План-Конспект урока по Тестированию и отладке ПО

Тема Введение

1.1 Тестирование как элемент жизненного цикла ПО

Цель: Сформировать представление содержании, целях и задачах учебной дисциплины, ее роли в формировании профессиональных компетенций будущего специалиста Познакомить с базовыми теоретическими понятиями, которые лежат в основе отладки и тестирования программного обеспечения.

Задачи:

1) *образовательные:*
   -научить высказывать общее суждение о содержании, целях и задачах дисциплины;
   -научить руководствоваться характеристиками современных систем программирования при выборе средств реализации прикладных задач.
2) *развивающие:*
   - способствовать развитию исследовательских способностей учащихся;
   - способствовать формированию умений анализировать, сравнивать, обобщать и делать выводы по полученному материалу;
3) *воспитательные:*
   - воспитывать ценностные отношения к научному знанию;
   - активизировать познавательный интерес к научной дисциплине;
   - способствовать формированию доброжелательных взаимоотношений друг с другом;

Ход урока

1. Организационный момент

Проверка отсутствующих, выяснение причин отсутствия и уровень подготовленности учащихся к занятию.

2. Актуализация опорных знаний
3. Изучение нового материала

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.

Отладка (debug, debugging) – процесс поиска, локализации и исправления ошибок в программе [9] [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.

Тестирование разделяют на статическое и динамическое:

Статическое тестирование выявляет формальными методами анализа без выполнения тестируемой программы неверные конструкции или неверные отношения объектов программы (ошибки формального задания) с помощью специальных инструментов контроля кода – CodeChecker.

Динамическое тестирование (собственно тестирование) осуществляет выявление ошибок только на выполняющейся программе с помощью специальных инструментов автоматизации тестирования – Testbed [9] или Testbench.

Отсюда вывод:

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.

4. Подведение итогов, рефлексия

   Посмотрите на поставленную перед вами цель урока. Подумайте, достигнута ли она вами лично и группой в целом. Что для вас оказалось трудным для восприятия. Вспомним основные понятия и определения.

5. Домашнее задание.

План-Конспект урока по Тестированию и отладке ПО

Тема 1.2 Типы процессов тестирования

Цель: Сформировать знания о типах процессов тестирования и верификации.

Задачи:

1) образовательные:

-научить высказывать общее суждение о содержании, целях и задачах дисциплины;

-научить руководствоваться характеристиками современных систем программирования при выборе средств реализации прикладных задач.

2) развивающие:

- способствовать развитию исследовательских способностей учащихся;

- способствовать формированию умений анализировать, сравнивать, обобщать и делать выводы по полученному материалу;

3) воспитательные:

- воспитывать ценностные отношения к научному знанию;

- активизировать познавательный интерес к научной дисциплине;

- способствовать формированию доброжелательных взаимоотношений друг с другом;

Ход урока

1. Организационный момент

Проверка отсутствующих, выяснение причин отсутствия и уровень подготовленности учащихся к занятию.

2. Актуализация опорных знаний

3. Изучение нового материала

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.

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 условно изображен управляющий граф некоторой программы. Трасса, проходящая через вершины 0-1-3-4-5 зафиксирована в Табл. 2.1. Строки таблицы отображают вершины управляющего графа программы, или breakpoints, в которых фиксировались текущие значения заказанных пользователем переменных.

Управляющий граф программы

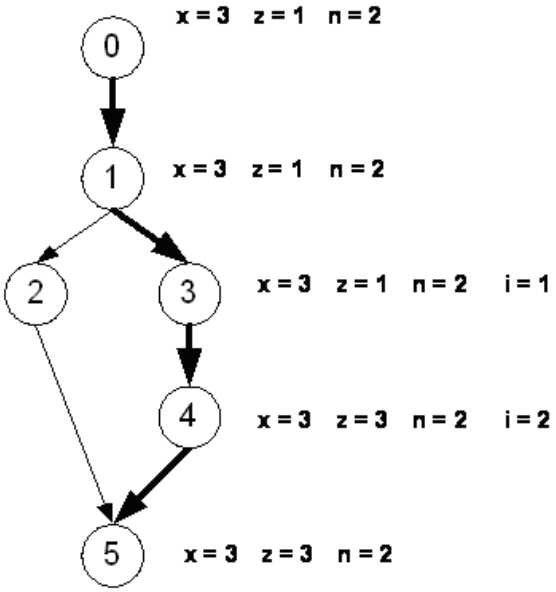


Рис. 2.1. Управляющий граф программы

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.

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.

В программе на Пример 2.4 фиксируются значения всех переменных после выполнения каждого оператора.

```
//  Метод вычисляет неотрицательную
//  степень n числа x
static public double PowerNonNeg(double x,
                          int n)
{
double z=1;
Console.WriteLine("x={0} z={1} n={2}",
              x,z,n);
 if (n>0)
 {
 Console.WriteLine("x={0} z={1} n={2}",
                x,z,n);
  for (int i=1;n>=i;i++)
  {
   z = z*x;
   Console.WriteLine(
       "x={0} z={1} n={2}" +
       " i={3}",x,z,n,i);
  }
 }
 else Console.WriteLine(
      "Ошибка ! Степень" +
      " числа n должна быть больше 0.");
 return z;
}
```

Зная структуру управляющего графа программы и имея значения всех переменных после выполнения каждого оператора, можно осуществить обратное

выполнение (например, в уме), подставляя значения переменных в операторы и двигаясь снизу вверх, начиная с последнего.

Итак, в процессе тестирования сравнение промежуточных результатов с полученными независимо эталонными результатами позволяет найти причины и место ошибки, исправить текст программы, провести повторную трансляцию и настройку на выполнение и продолжить тестирование.

Тестирование заканчивается, когда выполнилось или "прошло" (pass) успешно достаточное количество тестов в соответствии с выбранным критерием тестирования.

Тестирование – это:

Процесс выполнения ПО системы или компонента в условиях анализа или записи получаемых результатов с целью проверки (оценки) некоторых свойств тестируемого объекта.

The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component [9] .

Процесс анализа пункта требований к ПО с целью фиксации различий между существующим состоянием ПО и требуемым (что свидетельствует о проявлении ошибки) при экспериментальной проверке соответствующего пункта требований.

The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate features of software items [[IEEE Std.610-12.1990], [9] .

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

Реализация тестирования разделяется на три этапа:

Создание тестового набора (test suite) путем ручной разработки или автоматической генерации для конкретной среды тестирования (testing environment).

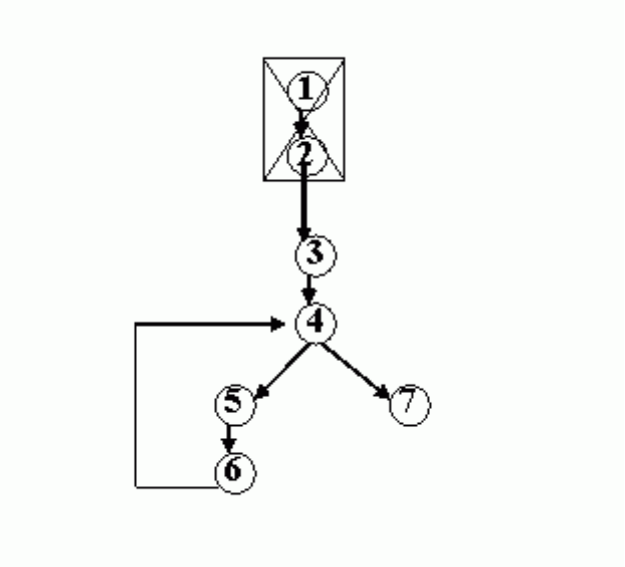
Прогон программы на тестах, управляемый тестовым монитором (test monitor, test driver [IEEE Std 829-1983], [9]) с получением протокола результатов тестирования (test log).

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.

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

2.6. Пример простой программы на языке C#



Управляющий граф программы (УГП) на Рис. 2.2 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(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 конечное 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 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 разрядными словами, тогда мощность множества тестов ||(X,Y)||=2**128

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

На Рис. 2.3 приведен фрагмент схемы программы управления схватом робота, где интервал между моментами срабатывания схвата не определен.

Этот тривиальный пример требует прогона бесконечного множества последовательностей входных значений с разными интервалами срабатывания схвата (Пример 2.8).

4. Подведение итогов, рефлексия

Посмотрите на поставленную перед вами цель урока. Подумайте, достигнута ли она вами лично и группой в целом. Что для вас оказалось трудным для восприятия. Вспомним основные понятия и определения.

5. Домашнее задание.

План-Конспект урока по Тестированию и отладке ПО

Тема 1.3 Критерии выбора тестов

Цель: Сформировать понятие о требованиях к идеальному критерию. Сформировать знания о классах критериев выбора тестов. Дать понятие об оценке покрытия программы и проекта. Сформировать понятие о тест требованиях.

Задачи:

1) образовательные:

-научить высказывать общее суждение о содержании, целях и задачах дисциплины;

-научить руководствоваться характеристиками современных систем программирования при выборе средств реализации прикладных задач.

2) развивающие:

- способствовать развитию исследовательских способностей учащихся;

- способствовать формированию умений анализировать, сравнивать, обобщать и делать выводы по полученному материалу;

3) воспитательные:

- воспитывать ценностные отношения к научному знанию;

- активизировать познавательный интерес к научной дисциплине;

- способствовать формированию доброжелательных взаимоотношений друг с другом;

Ход урока

1. Организационный момент

Проверка отсутствующих, выяснение причин отсутствия и уровень подготовленности учащихся к занятию.

2. Актуализация опорных знаний

3. Изучение нового материала

Требования к идеальному критерию были выдвинуты в работе [11] :

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

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

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

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

Структурные критерии используют информацию о структуре программы (критерии так называемого "белого ящика")

Функциональные критерии формулируются в описании требований к программному изделию (критерии так называемого "черного ящика")

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

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 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто - 2, или числом классов выходных путей).

На пример 3.1 приведен пример простой программы. Рассмотрим условия ее тестирования в соответствии со структурными критериями.

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

3.1. Пример простой программы, для тестирования по структурным критериям

```
1   void Method (int *x)
    {
2       if (*x>17)
3           *x = 17-*x;
4       if (*x==-13)
```

5 *x = 0;

6 }

3.1.1. Пример простой программы, для тестирования по структурным критериям

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

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

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

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

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

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

Набор условий для двух операторов if с метками 2 и 4 приведен в таблица 3.1

Таблица 3.1. Условия операторов if (30,0) (17,17) (-13,0) (21,-4)

2 if (x>17) > \le \le >

4 if (x==-13) \ne \ne = \ne

Критерий путей C2 проверяет программу более тщательно, чем критерии - C1, однако даже если он удовлетворен, нет оснований утверждать, что программа реализована в соответствии со спецификацией.

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

Структурные критерии не проверяют соответствие спецификации, если оно не отражено в структуре программы. Поэтому при успешном тестировании программы по критерию C2 мы можем не заметить ошибку, связанную с невыполнением некоторых условий спецификации требований.

Функциональные критерии (класс II)

Функциональный критерий - важнейший для программной индустрии критерий тестирования. Он обеспечивает, прежде всего, контроль степени выполнения требований заказчика в программном продукте. Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением. При функциональном тестировании преимущественно используется модель "черного ящика". Проблема функционального тестирования - это, прежде всего, трудоемкость; дело в том, что документы, фиксирующие требования к программному изделию (Software requirement specification, 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 объединяет отчасти особенности структурных и функциональных критериев. Он базируется на модели "полупрозрачного ящика", где явно указаны не только входы и выходы тестируемого компонента, но также состав и структура используемых методов (функций, процедур) и классов.

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

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

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 ув\in \{ у\} значению у, 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 [12] , согласно которой тестирование прекращается, если оцененный интервал времени между текущей ошибкой и следующей слишком велик для фазы тестирования приложения.

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.

4. Подведение итогов, рефлексия

Посмотрите на поставленную перед вами цель урока. Подумайте, достигнута ли она вами лично и группой в целом. Что для вас оказалось трудным для восприятия. Вспомним основные понятия и определения.

5. Домашнее задание.

План-Конспект урока по Тестированию и отладке ПО

Тема 1.4. Методы тестирования

Цель: Сформировать представление о методах тестирования ПО. Сформировать знания о составе тестового окружения.

Задачи:

1) образовательные:

-научить высказывать общее суждение о содержании, целях и задачах дисциплины;

-научить руководствоваться характеристиками современных систем программирования при выборе средств реализации прикладных задач.

2) развивающие:

- способствовать развитию исследовательских способностей учащихся;

- способствовать формированию умений анализировать, сравнивать, обобщать и делать выводы по полученному материалу;

3) воспитательные:

- воспитывать ценностные отношения к научному знанию;

- активизировать познавательный интерес к научной дисциплине;

- способствовать формированию доброжелательных взаимоотношений друг с другом;

Ход урока

1. Организационный момент

Проверка отсутствующих, выяснение причин отсутствия и уровень подготовленности учащихся к занятию.

2. Актуализация опорных знаний

3. Изучение нового материала

Случайные методы

Когда из-за ограничений по времени использование метода повторного прогона всех тестов невозможно, а программные средства отбора тестов недоступны, инженеры, ответственные за тестирование, могут выбирать тесты случайным образом или на основании "догадок", то есть предположительного соотнесения тестов с функциональными возможностями на основании предшествующих знаний или опыта. Например, если известно, что некоторые тесты задействуют особенно важные функциональные возможности или обнаруживали ошибки ранее, их было бы неплохо использовать также и для тестирования измененной программы. Один простой метод такого рода предусматривает случайный отбор предопределенного процента тестов из T. Подобные случайные методы принято обозначать random(x), где x - процент выбираемых тестов.

Случайные методы оказываются на удивление дешевыми и эффективными. Случайно выбранные входные данные могут давать больший разброс по покрытию кода, чем входные данные, которые используются в наборах тестов, основанных на покрытии, в одних случаях дублируя покрытие, а в других не обеспечивая его. При небольших интервалах тестирования их эффективность может быть как очень высокой, так и очень низкой. Это приводит и к большему разбросу статистики отбора тестов для таких наборов. Однако при увеличении интервала тестирования этот

разброс становится значительно меньше, и средняя эффективность случайных методов приближается к эффективности метода повторного прогона всех тестов с небольшими отклонениями для разных попыток. Таким образом, в последнем случае пользователь случайных методов может быть более уверен в их эффективности. Вообще, детерминированные методы эффективнее случайных методов, но намного дороже, поскольку выборочные стратегии требуют большого количества времени и ресурсов при отборе тестов.

Безопасные методы

Метод выборочного регрессионного тестирования называется безопасным, если при некоторых четко определенных условиях он не исключает тестов (из доступного набора тестов), которые обнаружили бы ошибки в измененной программе, то есть обеспечивает выбор всех тестов, обнаруживающих изменения. Тест называется обнаруживающим изменения, если его выходные данные при прогоне на P' отличаются от выходных данных при прогоне на P: P(t) \ne P'(t). Тесты, активизирующие измененный код, называются выполняющими изменение.

Выбор всех выполняющих изменение тестов является безопасным, но при этом отбираются некоторые тесты, не обнаруживающие изменений. Безопасный метод может включать в T' подмножество тестов, выходные данные которых для P и P' ни при каких условиях не отличаются. Поскольку не существует методики, кроме собственно выполнения теста, позволяющей для любой P' определить, будут ли выходные данные теста различаться для P и P', ни один метод не может быть безопасным и абсолютно точным одновременно. T' является безопасным подмножеством T тогда и только тогда, когда:

P(t) \ne P'(t) \Rightarrow t\in T'

Если P и P' выполняются в идентичных условиях и T' является безопасным подмножеством T, исполнение T' на P' всегда обнаруживает любые связанные с изменениями ошибки в P, которые могут быть найдены путем исполнения T. Если существует тест, обнаруживающий ошибку, безопасный метод всегда находит ее. Таким образом, ни один случайный метод не обладает такой же эффективностью обнаружения ошибок, как безопасный метод.

Методы минимизации

Процедура минимизации набора тестов ставит целью отбор минимального (в терминах количества тестов) подмножества T, необходимого для покрытия каждого элемента программы, зависящего от изменений. Для проверки корректности программы используются только тесты из минимального подмножества.

Матрица покрытия тестируемого кода

№	Строка кода	Тест				
		1	2	3	4	5
1	`Double Equation(int Print, float A, float B, float C, float& X1, float& X2) {`	*	*	*	*	*
2	`  float D = B * B - 4.0 * A * C;`	*	*	*	*	*
3	`  if (D >= 0) {`	*	*	*	*	*
4	`   X1 = (-B + sqrt(D)) / 2.0 / A;`	*		*	*	
5	`   X2 = (-B - sqrt(D)) / 2.0 / A; } else {`	*		*	*	
6	`   X1 = -B / 2.0 / A;`		*			*
7	`   X2 = sqrt(D); }`		*			*
8	`  if (Print)`	*	*	*	*	*
9	`   printf("Solution: %f, %f\n", X1, X2);`	*	*			
10	`  return D;`	*	*	*	*	*
11	`}`	*	*	*	*	*

Рис. 12.1. Матрица покрытия тестируемого кода

Обоснование применения методов минимизации состоит в следующем:

Корреляция между эффективностью обнаружения ошибок и покрытием кода выше, чем между эффективностью обнаружения ошибок и размером множества тестов. Неэффективное тестирование, например многочасовое выполнение тестов, не увеличивающих покрытие кода, может привести к ошибочному заключению о корректности программы.

Независимо от способа порождения исходного набора тестов, его минимальные подмножества имеют преимущество в размере и эффективности, так как состоят из меньшего количества тестов, не ослабляя при этом способности к обнаружению ошибок или снижая ее незначительно.

Вообще говоря, сокращенный набор тестов, отобранный при минимизации, может обнаруживать ошибки, не обнаруживаемые сокращенным набором того же размера, выбранным случайным или каким-либо другим способом. Такое преимущество минимизации перед случайными методами в эффективности является закономерным. Однако из всех детерминированных методов минимизация приводит к созданию наименее эффективных наборов тестов, хотя и самых маленьких. В частности, безопасные методы эффективнее методов минимизации, хотя и намного дороже.

Минимизация набора тестов требует определенных затрат на анализ. Если стоимость этого анализа больше затрат на выполнение некоторого порогового числа тестов, существует более дешевый случайный метод, обеспечивающий такую же эффективность обнаружения ошибок.

Хотя минимальные наборы тестов могут обеспечивать структурное покрытие измененного кода, зачастую они не являются безопасными, поскольку очевидно, что некоторые тесты, потенциально способные обнаруживать ошибки, могут остаться за чертой отбора. Набор функциональных тестов обычно не обладает избыточностью в том смысле, что никакие два теста не покрывают одни и те же функциональные требования. Если тесты исходно создавались по критерию структурного покрытия, минимизация приносит плоды, но когда мы имеем дело с функциональными тестами, предпочтительнее не отбрасывать тесты, потенциально способные обнаруживать ошибки. В существующей практике тестирования инженеры предпочитают не заниматься минимизацией набора тестов.

Методы, основанные на покрытии кода

Значение методов, основанных на покрытии кода, состоит в том, что они гарантируют сохранение выбранным набором тестов требуемой степени покрытия элементов P' относительно некоторого критерия структурного покрытия C, использовавшегося при создании первоначального набора тестов. Это не означает, что если атрибут программы, определенный C, покрывается первоначальным множеством тестов, он будет также покрыт и выбранным множеством; гарантируется только сохранение процента покрываемого кода. Методы, основанные на покрытии, уменьшают разброс по покрытию, требуя отбора тестов, активирующих труднодоступный код, и исключения тестов, которые только дублируют покрытие. Поскольку на практике критерии покрытия кода обычно применяются для отбора единственного теста для каждого покрываемого элемента, подходы, основанные на покрытии кода, можно рассматривать как специфический вид методов минимизации.

Разновидностью методов, основанных на покрытии кода, являются методы, которые базируются на покрытии потока данных. Эти методы эффективнее методов минимизации и почти столь же эффективны, как безопасные методы. В то же время, они могут требовать, по крайней мере, такого же времени на анализ, как и наиболее эффективные безопасные методы, и, следовательно, могут обходиться дороже безопасных методов и намного дороже других методов минимизации. Они имеют тенденцию к включению избыточных тестов в набор регрессионных тестов для покрытия зависящих от изменений пар определения-использования, что, в некоторых случаях, ведет к большому числу отобранных тестов. Этот факт зафиксирован экспериментально.

Методы, основанные на использовании потока данных, могут быть полезны и для других задач регрессионного тестирования, кроме отбора тестов, например, для нахождения элементов P, недостаточно тестируемых T'.

Метод стопроцентного покрытия измененного кода аналогичен методу минимизации. Так, для примера таблицы с Рис. 12.1 существует 4 способа отобрать 2 теста в соответствии с этим критерием. Одного теста недостаточно. Результаты сравнения методов выборочного регрессионного тестирования приведены в Табл. 12.1.

Таблица 12.1. Сравнение методов *выборочного регрессионного тестирования*

Класс методов	Случайные	Безопасные	Минимизации	Покрытия
Полнота	От 0% до 100%	100%	< 100%	< 100%
Размер набора тестов	Настраивается	Большой	Небольшой	Зависит от параметров метода
Время выполнения метода	Пренебрежимо мало	Значительное	Значительное	Значительное
Перспективные свойства методов регрессионного тестирования	Отсутствие средства поддержки регрессионного тестирования	Высокие требования по качеству	Стоимость пропуска ошибки невелика	Набор исходных тестов создается по критерию покрытия

4. Подведение итогов, рефлексия

Посмотрите на поставленную перед вами цель урока. Подумайте, достигнута ли она вами лично и группой в целом. Что для вас оказалось трудным для восприятия. Вспомним основные понятия и определения.

5. Домашнее задание.