Тренинги Intel Delta Course

«Дополнительные главы по Software Engineering»

Методики Тестирования

Боциев А.Я., Виценко А.Ю., Крюков А.К., Моренов О.А., Пряхин И.В.,
Семенов Д.С., Чиликин Е.В. Intel

Позитивные и негативные тесты

Позитивные тесты

- Тесты, предназначенные для проверки, что программа выполняет свое основное предназначение
- Тесты на основании «правильных» входных данных
- Тестирование с целью проверки соответствий требованиям

Негативные тесты

- Тесты для проверки устойчивости программы к негативным входным данным
- Тесты на проверки устойчивости программы к ошибкам пользователя
- Тесты на то, что у программы нет неожиданных побочных эффектов
- Тестирование с целью «сломаем это!»

Тестирование Черного Ящика

- Не знаем/Игнорируем устройство тестируемого объекта
- Можем управлять входными параметрами
- Среда, в которой проводим эксперименты, может считаться входным параметром
- Можем измерять выходные параметры

Черный Ящик. Когда применять.

Области для поиска ошибок:

Неправильные или пропущенные функции

Ошибки интерфейсов

Инициализация и завершение

Производительность

Структура данных

Стадии применения:

Unit-тестирование

Интеграционное тестирование

Системное тестирование

Приемочное тестирование

Тестирование Черного Ящика - Шаги

1. Изучение спецификаций и требований
2. Выбор входных значений
3. Определение ожидаемых выходных значений

Входные значения	Ожидаемые выходные значения
Вход 1	Выход 1
Вход 2	Выход 2
…	….
Вход N	Выход N

4. Исполнение тестов
5. Сравнение полученных результатов с ожидаемыми

Тестирование Черного Ящика. Стратегии

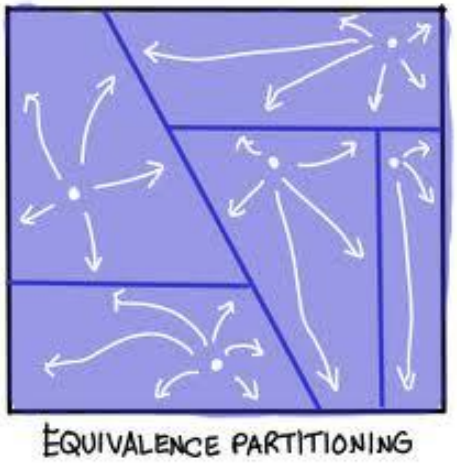
Число тестов определяется числом входов и диапазоном возможных значений входных параметров

Перебор всех возможных входных параметров, как правило, невозможен

Пример: Сложение двух 4хбайтовых целых -
2^{64} входных параметров

Стратегии уменьшения числа тестов:

Классы эквивалентности



Граничные значения

Черный Ящик. Преимущества и недостатки

Преимущества:

- Тестирование с точки зрения пользователя
- Не требует специальных знаний (например, конкретного языка программирования)
- Позволяет найти проблемы в спецификациях
- Можно создавать тесты параллельно с кодом
- Тестировщик может быть отделен от разработчиков

Недостатки:

- Эффективность зависит от выбора конкретных тестовых значений
- Необходимость наличия четких и полных спецификаций
- Невозможность сконцентрироваться на особо сложных частях кода
- Трудность локализации причины дефекта
- Возможность не протестировать часть кода

Тестирование Белого Ящика

- Используем знание об устройстве тестируемого объекта
- В случае ПО – имеем полный доступ к тестируемому коду

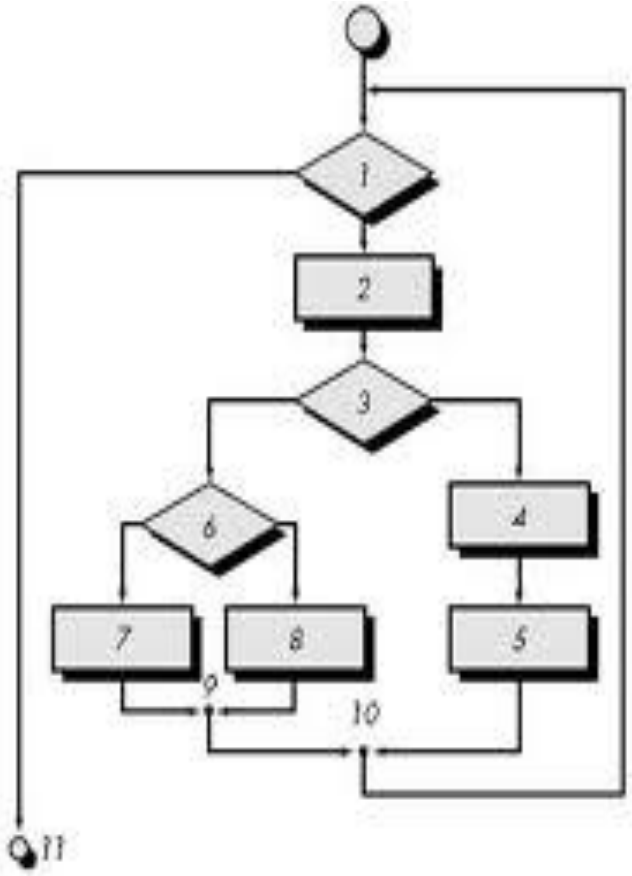
Стадии применения:

- Unit-тестирование
- Интеграционное тестирование

Белый Ящик. Шаги

Представляем программу в виде графа

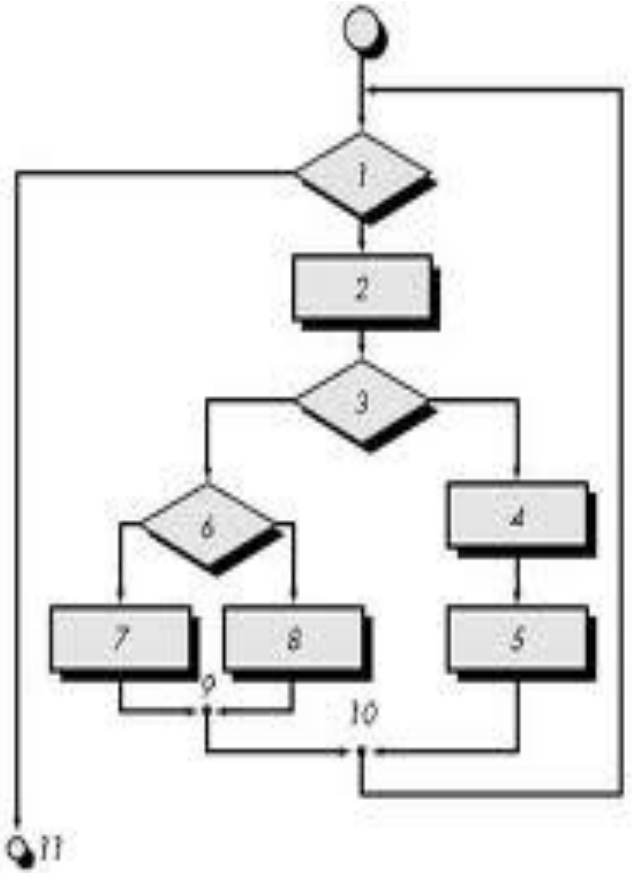
```
PROCEDURE average;

  * This procedure computes the average of 100 or fewer
    numbers that lie between bounding values; it also computes the
    sum and the total number valid.

  INTERFACE RETURNS average, total.input, total.valid;
  INTERFACE ACCEPTS value, minimum, maximum;

  TYPE value[1:100] IS SCALAR ARRAY;
  TYPE average, total.input, total.valid;
    minimum, maximum, sum IS SCALAR;
  TYPE i IS INTEGER;
1 i = 1;
  total.input = total.valid = 0;   2
  sum = 0;
  DO WHILE value[i] <> -999 AND total.input < 100   3
    4 increment total.input by 1;
    IF value[i] > = minimum AND value[i] < = maximum   6
  5     THEN increment total.valid by 1;
  7           sum = s sum + value[i]
        ELSE skip
  8   ENDIF
    increment i by 1;
  9 ENDDO
    IF total.valid > 0   10
      11 THEN average = sum / total.valid;
  12    ELSE average = -999;
  13 ENDIF
END average
```



Белый Ящик. Шаги

Создаем тестовые сценарии чтобы:

- Попасть в каждое ветвление
- Пройти хоть раз через все вершины
- Пройти всеми возможными путями
- Пройти через вновь добавленные участки
- Пройти через известные проблемные участки



Белый Ящик. Преимущества и Недостатки

Преимущества:

- Позволяет найти «скрытые» в коде дефекты
- Позитивные побочные эффекты (например, обучение команды)
- Нахождение проблем производительности
- Более надежное разбиение на классы эквивалентности
- Как правило, ускорение цикла нахождение-исправление

Недостатки:

- Не найдем пропущенное в коде
- Дорого

Отличия черного и белого ящиков

Критерий	Черный Ящик	Белый Ящик
Основной уровень применимости	Приемочное тестирование	Юнит-тестирование
Ответственный	Независимый тестировщик	Разработчик
Знание программирования	Не обязательно	Необходимо
Знание реализации	Не обязательно	Необходимо
Знание сценариев использования	Необходимо	Не обязательно
Основа тестовых сценариев	Спецификации	Код

Серый Ящик

Комбинация черного и белого ящиков:

- Знаем частично или полностью внутреннее устройство тестируемого объекта
- Тестировщик находится на уровне пользователя

Пример:

Зная особенности реализации модуля, создаем тестовые сценарии пользовательского уровня, которые покрывают потенциально проблемную область

Основная область применения: интеграционное тестирование

Покрытие программного кода

Вопрос: Для нашей программы имеется 100 тестов. Можем ли мы утверждать, что программа «хорошо» протестирована?

Необходимо измерение качества тестирования

Покрытие кода (code coverage) – мера измерения оттестированности имеющегося программного кода

Виды покрытия кода

Функциональное (**Function coverage**) – каждая функция вызывается хотя бы раз

Строковое **(Statement coverage) –** каждая строка кода выполнялась хотя бы раз

Решения (**Decision/Branch coverage**) – в каждом условном операторе прошли по всем веткам выбора

Условия (**Condition coverage**) – каждое атомарное булево выражение приняло значения и «истина» и «ложь»

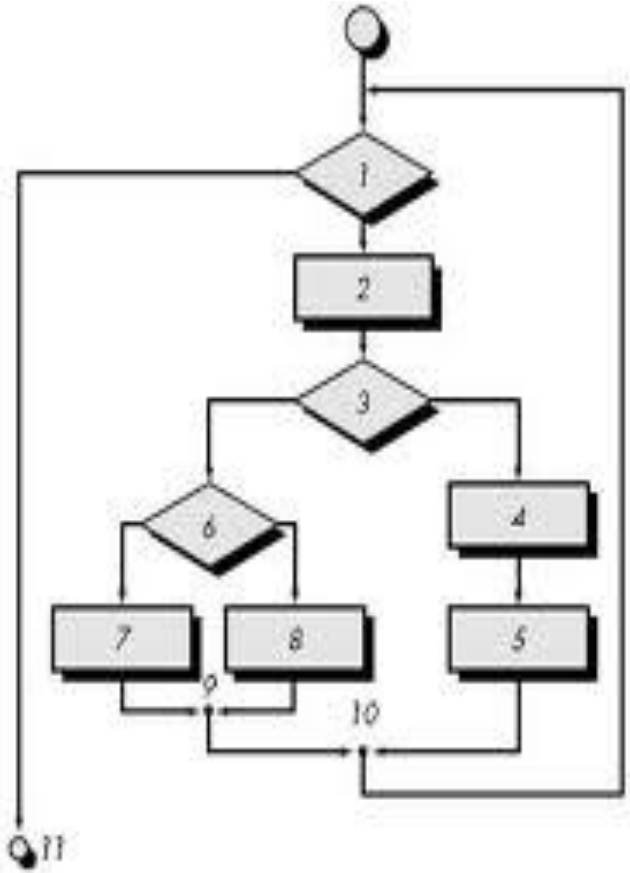
Параметров (**Parameter Value coverage**) – если метод имеет параметры, все значения параметра были использованы

Пути (**Path Coverage**) – все возможные пути в коде были пройдены

Циклы (**Loop coverage**) – Все циклы исполнялись 0,1,..N раз

Покрытие кода - пример

```
int foo (int x, int y)
{
    int z = 0;
    if ((x>0) && (y>0))
    {
        z = x;
    }
    return z;
}
```



Покрытие кода - Инструменты

Microsoft Visual Studio 2010(C++, C#)

DevPartner (C#, Java)

Codecov из Intel Compiler (C, C++, Fortran)

Jtest (Java)

Devel::Cover (Perl)

PHPUnit (PHP)

Coverage (Python)

CoverMe (Ruby)

….

Техники тестирования

Техника – «способ что-то сделать»

Техники тестирования концентрируются на следующих аспектах:

- **Границы** – что тестируем (например, функциональное тестирование)
- **Покрытие** – намерения по тестированию (например, каждую функцию)
- **Тестировщики** – кто тестирует (например, end-user testing)
- **Риски** – потенциальные проблемы которые ищем (например, утечки памяти, уязвимости в системе безопасности)
- **Активности** – как выполняются тесты (например, проверка всех пар)
- **Способ оценки результата** – как мы оцениваем результат (например, сравнение со стандартом)
- **Цель** – цель тестирования (например, проверка корректности сборки)

Техники, ориентированные на границы/покрытие

- Function testing
- Feature or function integration testing
- Tours
- Equivalence class analysis
- Boundary testing
- Best representative testing
- Domain testing
- Test idea catalogs
- Logical expressions
- Multivariable testing
- State transitions
- User interface testing
- Specification-based testing
- Requirements-based testing
- Compliance-driven testing
- Configuration testing
- Localization testing

Техники, ориентированные на исполнителя

- User testing
- Alpha testing
- Beta testing
- Bug bashes
- Subject-matter expert testing
- Paired testing
- Eat your own dogfood
- Localization testing

Техники, ориентированные на риски

- Boundary testing
- Quicktests
- Constraints
- Logical expressions
- Stress testing
- Load testing
- Performance testing
- History-based testing
- Risk-based multivariable testing
- Usability testing
- Configuration / compatibility testing
- Interoperability testing
- Long sequence regression

Техники, ориентированные на способ тестирования

- Guerilla testing
- All-pairs testing
- Random testing
- Use cases
- Scenario testing
- Installation testing
- Regression testing
- Long sequence testing
- Dumb monkey testing
- Load testing
- Performance testing

Техники, основанные на оракуле

- Function equivalence testing
- Mathematical oracle
- Constraint checks
- Self-verifying data
- Comparison with saved results
- Comparison with specifications or other authoritative documents
- Diagnostics-based testing
- Verifiable state models

Техники, основанные на цели

- Build verification
- Confirmation testing
- User acceptance testing
- Certification testing

Материалы и источники

1. The Art of Software Testing. Glenford J. Myers
2. Материалы сайта http://softwaretestingfundamentals.com
3. Материалы сайта http://www.onestoptesting.com
4. Материалы сайта http://www.sqatester.com
5. http://en.wikipedia.org/wiki/Code_coverage
6. http://www.testingeducation.org/BBST/testdesign/BBSTTestDesign2011pfinal.pdf

Домашнее задание - задача

Напишите набор тестовых сценариев для программы ниже, используя метод белого ящика. Необходимо добиться 100% покрытия statement coverage и condition coverage. Тестовые сценарии запишите с виде таблицы

a	b	c	Ожидаемое выходное значение

Заданы длины трех отрезков *a*, *b*, *c*. Необходимо определить, можно ли из них составить треугольник. В случае утвердительного ответа определить его тип: остроугольный, прямоугольный или тупоугольный.

Вход. Три целых числа *a*, *b*, *c* – длины трех отрезков.

Выход. Строка, содержащая информацию о треугольнике: “ACUTE”, если он остроугольный, “RIGHT” если прямоугольный и “OBTUSE” если тупоугольный. Если из трех отрезков составить треугольник нельзя, то вывести “NONE”.

Домашнее задание - программа

```
program triangle;
var
  a,b,c:integer;
  res:string;
begin
  readln(a,b,c);
  if ((a >= b + c) or (b >= a + c) or (c >= a + b))
      then res := 'NONE' else
  if ((a*a = b*b + c*c) or (b*b = a*a + c*c) or (c*c = a*a + b*b))
      then res := 'RIGHT' else
  if ((a*a < b*b + c*c) and (b*b < a*a + c*c) and (c*c < a*a + b*b))
      then res := 'ACUTE' else
  res := 'OBTUSE';
  writeln(res);
end.
```