## Лекция 1 Тестирование программного обеспечения

Качество программного продукта характеризуется набором свойств, определяющих, насколько продукт «хорош» с точки зрения заинтересованных сторон, таких как заказчик продукта, спонсор, конечный пользователь, разработчики и тестировщики продукта, инженеры поддержки, сотрудники отделов маркетинга, обучения и продаж. Каждый из участников может иметь различное представление о продукте и о том, насколько он хорош или плох, то есть о том, насколько высоко качество продукта.

С технической точки зрения тестирование заключается в выполнении приложения на некотором множестве исходных данных и сверке получаемых результатов с заранее известными (эталонными) с целью установить соответствие различных свойств и характеристик приложения заказанным свойствам. Как одна из основных фаз процесса разработки программного продукта (Дизайн приложения – Разработка кода – Тестирование), тестирование характеризуется достаточно большим вкладом в суммарную трудоемкость разработки продукта. Широко известна оценка распределения трудоемкости между фазами создания программного продукта: 40% – 20% – 40% (рис. 1.1), из чего следует, что наибольший эффект в снижении трудоемкости может быть получен прежде всего на фазах Design и Testing. Поэтому основные вложения в автоматизацию или генерацию кода следует осуществлять, прежде всего, на этих фазах. Хотя в современном индустриальном программировании автоматизация тестирования является широко распространенной практикой, в то же время технология верификации требований и спецификаций пока делает только свои первые шаги. Задачей ближайшего будущего является движение в сторону такого распределения трудоемкости (60% – 20% –20% (рис. 1.2)), чтобы суммарная цена обнаружения большинства дефектов стремилась к минимуму за счет обнаружения преимущественного числа на наиболее ранних фазах разработки программного продукта.

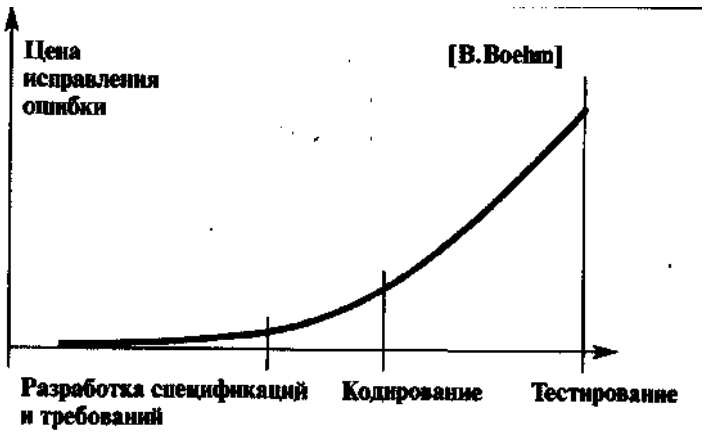


**Рис.** 1.1. Оценка трудоемкости обнаружения и исправления ошибок при создании программного продукта

условиях производства программного продукта.

**Концепция тестирования**

Программа – это аналог формулы в обычной математике. Формула для функции f, полученной суперпозицией функций $f_1, f_2, \ldots f_n$ – выражение, описывающее эту суперпозицию:

$$f = f_1 * f_2 * f_3 * \cdots * f_n$$

Если аналог $f_1, f_2, \ldots, f_n$ – операторы языка программирования, то их формула – программа.

**Основная терминология**

*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.

*Тестирование* обеспечивает выявление (констатацию наличия) фактов расхождений с требованиями (ошибок).

Как правило, на фазе тестирования осуществляется и исправление идентифицированных ошибок, включающее локализацию ошибок, нахождение причин ошибок и соответствующую корректировку программы *тестируемого приложения* (Application (AUT) или Implementation Under Testing (ШТ)).

Если программа не содержит синтаксических ошибок (прошла трансляцию) 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.

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

***Пример пошагового выполнения программы***

*При пошаговом выполнении программы код выполняется строчка за строчкой. В среде Microsoft VisualStudio.NETвозможны следующие команды пошагового выполнения:*

- ***Step Into*** *– если выполняемая строчка кода содержит вызов функции, процедуры или метода, то происходит вызов, и программа останавливается на первой строчке вызываемой функции, процедуры или метода.*
- ***Step Over*** *– 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.*
- ***Step Out*** *– предназначена для выхода из функции в вызывающую функцию. Эта команда продолжит выполнение функции и остановит выполнение на первой строчке после вызываемой функции.*

*Пошаговое выполнение до сих пор является мощным методом автономного тестирования и отладки небольших программ.*

***Пример выполнения программы с заказанными контрольными точками и анализом трасс и дампов***

*Контрольная точка (*breakpoint*) – точка программы, которая при ее достижении 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 (*break mode*). Вход в режим останова не прерывает и не заканчивает выполнение программы и позволяет анализировать состояние отдельных переменных или структур данных. Возврат из режима* break mode *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 редким 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 рис.* 2.4 *условно изображен управляющий граф некоторой программы. Трасса, проходящая через вершины 0-1-3-4-5зафиксирована в табл.* 2.1. *Строки таблицы отображают вершины управляющего графа программы, или* breakpoints*, в которых фиксировались текущие значения заказанных пользователем переменных.*

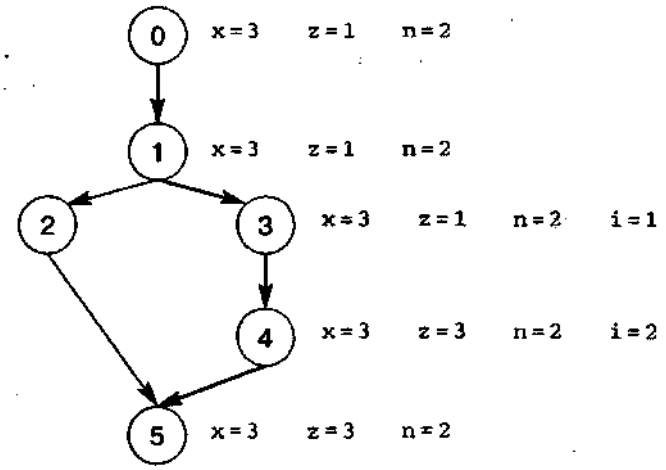


Рис. 2.4. Управляющий граф программы

Табл. 2.1. 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 (reversible execution)

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.

**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.*

**Разработка тестов**

*Определим области эквивалентности входных параметров.*

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

1) *x< 0(ошибочное);*
2) *x > 999 (ошибочное);*
3) *x – не число (ошибочное);*
4) *0 ≤ x ≤999 (корректное).*

*Для n – степени числа:*

5) *n < 1 (ошибочное);*
6) *n >99 (ошибочное);*
7) *n – не число (ошибочное);*
8) *1 ≤ n ≤ 100 (корректное).*

**Анализ тестовых случаев**

1. *Входные значения: (x = 2, n = 3) (покрывают классы 4, 8). Ожидаемый результат: The power n of x is 8.*
2. *Входные значения: {(x~-1, n= 2),(x = 1000, n = 5)} (покрывают классы 1, 2). Ожидаемый результат: Error: x must be in [0..999].*
3. *Входные значения: f(x = 100, n = 0),(x = 100, n = 200)} (покрывоют классы 5,6). Ожидаемый результат: Error: n must be in [1..100].*
4. *Входные значения:(x =ADS, n =ASD) (покрывают классы эквивалентности 3, 7). Ожидаемый результат: Error: Please enter a numeric argument.*
5. *Проверка на граничные значения:*

*5.1 Входные значения: (x = 999, n = 1).*

*Ожидаемый результат: The power n of x is 999.*

*5.2 Входные значения: x = 0, n = 100.*

*Ожидаемый результат: The power 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.*

***Три фазы тестирования***

Реализация тестирования разделяется на три этапа:

- *Создание тестового набора* (test suite) путем ручной разработки или автоматической генерации для конкретной среды тестирования (testing environment).
- *Прогон* программы на тестах, управляемый тестовым монитором (test monitor, test driver [IEEE Std 829-1983], [9]) с получением протокола результатов тестирования (test log).
- *Оценка результатов* выполнения программы на наборе тестов с целью принятия решения о продолжении или остановке тестирования.

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

***Простой пример***

Рассмотрим вопросы тестирования на примере простой программы (рис. 2.7) на языке C#. Текст этой программы и некоторых других несколько видоизменен с целью сделать иллюстрацию описываемых фактов более прозрачной.

```
//  Метод вычисляет неотрицательную степень n числа x
1 static public double Power(double x, int n){
2     double z=1;
3     for (int i=1;
4     n>=i;
5     i++)
6     {z = z*x;} //Возврат в п.4
7     return z;}
```

Рис. 2.7. Пример простой программы на языке C#.

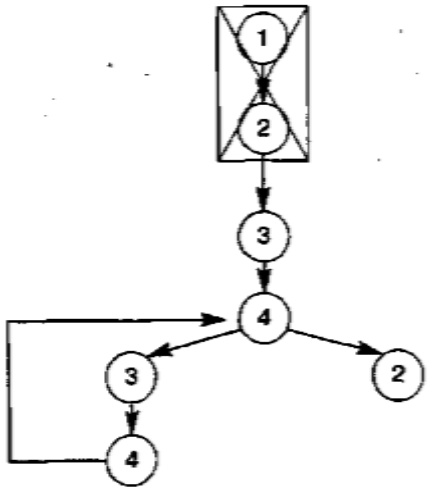


Рис. 2.8. Управляющий граф программы

Управляющий граф программы (УГП) на рис. 2.8 отображает поток управления программы. Нумерация узлов графа совпадает с нумерацией строк программы. Узлы 1 и 2 не включаются в УГП, поскольку отображают строки описаний, т.е. не содержат управляющих операторов.

***Управляющий граф программы***

*Управляющий граф программы (УГП)* – граф G(V,A), где $V(V_1,...,V_m)$ – множеств вершин (операторов), A(Aj,...,An) – множество дуг (управлений), соединяющих операторы-вершины.

*Путь* – последовательность вершин и дуг УГП, в которой любая дуга выходит из вершины Vj и приходит в вершину Vj, например: (3,4,7), (3,4,5,6,4,5,6), (3,4), (3,4,5,6)

*Ветвь* – путь ($V_1$, $V_2$, ..., $V_k$), где $V_1$ – либо первый, либо условный оператор программы, $V_k$- 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.

Существуют *реализуемые* и *нереализуемые* пути в программе, в нереализуемые пути в обычных условиях попасть нельзя.

```
   public static float H(float x,float y)
   {
      float H;
1     if (x*x+y*y+2<=0)
2     H = 17;
3     else H = 64;
4     return H*H+x*x;
   }
```

Рис. 2.9. Пример описания функции с реализуемыми и нереализуемыми путями

Например, для функции рис. 2.9 путь (1, 3, 4) реализуем, путь (1, 2, 4) нереализуем в условиях нормальной работы. Но при сбоях даже нереализуемый путь может реализоваться.

***Основные проблемы тестирования***

Рассмотрим два примера тестирования.

Пусть программа H(x: int, y: int) реализована в машине с 64 разрядным словом, тогда мощность множества тестов $\|(X,Y)\|=2^{64}$.

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.

**Критерии выбора тестов**

***Требования к идеальному критерию тестирования***

Требования к идеальному критерию были выдвинуты в работе:

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.

***Классы критериев***

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

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

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

IV. 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 (Unit testing, Integration testing).

Структурные критерии базируются на основных элементах УГП, операторах, ветвях и путях:

• Условие критерия *тестирования команд* (критерий C0) – 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 C1) – 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 C2) – набор тестов в совокупности должен обеспечить прохождение каждого пути не менее одного раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто – 2, или числом классов выходных путей).

***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, Functional specification 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 тестировании 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. При создании тестов классы входных данных сопоставляются с режимами использования тестируемого компонента или подсистемы приложения, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов. 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 (time out). 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 отчасти объединяет 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.

***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:

- Разработать программы – имитаторы случайных последовательностей входных сигналов {x}.
- Вычислить независимым способом значения {y} для соответствующих входных сигналов {x} и получить тестовый набор (X,Y).
- Протестировать приложение на тестовом наборе (X,Y), используя два способа контроля результатов:
  - *Детерминированный контроль* – проверка соответствия вычисленного значения увк{y} значению y, полученному в результате прогона теста на наборе {x} – случайной последовательности входных сигналов, сгенерированной имитатором.
  - *Стохастический контроль* – проверка соответствия множества значений {ув}, 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 ( $\chi^2$ ) 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.

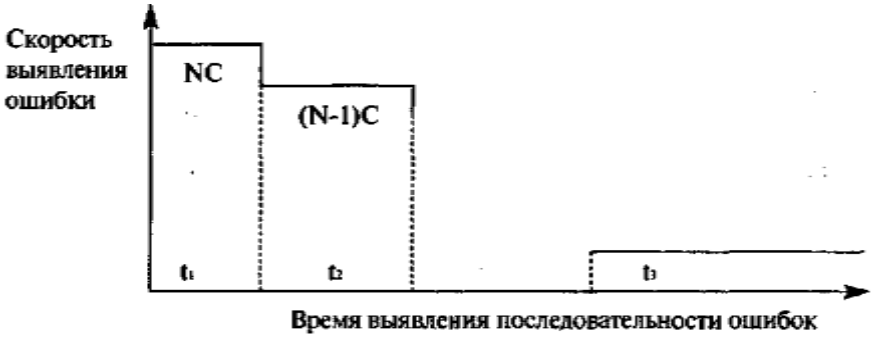


Рис. 3.2. Зависимость скорости выявления ошибок от времени выявления

При формализации модели скорости выявления ошибок (рис. 3.2) 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.

***Мутационный критерий (класс IV)***

Постулируется, что профессиональные программисты пишут сразу почти правильные программы, отличающиеся от правильных мелкими ошибками или описками типа – перестановка местами максимальных значений индексов в описании массивов, ошибки в знаках арифметических операций, занижение или завышение границы цикла на 1 и т.п. Предлагается подход, позволяющий на основе мелких ошибок 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.

**Оценка оттестированности проекта: метрики и методика интегральной оценки**

***Оценка Покрытия Программы и Проекта***

Тестирование программы P по некоторому критерию C означает покрытие множества компонентов программы P $M = \{m_1, \ldots, m_k\}$ по элементам или по связям.

$T = \{t_1, \ldots, t_n\}$ – кортеж неизбыточных тестов $t_i$.

Тест $t_i$ неизбыточен, если существует покрытый им компонент $m_i$ из M(P,C), не покрытый ни одним из предыдущих тестов $t_1, \ldots, t_{i-1}$. Каждому $t_i$ соответствует неизбыточный путь $p_i$ – последовательность вершин от входа до выхода.

V(P, C) – сложность тестирования P по критерию C – измеряется max числом неизбыточных тестов, покрывающих все элементы множества M(P,C).

DV(P, C, T) – остаточная сложность тестирования P по критерию C – измеряется max числом неизбыточных тестов, покрывающих элементы множества M(P,C), оставшиеся непокрытыми, после прогона набора тестов T. Величина DV строго и монотонно убывает от V до 0.

TV(P, C, T) = (V – DV)/V – оценка степени тестированности P по критерию C.

Критерий окончания тестирования TV(P ,C, T) ≥ L, где (0 ≤<L ≤ 1). L – уровень оттестированности, заданный в требованиях к программному продукту.

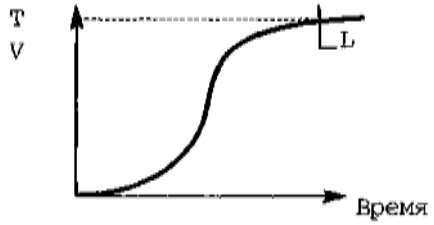


Рис. 4.1. Метрика оттестированности приложения

Рассмотрим две модели программного обеспечения, используемые при оценке оттестированноси.

Для оценки степени оттестированности часто используется УГП – управляющий граф программы. УГП многокомпонентного объекта G (рис. 4.2), содержит внутри себя два компонента G1 и G2, УГП которых раскрыты.

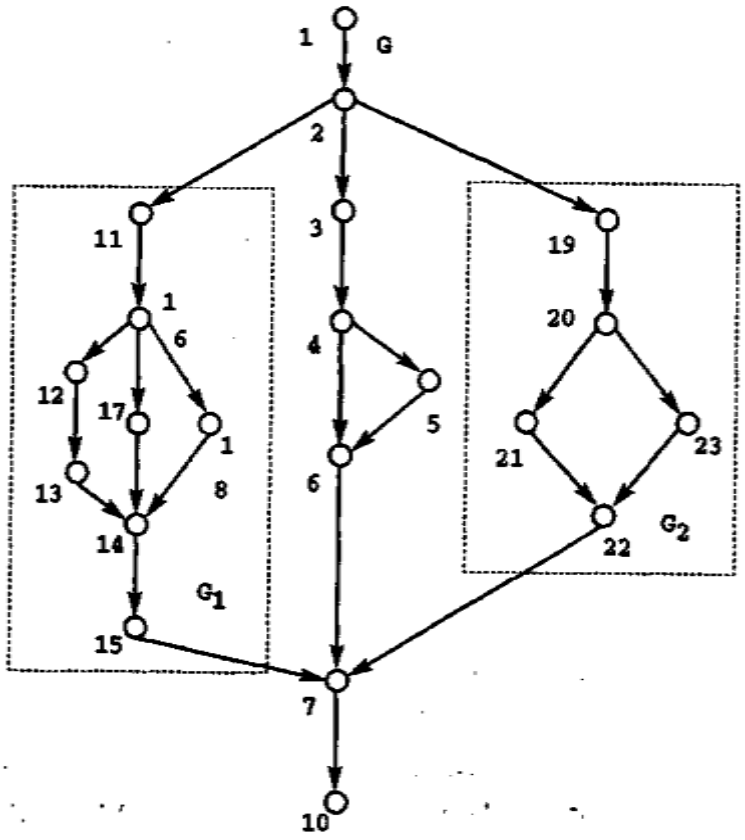


Рис. 4.2. Плоская модель УГП компонента G

В результате УГП компонента G имеет такой вид, как если бы компоненты G1 и G2 в его структуре специально не выделялись, а УГП компонентов G1 и G2 были вставлены в УГП G. Для тестирования компонента G в соответствии с критерием путей потребуется прогнать тестовый набор, покрывающий следующий набор трасс графа G (рис. 4.3):

$P_1(G)$ = 1-2-3-4-5-6-7-10;
$P_2(G)$ = 1-2-3-4-6-7-10;
$P_3(G)$ = 1-2-11-16-18-14-15-7-10;
$P_4(G)$ = 1-2-11-16-17-14-15-7-10;
$P_5(G)$ = 1-2-11-16-12-13-14-15-7-10;
$P_6(G)$ = 1-2-19-20-23-22-7-10;
$P_7(G)$ = 1-2-19-20-21-22-7-10;

Рис. 4.3. Набор трасс, необходимых для покрытия плоской модели УГП компонента G

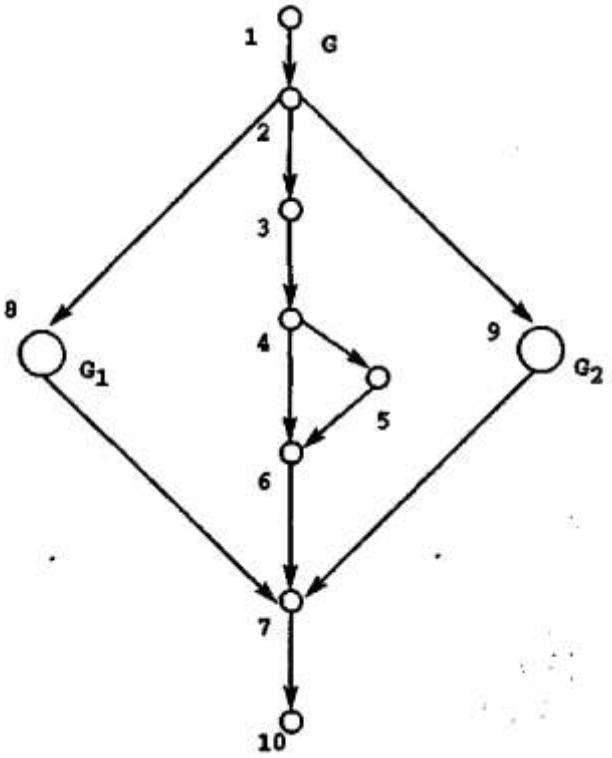


Рис. 4.4. Иерархическая модель УГП компонента G УГП компонента G, представленный в виде иерархической модели, приведен на рис. 4.4 и рис. 4.9.

В иерархическом УГП G входящие в его состав компоненты представлены ссылками на свои УГП G1 и G2.

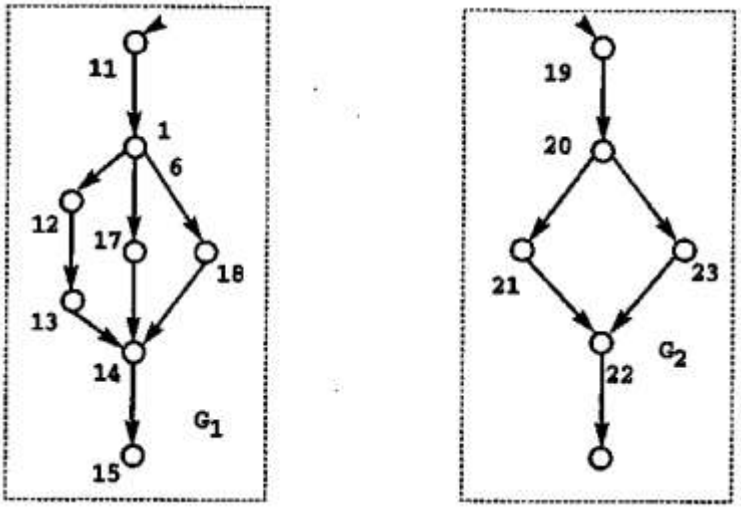


Рис. 4.5. Иерархическая модель: УГП компонентов G1 и G2

Для исчерпывающего тестирования иерархической модели компонента G в соответствии с критерием путей требуется прогнать следующий набор трасс (рис. 4.6):

$P^1(G)$ = 1-2-3-4-5-6-7-10;
$P^2(G)$ = 1-2-3-4-6-7-10;
$P^3(G)$ = 1-2-8-7-10;
$P^4(G)$ = 1-2-9-7-10.

Рис. 4.6. Набор трасс, необходимых для покрытия иерархической модели УГП компонента G

Приведенный набор трасс достаточен при условии, что компоненты G1 и G2 в свою очередь исчерпывающе протестированы. Чтобы обеспечить выполнение этого условия в соответствии с критерием путей, надо прогнать все трассы рис. 4.7.

```
P11(G1)=11-16-12-13-14-15;      P21(G2)=19-20-21-22.
P12(G1)=11-16-17-14-15.         P22(G2)=11-16-18-17-14-15
P13(G1)=19-20-23-22.
```

Рис. 4.7. Набор трасс иерархической модели УГП, необходимых для покрытия УГП компонентов G1 и G2

***Методика интегральной оценки тестированности***

1. Выбор критерия C и приемочной оценки тестированности программного проекта – L.
2. Построение дерева классов проекта и построение УГП для каждого модуля.
3. Модульное тестирование и оценка TV на модульном уровне.
4. Построение УГП, интегрирующего модули в единую иерархическую (классовую) модель проекта.
5. Выбор тестовых путей для проведения интеграционного или системного тестирования.
6. Генерация тестов, покрывающих тестовые пути шага 5.
7. Интегральная оценка тестированности проекта с учетом оценок тестированности модулей-компонентов.
8. Повторение шагов 5 – 7 до достижения заданного уровня тестированности L.

**Модульное и интеграционное тестирование**

**Разновидности тестирования *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 (а именно, спецификацию интерфейса), 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 на основе потока данных, т. е. информационного графа программы.

*Тестирование на основе потока управления.* Особенности использования структурных критериев тестирования C0, C1, C2 были рассмотрены в разделе 2. К ним следует добавить критерий покрытия условий, заключающийся в покрытии всех логических (булевских) условий в программе. Критерии покрытия решений (ветвей – C1) и условий не заменяют друг друга, поэтому на практике используется комбинированный критерий покрытия условий/решений, совмещающий требования по покрытию и решений, и условий.

К популярным критериям относятся критерий покрытия функций программы, согласно

которому каждая функция программы должна быть испытана хотя бы один раз, и критерий покрытия вызовов, согласно которому каждый вызов каждой функции в программе должен быть осуществлен хотя бы один раз. Критерий покрытия вызовов известен также как Критерий покрытия пар вызовов (call pair coverage).

*Тестирование на основе потока данных.* Этот вид тестирования направлен на выявление ссылок на неинициализированные переменные и избыточные присваивания (аномалии потока данных). Эта стратегия требует тестирования всех взаимосвязей, включающих в себя ссылку (использование) и определение переменной, на которую указывает ссылка (т. е. требуется покрытие дуг информационного графа программы). Недостаток стратегии в том, что она не включает критерий С1 и не гарантирует покрытия решений.

Стратегия требуемых пар также тестирует упомянутые взаимосвязи. Использование переменной в предикате дублируется в соответствии с числом выходов решения, и каждая из таких требуемых взаимосвязей должна быть протестирована. К популярным критериям принадлежит критерий CP, заключающийся в покрытии всех таких пар дуг v и w, что из дуги v достижима дуга w, поскольку именно на дуге может произойти значение переменной, которая в дальнейшем уже не должна использоваться. Для «покрытия» еще одного популярного критерия Cdu достаточно тестировать пары (вершина, дуга), поскольку определение переменной происходит в вершине УГП, а ее использование – на дугах, исходящих из решений, или в вычислительных вершинах.

*Методы проектирования тестовых путей для достижения заданной степени тестированности в структурном тестировании.*

Процесс построения набора тестов при структурном тестировании принято делить на три фазы:

- конструирование УГП;
- выбор тестовых путей;
- генерация тестов, соответствующих тестовым путям.

Первая фаза соответствует статическому анализу программы, задача которого состоит в получении графа программы и зависящего от него и от критерия тестирования множества элементов, которые необходимо покрыть тестами.

На третьей фазе по известным путям тестирования осуществляется поиск подходящих тестов, реализующих прохождение этих путей.

Вторая фаза обеспечивает выбор тестовых путей. Выделяют три подхода к построению тестовых путей:

- статические методы;
- динамические методы;
- методы реализуемых путей.

*Статические методы.* Самое простое и легко реализуемое решение – построение каждого пути посредством постепенного его удлинения за счет добавления дуг, пока не будет достигнута

выходная вершина управляющего графа программы. Эта идея может быть усилена в так называемых адаптивных методах, которые каждый раз добавляют только один тестовый путь (входной тест), используя предыдущие пути (тесты) как руководство для выбора последующих путей в соответствии с некоторой стратегией. Чаще всего адаптивные стратегии применяются по отношению к критерию С1. Основной недостаток статических методов заключается в том, что не учитывается возможная нереализуемость построенных путей тестирования.

*Динамические методы.* Такие методы предполагают построение полной системы тестов, удовлетворяющих заданному критерию, путем одновременного решения задачи построения покрывающего множества путей и тестовых данных. При этом можно автоматически учитывать реализуемость или нереализуемость ранее рассмотренных путей или их частей. Основной идеей динамических методов является подсоединение к начальным реализуемым отрезкам путей дальнейших их частей так, чтобы: 1) не терять при этом реализуемости вновь полученных путей; 2) покрыть требуемые элементы структуры программы.

*Методы реализуемых путей.* Данная методика заключается в выделении из множества путей подмножества всех реализуемых путей. После чего покрывающее множество путей строится из полученного подмножества реализуемых путей.

Достоинство статических методов состоит в сравнительно небольшом количестве необходимых ресурсов, как при использовании, так и при разработке. Однако их реализация может содержать непредсказуемый процент брака (нереализуемых путей). Кроме того, в этих системах переход от покрывающего множества путей к полной системе тестов пользователь должен осуществить вручную, а эта работа достаточно трудоемкая. Динамические методы требуют значительно больших ресурсов как при разработке, так и при эксплуатации, однако увеличение затрат происходит, в основном, за счет разработки и эксплуатации аппарата определения реализуемости пути (символический интерпретатор, решатель неравенств). Достоинство этих методов заключается в том, что их продукция имеет некоторый качественный уровень – реализуемость путей. Методы реализуемых путей дают самый лучший результат.

### *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.

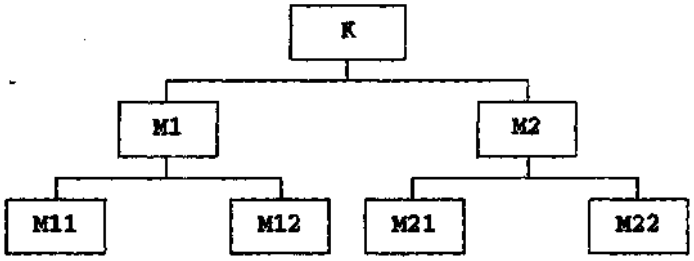


Рис. 5.3. Пример структуры комплекса программ

На рис. 5.3 приведена структура комплекса программ К, состоящего из оттестированных на этапе модульного тестирования модулей $M_1$, $M_2$, $M_{11}$, $M_{12}$, $M_{21}$, $M_{22}$ .Задача, решаемая методом интеграционного тестирования, – тестирование межмодульных связей, реализующихся при исполнении программного обеспечения комплекса К. 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.

Интеграционное тестирование применяется на этапе сборки модульно оттестированных 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 восходящее тестирование.
– «Снизу вверх» и соответственно нисходящее тестирование.

*Особенности монолитного тестирования* заключаются в следующем:

для замены неразработанных к моменту тестирования модулей, кроме самого верхнего (К на рис. 5.3), необходимо дополнительно разрабатывать *драйверы (test driver)* и/или *заглушки (stub)* [9], замещающие отсутствующие на момент сеанса тестирования модули нижних уровней.

Сравнение монолитного и интегрального 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.

Недостатки нисходящего тестирования:

- Проблема разработки достаточно «интеллектуальных» заглушек, т.е. заглушек, способных 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.

Недостатки восходящего тестирования:

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

**Разновидности тестирования: системное и регрессионное тестирование**

***Системное тестирование***

*Системное тестирование* качественно отличается от интеграционного и модульного уровней. Системное тестирование рассматривает тестируемую систему в целом и оперирует на уровне пользовательских интерфейсов, в отличие от последних фаз интеграционного тестирования, которое оперирует на уровне интерфейсов модулей. Различны и цели этих уровней тестирования. На уровне системы часто сложно и малоэффективно анализировать прохождение тестовых траекторий внутри программы или отслеживать правильность работы конкретных функций. Основная задача системного тестирования в выявлении дефектов, связанных с работой системы в целом, таких как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в применении и тому подобное.

Системное тестирование производится над проектом в целом с помощью метода «черного ящика». Структура программы не имеет никакого значения, для проверки доступны только входы и выходы, видимые пользователю. Тестированию подлежат коды и пользовательская документация.

*Категории тестов системного тестирования:*

1. Полнота решения функциональных задач.

2. Стрессовое тестирование – на предельных объемах нагрузки входного потока.
3. Корректность использования ресурсов (утечка памяти, возврат ресурсов).
4. Оценка производительности.
5. Эффективность защиты от искажения данных и некорректных действий.
6. Проверка инсталляции и конфигурации на разных платформах.
7. Корректность документации.

Поскольку системное тестирование проводится на пользовательских интерфейсах, создается иллюзия того, что построение специальной системы автоматизации тестирования не всегда необходимо. Однако объемы данных на этом уровне таковы, что обычно более эффективным подходом является полная или частичная автоматизация тестирования, что приводит к созданию тестовой системы гораздо более сложной, чем система тестирования, применяемая на уровне тестирования модулей или их комбинаций.

***Регрессионное тестирование***

*Регрессионное тестирование* – цикл тестирования, который производится при внесении изменений на фазе системного тестирования или сопровождения продукта. Главная проблема регрессионного тестирования – выбор между полным и частичным перетестированием и пополнение тестовых наборов. При частичном перетестировании контролируются только те части проекта, которые связаны с измененными компонентами. На ГМП это пути, содержащие измененные узлы, и, как правило, это методы и классы, лежащие выше модифицированных по уровню, но содержащие их в своем контексте. Пропуск огромного объема тестов, характерного для этапа системного тестирования, удается осуществить без потери качественных показателей продукта только с помощью регрессионного подхода.

***Комбинирование уровней тестирования***

В каждом конкретном проекте должны быть определены задачи, ресурсы и технологии для каждого уровня тестирования таким образом, чтобы каждый из типов дефектов, ожидаемых в системе, был «адресован», то есть в общем наборе тестов должны иметься тесты, направленные на выявление дефектов подобного типа. Табл. 7.1 суммирует характеристики свойств модульного, интеграционного и системного уровней тестирования. Задача, которая стоит перед тестировщиками и менеджерами, заключается в оптимальном распределении ресурсов между всеми тремя типами тестирования. Например, перенесение усилий на поиск фиксированного типа дефектов из области системного в область модульного тестирования может существенно снизить сложность и стоимость всего процесса тестирования.

Таблица 7.1. Характеристики модульного, интеграционного и системного тестирования

| | **Модульное** | **Интеграционное** | **Системное** |
|---|---|---|---|
| **Типы дефектов** | Локальные дефекты, такие какопечатки в реализации алгоритма, неверные операции, логические и математические выражения, циклы, ошибки в использовании локальных ресурсов, рекурсия и т.п. | Интерфейсные дефекты, такие как неверная трактовка параметров и их формат, неверное использование системных ресурсов и средств коммуникации, и т.п. | Отсутствующаяили некорректная функциональность, неудобство использования, непредусмотренные данные и их комбинации, непредусмотренные или не поддержанные сценарии работы, ошибки совместимости, ошибки пользовательской документации, ошибки переносимости продукта на различные платформы, проблемы производительности, инсталляции и т.п. |
| **Необходимость в системе тестирования** | Да | Да | Нет (*) |
| **Цена разработки системы тестирования** | Низкая | Низкая до умеренной | Умеренная до высокой или неприемлемой |
| **Цена процесса тестирования, то есть разработки, прогона и анализа тестов** | Низкая | Низкая | Высокая |

**(*)** прямой необходимости в системе тестирования нет, но цена процесса системного тестирования часто настолько высока, что требует использования систем автоматизации, несмотря на возможно высокую их стоимость.

**Автоматизация тестирования**

***Автоматизация тестирования***

Использование различных подходов к тестированию определяется их эффективностью применительно к условиям, определяемым промышленным проектом. В реальных случаях работа группы тестирования планируется так, чтобы разработка тестов начиналась с момента согласования требований к программному продукту (выпуск Requirement Book, содержащей высокоуровневые требования к продукту) и продолжалась параллельно с разработкой дизайна и кода продукта. В результате к началу системного тестирования создаются тестовые наборы, содержащие тысячи тестов. Большой набор тестов обеспечивает всестороннюю проверку функциональности продукта и гарантирует качество продукта, но пропуск такого количества тестов на этапе системного тестирования представляет проблему. Ее решение лежит в области автоматизации тестирования, т.е. в автоматизации разработки.

Собственно использование эффективной системы автоматизации тестирования сокращает до минимума (например, до одной ночи) время пропуска тестов, без которого невозможно подтвердить факт роста качества (уменьшения числа оставшихся ошибок) продукта. Системное тестирование осуществляется в рамках циклов тестирования (периодов пропуска разработанного тестового набора над build разрабатываемого приложения). Перед каждым циклом фиксируется разработанный или исправленный build, на который заносятся обнаруженные в результате тестового прогона ошибки. Затем ошибки исправляются, и на очередной цикл тестирования предъявляется новый build. Окончание тестирования совпадает с экспериментально подтвержденным заключением о достигнутом уровне качества относительно выбранного критерия тестирования или о снижении плотности необнаруженных ошибок до некоторой заранее оговоренной величины. Возможность ограничить цикл тестирования пределом в одни сутки или несколько часов поддерживается исключительно за счет средств автоматизации тестирования.


Рис. 8.2. Структура инструментальной системы автоматизации тестирования

На рис. 8.2 представлена обобщенная структура системы автоматизации тестирования, в которой создается и сохраняется следующая информация:

• Набор тестов, достаточный для покрытия тестируемого в соответствии с выбранным критерием тестирования – как результат ручной или автоматической разработки (генерации) тестовых наборов и драйвер/монитор пропуска тестового набора.

• Результаты прогона тестового набора, зафиксированные в log-файле. Log-файл содержит трассы («протоколы»), представляющие собой реализованные при тестовом прогоне последовательности некоторых событий (значений отдельных переменных или их совокупностей) и точки реализации этих событий на графе программы. В составе трасс могут присутствовать последовательности явно и неявно заданных меток, задающих пути реализации трасс на управляющем графе программы, совокупности значений переменных на этих метках, величины промежуточных результатов, достигнутых на некоторых метках, и т.п.

• Статистика тестового цикла, содержащая: 1) результаты пропуска каждого теста из тестового набора и их сравнения с эталонными величинами; 2) факты, послужившие основанием для принятия решения о продолжении или окончании тестирования; 3) критерий покрытия и степень его удовлетворения, достигнутая в цикле тестирования. Результатом анализа каждого прогона является список проблем в виде ошибок и дефектов, который заносится в базу развития проекта. Далее происходит работа над ошибками, где каждая поднятая проблема идентифицируется, относится к соответствующему модулю и разработчику, приоритезируется и отслеживается, что обеспечивает гарантию ее решения (исправления или отнесения к списку известных проблем, решение которых по тем или иным причинам откладывается) в последующих build. Исправленный и собранный для тестирования build поступает на следующий цикл тестирования, и цикл повторяется, пока нужное качество программного комплекса не будет достигнуто. В этом итерационном процессе средства автоматизации тестирования обеспечивают быстрый контроль результатов исправления ошибок и проверку уровня качества, достигнутого в продукте. Некачественный продукт зрелая организация не производит.

### *Издержки тестирования*

Интенсивность обнаружения ошибок на единицу затрат и надежность тесно связаны с временем тестирования и, соответственно, с гарантией качества продукта (рис. 8.3 А). Чем больше трудозатрат вкладывается в процесс тестирования, тем меньше ошибок в продукте остается незамеченными. Однако совершенство в индустриальном программировании имеет пределы, которые прежде всего связаны с затратами на получение программного продукта, а также с избытком качества, которое не востребовано заказчиком приложения. Нахождение оптимума – очень ответственная задача тестировщика и менеджера проекта.

Движение к уменьшению числа оставшихся ошибок или к качеству продукта приводит к применению различных методов отладки и тестирования в процессе создания продукта. На рис.8.3 В приведен затратный компонент тестирования в зависимости от совершенствования применяемого инструментария и методов тестирования.

На практике популярны следующие методы тестирования и отладки, упорядоченные по связанным с их применением затратам:

1. Статические методы тестирования.
2. Модульное тестирование.
3. Интеграционное тестирование.
4. Системное тестирование.
5. Тестирование реального окружения и реального времени. Зависимость эффективности применения перечисленных методов

или их способности к обнаружению соответствующих классов ошибок (C) сопоставлена на рис. 8.3 с затратами (B). График показывает, что со временем, по мере обнаружения более сложных ошибок и дефектов, эффективность низкозатратных методов падает вместе с количеством обнаруживаемых ошибок.

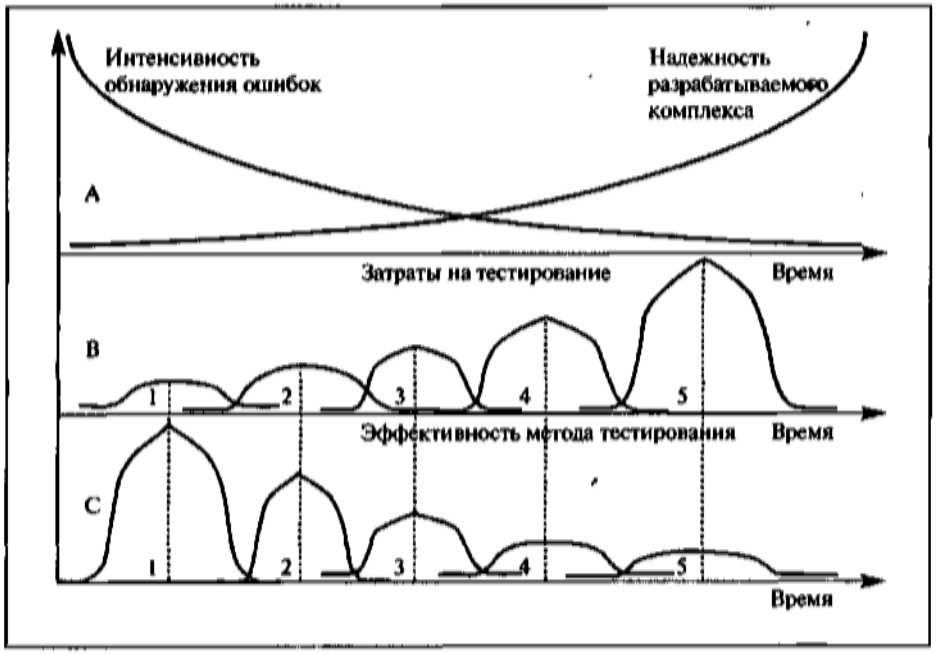


Рис. 8.3 Издержки тестирования

Отсюда следует, что все методы тестирования не только имеют право на существование, но и имеют свою нишу, где они хорошо обнаруживают ошибки, тогда как вне ниши их эффективность падает. Поэтому необходимо совмещать различные методы и стратегии отладки и тестирования с целью обеспечения запланированного качества программного продукта при ограниченных затратах, что достижимо при использовании процесса управления качеством программного продукта.

## Особенности индустриального тестирования

### Индустриальный подход. Особенности индустриального тестирования

***Качество программного продукта и тестирование***

Качество программного продукта можно оценить некоторым набором характеристик, определяющих, насколько продукт «хорош» с точки зрения всех потенциально заинтересованных в нем сторон. Такими сторонами являются:

- Заказчик продукта.
- Спонсор.
- Конечный пользователь.
- Разработчики продукта.
- Тестировщики продукта.
- Инженеры поддержки.
- Отдел обучения.
- Продаж и т.п.

Каждый из участников может иметь различное представление о продукте и по-разному судить о том, насколько он хорош или плох, то есть насколько высоко качество продукта. С точки зрения разработчика, продукт может быть настолько хорош, насколько хороши заложенные в нем алгоритмы и технологии. Пользователю продукта, скорее всего, безразличны детали внутренней реализации, его в первую очередь волнуют вопросы функциональности и надежности. Спонсора интересует цена и совместимость с будущими технологиями. Таким образом, задача обеспечения качества продукта выливается в задачу определения заинтересованных лиц, согласования их критериев качества и нахождения оптимального решения, удовлетворяющего этим критериям.

В рамках подобной задачи группа тестирования рассматривается не просто как еще одна заинтересованная сторона, но и как сторона, способная оценить удовлетворение выбранных критериев и сделать вывод о качестве продукта с точки зрения других участников. К сожалению, далеко не все критерии могут быть оценены группой тестирования. Поэтому ее внимание в основном сосредоточено на критериях, определяющих качество программного продукта с точки зрения конечного пользователя.

*Тестирование как способ обеспечения качества.* Тестирование, с технической точки зрения, есть процесс выполнения приложения на некоторых входных данных и проверка получаемых результатов с целью подтвердить их корректность по отношению к результату.

Тестирование не позиционируется в качестве единственного способа обеспечения качества. Оно является частью общей системы обеспечения качества продукта, элементы которой выбираются по критерию наибольшей эффективности применения в конкретном проекте. Рассмотрим пример. В качестве приложения возьмем программу для работы с сетью (browser),

критерии качества которой приведены в табл. 9.1.

Матрица критериев качества заинтересованных в них участников для рассматриваемого проекта приведена в табл. 9.2. Допустим, что вид матрицы критериев качества и проверяющих элементов системы обеспечения качества для данного проекта будет следующим.

Данные (табл. 9.1–9.2) показывают, что из восьми элементов общего качества продукта тестирование способно оценить и контролировать только три (1,3, 4), причем наиболее эффективно тестирование контролирует отсутствие дефектов (3).

Таблица 9.1. Критерии качества программы browser

| | **Пользователь** | **Заказчик** | **Инженер поддержки** |
|---|---|---|---|
| 1.Функциональная полнота | + | – | – |
| 2. Цена разработки | – | + | – |
| 3. Отсутствие дефектов | + | косвенно | + |
| 4. Удобство использования | + | – | – |
| 5. Возможность внесения изменений в будущем | – | косвенно | + |
| 6. Легкость исправления дефектов | – | – | + |
| 7. Документация на реализацию, в том числе комментарии | – | – | + |
| 8. Своевременность исполнения проекта | – | + | – |

В каждом конкретном проекте элементы системы должны быть выбраны так, чтобы обеспечить приемлемое качество, исходя из приоритетов и имеющихся ресурсов. Выбирая элементы для системы обеспечения качества конкретного продукта, можно применить комбинированное тестирование, обзоры кода, аудит. При подобном выборе некоторые качества, например легкость модификации и исправления дефектов, не будут оценены и, возможно, не будут выполнены. Задачей тестирования в рассматриваемом случае будет обнаружение дефектов и оценка удобства использования продукта, включая полноту функциональности. Исходя из задач, поставленных перед группой тестирования в конкретном проекте, выбирается соответствующая стратегия тестирования. Так, в данном примере, ввиду необходимости оценить удобство использования и полноту функциональности, преимущественный подход к разработке тестов следует планировать на основе использования сценариев.

Итак, основная последовательность действий при выборе и оценке критериев качества программного продукта включает:

1. Определение всех лиц, так или иначе заинтересованных в исполнении и результатах данного проекта.
2. Определение критериев, формирующих представление о качестве для каждого из

участников.

Таблица 9.2. Матрица критериев качества и элементов системы обеспечения качества

| | **Тестирование** | **Анализ рынка и специальные лаборатории (*)** | **Обзоры кода** | **Анализ дизайна** | **Аудиты процесса разработки** |
|---|---|---|---|---|---|
| 1. Полнота функциональности | +, не всегда эффективно | + | – | – | – |
| 2. Стоимость разработки | – | – | – | – | + |
| 3. Отсутствие дефектов | + | - | + | — | — |
| 4. Удобство использования | +, не всегда эффективно | + | – | – | – |
| 5. Возможность внесения изменений в будущем | – | – | +/– | + | – |
| 6. Легкость исправления дефектов | – | – | + | + | – |
| 7. Документация на реализацию, в том числе комментарии | – | – | + | – | + |
| 8. Своевременность исполнения проекта | – | – | – | – | + |

*(*) имеются в виду лаборатории исследования эффективности различных сценариев использования продукта (usability labs)*

3. Приоритезацию критериев, с учетом важности конкретного участника для компании, выполняющей проект, и важности каждого из критериев для данного участника.
4. Определение набора критериев, которые будут отслежены и выполнены в рамках проекта, исходя из приоритетов и возможностей проектной команды. Постановка целей по каждому из критериев.
5. Определение способов и механизмов достижения каждого критерия.
6. Определение стратегии тестирования исходя из набора критериев, подпадающих под ответственность группы тестирования, выбранных приоритетов и целей.

### *Процесс тестирования*

Как отмечалось ранее, в тестировании выделяются три основных уровня, или три фазы:

1. Модульное тестирование.
2. Интеграционное тестирование.
3. Системное тестирование.

Задача планирования активности тестирования состоит в оптимальном распределении ресурсов между всеми типами тестирования. В дальнейшем изложении мы сконцентрируемся на системной фазе тестирования, как на наиболее важной и критичной активности для разработки качественного программного продукта.

*Фазы процесса тестирования*

В процесс тестирования выделяют следующие фазы:

1. *Определение целей* (требований к тестированию), включающее следующую конкретизацию: какие части системы будут тестироваться, какие аспекты их работы будут выбраны для проверки, каково желаемое качество и т.п.
2. *Планирование:* создание графика (расписания) разработки тестов для каждой тестируемой подсистемы; оценка необходимых человеческих, программных и аппаратных ресурсов; разработка расписания тестовых циклов. Важно отметить, что расписание тестирования обязательно должно быть согласовано с расписанием разработки создаваемой системы, поскольку наличие исполняемой версии разрабатываемой системы *{Implementation Under Testing (IUT)* или *Application Under Testing (AUT)* – часто употребляемые обозначения для тестируемой системы) является одним из необходимых условий тестирования, что создает взаимозависимость в работе команд тестировщиков и разработчиков.
3. *Разработка тестов,* то есть тестового кода для тестируемой системы, если необходимо – кода системы автоматизации тестирования и тестовых процедур (выполняемых вручную).
4. *Выполнение тестов:* реализация тестовых циклов.
5. *Анализ результатов.*

После анализа результатов возможно повторение процесса тестирования, начиная с пунктов 3, 2 или даже 1.

*Тестовый цикл*

*Тестовый цикл* – это цикл исполнения тестов, включающий фазы 4 и 5 тестового процесса. Тестовый цикл заключается в прогоне разработанных тестов на некотором однозначно определяемом срезе системы (состоянии кода разрабатываемой системы). Обычно такой срез системы называют *build.* Тестовый цикл включает следующую последовательность действий:

1. *Проверка готовности системы и тестов* к проведению тестового цикла, включающая:
   - Проверку того, что все тесты, запланированные для исполнения на данном цикле, разработаны и помещены в систему версионного контроля.
   - Проверку того, что все подсистемы, запланированные для тестирования на данном цикле, разработаны и помещены в систему версионного контроля.
   - Проверку того, что разработана и задокументирована процедура определения и создания среза системы, или build.
   - Проверки некоторых дополнительных критериев.
2. *Подготовка тестовой машины* в соответствии с требованиями, определенными на этапе планирования (например, полная очистка и переустановка системного программного обеспечения). Конфигурация тестовой машины, так же, как и срез системы, должны быть

однозначно воспроизводимыми.

3. *Воспроизведение среза* системы.
4. *Прогон тестов* в соответствии с задокументированными процедурами.
5. *Сохранение тестовых протоколов (test log).* Test log может содержать вывод системы в STDOUT, список результатов сравнения полученных при исполнении данных с эталонными или любые другие выходные данные тестов, с помощью которых можно проверить правильность работы системы
6. *Анализ протоколов тестирования* и принятие решения о том, прошел или не прошел каждый из тестов *(Pass/Tail).*
7. *Анализ и документирование* результатов цикла.

Последний перед выпуском продукта тестовый цикл не должен включать изменений кода build или кода продукта тестируемой системы. Этот цикл называется «финальным». Таким образом обеспечивается ситуация, когда финальный цикл полностью повторяем, а выпускаемый продукт полностью совпадает с продуктом, который прошел тестирование. Финальный цикл необходим для гарантии достоверности результатов тестирования.

### ***Планирование тестирования***

*Тестовый план*

Тестовый план – это документ, или набор документов, содержащий следующую информацию:

1. Тестовые ресурсы.
2. Перечень функций и подсистем, подлежащих тестированию.
3. Тестовую стратегию, включающую:
    a) Анализ функций и подсистем с целью определения наиболее слабых мест, то есть областей функциональности тестируемой системы, где появление дефектов наиболее вероятно.
    b) Определение стратегии выбора входных данных для тестирования. Так как множество возможных входных данных программного продукта, как правило, практически бесконечно, выбор конечного подмножества, достаточного для проведения исчерпывающего тестирования, является сложной задачей. Для ее решения могут быть применены такие методы как покрытие классов входных и выходных данных, анализ крайних значений, покрытие модели использования, анализ временной линии и тому подобные. Выбранную стратегию необходимо обосновать и задокументировать.
    c) Определение потребности в автоматизированной системе тестирования и дизайн такой системы.
4. Расписание тестовых циклов.
5. Фиксацию тестовой конфигурации: состава и конкретных параметров аппаратуры и программного окружения.
6. Определение списка тестовых метрик, которые на тестовом цикле необходимо собрать и проанализировать. Например, метрик, оценивающих степень покрытия тестами набора

требований, степень покрытия кода тестируемой системы, количество и уровень серьезности дефектов, объем тестового кода и другие характеристики.

### *Типы тестирования*

В тестовом плане определяются и документируются различные типы тестов. Типы тестов могут быть классифицированы по двум категориям: по тому, что именно подвергается тестированию (по виду подсистемы) и по способу выбора входных данных.

Типы тестирования по виду подсистемы или продукта:

1. *Тестирование основной функциональности,* когда тестированию подвергается собственно система, являющаяся основным выпускаемым продуктом.
2. *Тестирование инсталляции* включает тестирование сценариев первичной инсталляции системы, сценариев повторной инсталляции (поверх уже существующей копии), тестирование деинсталляции, тестирование инсталляции в условиях наличия ошибок в инсталлируемом пакете, в окружении или в сценарии и т.п.
3. *Тестирование пользовательской документации* включает проверку полноты и понятности описания правил и особенностей использования продукта, наличие описания всех сценариев и функциональности, синтаксис и грамматику языка, работоспособность примеров и т.п.

Типы тестирования по способу выбора входных значений:

1. *Функциональное тестирование,* при котором проверяется:
   - покрытие функциональных требований;
   - покрытие сценариев использования.
2. *Стрессовое тестирование, при котором проверяются экстремальные режимы использования продукта.*
3. *Тестирование граничных значений.*
4. *Тестирование производительности.*
5. *Тестирование на соответствие стандартам.*
6. *Тестирование совместимости* с другими программно-аппаратными комплексами.
7. *Тестирование работы с окружением.*
8. *Тестирование работы на конкретной платформе.*

В реальных разработках используются и комбинируются различные типы тестов для обеспечения спланированного качества продукта.

### *Подходы к разработке тестов*

Рассмотрим разные подходы к разработке тестов: два к выбору тестовых данных, и два – к реализации тестового кода.

#### *Тестирование спецификации*

При разработке тестов, основанной на функциональной спецификации продукта,

требования к продукту являются основным источником, определяющим, какие тесты будут разработаны. Для каждого требования пишется один или более тестов, которые в совокупности должны проверить выполнение данного требования в продукте.

### *Тестирование сценариев*

Разработка тестов, основанных на использовании сценариев, осуществляется по следующей методике:

1. Определяется модель использования, включающая операционное окружение продукта и «актеров». Актером может быть пользователь, другой продукт, аппаратная часть и тому подобное, то есть все, с чем продукт обменивается информацией. Разделение на окружение и актеров условно и служит для описания оптимальных способов использования продукта.
2. Разрабатываются сценарии использования продукта. Описание сценария в зависимости от продукта и выбранного подхода может быть строго определенным, параметризованным или разрешать некоторую степень неопределенности. Например, описание сценария на языке MSC допускает задание параметризованных сценариев с возможностью переупорядочивания событий.
3. Разрабатывается набор тестов, покрывающих заданные сценарии. С учетом степени неопределенности, заложенной в сценарии, каждый тест может покрывать один сценарий, несколько сценариев, или, наоборот, часть сценария.

Использование сценариев не требует наличия полной формальной спецификации требований, но зато может потребовать больше времени на разработку и анализ.

Еще одна особенность тестирования сценариев заключается в том, что этот метод направляет тестирование на проверку конкретных режимов использования продукта, что позволяет находить дефекты, которые метод тестирования по требованиям может пропустить.

### *Ручная разработка тестов*

Наиболее распространенным способом разработки тестов является создание тестового кода вручную. Это наиболее гибкий способ разработки тестов, однако характерная для него производительность труда инженеров-тестировщиков в создании тестового кода не намного выше скорости создания кода продукта, а объемы тестового кода на практике зачастую превышают объем кода продукта в 10 раз. Учитывая этот факт, в современной индустрии все больше склоняются к более интеллектуальным способам получения тестового кода, таким как использование специальных тестовых языков (скриптов) и генерации тестов.

#### *Генерация тестов*

В настоящее время некоторые языки спецификаций, используемые для описания

алгоритмов тестирования, могут быть использованы для генерации тестового кода.

## Документирование и оценка индустриального тестирования

### *Выполнение тестов*

Рассмотрим два основных подхода к выполнению тестов: подход ручного тестирования и подход автоматического исполнения (прогон) тестов. Подходы рассмотрены на примере тестирования продукта, поддерживающего интерфейс командной строки. Тесты описывают вызов продукта с параметрами и проверку возвращаемого значения в виде фиксируемых при прогоне – текста из STDOUT и состояния некоторых файлов, зависящего от входных параметров.

#### *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 описание 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 документ2].*
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».*

### *Автоматизированное тестирование*

Попытка автоматизировать приведенный в разделе «Пример фрагмента процедуры» 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 табл. 10.1. 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 10.1. Сравнение ручного и автоматизированного подхода

| | **Ручное** | **Автоматизированное** |
|---|---|---|
| **ЗАДАНИЕ ВХОДНЫХ ЗНАЧЕНИЙ** | Гибкость в заданииданных. 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 кнезначительным 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 выполнения тестового набора** | Низкая | Высокая |

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

### *Документация и сопровождение тестов*

#### *Тестовые процедуры*

Тестовые процедуры – это формальный документ, содержащий описание необходимых шагов для выполнения тестового набора. В случае ручных тестов тестовые процедуры содержат полное описание всех шагов и проверок, позволяющих протестировать продукт и вынести вердикт PASS/FAIL.

Процедуры должны быть составлены таким образом, чтобы любой инженер, не связанный с данным проектом, был способен адекватно провести цикл тестирования, обладая только самыми базовыми знаниями о применяющемся инструментарии.

В случае описания автоматизированных тестов тестовые процедуры должны содержать достаточную информацию для запуска тестов и анализа результатов.

#### *Описание тестов*

Описание тестов разрабатывается для облегчения анализа и поддержки тестового набора. Описание может быть реализовано в произвольной форме, но при этом должны выполняться следующие задачи:

1. Анализировать степень покрытия продукта тестами на основании описания тестового набора.
2. Для любой функции тестируемого продукта найти тесты, в которых функция используется.
3. Для любого теста определить все функции и их сочетания, которые данный тест использует (затрагивает).
4. Понять структуру и взаимосвязи тестовых файлов.
5. Понять принцип построения системы автоматизации тестирования.

#### *Документирование и жизненный цикл дефекта*

Каждый дефект, обнаруженный в процессе тестирования, должен быть задокументирован и отслежен. При обнаружении нового дефекта его заносят в базу дефектов. Для этого лучше всего использовать специализированные базы, поддерживающие хранение и отслеживание дефектов – типа *DDTS*. При занесении нового дефекта рекомендуется указывать, как минимум, следующую информацию:

1. Наименование подсистемы, в которой обнаружен дефект.
2. Версия продукта (номер build), на котором дефект был найден.
3. Описание дефекта.
4. Описание процедуры (шагов, необходимых для воспроизведения дефекта).

5. Номер теста, на котором дефект был обнаружен.
6. Уровень дефекта, то есть степень его серьезности с точки зрения критериев качества продукта или заказчика.

Занесенный в базу дефектов новый дефект находится в состоянии «New». После того, как команда разработчиков проанализирует дефект, он переводится в состояние «Open» с указанием конкретного разработчика, ответственного за исправление дефекта. После исправления дефект переводится разработчиком в состояние «Resolved». При этом разработчик должен указать следующую информацию:

1. Причину возникновения дефекта.
2. Место исправления, как минимум, с точностью до исправленного файла.
3. Краткое описание того, что было исправлено.
4. Время, затраченное на исправление.

После этого тестировщик проверяет, действительно ли дефект был исправлен и если это так, переводит его в состояние «Verified». Если тестировщик не подтвердит факт исправления дефекта, то состояние дефекта изменяется снова на «Open».

Если проектная команда принимает решение о том, что некоторый дефект исправляться не будет, то такой дефект переводится в состояние «Postponed» с указанием лиц, ответственных за это решение, и причин его принятия.

*Тестовый отчет*

Тестовый отчет обновляется после каждого цикла тестирования и должен содержать следующую информацию для каждого цикла:

1. Перечень функциональности в соответствии с пунктами требований, запланированный для тестирования на данном цикле, и реальные данные по нему.
2. Количество выполненных тестов – запланированное и реально исполненное.
3. Время, затраченное на тестирование каждой функции, и общее время тестирования.
4. Количество найденных дефектов.
5. Количество повторно открытых дефектов.
6. Отклонения от запланированной последовательности действий, если таковые имели место.
7. Выводы о необходимых корректировках в системе тестов, которые должны быть сделаны до следующего тестового цикла.

Пример фрагмента из тестового отчета представлен на рис. 10.3. Приведенный фрагмент отчета содержит примерные данные для четырех циклов тестирования и иллюстрирует структуру отчета. Такой вид отчет имеет после тестирования, перед началом цикла тестирования поля не заполнены, заполнение осуществляется по окончании соответствующего цикла.

## *Оценка качества тестов*

Тесты нуждаются в контроле качества так же, как и тестируемый продукт. Поскольку тесты для продукта являются своего рода эталоном его структурных и поведенческих характеристик, закономерен вопрос о том, насколько адекватен эталон. Для оценки качества тестов используются различные методы, наиболее популярные из которых кратко рассмотрены ниже.

*Тестовые метрики*

Существует устоявшийся набор тестовых метрик, который помогает определить эффективность тестирования и текущее состояние продукта. К таким метрикам относятся следующие:

1. Покрытие функциональных требований.
2. Покрытие кода продукта. Наиболее применимо для модульного уровня тестирования.
3. Покрытие множества сценариев.
4. Количество или плотность найденных дефектов. Текущее количество дефектов сравнивается со средним для данного типа продуктов с целью установить, находится ли оно в пределах допустимого статистического отклонения. При этом обнаруженные отклонения как в большую, так и в меньшую сторону приводят к анализу причин их появления и, если необходимо, к выработке корректирующих действий.
5. Соотношение количества найденных дефектов с количеством тестов на данную функцию продукта. Сильное расхождение этих двух величин говорит либо о неэффективности тестов (когда большое количество тестов находит мало дефектов) либо о плохом качестве данного участка кода (когда найдено большое количество дефектов на не очень большом количестве тестов).
6. Количество найденных дефектов, соотнесенное по времени, или скорость поиска дефектов. Если производная такой функции близка нулю, то продукт обладает качеством, достаточным для окончания тестирования и поставки заказчику.

## Обзоры тестов и стратегии

Тестовый код и стратегия тестирования, зафиксированные в виде документов, заметно улучшаются, если подвергаются коллективному обсуждению. Такие обсуждения называются обзорами (review). Существует принятая в организации процедура проведения и оценки результатов обзора. Обзоры наряду с тестированием образуют мощный набор методов борьбы с ошибками с целью повышения качества продукта. Цели обзоров тестовой стратегии и тестового кода различны.

Цели обзора тестовой стратегии:

1. Установить достаточность проверок, обеспечиваемых тестированием.
2. Проанализировать оптимальность покрытия или адекватность распределения количества планируемых тестов по функциональности продукта.
3. Проанализировать оптимальность подхода к разработке кода, генерации кода, автоматизации тестирования.

Цели обзора тестового кода:

1. Установить соответствие тестового набора тестовой стратегии.
2. Проверить правильность кодирования тестов.
3. Оценить достигнутую степень качества кода, исходя из требований по стандартам, простоте поддержки, наличию комментариев и т.п.
4. Если необходимо, проанализировать оптимальность тестового кода с целью удовлетворения требований к быстродействию и объему.