имеет недостатки (т.е. пропускает определенные типы высокоэффективных тестов). Следующие два метода – анализ граничных значений и использование функциональных диаграмм (диаграмм причинно-следственных связей, или cause-effect graphing) – свободны от многих недостатков, присущих эквивалентному разбиению.

6.4.3. Анализ граничных значений

Как показывает опыт, тесты, исследующие *граничные условия*, приносят большую пользу, чем тесты, которые их не исследуют. Гра-

ничные условия – это ситуации, возникающие непосредственно на границах, а также выше или ниже границ входных и выходных классов эквивалентности. Анализ граничных значений отличается от эквивалентного разбиения в двух отношениях:

1) Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных значений осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.
2) При разработке тестов рассматривают не только входные условия (пространство входов), но и *пространство результатов* (т.е. выходные классы эквивалентности).

Трудно описать «кухню» анализа граничных значений, так как это требует определенной степени творчества и специализации в рассматриваемой проблеме (следовательно, анализ граничных значений, как и многие другие аспекты тестирования, в значительной мере основывается на способностях человеческого интеллекта). Тем не менее, приведем несколько правил этого метода.

1. Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений. Например, если правильная область входных значений есть –1.0…+1.0, то написать тесты для ситуаций –1.0, 1.0, –1.001 и 1.001.

2. Построить тесты для минимального и максимального значения условий и тесты, большие и меньшие этих значений, если входное условие удовлетворяет дискретному ряду значений. Например, если входной файл может содержать от 1 до 255 записей, то получить тесты для 0, 1, 255 и 256 записей.

3. Использовать правило 1 для каждого выходного условия. Например, если программа вычисляет ежемесячный расход, и если минимум расхода составляет $0,00, а максимум – $1165,25, то построить тесты, которые вызывают расходы с $0,00 по $1165,25. Кроме того, построить, если это возможно, тесты, которые вызывают отрицательный расход и расход больше $1165,25. Заметим, что важно проверить границы пространства результатов, поскольку не всегда границы входных областей представляют такой же набор условий, как и границы выходных областей (например, при рассмотрении подпрограммы вычисления синуса). Не всегда также можно получить результат вне выходной области, но, тем не менее, стоит рассмотреть эту возможность.

4. Использовать правило 2 для каждого входного условия. Например, если система информационного поиска отображает на

экране терминала наиболее релевантные рефераты в зависимости от входного запроса, но никак не более четырех рефератов, то построить тесты, такие, чтобы программа отображала нуль, один и четыре реферата, и тест, который мог бы вызвать выполнение программы с ошибочным отображением пяти рефератов.

5. Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.

6. Попробовать свои силы в поиске других граничных условий.

Чтобы проиллюстрировать необходимость анализа граничных значений, можно использовать программу анализа треугольника, приведенную в п. 6.1. Для задания треугольника входные значения должны быть целыми положительными числами, и сумма любых двух из них должна быть больше третьего. Если определены эквивалентные разбиения, то целесообразно определить одно разбиение, в котором это условие выполняется, и другое, в котором сумма двух целых не больше третьего. Следовательно, двумя возможными тестами являются 3 – 4 – 5 и 1 – 2 – 4. Тем не менее, здесь есть вероятность пропуска ошибки. Иными словами, если выражение в программе было закодировано как A+B≥C вместо A+B>C, то программа ошибочно сообщала бы нам, что числа 1 – 2 – 3 представляют правильный равносторонний треугольник. Таким образом, существенное различие между анализом граничных значений и эквивалентным разбиением заключается в том, что анализ граничных значений исследует ситуации, возникающие *на границах и вблизи границ эквивалентных разбиений.*

В качестве примера для применения метода анализа граничных значений рассмотрим следующую спецификацию программы.

MTEST есть программа, которая сортирует различную информацию об экзаменах. Входом программы является файл, названный OCR, который содержит 80-символьные записи. Первая запись представляет название; ее содержание используется как заголовок каждого выходного отчета. Следующее множество записей описывает правильные ответы на экзамене. Каждая запись этого множества содержит «2» в качестве последнего символа. В первой записи в колонках 1-3 задается число ответов, оно принимает значения от 1 до 999. Колонки 10-59 включают сведения о правильных ответах на вопросы с номерами 1-50, любой символ воспринимается, как ответ. Последующие записи содержат в колонках 10-59 сведения о правильных ответах на вопросы с номерами 51-100, 101-150 и т.д. Третье множество записей описывает ответы каждого студента; любая запись этого набора имеет

число «3» в восьмидесятой колонке. Для каждого студента первая запись в колонках 1-9 содержит его имя или номер (любые символы); в колонках 10-59 помещены сведения о результатах ответов студентов на вопросы с номерами 1-50. Если в тесте предусмотрено более чем 50 вопросов, то последующие записи для студента описывают ответы 51-100, 101-150 и т.д. в колонках 10-59. Максимальное число студентов – 200. Форматы входных записей показаны на рис. 4.4.

Название				
Число вопросов		Правильные ответы: 1-50		2
1 3	4 9	10 59	60 79	80
		Неправильные ответы: 51-100		2

Идентификатор студента	Ответы студента: 1-50		3
	Ответы студента: 51-100		3
Идентификатор студента	Ответы студента: 1-50		3

Рис. 4.4 – Структуры входных записей для программы MTEST

Выходными записями являются:

1) отчет, упорядоченный в лексикографическом порядке идентификаторов студентов и показывающий качество ответов каждого студента (процент правильных ответов) и его ранг;
2) аналогичный отчет, но упорядоченный по качеству;
3) отчет, показывающий среднее значение, медиану и среднеквадратическое отклонение качества ответов;
4) отчет, упорядоченный по номерам вопросов и показывающий процент студентов, отвечающих правильно на каждый вопрос.

Конец спецификации.

Начнем методичное чтение спецификации, выявляя входные условия. Первое граничное входное условие есть пустой входной файл. Второе входное условие – карта (запись) названия; граничными условиями являются отсутствие карты названия, самое короткое и самое длинное названия. Следующими входными условиями служат

наличие записей о правильных ответах и наличие поля числа вопросов в первой записи ответов. 1-999 не является классом эквивалентности для числа вопросов, так как для каждого подмножества из 50 записей может иметь место что-либо специфическое (т.е. необходимо много записей). Приемлемое разбиение вопросов на классы эквивалентности представляет разбиение на два подмножества: 1-50 и 51-999. Следовательно, необходимы тесты, где поле числа вопросов принимает значения 0, 1, 50, 51 и 999. Эти тесты покрывают большинство граничных условий для записей о правильных ответах. Однако существуют три более интересные ситуации – отсутствие записей об ответах, наличие записей об ответах типа «много ответов на один вопрос» и наличие записей об ответах типа «мало ответов на один вопрос». Например, число вопросов 60, и имеются три записи об ответах в первом случае и одна запись об ответах во втором. Таким образом, определены следующие тесты:

1. Пустой входной файл.
2. Отсутствует запись названия.
3. Название длиной в один символ.
4. Название длиной в 80 символов.
5. Экзамен из одного вопроса.
6. Экзамен из 50 вопросов.
7. Экзамен из 51 вопроса.
8. Экзамен из 999 вопросов.
9. 0 вопросов на экзамене.
10. Поле числа вопросов имеет нечисловые значения.
11. После записи названия нет записей о правильных ответах.
12. Имеются записи типа «много правильных ответов на один вопрос».
13. Имеются записи типа «мало правильных ответов на один вопрос».

Следующие входные условия относятся к ответам студентов. Тестами граничных значений в этом случае, по-видимому, должны быть:

14. 0 студентов.
15. 1 студент.
16. 200 студентов.
17. 201 студент.
18. Есть одна запись об ответе студента, но существуют две записи о правильных ответах.
19. Запись об ответе вышеупомянутого студента первая в файле.

20. Запись об ответе вышеупомянутого студента последняя в файле.
21. Есть две записи об ответах студента, но существует только одна запись о правильном ответе.
22. Запись об ответах вышеупомянутого студента первая в файле.
23. Запись об ответах вышеупомянутого студента последняя в файле.

Можно также получить набор тестов для проверки выходных границ, хотя некоторые из выходных границ (например, пустой отчет 1) покрываются приведенными тестами. Граничными условиями для отчетов 1 и 2 являются:

0 студентов (так же, как тест 14);

1 студент (так же, как тест 15);

200 студентов (так же, как тест 16);

24. Оценки качества ответов для всех студентов одинаковы.
25. Оценки качества ответов всех студентов различные.
26. Оценки качества ответов некоторых, но не всех студентов одинаковы (для проверки правильности вычисления рангов).
27. Студент получает оценку качества ответа 0.
28. Студент получает оценку качества ответа 100.
29. Студент имеет идентификатор наименьшей возможной длины (для проверки правильности упорядочивания).
30. Студент имеет идентификатор наибольшей возможной длины.
31. Число студентов таково, что отчет имеет размер, несколько больший одной страницы (для того чтобы посмотреть случай печати на другой странице).
32. Число студентов таково, что отчет располагается на одной странице.

Граничные условия отчета 3 (среднее значение, медиана, среднеквадратическое отклонение):

33. Среднее значение максимально (качество ответов всех студентов наивысшее).
34. Среднее значение равно 0 (качество ответов всех студентов равно 0).
35. Среднеквадратическое отклонение равно своему максимуму (один студент получает оценку 0, а другой – 100).
36. Среднеквадратическое отклонение равно 0 (все студенты получают одну и ту же оценку).

Тесты 33 и 34 покрывают и границы медианы. Другой полезный тест описывает ситуацию, где существует 0 студентов (проверка деления на 0 при вычислении математического ожидания), но он идентичен тесту 14.

Проверка отчета 4 дает следующие тесты граничных значений:

37. Все студенты отвечают правильно на первый вопрос.
38. Все студенты неправильно отвечают на первый вопрос.
39. Все студенты правильно отвечают на последний вопрос.
40. Все студенты отвечают на последний вопрос неправильно.
41. Число вопросов таково, что размер отчета несколько больше одной страницы.
42. Число вопросов таково, что отчет располагается на одной странице.

Опытный, программист, вероятно, согласится с той точкой зрения, что многие из этих 42 тестов позволяют выявить наличие общих ошибок, которые могут быть сделаны при разработке данной программы. Кроме того, большинство этих ошибок, вероятно, не было бы обнаружено, если бы использовался метод случайной генерации тестов или специальный метод генерации тестов. Анализ граничных значений, если он применен правильно, является одним из наиболее полезных методов проектирования тестов. Однако он часто оказывается неэффективным из-за того, его внешне выглядит простым. Необходимо понимать, что граничные условия могут быть едва уловимы и, следовательно, определение их связано с большими трудностями.

6.4.4. Применение функциональных диаграмм

Одним из недостатков анализа граничных значений и эквивалентного разбиения является то, что они не исследуют *комбинаций* входных условий. Например, пусть программа MTEST из предыдущего раздела не выполняется, если произведение числа вопросов и числа студентов превышает некоторый предел (например, объем памяти). Такая ошибка не обязательно будет обнаружена тестированием граничных значений.

Тестирование комбинаций входных условий – непростая задача, поскольку даже при построенном эквивалентном разбиении входных условий число комбинаций обычно астрономически велико. Если нет систематического способа выбора подмножества входных условий, то, как правило, выбирается произвольное подмножество, приводящее к неэффективному тесту.

Метод функциональных диаграмм или диаграмм причинно-следственных связей помогает систематически выбирать тесты с вы-

сокой результативностью. Он дает полезный побочный эффект, так как позволяет обнаруживать неполноту и неоднозначность исходных спецификаций.

Функциональная диаграмма представляет собой формальный язык, на который транслируется спецификация, написанная на естественном языке. Диаграмме можно сопоставить цифровую логическую цепь (комбинаторную логическую сеть), но для ее описания используется более простая нотация (форма записи), чем обычная форма записи, принятая в электронике. Для уяснения метода функциональных диаграмм вовсе не обязательно знание электроники, но желательно понимание булевской логики (т. е. логических операторов *и, или* и *не).* Построение тестов этим методом осуществляется в несколько этапов.

1) Спецификация разбивается на «рабочие» участки. Это связано с тем, что функциональные диаграммы становятся слишком громоздкими при применении данного метода к большим спецификациям. Например, когда тестируется система разделения времени, рабочим участком может быть спецификация отдельной команды. При тестировании компилятора в качестве рабочего участка можно рассматривать каждый отдельный оператор языка программирования.
2) В спецификации определяются причины и следствия. *Причина* есть отдельное входное условие или класс эквивалентности входных условий. *Следствие* есть выходное условие или преобразование системы (остаточное действие, которое входное условие оказывает на состояние программы или системы). Например, если сообщение программе приводит к обновлению основного файла, то изменение в нем и является преобразованием системы; подтверждающее сообщение было бы выходным условием. Причины и следствие определяются путем последовательного (слово за словом) чтения спецификации. При этом выделяются слова или фразы, которые описывают причины и следствия. Каждым причине и следствию приписывается отдельный номер.
3) Анализируется семантическое содержание спецификации, которая преобразуется в булевский граф, связывающий причины и следствия. Это и есть функциональная диаграмма.
4) Диаграмма снабжается примечаниями, задающими ограничения и описывающими комбинации причин и (или) следствий, которые являются невозможными из-за синтаксических или внешних ограничений.

5) Путем методического прослеживания состояний условий диаграммы она преобразуется в таблицу решений с ограниченными входами. Каждый столбец таблицы решений соответствует тесту.
6) Столбцы таблицы решений преобразуются в тесты.

Базовые символы для записи функциональных диаграмм показаны на рис. 4.5. Каждый узел диаграммы может находиться в двух состояниях – 0 или 1; 0 означает состояние «отсутствует», а 1 – «присутствует». Функция *тождество* устанавливает, что если значение *a* есть 1, то и значение *b* есть 1; в противном случае значение *b* есть 0. Функция *не* устанавливает, что если *a* есть 1, то *b* есть 0; в противном случае *b* есть 1. Функция *или* устанавливает, что если *a,* или *b,* или *c* есть 1, то *d* есть 1; в противном случае *d* есть 0. Функция *и* устанавливает, что если *d* и *b* есть 1, то и *c* есть 1; в противном случае *c* есть 0. Последние две функции разрешают иметь любое число входов.

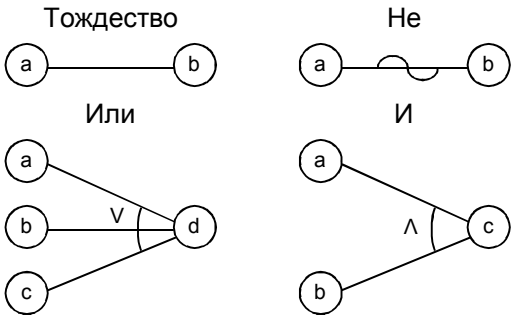


Рис. 4.5 – Базовые логические отношения функциональных диаграмм

Для иллюстрации изложенного рассмотрим диаграмму, отображающую спецификацию:

Символ в колонке 1 должен быть буквой «A» или «B», а в колонке 2 – цифрой. В этом случае файл обновляется. Если, первый символ неправильный, то выдается сообщение X12, а если второй символ неправильный – сообщение X13.

Причинами являются: 1 – символ «A» в колонке 1; 2 – символ «B» в колонке 1; 3 – цифра в колонке 2, а следствиями – 70 – файл обновляется; 71 – выдается сообщение X12; 72 – выдается сообщение X13.

Функциональная диаграмма показана на рис. 4.6. Отметим, что здесь создан промежуточный узел 11.

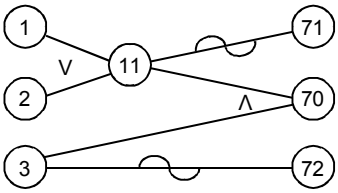


Рис. 4.6 – Пример функциональной диаграммы

На рис. 4.7 показана эквивалентная логическая схема.

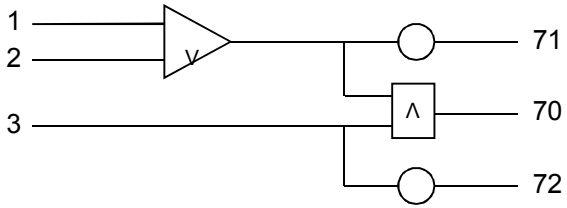


Рис. 4.7 – Логическая схема, эквивалентная диаграмме рис. 4.6

Хотя диаграмма отображает спецификацию, она содержат невозможную комбинацию причин – причины 1 и 2 не могут быть установлены в 1 одновременно. В большинстве программ определенные комбинации причин невозможны из-за синтаксических или внешних ограничений (например, символ не может принимать значения «A» и «B» одновременно).

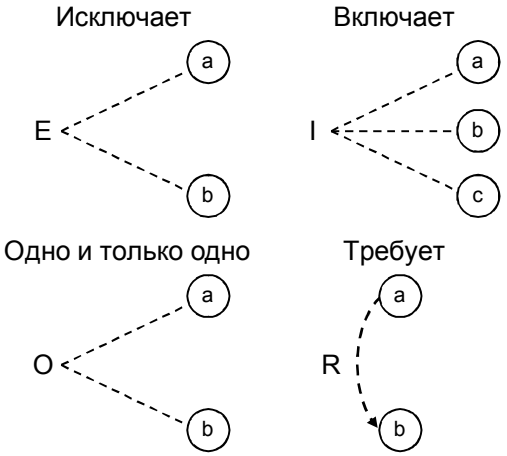


Рис. 4.8 – Символы ограничений

В этом случае используются дополнительные логические ограничения, изображенные на рис. 4.8. Ограничение *E* устанавливает, что *E* должно быть истинным, если хотя бы одна из причин – *a* или *b* – принимает значение 1 (*a* и *b* не могут принимать значение 1 одновременно). Ограничение *I* устанавливает, что, по крайней мере, одна из величин *a*, *b* или *c* всегда должна быть равной 1 (*a*, *b* и *c* не могут принимать значение 0 одновременно). Ограничение *O* устанавливает, что одна и только одна из величин *a* или *b* должна быть равна 1. Ограничение *R* устанавливает, что если *a* принимает значение 1, то и *b* должно быть равно 1 (т.е. невозможно, чтобы, *a* было равно 1, а *b* – 0).

Часто возникает необходимость в ограничениях для следствий. Ограничение *M* на рис. 4.9 устанавливает, что, если следствие *a* имеет значение 1, то следствие *b* должно иметь значение 0.

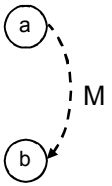


Рис. 4.9 – Символ для скрытого ограничения

Как видно из рассмотренного выше примера, физически невозможно, чтобы причины 1 и 2 присутствовали одновременно, но возможно, чтобы присутствовала одна из них. Следовательно, они связаны ограничением *E* (рис. 4.10.).

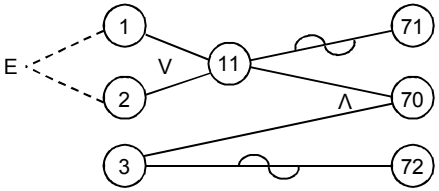


Рис. 4.10 – Пример функциональной диаграммы с ограничением «исключает»

Проиллюстрируем использование функциональных диаграмм для получения тестов. С этой целью воспользуемся спецификацией на команду отладки в интерактивной системе.

Команда DISPLAY используется для того, чтобы отобразить на экране распределение памяти. Синтаксис команды показан на рис, 4.11. Скобки представляют альтернативные необязательные операнды. Прописные буквы обозначают ключевые слова операндов, а буквы с предшествующими точками – значения операндов (т.е. действительные значения операндов, которые должны быть подставлены). Подчеркнутые операнды соответствуют стандартным значениям (т.е., если операнд опущен, то принимается стандартное значение).

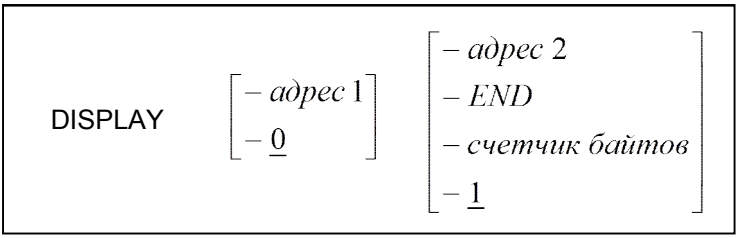


Рис. 4.11 – Синтаксис команды DISPLAY

Первый операнд (*адрес* 1) определяет адрес первого байта области памяти, содержимое которой должно быть отображено на экран. Длина адреса задается 1-6 шестнадцатеричными цифрами (0-9, A-F). Если первый операнд не определен, то предполагается, что адрес равен

0. Адрес должен принимать значение из действительной области памяти машины.

Второй операнд определяет объем памяти, который должен быть отображен. Если *адрес* 2 определен, то он, в свою очередь, определяет адрес последнего байта области памяти, которую необходимо отобразить на экран. Длина этого адреса задается также 1-6 шестнадцатеричными цифрами. Он должен быть больше или равен начальному адресу (*адрес* 1). Аналогично, *адрес* 2 обязан принимать значения из действительной области памяти машины. Если в качестве второго операнда определено *END*, то память отображается до последнего действительного адреса машины. Если же в качестве операнда определен *счетчик байтов*, то он, в свою очередь, определяет число байтов памяти, которые нужно отобразить (начиная с байта с адресом *адрес* 1). Операнд *счетчик байтов* является шестнадцатеричным целым числом (длиной от одной до шести цифр). Сумма значений операндов *счетчик байтов* и *адрес* 1 не должна превышать действительного размера памяти плюс единица, *а счетчик байтов* должен, по крайней мере, иметь значение 1. Состояние памяти отображается на экран терминала в виде одной или нескольких строк следующего формата:

xxxxxx = слово 1 слова 2 слово 3 слово 4,

где xxxxxx есть шестнадцатеричный адрес слова 1.

Всегда отображается полное число слов (четырехбайтовых последовательностей, где адрес первого байта в слове кратен четырем), независимо от значения операнда *адрес* 1 или отображаемого объема памяти. Все выходные строки всегда содержат четыре слова (16 байт). Первый байт отображаемой области памяти находится в пределах первого слова.

Могут иметь место следующие сообщения об ошибках:

M1 – НЕПРАВИЛЬНЫЙ СИНТАКСИС КОМАНДЫ
M2 – ЗАПРАШИВАЕТСЯ АДРЕС, БОЛЬШИЙ ДОПУСТИМОГО
M3 – ЗАПРАШИВАЕТСЯ ОБЛАСТЬ ПАМЯТИ С НУЛЕВЫМ ИЛИ ОТРИЦАТЕЛЬНЫМ АДРЕСОМ

Примеры команды DISPLAY:

DISPLAY

отображает первые четыре слова в памяти (стандартное значение начального адреса 0, а стандартное значение счетчика байтов 1).

DISPLAY 77F

отображает слово, содержащее байт с адресом 77F, и три последующих слова;

DISPLAY 77F – 407A

отображает слова, содержащие байты с адресами от 77F до 407A.

DISPLAY 77F.6

отображает слова, содержащие шесть байт, начиная с адреса 77F.

DISPLAY 50FF – END

отображает слова, содержащие байты с адреса 50FF до конца памяти.

Первый шаг заключается в тщательном анализе спецификации с тем, чтобы идентифицировать причины и следствия. Причинами являются:

1. Наличие первого операнда.
2. Операнд *адрес* 1 содержит только шестнадцатеричные цифры.
3. Операнд *адрес* 1 содержит от одного до шести символов.
4. Операнд *адрес* 1 находится в пределах действительной области памяти.
5. Второй операнд есть *END*.
6. Второй операнд есть *адрес* 2.
7. Второй операнд есть *счетчик байтов*.
8. Второй операнд отсутствует.
9. Операнд *адрес* 2 содержит только шестнадцатеричные цифры.
10. Операнд *адрес* 2 содержит от одного до шести символов.
11. Операнд *адрес* 2 находится в пределах действительной области памяти.
12. Операнд *адрес* 2 больше или равен операнду *адрес* 1.
13. Операнд *счетчик байтов* содержит только шестнадцатеричные цифры.
14. Операнд *счетчик байтов* содержит от одного до шести символов.
15. *Счетчик байтов* + *адрес* 1 ≤ *размер памяти* + 1.
16. *Счетчик байтов* ≥ 1.
17. Запрашиваемая область памяти настолько велика, что требуется много строк на экране.
18. Начало области не выровнено на границу слова.

Каждой причине соответствует произвольный единственный номер. Заметим, что для описания второго операнда необходимы четыре причины (5-8), так как второй операнд может принимать значения 1) *END*, 2) *адрес* 2, 3) *счетчик байтов*, 4) может отсутствовать и 5) неопределенное значение, т.е. ни одно из указанных выше.

Следствия:

91. На экран отображается сообщение M1.

92. На экран отображается сообщение М2.
93. На экран отображается сообщение М3.
94. Память отображается на одной строке.
95. Для отображения состояния памяти требуется много строк.
96. Первый байт отображаемой области памяти выровнен на границу слова.
97. Первый байт отображаемой области памяти не выровнен на границу слова.

Второй шаг – разработка функциональной диаграммы.

Узлы причин перечислены по вертикали у левого края страницы; узлы следствий собраны по вертикали у ее правого края. Тщательно анализируется семантическое содержание спецификации с тем, чтобы связать причины и следствия (т.е. показать, при каких условиях имеет место следствие).

На рис. 4.12 приведена начальная версия диаграммы. Промежуточный узел 32 представляет синтаксически правильный первый операнд, узел 35 – синтаксически правильный второй операнд, а узел 36 – синтаксически правильную команду. Если значение узла 36 есть 1, то следствие 91 (сообщение об ошибке) отсутствует. Если значение узла 36 есть 0, то следствие 91 имеет место.

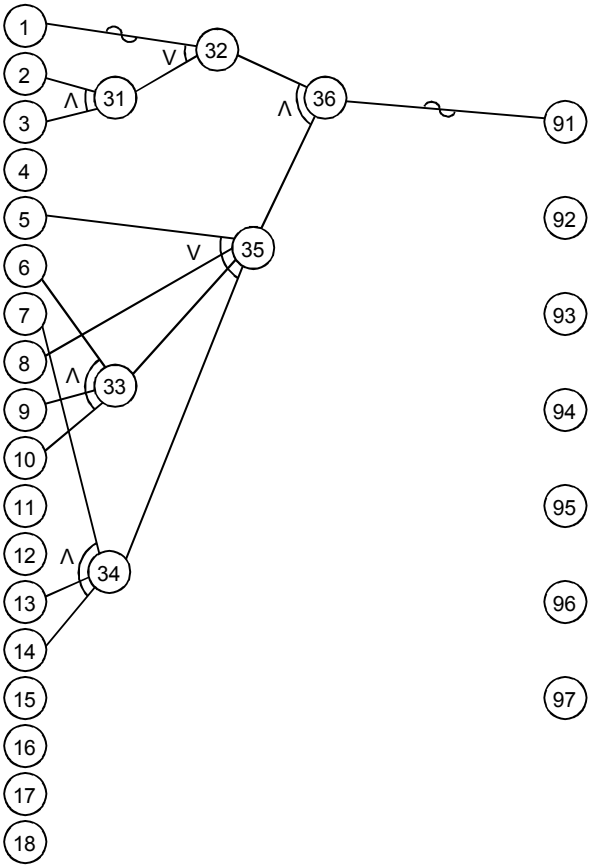


Рис. 4.12 – Начальная версия функциональной диаграммы команды DISPLAY

На рис. 4.13 изображена полная функциональная диаграмма. Если диаграмму на рис. 4.13 непосредственно использовать для построения тестов, то создание многих из них на самом деле окажется невозможным. Это объясняется тем, что определенные комбинации причин *не могут* иметь место из-за синтаксических ограничений.

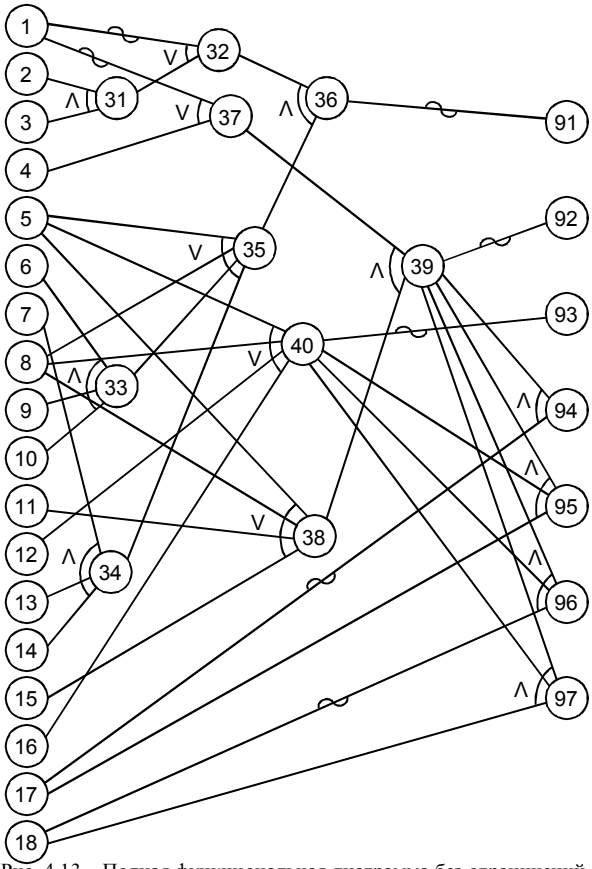


Рис. 4.13 – Полная функциональная диаграмма без ограничений

Например, причины 2 и 3 не могут присутствовать без причины 1. Причина 4 не может присутствовать, если нет причин 2 и 3. На рис. 4.14 показана окончательная диаграмма со всеми дополнительными ограничениям. Заметим, что может присутствовать только одна из причин 5, 6, 7 или 8. Другие ограничения причин являются условиями типа «*требует*». Причина 17 (много строк на экране) и причина 8 (второй операнд отсутствует) связаны отношением *не*; причина 17 может присутствовать только в отсутствии причины 8.

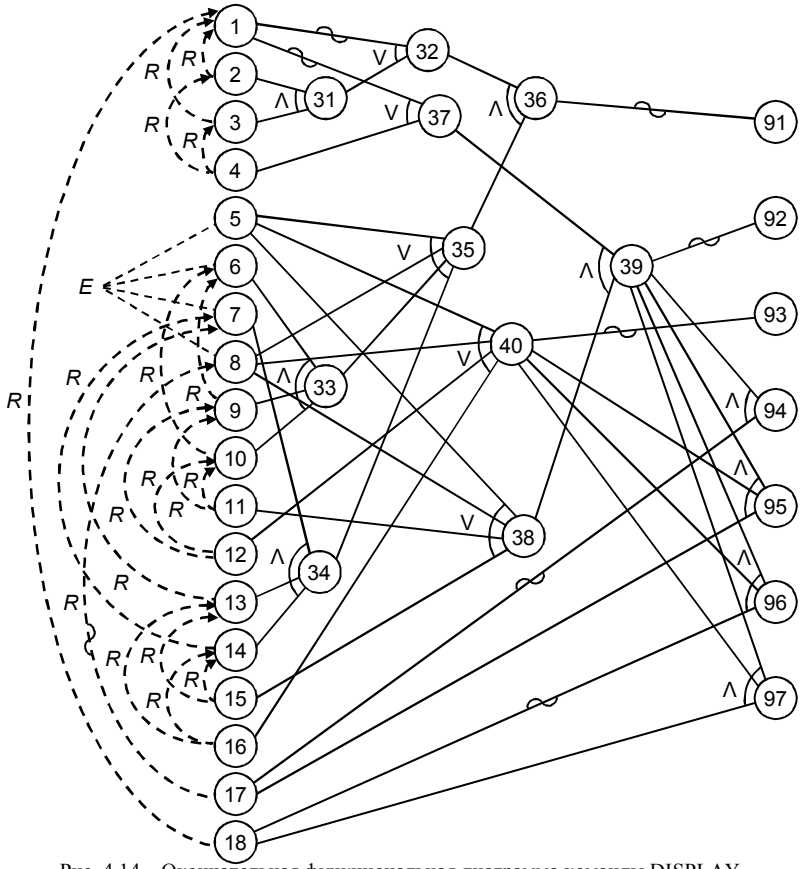


Рис. 4.14 – Окончательная функциональная диаграмма команды DISPLAY

Третьим шагом является генерация таблицы решений с ограниченными входами. В таблице решений причины есть условия, а следствия есть действия. Процедура генерации заключается в следующем:

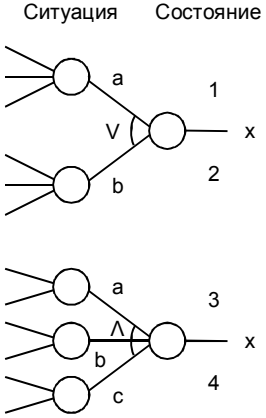
1) Выбрать некоторое следствие, которое должно быть в состоянии 1.
2) Найти все комбинации причин (с учетом ограничений), которые установят это следствие в 1, прокладывая из этого следствия обратную трассу через диаграмму.
3) Построить столбец в таблице решений для каждой комбинации причин.

4) Для каждой комбинации причин определить состояния всех других следствий и поместить их в соответствующий столбец таблицы решений.

При выполнении этого шага необходимо руководствоваться тремя положениями:

1) Если обратная трасса прокладывается через узел *или,* выход которого должен принимать значение 1, то одновременно не следует устанавливать в 1 более одного входа в этот узел. Такое ограничение на установку входных значений называется *чувствительностью* пути. Цель данного правила – избежать пропуска отдельных ошибок из-за того, что одна причина маскируется другой.
2) Если обратная трасса прокладывается через узел *и,* выход которого, должен принимать значение 0, то все комбинации входов, приводящие выход в 0, должны быть перечислены. Однако когда исследуется ситуация, где один вход есть 0, а один или более других входов есть 1, не обязательно перечислять все условия, при которых остальные входы могут быть 1.
3) Если обратная трасса прокладывается через узел *и*, выход которого должен принимать значение 0, то необходимо указать лишь одно условие, согласно которому *все* входы являются нулями. Когда узел *и* находится в середине графа, и его входы исходят из других промежуточных узлов, может существовать чрезвычайно большое число ситуаций, при которых все его входы принимают значения 0.

Эти положения кратко поясняются рис. 4.15. Рис. 4.16 приведен в качестве примера функциональной диаграммы. Пусть требуется так задать входные условия, чтобы установить выходное состояние в 0. Согласно положению 3, следует рассматривать только один случай, когда узлы 5 и 6 – нули. По положению 2, для состояния, при котором узел 5 принимает значение 1, а узел 6 – значение 0, следует рассматривать только один случай, когда узел 5 принимает значение 1 (не перечисляя другие возможные случаи, когда узел 5 может принимать значение 1).



1. Если *X* должен быть равен 1, то не следует рассматривать ситуацию, где *a* = *b* = 1 (положение 1).
2. Если *X* должен быть равен 0, то перечислить все ситуации, где *a* = *b* = 0.
3. Если *X* должен быть равен 1, то перечислить все ситуации, где *a* = *b* = *c* = 1.
4. Если *X* должен быть равен 0, то рассматриваем только одну ситуацию, где *a* = *b* = *c* = 0 (положение 3).

Для состояний *a*, *b* и *c* 001, 010, 100, 011, 101 и 110 рассмотреть только одну, любую из этих ситуаций (положение 2).

Рис. 4.15 – Положения, используемые при прокладке обратной трассы через диаграмму

Аналогично для состояния, при котором узел 5 принимает значение 0, а узел 6 – значение 1 следует рассматривать только один случай, когда узел 6 принимает значение 1 (хотя в данном случае он является единственным). В соответствии с положением 1, если узел 5 должен быть установлен в состояние 1, то не рекомендуется устанавливать узлы 1 и 2 в состояние 1 одновременно. Таким образом, возможны пять состояний узлов 1-4, например, значения

0 0 0 0 (5 = 0, 6 = 0),
1 0 0 0 (5 = 1, 6 = 0),
1 0 0 1 (5 = 1, 6 = 0),
1 0 1 0 (5 = 1, 6 = 0),
0 0 1 1 (5 = 0, 6 = 1),

а не 13, которые приводят к выходному состоянию 0.

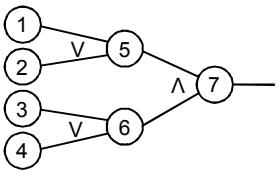


Рис. 4.16 – Пример функциональной диаграммы для иллюстрации обратной трассировки

На первый взгляд, эти положения могут показаться необъективными, но они преследуют важную цель: уменьшить комбинаторику

диаграммы. Их применение позволяет избежать ситуаций, которые приводят к получению малорезультативных тестов. Если не исключать малорезультативные тесты, то общее число тестов, порождаемых по большой функциональной диаграмме, получается астрономическим. Если же и без них число тестов все еще оказывается большим, то выбирается некоторое подмножество тестов, при этом не гарантируется, что малорезультативные тесты будут исключены. Поэтому самое лучшее – исключить их в процессе анализа диаграммы.

Преобразуем функциональную диаграмму, изображенную на рис. 4.14, в таблицу решений. Выберем первым следствием 91. Следствие 91 имеет место, если узел 36 принимает значение 0. Значение узла 36 есть 0, значение узлов 32 и 35 есть 0,0, 0,1 или 1,0 и применимы положения 2 и 3. Хотя подобное преобразование – трудоемкий процесс, но можно проложить обратную трассу, от следствий к причинам, учесть при этом ограничения причин и найти комбинации последних, которые приводят к следствию 91.

Результирующая таблица решений, при условии, что имеет место следствие 91, показана на рис. 4.17 (столбцы 1-11). Столбцы (тесты) 1-3 представляют условия, где узел 32 есть 0, а узел 35 есть 1. Столбцы 4-10 представляют условия, где узел 32 есть 1, а узел 35 есть 0. С помощью положения 3 определена только одна ситуация (колонка 11) из 21 возможной, когда узлы 32 и 35 есть 0. Пробелы представляют «безразличные» ситуации (т.е. состояние причины несущественно) или указывают на то, что состояние причины очевидно вследствие состояний других зависимых причин (например, для столбца 1 известно, что причины 5, 7 и 8 должны принимать значения 0, так как они связаны ограничением «одно и только одно» с причиной 6).

Столбцы 12-15 представляют ситуации, при которых имеет место следствие 92, а столбцы 16 и 17 – ситуации, при которых имеет место следствие 93. На рис. 4.18 показана остальная часть таблицы решений.

Последний шаг заключается в том, чтобы преобразовать таблицу решений в 38 тестов. Набор из 38 тестов представлен ниже.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4												1	1	0	0	1	1	1	1
5				0										1				1	
6	1	1	1	0	1	1	1				1	1			1	1			
7				0				1	1	1			1				1		
8				0															1
9	1	1	1		1	0	0				0	1			1	1			
10	1	1	1		0	1	0				1	1			1	1			
11												0			0	1			
12																0			
13								1	0	0			1				1		
14								0	1	0			1				1		
15													0				0		
16																			
17																		0	0
18																		1	1
91	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
94	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рис. 4.17 – Первая половина результирующей таблицы решений

	20	21	22	23	24	25	27	28	29	30	31	32	33	34	35	36	37	38
1	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	1	1	1
2	1	1					1	1	1	1	1	1				1	1	1
3	1	1					1	1	1	1	1	1				1	1	1
4	1	1					1	1	1	1	1	1				1	1	1
5			1							1			1			1		
6	1				1			1			1			1			1	
7		1				1			1			1			1			1
8				1			1											
9	1				1						1			1		1		
10	1				1						1			1		1		
11	1				1						1			1		1		
12	1				1						1			1		1		
13		1				1		0	1			1			1		1	1
14		1				1		0	1			1			1		1	1
15		1				1			1			1			1		1	1
16		1				1			1			1			1		1	1
17	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
18	1	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
94	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
95	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
96	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
97	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0

Рис. 4.18 – Вторая половина результирующей таблицы решений

Числа возле каждого теста обозначают следствия, которые, как ожидается, должны здесь иметь место. Предположим, что последний используемый адрес памяти машины есть 7FFF.

1.	DISPLAY 24AF74 – 123	(91)
2.	DISPLAY 2ZX4 – 3000	(91)
3.	DISPLAY HHHHHHHH – 200	(91)
4.	DISPLAY 200 200	(91)
5.	DISPLAY 0 – 22222222	(91)
6.	DISPLAY 1 – 2X	(91)
7.	DISPLAY 2 – ABCDEFGHI	(91)
8.	DISPLAY 3.11111111	(91)
9.	DISPLAY 44.$42	(91)
10.	DISPLAY 100.$$$$$$$	(91)
11.	DISPLAY 10000000 – M	(91)
12.	DISPLAY FF – 8000	(92)
13.	DISPLAY FFF.7001	(92)
14.	DISPLAY 8000 – END	(92)
15.	DISPLAY 8000 – 8001	(92)
16.	DISPLAY AA-A0	(93)
17.	DISPLAY 7000.0	(93)
18.	DISPLAY 7FF9 – END	(94,97)
19.	DISPLAY 1	(94,97)
20.	DISPLAY 21 – 29	(94,97)
21.	DISPLAY 4021.A	(94,97)
22.	DISPLAY – END	(94,96)
23.	DISPLAY	(94,96)
24.	DISPLAY – F	(94,96)
25.	DISPLAY .E	(94,96)
26.	DISPLAY 7FF8 – END	(94,96)
27.	DISPLAY 6000	(94,96)
28.	DISPLAY A0 – A4	(94,96)
29.	DISPLAY 20.8	(94,96)
30.	DISPLAY 7001 – END	(95,97)
31.	DISPLAY 5 – 15	(95,97)
32.	DISPLAY 4FF.100	(95,97)
33.	DISPLAY – END	(95,96)
34.	DISPLAY –20	(95,96)
35.	DISPLAY .11	(95,96)
36.	DISPLAY 7000 – END	(95,96)
37.	DISPLAY 4 –14	(95,96)
38.	DISPLAY 500.11	(95,96)

Заметим, что в этом случае двум и более различным тестам соответствует один и тот же набор причин, следует стараться выбирать различные значения причин с тем, чтобы хотя бы незначительно улучшить результативность тестов. Заметим также, что из-за ограниченного размера памяти тест 22 является нереализуемым (при его использовании будет получено следствие 95 вместо 94, как отмечено в тесте 33). Следовательно, реализуемы только 37 тестов.

Замечания.

Применение функциональных диаграмм – систематический метод генерации тестов, представляющих комбинации условий. Альтернативой является специальный выбор комбинаций, но при этом существует вероятность пропуска многих «интересных» тестов, определенных с помощью функциональной диаграммы.

При использовании функциональных диаграмм требуется трансляция спецификации в булевскую логическую сеть. Следовательно, этот метод открывает перспективы ее применения и дополнительные возможности спецификаций. Действительно, разработка функциональных диаграмм есть хороший способ обнаружения неполноты и неоднозначности в исходных спецификациях. Например, можно заметить, что предложенный процесс обнаруживает неполноту в спецификации команды DISPLAY. Спецификация устанавливает, что все выходные строки содержат четыре слова. Это справедливо не во всех случаях; так, для тестов 18 и 26 это неверно, поскольку для них начальный адрес отображаемой памяти отличается от конечного адреса памяти машины менее чем на 16 байт.

Метод функциональных диаграмм позволяет построить набор полезных тестов, однако его применение обычно не обеспечивает построение *всех* полезных тестов, которые могут быть определены. Так, в нашем примере мы ничего не сказали о проверке идентичности отображаемых на экран терминала значений данных данным в памяти и об установлении случая, когда программа может отображать на экран любое возможное значение, хранящееся в ячейке памяти. Кроме того, функциональная диаграмма неадекватно исследует граничные условия. Конечно, в процессе работы функциональными диаграммами можно попробовать покрыть граничные условия. Например, вместо определения единственной причины

адрес 2 ≥ адрес 1

можно определить две причины:

адрес 2 > адрес 1

адрес 2 = адрес 1

Однако, при этом граф существенно усложняется, и число тестов становится чрезвычайно большим. Поэтому, лучше отделить анализ граничных значений от метода функциональных диаграмм. Например, для спецификации команды DISPLAY могут быть определены следующие граничные условия:

1. *Адрес* 1 длиной в одну цифру.
2. *Адрес* 1 длиной в шесть цифр.
3. *Адрес* 1 длиной в семь цифр.
4. *Адрес* 1 = 0.
5. *Адрес* 1 = 7FFF.
6. *Адрес* 1 = 8000.
7. *Адрес* 2 длиной в одну цифру.
8. *Адрес* 2 длиной в шесть цифр.
9. *Адрес* 2 длиной в семь цифр.
10. *Адрес* 2 = 0.
11. *Адрес* 2 = 7FFF.
12. *Адрес* 2 = 8000.
13. *Адрес* 2 = *адрес* 1.
14. *Адрес* 2 = *адрес* 1.
15. *Адрес* 2 = *адрес* 1 – 1.
16. *Счетчик байтов* длиной в одну цифру.
17. *Счетчик байтов* длиной в шесть цифр.
18. *Счетчик байтов* длиной в семь цифр.
19. *Счетчик байтов* = 1.
20. *Адрес* 1 + *счетчик байтов* = 8000.
21. *Адрес* 1 + *счетчик байтов* = 8001.
22. Отображение шестнадцати байт (одна строка).
23. Отображение семнадцати байт (две строки).

Это вовсе не означает, что следует писать 60 (37 + 23) тестов. Поскольку функциональная диаграмма дает только направление в выборе определенных значений операндов, граничные условия могут входить в полученные из нее тесты. В нашем примере, переписывая некоторые из первоначальных 37 тестов, можно покрыть все 23 граничных условия без дополнительных тестов. Таким образом, мы получаем небольшой, но убедительный набор тестов, удовлетворяющий поставленным целям.

Заметим, что метод функциональных диаграмм согласуется некоторыми принципами тестирования, изложенными в п. 6.2. Его неотъемлемой частью является определение ожидаемого выхода каждого теста (все столбцы в таблице решений обозначают ожидаемые следствия). Заметим также, что данный метод помогает выявить оши-

бочные побочные следствия. Например, столбец (тест) 1 устанавливает, что должно присутствовать следствие 91 и что следствия 92-97 должны отсутствовать.

Наиболее трудным при реализации метода является преобразование диаграммы в таблицу решений. Это преобразование представляет собой алгоритмический процесс. Следовательно, его можно автоматизировать посредством написания соответствующей программы. Фирма IBM имеет ряд таких программ, но не поставляет их.

6.4.5. Предположение об ошибке

Замечено, что некоторые люди по своим качествам оказываются прекрасными специалистами по тестированию программ. Они обладают умением «выискивать» ошибки и без привлечения какой-либо методологии тестирования (такой, как анализ граничных значений или применение функциональных диаграмм).

Объясняется это тем, что человек, обладающий практическим опытом, часто подсознательно применяет метод проектирования тестов, называемый предположением об ошибке. При наличии определенной программы он интуитивно предполагает вероятные типы ошибок и затем разрабатывает тесты для их обнаружения.

Процедуру для метода предположения об ошибке описать трудно, так как он в значительной степени является интуитивным. Основная идея его заключается в том, чтобы перечислить в некотором списке возможные ошибки или ситуации, в которых они могут появиться, а затем на основе этого списка написать тесты. Например, такая ситуация возникает при значении 0 на входе и выходе программы. Следовательно, можно построить тесты, для которых определенные входные данные имеют нулевые значения и для которых определенные выходные данные устанавливаются на 0. При переменном числе входов или выходов (например, число искомых входных записей при поиске в списке) ошибки возможны в ситуациях типа «никакой» и «один» (например, пустой список, список, содержащий только одну искомую запись). Другая идея состоит в том, чтобы определить тесты, связанные с предположениями, которые программист может сделать во время чтения спецификаций (т.е. моменты, которые были опущены из спецификации либо случайно, либо из-за того, что автор спецификации считал их очевидными).

Поскольку данная процедура не может быть четко определена, лучшим способом обсуждения смысла предположения об ошибке представляется разбор примеров. Если в качестве примера рассмотреть

тестирование подпрограммы сортировки, то нужно исследовать следующие ситуации:

1. Сортируемый список пуст.
2. Сортируемый список содержит только одно значение.
3. Все записи в сортируемом списке имеют одно и тоже значение.
4. Список уже отсортирован.

Другими словами, требуется перечислить те специальные случаи, которые могут быть не учтены при проектировании программы. Если пример заключается в тестировании подпрограммы двоичного поиска, то можно проверить следующие ситуации:

1) существует только один вход в таблицу, в которой ведется поиск;
2) размер таблицы есть степень двух (например, 16);
3) размер таблицы меньше или больше степени двух (например, 15 или 17).

Рассмотрим программу MTEST, приведенную в разделе, посвященном анализу граничных значений. При тестировании этой программы методом предположения об ошибке целесообразно учесть следующие дополнительные тесты:

1. Допускает ли программа «пробел» в качестве ответа?
2. Запись типа 2 (ответ) появляется в наборе записей типа 3 (студент).
3. Запись без 2 или 3 в последней колонке появляется не как начальная запись (название).
4. Два студента имеют одно и то же имя или номер.
5. Поскольку медиана вычисляется по-разному в зависимости от того, четно или нечетно число элементов, необходимо протестировать программу как для четного, так и для нечетного числа студентов.
6. Поле числа вопросов имеет отрицательное значение.

Для команды DISPLAY из предыдущего раздела целесообразно рассмотреть следующие тесты метода предположения об ошибке:

1. DISPLAY 100 – (неполный второй операнд).
2. DISPLAY 100. (неполный второй операнд).
3. DISPLAY 100 – 10A42 (слишком большое значение операнда).
4. DISPLAY 000 – 0000FF (нули слева).

6.4.6. Стратегия

Методологии проектирования тестов, обсуждавшиеся в этом разделе, могут быть объединены в общую стратегию. Причина объединения их теперь становится очевидной: каждый метод обеспечивает создание определенного набора используемых тестов, но ни один из них сам по себе не может дать полный набор тестов. Приемлемая стратегия состоит в следующем:

1. Если спецификация содержит комбинации входных условий, то начать рекомендуется с применения метода функциональных диаграмм.

2. В любом случае, необходимо использовать анализ граничных значений. Напомним, что этот метод включает анализ граничных значений входных и выходных переменных. Анализ граничных значений дает набор дополнительных тестовых условий, но многие из них (если не все) могут быть включены в тесты метода функциональных диаграмм.

3. Определить правильные и неправильные классы эквивалентности для входных и выходных данных и дополнить, если это необходимо, тесты, построенные на предыдущих шагах.

4. Для получения дополнительных тестов рекомендуется использовать метод предположения об ошибке.

5. Проверить логику программы на полученном наборе тестов. Для этого нужно воспользоваться критерием покрытия решений, покрытия условий, покрытия решений/условий либо комбинаторного покрытия условий (последний критерий является более полным). Если необходимость выполнения критерия покрытия приводит к построению тестов, не встречающихся среди построенных на предыдущих четырех шагах, и если этот критерий не является нереализуемым (т.е. определенные комбинации условий невозможно создать вследствие природы программы), то следует дополнить уже построенный набор тестов тестами, число которых достаточно для удовлетворения критерия покрытия.

Эта стратегия, опять-таки, не гарантирует, что все ошибки будут найдены, но, вместе с тем, ее применение обеспечивает приемлемый компромисс. Реализация подобной стратегии весьма трудоемка, но ведь никто и никогда не утверждал, что тестирование программы – легкое дело.

Контрольные вопросы

1) Определение процесса тестирования. Хороший тест. Хороших прогон.

2) Тестирование программы как белого и черного ящика.
3) Принципы тестирования.
4) Технология ручного тестирования. Инспекция исходного текста. Сквозные просмотры. Метод оценки программ посредством просмотра.
5) Принципы проектирования тестов.
6) Технология тестирования по принципу белого ящика. Покрытие операторов. Покрытие решений. Покрытие условий. Покрытие решений/условий. Комбинаторное покрытие.
7) Технология тестирования по принципу черного ящика.
8) Эквивалентное разбиение. Способы формирования классов эквивалентности. Правила создания тестов по классам эквивалентности.
9) Анализ граничных условий.
10) Применение функциональных диаграмм. Способы ограничений на вход и выход. Технология построения функциональных диаграмм. Технология построения таблиц решений. Формирование тестов по таблице решений.
11) Предположение об ошибках.
12) Стратегия тестирования.

7. ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММ

Хотя разработка программ является в основном творческой деятельностью, существует множество стандартных алгоритмов, которые могут применяться для упрощения данного процесса. Знание этих алгоритмов часто облегчает формулирование задачи.

7.1. Разбиение задачи на независимые подзадачи

Основным алгоритмом, используемым для решения задач, является алгоритм разбиения на независимые подзадачи. Для того, чтобы решить задачу A, ее первоначально необходимо разбить на независимые подзадачи B, C, D и т.д.

```
A: procedure;
   Задача B;
   Задача C;
   Задача D;
end A;
```

Решение подзадач может производиться по любому алгоритму.

7.2. Разбиение задачи на одинаковые по сложности части

Этот метод, как и первый, наиболее часто используется в программировании. Данный подход означает разделение задачи на подзадачи, равные, примерно, по сложности, для того чтобы перейти к решению значительно более простых задач, чем первоначальная. Поиск нужного элемента в таблице является характерным примером такого алгоритма:

- проверить первый элемент;
- **if** заданный элемент не найден, **then** продолжить поиск среди оставшихся элементов.

Таким образом, задача разбивается на две части: проверку первого элемента и поиск среди оставшихся. Ясно, что такие алгоритмы не равнозначны по сложности. Модифицированный алгоритм – двоичный поиск – имеет вид:

- проверить первый элемент;
- **if** заданный элемент не найден, **then** продолжить поиск в верхней или нижней половине списка.

В этом случае задача разбивается на две части, примерно одинаковые по сложности. Каждая из частей может иметь рекурсивное решение.

7.3. Рекурсия и динамическое программирование

7.3.1. Рекурсия

Иногда возможно сформулировать решение задачи как известное преобразование другого, более простого, варианта той же задачи. Если этот процесс продолжить до тех пор, пока не получится нужное значение, то подобное решение называется рекурсивным.

Примером рекурсивного решения является вычисление факториала fact(N) = 1×2×…×N. Пусть неизвестно значение fact(460), если умножить fact(459) на 460, тогда задача сводится к вычислению fact(459). Если это преобразование далее повторить 458 раз, то можно вычислить значение факториала, поскольку fact(1) = 1. Таким образом, функция F является рекурсивной, если:

1) $F(N) = G(N, F(N-1))$ для некоторой известной функции G и
2) $F(1)$ равно некоторому известному значению.

Функция F может быть функцией нескольких параметров. Значение N должно быть задано в явном виде. Величина N может быть элементом во множестве, длиной интервала или некоторым другим параметром.

Рекурсивное решение всегда записывается проще, чем соответствующий нерекурсивный вариант. Относительно скорости решения однозначных выводов сделать нельзя. Это зависит от сложности функции G.

7.3.2. Динамическое программирование

Динамическое программирование – это табличный метод, при котором используется одна и та же схема вычисления всех подзадач данной задачи. При использовании этого метода однажды найденный результат записывается в таблицу и далее повторно не вычисляется.

7.3.3. Моделирование

Часто бывает невозможно получить точное решение задачи по соображениям стоимости, сложности или размера. Такие ситуации возникают при решении задач по управлению движением, экономическому прогнозированию и др. В этих случаях строится представление определенных свойств задачи, называемой моделью, а все другие свойства не учитываются. При этом проводится анализ поведения нужных свойств модели. Такой подход называется моделированием. Моделирование применяется во многих областях, однако при разработке программного обеспечения моделирование используется доста-

точно редко. В основном моделирование используется при исследовании на ЭВМ различных физических процессов, особенно когда известно математическое описание последних. Тем не менее, данный подход заслуживает пристального внимания программистов при разработке больших программных комплексов.

7.4. Поиск

Поиск – это процесс нахождения нужных значений в некотором множестве. Семейство элементов может быть структурированным либо нет. Если оно не структурировано, поиск обычно означает нахождение нужного элемента в списке. Если семейство элементов структурировано (дерево, граф и др.), то под поиском понимают нахождение правильного пути в структуре.

Обычно каждый элемент имеет три компонента:

- *ключ* – это элемент, используемый для доступа к данным (он может быть именем в справочнике, словом в индексе или названием раздела в таблице);
- *аргумен*т – это числа, связанные с элементом (адрес, номер телефона и др.);
- *структуру* – это дополнительная информация, принимаемая для описания данных, такая как адрес следующего элемента в семействе и т.п.

7.4.1. Поиск в списках

Данный поиск проводится четырьмя различными способами: прямым, линейным, двоичным и хешированием.

Прямой поиск. При прямом поиске местоположение элемента определяется непосредственно с помощью ключа (комната в здании). Прямой поиск применяется тогда, когда число элементов фиксировано и их сравнительно немного.

Линейный поиск. Данный вид поиска наиболее простой для программной реализации, хотя его выполнение занимает много времени. Элементы проверяются последовательно, по одному, пока нужный элемент не будет найден. В отличие от прямого поиска, здесь нет системы в расположении элементов. Если список содержит N элементов, то каждый раз в среднем будут проверяться $N/2$ элементов. Линейный поиск медленный, но позволяет легко добавлять новые элементы, помещая их в конец списка. Хорошо разработанная программа обычно имеет простую (линейную) структуру, и, следовательно, такую же

структуру должны иметь и данные, поэтому линейный поиск в таких структурах эффективен.

Двоичный поиск. Для ведения двоичного поиска ключи должны быть упорядочены по величине или алфавиту. В этом случае можно добиться среднего времени поиска $\log_2 N$. При двоичном поиске ключ сравнивается с ключом среднего элемента списка. Если значение ключа больше, то та же самая процедура повторятся для второй половины списка, если же меньше, то для первой. Таким образом, за шаг этого процесса список уменьшается наполовину.

Основная трудность при двоичном поиске – как вводить новые элементы. Так как список упорядочивается, то введение нового элемента может вызвать переписывание всех элементов для того, чтобы новый элемент поместить в нужном месте.

Хеш-поиск. Этот вид поиска является попыткой применить прямой поиск для поиска в больших наборах данных. Из изначального значения ключа вычисляется значение псевдоключа, называемого хеш-кодом, и этот код используется как индекс в таблице адресов элементов. Если все ключи соответствуют разным хеш-кодам, то любой элемент будет найден за один прямой поиск.

Пример. В абстрактном компиляторе может быть отведено 1000 мест для имен переменных. В языке допускаются слова из 6 символов. Это позволяет создавать 26^6 имен, что гораздо больше, чем размер таблицы символов. Но если каждое число, представляющее имя, уменьшить до числа, находящегося между 0 и 999, то имена могут быть записаны в таблицу. Алгоритмы вычисления хеш-кодов различны, например, имя (ключ) можно рассматривать как двоичное число. Это число можно разделить на размер таблицы, остаток использовать как код. Другие алгоритмы предлагают сложение битов внутри числа или некоторые другие функции с ключом.

Недостатком хеш-поиска является тот факт, что некоторые идентификаторы могут иметь одинаковые хеш-коды, поэтому возникает проблема обработки пересечений элементов, т.е. элементов, имеющих одинаковые хеш-коды. Когда определенное значение хеш-кода встречается первый раз, его помещают в определенное место в таблице. Если произошло пересечение с другим элементом, его помещают в другое место в соответствии со следующим алгоритмом:

```
INSERT: procedure (ключ, аргумент);
  declare 1 TABLE(размер),
          2 KEY,
          2 ARGUMENT;
  declare хеш–код;
```

```
  Вычислить хеш–код;
  /* если запись находится в таблице */
  if TABLE(хеш-код).KEY = ключ then return;
  /* обработка совпадающих хеш-кодов */
  do while (TABLE(хеш-код).KEY≠0 &
          TABLE(хеш-код).KEY≠ключ);
    хеш–код = новое значение, зависящее от
           (хеш–код, ключ);
  end;
  TABLE(хеш-код).KEY = ключ;
  TABLE(хеш-код).ARGUMENT = аргумент;
end INSERT;
```

Таким образом, в случае пересечения элементов имеется переход к следующему месту в таблице. Если нужное место занято, – то переход на место, расположенное на некотором расстоянии от занятого, или вычисление нового хеш-кода, исходя из имеющегося ключа и кода.

Основным недостатком хеш-поиска является тот факт, что при плотном заполнении таблицы алгоритм ввода элементов весьма трудоемок.

7.4.2. Деревья поиска

Если данные организованы как структура дерева, алгоритм поиска сводится к алгоритму, просматривающему все узлы дерева. Для ведения поиска существуют два основных алгоритма поиска: поиск в глубину и поиск в ширину.

Поиск в глубину. При поиске в глубину каждая ветвь дерева просматривается слева направо.

Для двоичных деревьев алгоритм поиска имеет простую рекурсивную структуру (рис. 7.1):

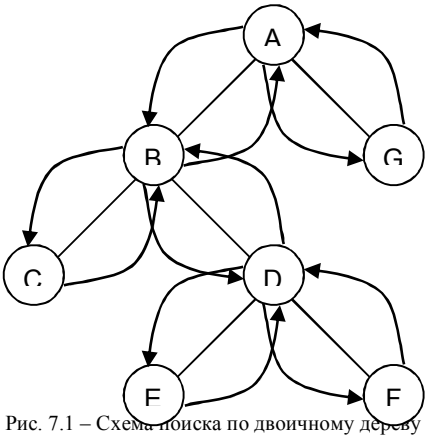


Рис. 7.1 – Схема поиска по двоичному дереву

```
DEPTFIRST: procedure(TREE);
 declare 1 TREE,
         2 KEY,
         2 ARGUMENT, /* данные */
         2 RightSon, /* ноль, если нет сына*/
         2 LtftSon;  /* ноль, если нет сына */
 if TREE = null then return;
 if текущий узел не KEY then do;
   call DEPTFIRST(LtftSon);
   call DEPTFIRST(RightSon);
 end;
end DEPTFIRST;
```

Заметим, что если дерево упорядочено таким образом, что LtftSon имеет ключ, величина которого меньше, чем значение корневого узла, а величина ключа для RightSon больше, чем значение корневого узла, то такой алгоритм аналогичен алгоритму двоичного поиска.

Поиск в ширину. Это алгоритм поиска, при котором просматривается каждый уровень в направлении сверху вниз (рис. 7.2).

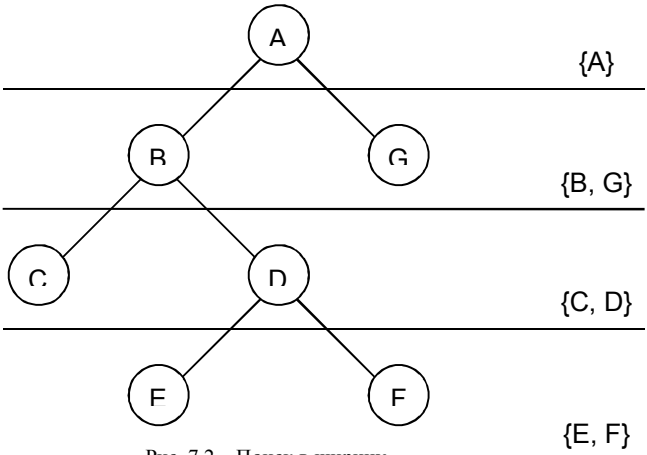


Рис. 7.2 – Поиск в ширину

При таком алгоритме параллельного поиска быстро находятся кратчайшие ветви дерева. В общем виде алгоритм имеет структуру:

```
BREADTHFIRST: procedure(solution);
 declare 1 TREE,
         2 ARGUMENT, /* данные */
         2 SONS(N);  /* произвольные деревья */
 declare (solution, NextStep) set of TREE;
 /* переход вниз для каждого уровня */
 do while (solution ≠ 0);
  NEXTSTEP = {TREE.SONS(I) для всех I и TREE
    в solution};
  call BREADTHFIRST(NextStep);
 end;
end BREADTHFIRST;
```

Поиск в ширину очень популярен при решении задачи о лабиринте.

Поиск с возвратом. Для многих алгоритмов часто требуется перебор возможных вариантов решения задачи. Один из таких способов перебора называется поиском с возвратом.

Алгоритм:

```
Вызвать первую часть;
do while (не окончится);
 вызвать следующую часть;
 do while (нет возможности продолжить);
  Вернуться на предыдущий уровень;
```

```
    Проверить возможность альтернативного решения
      для этого уровня;
  end;
end;
```

Алгоритм поиска в глубину является примером поиска с возвратом:

Начать с корневого узла;

```
do while (существуют не просмотренные узлы);
  Перейти к узлу «левый сын», если возможно;
  Иначе перейти к узлу «правый брат»;
  do while (нет узлов «левый сын» и «правый брат);
    Перейти к узлу «отец»;
    Перейти к узлу «правый брат», если можно;
  end;
end;
```

7.4.3. Стратегия распределения памяти

Одним из типов поиска являются задачи по распределению памяти (наиболее характерны они при создании ОС), однако во многих прикладных алгоритмах эти задачи возникают при создании прикладных программ, работающих с большим объемом данных. Если до начала обработки полный объем данных, которые подлежат обработке, неизвестен, то используется общий подход для распределения областей этой памяти согласно заданным требованиям.

Дана фиксированная область памяти, задача заключается в том, чтобы занимать и освобождать меньшие области памяти большой областью, без выхода за границы памяти. При распределении памяти проверяются наибольшие свободные области. Обычно эти области малы для размещения в них всех данных, и, следовательно, память становится фрагментной. Имеются следующие способы уменьшения фрагментности.

Первое возможное размещение. Последовательно просматриваются области памяти, пока не найдется первая, достаточная для размещения. Вся область организована в список и упорядочена по адресам (рис. 7.3).

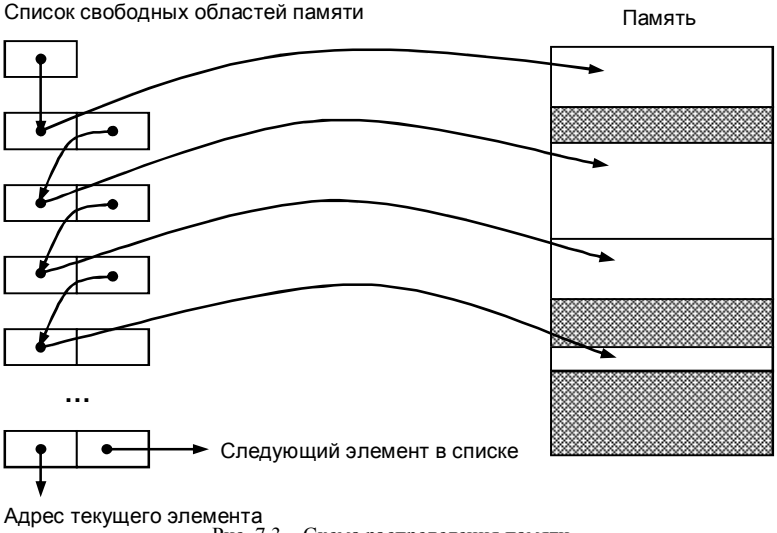


Рис. 7.3 – Схема распределения памяти

Поиск в этом списке осуществляется последовательно до тех пор, пока не будет найдена достаточно большая область для размещения данных. Тогда эта область изымается из списка, а на это место помещается меньшая свободная область – оставшаяся незанятая часть исключенной области. С помощью этого метода легко освобождать ранее занятые области памяти. С этой целью просматривается список, как только он будет упорядочен по адресам, то нетрудно найти место, чтобы пометить освобожденную область памяти. Соседние адреса – смежные в списке, поэтому легко объединить две свободные области в одну, большую по размеру.

Данный способ самый простой, однако, поскольку занятие области памяти осуществляется после нахождения первой области, достаточной для размещения данных, память становится фрагментной.

Наилучшее размещение. Последовательно просматриваются все области, выбирается наименьшая область, размер которой больше или равен требуемому объему для размещения данных. В этом случае области связываются друг с другом по размеру, а не по адресам. Алгоритм размещения – освобождения аналогичен первому возможному размещению. Однако в этом случае стратегия размещения приводит к меньшей фрагментности, потому что используется область наименьшего размера для размещения данных. Однако, так как свободная об-

ласть не упорядочена по адресам, алгоритм объединения двух свободных областей в область большего объема довольно сложен.

Сопрягаемые области памяти. Областями памяти являются N цепочек размером 2^N каждая. Если область размером 2^K отсутствует, а имеется свободная область размером 2^{K+1}, то она разбивается на две сопрягаемые области размером 2^K. После того, как области освободятся, они рассматриваются как области размером 2^K, которые можно объединить в область размером 2^{K+1}.

7.5. Сортировка

Сортировка – это процесс размещения элементов в списке в некоторой числовой или лексикографической последовательности (порядке). Различаются следующие виды сортировки.

Обменная сортировка. При этом способе наименьший элемент выдвигается в начало списка, следующий по величине – на вторую позицию и т.д. Для списка N элементов производится $N(N-1)/2$, т.е. $O(N^2)$ операций. Естественно, что для больших значений N использование этой сортировки нецелесообразно.

Алгоритм:

```
EXCHANGE_SORT procedure(LIST,N);
  declare LIST(N); /* массив сортировки */
  do I=1 to N-1;
    do J=I+1 to N;
      if LIST(J)< LIST(I) then
        Reverse(LIST(J), LIST(I));
    end;
  end;
end EXCHANGE_SORT;
```

Сортировка слиянием. Данный способ сортировки схож с методом разбиения на подзадачи. Вначале неотсортированный список делится на две части, затем каждая часть сортируется независимо. После этого списки объединяются.

Алгоритм:

```
MERGE_SORT procedure(LIST,N);
  declare LIST(N); /* массив сортировки */
  call MERGE_SORT(LIST(1:N/2),N/2,
    (LIST(N/2+1:N),N/2));
  /* объединение отсортированных списков */
end MERGE_SORT;
```

Процедура слияния отсортированных списков размером K выполняется за $2\times K$ операций путем просмотра первого элемента в каждом списке и перенесения меньшего из двух в новый список. Эти два действия повторяются для каждого из $2\times K$ элементов списка, получается $4\times K$ операций. Так как каждый из списков уже отсортирован, нужно проверить только первые элементы. Хотя для реализации этого способа нужна память большого объема, время сортировки путем последовательного разбиения каждого из подсписков на два можно уменьшить до величины

$$N \log_2 N.$$

7.6. Алгоритм выбора из конечного состояния

Часто задача может быть сведена к множеству действий, зависящих от текущего состояния программы. Такой способ решения задачи называется выбором из конечного состояния и обычно включает таблицу, описывающую выполняемые действия. Строки таблицы определяют состояние программы, а столбцы – возможные действия. Элементы таблицы описывают выполнение возможных действий, в частности, имя вызываемой подпрограммы для обработки действий.

Переменными в этом случае являются текущее состояние программы и допустимое воздействие, определяемое внешней средой. Пересечение этих значений в матрице решений определяет ответное действие и новое состояние программы. Такие состояния (алгоритмы) наиболее характерны для лексического анализатора трансляторов. Решение подобных задач строится по типу модели конечного автомата.

Таблица состояний:

	A	B	C	…	I	J	K	…	Y	Z
1										
2.										
…										
N–1										
N										
N+1										
…										
M										

7.7. Сопрограммы

В некоторых системах две или более задач должны обрабатываться посегментно, причем каждый сегмент выполняется с различной скоростью (рис. 7.4).

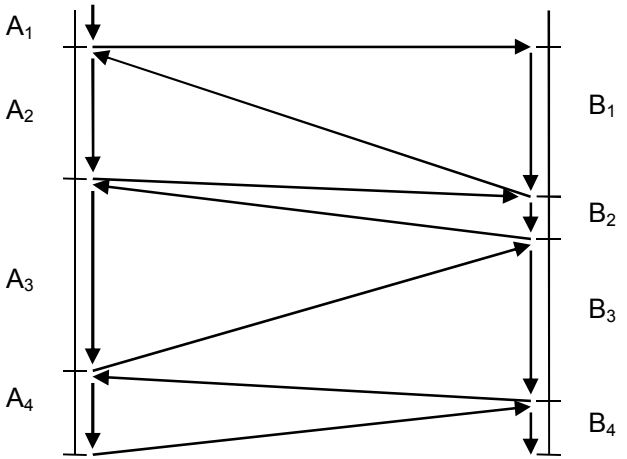


Рис. 7.4 – Организация сопрограмм

Использование сопрограмм может быть полезной управляющей структурой.

Сопрограмма – это такой вид программы, который сохраняет текущее состояние счетчика команд. Когда программа вызывается повторно, выполнение продолжается с адреса, записанного в счетчике программ, а не с начала программы.

Сопрограмма, возвращающая управление в процедуру X, определяется как **resume** X. Использование такого оператора языка PDL можно представить следующим образом:

```
call DATA;
goto Next_Statement;
/* Для элементов данных */
DATA: procedure;
  declare FS, DV;
  DV = значение данных 1;
  resume FORMAT(FS);
  call I/O – process(DV, FS);
  DV = значение данных 2;
  resume FORMAT(FS);
  call I/O – process(DV, FS);
```

```
  ...
  DV = значение данных n;
  call I/O – process(DV, FS);
end DATA;
FORMAT: procedure(FS);
  LOOP: FS = спецификация формата 1;
  resume DATA(FS);
  FS = спецификация формата 2;
  resume DATA(FS);
  ...
  FS = спецификация формата n;
  goto LOOP;
end FORMAT;
Next_Statement:
...
```

Здесь каждая сопрограмма вызывает другую сопрограмму. Таким образом, программа обработки данных вызывает программу обработки формата до следующего элемента формата.

К сожалению, сопрограммы отсутствуют в широко распространенных языках программирования.

Контрольные вопросы

1. Стандартные методы проектирования.
2. Разбиение задачи на независимые подзадачи.
3. Разбиение задачи на одинаковые по сложности части.
4. Рекурсия.
5. Динамическое программирование.
6. Моделирование.
7. Поиск. Поиск в списках. Прямой поиск. Линейный поиск.
8. Поиск с возвратом.
9. Стратегия распределения памяти.
10. Алгоритм выбора из конечного состояния.
11. Сопрограммы.

8. МЕТОДЫ УПРАВЛЕНИЯ ПРОЕКТИРОВАНИЕМ ПРОГРАММНЫХ ИЗДЕЛИЙ

Основная цель управления – организовать и связать взаимодействие исполнителей при создании программного продукта.

8.1. Организация управления проектированием программного изделия

8.1.1. Понятие изделия как средства общения

Действия при создании программного изделия по многим параметрам совпадают с действиями при создании технического изделия.

Изделие	Программа
1. Изучение рынка	1. Изучение рынка
2. Планирование	2. Планирование
3. Анализ окупаемости	3. Рассмотрение соглашения о требованиях
4. Разработка	4. Разработка
5. Конфигурационное управление	5. Конфигурационное управление
6. Обеспечение гарантии качества	6.Тестирование (автономное)
7. Изготовление	7. Копирование
8. Контроль качества	8. Тестирование (системное)
9. Отправка потребителю	9. Передача пользователю
10. Ввод в действие	10. Ввод в действие
11. Гарантийный ремонт	11. Сопровождение
12. Усовершенствование	12. Расширение функциональных возможностей

Т.е. самые общие действия при создании изделия имеют аналоги при создании программного продукта, таким образом, можно использовать опыт по управлению изготовлением.

8.1.2. Нисходящий анализ процесса управления проектированием программного изделия

Управление проектированием программного изделия включает в себя следующие функции:

- планирование;
- разработка;
- обслуживание;
- выпуск документации;

– испытания;
– поддержка;
– сопровождение;

Иерархическая декомпозиция управления разработкой программного изделия может быть представлена следующим образом (рис. 8.1).

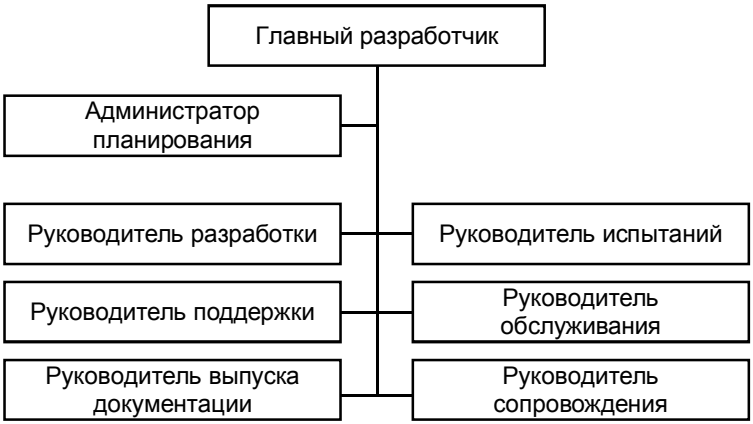


Рис. 8.1 – Декомпозиция управления

Такая идеализированная организация требует полную обособленность процессов, связанных с проектированием, от других видов деятельности и изолированности всех функций друг от друга. Естественно, что на практике это не реализуемо.

Каждая организация должна иметь администратора (директора), именно он несет ответственность за успех и неудачу разработки. В структуре также должно существовать то или иное важное направление деятельности, включая функцию разработки. Лицо, которое руководит этим направлением, считается ответственным за все аспекты создания изделия, выпускаемого организацией. Чтобы координировать процесс разработки, это лицо имеет право назначать администраторов изделия и руководителей проектов и обеспечивать их взаимодействие.

8.1.3. Организация взаимодействия

Если все взаимодействия хорошо определены, то и управление ими организовано должным образом. Если же взаимодействия плохо организованы, то даже при жесткой линейной структуре подчинения трудно будет создать конечное изделие.

Важнейшим принципом любого вида управления является разделение целого на части, и многие методы и средства основываются именно на этом принципе.

Совокупность точек, в которых две функциональные группы взаимодействуют друг с другом, называется *организационной границей*. Иногда функция может иметь один канал взаимодействия со всеми остальными функциями, однако более вероятно, что она имеет несколько границ соприкосновения. Границы функции определяются множеством зафиксированных и незафиксированных планов, стратегий, процедур, которые определяют функциональные обязанности. Чем больше сведений фиксируется в письменном виде, тем лучше, т.к. это уменьшает двусмысленность. Основное свойство организационной границы состоит в разграничении ответственности (кто и что делает, каким образом, для чего и т.д.). Неполное определение, двусмысленность и сложность приводят к невозможности описания, а, следовательно, и понимания природы взаимодействия.

Но ни сами документы, ни их коллективное обсуждение не могут обеспечить действенных взаимосвязей, если отсутствуют контакты функциональных групп.

Общие организационные обязанности могут устанавливаться с помощью должностных инструкций и целевых планов подразделений. Конкретные обязанности определяются планами выпуска изделия. Пропорциональное распределение ответственности обеспечивается соответствующими стратегиями управления. В них обязательно должны предусматриваться возможности невыполнения взятых обязательств и учитываться поправки на исключительные случаи, чтобы сделать планы и процедуры жизнеспособными.

8.1.4. Установление целей, средства их достижения

Первым шагом процесса установления и достижения целей является подбор необходимого персонала. Когда в организации происходят изменения, соответственно меняется и ее персонал. Некоторые изменения происходят периодически. Подобные изменения характеризуются экспоненциальным ростом числа устанавливаемых связей. Чтобы приспособиться к этим колебаниям, нужны руководители, способные к адаптации. Руководителей, продуктивных только в какомлибо одном виде деятельности, нужно заменять. Программирование – область деятельности, требующая высокой квалификации, оно привлекает к себе неординарных, эксцентричных людей. Такие люди редко понимают структуру организации, ориентированную на создание программного продукта, часто отказываются работать в условиях ограни-

чения свободы творчества, т.к. они вынуждены тратить время на документирование или защиту своих разработок. Однако их участие в «черновой работе» необходимо, и чтобы склонить этих людей к работе, необходимо комплектовать штат, руководствуясь соображениями эффективности. Это означает отход от идеальных установленных общих правил, предоставление свободного режима прихода и ухода с работы, выделение таким людям помощников, способных компенсировать их неумение четко документировать свои результаты. Но при этом следует сравнивать прямые затраты, связанные со стимулированием «привилегированных» сотрудников, а также неявные издержки, связанные с ухудшением морального состояния их сослуживцев, с получаемой организацией выгодой.

Основным методическим принципом управления разработкой является *целевое управление.*

Целевое управление представляет собой концепцию планирования и управления, с помощью которой руководитель устанавливает соответствующие цели через своего непосредственного руководителя более высокого ранга и участвует в установлении целей последнего; при этом результаты его деятельности оцениваются на основании конкретного обсуждения и документального рецензирования.

Цели присутствуют в планах самого различного уровня: целевых планах, бюджете, планах выпуска изделий, документации и т.д. Основной план для программного изделия – соглашение о требованиях. Цели, сформулированные в этом плане, должны включаться в индивидуальные рабочие планы, служащие тем механизмом, посредством которого в системе целевого управления достигается договоренность между исполнителями и их руководителями.

Достижение цели гарантируется включением общих целей создания изделия в индивидуальные рабочие планы и организацией текущего контроля за их выполнением.

Соглашение о требованиях, план поддержки, распределение бюджета и другие средства устанавливают те границы, в пределах которых не требуется использование обратной связи.

Существует три вида основных критериев оценки эффективности той или иной деятельности:

конкретные свойства;

затрачиваемое время;

стоимость.

Каждый из этих критериев имеет определенные границы действия, по достижению которых обязательно предоставление отчета руководству о результатах.

Однако следует учитывать, что управление созданием программных изделий является примером управления в условиях неопределенности. Качество такого управления зависит от способностей руководителей предвидеть трудности, планировать разработку с учетом случайных факторов и уметь защищать такого рода планирование от критики начальства, которое требует непременно «исключить случайность».

8.1.5. Подбор и обучение кадров

Поскольку разработка программных средств является достаточно сложной процедурой, то для ее реализации нужны специалисты высокой квалификации, т.е. для проектирования программного обеспечения необходимо выделять высококвалифицированный персонал на все участки работы. Следует искать таких людей, которые выполняли подобные функции достаточно хорошо, либо тех, кто выполнял очень грамотно функции чуть меньшей сложности.

Руководство проектами надо поручать лишь тем специалистам, которые обладают подобным опытом. Здесь очень хорошо работает концепция бригады главного программиста.

Но ведь люди где-то должны начинать работать? Так как в настоящее время подготовка в университете достаточно основательная (структурное, объектно-ориентированное программирование и др.), то нишу *проблемных* программистов могут заполнить выпускники университетов.

С наибольшей вероятностью квалифицированный персонал можно найти среди поставщиков универсального программного обеспечения. Основная черта, которой должен обладать кандидат, – способность подчиняться дисциплине. Он должен понимать важность *нисходящего подхода* в создании программного изделия и документирования программы до ее кодировки.

Надо обратить внимание на то, что результаты работы любого программиста должны быть понятны другим, и ими можно воспользоваться. Любя попытка выполнить нужную работу, с помощью специалистов не самого высокого класса, обречена на провал.

Важным фактором для успешного выполнения работ является обеспечение продвижения по службе. Целесообразно максимально использовать имеющиеся должностные инстанции. Составить формальные должностные инструкции, определяющие соответствующее повышение ответственности и обязательно устанавливающие одинаковые уровни квалификации для должностей одного уровня.

Кроме вертикального продвижения по служебной лестнице, следует обеспечить продвижение по горизонтальному уровню. Должна поощряться широта профессиональных интересов и упрощаться проблема подбора кадров для выполнения менее важных функций путем включения в должностные инструкции более высоких уровней описания тех функций, выполнения которых достаточно для занятия более низких должностей в тех или иных подразделениях.

Принимая на работу людей любого уровня квалификации, необходимо обеспечить им возможность продвижения по службе в результате дополнительного профессионального обучения. По каждой выполняемой функции должен быть подготовлен семинар, рассчитанный на слушателей с различным уровнем квалификации. Работу семинара следует организовать так, чтобы на нем рассматривались наиболее значимые и уникальные средства проектирования. Для стимулирования заинтересованности в этих семинарах можно сделать участие в них обязательным условием для получения конкретного назначения.

Не менее важно обучение вне рамок организации. Необходимо максимально использовать участие сотрудников проекта в семинарах по профилю, проводимых передовыми школами программирования. Естественно, что такое обучение дело дорогостоящее, поэтому необходимо иметь твердую уверенность, что затраты на обучение дадут должный эффект.

8.2. Организация планирования разработок программного изделия

Планы создания программного изделия должны охватывать этапы разработки, документирования, испытаний, обучения пользователей, сопровождения. Отсутствие планов – основная причина переделки программ. Естественно, что в планах невозможно предусмотреть все категории пользователей, однако можно принять разумные меры предосторожности на непредвиденные случаи.

Программное изделие – это собственно программа плюс документация, гарантия качества, рекламные материалы, обучение, распространение и сопровождение. Таким образом, для создания программного изделия требуется совокупность планов, охватывающих все аспекты разработки изделия и их связь с внешней средой.

8.2.1. Виды планов

Самым главным охватывающим планом является целевая программа. Цель ее – максимизировать доход от капиталовложений. Целевая программа отвечает на вопрос, для чего предназначена та или иная деятельность. В целевой программе не сообщается, как цель

должна быть достигнута, и редко устанавливаются сроки реализации. Обычно целевые программы редко имеют самостоятельную значимость, они предназначены для других программ, чтобы ответить на вопрос *зачем?*

Рис. 8.2 – Иерархия планов

Следующий уровень планирования – стратегический план. Он отвечает на вопросы *что?* и *когда?* При этом, чем более конкретна цель, тем более своеобразны будут стратегии ее достижения.

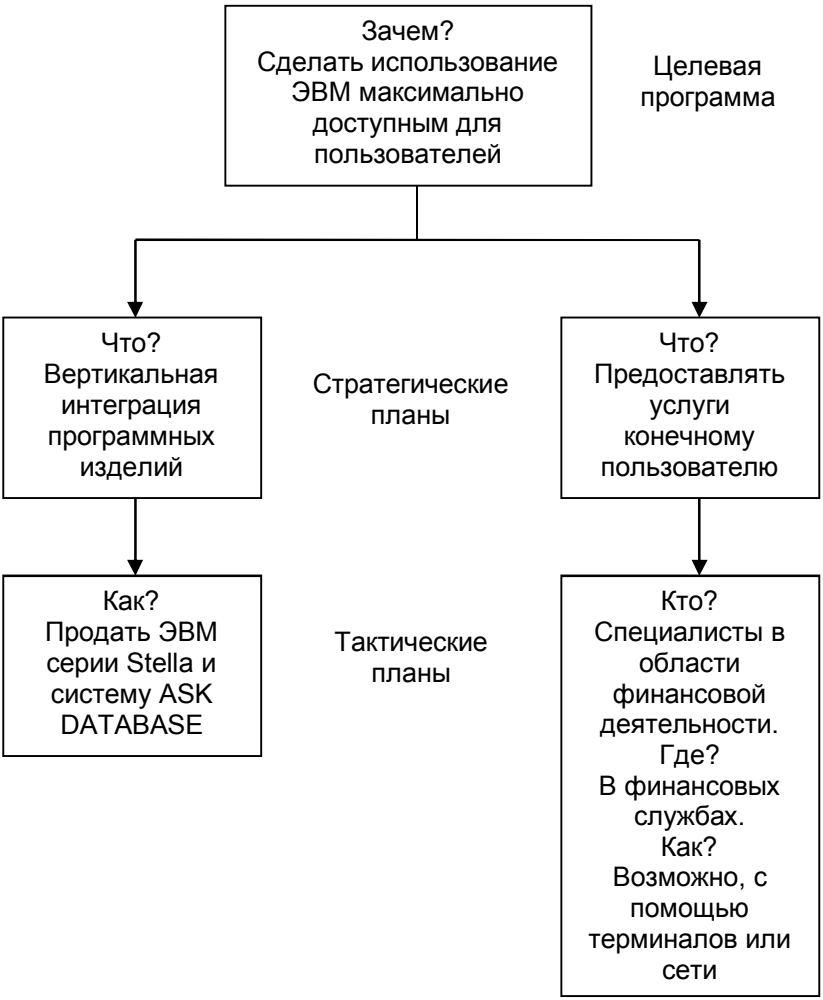


Рис. 8.3 – Отработка стратегий

Более конкретными являются тактические планы. Тактические планы отвечают на вопросы *кто?* и *где?*, т.е. конкретизируют способы достижения цели.

Стратегический план формулирует, что необходимо сделать для достижения цели и когда. Тактический план показывает, кто будет выполнять работу, как и когда (рис. 8.2).

Пример.

Цель фирмы – сделать использование ЭВМ доступным наибольшему числу людей. Одна из стратегий состоит в том, чтобы осуществить вертикальную интеграцию программных изделий. Другая стратегия заключается в том, чтобы предлагать услуги непосредственно пользователю (экономистам, физикам и т.д.). Третья стратегия состоит в том, чтобы самим использовать разработанный программный продукт для анализа разнообразных данных (рынка сбыта, экономической ситуации и т.д.)

Каждое программное изделие можно рассматривать как тактическое решение рассматриваемых стратегий (рис. 8.3).

Обычно целевая программа, стратегический и тактический план редко существуют как самостоятельные элементы. Однако, в тех случаях, когда формируется совокупность планов, необходимо классифицировать каждый из них в соответствии с тремя указанными категориями, чтобы облегчить понимание задач.

Рассмотрим несколько конкретных планов.

Бюджет – это такой план, который, вообще говоря, не устанавливает, «зачем», «как» и «что» делать. В нем устанавливаются важные количественные ограничения, связанные с вопросом «что». Это самый распространенный план. Благодаря бюджету обратное воздействие реальной окружающей обстановки оказывается довольно быстрым и точным.

Другим планом специального вида является календарный план. Время, как и деньги, может быть общим фактором оценки самых разнообразных видов деятельности.

Индивидуальные планы работ – это инструмент целевого управления, обеспечивающий фиксацию конкретных выработанных целей и сроков окончания работ, контроль за их завершением или прекращением работ. Т.е. он аккумулирует в себе целевой, стратегический и тактический планы на основе бюджета. Это действенное средство «одушевления» разработки программного изделия и выявления в ней роли конкретных участников.

Сетевой график является преимущественно тактическим планом. Он определяет в терминах заданий (работ) – как можно достигнуть цели и кто отвечает за выполнение конкретных работ. Это также и стратегический план, в котором устанавливаются даты окончания работ.

8.2.2. Декомпозиция планов

При планировании (создании планов) используется схема нисходящего планирования. Эта процедура называется декомпозицией планов:

– план создания семейства программных изделий;
– план создания серии;
– план создания совокупности изделий;
– план производства конкретного изделия.

Законченность плана означает, что рассмотрены все необходимые вопросы и, в то же время, обеспечен необходимый уровень детализации. Все имеющие отношение к делу вопросы должны обсуждаться на самом высоком уровне детализации и на самом низком уровне иерархии. Установить нужный уровень детализации и одновременно охватить все вопросы трудно. Самая серьезная ошибка – это упущение существующих аспектов. Вторая крупная ошибка – это излишняя детализация, которая приводит к «увязанию в трясине». Подобных ошибок можно избежать, если следовать заранее составленному вопроснику или определенной форме. Ошибок излишней детализации можно избежать, если каждый раз задавать вопрос «есть ли польза от дальнейшей детализации?».

В процессе декомпозиции планов в качестве основного принципа должен служить следующий принцип: «Какая свобода выбора желательна на следующем шаге планирования?». Обычно последующие этапы планирования выполняются на все более низких уровнях, и достаточно учитывать лишь область их полномочий.

Структура всех реальных иерархических планов обязательно должна совпадать. Важно лишь то, чтобы степень детализации каждого пункта плана увеличивалась по мере иерархического спуска вниз. Однако, слишком большое время, потраченное на обсуждение несуществующих деталей, делает планы нереальными.

Второй принцип декомпозиции – определить ограничения для следующего, более низкого, уровня «Сделай транслятор с языка за один год, потратив не более 300.000$». Здесь имеется набор трех ограничений:

- транслятор должен обрабатывать инструкции конкретного языка;
- для создания отведено не более года;
- бюджет – не более 300.000$.

Никаких других ресурсных ограничений нет. В этих рамках есть свобода выбора. Если продолжить декомпозицию плана, то возникнут все большие и большие ограничения, так что на самом низком уровне будет обеспечена их абсолютная детализация. Но при этом, если пол-

номочия распределены правильно, ни одно из первоначальных ограничений не будет нарушено, т.е. должна существовать обратная связь, в случае если потребуется изменение плана за пределами своих полномочий.

8.2.3. Организационная структура группы планирования

Естественно, что различные виды планов соответствуют различным уровням руководства. В любой разрабатывающей организации должно быть создано специальное подразделение, единственной обязанностью которого является планирование и управление организацией в целом. Оно необходимо потому, что кто-то должен гарантировать существование планов для всех подразделений, обеспечивать их совместимость и взаимную увязку. Для этого необходимо глубокое понимание структуры организации и ее деятельности. Люди, выполняющие эти функции, не должны одновременно иметь никаких других обязанностей. Планы подобны законам. Они должны выполняться неукоснительно. Т.е. люди, осуществляющие планирование и контроль, должны обладать соответствующими полномочиями. Управление планом должно быть организовано так же четко, как и само планирование.

Должностные лица, руководящие выполнением планов, называются администраторами планирования. Однако надо помнить, что слишком большое число людей в группах планирования порождает бюрократию, поэтому группа планирования должна быть организована по принципу предметно-производственной специализации. В должностных инструкциях должна быть четко определена одна-единственная точка соприкосновения, где каждая функциональная группа взаимодействует с группой планирования (рис. 8.4).

В большинстве организаций роль администратора планирования выполняет руководитель целевой программы.

Планирование должно опираться на гарантии правильного применения принципов конфигурационного управления. Для этой цели вводятся функции контроля документации, включая процедуры хранения и распространения проектной документации.

Рис. 8.4 – Схема взаимодействия группы планирования и функциональных подразделений

Вместе с тем, планирование невозможно только по иерархии сверху. Поэтому каждая функциональная группа должна иметь собственный персонал тактического планирования. Их взаимодействие должно строиться по схеме:

– подразделение планирования отвечает за целевое и стратегическое планирование, управление бюджетом и планами для всей организации;
– функциональная группа отвечает за тактическое планирование.

8.2.4. Планы, связанные с созданием программных изделий

При использовании нисходящего планирования осуществляется постепенный переход от общего к частному (от целевой программы к стратегическим и тактическим планам) и от глобального к локальному (от плана семейства изделий, через план выпуска серии и совокупности изделий, к проекту конкретного программного изделия). Для облегчения понимания структуры различных планов вводятся «стандартные» определения.

Программное изделие – совокупность отдельных программных средств, их документации, гарантии качества (ISO 9000), рекламных материалов, мер по обучению пользователей, распространению и сопровождению программного обеспечения. Независимо от того, является ли программное обеспечение целостным изделием или только частной его модификацией, обычно изделие представляет собой тот наименьший объект, относительно которого рассматриваются все вышеназванные элементы.

Совокупность изделий – это группа изделий, имеющих одну или несколько общих характеристик и работающих совместно в некоторой комбинации (ОС, компиляторы, сервисные программы, средства диагностики, которые управляются операционной системой, составляют такую группу).

Серия изделий – это сочетание аппаратных и программных средств, которые имеют одну и более общих связей и функционируют совместно в некоторой комбинации как самостоятельная система.

Семейство изделий – несколько связанных программных изделий, которые необязательно должны иметь какой-либо общий интерфейс и работать на одной и той же аппаратуре (например, все компиляторы языка FORTRAN, созданные некоторым поставщиком для различных ЭВМ).

Планом самого высокого уровня является план выпуска семейства или серии изделий. Так как в их названии присутствует слово изделие – это стратегический план. В целом он формулируется в терминах технических средств, программного обеспечения, задач обучения персонала и т.д. Он содержит элементы стратегии:
– как обеспечить совместимость с конкурирующими изделиями, благоприятствующую проникновению на рынок?

– периодичность усовершенствования в целях продления цикла жизни изделия и т.д.

Обычно элементы стратегии охватывают длительный интервал (5-10 лет).

Как только план серии одобрен, разрабатываются планы выпуска совокупности изделий. Лучшим средством представления таких планов является конфигуратор. Это понятие используется для определения таблицы, в которой кратко характеризуются взаимосвязи между операционными системами и подчиненными им элементами совокупности изделий (табл. 8.1).

Таблица 8.1 – Конфигуратор

	Совокупность программных изделий						
	VSOS2			VSOS3		VSOS4	
Наименование программного изделия	Страниц	Состояние	Уровень поддержки	Состояние	Уровень поддержки	Состояние	Уровень поддержки
VSOS2 VSOS3 VSOS4	4 4 5	S27 7 /// ///	3	/// s27 7 ///	2	/// /// 7.98	1
/// – изделие не будет доступно для использования. **дата** – месяц и год, когда изделие станет доступно для использования.	1 – поддержка через уведомление о выявленных дефектах, посылается сообщение об изменениях, рассматриваются заявки на расширение возможностей; 2 – поддержка через уведомление о выявленных дефектах, посылается сообщение об изменениях, заявки на расширение возможностей не принимаются; 3 – только обработка уведомлений о дефектах.						

Компактная форма конфигуратора позволяет передавать большое количество информации верхним уровням управления.

Когда наступит время исследовательской деятельности по разработке программного изделия, сначала создается приблизительный план работы, производится первоначальное выделение средств, фиксируются фонды, необходимые для завершения работы. Документ,

создаваемый при этом, есть не что иное, как распределение бюджета. Этот документ реализует концепцию приростного финансирования, он обеспечивает контроль за выполнением планов.

Окончательный план выпуска изделий вырабатывается после серьезного изучения и обсуждения проблемы.

Первая задача решается после распределения бюджета, формируется план, который называется соглашением о требованиях. В этом документе устанавливаются договорные отношения между разработчиком и пользователем, а также тем, кто должен заниматься продажей изделия. Помимо разработчика, который обычно готовит этот документ, в его составлении участвуют многие функциональные подразделения, в том числе и группа испытаний, группа выпуска документации, персонал, занятый внедрением, сопровождением, сбытом и др.

Рассмотрение, утверждение и корректировка соглашения о требованиях является наиболее важным моментом во всем процессе планирования разработки. Именно благодаря соглашению о требованиях все заинтересованные стороны знают, какие ожидания можно связать с созданием изделия.

Как только разработка становится реальностью, формируются планы отдельных групп: группы испытаний, план выпуска документации, план обеспечения поддержки.

8.2.5. Опытный образец изделия

Там, где существует хотя бы небольшая вероятность неудачи, план должен предусматривать возможность создания экспериментального изделия.

Опытный образец – это почти законченное изделие. Он включает в себя все компоненты завершенного изделия, в том числе и черновой вариант технической документации, и предполагает проведение небольшого объема независимых испытаний с целью обеспечения необходимых условий для выработки критических замечаний по доработке изделия и гарантий минимального риска пользователя.

При создании опытного образца необходимо следовать той же методике управления, которая применяется для разработки основного изделия.

Для принятия решения о том, какую документацию и какой контроль следует предусмотреть для опытного образца, обычно используют правило: *считать опытный образец первой версией реального изделия и тут же запланировать реализацию второй версии*.

Если программное изделие предназначено для многих пользователей, целесообразно выбрать одного или двух из них для поставки им

бесплатно и с полной поддержкой опытного образца. При этом пользователи настолько будут польщены вашим выбором, что обеспечат отдачу, значительно превышающую стоимость опытного образца.

8.2.6. Организация планирования в фазе исследования

Фаза исследований начинается тогда, когда подтверждается необходимость создания программного изделия, и заканчивается тогда, когда утверждены технические требования (рис. 8.5).

Деятельность группы планирования наиболее активна в фазе до начала анализа осуществимости, как только подтверждается необходимость изделия.

В процессе декомпозиции планов в некоторый момент времени поднимается вопрос о конкретном программном изделии. Обычно такое предложение содержится в стратегическом плане в разделах, касающихся торговых интересов, и плане создания семейства или серии изделий.

Как только признана необходимость изделия, сразу начинается анализ осуществимости такого предложения. Группа планирования анализирует возможность компенсации затрат, которые могут понадобиться для его осуществления, проводит финансовый анализ и распределение ресурсов.

Вторым важным событием фазы анализа осуществимости является выделение ресурсов для проведения исследований осуществимости проекта. Здесь используется принцип приростного финансирования, на основании которого могут быть субсидированы аналогичные исследования осуществимости проектов множества изделий. Причем лишь наиболее перспективные из них должны продолжаться финансироваться после завершения фазы анализа осуществимости. На этой стадии важно выяснить, укладывается ли разработка проекта в установленные сроки, т.е. максимальное внимание уделить трудоемкости работ.

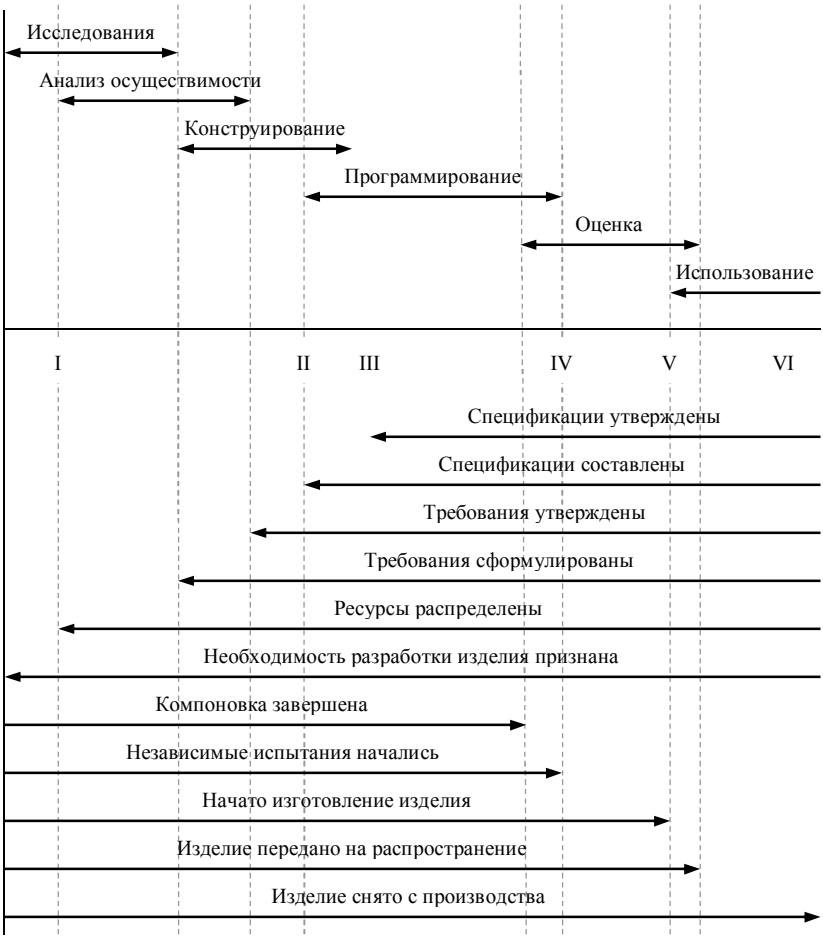


Рис. 8.5 – Жизненный цикл программного изделия

Результатом анализа осуществимости является отчет, дающий четкие рекомендации по реализуемости проекта и аргументированные предложения по прекращению работ. Если результаты анализа осуществимости проекта показывают, что изделие может быть создано, указанный отчет приобретает вид плана разработки (или соглашения о требованиях).

Анализ осуществимости выполняется той группой, которая в случае положительных результатов анализа будет нести ответственность за разработку изделия. Отчет (соглашение о требованиях) всегда составляется под руководством группы планирования. Это приводит к самому строгому из возможных подходов к выработке соглашения о требованиях т.к. разработчик в этом случае отвечает за выполнение своих собственных планов.

Группа планирования в этом случае считается ответственной за обеспечение соответствия соглашения о требованиях тактическим и стратегическим планам и целевой программе организации в целом. Подобная координация является основой успешного выполнения планов.

8.2.7. Организация планирования в стадии анализа осуществимости

В момент времени, когда ресурсы распределены (начало фазы осуществимости), но соглашения о требованиях еще нет (конец фазы осуществимости), новое изделие рассматривается с учетом будущих условий его использования. Для этого существует два средства: конфигуратор и план выпуска.

В конфигураторе перечисляются программные изделия, которые должны функционировать совместно с проектируемым изделием, т.е. во-первых, необходимые для его ввода в действие или эксплуатацию, во-вторых, зависящие от него в период их установки или эксплуатации, в-третьих, работающие параллельно с ним. Конфигуратор – это план создания интерфейсов высокого уровня, определяющих связи и способствующих взаимодействию функциональных групп, совместно разрабатывающих некоторое изделие.

План выпуска изделия имеет то же значение, что и конфигуратор, но охватывает только те элементы, которые должны быть готовы к определенному времени.

Конфигуратор и план выпуска необходимо периодически обновлять. Последней задачей, выполняемой группой планирования в фазе осуществимости, является рассмотрение и утверждение соглашения о требованиях к программному изделию. Это самый важный момент в планировании, т.к. он устанавливает направление разработки и инициирует начало разработки.

8.2.8. Организация планирования в фазах конструирования и кодирования

По окончанию фазы исследований группа планирования рассматривает и утверждает планы *организации поддержки* для каждого изделия или совокупности изделий. В течение всей фазы конструирования (проектирования) группа выпуска документации и группа испытаний готовят планы издания документов и планы испытаний. Группа планирования анализирует эти планы главным образом на их соответствие предписываемым формам и на их совместимость с соглашением о требованиях, конфигуратором и планом выпуска изделия. В течение фазы кодирования группа поддержки готовит свой план, а группа планирования рассматривает его в том же порядке, в котором до этого рассматривался план изданий и план испытаний.

8.2.9. Организация планирования в фазах оценки и использования

Следующий период жизненного цикла программного изделия связан с принятием решения о целесообразности широкого распространения изделия. Рекомендации дает группа испытаний, а если решение о выпуске принято без ее согласия, то группа испытаний дает заключение о степени готовности изделия. При этом группа планирования анализирует отчет группы испытаний и вновь выясняет у нее степень соблюдения соглашений о требованиях. Для принятия решения о распространении изделия всегда требуется согласие группы планирования, потому что после выпуска изделия очень трудно исправить ошибки не только в программах, но и в документации. Поэтому администратор планирования должен быть уверен в том, что все группы выполнили свои задачи и готовы к выполнению фазы использования.

Обычно группа поддержки настаивает на возможно более раннем выпуске изделия, чтобы своевременно закончить операции по сбыту или сохранить того или иного заказчика. Группа разработки выдвигает то же самое требование из практических соображений. Группа же испытаний обычно хочет продлить испытания. В этих условиях компромиссное решение принимает лишь группа планирования, как группа, отвечающая за комплексную увязку всех вопросов.

Группа планирования осуществляет текущий контроль за изделием в фазе использования, непрерывно наблюдая за уведомлениями о дефектах и запросами на расширение. Именно администратор группы планирования отвечает за организацию устранения замеченных дефектов и целесообразность создания новых версий с расширенными возможностями. При этом его основная задача – постоянно снижать уровень поддержки и сопровождения. Он также принимает решение о

снятии изделия с производства и обслуживания. Рекомендации о снятии изделия с производства и обслуживания могут поступить из любой функциональной группы. Однако, независимо от первоисточника рекомендации, последнее слово принадлежит группе планирования. Никакая другая группа не имеет такого круга обязанностей, который необходим для ответственного принятия подобного решения.

8.2.10. Обязанности группы планирования при рассмотрении и утверждении планов разработки программного изделия

Разделение жизненного цикла программного изделия на фазы обеспечивает несколько контрольных точек, в которых изделие оценивается (см. рис. 8.5). Сознательные решения, принимаемые в конце каждой фазы с целью заблаговременного определения судьбы изделия в будущем, называются фазовым планированием. Эта процедура выполняется следующим образом.

Сначала определяются фазы и основные события в конце каждой из них. Затем проводится формальный обзор на основе, по крайней мере, одного документа для каждого события.

Таблица 8.2 – Документы обзоров

Фаза	Обзор основных событий	Рассматриваемые вопросы
		1. Распределение бюджета 2. Извещение о календарных сроках 3. Соглашение о требованиях 4. Спецификации 5. Издание документации 6. План испытаний 7. План поддержки 8. Отчеты 9. План выпуска 10. Конфигуратор
I. Исследования II. Анализ осуществимости III. Конструирование IV. Программирование V. Оценка VI. Использование	Ресурсы распределены Требования утверждены Спецификации утверждены Испытания начаты Распространение начато Изделие снято	1, 2 1, 2, 3, 9, 10 1, 2, 4, 5, 6 1, 2, 8 2, 8, 9, 10 10

Ключевые решения, которые должны основываться на результатах фазового обзора, будут ответами на следующие вопросы:

Фаза I. Следует ли вкладывать ресурсы в продолжение анализа осуществимости проекта?

Фаза II. Обоснована ли реализуемость проекта и следует ли расходовать средства на проектирование?

Фаза III. Удовлетворяет ли проект потребностям пользователя в текущий момент времени и следует ли выделять средства для завершения работ?

Фаза IV. Закончена ли разработка изделия и можно ли ему дать объективную независимую оценку?

Фаза V. Достаточно ли высоко качество программного изделия для его поставки пользователю?

Фаза VI. Можно ли прекратить обслуживание программного изделия?

Администраторы планирования выступают в роли организаторов фазового обзора. Однако эту роль могут выполнять и представители различных конкурирующих групп.

Независимо от того, кто проводит обзор, группа планирования обладает решающим голосом в фазовых обзорах I, II, V и VI.

Перечень вопросов, подлежащих рассмотрению в каждом обзоре, строго стандартизирован:

- Строго ли выполняются планы?
- Строго ли соблюдаются все предварительные технические условия?
- Идет ли разработка проекта в соответствии с намеченным графиком?
- Не превышают ли расходы, связанные с проектированием, определенные статьи бюджета?
- Обеспечены ли все необходимые взаимодействия?
- Существуют ли какие-либо оправдания замеченным нарушениям?
- Каков элемент случайности, присутствующий в планах?
- В чем состоят основные трудности?
- Каковы возможные пути преодоления основных трудностей?
- С каким риском связано продолжение работ?
- С каким риском связано прекращение работ?
- Удовлетворяет ли программное изделие в его нынешнем виде текущим требованиям?

Когда эти вопросы изучены, определяются, согласовываются и предпринимаются необходимые действия. Набор этих действий также стандартизован:

- Продолжение работ по плану.
- Пересмотр планов и спецификаций с последующим продолжением работ согласно новым установкам.
- Ввод в действие планов в случае непредвиденных обстоятельств для обеспечения возможности возврата к исходным спецификациям, графикам работ и сметам затрат.
- Прекращение работ.

Проведение фазовых обзоров упрощается за счет использования стандартного обслуживания механизма обсуждения. Некоторые вопросы, поднимаемые в фазовых обзорах, относятся к сфере текущей интерпретации предварительных соглашений. Всякое несоответствие должно устраняться по ходу его обнаружения. Следует помнить, что соглашение о требованиях всегда должно правильно отражать реальную ситуацию, а спецификации должны быть в любой момент времени законченным документом, который правильно описывает, что представляет собой программное изделие.

Управление программным изделием основано на осуществлении контроля и сведении балансов: группа планирования постоянно следит за расхождением между реальным положением дел, связанных с проектированием программного изделия, планами и спецификациями. Механизм рассмотрения и утверждения должен обеспечивать возможность выявления расхождений и последующее их устранение. Технические советы, объединенные комиссии и фазовые обзоры как раз и являются таким механизмом.

8.3. Организация разработки программного изделия

Под управлением проектом мы будем понимать управление достижением требований, предъявляемых к программному изделию на основании использования матричной структуры связи функций и проектов. В этой матрице каждой функции соответствует группа руководителей, несущих ответственность за ее выполнение наилучшим образом, каждому программному изделию соответствует, в свою очередь, группа руководителей, внимание которых сосредоточено только на данном программном изделии (рис 8.6).

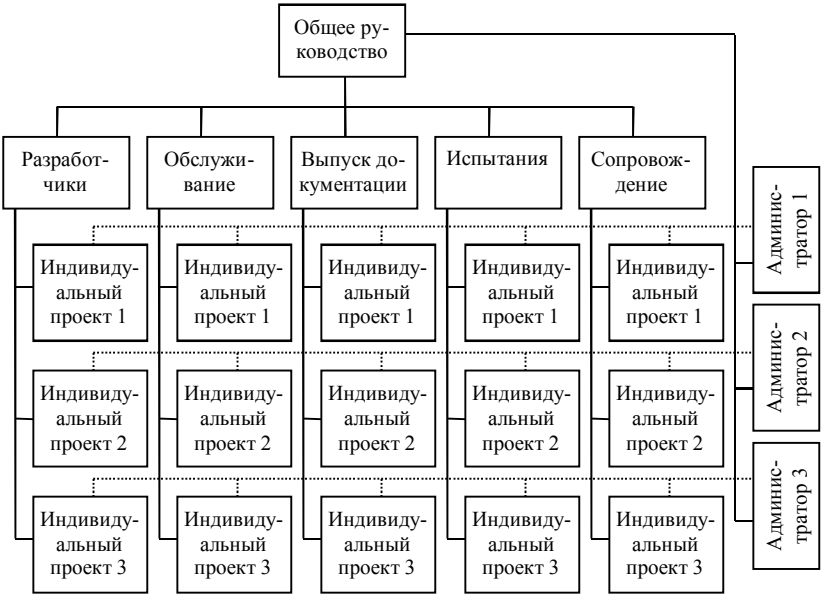


Рис. 8.6 – Схема матричного управления проектом

Администратор проекта (руководитель) занят одним-единственным проектом, каждая функция которого охватывает несколько индивидуальных разработок.

Администратор изделия регулирует степень участия каждой функциональной группы в разработке программного изделия, успешность которой определяется, прежде всего, степенью соблюдения ранее установленных технических требований и целей, допустимых границ затрат. Обычно он начинает выполнять свою роль, начиная с фазы анализа осуществимости и до окончания фазы оценки. Администратор – это, обычно, сотрудник подразделения разработки, хорошо понимающий работы, выполняемые другими людьми и подразделениями (часто бывает, что он несет ответственность за все, не имея полномочий).

В матричной структуре каждый разработчик имеет двух начальников. Однако, именно матричная структура наиболее эффективна для управления такими непредсказуемыми процессами, как разработка программ.

8.3.1. Организация разработки программного изделия в фазе исследований

Затраты труда при реализации функций разработки отдельного программного изделия, согласно кривой Релея, имеют максимальное значение в фазах проектирования и программирования. Осуществление функций разработки начинается в фазе исследований с того момента, когда будет признана необходимость создания конкретного программного изделия. В этом случае план – это план завоевания рынков сбыта, план выпуска серии программных изделий и бюджет. Основной параметр этих планов – срок, к которому возникает необходимость в данном программном изделии. Первая задача, выполняемая в рамках разработки, состоит в том, чтобы определить, когда следует начать разработку изделия, чтобы обеспечить его готовность к моменту, когда в нем возникнет необходимость. Функция разработки предусматривает согласование момента начала проектирования с возможностью выполнения других функций, чтобы гарантировать их совместное обеспечение в процессе проектирования. В рамках функции разработки фиксируются предлагаемые сроки начала и завершения всех видов работ, среди которых выделяют основные этапы проектирования, и запрашиваются ресурсы, необходимые для анализа осуществимости проекта – кадры, машинное время, командировки, фонды для проведения консультаций. Это достигается путем составления сметы затрат, в которой обязательно указывается, кто несет ответственность за выполнение проекта в каждой функциональной группе, здесь же предлагается кандидатура руководителя проекта в целом. Руководитель проекта (администратор) дает указание выполнить анализ результатов работ, выполненных в фазе I (фазовый обзор), и ходатайствует о соответствующем финансировании. Обычно администратор ограничивается выделением средств на работы, связанные с составлением соглашения о требованиях. Такая мера ограничит трату ресурсов, в которых нет необходимости во время фазы анализа осуществимости.

Для правильного принятия решения на основе результатов фазы I оценивается не только стоимость, но и календарные сроки проектирования. Поэтому один из участников проекта, ответственный за подготовку соглашения о требованиях, представляет формально законченный план – график работ. Этот первоначальный вариант плана должен обязательно фиксировать точный срок представления соглашения о требованиях и предполагаемый срок окончания разработки программного изделия.

Если на основании фазового обзора I руководство разрешает начать анализ осуществимости, выделяя для этого соответствующие

ресурсы, то предпринимается попытка составить соглашение о требованиях. Формальное составление соглашения о требованиях облегчается, если придерживаться строго формализованной формы. Требования должны быть выражены в ясной форме, и не допускать противоречивых толкований. В соглашениях о требованиях следует избегать пунктов, объясняющих, как надо решать поставленную задачу. Рассматривая соглашения о требованиях, функциональные группы обычно накладывают жесткие ограничения на эксплуатационные качества и другие характеристики программного изделия. Для обоснования этих ограничений (принять или нет) обязательно должен быть проведен достаточно глубокий анализ осуществимости.

Соглашение о требованиях – это договор между руководителем проекта и заказчиком, который также включает субподряды, заключенные между группой разработки и другими функциональными группами. Договорный характер соглашения о требованиях подчеркивается тем, что привязываются индивидуальные рабочие планы к конкретным положениям данного документа. И т.к. соглашение о требованиях связывает воедино усилия многих подразделений, его необходимо согласовывать на всех уровнях (вплоть до конкретного исполнителя).

На базе соглашения о требованиях составляется общий план для всего проекта. В нем указываются взаимодействия между функциональными группами и вклад каждой из них в проект, а также очерчиваются границы обязанностей каждой группы. В этом плане также дается основа описания всех задач, решаемых группой разработки. Обычно этот план составляется в виде сетевого графика.

8.3.2. Организация разработки программного изделия в фазе анализа осуществимости

Анализ осуществимости проводится одновременно всеми функциональными группами на основе изучения соглашения о требованиях. На этом этапе исследуется каждое предложение соглашения о требованиях. Руководитель проекта поясняет и защищает свой сетевой график. В процессе обсуждения вносятся изменения в эти документы и вновь представляются на утверждение. Может быть предусмотрено несколько неформальных процедур пересмотра плана, однако все они должны быть успешно закончены официальным рассмотрением – фазовым обзором II. В результате фиксируется согласие или несогласие каждой функциональной группы с предложениями соглашения о требованиях.

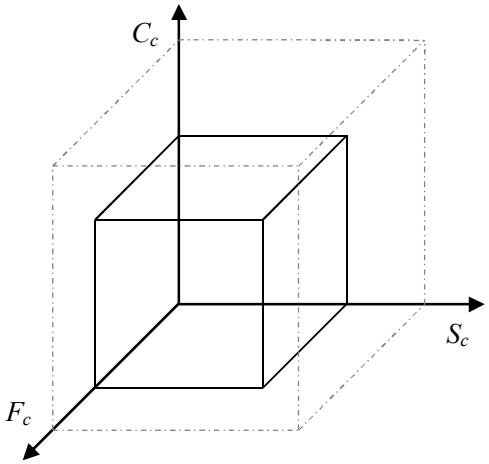


Рис. 8.7 – Границы изменяемости проекта C_c – увеличение стоимости; S_c – изменение времени; F_c – изменение технических характеристик.

Полученные в ходе фазового обзора II предложения относительно внесения изменений в соглашение о требованиях дают возможность оценить недостатки проекта. Оценку проводят по трем переменным: технические характеристики, календарные сроки и стоимость (рис. 8.7).

Т.е. внутри проекта выделяется некоторый диапазон изменения переменных; фактически это соответствует заданию небольшого запаса финансовых ресурсов, времени и свободы выбора технических характеристик системы.

На рассмотрение, переработку и утверждение соглашения о требованиях может уйти много времени, поэтому обычно эта работа совмещается с продолжением разработки в тех масштабах, которые позволяют ресурсы.

Параллельно с процессом рассмотрения соглашения о требованиях составляются индивидуальные рабочие планы и сводка трудозатрат, которые будут основой для определения будущих финансовых затрат.

8.3.3. Организация разработки программного изделия в фазе конструирования (проектирования)

Основная цель проектирования заключается в выработке и анализе требований к программному изделию. Процесс декомпозиции

проекта, начатый при составлении соглашения о требованиях, продолжается путем разработки спецификаций, и разбивки их на две части – внутренний и внешний проект. Внешний проект – это совокупность характеристик программного изделия, которые видит пользователь. Внутренний проект – совокупность характеристик программного изделия, скрытых от пользователя. Это делается, в первую очередь, для того, чтобы пользователи могли критически рассматривать те характеристики программного изделия, которые имеют к ним непосредственное отношение, не вдаваясь в критику внутренних характеристик изделия. Т.е. внешний проект описывает, что делает программное изделие, а внутренние спецификации указывают, каким образом изделие сконструировано для достижения внешних спецификаций. Внешние спецификации передаются для открытого и широкого обсуждения, в котором предпочтения отдаются предложениям пользователей.

Для того, чтобы провести границу между внутренним и внешним проектом, схему декомпозиции упорядочивают. Схема декомпозиции называется хорошо упорядоченной, если на ней отмечен каждый случай вызова одной функции другой, вплоть до некоторого уровня абстракции, удобного с точки зрения управления проектом. Далее, каждый функциональный модуль рассматривается как черный ящик, для которого можно определить внешнее поведение, однако ничего нельзя сказать относительно его внутреннего устройства. Свойства черного ящика определяются полным описанием его характеристик, видимых извне, включая описание условий, при которых к нему можно обратиться, чтобы инициировать его функционирование. Те атрибуты модулей, которые играют существенную роль при составлении описания программного изделия как единого целого, составляют внешний проект. Все остальные параметры модулей, полностью или частично скрытые внутри программного изделия, составляют внутренний проект.

С учетом четкого различия между внешним и внутренним проектом, следует составлять внешние и внутренние спецификации программного изделия. При этом необходимо избегать общих мест в разных документах. Формы спецификаций обычно строго стандартизированы (SADT, PDL, и др.).

После того, как внешние спецификации приобретают сравнительно стабильный характер, в рамках функции разработки производится их рассылка функциональным группам (с пометкой «проект»). Все замечания суммирует специальный орган – технических комитет, который после анализа передает их в группу разработки. Обязанность группы разработки – максимально учесть все замечания.

Во время создания внутренних и внешних спецификаций другие функциональные группы готовят планы выпуска документации, планы испытаний и др. Эти документы рассылаются на рассмотрение в конце фазы проектирования. Руководитель проекта изучает и утверждает их. Фаза проектирования обычно заканчивается утверждением внешних спецификаций.

8.3.4. Организация разработки программного изделия в фазе программирования

Рабочая нагрузка при выполнении функции разработки достигает наибольшей величины в фазе программирования. Основная задача организации разработки заключается в координации усилий большого числа сотрудников, занятых реализацией этой функции, а также в организации взаимодействия между различными функциональными группами. Метод – соблюдение принятых на данном предприятии стандартов программирования.

Кодирование начинается на ранней стадии программирования. При этом используется так называемый «волновой эффект» (табл. 8.3), когда составление внешних и внутренних спецификаций, кодирование, отладка и компоновка программ выполняется одновременно на различных уровнях структуры программного изделия. Например, в фазе программирования какого-то модуля внешние спецификации всего программного изделия уже могут быть утверждены, а внутренние спецификации составлены не до конца. И наоборот, хотя этапы кодирования, отладки и компоновки некоторых модулей уже могут быть завершены, их разработка в рамках целого программного продукта еще может быть не закончена.

Таблица 8.3 – Волновой эффект в разработке модулей изделия

	Модули													
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**N**	**N**
Составление внешних спецификаций завершено	×	×	–	–	×	×	×	–	–	–	×	×	×	×
Составление внутренних спецификаций завершено	×	×	×	×	–	×	×	×	×	–	–	×	×	×
Кодирование закончено	×	×	×	–	–	×	×	×	×	–	–	×	–	×
Отладка закончена	×	×	×	–	–	×	×	×	–	–	–	×	–	×
Компоновка закончена	×	×	×	–	–	–	×	×	–	–	–	×	–	–

Если желательно избрать более жесткий путь и если в программе нельзя выделить сравнительно слабо связанные компоненты, то целесообразно закончить не только внешний проект, но и внутренний и только после этого браться за программирование.

Следует иметь в виду один серьезный недостаток волнового эффекта: задержка с утверждением внешних спецификаций может крайне неблагоприятно отразиться на выполнении многих функций. Т.е. нельзя утверждать спецификации испытаний, невозможно закончить рекламные материалы и т.д.

Помимо кодирования, отладки и компоновки деятельность группы разработки связана с демонстрацией работающего изделия в конце фазы программирования и организацией взаимодействия группы с другими функциональными группами. Сначала на рассмотрение поступает план поддержки. В группе разработки должна, прежде всего, существовать уверенность в обоснованности плановых сроков и правильности предложений группы поддержки, касающихся описания программного изделия. Любые ошибки в описании характеристик про-

граммного изделия или определения сроков его готовности, которые могут дойти до пользователя через рекламные материалы, могут породить недоразумения, или, хуже того, штрафы за невыполнение обязательств.

В фазе программирования группа выпуска документации представляет на рассмотрение ряд вариантов справочных материалов. В середине фазы программирования группой испытаний представляется на рассмотрение график испытаний. Группа разработки тщательно изучает варианты документации и спецификации испытаний с тем, чтобы в них не было ошибок, порожденных неверными исходными предложениями. Если группа разработки в свое время подготовила корректные внешние спецификации, то их анализ не вызывает больших затруднений и займет немного времени. Если же некоторые положения внешних спецификаций пропущены или изложены недостаточно полно, то их проверка отнимет не только много времени, но и вызовет большие трудности. В этом случае придется изменять внешние спецификации, что может свести на нет запас времени, имеющийся в календарном плане проектирования.

Группа поддержки после утверждения изготовляет и направляет на рассмотрение рекламные материалы. Группа разработки анализирует эти материалы, чтобы не допустить технических ошибок, порождающих недоразумения и соответствующие финансовые санкции. Обычно во время фазы программирования оказывается целесообразной демонстрация программного изделия в действии, чтобы показать, что наиболее критические эксплутационные характеристики изделия реализованы в соответствии с требованиями, или чтобы установить, насколько далеко продвинулся проект. Группа разработки стремится закончить этот этап как можно раньше, чтобы учесть замечания тех, кому демонстрировалось программное изделие.

8.3.5. Организация разработки программного изделия в фазе оценки

Фаза оценки открывается началом испытаний класса А, проводимых группой разработки. Испытания класса А – это всесторонняя проверка программного изделия, которая начинается после того, как все модули программ были подвергнуты индивидуальной проверке и включены в работоспособную систему. Испытания класса А начинаются сразу же после того, как в систему включен последний модуль.

Проводя испытания класса А, группа разработки прогоняет как можно больше контрольных примеров, предложенных группой испытаний. Это ускоряет фактическое завершение независимых испытаний,

которые проводит группа испытаний, и помогает устранить ошибки в самих тестах, являющиеся часто причиной разногласий между этими двумя группами. К концу испытаний класса А группа разработки подготавливает спецификацию выпуска – документ, который связывает воедино составные части программного изделия. Форма спецификации строго стандартизована. После того, как число ошибок, обнаруживаемых во время испытаний класса А, становится незначительным, группа разработки начинает приемочные испытания по программе, составленной группой испытаний.

Приемочные испытания основываются на наборе тестов, выбранных из общей программы испытаний, и предназначены для выявления недостатков плохо продуманного программного изделия. К сожалению, под впечатлением результатов испытаний группа разработки склонна к проведению поспешных изменений в модулях, которые могут разрушить целостность программного изделия. Проведение приемочных испытаний убеждает всех, что для внесения дополнительных изменений в модули нет оснований.

Испытания класса В представляют собой независимую проверку программного изделия на соответствие спецификациям. Программное изделие считается готовым к проведению испытаний класса В после успешного проведения приемочных испытаний. Группа разработки составляет отчет об испытаниях класса А, подытоживая результаты этих испытаний, в том числе и приемочных, свидетельствующих о готовности программного изделия к испытанию класса В.

Группа испытаний подготавливает набор тестов. Испытания на основе этих тестов обычно проводятся циклами, начиная с повторных приемочных испытаний. Цикл испытаний предполагает прогон как можно большего числа тестов в максимально сжатые сроки и завершается отчетом о результатах испытаний, который направляется в группу разработки. Если после цикла испытаний в программном изделии будут обнаружены недостатки, препятствующие его выпуску, группа разработки с максимальной быстротой реагирует на результаты цикла и предъявляет программное изделие в исправленном виде для нового цикла испытаний. Группа разработки получит наивысшую оценку, если испытания класса В пройдут за один цикл. Хотя это иногда и случается, чаще всего приходится проводить около трех циклов испытаний. Однако на практике известны случаи, когда количество циклов достигало 10.

В то время, как группа испытаний проводит испытания класса В, группа выпуска документации представляет на рассмотрение справочные материалы. Группа разработки имеет последнюю возможность

исправить ошибки в этих материалах, и поэтому их рассмотрение должно проводиться наиболее тщательно. Группа выпуска документации учитывает замечания разработчиков и проводит последний просмотр материала перед сдачей в печать.

Фаза оценки заканчивается тогда, когда группа испытаний излагает свои замечания в отчете об испытаниях класса В. Отчет составляется после того, как группа испытаний приходит к выводу, что программное изделие удовлетворяет или не удовлетворяет критериям испытаний. Чаще всего при испытаниях выявляется ряд нерешенных проблем, к рассмотрению которых привлекаются разработчики. Решение о выпуске программного изделия для широкого использования принимается на основе отчета группы испытаний и пояснительной записки группы разработки, которая обычно предлагает план устранения недостатков. Поэтому группа разработки тщательно изучает отчет об испытаниях класса В и рекомендует меры для устранения всех замеченных дефектов. При этом группа разработки может вступать во взаимодействие с группой сопровождения, если выявленные дефекты могут быть компенсированы какими-либо средствами во время эксплуатации.

8.3.6. Окончание проекта

Независимо от причин, вызвавших завершение проекта, группа разработки выпускает отчет, из которого могут почерпнуть опыт разработчики будущего проекта. Заключительный отчет составляется всеми участниками проекта и содержит, как минимум, следующую информацию:

- опыт преодоления наибольших трудностей, встретившихся при разработке проекта;
- рекомендации по разработке последующих проектов (включая различные варианты);
- сводку плановых и фактических сроков выполнения этапов (включая все случаи изменения запланированных сроков);
- сводку запланированных и фактических расходов;
- сводку запланированных и фактических трудозатрат;
- сводку запланированных и фактически используемых машинных ресурсов;
- хронологию затруднений в работе с оборудованием и рекомендации по устранению недостатков в будущем;
- хронологию возникновения трудностей, связанных с взаимодействием функциональных подразделений;

- рекомендации по планированию в условиях неопределенности;
- хронологическую запись наиболее значимых событий.

Если разработка программного изделия завершена полностью, то заключительный отчет включается в спецификацию сопровождения.

Нормальное завершение проекта наступает на этапе фазового обзора V, когда принимается решение о выпуске программного продукта. Группа разработки составляет заключительный отчет как можно быстрее, прежде чем сотрудники проекта окажутся занятыми своими новыми обязанностями. Она также выпускает заключительное уведомление о календарных сроках и просит о закрытии финансового счета. В случае нормального завершения проекта сотрудники переключаются на другую работу. Преждевременное завершение работ (прекращение финансирования, исчезновение необходимости в программном изделии) обычно застает разработчиков врасплох. В любом случае, руководитель обязан обеспечить порядок перехода на новую работу. Он должен проследить, чтобы сотрудники завершили документирование, сдали в архив все необходимые данные (тексты программ, тесты и т.д.), составили заключительный отчет. Заключительные операции надо проводить и в том случае, если проект окажется неудачным.

8.3.7. Участие группы разработки в фазовых обзорах

Группа разработки участвует в пяти из шести предусмотренных фазовых обзорах (табл. 8.4). В фазовом обзоре I эта группа дает первоначальную оценку стоимости проекта и составляет предварительный график проектирования. Она рассматривает все представленные данные, утверждает календарные сроки, в особенности срок готовности соглашения о требованиях, а также распределение ресурсов. На этом этапе целесообразно планировать только те расходы, которые необходимы для доведения проекта до этапа утверждения соглашения о требованиях, что позволит избежать перерасхода средств, если этот документ не будет утвержден, или если придется в значительной степени пересмотреть характеристики программной системы.

В фазовом обзоре II в центре внимания находится соглашение о требованиях.

Таблица 8.4 – Участие группы разработки в фазовых обзорах

Фаза	Фазовый обзор	Форма участия при обсуждении документов
Исследование	I	Рассмотрение и утверждение
Анализ осуществимости	II	Рассмотрение и утверждение
Конструирование	III	Рассмотрение и утверждение
Программирование	IV	Рассмотрение и утверждение
Оценка	V	Рассмотрение и утверждение
Использование	VI	Не участвует

Группа разработки рассматривает технические требования, распределение бюджета, новый календарный план, а также некоторые элементы предварительного варианта конфигуратора и график выпуска изделия. Она утверждает соглашение о требованиях, выделение дополнительных средств из распределения бюджета и дополнительные обязательства по календарным срокам.

В период между II и III фазовыми обзорами группа разработки проверяет и утверждает планы выпуска документации и испытаний. В фазовом обзоре III внимание фокусируется на внешних спецификациях. Кроме того, группа разработки рассматривает распределение бюджета и новое извещение о календарных сроках. Здесь же утверждаются внешние спецификации, уточненное распределение бюджета и новые обязательства по календарным срокам.

В период между фазовыми обзорами III и IV группа разработки анализирует и утверждает план группы поддержки, спецификации испытаний, а также имеющиеся рекламные материалы. Для фазового обзора IV она представляет отчет об испытаниях класса А и запрашивает разрешение на испытание класса В. Затем группа разработки вновь рассматривает распределение бюджета и извещения о календарных сроках, утверждая необходимые поправки в документах.

В период между фазовыми обзорами IV и V группа разработки участвует только в редактировании и заключительном рассмотрении выпускаемой документации, которое заканчивается ее утверждением. В фазовом обзоре V предметом рассмотрения является отчет об испытаниях класса В. Группа разработки изучает этот документ и утверждает соответствующее решение, принятое на основе результатов испытаний. Она также проверяет календарный план и утверждает изменения к нему. На этом заканчивается участие группы разработки в рас-

смотрении и утверждении документов проекта. В фазе использования, которая начинается после фазового обзора V, группа разработки остается свободной и приводит в порядок проектную документацию, завершая проект. К моменту завершения фазового обзора VI группа разработки уже не участвует в проекте.

8.4. Организация обслуживания разработки программного изделия

При любом способе организации разработки программного изделия необходима группа обслуживания, в функции которой входят:

- производственные операции – приобретение, эксплуатация и обслуживание вычислительного оборудования и программных средств;
- распространение – регистрация и рассылка программных средств;
- системное обеспечение – введение и слежение за соблюдением стандартов, процедур и форматов;
- конфигурационное управление – идентификация и проверка всех частей программного обеспечения.

Группа обслуживания также участвует в решении всех административных, кадровых и производственных проблем, возникающих при разработке программных средств.

Системное обслуживание включает управление планами, т.е. наблюдение за составлением и фактическим выполнением планов. Это означает, что, по крайней мере, один человек – администратор планирования – часть своего времени посвящает выполнению функции обслуживания.

Конфигурационное управление включает контроль, регистрацию и распространение проектной документации, т.е. эти функции перекликаются с функциями группы выпуска документации.

8.4.1. Организационная структура группы обслуживания

Общий объем и номенклатура услуг, оказываемых в рамках функции обслуживания, определяет численный состав и внутреннюю структуру группы обслуживания. Как уже отмечалось, управление планами может занимать либо часть рабочего времени одного сотрудника, либо все время нескольких сотрудников. Подобное утверждение справедливо для любого вида деятельности, т.е. в реальных условиях группа обслуживания постоянно реорганизуется, чтобы удовлетворить множество новых запросов. Основная функция группы обслуживания – своевременно обеспечивать выполнение запрашиваемых услуг с минимальными затратами. Другие функциональные группы не обязаны

отчитываться перед группой обслуживания в правомерности своих запросов.

Основной ошибкой в деятельности группы обслуживания является «локальная оптимизация» при потере целей глобальной оптимизации (одна ЭВМ вместо двух для территориально рассеянных групп разработчиков, централизация материально-технического обеспечения и др.). *Функциональная группа должна быть обеспечена всем необходимым, даже если это противоречит локальной оптимизации.*

8.4.2. Организация обслуживания программного изделия в фазе исследования

В фазе исследований группа обслуживания занимается исключительно планированием своей работы на будущее, т.к. пока не будет доказана осуществимость проекта, т.е. пока не будут утверждены соглашения о требованиях, группа обслуживания решает лишь одну задачу: определить технические средства, необходимые для разработки программного изделия. Но поскольку связанные с ними расходы могут быть значительными, группе обслуживания следует воздержаться от фактического приобретения этих средств до утверждения соглашения о требованиях.

После утверждения первоначального распределения бюджета группа обслуживания вместе с руководителем проекта планирует потребность в машинных ресурсах, объемы канцелярских работ и другие виды постоянных услуг. После утверждения бюджета начинается работа и по учету затрат по бюджету. Группа обслуживания регулярно поставляет сводки о расходах, одновременно группа обслуживания осуществляет учет выполнения календарных сроков, составляет регулярные сводки о ходе выполнения календарного плана.

Конфигурационное управление также начинается после утверждения бюджета. Администратор планирования заводит в группе контроля дело на данный проект. С этого момента он наблюдает за тем, чтобы вся проектная документация, выпускаемая в соответствии с планом, подшивалась в дело.

8.4.3. Организация обслуживания в фазах анализа осуществимости и конструирования

В конце фазы исследований руководитель проекта передает на рассмотрение составленное им соглашение о требованиях. Администратор планирования проверяет его на совместимость с существующими планами и передает руководителю проекта результаты своего анализа. Группа обслуживания также изучает соглашение о требова-

ниях с точки зрения потребностей в обеспечении научно-исследовательских работ техническими и другими ресурсами. Средства, приобретение которых занимает большой период времени, заказываются заранее, однако фактически закупка начинается после утверждения соглашения о требованиях.

Группа обслуживания обычно имеет в своем распоряжении вычислительный центр или исследовательскую лабораторию, располагающую соответственным оборудованием и программными средствами, которые обеспечивают испытания класса С. Целью испытаний класса С является проверка возможности внедрения программного изделия и его совместимости с различными конфигурациями программной и аппаратной среды. Наибольший эффект дает это испытание силами малоквалифицированных работников, которые, хотя и не знакомы с объектом испытаний, тем не менее, должны суметь определить его необходимую конфигурацию и возможность ввода в действие.

Соглашение о требованиях определяет характер испытаний класса С и необходимую квалификацию персонала, проводящего эти испытания. Участвуя в этих испытаниях, группа обслуживания проверяет соответствующие пункты соглашения о требованиях. Кроме того, группа обслуживания выполняет те пункты соглашения о требованиях, в которых описывается распространение изделия – куда, каким образом, на каких носителях и какие компоненты изделия должны быть поставлены. Группа обслуживания проверяет компоненты изделия, а также условия поставки и ввода его в действие, чтобы убедиться, что они удовлетворяют принципам конфигурационного управления и что в распоряжении группы обслуживания имеются необходимые средства для распространения изделия.

К этому времени следует принять решение относительно защиты прав собственности на программное изделие. Оно может быть *запатентовано*, на него могут быть заявлены *авторские права* или *право собственности* владельца. Выбранный способ защиты прав следует оговорить в соглашении о требованиях, чтобы в рамках функции обслуживания можно было правильно зарегистрировать компоненты программного изделия и установить контроль за его распространением.

Сразу же после утверждения соглашения о требованиях группа обслуживания приступает к фактическим закупкам оборудования на основе предварительных заявок. Вскоре после утверждения соглашения о требованиях группа испытаний передает свой план испытаний. Этот план проверяется, прежде всего, в группе обслуживания, которая убеждается, что для проведения испытаний класса В и С имеются со-

ответствующие программные и технические средства. Для приобретения и установки этих средств заключаются окончательные договоры.

К моменту передачи плана испытаний руководству уже подготавливается отчет о состоянии проекта. Группа обслуживания принимает участие в подготовке этого отчета.

8.4.4. Организация обслуживания в фазе программирования и оценки

В фазах программирования и оценки группы разработки и испытаний требуют наибольшего внимания со стороны группы обслуживания, и, в первую очередь, это относится к обеспечению проводимых работ вычислительными ресурсами. В этих фазах по различным причинам могут произойти значительные отставания фактических сроков от плановых. Именно в таких ситуациях группа обслуживания работает с максимальной нагрузкой. Предусмотрительный руководитель технической службы всегда имеет запланированные резервы времени и технических средств, даже если руководитель проекта уже исчерпал все свои ресурсы.

Другая задача группы обслуживания в фазе программирования состоит в изучении спецификаций испытаний и подготовке необходимого материально-технического обеспечения для них.

По мере перехода от фазы программирования к фазе оценки нагрузка на группу обслуживания возрастает. Спустя некоторое время после начала испытаний класса А, возможно, появится необходимость передать изделие одному или нескольким пользователям. Такой выпуск называется *предварительным*. В этом случае группа обслуживания выпускает формуляр предварительного выпуска. Он содержит сведения о причинах предварительного выпуска изделия, конкретном его получателе, а также уведомление об отсутствии его поддержки. Данные, приводимые в формуляре, утверждают, что выпуск изделия является исключением из правила, и что пользователь должен сопоставить все выгоды и потери, связанные с использованием неготовых программных средств.

Вскоре после начала испытаний класса В группа обслуживания начинает готовиться к распространению изделия. Основной задачей на этом этапе является подготовка информационного листка выпуска, который составляется для пользователей и указывает, какие компоненты программного изделия подлежат передаче и как их внедрять. Информационный листок выпуска подготавливается на основе материалов группы разработки и передается в группу испытаний, где исполь-

зуется в качестве одного из руководящих документов приемных испытаний класса И.

Информационный листок выпуска принимает законченный вид после того, как группа испытаний закончит испытания и составит отчет по испытаниям класса В.

В информационным листке описываются все «незначительные» дефекты программного изделия, обнаруженные во время испытаний класса В (как в программах, так и в документации). Следующий шаг – производство программного изделия. Оно включает в себя следующие действия:

тиражирование программ;

упаковку и отправку потребителям;

контроль качества изделия путем испытания класса С.

Эти испытания начинаются сразу же после появления окончательной редакции информационного листка. После принятия решения о выпуске группа обслуживания печатает информационные листки. Затем группа обслуживания подготавливает достаточное количество экземпляров программного изделия (плюс несколько про запас), комплектует и упаковывает комплекты.

Далее, она завершает испытания класса С, выбирая в качестве объекта испытаний (случайным образом) один или несколько комплектов программного изделия.

Группа обслуживания несет ответственность за правильность распространения конкретных компонентов программного изделия среди определенных потребителей; фиксирует авторские права и права собственности на переданные материалы, что позволяет в любое время контролировать их распространение; следит за тем, чтобы пользователи подписывали соответствующие обязательства, гарантирующие принятие мер против вторичного распространения; ведет полный и точный учет пересылок материалов и, что наиболее важно, не разрешает вносить (кому бы то ни было) самостоятельные изменения в передаваемый пользователю комплект материалов.

Группа обслуживания также обеспечивает защиту программных средств от стихийных бедствий. Она создает дубликаты и обеспечивает хранение всех жизненно важных материалов в географически удаленных архивах, обеспеченных противопожарной защитой.

Заключительный этап процесса распространения состоит в том, что группа обслуживания создает специальную контрольную ведомость по распространению программного продукта по строго стандартизованной форме.

8.4.5. Организация обслуживания в фазе использования

Пока программное изделие не будет передано всем пользователям, группа обслуживания проводит испытания класса С.

Использование программного изделия продолжается до тех пор, пока оно отвечает интересам производителя программных средств, и в то же время производитель должен быть готов к соответствующей поддержке программных средств. Для группы обслуживания это означает организацию исправления ошибок, редактирование и расширение возможностей программного изделия.

Все замечания пользователей по дефектам продукта поступают в группу сопровождения. Группа обслуживания вместе с группой сопровождения обрабатывает их, классифицирует и направляет уведомление о получении заявки автору. Различаются три вида заявок:

заявки на исправление ошибок;

заявки на проведение проверки;

заявки на расширение функций.

Сообщение о подозреваемой ошибке называется *заявкой на исправление*. Обязательное изменение программного изделия для обеспечения его совместимости с конкретной конфигурацией ЭВМ – *заявка на проведение проверки*. Желательное, но не обязательное, усовершенствование программного изделия – *заявка на расширение*.

Важность уведомления автора о поставке на учет посланного им запроса обусловлена двумя причинами:

- такое уведомление убеждает пользователя, что его заявка не осталась незамеченной;
- в уведомлении всегда указывается срок, к которому будет дан ответ.

Т.е. пользователь знает, сколько ему осталось работать в условиях неопределенности, и группа сопровождения вынуждена провести анализ в указанный срок.

При обработке заявки на исправление ошибки копия передается в группу поддержки для учета и контроля, а сама заявка (вместе с уведомлением) – в группу сопровождения и группу испытаний. Группа обслуживания сообщает пользователю о результатах анализа. Если анализ показывает, что пользователь ошибся, неправильно интерпретируя функционирование программного изделия, или его предложение требует модернизации программы, то по заявке не принимается никаких действий, и об этом уведомляются все группы. Если ошибка действительно существует, группа сопровождения описывает предполагаемый способ ее разрешения и составляет график работ по этой заявке.

После завершения работ по этой заявке группа сопровождения сообщает о результатах группе испытаний и группе обслуживания. Группа обслуживания уведомляет о результатах рассмотрения заявки пользователя, а также группу поддержки, осуществляющую контроль и учет. Если в ответе предусматривается изменение текста программ, группа испытаний подготавливает контрольные примеры для проведения повторных испытаний, а группа обслуживания контролирует выполнение заявок, ожидая исправленной редакции программ.

Порядок рассмотрения заявок на расширение и проверку программного изделия мало чем отличается от обработки заявок на исправление.

Новая редакция программного изделия, полученная после обработки заявок на исправление (или расширение), вызывает иногда необходимость повторного рассмотрения в фазе использования, которое осуществляется группой обслуживания в том же порядке, как и первоначальное распространение.

8.4.6. Участие группы обслуживания в фазовых обзорах

В фазовом обзоре I группа обслуживания контролирует распределение бюджета проекта и обсуждает задачи, возлагаемые на нее после того, как проект минует фазу осуществимости. Основное внимание сосредотачивает на оценке потребности в оборудовании и других средствах, которые понадобятся в процессе разработки. Т.к. в этот период закупки не производятся, роль этой группы ограничивается лишь участием. Никакие документы она не утверждает.

Таблица 8.5 – Участие группы обслуживания в фазовых обзорах

Фаза	Фазовый обзор	Форма участия при обсуждении документов
Исследование	I	Рассмотрение
Анализ осуществимости	II	Рассмотрение и утверждение
Конструирование	III	Не участвует
Программирование	IV	Рассмотрение
Оценка	V	Рассмотрение
Использование	VI	Рассмотрение

В фазовом обзоре II группа обслуживания ведет переговоры о приобретении оборудования и других материалов для проведения испытаний класса С и распространения программного изделия. Она также рассматривает и планирует растущий объем услуг в конфигураци-

онном управлении, защите прав собственности, ведении документации и управлении сопровождением.

Группа обслуживания не участвует в фазовом обзоре III, однако в промежутках между фазами II и IV она рассматривает и утверждает план испытаний, чтобы убедиться в соответствии этих документов требованиям испытаний класса С. Она также изучает и утверждает обязательства по обеспечению испытаний аппаратурой и другими средствами в соответствии с планом испытаний.

За исключением управления планами и утверждения календарных планов, группа обслуживания не принимает участие в других обзорах. Администратор планирования рассматривает соглашение о требованиях в фазе II, конфигуратор – в фазах III, IV и V и календарный план – в фазах I-IV. Здесь он выступает в роли консультанта, и поэтому не участвует в утверждении.

8.5. Организация выпуска документации

Термины «документация» и «справочные материалы» относятся к печатной продукции, предназначенной для пользователей программного изделия. Термин «документы» используется по отношению к планам и спецификациям, которые изготавливаются для сотрудников, участвующих в проектировании. Следует различать *рекламные* и *справочные* материалы, т.к. они предназначены для разных категорий пользователей и при их создании ставятся разные цели.

Рекламные материалы предназначены преимущественно для административного персонала, в то время как справочные материалы используются операторами, программистами и системными аналитиками. Группа поддержки составляет рекламные материалы в рамках своей деятельности, связанной со сбытом, а изготовление справочного материала входит в обязанности группы выпуска документации.

8.5.1. Организационная структура группы выпуска документации

Прежде всего, очевиден тот факт, что большинство программистов не любит писать никаких бумаг, кроме программ, поэтому введение специальной группы для выполнения этих работ достаточно обосновано. Централизация планирования и выпуска документации играет важную роль в обеспечении согласованности различных материалов по программному изделию. Эти цели можно достичь, создав такое организационное подразделение, в котором могут работать бригады редакторов и производственные бригады.

Редакторы могут быть прикреплены к конкретному проекту, и иметь различные служебные полномочия как в рамках функции вы-

пуска документации, так и по отношению к опекаемым ими группам разработчиков. Цели группы выпуска документации направлены на создание эффективного рабочего взаимодействия с группой разработки.

Производственные бригады должны находиться целиком в подчинении группы выпуска документации. Это технические редакторы, машинистки, художники, граверы и др.

Для осуществления деятельности группы выпуска документации важное значение имеет установление и соблюдение стандартов, регулирующих работу редакционных и производственных бригад. Создание стандартов обычно поручается одному или нескольким аналитикам группы выпуска документации.

Положение группы выпуска документации внутри проектной организации зависит от характера и объема выполняемых работ (либо в отделе сбыта, либо в разрабатывающих подразделениях) (рис. 8.8).

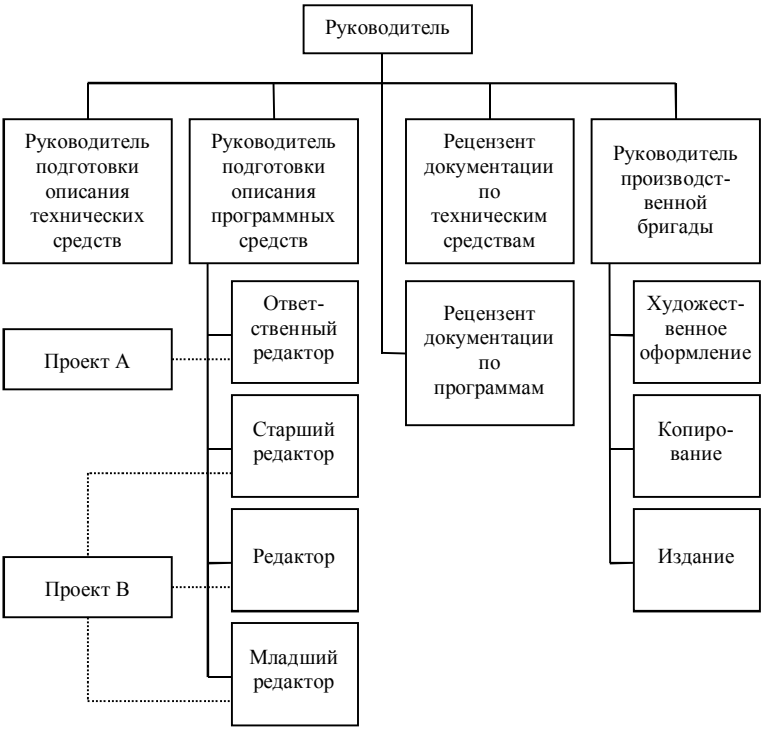


Рис. 8.8 – Организационная структура группы выпуска документации

Комплектование производственной бригады не вызывает проблем, однако найти редакторов очень нелегко. Трудно (или невозможно) заставить переквалифицироваться программиста на редактора, еще труднее филолога обучить писать материалы по программному изделию. Высокая стоимость подготовки рукописей по программному изделию заставляет руководителей давать преимущества производственного роста именно редакторам, тем самым, привлекая программистов к этой работе.

8.5.2. Стандарты и практические руководства

Если не установить правила оформления документации, редакторы, художники и т.п., подобно программистам, будут действовать в зависимости от своих вкусов и наклонностей. Чтобы продукция имела профессиональный вид, следует начать с подбора соответствующих стандартов.

Одним из важнейших стандартов является *руководство по стилю*. В нем устанавливаются виды документации и даются основные рекомендации по оформлению: размер страницы, шрифт, стиль художественного оформления, требования к словарю терминов, порядок индексирования, способ нумерации разделов и отыскания нужных сведений, формат печати и др. Выделяются следующие основные виды документации:

- Техническое описание системы. Это введение, излагающее основные концепции программного изделия. Оно дает будущему пользователю сведения о принципах функционирования и возможностях программного изделия, характеризует программное изделие в терминах наиболее общих его функций и освещает самые существенные свойства. Изложение должно быть четким и касаться наиболее важных характеристик системы. Это описание предназначено для лиц, которые могут быть не знакомы с программированием и принципами работы ЭВМ, но которые определяют выбор и приобретение средств вычислительной техники. Основным первоисточником для технического описания должно быть соглашение о требованиях, дополненное материалами групп разработки и поддержки.
- Справочное руководство. В нем подробно описаны все внешние характеристики программного изделия. Оно обеспечивает пользователю набор информации о назначении программного изделия, о необходимом оборудовании и его конфигурации, о языках программирования, включая форматы команд и сообщений, о структурах входных и выходных данных, о взаимосвязях с другими программными изделиями. При этом предполагается, что читатель

справочного руководства имеет опыт работы с программами и техническими средствами ЭВМ. Справочное руководство составляется на основе внешних спецификаций. Дополнительный материал поставляет группа разработки. Это руководство предназначено для программистов и системных аналитиков, которые являются непосредственными пользователями программного изделия.

- Справочный буклет. Он обычно имеет карманный формат и содержит краткие сведения из справочного руководства. Он позволяет быстро ознакомиться с форматами операторов, мнемоникой операций, таблицами преобразования кодов, синтаксисом языка. В наиболее сжатой форме этот документ называют справочным формуляром.
- Руководство оператора содержит информацию для пользователей, которая указывает, как эксплуатировать программу и соответствующее оборудование. Оно описывает требуемые характеристики технических средств, управляющие сообщения и рабочие процедуры. В характеристиках оборудования излагаются сведения о минимально необходимой конфигурации, специальных возможностях оборудования и способах закрепления аппаратных ресурсов за программами. В описании управляющих сообщений даются их форматы и правила использования, а также конкретные примеры. В рабочих процедурах описываются все последовательные этапы управления системой, включая начальную загрузку, выполнение заданий и восстановление после отказов. Основным источником для написания руководства оператора служат внешние спецификации. Дополнительная информация поступает от группы разработки и группы испытаний. Руководство предназначено для операторов, которые могут и не иметь большого опыта работы.
- Указатель системных сообщений. В этом документе описываются все сообщения, порождаемые программой, а также ситуации, в которых они возникают, и необходимые действия оператора. Указатель составляется, в основном, по листингам программ и предназначен для операторов, программистов и системных аналитиков.

В интерактивном режиме появляется возможность оперативно выводить справочную информацию на экран. В таких системах выдается большое количество сообщений – подсказок и сообщений об ошибках. Группа разработки проектирует лишь программный механизм подобного взаимодействия, ответственность за окончательную реализацию текстов сообщений, их стиль и форматы несет группа выпуска документации. Она приводит и проверяет соответствие печатных материалов сообщениям, выдаваемым в режиме on-line.

Кроме соблюдения стиля документов, группа выпуска документации определят поток работ, начиная от планирования и кончая распространением программного изделия. В этой инструкции описываются методы, позволяющие планировать процесс выпуска документации и контролировать выполнение календарного плана.

8.5.3. Организация выпуска документации в фазах исследований и анализа осуществимости

Группа выпуска документации начинает участвовать в работе над программным изделием в рамках фазового обзора I, выполняя предварительную оценку стоимости издания документации. Для этого приходится принимать некоторые предположения относительно вида документации, времени издания и расценок. Чем определенней составлены стандарты оформления документации, тем точнее могут оказаться оценки, полученные в этой фазе.

В фазовом обзоре I начинается рассмотрение извещения о календарных сроках. На этом этапе необязательно готовить подробный план выпуска документации. После фазового обзора I эта группа фактически перестает участвовать в проекте, вплоть до утверждения соглашения о требованиях.

В утвержденном соглашении о требованиях указывается, какие виды документации надо издавать, когда они будут готовы и в каких видах выражается дополнительное участие группы выпуска документации в работе над программным изделием. Если эта группа изъявляет желание участвовать в рассмотрении внешних спецификаций, она может быть включена в техническую ревизионную комиссию.

Если группа выпуска документации не планирует издание каких-либо документов, отличающихся от стандартных, то экспертам, изучающим соглашение о требованиях, достаточно будет рассмотреть в этом соглашении полноту документов, предусмотренных стандартом. При этом участие группы выпуска документации в составлении соглашения о требованиях будет минимальным.

В период между составлением соглашения о требованиях и его утверждением, на этапе фазового обзора II, все заинтересованные группы должны оценить требования к программному изделию и его возможности. Хотя группа выпуска документации привлекается только для консультаций при составлении соглашения о требованиях, она тщательно изучает его, чтобы убедиться, что ее обязательства соответствуют назначению программного изделия.

8.5.4. Организация выпуска документации в фазах конструирования и программирования

Активная работа группы выпуска документации начинается в этих фазах. В фазе конструирования создаются макеты будущей документации. План выпуска документации детализирует положения соглашения о требованиях, касающиеся документации. В них учитываются сроки проведения работ в рамках других функций и ранее проведенных изменений проекта. В них учитываются стандарты, характеристика документации и этапы проверки. Иногда план выпуска документации для нескольких проектов объединяется. Утвержденный план подкрепляется индивидуальными планами.

В фазе конструирования группа разработки передает внешние спецификации на рассмотрение другим группам. Группа выпуска документации убеждается в том, что внешние спецификации имеют завершенный вид, и начинает готовить рукописи справочных материалов. Если в документацию включено техническое описание, то работа начинается именно с этого документа, т.к. он должен быть готов в то же самое время, когда группа поддержки выпустит рекламные материалы.

Если для программного продукта предусмотрен выпуск справочного материала в режиме on-line, группа выпуска документации совместно с группой разработки подготавливает внешнюю спецификацию на изделие, в которой описываются форматы и содержание всех сообщений, выдаваемых пользователю.

Максимальная нагрузка группы выпуска документации наступает в фазе программирования. Группа изучает план группы поддержки, чтобы убедиться, что рукописи уже изданных материалов, необходимые для обучения пользователей или для создания рекламных материалов, появятся к заданному сроку. В течение этой фазы группа выпуска документации один или несколько раз передает на рассмотрение группы разработки рукописи справочных материалов с тем, чтобы выявить расхождения в технических деталях.

В конце фазы программирования на рассмотрение поступают рукописи материалов по обучению пользователей, подготовленные группой поддержки. Группа выпуска документации проверяет эти материалы на соответствие стилю и содержанию остальной документации.

С момента утверждения внешних спецификаций и до начала испытаний класса В группа выпуска документации следит за всеми изменениями внешних спецификаций. Т.к. она не участвует в обсуждении, то ей приходится соглашаться с вносимыми изменениями и

настаивать на соответствующих изменениях сроков издания и объемов финансирования. Это осуществляется путем участия в технической ревизионной комиссии.

8.5.5. Организация выпуска документации в фазах оценки и использования

В начале фазы оценки группа выпуска документации заканчивает составление окончательных вариантов всей документации вместе с иллюстрациями. Рукописи направляются на рассмотрение другим группам. Группы подготавливают перечень дефектов, замеченных в документации. Решающая роль при анализе документации принадлежит группе испытаний. Если группа испытаний удовлетворена содержанием рукописей, последние сдаются в печать, и готовая документация поступает на распространение.

В фазе оценки группа обслуживания подготавливает руководство по вводу программного изделия в действие – *информационный листок выпуска*. Этот листок создается в соответствии со стандартами группы выпуска документации, которая и обеспечивает его выход в свет.

Группа выпуска документации привлекается к обсуждению решения о выпуске изделия, если будет предложено оставить без исправления замеченные в документах дефекты. Это возможно лишь в том случае, если устранение дефектов приведет к значительным затратам.

После того, как изделие вступило в фазу использования, сопровождение документации осуществляется по той же схеме, что и сопровождение программного изделия. Несмотря на меры анализа и контроля, пользователи обнаруживают некоторые ошибки в программном изделии. По мере поступления сведений о замеченных ошибках (заявки на техническое обслуживание) они устраняются, и накопленные исправления вносятся в документацию. Коррекция осуществляется несколькими способами:

- заново перепечатывается весь материал;
- печатаются отдельные страницы с перечнем исправлений;
- перепечатываются целые страницы для замены страниц с ошибками.

Рассылка корректирующего материала осуществляется либо путем непосредственного исправления, либо путем включения перечня исправлений в информационные листки выпуска для издаваемых новых выпусков изделия, или в ответ на заявку на техническое обслуживание. Независимо от формы проведения коррекции или способа распространения, должна соблюдаться дисциплина конфигурационного

управления. Каждое издание документации, каждая замена страницы и каждый перечень исправлений должны содержать указание на конкретную внешнюю спецификацию и наоборот.

8.5.6. Участие группы выпуска документации в фазовых обзорах

До фазового обзора I в группу выпуска документации поступает запрос на проведение предварительных оценок необходимых ресурсов, на основании которых производится распределение бюджета. Утверждение средств происходит в ходе фазового обзора II.

В ходе фазового обзора II группа пересматривает свои исходные позиции в отношении соглашения о требованиях, руководствуясь окончательным бюджетом, и утверждает пересмотренное соглашение о требованиях. Проводится также анализ распределения бюджета внутри проекта, после чего группа выпуска документации утверждает расходы на печатную продукцию. Аналогичным образом утверждается календарный план.

Таблица 8.6 – Участие группы выпуска документации в фазовых обзорах

Фаза	Фазовый обзор	Форма участия при обсуждении документов
Исследование	I	Рассмотрение
Анализ осуществимости	II	Рассмотрение и утверждение
Конструирование	III	Рассмотрение
Программирование	IV	Не участвует
Оценка	V	Не участвует
Использование	VI	Не участвует

До фазового обзора III группа участвует в работе над внешней спецификацией. Ко времени фазового обзора III она должна изучить внешнюю спецификацию и убедиться, что в ней правильно представлены интересы группы выпуска документации.

В фазовом обзоре IV все внимание участников проекта сосредоточено на передаче программ в распоряжение группы испытаний. Группа выпуска документации участвует в этом обзоре лишь в том случае, когда для создания документации нужны дополнительные источники, или если план выпуска документации нуждается в коррекции.

В фазовом обзоре V группа выпуска документации участвует только в том случае, когда нужны ее рекомендации при выпуске изделия с недоработками.

В фазовом обзоре VI основное внимание уделяется ослаблению поддержки изделия или ее полной отмене. К этому моменту изменения в документации настолько незначительные, что группа в этом обзоре не участвует.

8.6. Организация испытаний программных изделий

В широком смысле слова, под испытаниями понимают не отладку, призванную определить, почему в программе возникает та или иная ошибка и устранить ее причины, а процесс установления самого факта наличия дефектов и расхождения между истинными свойствами программного изделия и его спецификациями. Нельзя сказать, что испытания программного изделия гарантируют обеспечение его качества. Обеспечение качества программного изделия включает, помимо испытаний, еще целый ряд других процедур (анализ эксплутационных характеристик, использование «стандартных» методов проектирования и программирования, восстанавливаемость после отказа, простота сопровождения, повторяемость результатов и др.) Однако испытания – важнейшая из этих процедур.

8.6.1. Современное состояние методов обеспечения качества программного изделия

Группа испытаний оказывает значительное влияние на качественную сторону проектирования, используя такие воздействия на качественную сторону проектирования, как технические ревизионные комиссии, соглашения о требованиях, спецификации и обзоры состояния проекта в различных фазах. Однако, группа испытаний не может нести ответственность за качество изделия, т.к. она не управляет процессом создания отдельных компонентов программного обеспечения. В задачи группы испытаний входят:

- проведение испытаний;
- выработка оценок;

участие в фазовых обзорах с целью влияния на ход разработок.

Независимо от задач, решаемых группой испытаний в процессе создания программного изделия, особое значение придается характеру самих тестовых задач, включая программу испытаний. В настоящее время все шире применяется технология аттестации программного изделия независимыми организациями или на основе конкретных стандартов испытаний. Широкое применение нашли стандартные тесты, примерами которых могут быть тесты национального бюро стандартов США. Важную роль в совершенствовании «стандартных» те-

стов сыграли методы структурного проектирования и программирования.

8.6.1.1. Виды испытаний программного изделия. Стадии испытаний

В общем случае, испытания проводятся в несколько стадий, разделенных по времени.

К первой стадии относятся испытания класса А, которые проводятся в конце фазы программирования после того, как будут отлажены и включены в систему все модули изделия. Этот процесс сопровождается системной отладкой, когда исправляются ошибки сопряжения модулей.

Ко второй стадии относятся испытания класса В, когда осуществляется независимая (от группы разработки) проверка компонент законченного изделия, как отдельно, так и во взаимодействии друг с другом. В идеальном случае, испытания класса В начинаются после того, как разработчики объявляют, что изделие готово к передаче потребителю. В ходе испытаний класса В функционирование проверяется на соответствие требованиям, спецификациям, документации и цели.

Испытания класса С осуществляются после того, как группа испытаний рекомендует выпуск изделия и его распространение. Испытания класса С похожи на выборочный контроль производства, поскольку с полки случайным образом выбирают экземпляр программного изделия и выполняют прогон программ, бегло анализируя результаты.

8.6.1.2. Режимы испытаний программ

Испытания различаются в зависимости от того, кто их проводит. Основная идея – независимость функции испытаний от функции разработки.

Режим I подразумевает полный цикл деятельности группы испытаний, включая планирование испытаний, разработку тестов, их прогон и анализ результатов. Обычно эта процедура является высшей и наиболее строгой формой контроля и используется для проверки универсальных программных изделий.

Режим II позволяет проводить ускоренные испытания изделия, поскольку в этом случае группа испытаний несет ответственность только за анализ результатов испытаний, а составление плана и спецификаций испытаний, построение тестов и их прогон поручается разработчикам.

Режим III реализуется без участия группы испытаний. Этот режим используется лишь в случаях крайней необходимости, например,

при сильном нарушении сроков проектирования опытного образца, когда независимые испытания изделия или, по крайней мере, независимый контроль за испытаниями исключаются. Для гарантии успеха в этом неблагоприятном случае следует предусмотреть ввод в действие и поддержку такого изделия группой разработки. При этом качество программного изделия весьма сомнительно.

8.6.1.3. Категории испытания программного изделия

Стадии испытания указывают на *время* проведения проверок, а режимы определяют тех, *кто проводит*. Категории испытаний устанавливают *характер и назначение тестов.* Продуманное деление испытаний изделий на категории облегчает общение между различными функциональными группами и степень их участия в работе. На практике выделяют следующие категории испытаний:

- Демонстрация в действии. Во время демонстрации прогоняют специально подобранные тесты, обеспечивающие желаемый результат. Тесты обычно подбираются и выполняются в рамках функции разработки во время испытаний класса А, чтобы убедить руководителей всех заинтересованных функциональных групп в том, что изделие достигло определенного уровня завершенности.
- Аттестация. Аттестация призвана гарантировать способность данного программного изделия правильно обрабатывать реальные входные данные в условиях пользователя и давать верные результаты. Испытания этой категории проводятся для того, чтобы удовлетворить требования рынка сбыта и заказчика, а также для того, чтобы продемонстрировать совместимость или эксплуатационные качества изделия. Спецификация испытаний готовится группой поддержки, а аттестация проводится группой разработки по окончанию испытаний класса А.
- Полная функциональная проверка. Цель этой категории испытаний – показать, что изделие обладает всеми функциональными возможностями, указанными во внешних спецификациях, и работает правильно. Если объектом испытания является новая версия существующего изделия, проверке подвергаются как новые, так и старые функциональные возможности изделия, отдельно и во взаимодействии друг с другом. Испытания этой категории включаются в состав испытаний классов А и В.
- Проверка новых свойств. Этим испытаниям подвергаются только новые версии существующих программных систем в целях оценки их новых функциональных качеств. Проверка новых свойств обычно проводится в рамках испытаний класса А и В и выполняется в

тех случаях, когда изделие подвергается лишь незначительным изменениям.

- Эксплуатационные испытания. В результате этой проверки оцениваются эксплуатационные характеристики программного изделия, такие, как скорость выполнения операций, объем занимаемой памяти, пропускная способность, скорость пересылки данных, время транслирования, компоновки или генерации, время реакции и условия взаимодействия с пользователем. Эксплуатационные свойства оцениваются в ходе испытаний класса А и В.
- Надежностные испытания. Во время этих испытаний изделие ставится в условия, позволяющие оценить его способность к устойчивой работе или восстановлению после отказа. Обычно в ходе этих испытаний преднамеренно вносятся искусственно созданные ошибки, испытывают изделие в условиях непрерывной работы в течение нескольких часов и проверяют все восстановительные процедуры. Надежностные испытания входят в состав испытаний класса А и В.
- Проверка устойчивости. Эти испытания призваны гарантировать правильность объединения программных изделий в систему. Они должны убедить всех в том, что взаимодействие различных программных средств не создает ошибочных ситуаций. В отношении отдельных изделий фиксируется среднее время между отказами. Проверка проводится в рамках испытаний класса А и В.
- Возвратная проверка. В эту категорию испытаний входит проверка новой версии или редакции изделия, подтверждающая, что ранее замеченные дефекты исправлены, и исправления не привели к появлению новых ошибок. Возвратная проверка входит в состав испытаний класса А и В.
- Пусковые испытания. Эти испытания подтверждают, что ввод программного изделия в действие может быть осуществлен в полном соответствии с описанием, т.е. в отведенное для этого время, силами персонала, обученного соответствующим образом, с помощью технической документации и с помощью только тех средств, которые были предусмотрены в описании. Испытания проводятся на различных конфигурациях технических средств ЭВМ и обычно входят в состав испытаний классов А и В.
- Конфигурационные испытания. Эти испытания призваны гарантировать, что изделие правильно функционирует на всех конфигурациях вычислительной техники, которые были предусмотрены проектом. В процессе этих испытаний создаются минимальные базо-

вые конфигурации и имитируются максимальные. Конфигурационные испытания выполняются в рамках испытаний классов А и В.

Стадии, режимы и категории испытаний наглядно можно представить в табличной форме.

Таблица 8.7 – Перечень сводных испытаний

Проверка изделий Уровень испытаний			
Категория испытаний	Класс испытаний		
	A	B	C
Демонстрация в действии		/	/
Аттестация	Р	/	/
Полная функциональная проверка	Р	И	/
Проверка новых свойств			/
Эксплуатационные испытания	Р	И	/
Надежностные испытания			/
Проверка устойчивости			/
Возвратная проверка			/
Пусковые испытания	Р	И	О
Конфигурационные испытания	Р	И	О
Режимы испытаний I – Проводится группой испытаний (X) II – Контролируется группой испытаний ( ) III – Группа испытаний не участвует ( ) Подразделения, проводящие испытания Р – группа разработки О – группа обслуживания И – группа испытаний / – испытания исключены			

8.6.2. Организационная структура группы испытаний

Общее правило организации деятельности по обеспечению качества программного изделия заключается в установлении подотчетности соответствующих процедур как можно более высокому уровню руководства фирмы, и отделение их от функции разработки. Группа испытаний в этом смысле должна быть органом контроля деятельности других функциональных групп, особенно групп разработки и выпуска документации, и потому следует принять меры, предотвращаю-

щие причины разногласий между ними. Наиболее рациональная схема взаимодействия представлена на рис. 8.9, где группа испытаний входит в состав сектора компоновки и выпуска.

Организуя испытания программного изделия, необходимо иметь четкий ответ на вопрос: «где кончается процесс оценки и начинается процесс отладки программного изделия?» Прежде всего, следует ограничить деятельность испытателей, возложив на них только обязанность фиксировать факт наличия ошибки. Им нельзя разрешать диагностировать причины ошибок, и более того, указывать точное место их возникновения. Если не проводить в жизнь такое распределение труда, то никогда не удастся отделить задачи от обязанностей испытателей, и разработчики будут уповать на то, что группа испытаний сама завершит отладку программ.

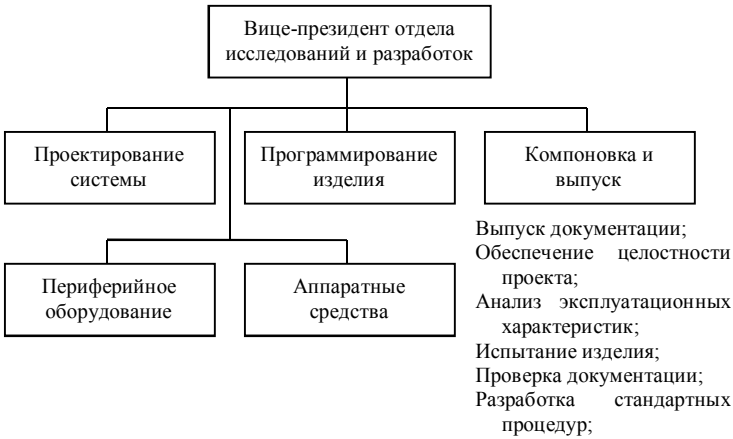


Рис. 8.9 – Организация взаимосвязи при проведении испытаний

Необходимо продумать вопрос, на кого следует возложить ответственность за компоновку программ в систему, особенно в условиях, когда в проекте участвует несколько фирм-подрядчиков. Объединение функций компоновки и испытаний уменьшает число подразделений-участников проекта и не приводит к утрате объективности оценок. При этом должно четко выполняться требование – на группе разработки лежит полная ответственность за устранение обнаруживаемых ошибок.

Типичный путь эволюции испытаний можно представить следующим образом:

- Испытания не проводятся, разработчики сами отлаживают программные средства и выпускают их для распространения.
- Пользователи присылают много замечаний, указывающих на обилие ошибок, в организации выделяется автономная группа испытаний, которая не оказывает влияние на проектирование.
- Деятельность группы испытаний становится столь активной, что руководитель группы разработки протестует, заявляя: «Как я могу уложиться в плановые сроки, если группа испытаний придирается к любой мелочи и не соглашается рекомендовать изделие к выпуску?» Тогда группа испытаний становится частью группы разработки, и их совместная деятельность продолжается успешнее, т.к. испытатели утвердили «независимый» стиль работы и не склонны допускать нарушение этой традиции.
- Руководители проекта и руководство фирмы осознают, что проблема испытаний требует профессионального подхода, и что сотрудники группы испытаний могут играть важную роль в процессах планирования и проектирования изделий. Эта группа начинает постоянно участвовать в составлении планов и спецификаций. В результате вновь возникают трения, т.к. руководитель группы разработки часто нейтрализует предложения испытателей. «Если мы все будем делать так, как требуют испытатели, мы не выполним календарный план, и нам не хватит отпущенных ресурсов». Поэтому группа испытаний выделяется из группы разработки и становится самостоятельной организационной единицей.

Независимо от места, которое занимает группа испытаний в проектной организации, следует не упускать из виду необходимость стимулирования работы ее сотрудников. Ошибочно желание укомплектовать группу испытаний стажерами или людьми, не проявившими себя как разработчики. Испытания, проведенные этими людьми, несут больше вреда, чем просто отсутствие испытаний. Будет затрачена масса времени и сил на отладку самих тестов, общение между испытателями и разработчиками будет происходить на крайне низком уровне взаимодействия, и программный продукт получит крайне плохую репутацию среди потребителей. Только из хороших проектировщиков и программистов получаются хорошие испытатели. Особенно удачно использование системных программистов и специалистов по системному прикладному анализу. Должны быть предусмотрены пути самостоятельного служебного роста для таких людей.

8.6.3. Организация испытаний в фазах исследований и анализа осуществимости

В фазе исследований деятельность группы испытаний должна предусматривать интенсивное общение с другими группами для решения вопросов о том, какие режимы испытаний следует установить, когда начнутся испытания класса В, какие новые методики или средства должны быть привлечены для этих испытаний. Необходимо провести анализ, какие ресурсы потребуются на каждой стадии испытаний, и дать оценку сроков выполнения и предполагаемым затратам.

На этой стадии составляется предварительный бюджет, обычно этот документ составляет кто-либо из сотрудников группы разработки на основании данных, поступивших от остальных групп. Роль группы испытаний на этом этапе – наиболее объективно оценить свои предполагаемые затраты (бюджет). Основной алгоритм, лежащий в этой оценке – стоимость испытаний пропорциональна стоимости разработки.

В стадии анализа осуществимости группа обработки готовит соглашение о требованиях. В этот период группа испытаний имеет первую возможность повлиять на качество изделия, предъявляя свои материалы для выработки соглашения о требованиях. В процессе поиска компромиссных решений, пересмотра и утверждения этого документа утверждаются и необходимые уровни испытаний. Группа испытаний здесь играет активную роль, побуждая разработчиков к введению количественных характеристик в требования и цели.

Соглашения о требованиях являются одновременно и планом разработки, они содержат довольно много сведений о путях реализации проекта. В этих соглашениях предусматривается специальный раздел, в котором группа испытаний описывает свой план проверок.

В этой фазе группа испытаний особое внимание уделяет тем разделам соглашения о требованиях, где речь идет о внешних условиях, в которых должно работать программное изделие, об эргономических характеристиках и действующих ограничениях.

На протяжении всей фазы анализа осуществимости группа испытаний изучает соглашение о требованиях и на основе этого анализа составляет собственные планы (план разработки спецификаций испытаний, установление начальных дат испытаний класса В и каждого цикла и др.).

8.6.4. Организация испытаний в фазах конструирования и программирования

Составление плана испытаний является основной работой, которую выполняет группа испытаний в этой фазе. Обязательным условием завершения этой работы является утверждение соглашения о требованиях, и поэтому ее окончание планируется на период фазового обзора.

При составлении плана испытаний используется принцип декомпозиции планов. План испытаний является детализацией соглашения о требованиях и содержит подробную информацию, достаточную для того, чтобы рецензенты смогли уловить степень соответствия программы испытаний класса В этому испытанию. Создавая план испытаний, группа испытаний принимает все меры к тому, чтобы спланировать все виды работ – оборудование, программное обеспечение, расходные материалы и др.

Фаза программирования начинается с момента появления внешней спецификации. Сразу после получения этого документа группа испытаний начинает анализировать его на соответствие соглашению о требованиях, сообщая группе разработки свои замечания.

После завершения работ над внешней спецификацией группа испытаний начинает готовить спецификацию испытаний. Спецификация испытаний представляет собой декомпозицию плана испытаний и подробно описывает все тесты, которые предстоит выполнить, включая описание ожидаемых результатов. Поскольку спецификация испытаний в значительной мере зависит от содержания внешних спецификаций, то она утверждается после утверждения последних.

Основную часть работы группы в этой фазе составляет разработка тестовых (контрольных) примеров. При этом все тестовые примеры должны быть приготовлены до окончания фазы программирования. Контрольные примеры становятся известны группе разработки перед самым началом фазы оценки.

Успешное проведение приемочных испытаний позволяет руководству сделать пробный выпуск изделия для передачи в руки критически настроенного заказчика. Для этого группа испытаний должна определить набор контрольных примеров, составляющих программу приемных испытаний, как можно раньше. Разработчики должны иметь возможность ознакомиться с этими примерами.

Ущерб от плохой организации работ, предшествующих испытаниям класса В, может быть сравним с затратами (временными) на программирование.

8.6.5. Организация испытаний в фазе оценки

На этапе оценки уровень трудозатрат группы испытаний достигает максимума. Первая обязанность группы состоит в том, чтобы установить готовность изделия к испытаниям класса В. Группа разработки нередко усваивает местнический взгляд на свою роль, планируя сжатые сроки работ перед началом испытаний класса В. В этой спешке может пострадать (и обычно страдает) качество изделия, поступающего в группу испытаний. Надлежащий контроль процедур передачи может быть обеспечен, если группа испытаний будет настаивать на том, чтобы приемочные испытания не признавались законченными до тех пор, пока не завершится успешный прогон всех приемочных тестов.

Если испытания класса В начались, то прогону подлежат все контрольные примеры, сначала в логически обоснованном порядке, а затем в любом произвольном. Необходимость изменения логической последовательности запуска возникает из-за того, что некоторые из тестов зависят от успешного выполнения какой-либо функции испытываемого изделия. Если эта функция выполняется неправильно, проведение соответствующих тестов откладывается до исправления дефектов изделия.

После первой передачи изделия испытателям редко удается с успехом выполнить все контрольные тесты. Когда число обнаруживаемых ошибок резко уменьшается, группа испытаний заканчивает прогон тестов и просит разработчиков после исправления дефектов предоставить изделия на повторное испытание. Этот период называется циклом испытаний.

Обнаруживая тот ли иной дефект, испытатели доказывают факт его наличия с помощью различных материалов (консольных сообщений, распечатки текущего состояния системы, дамп и др.). При этом испытатели не имеют права самостоятельно исправлять дефект, так как в этом случае заказчику может попасть программа, «заплатанная» группой испытаний, и разработчики снимут с себя всякую ответственность.

С административной точки зрения каждый дефект, обнаруженный испытателями, рассматривается как ошибка, найденная после выпуска изделия для пользования. Эти дефекты нумеруются в порядке, зависящем от степени серьезности, и фиксируются в перечне дефектов. Этот перечень отправляется в группу разработки для принятия мер по устранению дефектов. Иногда (по усмотрению группы испытаний) эти исправления могут быть внесены сразу в данном цикле испытаний. Однако это применяется крайне редко из-за увеличения трудоемкости работы группы испытаний.

Испытаниям класса В подвергаются как программные средства, так и соответствующая документация. Между этими объектами испытаний не должно быть расхождений, они вместе должны соответствовать соглашению о требованиях. Каждый цикл испытаний завершается составлением краткого отчета о результатах проверки изделия в данном цикле.

Иногда группа поддержки настойчиво требует передать изделие в ограниченное пользование (обычно это бывает тогда, когда график выпуска резко нарушается.) В этом случае выпуск называется предварительным. Предварительные выпуски сопровождаются документами, предупреждающими пользователей о возможных затруднениях. Группа испытаний дополняет эти материалы уведомлением об обнаруженных дефектах (перечень неустраненных дефектов).

На этапе окончания испытаний класса В руководство может принять решение об испытаниях в условиях пользователя (полевые испытания). Для них также готовится предварительный выпуск изделия. Группа испытаний проводит эти испытания на вычислительном центре пользователя, составляя перечень обнаруженных дефектов.

Наконец, наступает момент, когда группа испытаний рекомендует выпуск изделия (либо как-то иначе определяет его судьбу). Если группа испытаний приходит к выводу, что продолжение испытаний не приведет к принятию решения о выпуске, она собирает представителей всех групп, участвующих в фазовом обзоре V, чтобы согласовать дальнейшие действия. Во всех других случаях группа испытаний самостоятельно принимает решение о выпуске изделия (с дефектами или без).

Решение о выпуске изделия принимает руководитель группы испытаний на основе ответов сотрудников группы на следующие вопросы:

- Всеми ли имеющимися тестами испытано изделие?
- Можно ли отнести большинство неисправленных ошибок к разряду малозначительных?
- Не помогут ли дальнейшие испытания выявить серьезные ошибки?
- Уменьшались ли число и степень серьезности ошибок по мере перехода от одного цикла испытаний к другому?
- Является ли характер обнаруженных серьезных ошибок таким, что их влияние можно устранить, сделав соответствующее пояснение в информационном листке выпуска?
- Имеют ли группы разработки и выпуска документации обоснованные планы устранения всех обнаруженных ошибок в фазе использования?

Если ответы положительные, то изделия можно рекомендовать к выпуску. Санкционируя выпуск, группа испытаний подводит итоги испытаний класса В как в количественном, так и в качественном отношении, прилагая статистические данные о количестве дефектов и сведения о фактически ожидаемых эксплуатационных характеристиках – отчет испытаний класса В. Он является главным объектом обсуждения в фазовом обзоре V.

8.6.6. Организация испытаний в фазе использования

В фазе использования группа испытаний осуществляет текущий контроль за проведением испытаний класса С. Роль группы испытаний состоит в том, чтобы убедиться, что выполнение контрольных проверок компонентов изделия, подготовленного к правке, может осуществлять необученный персонал.

Ввод в действие и эксплуатация изделия пользователями позволяют выявить новые дефекты. По мере поступления сообщений о выявленных дефектах группа испытаний разрабатывает новые контрольные примеры, позволяющие проверить правильность изменений, которые были внесены группой сопровождения. Прежде чем приступить к распространению исправленной версии, группа испытаний проводит возвратную проверку, призванную гарантировать, что не только устранен прежний дефект, но не появилось новых.

8.6.7. Участие группы испытаний в фазовых обзорах

Группа испытаний участвует в пяти из шести фазовых обзорах (табл. 8.8).

Таблица 8.8 – Участие группы выпуска документации в фазовых обзорах

Фаза	Фазовый обзор	Форма участия при обсуждении документов
Исследование	I	Поверка
Анализ осуществимости	II	Проверка и обсуждение
Конструирование	III	Проверка и обсуждение
Программирование	IV	Поверка
Оценка	V	Утверждение
Использование	VI	Не участвует

В фазовом обзоре I группа испытаний дает предварительную оценку ресурсам, необходимым для обеспечения ее деятельности, и предварительным срокам их использования.

В фазовом обзоре II группа испытаний должна определить режим испытаний и одну или несколько категорий испытаний. На этом этапе группа испытаний совместно с группой разработки обсуждает соглашение о требованиях и особенно вопросы эксплуатационных характеристик, удобства внедрения и использования изделия.

В ходе фазового обзора III группа испытаний сопоставляет внешнюю спецификацию с соглашением о требованиях, т.е. она участвует в утверждении внешней спецификации, осуществляет контроль качества проекта.

В фазовом обзоре IV в центре внимания отчет об испытаниях класса А. Группа испытаний лишь рассматривает этот документ.

В ходе фазового обзора V обсуждается отчет об испытаниях класса В. Для того, чтобы начать производство программного изделия, группа испытаний должна рекомендовать его выпуск. Т.е. группа испытаний утверждает документы.

Участие группы испытаний в фазовом обзоре VI необязательно. Ее мнение мало влияет на принятие решения относительно снижения уровня поддержки изделия или ее полного прекращения.

Контрольные вопросы

1) Понятие изделия, как средства общения.
2) Нисходящий анализ процесса управления созданием программного изделия.
3) Установление целей и средства их достижения.
4) Подбор и обучение кадров.
5) Организация планирования разработки программного изделия. Виды планов. Декомпозиция планов.
6) Организационная структура группы планирования.
7) Виды планов, связанных с созданием программного изделия.
8) Организация планирования разработки программного изделия.
9) Вопросы, рассматриваемые в фазовых обзорах группой планирования,
10) Управление проектом.
11) Организация работы группы разработки в фазах создания программного изделия.
12) Организация работы группы обслуживания в фазах создания программного изделия.
13) Организация работы группы выпуска документации в фазах создания программного изделия.
14) Организация испытаний программного изделия.

Список литературы

1. Зельковиц М., Шоу А., Гэннон Дж. Принципы разработки программного обеспечения. – М.: Мир, 1982 – 368 с.
2. Гантер Р. Методы управления проектированием программного обеспечения. – М.: Мир, 1981 – 388 с.
3. Вольховер В.Т., Иванов Л.А. Производственные методы разработки программ. – М.: Финансы и статистика, 1983 – 236 с.
4. Гласс Р. Руководство по надежному программированию. – М.: Финансы и статистика, 1983 – 176 с.