**Лекция 1**

**Основные понятия тестирования**

**Основные вопросы**

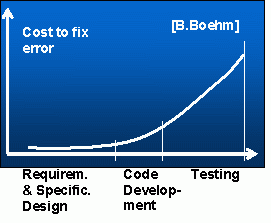
1. Способ обеспечения качества продукта. Общая концепция.
2. Основная терминология.
3. Организация тестирования.
4. Спецификация программы.
5. Разработка тестов.
6. Управляющий граф программы.
7. Основные проблемы тестирования.

**Способ обеспечения качества продукта. Общая концепция.**

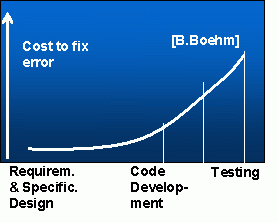
Качество программного продукта характеризуется набором свойств, определяющих, насколько продукт "хорош" с точки зрения заинтересованных сторон, таких как заказчик продукта, спонсор, конечный пользователь, разработчики и тестировщики продукта, инженеры поддержки, сотрудники отделов маркетинга, обучения 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. Таким образом, постановка задачи обеспечения качества продукта выливается в задачу определения заинтересованных лиц, их критериев качества и затем нахождения оптимального решения, удовлетворяющего этим критериям. Тестирование является одним из наиболее устоявшихся способов обеспечения качества разработки программного обеспечения и входит в набор эффективных средств современной системы обеспечения качества программного 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.

Оценка трудоемкости обнаружения и исправления ошибок при создании программного продукта

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

Аналогичная оценка при автоматизации тестирования

 Рис. 1.2. Аналогичная оценка при автоматизации тестирования

**Основная терминология.**

Отладка (debug, debugging) – процесс поиска, локализации и исправления ошибок в программе [IEEE Std.610-12.1990].

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 Under Testing (AUT) или Implementation Under Testing (IUT)).

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 сравнения словесного описания пункта спецификации с результатом выполнения фрагмента кода

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

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

Проверка результата выполнения возможна, когда результат вычисления заранее известен – 4. Если результат выполнения 22 = 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,Yв)\notin (X,Y) 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

Можно выводить промежуточные значения переменных при выполнении программы. Код, осуществляющий вывод, расположен ниже ([Пример 2.3](https://www.intuit.ru/studies/courses/48/48/lecture/1426?page=2#example.2.3)). Этот *метод* относится к наиболее популярным средствам автоматизации *отладки* программистов прошлых десятилетий. 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 (int i=1;n>=i;i++)

{

z = z\*x;

Console.WriteLine("i = {0} z = {1}",

i, z);

}

return z;

}

2.3. Исходный текст метода Power со вставкой оператора протоколирования

double Power(double x, int n)

{

double z=1;

int i;

for (i=1;n>=i;i++)

{

z = z\*x;

printf("i = %d z = %f\n",i,z);

}

return z;

}

2.3.1. Исходный текст метода Power со вставкой оператора протоколирования

1. *Пошаговое выполнение* программы (*single-step* running).

#### Пример пошагового выполнения программы

При *пошаговом выполнении* программы код выполняется строчка за строчкой. В среде Microsoft Visual Studio.NET возможны 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.

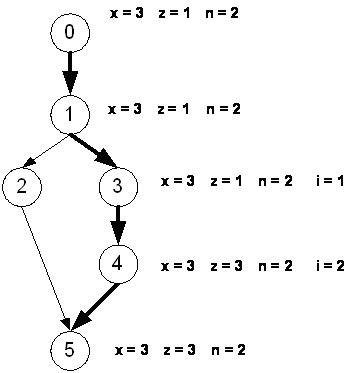
*Пошаговое выполнение* до сих пор является мощным методом автономного *тестирования* и *отладки* небольших программ.

1. Выполнение с заказанными *остановками (breakpoints)*, анализом трасс (traces) или состояний памяти - *дампов (dump)*.

#### 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 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 [Рис. 2.1](https://www.intuit.ru/studies/courses/48/48/lecture/1426?page=2#image.2.1) условно изображен *управляющий граф* некоторой программы. Трасса, проходящая через вершины 0-1-3-4-5 зафиксирована в [Табл. 2.1](https://www.intuit.ru/studies/courses/48/48/lecture/1426?page=2#table.2.1). Строки таблицы отображают вершины *управляющего графа* программы, или *breakpoints*, в которых фиксировались текущие значения заказанных пользователем переменных.

**Рис. 2.1.**Управляющий граф программы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 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.

1. 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:

1. n < 1 (ошибочное)
2. n > 100 (ошибочное)
3. n - не число (ошибочное)
4. 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.

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

Ожидаемый результат: Error : x must be in [0..999].

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

Ожидаемый результат: Error : n must be in [1..100].

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

Ожидаемый результат: Error : Please enter a numeric argument.

1. Проверка на граничные значения:
   1. Входные значения: (x = 999, n = 1).

Ожидаемый результат: The power n of x is 999.

* 1. 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 YВ. В процессе *тестирования* производится оценка результатов выполнения путем сравнения получаемого результата с ожидаемым.

**Разработка тестов.**

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]) 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.

**Управляющий граф программы.**

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

***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)

***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 конечноe число.

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

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;

}

2.7. Пример описания функции с реализуемыми и нереализуемыми путями

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;

}

2.7.1. Пример описания функции с реализуемыми и нереализуемыми путями

Например, для функции [Пример 2.7](https://www.intuit.ru/studies/courses/48/48/lecture/1426?page=4#example.2.7) *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**

* *Тестирование* программы на всех входных значениях невозможно.
* Невозможно *тестирование* и на всех *путях*.
* 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.