Позитивные и негативные тесты. Классы данных для тестов. Классы эквивалентности. Покрытие программного кода

Parallels™

Profit from the Cloud

02 Июля 2014

Стельмашенко Светлана

Содержание:

1. Позитивные и негативные тесты
2. Техники анализа классов эквивалентности и граничных значений
3. Покрытие кода

Позитивные тесты

Предполагают нормальное, «правильное»

- **использование и/или**
- **работу системы.**

Негативные тесты

Проверяют ситуацию, связанную с:

- **потенциальной ошибкой (error) пользователя и/или**
- **потенциальным дефектом (failure) в системе**

Что нужнее?

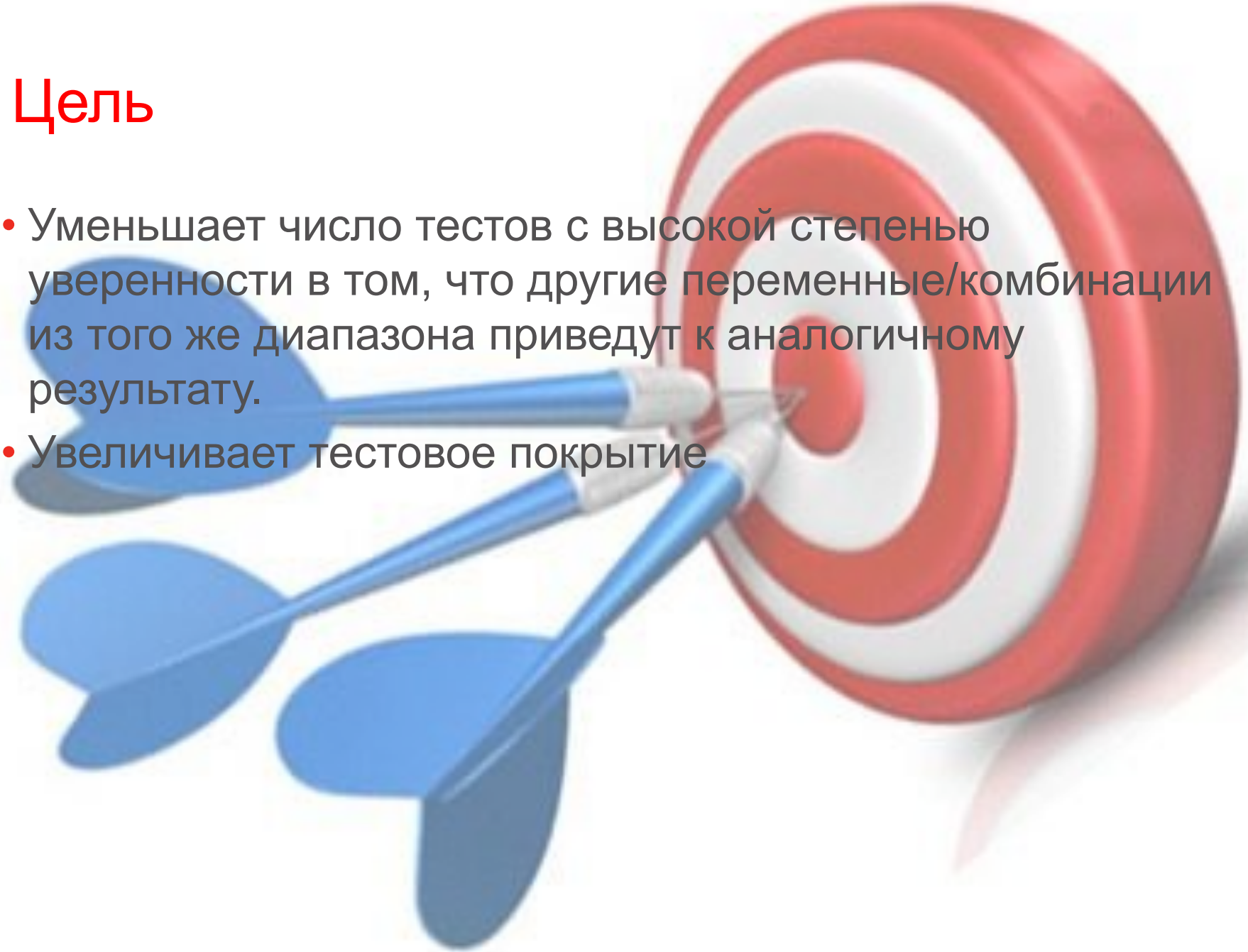
Техники анализа классов эквивалентности и граничных значений

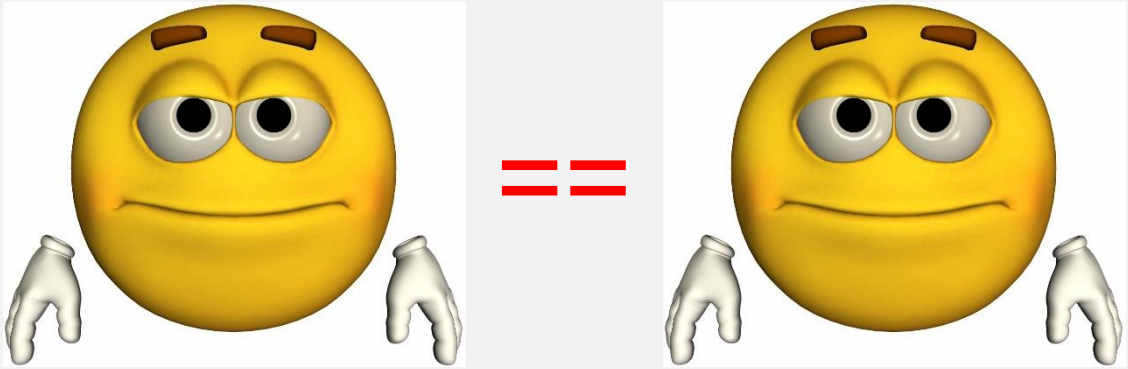
Почему?

- Могут использоваться на разных уровнях – от отдельных функций до целого продукта.
- Многие тестировщики пользуются ими интуитивно каждый день.
- Неправильное использование этих техник может привести к пропуску серьезных ошибок.

Цель

- Уменьшает число тестов с высокой степенью уверенности в том, что другие переменные/комбинации из того же диапазона приведут к аналогичному результату.
- Увеличивает тестовое покрытие





Два теста считаются эквивалентными, если приводят к одному и тому же результату.

- Они тестируют одну и ту же вещь (функцию, модуль, часть системы).

- Если один из тестов ловит ошибку, то другой скорее всего тоже её поймает.

- Если один из них не ловит ошибку, то другой скорее всего тоже не поймает

Входные параметры, которые приводят к одинаковому поведению программы, мы будем считать эквивалентными.

Классы эквивалентности

- это одно или больше значений ввода, к которым ПО применяет одинаковую логику.

Техника анализа граничных значений

- Техника проверки ошибок на границах классов эквивалентности.

Алгоритм:

- выделить классы эквивалентности.
- определить граничные значения этих классов.
- определить к какому классу будет относиться каждая граница.
- Определить уникальные данные (исключения)
- Для каждой границы нам нужно провести тесты по проверке значения до границы, на границе, и сразу после границы.

Покрытие кода (code coverage)

насколько исходный код программы был протестирован.

Виды:

1. Покрытие требований к ПО (на основе спецификаций, документаций, здравого смысла)
2. Покрытие кода (белый ящик)

Tcov = (Ltc/Lcode) * 100%
где:
Tcov - тестовое покрытие
Ltc - кол-ва строк кода, покрытых тестами
Lcode - общее кол-во строк кода.

Tcov = (Lcov/Ltotal) * 100%
где:
Tcov - тестовое покрытие
Lcov - количество требований, проверяемых тест кейсами
Ltotal - общее количество требований

Уровни покрытия:

- **По строкам программного кода (Statement Coverage)**

каждый оператор должен быть выполнен хотя бы один раз.

В данном фрагменте кода входными переменными являются prev и ShowMessage.

```
if (prev == "оператор" || prev == "унарный оператор") {
    if (ShowMessage)  {
        MessageBox.Show(“text1");
    }  else   {
       Log.Write(" text2 ");
    }
    Program.res = 4;
    return "&Error 04";
}
```

	1	2
prev	оператор	оператор
ShowMessage	true	false

Уровни покрытия:

- **По веткам условных операторов (Decision Coverage)**

Каждая точка входа и выхода в программе и во всех ее функциях должна быть выполнена по крайней мере один раз и все логические выражения в программе должны принять каждое из возможных значений хотя бы один раз; таким образом, для покрытия по веткам требуется как минимум два тестовых примера.

первый условный оператор if имеет неявную ветвь – пустую ветвь else. Для обеспечения покрытия по веткам необходимо покрывать и пустые ветви.

	1	2	3
prev	оператор	оператор	операнд
ShowMessage	true	false	true

Уровни покрытия:

- **Покрытие по условиям (Condition Coverage)**

каждая компонента логического условия в результате выполнения тестовых примеров должна принимать все возможные значения, но при этом не требуется, чтобы само логическое условие принимало все возможные значения.

```
if (condition1 || condition2)
  MethodA();
else
  MethodB();
```

	1	2
condition1	true	False
condition2	false	True

Уровни покрытия:

- **Покрытие по веткам/условиям (Condition/Decision Coverage)**

для обеспечения полного покрытия необходимо, чтобы как логическое условие, так и каждая его компонента приняла все возможные значения.

```
if (condition1 || condition2)
  MethodA();
else
  MethodB();
```

	1	2
condition1	true	false
condition2	true	false

Уровни покрытия:

- **Покрытие по всем условиям (Multiple Condition Coverage)**

Проверяются все возможные наборы значений компонент логических условий. Т.е. в случае n компонент потребуется 2n тестовых примеров, каждый из которых проверяет один набор значений, Тесты, необходимые для полного покрытия по данному методу, дают полную таблицу истинности для логического выражения.

Метод редко применяется на практике в связи с его сложностью и избыточностью.

Уровни покрытия:

- **Метод MC/DC для уменьшения количества тестовых примеров при 3-м уровне покрытия кода (**Для уменьшения количества тестовых примеров Modified Condition/*Decision Coverage*)

Для обеспечения полного покрытия по этому методу необходимо выполнение следующих условий:

- каждое логическое условие должно принимать все возможные значения;
- каждая компонента логического условия должна хотя бы один раз принимать все возможные значения;
- должно быть показано независимое влияние каждой из компонент на значение логического условия, т.е. влияние при фиксированных значениях остальных компонент.
- Покрытие по этой метрике требует достаточно большого количества тестов для того, чтобы проверить каждое условие, которое может повлиять на результат выражения, однако это количество значительно меньше, чем требуемое для метода покрытия по всем условиям.

Уровни покрытия:

```
if (A || B)
{
 if (C)
  {
   ...
  }
 else
  {
   ...
  }
}
```

Для тестирования первого условия по MC/DC надо показать независимость результата (т.е. функции A || B) от каждого аргумента. Соответственно, для этого используются три тестовых примера:

A = 0, B = 0, A || B = 0 (начальное значение)
A = 1, B = 0, A || B = 1 (показано влияние аргумента A)
A = 0, B = 1, A || B = 1 (показано влияние аргумента B)

Для тестирования ветвей (входящего в MC/DC) в зависимости от условия C необходимо, чтобы в тестовых примерах C принимало значение как true, так и false.

!!!

К анализу покрытия программного кода можно приступать только после полного покрытия требований.

Полное покрытие программного кода не гарантирует того, что тесты проверяют все требования к системе.

Покрытие кода:

- ***покрытие операторов:*** каждый оператор должен быть выполнен как минимум один раз.
- ***покрытие условий:*** каждое условие, имеющее TRUE и FALSE на выходе, выполнено как минимум один раз.
- ***покрытие путей:*** все ли возможные пути через заданную часть кода были выполнены и протестированы
- ***покрытие функций:*** каждая ли функция программы была выполнена
- ***покрытие вход/выход:*** все ли вызовы функций и возвраты из них были выполнены
- ***покрытие значений параметров:*** все ли типовые и граничные значения параметров были проверены.

Анализ покрытия кода

какое количество кода работает с автоматическими тестами, выражается в %. 100% покрытие означает, что каждая строка кода была вызвана в тестах хотя бы один раз.

Code coverage calculation statistic

PLESK_11_5_20 ▾ Find Test Cases

× PLESK_12_0_18

Run date	Project	OS family	Build	Status	Ccov %	Total TC	Passed TC	F/B/S TC	Log	Resul
2014-06-24 16:05:02	PLESK_12_0_18	unix	140624 (rev: 333351)	completed	20.57	1712	1548	57/107/0	Link	Link
2014-06-20 16:05:01	PLESK_12_0_18	unix	140620 (rev: 333303)	completed	20.51	1712	1581	124/7/0	Link	Link
2014-06-17 16:05:01	PLESK_12_0_18	unix	140617 (rev: 333191)	completed	20.64	1712	1569	133/10/0	Link	Link

Отчет о покрытии - список структурных элементов покрываемого кода (функций или методов), содержащий для каждого структурного элемента информацию:

- Название функции или метода
- Тип покрытия (по строкам, по ветвям, MC/DC или иной)
- Количество покрываемых элементов в функции или методе (строк, ветвей, логических условий)
- Степень покрытия функции или метода (в процентах или в абсолютном выражении)
- Список непокрытых элементов (в виде участков непокрытого программного кода с номерами строк)
- Заголовочную информацию и общий итог - общую степень покрытия всех функций, для которых собирается информация о покрытии.

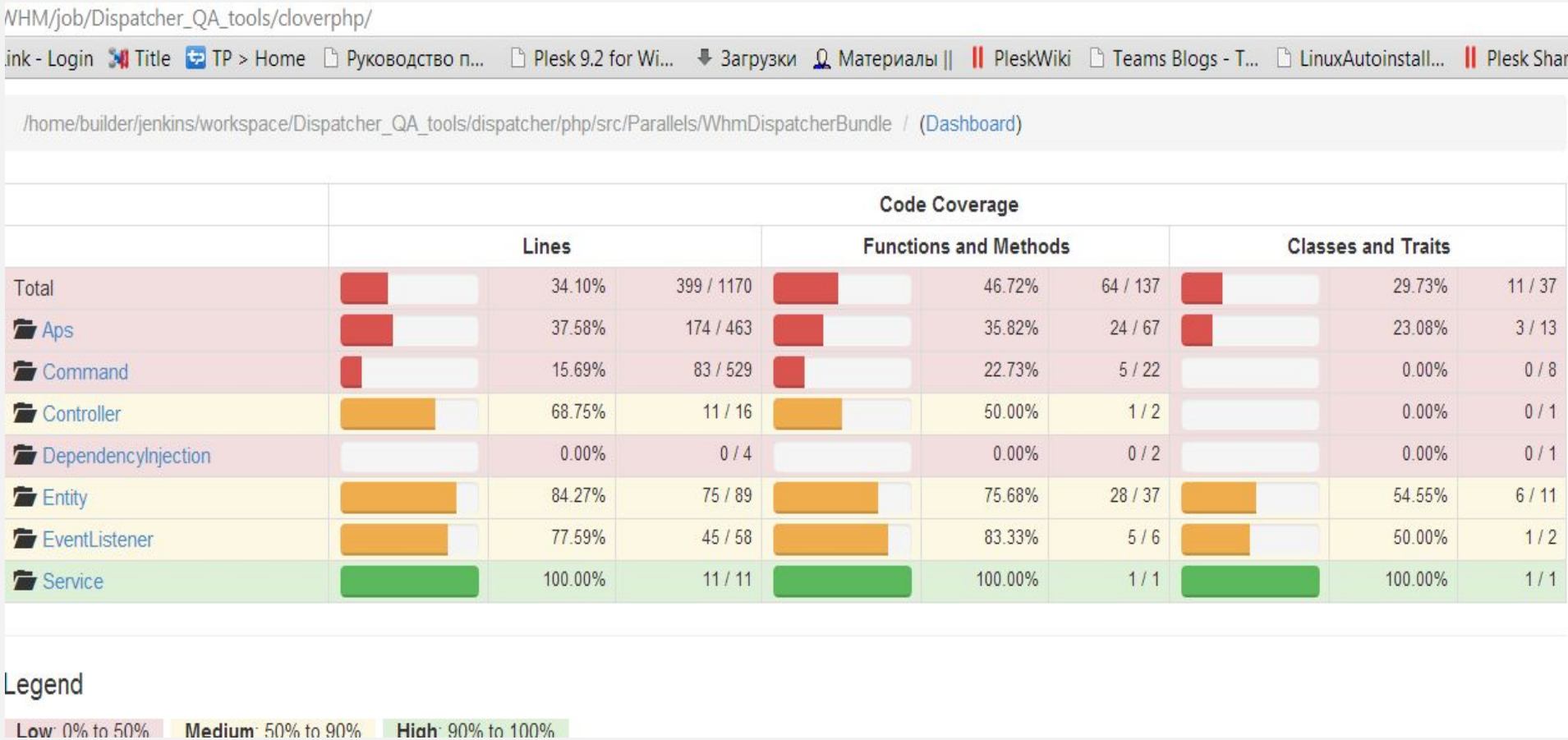
Code

Current file: /usr/share/aps_php/aps_php.php

Legend: executed | not executed | dead code

	Lines coveraged		
Total		21.24%	1377 / 648

```
1             : <?php
2             :
3             : /* ----------------------------------------------------------------------------
4             :  * This file was automatically generated by SWIG (http://www.swig.org).
5             :  * Version 2.0.1
6             :  *
7             :  * This file is not intended to be easily readable and contains a number of
8             :  * coding conventions designed to improve portability and efficiency. Do not make
9             :  * changes to this file unless you know what you are doing--modify the SWIG
10            :  * interface file instead.
11            :  * ----------------------------------------------------------------------------- */
12            :
13            : // Try to load our extension if it's not already loaded.
14          1 : if (!extension_loaded('aps_php')) {
15          0 :   if (strtolower(substr(PHP_OS, 0, 3)) === 'win') {
16          0 :     if (!dl('php_aps_php.dll')) return;
17            :   } else {
18            :     // PHP_SHLIB_SUFFIX gives 'dylib' on MacOS X but modules are 'so'.
19          0 :     if (PHP_SHLIB_SUFFIX === 'dylib') {
20            :       if (!dl('aps_php.so')) return;
21            :     } else {
22          0 :       if (!dl('aps_php.'.PHP_SHLIB_SUFFIX)) return;
23            :     }
```


WHM/job/Dispatcher_QA_tools/cloverphp/
ink - Login
Title
TP > Home
Руководство п...
Plesk 9.2 for Wi...
Загрузки
Материалы ||
PleskWiki
Teams Blogs - T...
LinuxAutoinstall...
Plesk Shar
/home/builder/jenkins/workspace/Dispatcher_QA_tools/dispatcher/php/src/Parallels/WhmDispatcherBundle / (Dashboard)
Code Coverage
Lines
Functions and Methods
Classes and Traits
Total 34.10% 399 / 1170 46.72% 64 / 137 29.73% 11 / 37
Aps 37.58% 174 / 463 35.82% 24 / 67 23.08% 3 / 13
Command 15.69% 83 / 529 22.73% 5 / 22 0.00% 0 / 8
Controller 68.75% 11 / 16 50.00% 1 / 2 0.00% 0 / 1
DependencyInjection 0.00% 0 / 4 0.00% 0 / 2 0.00% 0 / 1
Entity 84.27% 75 / 89 75.68% 28 / 37 54.55% 6 / 11
EventListener 77.59% 45 / 58 83.33% 5 / 6 50.00% 1 / 2
Service 100.00% 11 / 11 100.00% 1 / 1 100.00% 1 / 1
Legend
Low: 0% to 50%
Medium: 50% to 90%
High: 90% to 100%

Покрытие пользовательского интерфейса

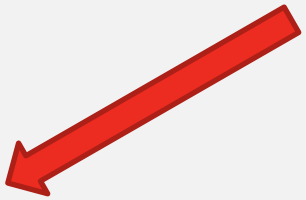
- ***функциональное покрытие*** - требований к UI
- ***структурное покрытие*** - каждый UI элемент должен быть использован в тестовых примерах хотя бы раз
- ***структурное покрытие с учетом состояния элементов интерфейса*** - необходимо не только использовать каждый элемент интерфейса, но и привести его во все возможные состояния (для чек-боксов - отмечен/не отмечен, для полей ввода - пустое/заполненное не целиком/заполненное полностью...)
- ***структурное покрытие с учетом состояния элементов интерфейса и внутреннего состояния системы*** - поведение некоторых UI элементов может изменяться в зависимости от внутреннего состояния системы. Каждое различимое поведение элемента должно быть проверено.

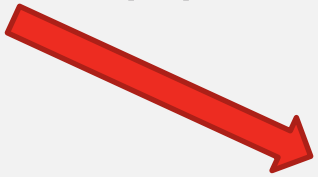
Функциональное тестирование пользовательских интерфейсов

5 фаз:

- анализ требований к пользовательскому интерфейсу;
- разработка тест-требований и тест-планов для проверки пользовательского интерфейса;
- выполнение тест-кейсов и сбор информации
- определение полноты покрытия пользовательского интерфейса требованиями;
- составление отчетов о проблемах в случае несовпадения поведения системы и требований либо в случае отсутствия требований на отдельные интерфейсные элементы.

Требования к пользовательскому интерфейсу





к внешнему виду пользовательского интерфейса и формам взаимодействия с пользователем

- Требования к размещению элементов управления на экранных формах
- Требования к содержанию и оформлению выводимых сообщений
- Требования к форматам ввода, в т.ч. определяющие реакцию системы на некорректный ввод.

требования по доступу к внутренней функциональности системы при помощи пользовательского интерфейса.

- Требования к реакции системы на ввод пользователя
- Требования к времени отклика на команды пользователя

Тестирование удобства использования пользовательских интерфейсов (usability)

Влияющие факторы :

- ***легкость обучения*** - быстро ли человек учится использовать систему;
- ***эффективность обучения*** - быстро ли человек работает после обучения;
- ***запоминаемость обучения*** - легко ли запоминается все, чему человек научился;
- ***ошибки*** - часто ли человек допускает ошибки в работе;
- ***общая удовлетворенность*** - является ли общее впечатление от работы с системой положительным.

этапы тестирования удобства использования пользовательского интерфейса:

- ***Исследовательское*** - после формулирования требований к системе и разработки прототипа интерфейса.
- ***Оценочное*** - после разработки низкоуровневых требований и детализированного прототипа пользовательского интерфейса.
- ***Валидационное*** - проводится ближе к этапу завершения разработки
- ***Сравнительное*** - может проводиться на любом этапе разработки интерфейса. В ходе сравнительного тестирования сравниваются два или более вариантов реализации пользовательского интерфейса.

10 характеристик удобного UI:

- *Наблюдаемость состояния системы (оповещение).*
- *Соотнесение с реальным миром (терминология)*
- *Пользовательское управление и свобода действий (дуракоустойчивость)*
- *Целостность и стандарты (терминология).*
- *Помощь пользователям в распознавании, диагностике и устранении ошибок.*
- *Предотвращение ошибок (дизайн + валидация)*
- *Распознавание, а не вспоминание.*
- *Гибкость и эффективность использования (hotkeys).*
- *Эстетичный и минимально необходимый дизайн.*
- *Помощь и документация.*

- Вопросы?

Стельмашенко Светлана
stsvisa@gmail.com

Что почитать по теме:

- Practitioner’s Guide to Software Test Design (Lee Copeland).
- Тестирование программного обеспечения (Канер, Фолк, Нгуен).
- Testing Web applications (Нгуен).
- Тестирование dot com (Савин).
- How we test software at Microsoft.
- Чудо-курс http://www.intuit.ru/studies/courses/1040/209/info