ПРИЧИНЫ И ТИПЫ ОШИБОК

Классификация ошибок по причине возникновения

- синтаксические ошибки;
- семантические ошибки;
- логические ошибки.

Синтаксические ошибки

это ошибки, возникающие в связи с нарушением синтаксических правил написания предложений используемого языка программирования (к таким ошибкам относятся пропущенные точки с запятой, ссылки на неописанные переменные, присваивание переменной значений неверного типа и т. д.).

Семантические ошибки

- Причина возникновения ошибок данного типа связана с нарушением семантических правил написания программ (примером являются ситуации попытки открыть несуществующий файл или выполнить деление на нуль).

Логические ошибки

- связаны с неправильным применением тех или иных алгоритмических конструкций.
- Эти ошибки при выполнении программы могут проявиться явно (выдано сообщение об ошибке, нет результата или выдан неверный результат, программа "зацикливается"), но чаще они проявляют себя только при определенных сочетаниях параметров или вообще не вызывают нарушения работы программы, которая в этом случае выдает правдоподобные, но неверные результаты.

Классификация ошибок по этапу обработки программы

Ошибки, которые могут быть в программе, принято делить на три группы:

- ошибки компиляции;
- ошибки компоновки;
- ошибки выполнения.

Ошибки компиляции

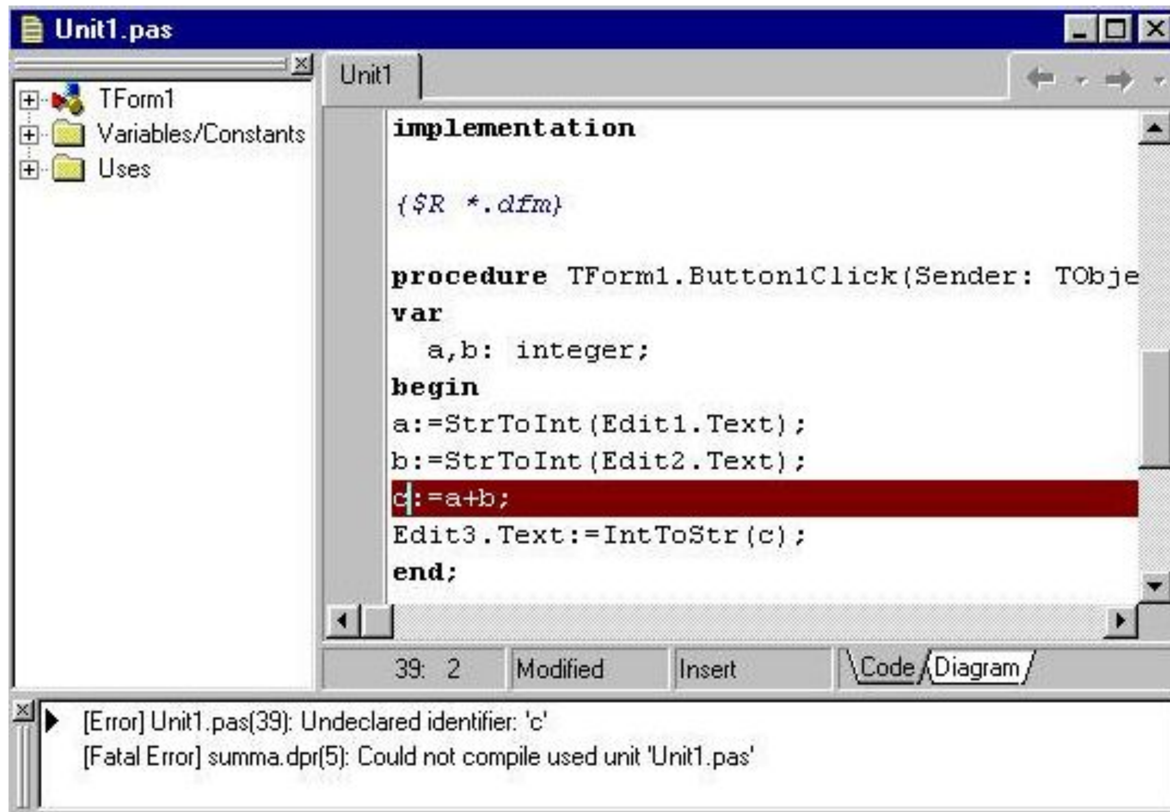
Ошибки компиляции (Compile-time error) – ошибки, фиксируемые компилятором (транслятором, интерпретатором) при выполнении синтаксического и частично семантического анализа программы;

Наиболее легко устранимы.

Их обнаруживает компилятор, а программисту остается только внести изменения в текст программы и выполнить повторную компиляцию.

Компилятор просматривает программу от начала. Если обнаруживается ошибка, то процесс компиляции приостанавливается и в окне редактора кода выделяется строка, которая, по мнению компилятора, содержит ошибочную конструкцию.

Ошибки компиляции



В нижнюю часть окна редактора кода компилятор выводит сообщения об ошибках. Первая ошибка – это первая от начала текста программы синтаксическая ошибка, обнаруженная компилятором. Наличие в тексте даже одной синтаксической ошибки приводит к возникновению второй, фатальной ошибки (Fatal Error) – невозможности генерации исполняемой программы.

Наиболее типичные ошибки компиляции

Сообщения компилятора	Вероятная причина
`Undeclared identifier` (Необъявленный идентификатор)	Используется переменная, не объявленная в разделе `var` программы; Ошибка при написании имени переменной; Ошибка при написании имени инструкции (оператора).
`Unterminated string` (Незавершенная строка)	При записи строковой константы не поставлена завершающая кавычка.
`Incompaible types … and …` (Несовместимые типы)	В операторе присваивания тип выражения не соответствует или не может быть приведен к типу переменной, получающей значение выражения.
`Missing operator or semicolon` (Отсутствует оператор или точка с запятой)	Не поставлена точка с запятой после инструкции программы.

Ошибки компоновки

Ошибки компоновки – ошибки, обнаруженные компоновщиком (редактором связей) при объединении модулей программы.

Эти ошибки связаны с проблемами, обнаруженными при разрешении внешних ссылок. Например, предусмотрено обращение к подпрограмме другого модуля, а при объединении модулей данная подпрограмма не найдена или не стыкуются списки параметров.

В большинстве случаев ошибки такого рода также удается быстро локализовать и устранить.

Ошибки выполнения

Ошибки выполнения – ошибки, обнаруженные операционной системой, аппаратными средствами или пользователем при выполнении программы.

Могут иметь разную природу, и соответственно по-разному проявляться.

Часть ошибок обнаруживается и документируется операционной системой.

Ошибки выполнения

Выделяют четыре **способа проявления** таких ошибок:

- появление сообщения об ошибке, зафиксированной схемами контроля выполнения машинных команд, например, переполнении разрядной сетки, нарушении адресации и т.п.;
- появление сообщения об ошибке, обнаруженной операционной системой, например, нарушении защиты памяти, попытке записи на устройства, защищенные от записи, отсутствии файла с заданным именем и т.п.;
- «зависание» компьютера, как простое, когда удается завершить программу без перезагрузки операционной системы, так и «тяжелое», когда для продолжения работы необходима перезагрузка;
- несовпадение полученных результатов с ожидаемыми.

Причины ошибок выполнения

Все возможные причины ошибок можно разделить на следующие группы:

- неверное определение исходных данных,
- логические ошибки,
- накопление погрешностей результатов вычислений.

Причины ошибок выполнения

Предотвращение и обработка исключений

- При разработке проекта программист должен предусмотреть все возможные варианты некорректных действий пользователя, которые могут привести к возникновению ошибок времени выполнения, и обеспечить способы защиты от них.

Предотвращение и обработка исключений

Инструкция обработки исключения в общем виде:

try // *инструкции, выполнение которых может вызвать исключение*
except // *начало секции обработки исключений*
on *ТипИсключения1* **do** *Обработка1;*
on *ТипИсключения2* **do** *Обработка2;*
…;
else // *инструкции обработки остальных исключений*
end;

Предотвращение и обработка исключений

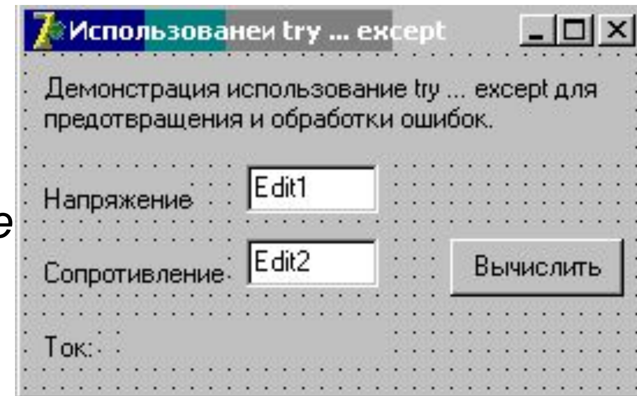
где:

- **try** — ключевое слово, обозначающее, что далее следуют инструкции, при выполнении которых возможно возникновение исключений, и что обработку этих исключений берет на себя программа;
- **except** — ключевое слово, обозначающее начало секции обработки исключений. Инструкции этой секции будут выполнены, если в программе возникнет ошибка;
- **on** — ключевое слово, за которым следует тип исключения, обработку которого выполняет инструкция, следующая за **do**;
- **else** — ключевое слово, за которым следуют инструкции, обеспечивающие обработку исключений,

Типичные исключения

Тип исключения	**Возникает**
EZeroDivide	При выполнении операции деления, если делитель равен нулю
EConvertError	При выполнении преобразования, если преобразуемая величина не может быть приведена к требуемому виду. Наиболее часто возникает при преобразовании строки символов в число
EFilerError	При обращении к файлу. Наиболее частой причиной является отсутствие требуемого файла или, в случае использования сменного диска, отсутствие диска в накопителе

Пример: Обработка исключения типа **EZeroDivide**




```
procedure TForm1.Button1Click(Sender: TObject);
Var u, r, i: real; // напряжение , сопротивление, ток
begin
Labels.Caption := ' ';
try // инструкции, которые могут вызвать исключе
u := StrToFloat(Edit1.Text);
r := StrToFloat(Edit2.Text);
i := u/r;
except // секция обработки исключений
onEZeroDivide do // деление на ноль
begin
ShowMessage('Сопротивление не может быть равно нулю!');
exit;
end;
on EConvertError do // ошибка преобразования строки в число
begin
ShowMessage('Напряжение и сопротивление должны быть заданы числом. ' );
exit;
end; end;
```

ОТЛАДКА И ТЕСТИРОВАНИЕ

Немного истории

Долгое время было принято считать, что целью тестирования является доказательство отсутствия ошибок в программе.

Но **полный перебор** всех возможных вариантов выполнения программы находится за пределами вычислительных возможностей даже для очень небольших программ.

"Тестирование – это процесс выполнения программ с целью обнаружения ошибок".

Гленфорд Майерс

Майерс, Г. Искусство тестирования программ, 1982

Немного истории

До начала 80-х годов процесс тестирования программного обеспечения (ПО) был разделен с процессом разработки: вначале программисты реализовывали заданную функциональность, а затем тестировщики приступали к проверке качества созданных программ.

Проблемы:

- разработка программ может оказаться достаточно длительной – чем в это время должны заниматься тестировщики?
- Плохая предсказуемости результатов такого процесса разработки. Ключевой вопрос: сколько времени потребуется на завершение продукта, в котором существует 500 известных ошибок?

Немного истории

Статистика:

Даже однострочное изменение в программе с вероятностью 55 % либо не исправляет старую ошибку, либо вносит новую. Если же учитывать изменения любого объема, то в среднем **менее 20 % изменений корректны с первого раза**.

Немного истории

В 90-х годах появилась другая методика разработки (**zero-defect mindset)**, основная идея которой заключается в том, что качество программ проверяется **постоянно** в процессе разработки.

Тестирование становится центральной частью любого процесса разработки программ

Данная методика предъявляет существенно более **высокие требования** к квалификации инженера тестирования: в сферу его ответственности попадает не только функциональное тестирование, но и организация процесса разработки (процесс ежедневной сборки, участие в инспекциях, сквозных просмотрах и обычное чтение исходных текстов тестируемых программ). Поэтому идеальной кандидатурой на позицию тестировщика становится **наиболее опытный программист в команде**.

Зависимость вероятности правильного исправления ошибок и стоимости исправления ошибок от этапа разработки

Многократно проводимые исследования показали, что чем раньше обнаруживаются те или иные несоответствия или ошибки, тем больше вероятность их правильного исправления (рис. а) и ниже его стоимость (рис. б).

а

б

Основные понятия, связанные с тестированием и отладкой

Отладка программного средства – это деятельность, направленная на обнаружение и исправление ошибок в ПС с использованием процессов выполнения его программ.

Тестирование программного средства - процесс выполнения программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ.

Отладка = Тестирование + Поиск ошибок + Редактирование

Основные понятия, связанные с тестированием и отладкой

Процесс отладки включает:

- действия, направленные на выявление ошибок (тестирование);
- диагностику и локализацию ошибок (определение характера ошибок и их местонахождение);
- внесение исправлений в программу с целью устранения ошибок (редактирование).

Отладка = Тестирование + Поиск ошибок + Редактирование

Самым трудоемким и дорогим является тестирование, затраты на которое приближаются к **45%** общих затрат на разработку ПС и **от 30 до 60%** общей трудоемкости создания программного продукта.

Две задачи тестирования

Первая задача тестирования – подготовить набор тестов и применить к ним ПС, чтобы обнаружить в нём по возможности большее число несоответсвий.

Вторая задача тестирования - определить момент окончания отладки ПС (или отдельной его компоненты).

Для повышения **качества тестирования** рекомендуется соблюдать следующие основные принципы:

- предполагаемые результаты должны быть известны до тестирования;
- следует избегать тестирования программы автором;
- необходимо досконально изучать результаты каждого теста;
- необходимо проверять действия программы на неверных данных;
- необходимо проверять программу на неожиданные побочные эффекты на неверных данных.

Требования к программному продукту и тестирование

Разработка любого программного продукта начинается с выявления требований к этому продукту.

Спецификация (англ. *Software Requirements Specification*, SRS) - документ, в котором отражены все требования к продукту - описываются, как функциональные (что должна делать программа, варианты взаимодействия между пользователями и программным обеспечением), так и нефункциональные (например, на каком оборудовании должна работать программа, производительность, стандарты качества) требования.

Рекомендуемая стандартом IEEE 830 структура SRS

- Введение
  - Цели
  - Соглашения о терминах
  - Предполагаемая аудитория и последовательность восприятия
  - Масштаб проекта
  - Ссылки на источники
- Общее описание
  - Видение продукта
  - Функциональность продукта
  - Классы и характеристики пользователей
  - Среда функционирования продукта (операционная среда)
  - Рамки, ограничения, правила и стандарты
  - Документация для пользователей
  - Допущения и зависимости
- Функциональность системы
  - Функциональный блок X (таких блоков может быть несколько)
    - Описание и приоритет
    - Причинно-следственные связи, алгоритмы
    - Функциональные требования

Рекомендуемая стандартом IEEE 830 структура SRS (продолжение)

- Требования к внешним интерфейсам
  - Интерфейсы пользователя (UX)
  - Программные интерфейсы
  - Интерфейсы оборудования
  - Интерфейсы связи и коммуникации
- Нефункциональные требования
  - Требования к производительности
  - Требования к сохранности (данных)
  - Критерии качества программного обеспечения
  - Требования к безопасности системы
- Прочие требования
  - Приложение А: Глоссарий
  - Приложение Б: Модели процессов и предметной области и другие диаграммы
  - Приложение В: Список ключевых задач

Подходы к выработке стратегии проектирования тестов

1. Тестирование по отношению к спецификациям - **функциональный подход**

2. Тестирование по отношению к текстам программ - **структурный подход**

Стратегия проектирования тестов

В тестирование ПС входят

- постановка задачи для теста,
- проектирование,
- написание тестов,
- выполнение тестов,
- изучение результатов тестирования.

По объекту тестирования

Функциональное тестирование

Тестирование производительности

- Нагрузочное тестирование
- Стресс-тестирование
- Тестирование стабильности

Конфигурационное тестирование

Юзабилити-тестирование

Тестирование интерфейса пользователя

Тестирование безопасности

Тестирование локализации

Тестирование совместимости

По знанию системы

Тестирование чёрного ящика

Тестирование белого ящика

Тестирование серого ящика

По степени автоматизации –

Ручное тестирование

Автоматизированное тестирование

Полуавтоматизированное

По степени изолированности компонентов

Модульное тестирование

Интеграционное тестирование

Системное тестирование

По времени проведения тестирования

Альфа-тестирование

Дымовое тестирование

Тестирование новой функции

Подтверждающее тестирование

Регрессионное тестирование

Приёмочное тестирование

Бета-тестирование

По признаку позитивности сценариев

Позитивное тестирование

Негативное тестирование

По степени подготовленности к тестированию

Тестирование по документации (формальное тестирование)

Интуитивное тестирование (англ. ad hoc testing)

Подходы к выработке стратегии проектирования тестов

Функциональный подход основывается на том, что структура программного обеспечения не известна (программа рассматривается как «черный ящик»). В этом случае тесты проектируют, исследуя внешние спецификации или спецификации сопряжения программы или модуля, которые он тестирует.

Логика проектировщика тестов такова: «Меня не интересует, как выглядит эта программа, и выполнил ли я все команды. Я удовлетворен, если программа будет вести себя так, как указано в спецификациях».

В идеале - **проверить все возможные комбинации и значения на входе**.

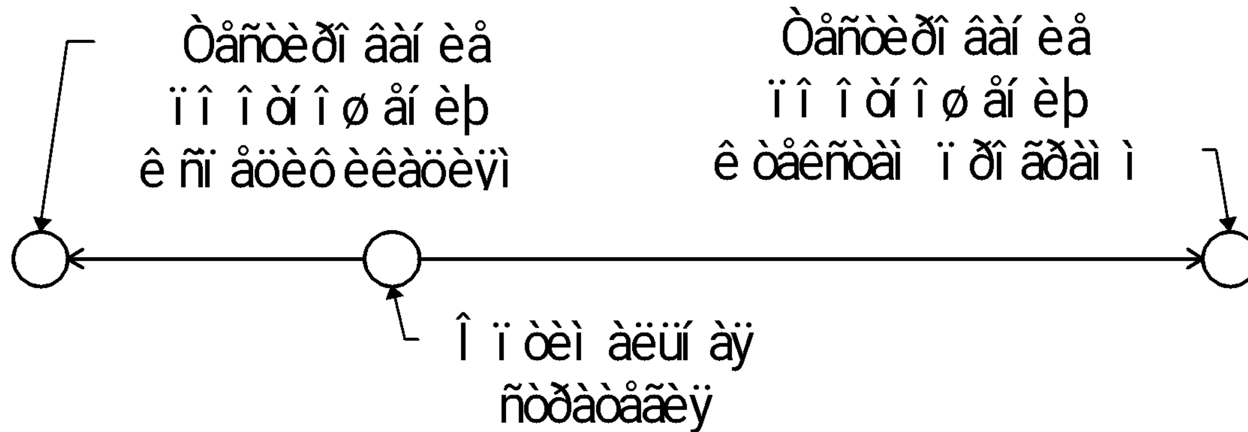
Подходы к выработке стратегии проектирования тестов

Структурный подход базируется на том, что известна структура тестируемого программного обеспечения, в том числе его алгоритмы («стеклянный ящик»). В этом случае тесты строят так, чтобы проверить правильность реализации заданной логики в коде программы.

Проектировщики тестов стремятся подготовить достаточное число тестов, чтобы каждая команда была выполнена, хотя бы, один раз. Чтобы каждая команда условного перехода выполнялась в каждом направлении хотя бы раз.

В идеале - **проверить каждый путь, каждую ветвь алгоритма**.

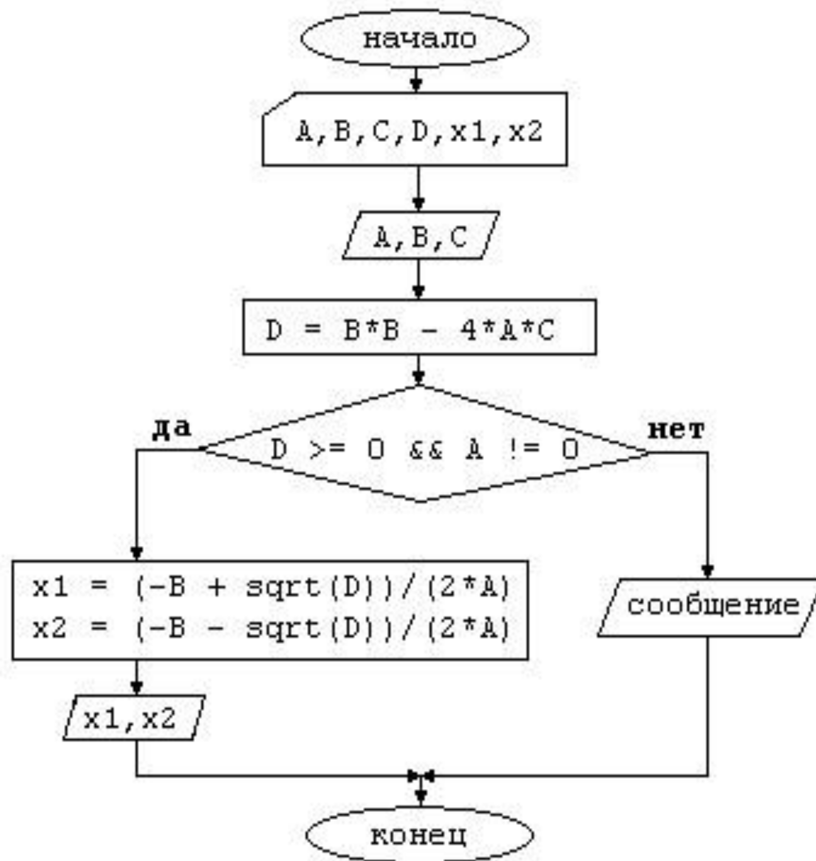
Подходы к выработке стратегии проектирования тестов

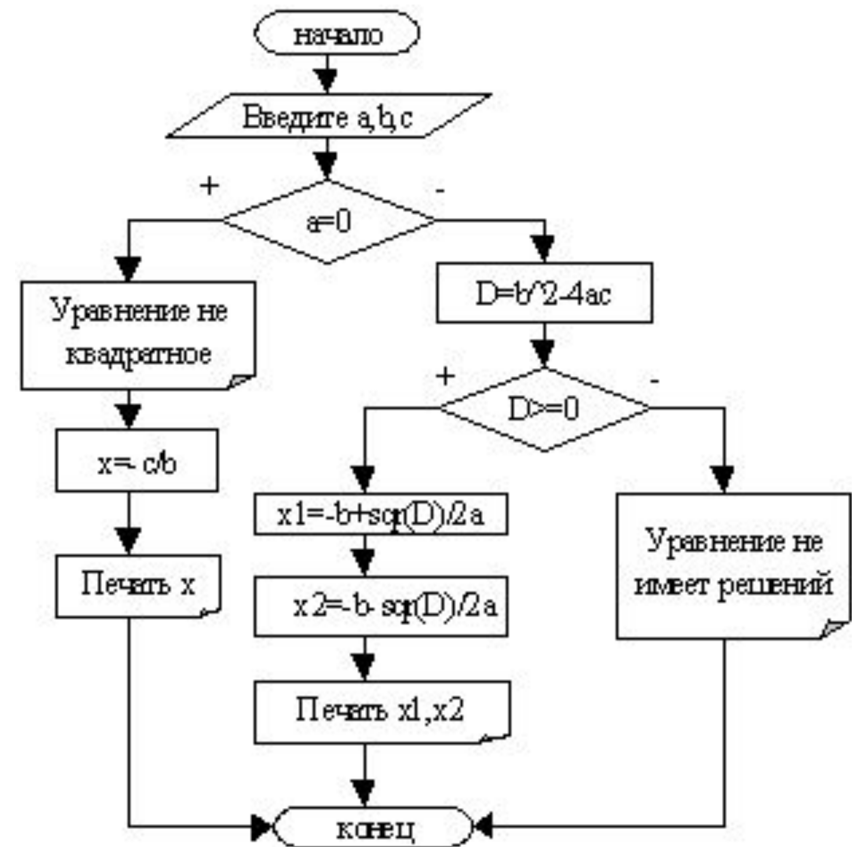


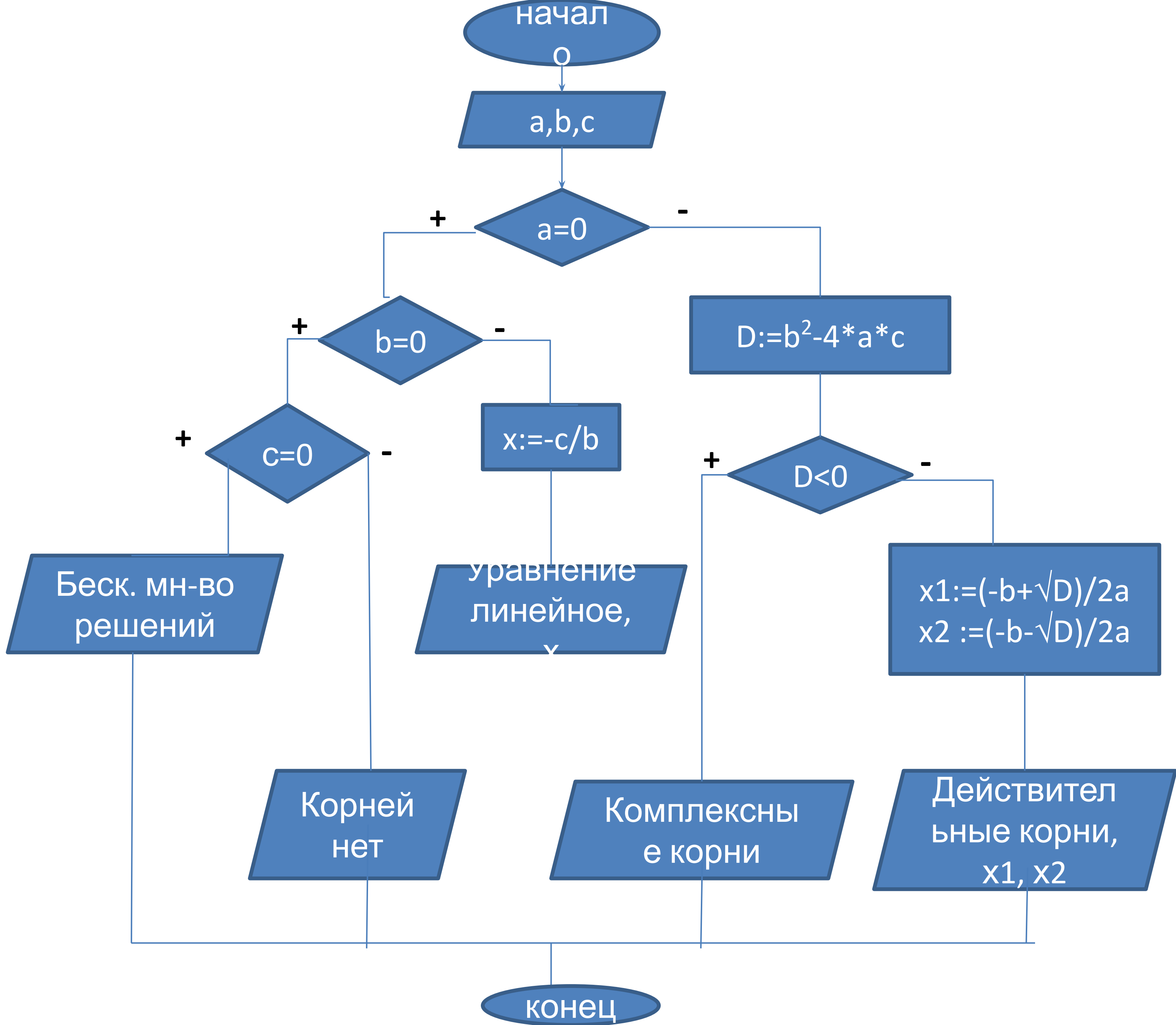
Оптимальная стратегия проектирования тестов расположена внутри интервала между этими крайними подходами, но ближе к левому краю

Наборы тестов, полученные в соответствии с методами этих подходов, обычно объединяют, обеспечивая всестороннее тестирование программного обеспечения.

Пример. Нахождение корней квадратного уравнения.
План тестирования, сами тесты и их количество зависят от требований к программе: какие сообщения и в каких случаях должен получить пользователь.






начал
о
a,b,c
a=0
+
-
b=0
+
-
D:=b2-4*a*c
c=0
+
-
x:=-c/b
D<0
+
-
Беск. мн-во
решений
уравнение
линейное,
x
x1:=(-b+√D)/2a
x2 :=(-b-√D)/2a
Корней
нет
Комплексны
е корни
Действител
ьные корни,
x1, x2
конец

Критерии полноты тестирования

Цель тестирования — обнаружить ситуацию, когда результаты работы программы не соответствуют входным данным. Самый простой способ сделать это — перебрать все возможные варианты входных данных и проверить правильность получаемых результатов. К сожалению, воспользоваться этим способом почти никогда не удается. Даже для простейших программ количество вариантов входных данных оказывается астрономическим.

Критерии, по которым проводится классификация всех возможных вариантов выполнения программы с точки зрения проверки ее правильности, называются **критериями полноты тестирования.**

Критерии полноты тестирования

Только на основании выбранного критерия можно определить тот момент времени, когда конечное множество тестов окажется достаточным для проверки программы с некоторой полнотой (степень полноты, определяется экспериментально).
Используется два вида критериев: критерии черного и белого ящика.
Соответственно тесты делятся на функциональные и структурные.

- **функциональные тесты** составляются исходя из спецификации программы;
- **структурные тесты** составляются исходя из текста программы.

Критерии полноты тестирования

- Функциональные критерии:

1) тестирование функций;
2) тестирование классов входных данных;
3) тестирование классов выходных данных;

- Структурные критерии:
  1) Покрытие операторов
  2) Покрытие условий
  3) Покрытие путей
  4) Покрытие функций
  5) Покрытие вход/выход

Критерий тестирования функций

Критерий **тестирования функций** актуален для многофункциональных программ. Он требует подобрать такой набор тестов, чтобы был выполнен хотя бы один тест для каждой из функций, реализуемых программой.

Рассмотрим программу для учета кадров предприятия. Скорее всего, она будет иметь следующие функции:

— принять на работу,
— уволить с работы,
— перевести с одной должности на другую,
— выдать кадровую сводку.

Критерии тестирования входных и выходных данных

Критерий **тестирования классов входных данных** требует классифицировать входные данные, разделить их на классы таким образом, чтобы все данные из одного класса были равнозначны с точки зрения проверки правильности программы. Считается, что если программа работает правильно на одном наборе входных данных из этого класса, то она будет правильно работать на любом другом наборе данных из этого же класса. Критерий требует выполнения хотя бы одного теста для каждого класса входных данных.

Критерий **тестирования классов выходных данных** выглядит аналогично предыдущему критерию, только проверяются не входные данные, а выходные.

Критерии тестирования входных и выходных данных

- Пример. Программа для учета кадров предприятия

Классы входных данных:

— приказ о приеме,
— приказ об увольнении,
— приказ о переводе,
— заявка на кадровую сводку.

Классы выходных данных:

— запись о приеме,
— запись об увольнении,
— запись о переводе,
— кадровая сводка.

Критерии тестирования входных и выходных данных

Тестирование области допустимых значений

Процесс тестирования области допустимых значений можно разделить на три этапа:

1. Проверка в нормальных условиях.
2. Проверка в экстремальных условиях.
3. Проверка в исключительных ситуациях.

Проверка в нормальных условиях

Проверка в нормальных условиях предполагает тестирование на основе данных, которые характерны для реальных условий функционирования программы. Проверка в нормальных условиях должна показать, что программа выдает правильные результаты для характерных совокупностей данных.

Критерии тестирования входных и выходных данных

- **Проверка в экстремальных условиях**

Тестовые данные этого этапа включают **граничные** значения области изменения входных переменных, которые должны восприниматься программой как правильные данные.

Для *нецифровых* данных необходимо использовать подобные типичные символы, охватывающие все возможные ситуации.

Для *цифровых* данных в качестве экстремальных условий следует брать начальное и конечное значения допустимой области изменения переменной при одновременном изменении длины соответствующего поля от минимальной до максимальной. Типичными примерами таких экстремальных значений являются *очень большие числа, очень малые числа и отсутствие информации*. Каждая программа характеризуется своими собственными экстремальными данными, которые должны подбираться программистом.

Критерии тестирования входных и выходных данных

Проверка в экстремальных условиях (продолжение)

Особый интерес представляют так называемые *нулевые примеры.*

Для цифрового ввода — это обычно нулевые значения вводимых данных; для последовательностей символов — это цепочка пробелов или нулей.

Нулевые примеры представляют собой один из лучших тестов, поскольку они имитируют состояние данных, которое время от времени имеет место в реальных условиях эксплуатации программы. Если подобное тестирование не выполняется, то впоследствии часто приходится сталкиваться с непонятным поведением программы.

Критерии тестирования входных и выходных данных

- **Проверка в исключительных ситуациях.** проводится с использованием данных, значения которых лежат за пределами допустимой области изменения.

Например:

- Что произойдет, если программе, не рассчитанной на обработку отрицательных или нулевых значений переменных, в результате какой-либо ошибки придется иметь дело как раз с такими данными?
- Как будет вести себя программа, работающая с массивами, если количество их элементов превысит величину, указанную в описании?
- Что случится, если цепочки символов окажутся длиннее или короче, чем это предусмотрено?

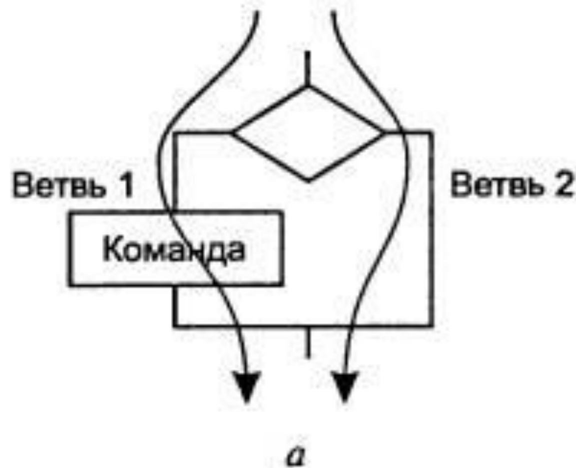
Структурные критерии

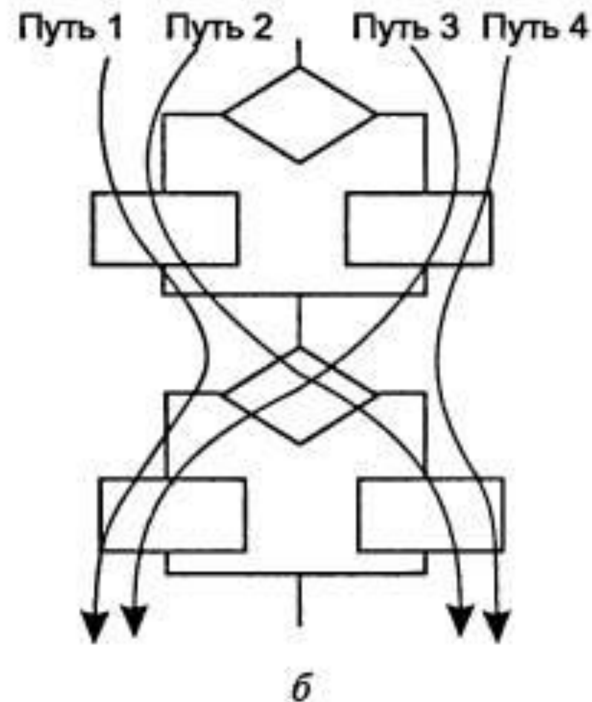
Структурные критерии - критерии покрытия кода.

Покрытие кода — мера, используемая при тестировании программного обеспечения. Она показывает процент, насколько исходный код программы был протестирован.

- Покрытие **операторов** — каждая ли строка исходного кода была выполнена и протестирована?
- Покрытие **условий** — каждая ли точка решения (вычисления истинно ли или ложно выражение) была выполнена и протестирована?
- Покрытие **путей** — все ли возможные пути через заданную часть кода были выполнены и протестированы?
- Покрытие **функций** — каждая ли функция программы была выполнена
- Покрытие **вход/выход** — все ли вызовы функций и возвраты из них были выполнены

Пример. Показывает отличие количества тестов при различных выбранных структурных критериях.





В случае выбора критерия «Покрытие операторов» достаточен 1 тест (рис.а)
В случае выбора критерия «Покрытие условий» достаточно двух тестов, покрывающих пути 1, 4 или 2, 3 (рис.б)
В случае выбора критерия «Покрытие путей необходимо четыре теста для всех четырех путей (рис.б)

Покрытие **операторов**

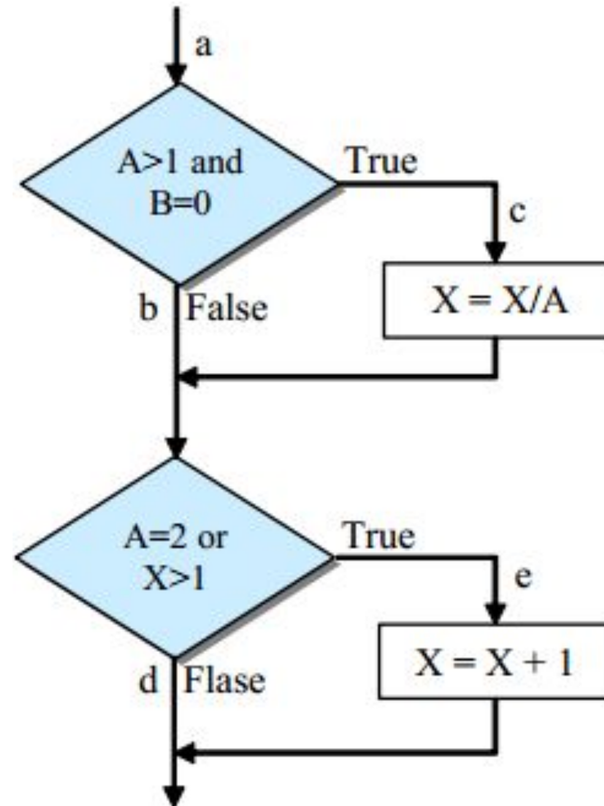
Пример 1

If ((A>1) and (B =0))

then X := X/A;

If ((A=2) or (X>1))

then X:=X+1;

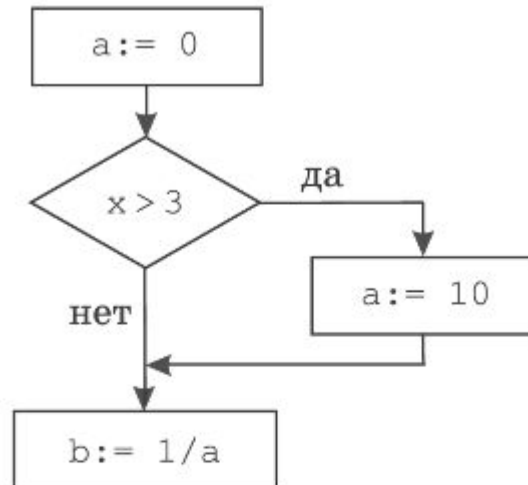


Можно выполнить каждый оператор, записав один-единственный тест, который реализовал бы путь ace. Иными словами, если бы в точке a были
установлены значения A = 2, B = 0 и X = 3, каждый оператор выполнялся бы один раз (в действительности X может принимать любое значение)

Покрытие **операторов**

Пример 2

```
a:= 0;
if x>3 then a:= 10;
b:= 1/a;
```



Для того чтобы удовлетворить критерию покрытия операторов, достаточно одного выполнения. Такого, чтобы x был больше 3. Очевидно, что ошибка в программе этим тестом обнаружена не будет. Она проявится как раз в том случае, когда $x \leq 3$. Но такого теста критерий покрытия операторов от нас не требует.

Покрытие **условий**

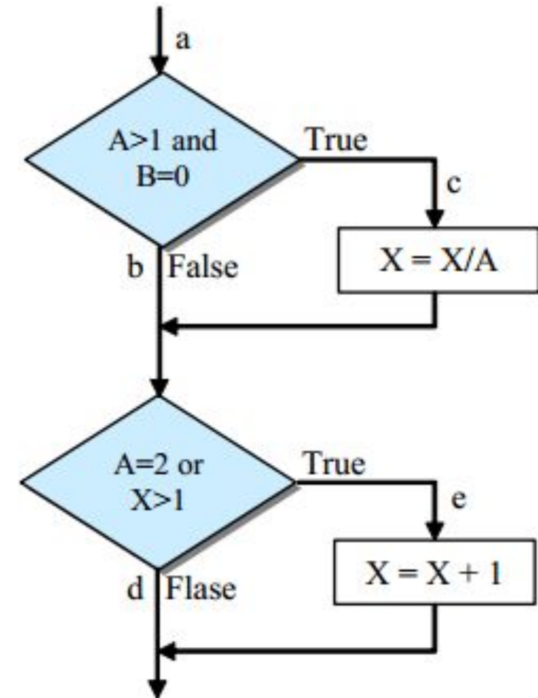
Пример 1

If ((A>1) and (B =0))

then X = X/A;

If ((A=2) or (X>1))

then X:=X+1;

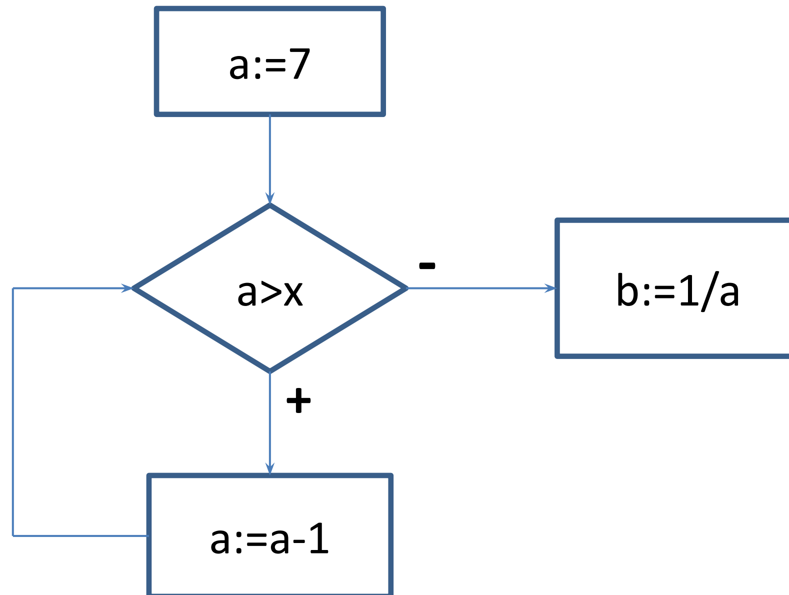


Покрытие условий может быть выполнено двумя тестами, покрывающими либо пути ace и abd, либо пути acd и abe.
Если мы выбираем последнее альтернативное покрытие, то входа-ми двух тестов являются A = 3, B = 0, X = 1 и A = 2, B = 1, X = 1.

Покрытие **условий**

Пример 2

a:=7;
while a>x do a:=a-1;
b:=1/a;



Для того чтобы удовлетворить критерию покрытия ветвей в данном случае достаточно одного теста. Например такого, чтобы x был равен 6 или 5. Все ветви будут пройдены. Но ошибка в программе обнаружена так и не будет. Она проявится в единственном случае, когда x=0. Но такого теста от нас критерий покрытия ветвей не требует.

Покрытие **путей**

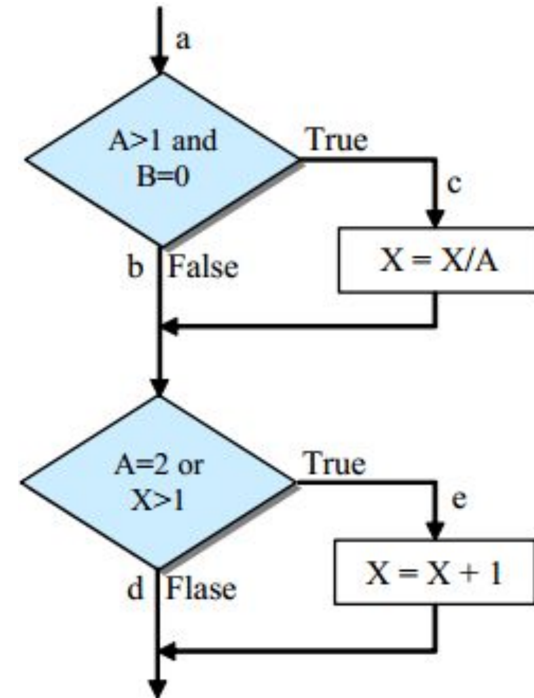
Пример 1

If ((A>1) and (B =0))

then X = X/A;

If ((A=2) or (X>1))

then X:=X+1;



Покрытие **путей** (все возможные пути через заданную часть кода должны быть выполнены и протестированы) может быть выполнено четырьмя тестами:

a,c,e – A=2, B=0, X=3

a,b,e – A=2, B=1, X=1

a,b,d – A=3, B=1, X=1

a,c,d – A=3, B=0, X=1

Покрытие **путей**

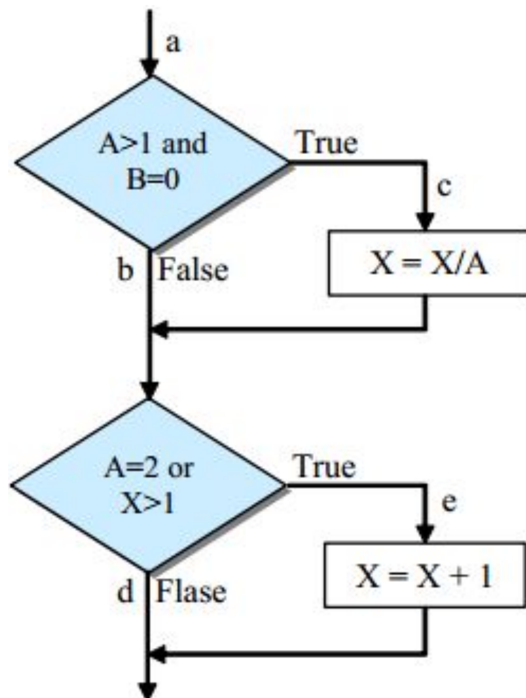
Пример 1

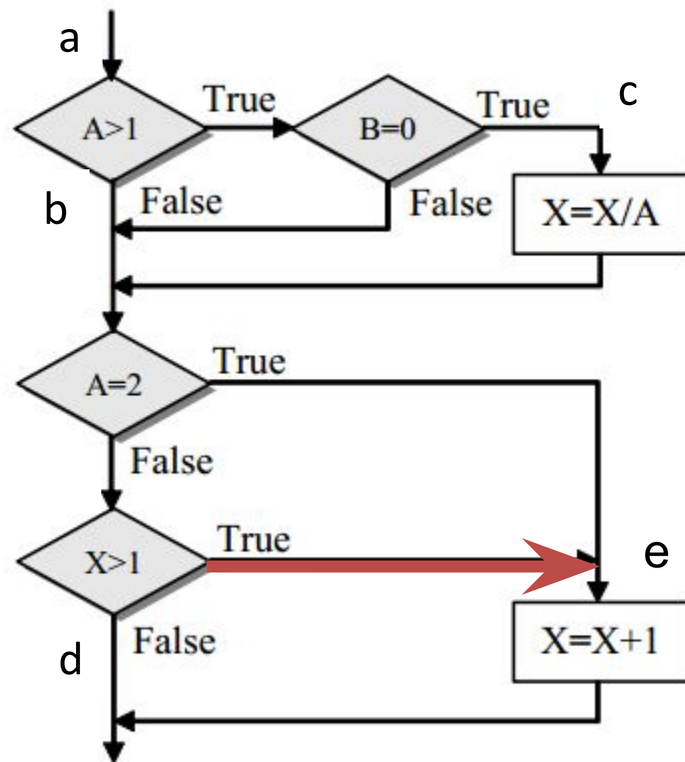
If ((A>1) and (B =0))

then X = X/A;

If ((A=2) or (X>1))

then X:=X+1;





Критерий комбинаторного покрытия условий

Пример 2

```
If (a=0) or (b=0) or (c=0)
Then d:=1/(a+b)
Else d:=1;
```

Ошибка будет выявлена только при a=0 и b=0. Критерий покрытия путей не гарантирует проверки такой ситуации.

Для решения этой проблемы был предложен критерий комбинаторного покрытия условий, который требует подобрать такой набор тестов, чтобы хотя бы один раз выполнялась *любая комбинация простых условий.*

Критерий значительно более надежен, чем покрытие путей, но обладает двумя существенными недостатками.

- Во-первых, он может потребовать очень большого числа тестов. Количество тестов, необходимых для проверки комбинаций n простых условий, равно 2^n.
- Во-вторых, даже комбинаторное покрытие условий не гарантирует надежную проверку циклов.

Два основных вида тестирования

1. **Автономное (модульное) тестирование** – последовательное раздельное тестирование различных частей программ, входящих в ПС, с поиском и исправлением в них фиксируемых при тестировании ошибок. Включает отладку каждого программного модуля и их сопряжения.
2. **Комплексное (системное) тестирование** – тестирование ПС в целом с поиском и исправлением фиксируемых при тестировании ошибок во всех рабочих продуктах;

Уровни тестирования

- **Модульное тестирование (автономное тестирование, юнит-тестирование)** — тестируется минимально возможный для тестирования компонент, например, отдельный класс или функция. Часто модульное тестирование осуществляется разработчиками ПО.
- **Интеграционное тестирование** — тестируются интерфейсы между компонентами, подсистемами. При наличии резерва времени на данной стадии тестирование ведётся итерационно, с постепенным подключением последующих подсистем.
- **Системное тестирование** — тестируется интегрированная система на её соответствие требованиям.

Основные этапы разработки сценария автономного тестирования

1. На основании спецификации отлаживаемого модуля подготовить тесты для
   - каждой логической возможности ситуации;
   - каждой границы областей возможных значений всех входных данных;
   - каждой области недопустимых значений;
   - каждого недопустимого условия.
2. Проверить текст модуля, чтобы убедиться, что каждое направление любого разветвления будет пройдено хотя бы один раз. Добавить недостающие тесты.

Основные этапы разработки сценария автономного тестирования

3. Проверить текст модуля, чтобы убедиться, что для каждого цикла существуют тесты, обеспечивающие, по крайней мере, три следующие ситуации
   - тело цикла не выполняется ни разу;
   - тело цикла выполняется один раз;
   - тело цикла выполняется максимальное число раз;
4. Проверить текст модуля, чтобы убедиться, что существуют тесты, проверяющие чувствительность к отдельным особым значениям входных данных. Добавить недостающие тесты.

Таблица 5.3. **Поиск численного решения минимального набора тестов**

Номер теста	*a*	*b*	*c*	Ожидаемый результат	Что проверяется
1	2	–5	2	$x_1 = 2,\ x_2 = 0{,}5$	Случай вещественных корней
2	3	2	5	Сообщение	Случай комплексных корней
3	3	–12	0	$x_1 = 4,\ x_2 = 0$	Нулевой корень
4	0	0	10	Сообщение	Неразрешимое уравнение
5	0	0	0	Сообщение	Неразрешимое уравнение
6	0	5	17	Сообщение	Неквадратное уравнение (деление на 0)
7	9	0	0	$x_1 = x_2 = 0$	Корень из 0

Основная особенность практики тестирования ПС

По мере роста числа обнаруженных и исправленных ошибок в ПС **растёт** также относительная вероятность существования в нём **необнаруженных** ошибок.

Это подтверждает важность предупреждения ошибок на **всех стадиях разработки ПС**.

Пример автономного структурного тестирования фрагмента программы

```
Procedure m (a, b: real; var x: real);
begin
    if (a>1) and (b=0) then x:=x/a;
    if (a=2) or (x>1) then x:=x+1;
end;
```

Для формирования тестов программу представляют в виде **графа**, вершины которого соответствуют операторам программы, а дуги представляют возможные варианты **передачи управления**.

Творческая работа

1. Разделиться на группы
2. Получить тему (практические работы по Delphi №№ 3, 5, 7, 9, 10)
3. Составить спецификацию
4. Разработать программу тестирования:
    4.1. Определить виды тестирования
    4.2. Определить объекты тестирования
    4.3. Определить субъекты тестирования
    4.4. Определить классы входных данных
    4.5. Написать тест-кейсы для тестирования функций и ожидаемые результаты
    4.6. Написать тест-кейсы для структурного тестирования и ожидаемые результаты

Составить чек-листы для проведения всех видов тестирования

5. Провести тестирование
6. Сделать выводы

Содержание ПЗ к проекту

- Титульный лист
- Бриф
- Спецификация
- ТЗ
- Пользователи
- Интерфейсы
- Информационно-логическая схема
- Схема БД
- Алгоритм одной процедуры
- Программа тестирования
- Результаты тестирования