Свиридова О.В., Рыбанов А.А.

# *Тестирование и отладка программного обеспечения*

Волжский

2024

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ВОЛЖСКИЙ ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ (ФИЛИАЛ)
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО БЮДЖЕТНОГО ОБРАЗОВАТЕЛЬНОГО
УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
КАФЕДРА «ИНФОРМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ»

О.В. Свиридова, А.А. Рыбанов

# *Тестирование и отладка программного обеспечения*

*Электронное учебное пособие*

Волжский

2024

УДК 004.45(07)
ББК 32.973.202я73
С 247

Р е ц е н з е н т ы :
заведующий кафедрой методики преподавания математики и физики, ИКТ ФГБОУ ВО «Волгоградский государственный социально-педагогический университет (ВГСПУ)», доктор педагогических наук, профессор
*Смыковская Т.К.;*
директор ООО НПЦ «АИР»
*Шуревский А.Н.*

Издается по решению редакционно-издательского совета
Волгоградского государственного технического университета

Свиридова, О.В.
Тестирование и отладка программного обеспечения [Электронный ресурс] : учебное пособие / О. В. Свиридова, А. А. Рыбанов ; Министерство науки и высшего образования Российской Федерации, ВПИ (филиал) ФГБОУ ВО ВолгГТУ. – Электронные текстовые данные (1 файл: 1,17 МБ). – Волжский, 2024. – Режим доступа: http://lib.volpi.ru. – Загл. с титул. экрана.
ISBN 978-5-9948-4894-4

В учебном пособии представлены два раздела – общие понятия тестирования и классификация тестов по целям и задачам тестирования. Каждый раздел пособия содержит теоретическую часть, практические примеры, систематизированную подборку контрольных заданий.

Предназначено для студентов, обучающихся по направлению 09.03.04 «Программная инженерия» в рамках курса «Тестирование и отладка программного обеспечения».

Илл. 21, табл. 6, библиограф.: 6 назв.

ISBN 978-5-9948-4894-4

## Содержание

## ВВЕДЕНИЕ

Электронный ресурс «Тестирование и отладка программного обеспечения» является учебным пособием, которое содержит теоретический материал по общим понятиям тестирования и классификации тестов по целям и задачам тестирования и является дополнением к лекциям по дисциплине «Тестирование и отладка программного обеспечения».

Электронное учебное пособие разработано в соответствии с требованиями Федерального государственного стандарта высшего профессионального образования и предназначено для студентов бакалавриата, обучающихся по направлению 09.03.04 «Программная инженерия». Областью применения данного электронного ресурса является учебный процесс в Волжском политехническом институте (филиал) Волгоградского технического университета, объектом обучения могут выступать любые студенты направления «Программная инженерия». Электронное учебное пособие признано обеспечить взаимодействие студента с предлагаемым преподавателем учебным материалом, позволяя работать как самостоятельно, так и в аудитории с помощью последнего.

Структура учебного пособия, которое состоит из введения, двух разделов и библиографического списка, соответствует общим требованиям к оформлению такого вида изданий.

В первом разделе рассматриваются общие понятия тестирования, в том числе концепция и организация тестирования, а также проблемы тестирования. Кроме того, в этом разделе описаны требования к идеальному критерию тестирования и представлены классы критериев.

Во втором разделе описана классификация тестов по целям и задачам тестирования, в частности статическое, динамическое и стохастическое, которые включают в себя различные методы и стратегии тестирования.

## 1. Общие понятия тестирования

### 1.1. Основная терминология

- ***Тестирование*** – процесс выявления фактов расхождений с требованиями (ошибок).
- ***Отладка*** (debug, debugging) – процесс поиска, локализации и исправления ошибок в программе.

Термин «отладка» в отечественной литературе используется двояко: для обозначения активности по поиску ошибок (собственно тестирование), по нахождению причин их появления и исправлению или активности по локализации и исправлению ошибок.

Как правило, на ***фазе тестирования*** осуществляется исправление идентифицированных ошибок, включающее:

- локализацию ошибок;
- нахождение причин ошибок;
- корректировку программы.

Судить о правильности результатов выполнения программы можно только сравнивая спецификацию функции с результатами ее вычисления.

***Основная проблема тестирования*** – определение достаточности множества тестов для истинности вывода о правильности реализации программы, а также нахождения множества тестов, обладающего этим свойством.

**Определения тестирования по стандарту**

- Процесс выполнения ПО системы или компоненты при заданных условиях с анализом или записью результатов и оценкой некоторых свойств тестируемого объекта.
- Процесс анализа ПО с целью фиксации различий между существующим состоянием ПО и требуемым (что свидетельствует о проявлении ошибки) и оценки свойств тестируемого ПО.

- Контролируемое выполнение программы на конечном множестве тестовых данных и анализ результатов этого выполнения для поиска ошибок [IEEE Std 829-1983].

**Статическое и динамическое тестирование**

Тестирование программного обеспечения делится на статическое и динамическое. Главной задачей статического тестирования является найти недостатки уже в фазах проектирования программы и спецификации. Во время статического тестирования можно также проверить свойства системы, такие как ремонтопригодность, надежность, анализируемость.

Проведение статического тестирования может значительно снизить затраты на разработку программного обеспечения и уменьшить время, необходимое для разработки программного обеспечения. Тем не менее, необходимо помнить, что статическое тестирование не является заменой динамического тестирования и нельзя гарантировать, что программное обеспечение, прошедшее только статическое тестирование, будет работать безупречно. Инструменты статического анализа уведомляют разработчиков программного обеспечения о таких недостатках как непригодный для использования программный код, неописанные переменные и т.д.

Динамическое тестирование делится на:

- функциональное тестирование;
- структурное тестирование.

Функциональные методы тестирования также известны как «*Black Box*» (черный ящик), а также техники тестирования входов / выходов.

При структурных методах тестирования применяемые тесты исходят из внутренней структуры программного обеспечения и их называют также методы «WhiteBox» (белый ящик), так как при их применении надо знать, как внедрено программное обеспечение и как оно работает. Как правило, этими методами пользуются сами разработчики программного обеспечения. Методы структурного тестирования – типичные методы модуль-тестирования, при которых те-

стируются только компоненты системы программного обеспечения.

Статическое тестирование (Statictesting) – тестирование, в ходе которого тестируемая программа (код) не выполняется (запускается). Анализ программы происходит на основе исходного кода, который вычитывается вручную либо анализируется специальными инструментами.

Статическое тестирование выполняется, например, с помощью специальных инструментов контроля кода – CodeChecker.

Примеры статического тестирования:

- Обзоры (Reviews);
- Инспекции (Inspections);
- Сквозные просмотры (Walkthroughs);
- Аудиты (Audits);
- также к статическому тестированию относят тестирование требований, спецификаций, документации.

Динамическое тестирование (собственно тестирование) выполняется, например, с помощью специальных инструментов автоматизации тестирования – Testbed или Testbench.

Статическое тестирование выявляет неверные конструкции или неверные отношения объектов программы (ошибки формального задания) формальными методами анализа без выполнения тестируемой программы:

- с помощью специальных инструментов контроля кода;
- Обзоры (Reviews);
- Инспекции (Inspections);
- Сквозные просмотры (Walkthroughs);
- Аудиты (Audits);
- тестирование требований, спецификаций, документации.

***Динамическое*** тестирование осуществляет выявление ошибок на выполняющейся программе. Тестирование заканчивается, когда выполнилось или «прошло» (pass) успешно достаточное количество тестов в соответствии с вы-

бранным критерием тестирования.

Примеры динамического тестирования:

- модульное тестирование (unittesting);
- интеграционное тестирование (integratedtesting);
- приемочное тестирование (acceptancetesting).

**В чем разница между понятиями статическое и динамическое тестирование?** Статическое тестирование производится без запуска программного кода продукта. Тестирование осуществляется путем анализа программного кода (codereview) или скомпилированного кода. Анализ может производиться как вручную, так и с помощью специальных инструментальных средств. Целью анализа является раннее выявление ошибок и потенциальных проблем в продукте.

С помощью codereview на раннем этапе могут быть выявлены ошибки в коде продукта. Как правило, codereview производится самими разработчиками.

Примерами ошибок, которые потенциально можно выявить с помощью автоматического статического тестирования, могут быть:

- утечки ресурсов (утечки памяти, неосвобождаемые файловые дескрипторы и т.д.);
- возможность переполнения буфера (bufferoverflows);
- ситуации частичной (неполной) обработки ошибок.

Как правило, результатом автоматического анализа кода является список рекомендаций для ручного review некоторых участков кода, потенциально содержащих ошибки.

В отличии от статического, динамическое тестирование производится путем запуска продукта и проверки его функционала. Проверка осуществляется с помощью ручного или автоматического выполнения заранее подготовленного набора тестов.

## 1.2. Концепция тестирования

***Программа*** – это аналог формулы в обычной математике.

Формула для функции f, полученной *суперпозицией* функций f1, f2, ... fn – *выражение*, описывающее эту *суперпозицию:*

***f = f1* f2* f3*... * fn***

Если аналог f1, f2,... fn – *операторы* языка программирования, то их формула – *программа*.

Существует два метода обоснования *истинности формул*.

***1) Формальный подход*** или ***доказательство*** применяется, когда из исходных формул-аксиом с помощью формальных процедур (правил вывода) выводятся искомые формулы и утверждения (теоремы). Вывод осуществляется путем перехода от одних формул к другим по строгим правилам, которые позволяют свести процедуру перехода от формулы к формуле к последовательности текстовых подстановок:

***A**3 = A*A*A***

***A*A*A = A -> R,  A*R -> R***

***2) Интерпретационный подход*** применяется, когда осуществляется подстановка констант в формулы, а затем интерпретация формул как осмысленных утверждений в элементах множеств конкретных значений. *Истинность* интерпретируемых формул проверяется на конечных множествах возможных значений. Сложность подхода состоит в том, что на конечных множествах комбинации возможных значений для реализации исчерпывающей проверки могут оказаться достаточно велики.

Интерпретационный подход используется при экспериментальной проверке соответствия программы своей спецификации. Применение интерпретационного подхода в форме экспериментов над исполняемой программой составляет суть *отладки* и *тестирования*.

Если программа не содержит синтаксических ошибок (прошла трансля-

цию) и может быть выполнена на компьютере, она обязательно вычисляет какую-либо функцию, осуществляющую отображение входных данных в выходные. Это означает, что компьютер на своих ресурсах доопределяет частично определенную программой функцию до тотальной определенности. Следовательно, судить о правильности или неправильности результатов выполнения программы можно, только сравнивая спецификацию желаемой функции с результатами ее вычисления, что и осуществляется в процессе тестирования.

Примеры поиска и исправления ошибки представлены на рисунках 1, 2. Отладка обеспечивает локализацию ошибок, поиск причин ошибок и соответствующую корректировку программы (пример программы на C#, пример программы на C++).

```
// Метод вычисляет неотрицательную степень n
// числа x
static public double Power(double x, int n)
{
double z=1;
for (int i=1;n>=i;i++)
{
z = z*x;
}
return z;
}
```

Рисунок 1. Пример программы на C#

```
Power double Power(double x,int n)
{
double z=1;
int i;
for (i=1;n>=i; i++)
  {
    z=z*x;
  }
return z;
}
```

Рисунок 2. Пример программы на C++

Если вызвать метод Power с отрицательным значением степени n Power(2,-1), то получим некорректный результат 1. Исправим метод так, чтобы ошибочное значение параметра (недопустимое по спецификации значение) идентифицировалось специальным сообщением, а возвращаемый результат был равен 1 (рис. 3, 4). Если вызвать скорректированный метод

PowerNonNeg(2,-1) с отрицательным значением параметра степени, то сообщение об ошибке будет выдано автоматически.

```
// Метод вычисляет неотрицательную
// степень n числа x
static public double PowerNonNeg(double x,
int n)
{
double z=1;
if (n>0)
 {
for (int i=1;n>=i;i++)
  {
    z = z*x;
  }
 }
 elseConsole.WriteLine(
"Ошибка ! Степеньчисла n" +
" должна быть больше 0.");
return z;
}
```

Рисунок 3. Скорректированный исходный текст программы на C#

```
doublePowerNonNeg(double x, int n)
{
double z=1;
 inti;
if (n>0)
 {
for (i=1;n>=i;i++)
  {
    z = z*x;
}
 }
elseprintf("Ошибка! Степень числа n должна быть
больше 0.\n");
return z;
}
```

Рисунок 4. Скорректированный исходный текст программы на C++

### 1.3. Организация тестирования

Тестирование осуществляется на заданном заранее множестве входных данных X и множестве предполагаемых результатов Y – (X,Y), которые задают график желаемой функции. Кроме того, зафиксирована процедура Оракул (oracle), которая определяет, соответствуют ли выходные данные – Yв (вычисленные по входным данным – X) желаемым результатам – Y, т.е. принадлежит ли каждая вычисленная точка (X,Yв) графику желаемой функции (X,Y).

*Оракул* дает заключение о факте появления неправильной пары (X,Yв) и ничего не говорит о том, каким образом она была вычислена или каков правильный *алгоритм* – он только сравнивает вычисленные и желаемые результаты. *Оракулом* может быть даже Заказчик или программист, производящий соот-

ветствующие вычисления в уме, поскольку *Оракулу* нужен какой-либо альтернативный способ получения функции (X,Y) для вычисления эталонных значений Y.

*Пример сравнения словесного описания пункта спецификации с результатом выполнения фрагмента.*

Пункт спецификации: "Метод Power должен принимать входные параметры: x – целое число, возводимое в степень, и n – неотрицательный порядок степени. Метод должен возвращать вычисленное значение $x^n$".

Выполняем метод со следующими параметрами: Power(2,2).

Проверка результата выполнения возможна, когда результат вычисления заранее известен – 4. Если результат выполнения $2^2 = 4$, то он соответствует спецификации.

В процессе тестирования Оракул последовательно получает элементы множества (X,Y) и соответствующие им результаты вычислений (X,Yв) для идентификации фактов несовпадений (test incident).

При выявлении (X,Y) ≠(X,Yв) запускается процедура исправления ошибки, которая заключается во внимательном анализе (просмотре) протокола промежуточных вычислений, приведших к (X,Yв), с помощью следующих *методов*:

1) "Выполнение программы в уме" (deskchecking);

2) вставка операторов протоколирования (печати) промежуточных результатов (logging).

Пример вставки операторов протоколирования промежуточных результатов кода представлен на рисунках 5, 6.

Можно выводить промежуточные значения переменных при выполнении программы на указанных рисунках. Этот *метод* относится к наиболее популярным средствам автоматизации *отладки*. В настоящее время он известен как *метод* внедрения «агентов» в текст отлаживаемой программы.

```
// Метод вычисляет неотрицательную
// степень n числа x
static public double Power(double x, int n)
{
double z=1;
for (inti=1;n>=i;i++)
  {
    z = z*x;
Console.WriteLine("i = {0} z = {1}",
i, z);
}
return z;
}
```

Рисунок 5. Исходный текст метода Power со вставкой оператора протоколирования на C#

```
double Power(double x, int n)
{
double z=1;
inti;
for (i=1;n>=i;i++)
  {
    z = z*x;
printf("i = %d z = %f\n",i,z);
}
return z;
}
```

Рисунок 6. Исходный текст метода Power со вставкой оператора протоколирования на C++

*Пошаговое выполнение программы (single-step running)*

При пошаговом выполнении программы код выполняется строчка за строчкой. В среде MicrosoftVisual Studio.NET возможны следующие команды пошагового выполнения:

1) StepInto – если выполняемая строчка кода содержит вызов функции, процедуры или метода, то происходит вызов, и программа останавливается на первой строчке вызываемой функции, процедуры или метода;

2) StepOver – если выполняемая строчка кода содержит вызов функции, процедуры или метода, то происходит вызов и выполнение всей функции и программа останавливается на первой строчке после вызываемой функции;

3) StepOut – предназначена для выхода из функции в вызывающую функцию. Эта команда продолжит выполнение функции и остановит выполнение на первой строчке после вызываемой функции.

*Выполнение программ с заказанными остановками (breakpoints), анализом трасс (traces) или состояний памяти – дампов (dump)*

***Контрольная точка (breakpoint)*** – точка программы, которая при ее до-

стижении посылает отладчику сигнал. По этому сигналу либо временно приостанавливается выполнение отлаживаемой программы, либо запускается программа «агент», фиксирующая состояние заранее определенных переменных или областей в данный момент.

Когда выполнение в контрольной точке приостанавливается, отлаживаемая программа переходит в режим останова (breakmode). Вход в режим останова не прерывает и не заканчивает выполнение программы и позволяет анализировать состояние отдельных переменных или структур данных. Возврат из режима breakmode в режим выполнения может произойти в любой момент по желанию пользователя.

Когда в контрольной точке вызывается программа «агент», она тоже приостанавливает выполнение отлаживаемой программы, но только на время, необходимое для фиксации состояния выбранных переменных или структур данных в специальном электронном журнале – Log-файле, после чего происходит автоматический возврат в режим исполнения.

**Трасса** – это «сохраненный *путь»* на *управляющем графе* программы, т.е. зафиксированные в журнале записи о состояниях переменных в заданных точках в ходе выполнения программы.

Например, на рисунке 7 условно изображен *управляющий граф* некоторой программы. Трасса, проходящая через вершины 0-1-3-4-5, зафиксирована в таблице 1. Строки таблицы отображают вершины *управляющего графа* программы, или *breakpoints*, в которых фиксировались текущие значения заказанных пользователем переменных.

***Управляющий граф программы (УГП)*** – граф G(V,A), где V(V1,… Vm) – множество вершин (операторов), A(A1,… An) – множество дуг (управлений), соединяющих операторы-вершины.

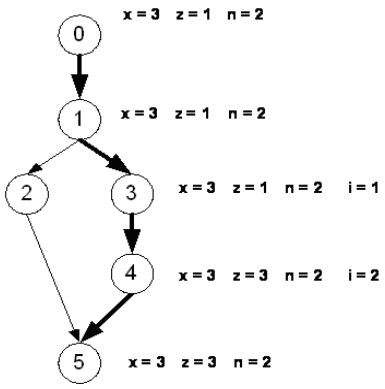


Рисунок 7. Управляющий граф программы

***Ветвь*** – *путь* (V1, V2, … Vk), где V1 – либо первый, либо условный оператор программы, Vk – либо условный оператор, либо оператор выхода из программы, а все остальные операторы – безусловные, например: (3,4) (4,5,6,4) (4,7). *Пути*, различающиеся хотя бы числом прохождений цикла, – разные *пути*, поэтому число *путей* в программе может быть не ограничено. *Ветви* – линейные участки программы, их конечное число.

***Путь*** – последовательность вершин и дуг УГП, в которой любая дуга выходит из вершины Vi и приходит в вершину Vj, например: (3,4,7), (3,4,5,6,4,5,6), (3,4), (3,4,5,6). Существуют реализуемые и нереализуемые *пути* в программе, в нереализуемые *пути* в обычных условиях попасть нельзя.

Таблица 1. Трасса, проходящая через вершины 0-1-3-4-5

| № вершины-оператора | Значение переменной x | Значение переменной z | Значение переменной n | Значение переменной i |
|---|---|---|---|---|
| 0 | 3 | 1 | 2 | не зафиксировано |
| 1 | 3 | 1 | 2 | не зафиксировано |
| 3 | 3 | 1 | 2 | 1 |
| 4 | 3 | 3 | 2 | 2 |
| 5 | 3 | 3 | 2 | не зафиксировано |

***Дамп*** – область памяти, состояние которой фиксируется в контрольной точке в виде единого массива или нескольких связанных массивов. При анализе, который осуществляется после выполнения трассы в режиме off-line, состояния *дампа* структурируются, и выделенные области или поля сравниваются с состояниями, предусмотренными спецификацией. Например, при моделировании поведения управляющих программ контроллеров в виде *дампа* фиксируются области общих и специальных регистров или целые области оперативной памяти, состояния которой определяет алгоритм управления внешней средой.

*Реверсивное (обратное) выполнение (reversibleexecution)*

Обратное выполнение программы возможно при условии сохранения на каждом шаге программы всех значений переменных или состояний программы для соответствующей трассы. Тогда, поднимаясь от конечной точки трассы к любой другой, можно по шагам произвести вычисления состояний, двигаясь от следствия к причине, от состояний на выходе преобразователя данных к состояниям на его входе. Естественно, такие возможности мы получаем в режиме off-line анализа при фиксации в Log – файле всей истории выполнения трассы.

```
// Метод вычисляет неотрицательную
// степень n числа x
static public double PowerNonNeg(double x,
int n)
{
double z=1;
Console.WriteLine("x={0} z={1} n={2}",
x,z,n);
if (n>0)

{
Console.WriteLine("x={0} z={1} n={2}",
x,z,n);
for (inti=1;n>=i;i++)
   {
     z = z*x;
Console.WriteLine(
          "x={0} z={1} n={2}" +
" i={3}",x,z,n,i);
   }
  }
elseConsole.WriteLine(
"Ошибка ! Степень" +
" числа n должна быть больше 0.");
return z;
}
```

Рисунок 8. Исходный текст метода Power с фиксированием значении всех переменных на C#

Пример обратного выполнения для программы вычисления степени числа x представлен на рисунках 8, 9. В этой программе фиксируются значения всех переменных после выполнения каждого оператора.

```
{
double z=1;
inti;
printf("x=%f z=%f n=%d\n",x,z,n);
if (n>0)
 {
printf("x=%f z=%f n=%d\n",x,z,n);

for (i=1;n>=i;i++)
   {
     z = z*x;
printf("x=%f z=%f n=%d i=%d\n",
x,z,n,i);
   }
 }
elseprintf(
"Ошибка ! Степень "
"числа n должна быть больше 0.\n");
return z;
}
```

Рисунок 9. Исходный текст метода Power с фиксированием значении всех переменных на C++

*Разработка тестов с учетом спецификации*

**Спецификация программы**

- На вход программа принимает два параметра: x – число, n – степень. Результат вычисления выводится на консоль.
- Значения числа и степени должны быть целыми.
- Значения числа, возводимого в степень, должны лежать в диапазоне – [0..999].
- Значения степени должны лежать в диапазоне – [1..100].

Если числа, подаваемые на вход, лежат за пределами указанных диапазонов, то должно выдаваться сообщение об ошибке.

Определим области эквивалентности входных параметров.

Для x – числа, возводимого в степень, определим классы возможных значений:

- x < 0 (ошибочное);
- x > 999 (ошибочное);

- x – не число (ошибочное);
- 0 <= x <= 999 (корректное).

Для n – степени числа:

- n < 1 (ошибочное);
- n > 100 (ошибочное);
- n – не число (ошибочное);
- 1 <= n <= 100 (корректное).

На рисунках 10, 11 приведены коды программ с учетом заданной спецификации.

```
// Метод вычисляет степень n числа x
static public double Power(int x, int n)
{
int z=1;
for (inti=1;n>=i;i++)
  {
    z = z*x;
  }
return z;
}

[STAThread]
static void Main(string[] args)
{
int x;
int n;
try
  {
Console.WriteLine("Enter x:");
    x=Convert.ToInt32(Console.ReadLine());
if ((x>=0) & (x<=999))
    {
Console.WriteLine("Enter n:");
      n=Convert.ToInt32(Console.ReadLine());
if ((n>=1) & (n<=100))
      {
Console.WriteLine("The power n" + " of x is {0}",
Power(x,n));
Console.ReadLine();
      }
else
      {
Console.WriteLine("Error : n " + "must be in
[1..100]");
Console.ReadLine();
      }
    }

else
    {
Console.WriteLine("Error : x " + "must be in
[0..999]");
Console.ReadLine();
    }
  }
catch (Exception e)
  {
Console.WriteLine("Error : Please enter " + "a
numeric argument.");
Console.ReadLine();
  }
}
```

Рисунок 10. Исходный текст метода Power с учетом спецификации на C#

```
#include <stdio.h>
double Power(int x, int n)
{
int z=1;
inti;
for (i=1;n>=i;i++)
  {
    z = z*x;
  }
return z;
}

void main(void)
{
int x;
int n;
printf("Enter x:");
if(scanf("%d",&x))
  {
if ((x>=0) & (x<=999))
   {
printf("Enter n:");
if(scanf("%d",&n)) {
if ((n>=1) & (n<=100))
       {
printf("The power n of x is %f\n", Power(x,n));
       }
else
       {
printf("Error : n must be in [1..100]\n");
       }
     }
else
      {
printf("Error : Please enter a numeric argument\n");
      }
   }
else
   {
printf("Error : x must be in [0..999]\n");
   }
  }
else
  {
printf("Error : Please enter a numeric argument\n");
}
}
```

Рисунок 11. Исходный текст метода Power с учетом спецификации на C++

**Анализ тестовых случаев**

1) Входные значения: (x = 2, n = 3) (покрывают классы 4, 8).

Ожидаемый результат: Thepower n of x is 8.

2) Входные значения: {(x=-1, n=2),(x=1000, n= 5)} (покрывают классы 1, 2).

Ожидаемыйрезультат: Error : x must be in [0..999].

3) Входные значения: {(x = 100, n = 0),(x = 100, n = 200)} (покрывают классы 5,6).

Ожидаемыйрезультат: Error : n must be in [1..100].

4) Входные значения: (x = *ADS* n = ASD) (покрывают классы эквивалентности 3, 7).

Ожидаемыйрезультат: Error : Please enter a numeric argument.

5) Проверка на граничные значения:

Входные значения: (x = 999 n = 1).

Ожидаемый результат: Thepower n of x is 999.

Входные значения: x = 0 n = 100.

Ожидаемый результат: Thepower n of x is 0.

**Выполнение тестовых случаев**

Запустим программу с заданными значениями аргументов.

**Оценка результатов выполнения программы на тестах**

В процессе *тестирования Оракул* последовательно получает элементы множества (X,Y) и соответствующие им результаты вычислений YB. В процессе *тестирования* производится оценка результатов выполнения путем сравнения получаемого результата с ожидаемым.

## 1.4. Этапы тестирования

Процесс тестирования можно разделить на 6 основных этапов (рис. 12), которые проходит тестировщик в рамках жизненного цикла тестирования программы.

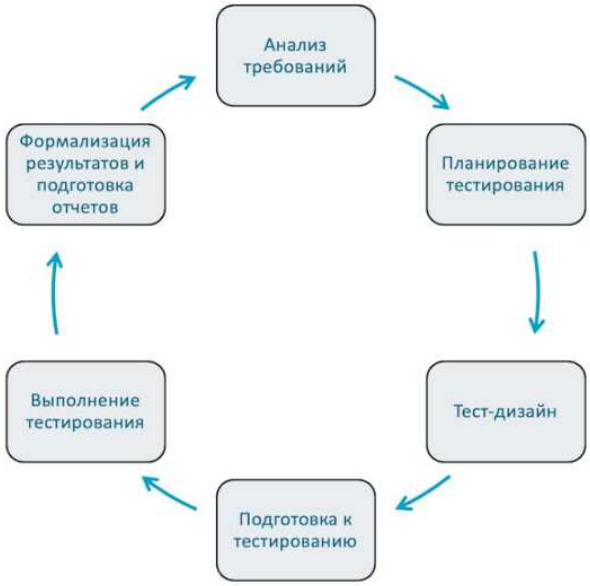


Рисунок 12. Этапы тестирования

### 1. Анализ требований

В классическом подходе к тестированию анализ требований представляет собой активности по исследованию новых изменений, анализу документации к ним, выполнению статического тестирования требований при необходимости.

Как результат, тестировщик получает понимание об изменениях, которые планируются к разработке и внедрению. Чаще всего данный этап начинается после завершения согласования технических требований и спецификаций аналитиком команлы.

Результатом данного этапа является фактически сформированный чеклист тестовых требований, который предстоит проверить тестировщику. Это может быть как формализованный документ, так и просто знания в голове конкретного специалиста по тестированию.

**2. Планирование тестирования**

На данном этапе тестировщик определяется с целями тестирования, необходимыми видами тестирования, требованиями к тестовому окружению (тестовой среде), определяет критерии начала и завершения каждого вида тестирования, оценивает предварительные сроки тестирования на основе полученных ранее тестовых требований, идентифицирует потенциальные риски и совместно с другими участниками команды прорабатывает варианты их минимизации, а также любые иные специфические требования к тестированию, которые могут возникнуть в результате выполнения тестов.

**3. Тест-дизайн (или разработка тестов)**

В классическом подходе этот этап может занимать достаточно продолжительное время, особенно если речь идет о большом проекте. Тестировщик первым делом составляет чек-лист проверок, формирует структуру тест-кейсов и затем приступает к их наполнению шагами, различными условиями, а также определению тестовых данных для их успешного выполнения. После завершения создания кейс-тестов их могут отправить на ревью другому тестировщику, а также согласовать с заказчиками возможные изменения. Кроме того, на данном этапе могут определяться тест-кейсы, которые будут подлежать дальнейшей автоматизации.

**4. Подготовка к тестированию**

На этом этапе происходят следующие действия:

- создание тестового окружения с необходимой конфигурацией;
- подготовка тестовых данных для тестирования;
- доработка скриптов автоматизации тестирования.

В качестве результата тестировщик получает работоспособную тестовую среду с тестовыми данными и установленным приложением для тестирования.

**5. Выполнение тестирования**

Классический подход предполагает выполнение последовательно нескольких типов тестирования до момента получения положительного заключения и успешного прохождения тестов. Сначала выполняется интеграционное или системное тестирование, что зависит от специфики тестирования в конкретной компании. После завершения системного тестирования выполняются регрессионное тестирование и приемочное тестирование.

Результатами данного этапа можно считать успешно пройденные тест-кейсы или проверки по чек-листу.

**6. Формализация результатов и подготовка отчета**

Классический подход к разработке предполагает создание формальных отчетов по результатам тестирования изменений, которые могут включать в себя общие результаты тестирования, открытые проблемы и дефекты, а также рекомендации для сопровождения по установке изменений на продуктивную среду. Этот отчет чаще всего может быть реализован в виде документа или в тексте письма e-mail.

## 1.5. Проблемы тестирования

Рассмотрим два примера тестирования.

1)) Пусть программа H(x:int, y:int) реализована в машине с 64 разрядными словами, тогда мощность множества тестов ||(X,Y)||=2**128

Это означает, что компьютеру, работающему на частоте 1Ггц, для прогона этого набора тестов (при условии, что один тест выполняется за 100 команд) потребуется ~ 3К лет.

2) На рисунке 13 приведен фрагмент схемы программы управления схватом робота, где интервал между моментами срабатывания схвата не определен.

Отсюда вывод:

*Тестирование* программы на всех входных значениях невозможно.

Невозможно *тестирование* и на всех *путях*.


Рисунок 13. Тестовая последовательность сигналов датчика схвата

Следовательно, надо отбирать конечный *набор тестов*, позволяющий проверить программу **на основе наших интуитивных представлений.**

**Требование к тестам** – программа на любом из них должна останавливаться, т.е. не зацикливаться.

**Можно ли заранее гарантировать останов на любом тесте?**

В теории алгоритмов доказано, что не существует общего метода для решения этого вопроса, а также вопроса, достигнет ли программа на данном тесте заранее фиксированного оператора.

Задача о *выборе конечного набора тестов* (X,Y) для проверки программы в общем случае неразрешима.

Поэтому для решения практических задач остается искать частные случаи решения этой задачи.

### 1.6. Требования к идеальному критерию тестирования

1. Критерий должен быть достаточным, т.е. показывать, когда некоторое конечное множество тестов достаточно для тестирования данной программы.

2. Критерий должен быть полным, т.е. в случае ошибки должен существовать тест из множества тестов, удовлетворяющих критерию, который раскрывает ошибку.

3. Критерий должен быть надежным, т.е. любые два множества тестов, удовлетворяющих ему, одновременно должны раскрывать или не раскрывать ошибки программы.

4. Критерий должен быть легко проверяемым, например, вычисляемым на тестах.

Для нетривиальных классов программ в общем случае не существует полного и надежного критерия, зависящего от программ или спецификаций.

Поэтому мы стремимся к идеальному общему критерию через реальные частные.

### 1.7. Классы критериев

К классам критериев тестирования относятся:

1. *Структурные критерии* используют информацию о структуре программы (критерии так называемого «белого ящика»).

2. *Функциональные критерии* формулируются в описании требований к программному изделию (критерии так называемого «черного ящика»).

3. Критерии *стохастического тестирования* формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы.

4. *Мутационные критерии* ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.

**Структурные критерии (класс I)**

Структурные критерии используют модель программы в виде «белого ящика», что предполагает знание исходного текста программы или спецификации программы в виде потокового графа управления. Структурная информация понятна и доступна разработчикам подсистем и модулей приложения, поэтому данный класс критериев часто используется на этапах модульного и интеграционного тестирования (Unittesting, Integrationtesting).

Структурные критерии базируются на основных элементах УГП, операторах, ветвях и путях. Условие критерия тестирования команд (**критерий С0**) – набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза. Это слабый критерий, он, как правило, используется в больших программных системах, где другие критерии применить невозможно.

Условие критерия тестирования ветвей (**критерий С1**) – набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза. Это достаточно сильный и при этом экономичный критерий, поскольку множество ветвей в тестируемом приложении конечно и не так уж велико. Данный критерий часто используется в системах автоматизации тестирования.

Условие критерия тестирования путей (**критерий С2**) – набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто – 2, или числом классов выходных путей).

На рисунках 14, 15 приведен пример простой программы. Рассмотрим условия ее тестирования в соответствии со структурными критериями.

```
1   public void Method (ref int x)
{
2       if (x>17)
3           x = 17-x;
4       if (x==-13)
5           x = 0;
6   }
```

Рисунок 14. Пример простой программы на C# для тестирования по структурным критериям

```
void Method (int *x)
{
    if (*x>17)
        *x = 17-*x;
    if (*x==-13)
        *x = 0;
}
```

Рисунок 15. Пример простой программы на C++ для тестирования по структурным критериям

Тестовый набор из одного теста, удовлетворяет критерию команд (C0):

- (X,Y)={(xвх=30, xвых=0)} покрывает все операторы трассы 1-2-3-4-5-6.

Тестовый набор из двух тестов, удовлетворяет критерию ветвей (C1):

- (X,Y)={(30,0), (17,17)} добавляет 1 тест к множеству тестов для C0 и трассу 1-2-4-6. Трасса 1-2-3-4-5-6 проходит через все ветви достижимые в операторах if при условии true, а трасса 1-2-4-6 через все ветви, достижимые в операторах if при условии false.

Тестовый набор из четырех тестов, удовлетворяет критерию путей (C2):

- (X,Y)={(30,0), (17,17), (-13,0), (21,-4)}.

Набор условий для двух операторов if с метками 2 и 4 приведен в таблице 2.

Таблица 2. Условия операторов if

| | (30,0) | (17,17) | (-13,0) | (21,-4) |
|---|---|---|---|---|
| 2 if (x>17) | > | ≤ | ≤ | > |
| 4 if (x==-13) | ≠ | ≠ | = | ≠ |

Критерий путей С2 проверяет программу более тщательно, чем критерии С1, однако даже если он удовлетворен, нет оснований утверждать, что программа реализована в соответствии со спецификацией.

Например, если спецификация задает условие, что |x|<=100, невыполнимость которого можно подтвердить на тесте (-177,-177). Действительно, операторы 3 и 4 на тесте (-177,-177) не изменят величину x=-177, и результат не будет соответствовать спецификации.

Структурные критерии не проверяют соответствие спецификации, если оно не отражено в структуре программы. Поэтому при успешном тестировании программы по критерию С2 можно не заметить ошибку, связанную с невыполнением некоторых условий спецификации требований.

**Функциональные критерии (класс II)**

*Функциональный критерий* – важнейший для программной индустрии критерий тестирования. Он обеспечивает, прежде всего, *контроль* степени выполнения требований заказчика в программном продукте. Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением. При *функциональном тестировании* преимущественно используется модель «черного ящика».

Проблема *функционального тестирования* – это, прежде всего, трудоемкость; дело в том, что документы, фиксирующие требования к программному изделию (*Software* requirement *specification*, *Functionalspecification* и т.п.), как правило, достаточно объемны, тем не менее, соответствующая проверка должна быть всеобъемлющей.

Ниже приведены частные виды функциональных критериев.

**Тестирование пунктов спецификации** – набор тестов в совокупности должен обеспечить проверку каждого тестируемого пункта не менее одного раза.

Спецификация требований может содержать сотни и тысячи пунктов требований к программному продукту и каждое из этих требований при тестирова-

нии должно быть проверено в соответствии с критерием не менее чем одним тестом.

**Тестирование классов входных данных** – набор тестов в совокупности должен обеспечить проверку представителя каждого класса входных данных не менее одного раза.

Следует заметить, что перебирая в соответствии с критерием величины входных переменных (например, различные файлы – источники входных данных), мы вынуждены применять мощные тестовые наборы.

Действительно, наряду с ограничениями на величины входных данных, существуют ограничения на величины входных данных во всевозможных комбинациях, в том числе проверка реакций системы на появление ошибок в значениях или структурах входных данных. Учет этого многообразия – процесс трудоемкий, что создает сложности для применения критерия.

**Тестирование правил** – набор тестов в совокупности должен обеспечить проверку каждого правила, если входные и выходные значения описываются набором правил некоторой грамматики.

Следует заметить, что грамматика должна быть достаточно простой, чтобы трудоемкость разработки соответствующего набора тестов была реальной (вписывалась в сроки и штат специалистов, выделенных для реализации *фазы тестирования*).

**Тестирование классов выходных данных** – набор тестов в совокупности должен обеспечить проверку представителя каждого выходного класса, при условии, что выходные результаты заранее расклассифицированы, причем отдельные классы результатов учитывают, в том числе, ограничения на ресурсы или на время (timeout).

При создании тестов классы выходных данных сопоставляются с режимами использования тестируемого компонента или подсистемы, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов.

**Тестирование функций** – набор тестов в совокупности должен обеспечить проверку каждого действия, реализуемого тестируемым модулем, не менее одного раза.

Очень популярный на практике критерий, который, однако, не обеспечивает покрытия части функциональности тестируемого компонента, связанной со структурными и поведенческими свойствами, описание которых не сосредоточено в отдельных функциях (т.е. описание рассредоточено по компоненту).

Критерий тестирования функций объединяет отчасти особенности структурных и функциональных критериев. Он базируется на модели «полупрозрачного ящика», где явно указаны не только входы и выходы тестируемого компонента, но также состав и структура используемых методов (функций, процедур) и классов.

**Комбинированные критерии для программ и спецификаций** – набор тестов в совокупности должен обеспечить проверку всех комбинаций непротиворечивых условий программ и спецификаций не менее одного раза.

При этом все комбинации непротиворечивых условий надо подтвердить, а условия противоречий следует обнаружить и ликвидировать.

Рассмотрим пример применения функциональных критериев тестирования для разработки набора тестов по критерию классов входных данных.

Пусть для решения задачи тестирования системы «Система управления автоматизированным комплексом хранения подшипников» был разработан следующий фрагмент спецификации требований:

**1.** Произвести опрос статуса склада (вызвать функцию GetStoreStat ). Добавить в журнал сообщений запись "СИСТЕМА : Запрошен статус СКЛАДА". В зависимости от полученного значения произвести следующие действия.

Полученный статус склада = 32. В приемную ячейку склада поступил подшипник. Система должна:

1. Добавить в журнал сообщений запись "СКЛАД : Статус СКЛАДА = 32".

2. Получить параметры поступившего подшипника с терминала подшипника (должна быть вызвана функция GetRollerPar ).

3. Добавить в журнал сообщений запись "СИСТЕМА: Запрошены параметры подшипника".

4. В зависимости от возвращенного функцией GetRollerPar значения должны быть выполнены следующие действия:

Таблица 3. Действия и результаты функции GetRollerPar

| Значение, возвращенное функцией GetRollerPar | Действия системы |
|---|---|
| ... | ... |
| 0 | 1. Добавить на первое место команду GetR - "ПОЛУЧИТЬ ИЗ ПРИЕМНИКА В ЯЧЕЙКУ"<br>2. Добавить в журнал сообщений запись "ТЕРМИНАЛ ПОДШИПНИКА: 0 - параметры возвращены <Номер_группы>" |
| 1 | Добавить в журнал сообщений запись "ТЕРМИНАЛ ПОДШИПНИКА: 1 - нет данных" |
| ... | ... |

Произвести опрос терминала оси (вызвать функцию получения сообщения от терминала – GetAxlePar ). В журнал сообщений должно быть добавлено сообщение "СИСТЕМА : Запрошены параметры оси". В зависимости от возвращенного функцией GetAxlePar значения должны быть выполнены следующие действия (таблица 4).

Таблица 4. Действия по результатам функции GetRollerPar

| Значение, возвращенное функцией GetAxlePar | Действия системы |
|---|---|
| ... | ... |
| 1 | Добавить в журнал сообщений запись "ТЕРМИНАЛ ОСИ: 1 - нет данных" |
| ... | ... |

Определим классы входных данных для параметра – статус склада:

- Статус склада = 0 (правильный).
- Статус склада = 4 (правильный).

- Статус склада = 16 (правильный).
- Статус склада = 32 (правильный).
- Статус склада = любое другое значение (ошибочный).

Теперь рассмотрим тестовые случаи:

1) Тестовый случай 1 (покрывает класс 4):

- Состояние окружения (входные данные – X ):
- Статус склада – 32.
- ...
- Ожидаемая последовательность событий (выходные данные – Y ):
- Система запрашивает статус склада (вызов функции GetStoreStat ) и получает 32.

2) Тестовый случай 2 (покрывает класс 5):

- Состояние окружения (входные данные – X ):
- Статус склада – 12dfga.
- ...
- Ожидаемая последовательность событий (выходные данные – Y ):
- Система запрашивает статус склада (вызов функции GetStoreStat ) и согласно пункту спецификации при ошибочном значении статуса склада в журнал добавляется сообщение "СКЛАД : ОШИБКА : Неопределенный статус".

**Стохастические критерии (класс III)**

Стохастическое тестирование применяется при тестировании сложных программных комплексов, когда набор детерминированных тестов (X,Y) имеет громадную мощность. В случаях, когда подобный набор невозможно разработать и исполнить на фазе тестирования, можно применить следующую методику.

1) Разработать программы-имитаторы случайных последовательностей входных сигналов {x}.

2) Вычислить независимым способом значения {y} для соответствующих входных сигналов {x} и получить тестовый набор (X,Y).

3) Протестировать приложение на тестовом наборе (X,Y), используя два способа контроля результатов:

1. Детерминированный контроль – проверка соответствия вычисленного значения значению y, полученному в результате прогона теста на наборе {x} – случайной последовательности входных сигналов, сгенерированной имитатором.

2. Стохастический контроль – проверка соответствия множества значений {yв}, полученного в результате прогона тестов на наборе входных значений {x}, заранее известному распределению результатов F(Y).

В этом случае множество Y неизвестно (его вычисление невозможно), но известен закон распределения данного множества.

**Критерии *стохастического* тестирования**

Статистические методы окончания тестирования – стохастические методы принятия решений о совпадении гипотез о распределении случайных величин. К ним принадлежат широко известные: метод Стьюдента ( St ), метод Хи-квадрат и т.п.

Метод *оценки скорости выявления ошибок* основан на модели скорости выявления ошибок, согласно которой тестирование прекращается, если оцененный интервал времени между текущей ошибкой и следующей слишком велик для *фазы тестирования* приложения.

При формализации модели скорости выявления ошибок (рис. 16) использовались следующие обозначения:

- N – исходное число ошибок в программном комплексе перед тестированием,
- C – константа снижения скорости выявления ошибок за счет нахождения очередной ошибки,

- $t_1, t_2, \ldots t_n$ – *кортеж* возрастающих интервалов обнаружения последовательности из n ошибок,
- T – время выявления n ошибок.

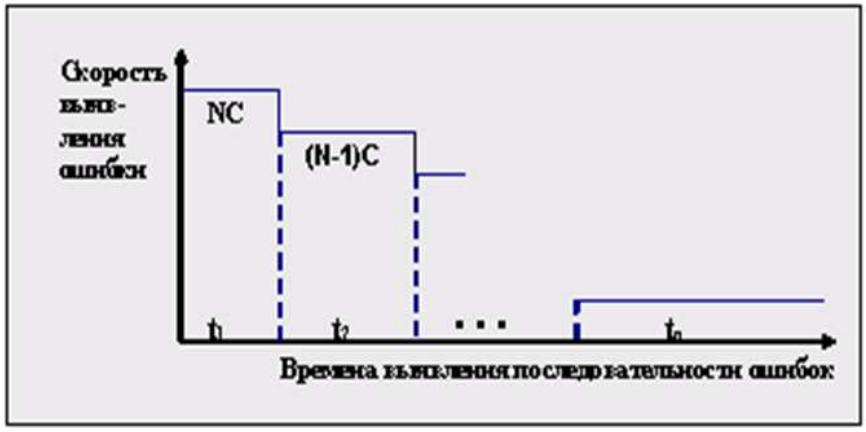


Рисунок 16. Зависимость скорости выявления ошибок от времени выявления

Если допустить, что за время T выявлено n ошибок, то справедливо соотношение (1), утверждающее, что произведение скорости выявления i ошибки и времени выявления i ошибки есть 1 по определению:

$$(N-i+1)\cdot C\cdot t_i = 1 \qquad (1)$$

В этом предположении справедливо соотношение (2) для n ошибок:

$$N\cdot C\cdot t_1 + (N-1)\cdot C\cdot t_2 + \ldots + (N-n+1)\cdot C\cdot t_n = n\cdot N\cdot C\cdot (t_1+t_2+\ldots+t_n) - C\cdot\sum(i-1)\cdot t_i = \\ = n\cdot N\cdot C\cdot T - C\cdot\sum(i-1)\cdot t_i = n \qquad (2)$$

Если из (1) определить $t_i$ и просуммировать от 1 до n, то придем к соотношению (3) для времени T выявления n ошибок:

$$\sum\frac{1}{N-i+1} = T\cdot C \qquad (3)$$

Если из (2) выразить C, приходим к соотношению (4):

$$C = \frac{n}{NT - \sum(i-1)\cdot t_i} \qquad (4)$$

Наконец, подставляя C в (3), получаем окончательное соотношение (5), удобное для оценок:

$$\sum \frac{1}{N-i+1} = \frac{n}{N - \frac{1}{T} \cdot \sum (i-1) \cdot t_i} \qquad (5)$$

Если оценить величину N приблизительно, используя известные методы оценки числа ошибок в программе или данные о *плотности ошибок* для проектов рассматриваемого класса из исторической *базы данных* проектов, и, кроме того, использовать текущие данные об *интервалах между ошибками* $t_1, t_2 \ldots t_n$, полученные на *фазе тестирования*, то, подставляя эти данные в (5), можно получить оценку $t_{n+1}$ – временного интервала, необходимого для нахождения и исправления очередной ошибки.

Если $t_{n+1} > Td$ – допустимого времени тестирования проекта, то тестирование заканчиваем, в противном случае продолжаем *поиск* ошибок.

Наблюдая последовательность интервалов ошибок $t_1, t_2 \ldots t_n$ и время, потраченное на выявление n ошибок $T = \sum t_i$, можно прогнозировать *интервал* времени до следующей ошибки и уточнять в соответствии с (4) величину C.

**Мутационный критерий (класс IV)**

Предлагается подход, позволяющий на основе мелких ошибок (*перестановка* местами максимальных значений индексов в описании массивов, ошибки в знаках арифметических операций, занижение или завышение границы *цикла* на 1 и т.п.) оценить общее число ошибок, оставшихся в программе.

Подход базируется на следующих понятиях:

- ***Мутации*** – мелкие ошибки в программе.
- ***Мутанты*** – программы, отличающиеся друг от друга *мутациями*.
- **Метод мутационного тестирования** – в разрабатываемую программу P вносят *мутации*, т.е. искусственно создают программы-*мутанты* P1, P2... Затем *программа* P и ее *мутанты* тестируются на одном и том же наборе тестов (X,Y). Если на наборе (X,Y) подтверждается *правильность программы* P и, кроме того, выявляются все внесенные в программы-*мутанты* ошибки, то **набор тестов (X,Y) соответствует** *мутационному критерию*, а тестируемая *программа* объявляется **правильной**. Если

некоторые *мутанты* не выявили всех *мутаций*, то надо расширять набор тестов (X,Y) и продолжать тестирование.

**Пример применения мутационного критерия**

Тестируемая программа P приведена на рисунке 17. Для нее создается две программы-мутанты P1 и P2.

```
doublePowerNonNeg(double x, int n)
{
double z=1;
int i;
if (n>0)
   {
for (i=1;n-1>=i;i++)
      {
         z = z*x;
}
   }
elseprintf(
     "Ошибка ! Степень числа n должна
     быть больше 0.\n");
return z;
}
```

Рисунок 17. Основная программа P

В P1 изменено начальное значение переменной z с 1 на 2 (рис. 18).

```
double PowerMutant1(double x, int n)
{
double z=2;
int i;
if (n>0)
   {
for (i=1;n>=i;i++)
      {
         z = z*x;
}
   }
elseprintf("Ошибка ! Степень числа n должна быть
больше 0.\n");
return z;
}
```

Рисунок 18. Программа-мутант P1

В программе P2 изменено начальное значение переменной i с 1 на 0 и граничное значение индекса цикла с n на n-1 (рис. 19).

При запуске тестов (X,Y) = {(x=2, n=3, y=8), (x=999, n=1, y=999), (x=0, n=100, y=0)} выявляются все ошибки в программах-мутантах и ошибка в основной программе, где в условии цикла вместо n стоит n-1.

```
double PowerMutant2(double x, int n)
{
double z=1;
int i;
if (n>0)
    {
for (i=0;n-1>=i;i++)
        {
            z = z*x;
}
    }
elseprintf("Ошибка ! Степень числа n должна быть
больше 0.\n");
return z;
}
```

Рисунок 19. Программа-мутант P2

**Контрольные задания**

1. Составить тестовые задания для программы «Нахождение наибольшего общего делителя при условии, что начальные значения x1 и x2 положительны».

2. Составить тестовые задания для программы «Вычисление факториала неотрицательного целого числа».

3. Составить тестовые задания для программы «Вычисление чисел Фибоначчи».

4. Составить тестовые задания для программы «Поиск заданного символа в строке».

5. Составить тестовые задания для программы «Определить частное q и остаток r от деления x на y».

6. Составить тестовые задания для программы «Поиск значения x в двумерном массиве».

7. Составить тестовые задания для программы «Суммирование элементов массива B[0..n – 1] , где n > 0».

8. Составить тестовые задания для программы «Суммирование комплексных чисел».

9. Составить тестовые задания для программы «Умножение комплексных чисел».

10. Составить тестовые задания для программы «Деление комплексных чисел».

## 2. Классификация методов тестирования

### 2.1. Функциональное тестирование

Функциональное тестирование проверяет системное приложение в отношении функциональных требований с целью обнаружения несоответствия требованиям конечного пользователя. Для большинства программ тестирования программного продукта данный метод тестирования является главным. Его основная задача – оценка того, работает ли приложение в соответствии с предъявляемыми требованиями.

Функциональное тестирование основано на принципе тестирования «черного ящика», оно позволяет получить комбинации входных данных, обеспечивающих полную проверку всех функциональных требований к программе. Программное изделие здесь рассматривается как «черный ящик», чье поведение можно определить только исследованием его входов и соответствующих выходов. При таком подходе желательно иметь:

- набор, образуемый такими входными данными, которые приводят к аномалиям в поведении программы (назовем его IT);
- набор, образуемый такими входными данными, которые демонстрируют дефекты программы (назовем его OT).

Любой способ тестирования «черного ящика» должен:

- выявить такие входные данные, которые с высокой вероятностью принадлежат набору IT;
- сформулировать такие ожидаемые результаты, которые с высокой вероятностью являются элементами набора OT.

Принцип «черного ящика» не альтернативен принципу «белого ящика». Скорее это дополняющий подход, который обнаруживает другой класс ошибок.

Тестирование «черного ящика» обеспечивает поиск следующих категорий ошибок:

- некорректных или отсутствующих функций;
- ошибок интерфейса;

- ошибок во внешних структурах данных или в доступе к внешней базе данных;
- ошибок характеристик (необходимая емкость памяти и т.д.);
- ошибок инициализации и завершения.

Подобные категории ошибок способами «белого ящика» не выявляются.

В отличие от тестирования «белого ящика», которое выполняется на ранней стадии процесса тестирования, тестирование «черного ящика» применяют на поздних стадиях тестирования. При тестировании «черного ящика» пренебрегают управляющей структурой программы. Здесь внимание концентрируется на информационной области определения программной системы.

Техника «черного ящика» ориентирована на решение следующих задач:

- сокращение необходимого количества тестовых вариантов (из-за проверки не статистических, а динамических аспектов системы);
- выявление классов ошибок, а не отдельных ошибок.

Тестирование на основе стратегии черного ящика возможно лишь при наличии установленных открытых интерфейсов, таких как интерфейс пользователя или программный интерфейс приложения (API).

Если тестирование на основе стратегии «белого ящика» исследует внутреннюю работу программы, то методы тестирования «черного ящика» сравнивают поведение приложения с соответствующими требованиями. Кроме того, эти методы обычно направлены на выявление трех основных видов ошибок:

- ❖ функциональности, поддерживаемой программным продуктом;
- ❖ производимых вычислений;
- ❖ допустимого диапазона или области действия значений данных, которые могут быть обработаны программным продуктом.

На этом уровне тестировщики не исследуют внутреннюю работу компонентов программного продукта, тем не менее они проверяются неявно.

Группа тестирования изучает входные и выходные данные программного продукта. В этом ракурсе тестирование с помощью методов черного ящика рас-

сматривается как синоним тестирования на уровне системы, хотя методы «черного ящика» могут также применяться во время модульного или компонентного тестирования.

При тестировании методами «черного ящика» важно участие пользователей, поскольку именно они лучше всего знают, каких результатов следует ожидать от бизнес-функций. Ключом к успешному завершению системного тестирования является корректность данных. Поэтому на фазе создания данных для тестирования крайне важно, чтобы конечные пользователи предоставили как можно больше входных данных.

Тестирование при помощи методов «черного ящика» направлено на получение множеств входных данных, которые наиболее полно проверяют все функциональные требования системы. Это не альтернатива тестированию по методу «белого ящика». Этот тип тестирования нацелен на поиск ошибок, относящихся к целому ряду категорий, среди них:

- неверная или пропущенная функциональность;
- ошибки интерфейса;
- проблемы удобства использования;
- методы тестирования на основе автоматизированных инструментов;
- ошибки в структурах данных или ошибки доступа к внешним базам данных;
- проблемы снижения производительности и другие ошибки производительности;
- ошибки загрузки;
- ошибки многопользовательского доступа;
- ошибки инициализации и завершения;
- проблемы сохранения резервных копий и способности к восстановлению работы;
- проблемы безопасности.

***Методы тестирования на основе стратегии «черного ящика»***

**Эквивалентное разбиение.** Исчерпывающее тестирование входных данных, как правило, неосуществимо. Поэтому следует проводить тестирование с использованием подмножества входных данных. При тестировании ошибок, связанных с выходом за пределы области допустимых значений, применяют три основных типа эквивалентных классов: значения внутри границы диапазона, за границей диапазона и на границе.

**Анализ граничных значений.** Анализ граничных значений можно применить как на структурном, так и на функциональном уровне тестирования. Границы определяют данные трех типов: правильные, неправильные и лежащие на границе. Тестирование границ использует значения, лежащие внутри или на границе (например, крайние точки), и максимальные/минимальные значения (например, длины полей). При таком исследовании всегда должны учитываться значения на единицу больше и меньше граничного. При тестировании за пределами границы используется репрезентативный образец данных, выходящих за границу, т.е. неверные значения.

**Диаграммы причинно-следственных связей**. Составление диаграмм причинно-следственных связей – это метод, дающий четкое представление о логических условиях и соответствующих действиях. Метод предполагает четыре этапа:

1) составление перечня причин (условий ввода) и следствий (действий) для модуля и присвоение идентификатора каждому модулю;
2) разрабатывается диаграмма причинно-следственных связей;
3) диаграмма преобразуется в таблицу решений;
4) установление причин и следствий в процессе чтения спецификации функций.

**Составление наборов тестовых данных для функционального тестирования. Стратегия «черного ящика»**

***Пример.*** Пусть необходимо выполнить тестирование программы, определяющей точку пересечения двух прямых на плоскости. При этом она должна определять параллельность прямой одной из осей координат.

В основе программы лежит решение системы линейных уравнений:

Ax + By = C, Dx + Ey = F.

*По методу эквивалентных разбиений* формируем для каждого коэффициента один правильный класс эквивалентности (коэффициент – вещественное число) и один неправильный (коэффициент – не вещественное число). Откуда генерируем 7 тестов:

1) все коэффициенты – вещественные числа (1 тест);

2-7) поочередно каждый из коэффициентов – не вещественное число (6 тестов).

*По методу граничных значений* можно считать, что для исходных данных граничные значения отсутствуют, т.е. коэффициенты – «любые» вещественные числа.

Для результатов получаем, что возможны варианты: единственное решение, прямые сливаются – множество решений, прямые параллельны – отсутствие решений. Следовательно, целесообразно предложить тесты с результатами внутри областей возможных значений результатов:

8) результат – единственное решение ($\delta \neq 0$ (определитель $\delta$=AE-BD));

9) результат – множество решений ($\delta = 0$ и $\delta X = \delta У = 0$, ($\delta X$=CE-BF, $\delta У$=AF-CD));

10) результат – отсутствие решений ($\delta = 0$, но $\delta X \neq 0$ или $\delta У \neq 0$);

и с результатами на границе:

11) $\delta = 0{,}01$;

12) $\delta = -0{,}01$;

13) $\delta = 0$, $\delta X = 0{,}01$, $\delta У = 0$;

14) δ = 0, δ У =-0,01, δ Х = 0.

### 2.2. Структурное тестирование

Структурное тестироавание основывается на стратегии «белого ящика», управляемого логикой программы, и позволяет исследовать внутреннюю структуру программы. При данном тестировании программа рассматривается как объект с известной внутренней структурой. Поэтому такой принцип тестирования называется структурным, или «стеклянным ящиком», или «белым ящиком». Тестировщик (как правило, программист) разрабатывает тесты, основываясь на знании исходного кода, к которому он имеет полный доступ. Главной его идей является правильный выбор тестируемого программного пути. Таким образом, тестирование методом «белого ящика» обладает следующими преимуществами.

a) *Направленность тестирования*. Программист может тестировать программу по частям, разрабатывать специальные тестовые подпрограммы, вызывающие тестируемый модуль, и передают ему интересующие данные. Отдельный модуль гораздо легче протестировать именно тогда, когда он является открытым.

b) *Полный охват кода*. Тестировщик всегда может определить, какие именно фрагменты кода работают в каждом тесте. Он может увидеть какие ветви кода остались не протестированными и может подобрать условия, в которых они будут выполнены.

c) *Управление потоком*. Программист знает, какая функция должна выполняться в программе следующей и каким должно быть её текущее состояние. Чтобы выяснить, правильно ли работает программа, программист может включить в неё отладочные команды, отображающие информацию о ходе выполнения программы.

d) *Отслеживание целостных данных*. Программисту известно, какая часть программы должна изменять каждый элемент данных. Отслеживая состояния данных (к примеру, с помощью отладчика), можно выявить такие ошибки, как

изменение данных не теми модулями, их неверная интерпретация или плохая организация.

e) *Внутренние граничные точки*. В исходном коде видны те граничные точки программы, которые скрыты при отсутствии такого кода. К примеру, для выполнения некоторого действия можно использовать множество различных алгоритмов, и, не заглянув в код, невозможно определить, какой из этих алгоритмов выбрал программист. Другим типичным примером является проблема переполнения буфера для хранения входных данных. Программист сразу может сказать, при каком количестве данных буфер переполнится, и ему при этом не нужно проводить тысячи тестов.

f) *Тестирование, определяемое выбранным алгоритмом*. Для тестирования обработки данных, использующей очень сложные вычислительные алгоритмы, могут понадобиться специальные технологии. В качестве классического примера можно привести сортировку данных. Тестировщику нужно знать точно, какие алгоритмы используются, и обратиться к специальной литературе.

Тестирование «белого ящика» можно рассматривать как часть процесса программирования. Программисты регулярно выполняют это тестирование после написания программы, её модулей и даже отдельных блоков кода.

Существуют четыре стратегии (способа) тестирования методом «белого ящика»:

a) тестирование базового пути (покрытие решений);

b) тестирование условий (покрытие условий);

c) тестирование потоков данных;

d) тестирование циклов.

Тестирование по принципу «белого ящика» характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы. Исчерпывающее тестирование по принципу белого ящика предполагает выполнение каждого пути в программе, но поскольку в программе с циклами выпол-

нение каждого пути обычно нереализуемо, то тестирование всех путей не рассматривается.

Составление наборов тестовых данных для структурного тестирования рассматривается как стратегия «белого (прозрачного) ящика».

***Последовательность составления тестов*** следующая:

1. На основе алгоритма (текста) программы формируется потоковый граф (или граф-схема). Узлы (вершины) потокового графа соответствуют линейным участкам программы, включают один или несколько операторов программы. Дуги (ориентированные ребра) потокового графа отображают поток управления в программе (передачи управления между операторами).

Различают операторные и предикатные узлы. Из операторного узла выходит одна дуга, а из предикатного − две дуги. *Предикатные узлы соответствуют* ***простым*** *условиям в программе.* Составное условие программы (условие, в котором используется одна или несколько булевых операций (OR, AND)) отображается в несколько предикатных узлов. На рисунке 20 показан пример такого преобразования для фрагмента программы: *if a OR b then x else y end if.*

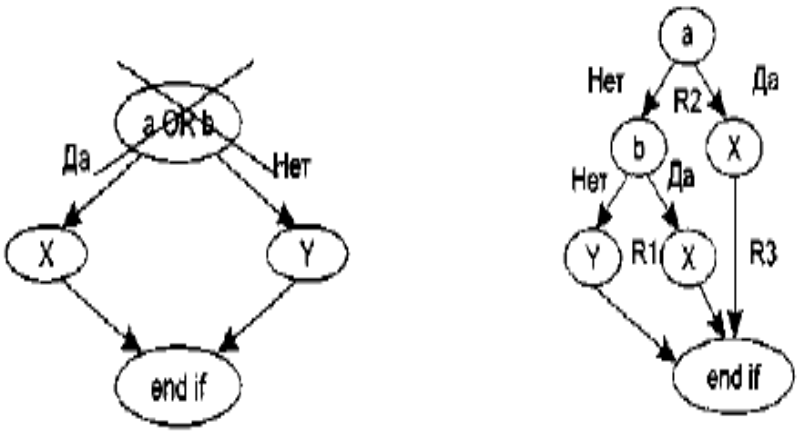


Рисунок 20. Прямое отображение фрагмента программы в потоковый граф и преобразованный потоковый граф

2. Выбирается критерий тестирования. Формирование тестовых наборов для тестирования маршрутов может осуществляться по нескольким критериям: покрытие маршрутов, покрытие операторов, покрытие решений (переходов),

покрытие условий, покрытие решений/условий, комбинаторное покрытие условий, покрытие потоков данных, покрытие циклов.

3. Подготавливаются тестовые варианты, инициирующие выполнение каждого пути. Каждый тестовый вариант формируется в следующем виде:

- Исходные данные (ИД):
- Ожидаемые результаты (ОЖ.РЕЗ.):

Число независимых линейных путей в базовом множестве определяется ***цикломатической сложностью алгоритма***, которая вычисляется одним из трех способов.

1) Цикломатическая сложность V(G) равна количеству регионов потокового графа. Регион – замкнутая область, образованная дугами и узлами. Окружающая граф среда рассматривается как дополнительный регион. Например, показанный на рисунке 20 граф имеет три региона − Rl, R2, R3;

2) Цикломатическая сложность определяется по формуле

V(G) = E-N+2,

где E − количество дуг, N − количество узлов потокового графа;

3) Цикломатическая сложность V(G) = p +1, где p – количество предикатных узлов G.

**Пример**.

Цикломатическая сложность алгоритма на рисунке 20:

1) V(G) = 3 региона;

2) V(G) = 7 дуг - 6 узлов + 2 = 3;

3) V(G) = 2 предикатных узлов + 1 = 3.

Независимые пути:

Путь 1: a - x – end if.

Путь 2: a - b - x – end if.

Путь 3: a - b - y – end if.

### 2.3. Модульное тестирование

*Модульное тестирование* – это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель *модульного тестирования* состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. *Модульное тестирование* проводится по принципу «белого ящика», то есть основывается на знании внутренней структуры программы, и часто включает те или иные методы анализа покрытия кода.

*Модульное тестирование* обычно подразумевает создание вокруг каждого модуля определенной среды, включающей заглушки для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне *модульного тестирования* проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно пропускаются на уровне *модульного тестирования* и выявляются на более поздних стадиях тестирования.

Именно эффективность обнаружения тех или иных типов дефектов должна определять стратегию *модульного тестирования*, то есть расстановку акцентов при определении набора входных значений. У организации, занимающейся разработкой программного обеспечения, как правило, имеется историческая *база данных* (**Repository**) разработок, хранящая конкретные сведения о разработке предыдущих проектов: о версиях и сборках кода (**build**), зафиксированных в процессе разработки продукта, о принятых решениях, допущенных просчетах, ошибках, успехах и т.п.

Проведя *анализ* характеристик прежних проектов, подобных заказанному организации, можно предохранить новую разработку от старых ошибок, например, определив типы дефектов, *поиск* которых наиболее эффективен на различных этапах тестирования.

В данном случае анализируется этап модульного тестирования. Если анализ не дал нужной информации, например, в случае проектов, в которых соответствующие данные не собирались, то основным правилом становится поиск локальных дефектов, у которых код, ресурсы и информация, вовлеченные в дефект, характерны именно для данного модуля. В этом случае на модульном уровне ошибки, связанные, например, с неверным порядком или форматом параметров модуля, могут быть пропущены, поскольку они вовлекают информацию, затрагивающую другие модули, в то время как ошибки в алгоритме обработки параметров довольно легко обнаруживаются.

Являясь по способу исполнения структурным тестированием или тестированием «белого ящика», *модульное тестирование* характеризуется степенью, в которой тесты выполняют или покрывают логику программы (исходный текст). Тесты, связанные со структурным тестированием, строятся по следующим принципам.

- На основе анализа *потока управления*. В этом случае элементы, которые должны быть покрыты при прохождении тестов, определяются на основе *структурных критериев* тестирования C0, C1,C2. К ним относятся вершины, дуги, пути *управляющего графа* программы (УГП), условия, комбинации условий и т.п.
- На основе анализа *потока данных*, когда элементы, которые должны быть покрыты, определяются при помощи *потока данных*, т.е. *информационного графа* программы.

## 2.4. Интеграционное тестирование

Интеграционное тестирование – это тестирование части системы, состоящей из двух и более модулей. Основная задача интеграционного тестирования – поиск дефектов, связанных с ошибками в реализации и интерпретации интерфейсного взаимодействия между модулями.

С технологической точки зрения интеграционное тестирование является количественным развитием модульного, поскольку так же, как и модульное тестирование, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки (Stub) на месте отсутствующих модулей.

Основная разница между модульным и интеграционным тестированием состоит в целях, то есть в типах обнаруживаемых дефектов, которые, в свою очередь, определяют стратегию выбора входных данных и методов анализа. В частности, на уровне интеграционного тестирования часто применяются методы, связанные с покрытием интерфейсов, например, вызовов функций или методов или анализ использования интерфейсных объектов, таких как глобальные ресурсы, средства коммуникаций, предоставляемых операционной системой.

На рисунке 21 приведена структура комплекса программ K, состоящего из оттестированных на этапе *модульного тестирования* модулей M1, M2, M11, M12, M21, M22. Задача, решаемая методом *интеграционного тестирования*, – тестирование межмодульных связей, реализующихся при исполнении программного обеспечения комплекса K. *Интеграционное тестирование* использует модель «белого ящика» на модульном уровне. Поскольку тестировщику текст программы известен с детальностью до вызова всех модулей, входящих в тестируемый комплекс, применение структурных критериев на данном этапе возможно и оправдано.

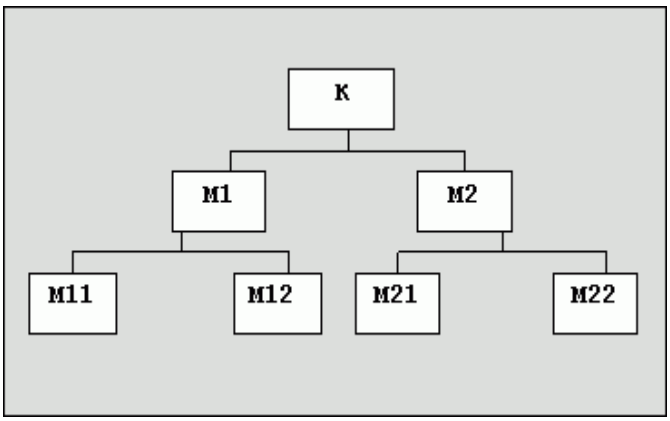


Рисунок 21. Пример структуры комплекса программ

Интеграционное тестирование применяется на этапе сборки модульно отстестированных модулей в единый комплекс. Известны два метода сборки модулей.

- Монолитный, характеризующийся одновременным объединением всех модулей в тестируемый комплекс.
- Инкрементальный, характеризующийся пошаговым (помодульным) наращиванием комплекса программ с пошаговым тестированием собираемого комплекса. В инкрементальном методе выделяют две стратегии добавления модулей:
  - «сверху вниз» и соответствующее ему нисходящее тестирование;
  - «снизу вверх» и соответственно восходящее тестирование.

**Особенности монолитного тестирования** заключаются в следующем: для замены неразработанных к моменту тестирования модулей, кроме самого верхнего (К на рис. 21), необходимо дополнительно разрабатывать **драйверы** (**testdriver**) и/или **заглушки** (**stub**), замещающие отсутствующие на момент сеанса тестирования модули нижних уровней.

Сравнение *монолитного* и инкрементального подхода дает следующее.

- *Монолитное тестирование* требует больших трудозатрат, связанных с дополнительной разработкой драйверов и заглушек и со сложностью идентификации ошибок, проявляющихся в пространстве собранного кода.

- Пошаговое тестирование связано с меньшей трудоемкостью идентификации ошибок за счет постепенного наращивания объема тестируемого кода и соответственно локализации добавленной области тестируемого кода.

*Монолитное тестирование* предоставляет большие возможности распараллеливания работ особенно на начальной *фазе тестирования*.

Особенности *нисходящего тестирования* заключаются в следующем: организация среды для исполняемой очередности вызовов оттестированными модулями тестируемых модулей, постоянная разработка и использование заглушек, организация приоритетного тестирования модулей, содержащих *операции* обмена с окружением, или модулей, критичных для тестируемого алгоритма.

Например, порядок тестирования комплекса K (рис. 21) при *нисходящем тестировании* может быть таким, как показано в примере ниже, где тестовый набор, разработанный для модуля Mi, обозначен как XYi = (X, Y)i.

**Пример**

Возможный порядок тестов при нисходящем тестировании:

- 1) K->XYK
- 2) M1->XY1
- 3) M11->XY11
- 4) M2->XY2
- 5) M22->XY22
- 6) M21->XY21
- 7) M12->XY12

Недостатки *нисходящего тестирования*

- Проблема разработки достаточно «интеллектуальных» заглушек, т.е. заглушек, пригодных к использованию при моделировании различных режимов работы комплекса, необходимых для тестирования.
- Сложность организации и разработки среды для реализации исполнения модулей в нужной последовательности.

- Параллельная разработка модулей верхних и нижних уровней приводит к не всегда эффективной реализации модулей из-за подстройки (специализации) еще не тестированных модулей нижних уровней к уже оттестированным модулям верхних уровней.

*Особенности восходящего тестирования* в организации порядка *сборки* и перехода к тестированию модулей, соответствующему порядку их реализации.

Например, порядок тестирования комплекса K (рис. 21) при *восходящем тестировании* может быть в следующем примере.

**Пример**

Возможный порядок тестов при восходящем тестировании:

- 1) M11->XY11
- 2) M12->XY12
- 3) M1->XY1
- 4) M21->XY21
- 5) M2(M21, Stub(M22))->XY2
- 6) K(M1, M2(M21, Stub(M22)) ->XYK
- 7) M22->XY22
- 8) M2->XY2
- 9) K->XYK

Недостатки *восходящего тестирования*

- Запаздывание проверки концептуальных особенностей тестируемого комплекса.
- Необходимость в разработке и использовании драйверов.

**Временн*а*я классификация методов интеграционного тестирования**

На практике чаще всего в различных частях проекта применяются все рассмотренные в предыдущем разделе методы в совокупности. Каждый модуль тестируют по мере готовности отдельно, а потом включают в уже готовую композицию. Для одних частей тестирование получается нисходящим, для других – восходящим. В связи с этим представляется полезным рассмотреть еще один

тип классификации типов интеграционного тестирования – классификацию по времени интеграции.

В рамках этой классификации выделяют:

- тестирование с поздней интеграцией;
- тестирование с постоянной интеграцией;
- тестирование с регулярной или послойной интеграцией.

*Тестирование с поздней интеграцией* – практически полный аналог монолитного тестирования. Интеграционное тестирование при такой схеме откладывается на как можно более поздние сроки проекта. Этот подход оправдан в том случае, если система является конгломератом слабо связанных между собой модулей, которые взаимодействуют по какому-либо стандартному интерфейсу, определенному вне проекта (например, в случае, если система состоит из отдельных Web-сервисов).

Схематично тестирование с поздней интеграцией может быть изображено в виде цепочки R-C-V-R-C-V-R-C-V-I-R-C-V-R-C-V-I, где R – разработка требований на отдельный модуль, C – разработка программного кода, V – тестирование модуля, I – интеграционное тестирование всего, что было сделано раньше.

*Тестирование с постоянной интеграцией* подразумевает, что, как только разрабатывается новый модуль системы, он сразу же интегрируется со всей остальной системой. При этом тесты для этого модуля проверяют как сугубо его внутреннюю функциональность, так и его взаимодействие с остальными модулями системы.

Таким образом, этот подход совмещает в себе модульное тестирование и интеграционное. Разработки заглушек при таком подходе не требуется, но может потребоваться разработка драйверов. В настоящее время именно этот подход называют *unittesting*, несмотря на то, что в отличие от классического модульного тестирования здесь не проверяется функциональность изолированного модуля. Локализация ошибок межмодульных интерфейсов при таком подходе

несколько затруднена, но все же значительно ниже, чем при монолитном тестировании.

Большая часть таких ошибок выявляется достаточно рано именно за счет частоты интеграции и за счет того, что за одну итерацию тестирования проверяется сравнительно небольшое число межмодульных интерфейсов.

Схематично тестирование с постоянной интеграцией может быть изображено в виде цепочки R-C-I-R-C-I-R-C-I, в которой *фаза тестирования* модуля намеренно опущена и заменена на тестирование интеграции.

При *тестировании с регулярной или послойной интеграцией* интеграционному тестированию подлежат сильно связанные между собой группы модулей (слои), которые затем также интегрируются между собой. Такой вид интеграционного тестирования называют также *иерархическим интеграционным тестированием*, поскольку укрупнение интегрированных частей системы, как правило, происходит по иерархическому принципу. Однако в отличие от нисходящего или восходящего тестирования, направление прохода по иерархии в этом подходе не задано.

*Особенности интеграционного тестирования для объектно-ориентированного программирования*

Объектно-ориентированное *программное обеспечение* является событийно управляемым. Передача управления внутри программы осуществляется не только путем явного указания последовательности обращений одних функций программы к другим, но и путем генерации сообщений различным объектам, разбора сообщений соответствующим обработчиком и передача их объектам, для которых данные сообщения предназначены.

Графовая модель программ (ГМП) в данном случае требует адаптации к требованиям, вводимым объектно-ориентированным подходом к написанию программного обеспечения. При этом происходит переход от модели, описывающей структуру программы, к модели, описывающей поведение программы, что для тестирования можно классифицировать как положительное свойство

данного перехода. Отрицательным аспектом совершаемого перехода для применения рассмотренных ранее моделей является потеря заданных в явном виде связей между модулями программы.

Тестирование объектно-ориентированной программы должно включать те же уровни, что и тестирование процедурной программы – модульное, интеграционное и системное. Внутри класса отдельно взятые методы имеют императивный характер исполнения. Все языки ООП возвращают контроль вызывающему объекту, когда сообщение обработано. Поэтому каждый метод (функция – член класса) должен пройти традиционное модульное тестирование по выбранному критерию C (как правило, C1 ). В соответствии с введенными выше обозначениями, назовем метод Modi, а сложность тестирования – V(Modi,C).

Каждый класс должен быть рассмотрен и как субъект интеграционного тестирования. Интеграция для всех методов класса проводится с использованием инкрементальной стратегии снизу вверх. При этом мы можем переиспользовать тесты для классов-родителей тестируемого класса, что следует из принципа наследования – от базовых классов, не имеющих родителей, к самым верхним уровням классов.

Интеграционное тестирование гарантирует то, что компоненты приложения функционируют корректно, когда собраны вместе. ASP.NET 5 поддерживает интеграционное тестирование с помощью фреймворков для тестирования и встроенного тестового веб хоста, который можно использовать для обработки запросов без нагрузки на сеть.

В отличии от модульного тестирования интеграционные тесты часто запрашивают компоненты инфраструктуры приложения, например, БД, файловую систему, веб запросы и ответы и так далее. На месте этих компонентов юнит тесты используют mock-объекты, а интеграционные тесты должны проверять, хорошо ли эти компоненты работают в системе.

Тестирование логики внутри методов обычно является задачей юнит тестов. А интеграционные тесты входят в игру тогда, когда надо проверить, как приложение работает во фреймворке (например, ASP.NET Core) или с БД. Вам не нужно создавать слишком много интеграционных тестов, чтобы проверить, что вы можете добавить запись или извлечь ее из БД. Вам не нужно тестировать каждое возможное изменение в коде для доступа к данным – вы просто должны убедиться, что приложение работает должным образом.

Для автоматизации интеграционного тестирования применяются ***системы непрерывной интеграции*** (англ. *Continuous Integration System*, CIS). Принцип действия таких систем состоит в следующем:

1. CIS производит мониторинг системы контроля версий;
2. при изменении исходных кодов в репозитории производится обновление локального хранилища;
3. выполняются необходимые проверки и модульные тесты;
4. исходные коды компилируются в готовые выполняемые модули;
5. выполняются тесты интеграционного уровня;
6. генерируется отчет о тестировании.

Таким образом, автоматические интеграционные тесты выполняются сразу же после внесения изменений, что позволяет обнаруживать и устранять ошибки в короткие сроки.

**Спецификация** модуля (или класса) ПП определяет, что этот модуль должен делать, т.е. она описывает допустимые наборы входных данных, подаваемых на вход модуля, включая ограничения на то, как многократные вводы данных должны соотноситься друг с другом, и какие выходные данные соответствуют различным наборам входных данных.

Взаимодействие объектов представляет собой просто *запрос* одного объекта (**отправителя**) на выполнение другим объектом (**получателем**) одной из операций получателя и всех видов обработки, необходимых для завершения этого запроса.

В ситуациях, когда в качестве основы тестирования взаимодействий объектов выбраны только спецификации общедоступных операций, тестирование намного проще, чем когда такой основой служит реализация. Мы ограничимся тестированием общедоступного интерфейса. Такой подход вполне оправдан, поскольку мы полагаем, что классы уже успешно прошли модульное тестирование. Тем не менее, выбор такого подхода отнюдь не означает, что не нужно возвращаться к спецификациям классов, дабы убедиться в том, что тот или иной метод выполнил все необходимые вычисления. Это обусловливает необходимость проверки значений атрибутов внутреннего состояния получателя, в том числе любых агрегированных атрибутов, т.е. атрибутов, которые сами являются объектами. Основное внимание уделяется отбору тестов на основе спецификации каждой операции из общедоступного интерфейса класса.

Взаимодействия неявно предполагаются в спецификации класса, в которой установлены ссылки на другие объекты. Такие объекты (примитивные классы) представляют собой простейшие компоненты системы и, несомненно, играют важную роль при выполнении любой программы. Тем не менее, в объектно-ориентированной программе существует сравнительно небольшое количество примитивных классов, которые реалистично моделируют объекты задачи и все отношения между этими объектами. Обычным явлением для хорошо спроектированных объектно-ориентированных программ является использование непримитивных классов; в этих программах им отводится главенствующая роль.

Выявить такие взаимодействующие классы можно, используя отношения ассоциации (в том числе отношения агрегирования и композиции), представленные на диаграмме классов. Ассоциации такого рода преобразуются в интерфейсы класса, а тот или иной *класс* взаимодействует с другими классами посредством одного или нескольких способов.

**Тип 1**. Общедоступная операция имеет один или большее число формальных параметров объектного типа. Сообщение устанавливает ассоциацию между

получателем и параметром, которая позволяет получателю взаимодействовать с этим параметрическим объектом.

**Тип 2**. Общедоступная операция возвращает значения объектного типа. На *класс* может быть возложена задача создания возвращаемого объекта, либо он может возвращать модифицированный *параметр*.

**Тип 3**. Метод одного класса создает экземпляр другого класса как часть своей реализации.

**Тип 4**. Метод одного класса ссылается на глобальный экземпляр некоторого другого класса. Разумеется, принципы хорошего тона в проектировании рекомендуют минимальное использование глобальных объектов. Если реализация какого-либо класса ссылается на некоторый глобальный *объект*, рассматривайте его как неявный *параметр* в методах, которые на него ссылаются.

### 2.5. Ручное тестирование

Ручное тестирование заключается в выполнении задокументированной процедуры, где описана методика выполнения тестов, задающая порядок тестов и для каждого теста – список значений параметров, который подается на вход, и список результатов, ожидаемых на выходе. Поскольку процедура предназначена для выполнения человеком, в ее описании для краткости могут использоваться некоторые значения по умолчанию, ориентированные на здравый смысл, или ссылки на информацию, хранящуюся в другом документе.

**Пример фрагмента процедуры**

1. Подать на вход три разных целых числа.

2. Запустить тестовое исполнение.

3. Проверить, соответствует ли полученный результат таблице [ссылка на документ 1] с учетом поправок [ссылка на документ 2].

4. Убедиться в понятности и корректности выдаваемой сопроводительной информации.

В приведенной процедуре тестировщик использует два дополнительных документа, а также собственное понимание того, какую сопроводительную информацию считать «понятной и корректной». Успех от использования процедурного подхода достигается в случае однозначного понимания тестировщиком всех пунктов процедуры. Например, в п.1 приведенной процедуры не уточняется, из какого диапазона должны быть заданы три целых числа, и не описывается дополнительно, какие числа считаются «разными».

### 2.6. Автоматизированное тестирование

Попытка автоматизировать приведенный выше тест приводит к созданию скрипта, задающего тестируемому продукту три конкретных числа и перенаправляющего вывод продукта в файл с целью его анализа, а также содержащего конкретное значение желаемого результата, с которым сверяется получаемое при прогоне теста значение. Таким образом, вся необходимая информация должна быть явно помещена в текст (скрипт) теста, что требует дополнительных по сравнению с ручным подходом усилий. Также дополнительных усилий и времени требует создание разборщика вывода (программы согласования форматов представления эталонных значений из теста и вычисляемых при прогоне результатов) и, возможно, создание базы хранения состояний эталонных данных.

**Пример скрипта**

Приведем пример последовательности действий, закладываемых в скрипт.

1. Выдать на консоль имя или номер теста и время его начала.

2. Вызвать продукт с фиксированными параметрами.

3. Перенаправить вывод продукта в файл.

4. Проверить возвращенное продуктом значение. Оно должно быть равно ожидаемому (эталонному) результату, зафиксированному в тесте.

5. Проверить вывод продукта, сохраненный в файле (п.3), на равенство заранее приготовленному эталону.

6. Выдать на консоль результаты теста в виде вердикта PASS/FAIL и в случае FAIL – краткого пояснения, какая именно проверка не прошла.

7. Выдать на консоль время окончания теста.

**Сравнение ручного и автоматизированного тестирования**

Результаты сравнения приведены в таблтце 5. Сравнение показывает тенденцию современного тестирования, ориентирующую на максимальную автоматизацию процесса тестирования и генерацию тестового кода, что позволяет справляться с большими объемами данных и тестов, необходимых для обеспечения качества при производстве программных продуктов.

Таблица 5. Сравнение ручного и автоматизированного подхода

| | **Ручное** | **Автоматизированное** |
|---|---|---|
| **Задание входных значений** | Гибкость в задании данных. Позволяет использовать разные значения на разных циклах прогона тестов, расширяя покрытие | Входные значения строго заданы |
| **Проверка результата** | Гибкая, позволяет тестировщику оценивать нечетко сформулированные критерии | Строгая. Нечетко сформулированные критерии могут быть проверены только путем сравнения с эталоном |
| **Повторяемость** | Низкая. Человеческий фактор и нечеткое определение данных приводят к неповторяемости тестирования | Высокая |
| **Надежность** | Низкая. Длительные тестовые циклы приводят к снижению внимания тестировщика | Высокая, не зависит от длины тестового цикла |
| **Чувствительность к незначительным изменениям в продукте** | Зависит от детальности описания процедуры. Обычно тестировщик в состоянии выполнить тест, если внешний вид продукта и текст сообщений несколько изменились | Высокая. Незначительные изменения в интерфейсе часто ведут к коррекции эталонов |

| | | |
|---|---|---|
| **Скорость выполнениятестового набора** | Низкая | Высокая |
| **Возможность генерации тестов** | Отсутствует. Низкая скорость выполнения обычно не позволяет исполнить сгенерированный набор тестов | Поддерживается |

## 2.7. Системное тестирование

*Системное тестирование* – это синоним «тестирование с помощью методов «черного ящика»», поскольку во время системного тестирования группа тестирования рассматривает в основном «внешнее поведение» приложения. Системное тестирование включает в себя несколько подтипов тестирования: функциональное, регрессионное, безопасности, перегрузок, производительности, удобства использования, случайное, целостности данных, преобразования данных, сохранения резервных копий и способности к восстановлению, готовности к работе, приемо-сдаточные испытания и альфа/бета тестирование.

Системное тестирование завершает проверку реализации приложения. В ходе системного тестирования проводится функциональное тестирование, а также характеристики разработанного ПО, в том числе устойчивость, производительность, надежность и безопасность. Для системного тестирования применяется подход черного ящика: приложение рассматривается как единое целое, на вход подаются реальные данные, работа приложения анализируется по полученным результатам.

На этапе системного тестирования выявляются ошибки, связанные с неправильной реализацией функций ПО, неправильным взаимодействием с другими системами, аппаратным обеспечением, неправильным распределением памяти, отсутствием корректного освобождения ресурсов и т.п. Источником данных выступают техническое задание на разработку приложения, спецификации на компоненты приложения и его окружения и используемые стандарты.

## 2.8. Регрессионное тестирование

Смысл проведения *регрессионного тестирования* заключается в обнаружении дефектов, их документировании и отслеживании вплоть до устранения. Тестировщик должен быть уверен в том, что меры, принимаемые для устранения найденных ошибок, не породят, в свою очередь, новых ошибок в других областях системы. Регрессионное тестирование позволяет выяснить, не появились ли какие-либо ошибки в результате ликвидации уже обнаруженных ошибок. Именно для регрессионного тестирования применение инструментов автоматизированного тестирования дает наибольшую отдачу. Все созданные ранее скрипты можно использовать снова для подтверждения того, что в результате изменений, внесенных при устранении ошибки, не появились новые дефекты.

*Цели и задачи регрессионного тестирования*

Регрессионное тестирование – дорогостоящая, но необходимая деятельность в рамках этапа сопровождения, направленная на перепроверку корректности измененной программы. В соответствии со стандартным определением, регрессионное тестирование – это выборочное тестирование, позволяющее убедиться, что изменения не вызвали нежелательных побочных эффектов или что измененная система по-прежнему соответствует требованиям.

Главной задачей этапа сопровождения является реализация систематического процесса обработки изменений в коде. После каждой модификации программы необходимо удостовериться, что на функциональность программы не оказал влияния модифицированный код. Если такое влияние обнаружено, говорят о регрессионном дефекте. Для регрессионного тестирования функциональных возможностей, изменение которых не планировалось, используются ранее разработанные тесты. Одна из целей регрессионного тестирования состоит в том, чтобы, в соответствии с используемым критерием покрытия кода (например, критерием покрытия потока операторов или потока данных), гарантировать тот же уровень покрытия, что и при полном повторном тестировании програм-

мы. Для этого необходимо запускать тесты, относящиеся к измененным областям кода или функциональным возможностям.

Другая цель регрессионного тестирования состоит в том, чтобы удостовериться, что программа функционирует в соответствии со своей спецификацией и что изменения не привели к внесению новых ошибок в ранее протестированный код. Эта цель всегда может быть достигнута повторным выполнением всех тестов регрессионного набора, но более перспективно отсеивать тесты, на которых выходные данные модифицированной и старой программы не могут различаться. Результаты сравнения выборочных методов и метода повторного прогона всех тестов приведены в таблице 6.

Таблица 6. Выборочное регрессионное тестирование и повторный прогон всех тестов

| **Повторный прогон всех тестов** | **Выборочное регрессионное тестирование** |
|---|---|
| Прост в реализации | Требует дополнительных расходов при внедрении |
| Дорогостоящий и неэффективный | Способно уменьшать расходы за счет исключения лишних тестов |
| Обнаруживает все ошибки, которые были бы найдены при исходном тестировании | Может приводить к пропуску ошибок |

Важной задачей регрессионного тестирования является также уменьшение стоимости и сокращение времени выполнения тестов.

**Контрольные задания**

1. Объясните, что такое тестирование ПО?

2. Приведите несколько примеров, которые объясняют критерии входа для тестирования ПО.

3. Что такое критерии выхода?

4. Приведите несколько инструментов, которые могут использоваться для автоматизации тестирования.

5. Объясните Покрытие кода.

6. Что такое тестирование Белого ящика?

7. Что такое тестирование Чёрного ящика?

8. Что такое интеграционное тестирование?

9. Что такое ручное тестирование?

10. Что такое регрессионное тестирование?

**Библиографический список**

1. Верификация программного обеспечения : курс лекций / С.В. Синицын, Н. Ю. Налютин. — Москва: Интуит НОУ, 2016. — 446 с.

2. Степанченко И. В. – Методы тестирования программного обеспечения. Учебное пособие ВГТУ. – Волгоград, 2006. – 74 с.

3. Software-testing Инструменты автоматизации тестирования. [электронный ресурс]. – Режим доступа: http://software-testing.ru/library/testing/mobile-testing/2735-mobile-autotesting

4. Автоматизация тестирования UiAutomator [электронный ресурс]. – Режим доступа: https://habr.com/en/company/intel/blog/205864/

5. Jmeter: нагрузочное тестирование базы данных, [электронный ресурс]. - Режим доступа: https://blog.soft-industry.com/ru/jmeter-databases-load-testing-part-i/

6. Сертификация программного обеспечения (ПО) [Электронный ресурс]/ Национальная сертификационная палата. URL: http://www.nspru.ru/sertsoftware/

Электронное учебное издание

Ольга Викторовна **Свиридова**
Александр Александрович **Рыбанов**

***Тестирование и отладка программного обеспечения***

*Учебное пособие*

*Электронное издание сетевого распространения*

Редактор Матвеева Н.И.

Темплан 2024 г. Поз. № 3.
Подписано к использованию 31.05.2024. Формат 60х84 1/16.
Гарнитура Times. Усл. печ. л. 4,0.

Волгоградский государственный технический университет.
400005, г. Волгоград, пр. Ленина, 28, корп. 1.

ВПИ (филиал) ВолгГТУ.
404121, г. Волжский, ул. Энгельса, 42а.