Лекция 1. Понятие отладки программы

Отладка бывает двух видов:

Синтаксическая отладка. Синтаксические ошибки выявляет компилятор, поэтому исправлять их достаточно легко.

Семантическая (смысловая) отладка. Ее время наступает тогда, когда синтаксических ошибок не осталось, но результаты программа выдает неверные. Здесь компилятор сам ничего выявить не сможет, хотя в среде программирования обычно существуют вспомогательные средства отладки, о которых мы еще поговорим.

Отладка — это процесс локализации и исправления ошибок в программе.

Как бы тщательно мы ни писали, отладка почти всегда занимает больше времени, чем программирование.

Локализация ошибок

Локализация — это нахождение места ошибки в программе.

В процессе поиска ошибки мы обычно выполняем одни и те же действия:

- прогоняем программу и получаем результаты;
- сверяем результаты с эталонными и анализируем несоответствие;
- выявляем наличие ошибки, выдвигаем гипотезу о ее характере и месте в программе;
- проверяем текст программы, исправляем ошибку, если мы нашли ее правильно.

Способы обнаружения ошибки:

Аналитический - имея достаточное представление о структуре программы, просматриваем ее текст вручную, без прогона.

Экспериментальный - прогоняем программу, используя отладочную печать и средства трассировки, и анализируем результаты ее работы.

Оба способа по-своему удобны и обычно используются совместно.

Принципы отладки

Принципы локализации ошибок:

Большинство ошибок обнаруживается вообще без запуска программы - просто внимательным просматриванием текста.

Если отладка зашла в тупик и обнаружить ошибку не удается, лучше отложить программу. Когда глаз "замылен", эффективность работы упорно стремится к нулю. Чрезвычайно удобные вспомогательные средства - это отладочные механизмы среды разработки: *трассировка*, *промежуточный контроль значений*. Можно использовать даже дамп памяти, но такие радикальные действия нужны крайне редко. Экспериментирования типа "а что будет, если изменить плюс на минус" - нужно избегать всеми силами. Обычно это не дает результатов, а только больше запутывает процесс отладки, да еще и добавляет новые ошибки.

Принципы исправления ошибок еще больше похожи на законы Мерфи:

1. Там, где найдена одна ошибка, возможно, есть и другие.
2. Вероятность, что ошибка найдена правильно, никогда не равна ста процентам.
3. Наша задача - найти саму ошибку, а не ее симптом.

Это утверждение хочется пояснить. Если программа упорно выдает результат 0,1 вместо эталонного нуля, простым округлением вопрос не решить. Если результат получается отрицательным вместо эталонного положительного, бесполезно брать его по модулю - мы получим вместо решения задачи ерунду с подгонкой. Исправляя одну ошибку, очень легко внести в программу еще парочку. "Наведенные" ошибки - настоящий бич отладки.
Исправление ошибок зачастую вынуждает нас возвращаться на этап составления программы. Это неприятно, но порой неизбежно.

Методы отладки

Силовые методы

- Использование дампа (распечатки) памяти. Это интересно с познавательной точки зрения: можно досконально разобраться в машинных процессах. Иногда такой подход даже необходим - например, когда речь идет о выделении и высвобождении памяти под динамические переменные с использованием недокументированных возможностей языка. Однако, в большинстве случаев мы получаем огромное количество низкоуровневой информации, разбираться с которой - не пожелаешь и врагу, а результативность поиска - исчезающе низка.

- Использование отладочной печати в тексте программы - произвольно и в большом количестве. Получать информацию о выполнении каждого оператора тоже небезынтересно. Но здесь мы снова сталкиваемся со слишком большими объемами информации. Кроме того, мы здорово захламляем

программу добавочными операторами, получая малочитабельный текст, да еще рискуем внести десяток новых ошибок.

- Использование автоматических средств отладки - трассировки с отслеживанием промежуточных значений переменых. Это самый распространенный способ отладки. Не нужно только забывать, что это только один из способов, и применять всегда и везде только его - часто невыгодно. Сложности возникают, когда приходится отслеживать слишком большие структуры данных или огромное их число. Еще проблематичнее трассировать проект, где выполнение каждой подпрограммы приводит к вызову пары десятков других. Но для небольших программ трассировки вполне достаточно.

С точки зрения "правильного" программирования силовые методы плохи тем, что не поощряют анализ задачи.

Суммируя свойства силовых методов, получаем практические советы:
- использовать трассировку и отслеживание значений переменных для небольших проектов, отдельных подпрограмм;
- использовать отладочную печать в небольших количества и "по делу";
- оставить дамп памяти на самый крайний случай.

Метод индукции - анализ программы от частного к общему.
Просматриваем симптомы ошибки и определяем данные, которые имеют к ней хоть какое-то отношение. Затем, используя тесты, исключаем маловероятные гипотезы, пока не остается одна, которую мы пытаемся уточнить и доказать.
Метод дедукции - от общего к частному.
Выдвигаем гипотезу, которая может объяснить ошибку, пусть и не полностью. Затем при помощи тестов эта гипотеза проверяется и доказывается.

Обратное движение по алгоритму. Отладка начинается там, где впервые встретился неправильный результат. Затем работа программы прослеживается (мысленно или при помощи тестов) в обратном порядке, пока не будет обнаружено место возможной ошибки.
Метод тестирования.

Давайте рассмотрим процесс локализации ошибки на конкретном примере. Пусть дана небольшая программа, которая выдает значение максимального из трех введенных пользователем чисел.

```
var
a, b, c: real;
begin
```

```
writeln('Программа находит значение максимального из трех введенных
чисел');
write('Введите первое число '); readln(a);
write('Введите второе число '); readln(b);
write('Введите третье число '); readln(c);
if (a>b)and(a>c) then
writeln('Наибольшим оказалось первое число ',a:8:2)
else if (b>a)and(a>c) then
writeln('Наибольшим оказалось второе число ',b:8:2)
else
writeln('Наибольшим оказалось третье число ',b:8:2);
end.
```

Обе выделенные ошибки можно обнаружить невооруженным глазом: первая явно допущена по невнимательности, вторая - из-за того, что скопированную строку не исправили.

Тестовые наборы данных должны учитывать все варианты решения, поэтому выберем следующие наборы чисел:

Данные. Ожидаемый результат
a=10; b=-4; c=1 max=a=10
a=-2; b=8; c=4 max=b=8
a=90; b=0; c=90.4 max=c=90.4

В результате выполнения программы мы, однако, получим следующие результаты:
Для a=10; b=-4; c=1:

Наибольшим оказалось первое число 10 .00

Для a=-2; b=8; c=4: Наибольшим оказалось третье число 8.00Для a=90; b=0; c=90.4:

Наибольшим оказалось третье число 0.00

Вывод во втором и третьем случаях явно неверен. Будем разбираться.

1. Трассировка и промежуточная наблюдение за переменными

Добавляем промежуточную печать или наблюдение за переменными:

- вывод a, b, c после ввода (проверяем, правильно ли получили данные)

- вывод значения каждого из условий (проверяем, правильно ли записали условия)

Листинг программы существенно увеличился и стал вот таким:

```
var
a, b, c: real;
begin
writeln('Программа находит значение максимального из трех введенных
чисел');
```

```
write('Введите первое число '); readln(a);
writeln('Вы ввели число ',a:8:2); {отл.печать}
write('Введите второе число '); readln(b);
writeln('Вы ввели число ',b:8:2); {отл.печать}
write('Введите третье число '); readln(c);
writeln('Вы ввели число ',c:8:2); {отл.печать}
writeln('a>b=',a>b,', a>c=',a>c,', (a>b)and(a>c)=',(a>b)and(a>c));
{отл.печать}
if (a>b)and(a>c) then
writeln('Наибольшим оказалось первое число ',a:8:2)
else begin
writeln('b>a=',b>a,', b>c=',b>c,', (b>a)and(b>c)=',(b>a)and(b>c));
{отл.печать}
if (b>a)and(a>c) then
writeln('Наибольшим оказалось второе число ',b:8:2)
else
writeln('Наибольшим оказалось третье число ',b:8:2);
end;
end.
```

В принципе, еще при наборе у нас неплохой шанс отловить ошибку в условии: подобные кусочки кода обычно не перебиваются, а копируются, и если дать себе труд слегка при этом задуматься, ошибку найти легко.

Но давайте считать, что глаз "замылен" совершенно, и найти ошибку не удалось.

Вывод для второго случая получается следующим:

Программа находит значение максимального из трех введенных чисел
Введите первое число -2
Вы ввели число -2.00
Введите второе число 8
Вы ввели число 8.00
Введите третье число 4
Вы ввели число 4.00
a>b=FALSE, a>c=FALSE, (a>b)and(a>c)=FALSE
b>a=TRUE, b>c=TRUE, (b>a)and(b>c)=TRUE
Наибольшим оказалось третье число 8.00

Со вводом все в порядке. Впрочем, в этом сомнений и так было немного. А вот что касается второй группы операторов печати, то картина вышла интересная: в результате выводится верное число (8.00), но неправильное слово ("третье", а не "второе").

Вероятно, проблемы в выводе результатов. Тщательно проверяем текст и обнаруживаем, что действительно в последнем случае выводится не c, а b. Однако к решению текущей проблемы это не относится: исправив ошибку, мы получаем для чисел -2.0, 8.0, 4.0 следующий результат.

Наибольшим оказалось третье число 4.00

Теперь ошибка локализована до расчетного блока и, после некоторых усилий, мы ее находим и исправляем.

Метод индукции

Судя по результатам, ошибка возникает, когда максимальное число - второе или третье (если максимальное - первое, то определяется оно правильно, для доказательства можно програть еще два-три теста).

Просматриваем все, относящееся к переменным b и c. Со вводом никаких проблем не замечено, а что касается вывода - то мы быстро натыкаемся на замену b на c. Исправляем.

Как видно, невыявленные ошибки в программе остаются. Просматриваем расчетный блок: все, что относится к максимальному b (максимум c получается "в противном случае"), и обнаруживаем пресловутую проблему "a>c" вместо "b>c". Программа отлажена.

Метод дедукции

Неверные результаты в нашем случае могут получиться из-за ошибки в:

- вводе данных;

- расчетном блоке;

- собственно выводе.

Для доказательства мы можем пользоваться отладочной печатью, трассировкой или просто набором тестов. В любом случае мы выявляем одну ошибку в расчете и одну в выводе.

Обратное движение по алгоритму

Зная, что ошибка возникает при выводе результатов, рассматриваем код, начиная с операторов вывода. Сразу же находим лишнюю b в операторе writeln.

Далее, смотрим по конкретной ветке условного оператора, откуда взялся результат. Для значений -2.0, 8.0, 4.0 расчет идет по ветке с условием if (b>a)and(a>c) then... где мы тут же обнаруживаем искомую ошибку.

5. Тестирование

В нашей задаче для самого полного набора данных нужно выбрать такие переменные, что
a > b > c
a > c > b

b > a > c
b > c > a
c > a > b
c > b > a

Анализируя получившиеся в каждом из этих случаев результаты, мы приходим к тому, что проблемы возникают при b>c>a и c - максимальном. Зная эти подробности, мы можем заострить внимание на конкретных участках программы.

Конечно, в реальной работе мы не расписываем так занудно каждый шаг, не прибегаем исключительно к одной методике, да и вообще частенько не задумываемся, каким образом искать ляпы. Теперь, когда мы разобрались со всеми подходами, каждый волен выбрать те из них, которые кажутся самыми удобными.

Средства отладки

Помимо методик, хорошо бы иметь представление о средствах, которые помогают нам выявлять ошибки. Это:

1) Аварийная печать - вывод сообщений о ненормальном завершении отдельных блоков и всей программы в целом.

2) Печать в узлах программы - вывод промежуточных значений параметров в местах, выбранных программистом. Обычно, это критичные участки алгоритма (например, значение, от которого зависит дальнейший ход выполнения) или составные части сложных формул (отдельно просчитать и вывести числитель и знаменатель большой дроби).

3) Непосредственное слежение:

- арифметическое (за тем, чему равны, когда и как изменяются выбранные переменные),

- логическое (когда и как выполняется выбранная последовательность операторов),

- контроль выхода индексов за допустимые пределы,

- отслеживание обращений к переменным,

- отслеживание обращений к подпрограммам,

- проверка значений индексов элементов массивов и т.д.

Нынешние среды разработки часто предлагают нам реагировать на возникающую проблему в диалоговом режиме. При этом можно:

- просмотреть текущие значения переменных, состояние памяти, участок алгоритма, где произошел сбой;

- прервать выполнение программы;

- внести в программу изменения и повторно запустить ее (в компиляторных средах для этого потребуется перекомпилировать код, в интерпретаторных выполнение можно продолжить прямо с измененного оператора).

Классификация ошибок

Ошибки в программах могут допускаться от самого начального этапа составления алгоритма решения задачи до окончательного оформления программы. Разновидностей ошибок достаточно много. Рассмотрим некоторые группы ошибок и соответствующие примеры:

Вид ошибки	*Пример*
Неправильная постановка задачи	Правильное решение неверно сформулированной задачи
Неверный алгоритм	Выбор алгоритма, который привел к неточному или неэффективному решению задачи
Ошибка анализа	Неполный перечень ситуаций, которые могут возникнуть при решении задачи; наличие логических ошибок
Семантические ошибки	Неправильное понимание порядка выполнения оператора
Синтаксические ошибки	Нарушение правил, которые определяются выбранным языком программирования
Ошибки при выполнении операций	Использование слишком большого числа, деления на ноль, извлечения квадратного корня из отрицательного числа и т. п.
Ошибки в данных	Неправильно определен возможный диапазон изменения данных
Опечатки	Неправильное использование схожих по внешнему виду символов, например, вместо цифры *1* используется буква *I, l*
Ошибки ввода-вывода	Неправильное считывание входных данных, неправильное задание форматов данных

Ошибки могут относиться к самым разным частям кода:

- ошибки обращения к данным,

- ошибки описания данных,

- ошибки вычислений,

- ошибки при сравнении,

- ошибки в передаче управления,

- ошибки ввода-вывода,

- ошибки интерфейса,

и т д

Классификация ошибок по этапу обработки программы

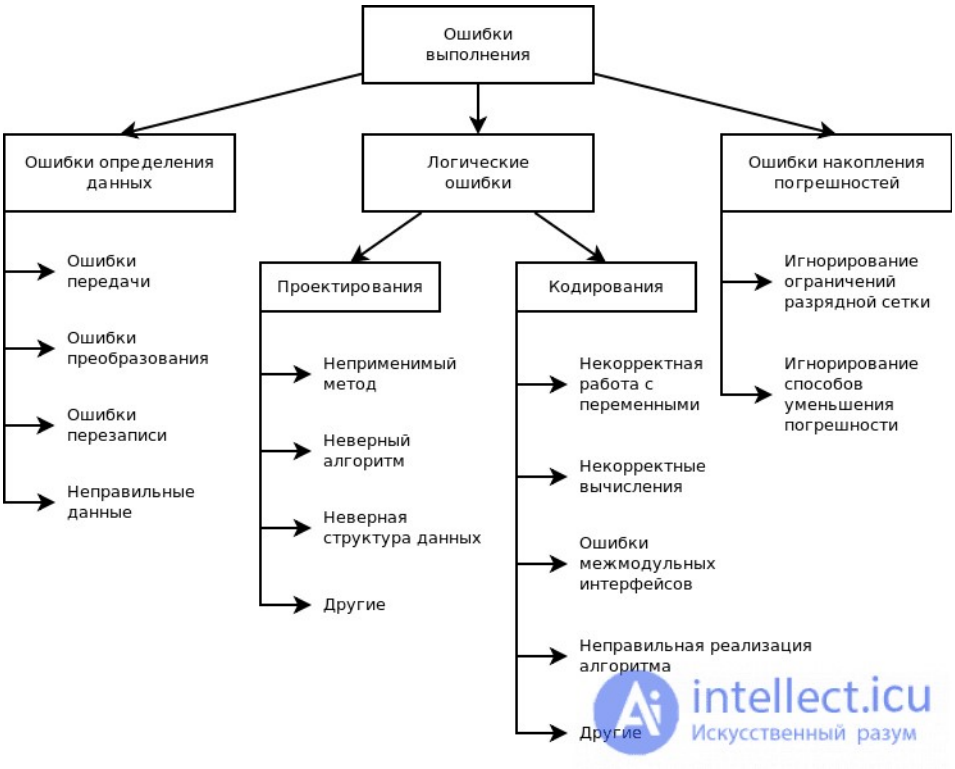

Рис. Классификация ошибок этапа выполнения по возможным причинам

Синтаксические ошибки

Синтаксические ошибки зачастую выявляют уже на этапе трансляции. К сожалению, многие ошибки других видов транслятор выявить не в силах, т.к.

ему не известен задуманный или требуемый результат работы программы. Отсутствие сообщений транслятора о наличии синтаксических ошибок является необходимым условием правильности программы, но не может свидетельствовать о том, что она даст правильный результат.

Примеры синтаксических ошибок:

1. отсутствие знака пунктуации;
2. несоответствие количества открывающих и закрывающих скобок;
3. неправильно сформированный оператор;
4. неправильная запись имени переменной;
5. ошибка в написании служебных слов;
6. отсутствие условия окончания цикла;
7. отсутствие описания массивов и т.п.

Ошибки, которые не обнаруживает транслятор

В случае правильного написания операторов в программе может присутствовать большое количество ошибок, которые транслятор не может обнаружить. Рассмотрим примеры таких ошибок:

Логические ошибки: после проверки заданного условия неправильно указана ветвь алгоритма; неполный перечень возможных условий при решении задачи; один или более блоков алгоритма в программе пропущен.

Ошибки в циклах: неправильно указано начало цикла; неправильно указаны условия окончания цикла; неправильно указано количество повторений цикла; использование бесконечного цикла.

Ошибки ввода-вывода; ошибки при работе с данными: неправильно задан тип данных; организовано считывание меньшего или большего объема данных, чем нужно; неправильно отредактированы данные.

Ошибки в использовании переменных: используются переменных, для которых не указаны начальные значения; ошибочно указана одна переменная вместо другой. Ошибки при работе с массивами: пропущено предварительное обнуление массивов; неправильное описание массивов; индексы массивов следуют в ошибочном порядке.

ошибки безопасности, умышленные и не умышленные уязвимости в системе, открытость к отказам в обслуживании. несанкционированном доступе. екхолы

Ошибки в арифметических операциях: неправильное использование типа переменной (например, для сохранения результата деления используется целочисленная переменная); неправильно определен порядок действий; выполняется деление на нуль; при расчете выполняется попытка извлечения

квадратного корня из отрицательного числа; не учитываются значащие разряды числа.

ошибки в архитектуре приложения, приводящие к увеличение технического долга

Методы (пути) снижение ошибок в программировании:

1. использование тестирования
2. использование более простых решений
3. использование систем с наименьшим числом составлящих
4. использование ранее использованных и проверенных компонентов
5. использование более квалифицрованных специалистов

Советы отладчику

1) Проверяйте тщательнее: ошибка скорее всего находится не в том месте, в котором кажется.

2) Часто оказывается легче выделить те места программы, ошибок в которых нет, а затем уже искать в остальных.

3) Тщательнее следить за объявлениями констант, типов и переменных, входными данными.

4) При последовательной разработке приходится особенно аккуратно писать драйверы и заглушки - они сами могут быть источником ошибок.

5) Анализировать код, начиная с самых простых вариантов. Чаще всего встречаются ошибки:
- значения входных аргументов принимаются не в том порядке,
- переменная не проинициализирована,
- при повторном прохождении модуля, перемен ная повторно не инициализируется,
- вместо предполагаемого полного копирования структуры данных, копируется только верхний уровень (например, вместо создания новой динамической переменной и присваивания ей нужного значения, адрес тупо копируется из уже существующей переменной),
- скобки в сложном выражении расставлены неправильно.

6) При упорной длительной отладке глаз "замыливается". Хороший прием - обратиться за помощью к другому лицу, чтобы не повторять ошибочных рассуждений. Правда, частенько остается проблемой убедить это другое лицо помочь вам.

7) Ошибка, скорее всего окажется вашей и будет находиться в тексте программы. Гораздо реже она оказывается:

в компиляторе,

операционной системе,

аппаратной части,

электропроводке в здании и т.д.

Но если вы совершенно уверены, что в программе ошибок нет, просмотрите стандартные модули, к которым она обращается, выясните, не менялась ли версия среды разработки, в конце концов, просто перегрузите компьютер - некоторые проблемы (особенно в DOS-средах, запускаемых из-под Windows) возникают из-за некорректной работы с памятью.

8) Убедитесь, что исходный текст программы соответствует скомпилированному объектному коду (текст может быть изменен, а запускаемый модуль, который вы тестируете - скомпилирован еще из старого варианта).

9) Навязчивый поиск одной ошибки почти всегда непродуктивен. Не получается - отложите задачу, возьмитесь за написание следующего модуля, на худой конец займитесь документированием.

10) Старайтесь не жалеть времени, чтобы уясненить причину ошибки. Это поможет вам:
исправить программу,
обнаружить другие ошибки того же типа,
не делать их в дальнейшем.

11) Если вы уже знаете симптомы ошибки, иногда полезно не исправлять ее сразу, а на фоне известного поведения программы поискать другие ляпы.

12) Самые труднообнаруживаемые ошибки - наведенные, то есть те, что были внесены в код при исправлении других.

Тестирование

Тестирование — это выполнение программы для набора проверочных входных значений и сравнение полученных результатов с ожидаемыми.

Цель тестирования - проверка и доказательство правильности работы программы. В противном случае - выявление того, что в ней есть ошибки.

Тестирование само не показывает местонахождение ошибки и не указывает на ее причины.

Принципы тестирования.

1) Тест - просчитанный вручную пример выполнения программы от исходных данных до ожидаемых результатов расчета. Эти результаты считаются эталонными. Полномаршрутным будет такое тестирование, при котором каждый линейный участок программы будет пройден хотя бы при выполнении одного теста.

2) При прогоне программы по тестовым начальным данным, полученные результаты нужно сверить с эталонными и проанализировать разницу, если она есть.

3) При разработке тестов нужно учитывать не только правильные, но и неверные исходные данные.

4) Мы должны проверить программу на нежелательные побочные эффекты при задании некоторых исходных данных (деление на ноль, попытка считывания из несуществующего файла и т.д.).

5) Тестирование нужно планировать: заранее выбрать, что мы контролируем и как это сделать лучше. Обычно тесты планируются на этапе алгоритмизации или выбора численного метода решения. Причем, составляя тесты, мы предполагаем, что ошибки в программе есть.

6) Чем больше ошибок в коде мы уже нашли, тем больше вероятность, что мы обнаружим еще не найденные.
Хорошим называют тест, который с большой вероятностью должен обнаруживать ошибки, а удачным - тот, который их обнаружил.

Проектирование тестов

Тесты просчитываются вручную, значит, они должны быть достаточно просты для этого.
Тесты должны проверять каждую ветку алгоритма. По возможности, конечно. Так что количество и сложность тестов зависит от сложности программы.
Тесты составляются до кодирования и отладки: во время разработки алгоритма или даже составления математической модели.
Обычно для экономии времени сначала пропускают более простые тесты, а затем более сложные.

Задачу: нужно проверить, попадает ли введенное число в заданный пользователем диапазон.

```
                              program Example;
(*****************************************************
* Задача: проверить, попадает ли введенное число в *
* заданный пользователем диапазон *
*****************************************************)

var
min, max, A, tmp: real;
begin
writeln('Программа проверяет, попадают ли введенные пользователем');
writeln('значения в заданный диапазон');
writeln;
writeln('Введите нижнюю границу диапазона '); readln(min);
writeln('Введите верхнюю границу диапазона '); readln(max);
if min>max then begin
writeln('Вы перепутали диапазоны, и я их поменяю');
tmp:=min;
min:=max;
max:=tmp;
end;
repeat
writeln('Введите число для проверки (0 - конец работы) '); readln(A);
if (A>=min)and(A<=max) then
writeln('Число ',A,' попадает в диапазон [',min,'..',max,']')
else
writeln('Число ',A,' не попадает в диапазон [',min,'..',max,']');
until A=0;
writeln;
end.
```

Если исходить из алгоритма программы, мы должны составить следующие тесты:
ввод границ диапазона
- min< max
- min>max
ввод числа
- A < min (A<>0)
- A > max (A<>0)
- min <= A <= max (A<>0)
- A=0

Как видите, программа очень мала, а тестов для проверки всех ветвей ее алгоритма, требуется довольно много.

Стратегии тестирования

1) Тестирование программы как "черного ящика".

Мы знаем только о том, что делает программа, но даже не задумываемся о ее внутренней структуре. Задаем набор входных данных, получаем результаты, сверяем с эталонными.

При этом обнаружить все ошибки мы можем только если составили тесты для всех возможных наборов данных. Естественно, это противоречит экономическим принципам, да и просто достаточно глупо.

"Черным ящиком" удобно тестировать небольшие подпрограммы.

2) Тестирование программы как "белого ящика".

Здесь перед составлением теста мы изучаем логику программы, ее внутреннюю структуру. Тестирование будет считаться удачным, если проверяет программу по всем направлениям. Однако, как мы уже говорили, это требует огромного количества тестов.

На практике мы, как всегда, совместно используем оба принципа.

3) Тестирование программ модульной структуры.

Мы снова возвращаемся к вопросу о структурном программировании. Если вы помните, программы строятся из модулей не в последнюю очередь для того, чтобы их легко было отлаживать и тестировать. Действительно, структурированную программу мы будем тестировать частями. При этом нам нужно:
строить набор тестов;
комбинировать модули для тестирования.

Такое комбинирование может строиться двумя способами: Пошаговое тестирование - тестируем каждый модуль, присоединяя его к уже оттестированным. При этом можем соединять части программы сверху вниз (нисходящий способ) или снизу вверх (восходящий). Монолитное тестирование - каждый модуль тестируется отдельно, а затем из них формируется готовая рабочая программа и тестируется уже целиком.

Чтобы протестировать отдельный модуль, нужен модуль-драйвер (всегда один) и модули-заглушки (этих может быть несколько). Модуль-драйвер содержит фиксированные исходные данные. Он вызывает тестируемый модуль и отображает (а возможно, и анализирует) результаты. Модуль-заглушка нужен, если в тестируемом модуле есть вызовы других. Вместо этого вызова управление передается модулю-заглушке, и уже он имитирует необходимые действия.

К сожалению, мы опять сталкиваемся с тем, что драйверы и заглушки сами могут оказаться источником ошибок. Поэтому создаваться они должны с большой осторожностью.