**Лекция Критерии выбора тестов**

**Критерии выбора тестов**

**Аннотация:**Рассматриваются требования к идеальному критерию тестирования и классы частных критериев. Рассматриваются особенности применения структурных и функциональных критериев на базе конкретных примеров. Рассматриваются особенности применения методов стохастического тестирования и метод оценки скорости выявления ошибок. Описывается мутационный критерий и на примере иллюстрируется техника работы с ним.

**Ключевые слова:**идеальный критерий тестирования,

структурный критерий,  функциональный критерий,  стохастическое тестирование,  мутационный критерий, модель программы, знание, спецификация программы,  информация,  класс,  интеграционное тестирование,  unit testing, integration testing,  Автоматизация тестирования,  число классов,  тестовый набор,  операторы, C2,  программа, контроль,  функциональное тестирование,  software,  specification,  functional specification,  фаза тестирования,  мощность,  прогон тестов, оценка скорости выявления ошибок, кортеж, Произведение, плотность ошибок, базы данных, интервал между ошибками, поиск, интервал, перестановка, цикла, мутация, мутант, правильная программа

**Требования к идеальному критерию тестирования**

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

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**

1. *Структурные критерии* используют информацию о структуре программы (критерии так называемого "белого ящика")
2. *Функциональные критерии* формулируются в описании требований к программному изделию ( **критерии так называемого "черного ящика"** )
3. Критерии *стохастического тестирования* формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы.
4. *Мутационные критерии* ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.

**Структурные критерии (класс I).**

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

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

* Условие критерия **тестирования команд** (критерий С0) - набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза. Это слабый критерий, он, как правило, используется в больших программных системах, где другие критерии применить невозможно.
* Условие критерия **тестирования ветвей** (критерий С1) - набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза. Это достаточно сильный и при этом экономичный критерий, поскольку множество ветвей в тестируемом приложении конечно и не так уж велико. Данный критерий часто используется в системах *автоматизации тестирования*.
* Условие критерия **тестирования путей** (критерий С2) - набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто - 2, или *числом классов* выходных путей).

На  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 С0 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 c метками 2 и 4 приведен в  таблице 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 С2 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 х=-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.

**Пример применения функциональных критериев тестирования для разработки набора тестов по критерию классов входных данных**

Пусть для решения задачи тестирования системы "Система управления автоматизированным комплексом хранения подшипников" (см. Приложение 1, FS) был разработан следующий фрагмент спецификации требований:

1. Произвести опрос статуса склада (вызвать функцию GetStoreStat ). Добавить в журнал сообщений запись "СИСТЕМА : Запрошен статус СКЛАДА". В зависимости от полученного значения произвести следующие действия:
   * Полученный статус склада = 32. В приемную ячейку склада поступил подшипник. Система должна:
     1. Добавить в журнал сообщений запись "СКЛАД : Статус СКЛАДА = 32".
     2. Получить параметры поступившего подшипника с терминала подшипника (должна быть вызвана функцияGetRollerPar ).
     3. Добавить в журнал сообщений запись "СИСТЕМА: Запрошены параметры подшипника".
     4. В зависимости от возвращенного функцией GetRollerPar значения должны быть выполнены следующие действия ( таблица 3.2):

Таблица 3.2 - Действия по результатам функции GetRollerPar

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

1. Произвести опрос терминала оси (вызвать функцию получения сообщения от терминала - GetAxlePar ). В журнал сообщений должно быть добавлено сообщение "СИСТЕМА : Запрошены параметры оси". В зависимости от возвращенного функцией GetAxlePar значения должны быть выполнены следующие действия ( таблица 3.3):
2. Таблица 3.3 - Действия по результатам функции GetAxlePar

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

Определим классы входных данных для параметра - статус склада:

1. Статус склада = 0 (правильный).
2. Статус склада = 4 (правильный).
3. Статус склада = 16 (правильный).
4. Статус склада = 32 (правильный).
5. Статус склада = любое другое значение (ошибочный).

Теперь рассмотрим тестовые случаи:

1. Тестовый случай 1 (покрывает класс 4):

Состояние окружения (входные данные - X ):

Статус склада - 32.

...

Ожидаемая последовательность событий (выходные данные - Y ):

Система запрашивает статус склада (вызов функции GetStoreStat ) и получает 32

...

1. Тестовый случай 2 (покрывает класс 5):

Состояние окружения (входные данные - X ):

Статус склада - 12dfga.

...

Ожидаемая последовательность событий (выходные данные - Y ):

Система запрашивает статус склада (вызов функции GetStoreStat ) и согласно пункту спецификации при ошибочном значении статуса склада в журнал добавляется сообщение "СКЛАД : ОШИБКА : Неопределенный статус".

...

**Стохастические критерии (класс III)**

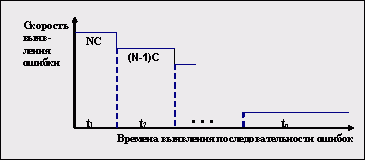
*Стохастическое тестирование* применяется при тестировании сложных программных комплексов - когда набор детерминированных тестов (X,Y) имеет громадную *мощность*. В случаях, когда подобный набор невозможно разработать и исполнить на *фазе тестирования*, можно применить следующую методику.

* 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 yв\in \{ y\} значению y, полученному в результате *прогона теста* на наборе {x} - случайной последовательности входных сигналов, сгенерированной имитатором.
  + Стохастический контроль - проверка соответствия множества значений {yВ}, полученного в результате прогона тестов на наборе входных значений {x}, заранее известному распределению результатов F(Y).

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

**Критерии****стохастического тестирования**

* **Cтатистические 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.



**Рисунок 3.1 -**Зависимость скорости выявления ошибок от времени выявления

При формализации модели скорости выявления ошибок ( рис. 3.1) использовались следующие обозначения:

N - исходное число ошибок в программном комплексе перед тестированием,

C - константа снижения скорости выявления ошибок за счет нахождения очередной ошибки,

t1, t2,… tn - *кортеж* возрастающих интервалов обнаружения последовательности из n ошибок,

T - время выявления n ошибок.

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

(1) (N-i+1)\*C\*ti = 1

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

(2) N*C*t_{1}+(N-1)*C*t_{2}+…+(N-n+1)*C*t_{n}=n 
\\
    N*C*(t_{1}+t_{2}+…+t_{n}) - C*\Sigma (i-1)t_{i} = n
\\
    NCT - C*\Sigma (i-1)t_{i} = n

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


(3) \Sigma 1/(N-i+1) = TC

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

(4) C = n/(NT - \Sigma (i-1)t_{i})

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

(5) \Sigma 1/(N-i+1) = n/(N - 1/T*\Sigma (i-1)t_{i})

Если оценить величину N приблизительно, используя известные методы оценки числа ошибок в программе [ 2 ] , [ 13 ] или данные о *плотности ошибок* для проектов рассматриваемого класса из исторической *базы данных* проектов, и, кроме того, использовать текущие данные об *интервалах между ошибками* t1, t2 … tn, полученные на *фазе тестирования*, то, подставляя эти данные в (5), можно получить оценку tn+1 -временного интервала необходимого для нахождения и исправления очередной ошибки (будущей ошибки).

Если tn+1>Td - допустимого времени тестирования проекта, то тестирование заканчиваем, в противном случае продолжаем *поиск* ошибок.

Наблюдая последовательность интервалов ошибок t1, t2 … tn, и время, потраченное на выявление n ошибок T=\Sigma t_{i}, можно прогнозировать *интервал* времени до следующей ошибки и уточнять в соответствии с (4) величину C.

Критерий Moranda очень практичен, так как опирается на информацию, традиционно собираемую в процессе тестирования.

**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.

**Пример применения мутационного критерия**

Тестируемая программа P приведена на пример 3.2. Для нее создается две программы-мутанта P1 и P2.

В P1 изменено начальное значение переменной z с 1 на 2 ( пример 3.3).

В P2 изменено начальное значение переменной i с 1 на 0 и граничное значение индекса цикла с n на n-1 ( пример 3.4).

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

**// Метод вычисляет неотрицательную**

**// степень n числа x**

**static public double PowerNonNeg(double x, int n)**

**{**

**double z=1;**

**if (n>0)**

**{**

**for (int i=1;n-1>=i;i++)**

**{**

**z = z\*x;**

**}**

**}**

**else Console.WriteLine(**

**"Ошибка ! Степень числа n должна**

**быть больше 0.");**

**return z;**

**}**

3.2. Основная программа P

**double PowerNonNeg(double x, int n)**

**{**

**double z=1;**

**int i;**

**if (n>0)**

**{**

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

**{**

**z = z\*x;**

**}**

**}**

**else printf(**

**"Ошибка ! Степень числа n должна**

**быть больше 0.\n");**

**return z;**

**}**

3.2.1. Основная программа P

Измененное начальное значение переменной z в *мутанте* Р1 ( z=2 ):

**// Метод вычисляет неотрицательную**

**// степень n числа x**

**static public double PowerMutant1(double x, int n)**

**{**

**double z=2;**

**if (n>0)**

**{**

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

**{**

**z = z\*x;**

**}**

**}**

**else Console.WriteLine(**

**"Ошибка ! Степень числа n должна**

**быть больше 0.");**

**return z;**

**}**

3.3. Программа мутант P1.

**double PowerMutant1(double x, int n)**

**{**

**double z=2;**

**int i;**

**if (n>0)**

**{**

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

**{**

**z = z\*x;**

**}**

**}**

**else printf(**

**"Ошибка ! Степень числа n должна**

**быть больше 0.\n");**

**return z;**

**}**

3.3.1. Программа мутант P1.

Измененное начальное значение переменной i и границы цикла в *мутанте* P2 ( i=0;n-1 ):

**// Метод вычисляет неотрицательную**

**// степень n числа x**

**static public double PowerMutant2(double x, int n)**

**{**

**double z=1;**

**if (n>0)**

**{**

**for (int i=0;n-1>=i;i++)**

**{**

**z = z\*x;**

**}**

**}**

**else Console.WriteLine(**

**"Ошибка ! Степень числа n должна**

**быть больше 0");**

**return z;**

**}**

3.4. Программа-мутант P2.

**double PowerMutant2(double x, int n)**

**{**

**double z=1;**

**int i;**

**if (n>0)**

**{**

**for (i=0;n-1>=i;i++)**

**{**

**z = z\*x;**

**}**

**}**

**else printf(**

**"Ошибка ! Степень числа n должна**

**быть больше 0.\n");**

**return z;**

**}**

3.4.1. Программа-мутант P2.

## **Контрольные вопросы**

1. Перечислите требования к идеальному критерию тестирования?
2. Раскройте понятие достаточность критерия тестирования.
3. Раскройте понятие полноты критерия тестирования.
4. Раскройте понятие надёжности критерия тестирования.
5. Раскройте понятие проверяемость критерия тестирования.
6. Перечислите классы критериев тестирования?
7. Что такое структурные критерии тестирования?
8. Что такое функциональные критерии тестирования?
9. Что такое мутационные критерии тестирования?
10. Что такое критерии стохастического тестирования?
11. На чём базируются структурные критерии тестирования?
12. В чём состоит условие критерия тестирования команд (критерий С0)?
13. В чём состоит условие критерия **тестирования ветвей** (критерий С1)?
14. В чём состоит условие критерия **тестирования путей** (критерий С2)?
15. Приведите частные виды функциональных критериев.
16. Что такое Стохастическое тестирование?
17. В чём сущность Мутационного критерия (класс IV) тестирования?