Software Testing

KRISTINE ELAINE P. BAUTISTA
CMSC 128 – INTRODUCTION TO SOFTWARE ENGINEERING
2ND SEMESTER, 2014-2015

What is testing?

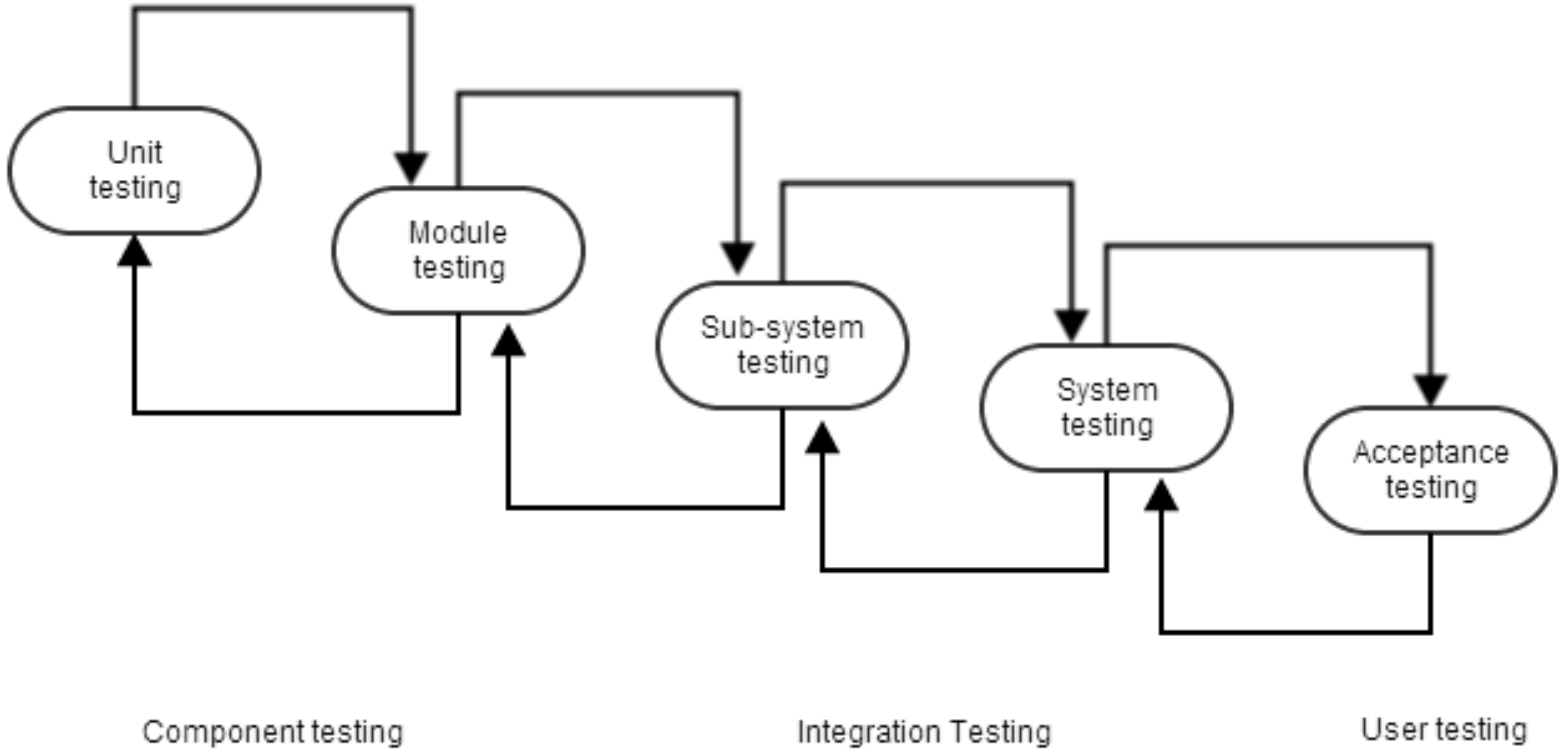
Testing

- intends to show that a program does what it is intended to do
- discovers program defects before it is put into use

Test Case

- a formal description of a *starting state*, one or more events to which the *software must respond*, and the expected response or *end state*
- are input and output specifications plus a statement of the function under test
- impossible to be generated automatically because it needs the output of the test to be predicted

The testing process



Stages of Testing

The Five Stages of Software Testing

1. Unit Testing
2. Module Testing
3. Sub-system Testing
4. System Testing
5. Acceptance Testing

Unit testing

- Individual components are tested to ensure that they operate correctly.
- Each component is tested independently, without other system components.

Module testing

- Module
  - collection of dependent components such as an object class, an abstract data type or some looser collection of procedures and functions
- A module can be tested without other system modules.

Sub-system testing

- involves testing collections of modules which have been integrated into sub-systems
- concentrates on the detection of interface errors by rigorously exercising these interfaces

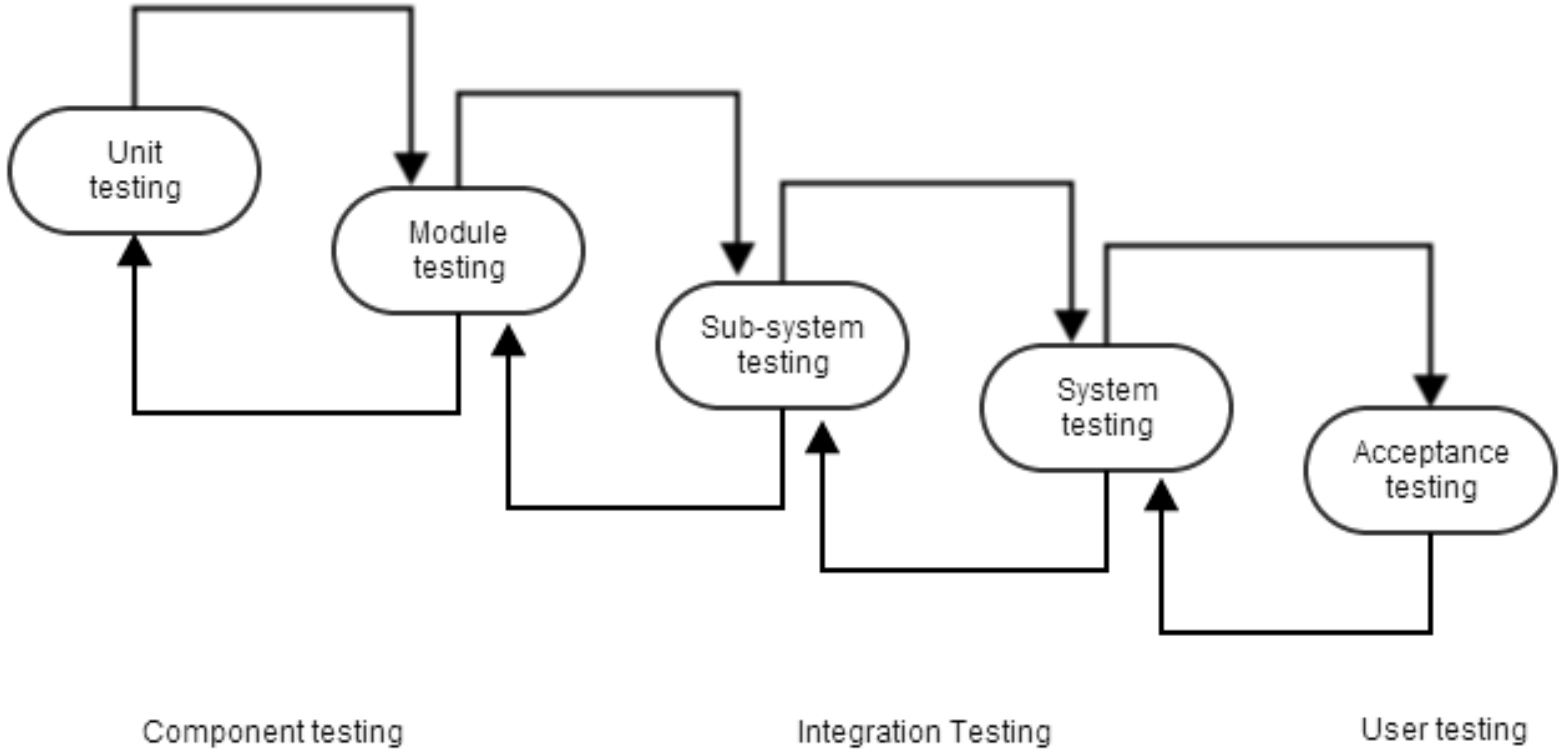
System testing

- concerned with finding errors which result from unanticipated interactions between sub-systems and system components
- also concerned with validating that the system meets its functional and non-functional requirements

Acceptance testing

- final stage of the testing process before the system is accepted for operational use
- may reveal errors and omissions in the system requirements definition because the real data exercises the system in different ways from the test data
- may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is unacceptable

The testing process



Types of Testing

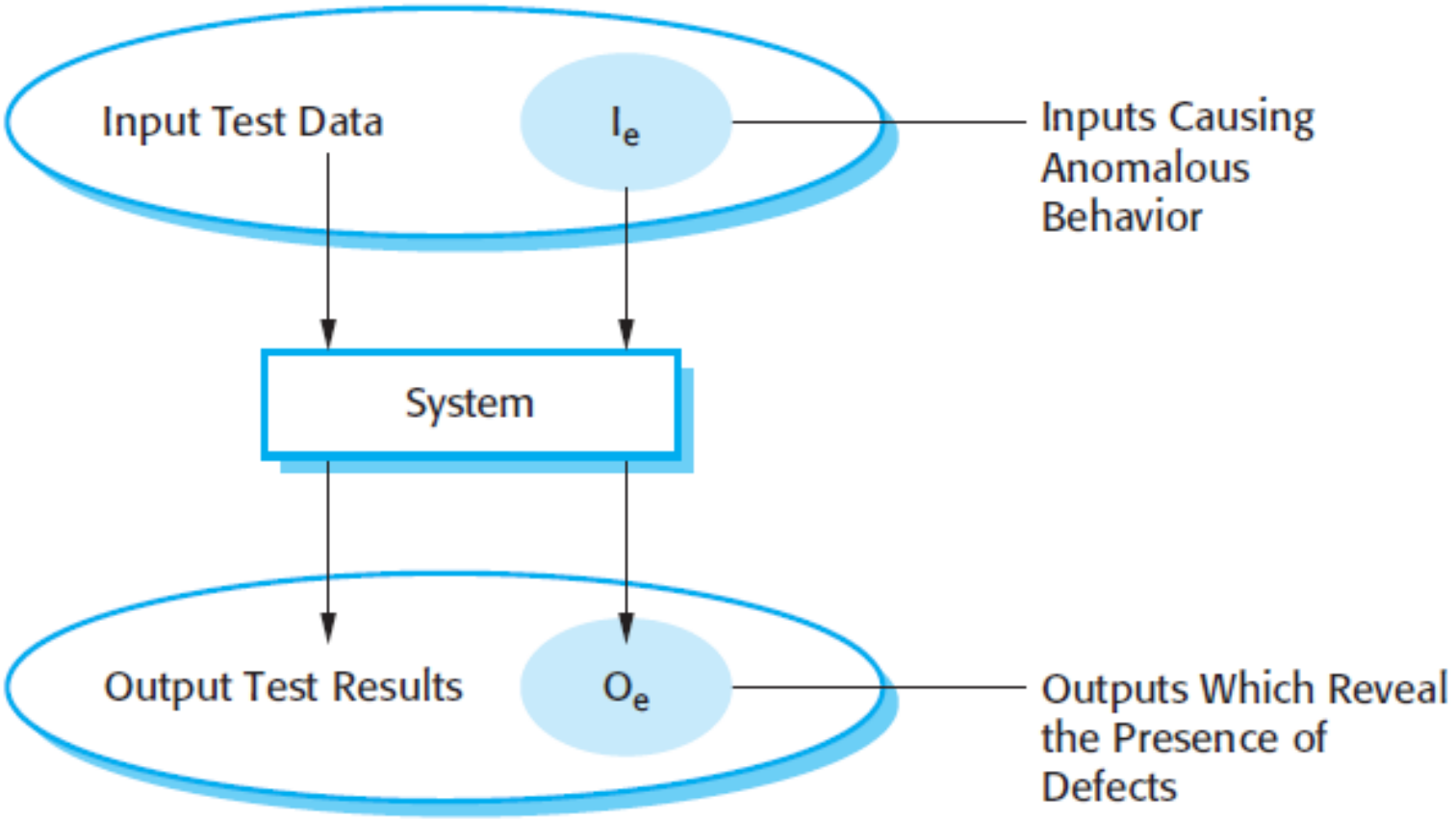
Black-box Testing

TYPES OF TESTING

Black-box Testing

- also called *functional* or *behavioral testing*
- focuses on functional requirements of the software
- tests are derived from the program specification
- enables to derive sets of input conditions that will fully exercise all functional requirements of a program

Black-box Testing



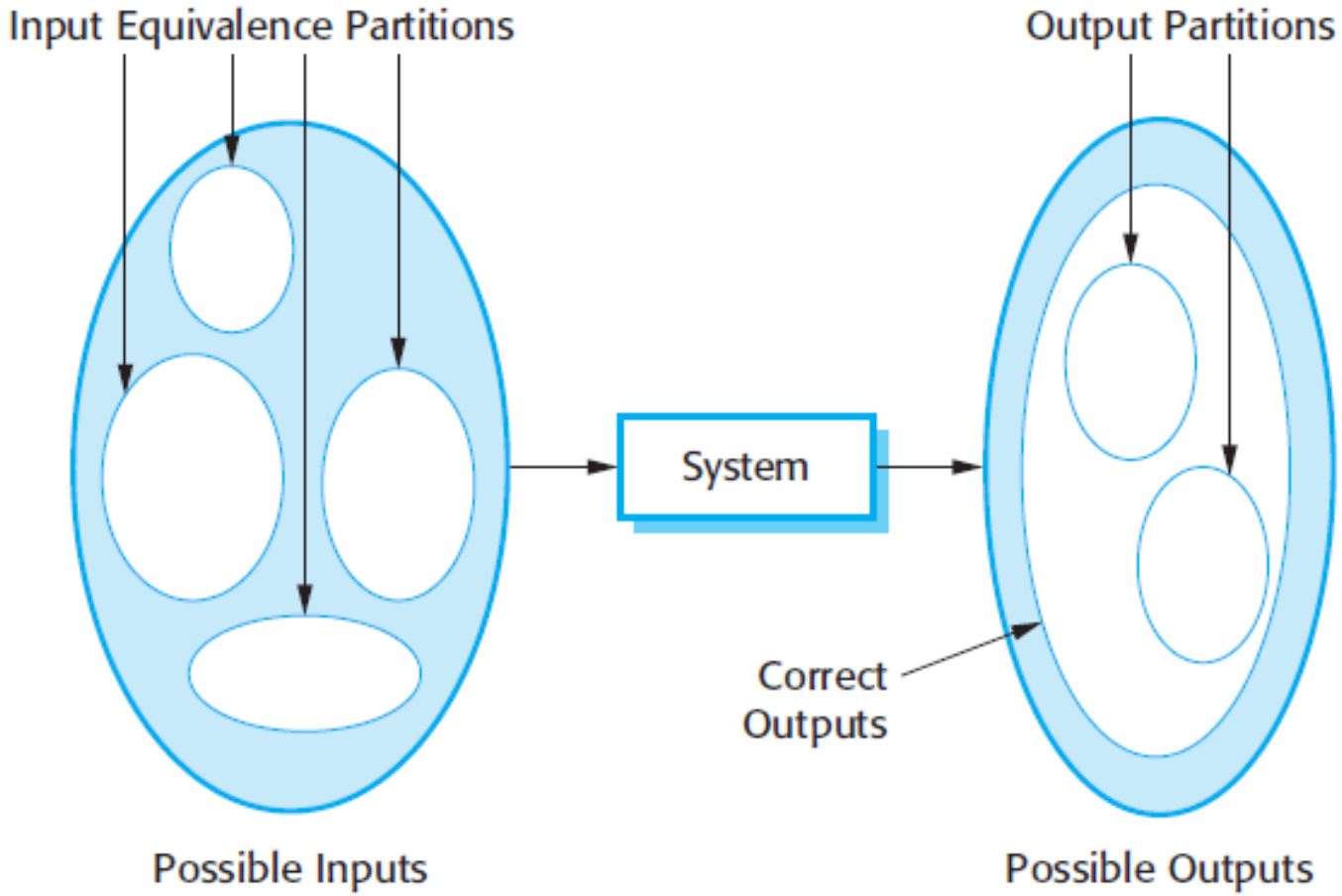
Equivalence Partitioning

BLACK-BOX TESTING

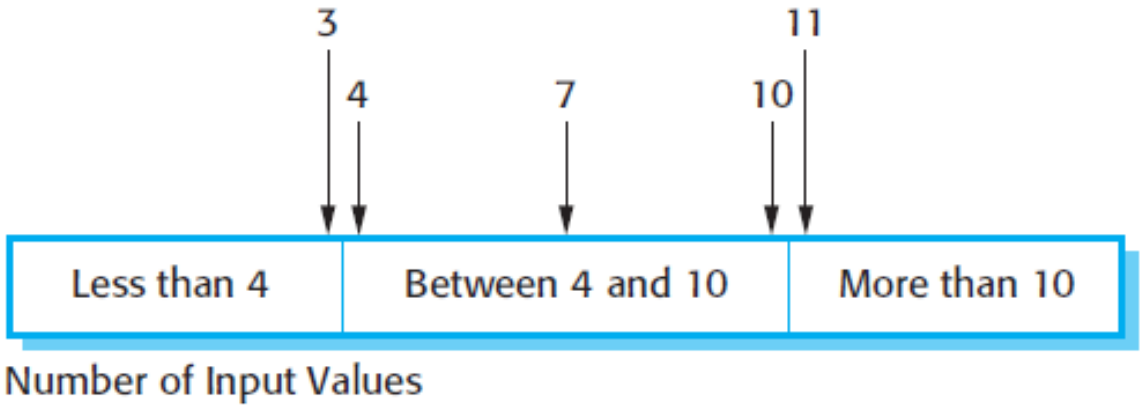
Equivalence Partitioning

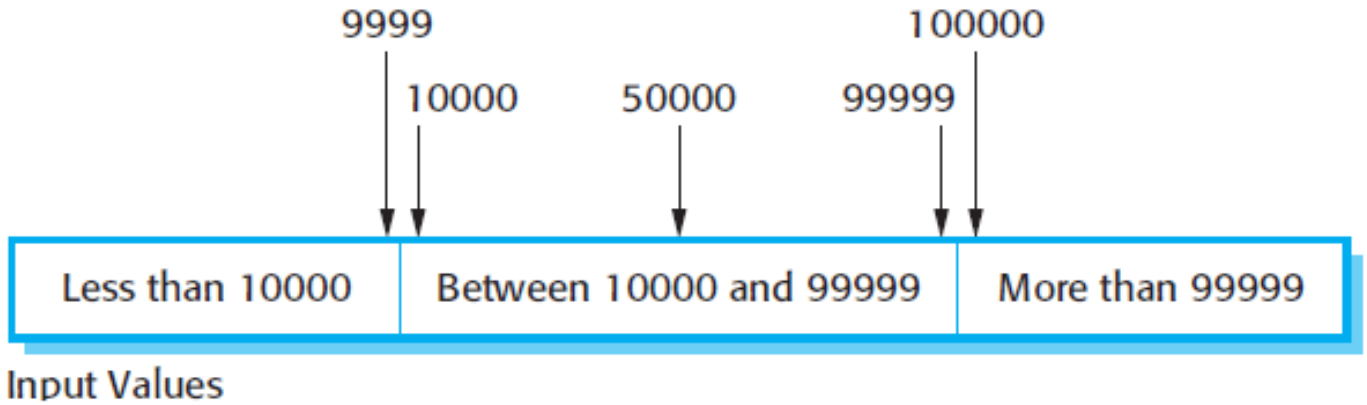
- a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived
- The input data to a program usually fall into a number of different classes. These classes have common characteristics.
  - e.g. positive numbers, negative numbers, strings without blanks, etc.

Equivalence Partitioning



Equivalence Partitions





Example of Equivalence Partition

Specification of a Search Function in C

```c
int search(int array[10],int x);

main(){
    int x, array[10];
    int location;

    /**Pre-condition**/
    /*The array has 0 or more (up to 10) elements*/

    //get number to be searched
    printf("Enter number to be searched: ");
    scanf("%d",&x);

    //call search function
    location = search(array,x);

    /**Post-condition**/
    /*
        The element is found and index of in the array location is returned.
        or
        The element is not found in the array and location is returned as -1.
    */
}
```

Example of Equivalence Partition

Two obvious equivalence partitions can be identified:

1. Inputs where the value (x) is a member of the array (location > -1).
2. Inputs where the value (x) is not a member of the array (location == -1).

Identified input partitions for search function

Array	Element
No value	Not in array
Single value	In array
Single value	Not in array
More than 1 value	First element in array
More than 1 value	Last element in array
More than 1 value	Middle element in array
More than 1 value	Not in array

Test cases for search function

Input array (array)	Value to be searched (x)	Expected output (Location)
17	17	0
17	0	-1 //value not found
17, 29, 21, 23	17	0
41, 18, 9, 31, 30, 16, 45	45	6
17, 18, 21, 23, 29, 41, 38	23	3
21, 23, 29, 33, 38	25	-1 //value not found

Two further equivalence partitions of the input array can be generated:

1. The input array has a single value.
2. The number of elements in the input array is greater than 1.

White-box Testing

TYPES OF TESTING

White-box Testing

- also called *structural* or *glass-box testing*
- tests are derived from knowledge of the program's structure and implementation
- uses control structure described as part of component-level design to derive test cases

White-box testing derives…

1. guarantee that all independent paths within a module have been exercised at least once
2. exercise all logical decisions on their true and false sides
3. execute all loops at their boundaries within their operational bounds
4. exercise internal data structures to ensure their validity

Path Testing

WHITE-BOX TESTING

Path testing

- a white-box testing strategy whose objective is to exercise every independent execution path through the component

Example of Path Testing

C++ implementation of a binary search function

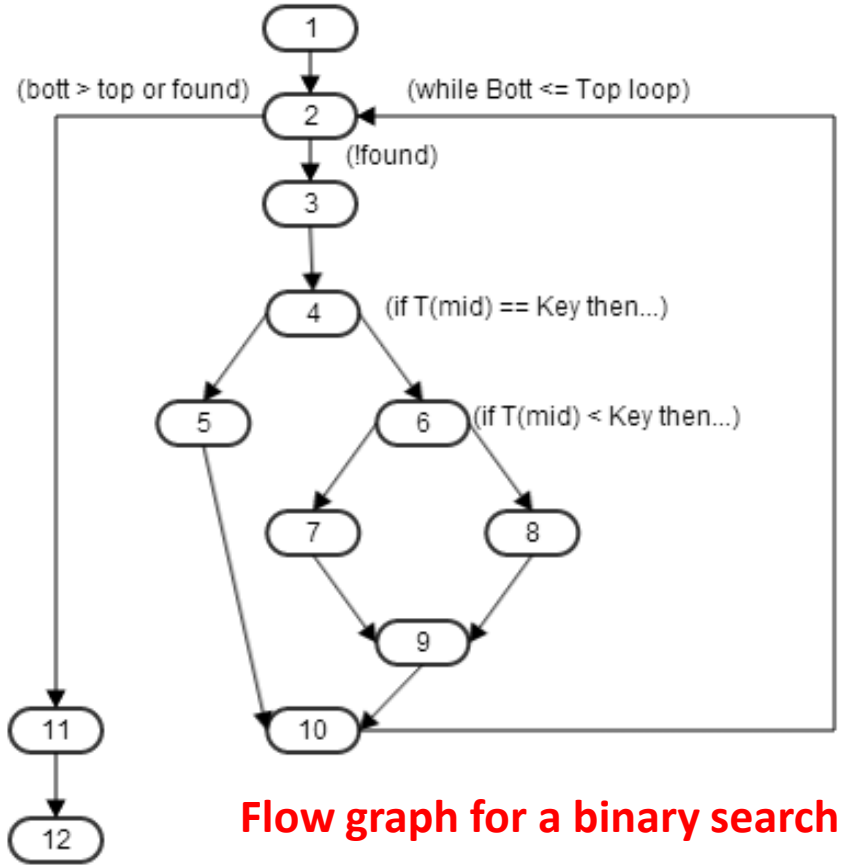
```
void Binary_search(elem key,elem* T, int size,
    boolean &found,int &L)
{
    int bott, top, mid;
    bott = 0;
    top = size - 1;
    found = false;

    while(bott <=top && !found)
    {
        mid = (top + bott)/2;
        if(T[mid] == key)
        {
            found = true;
            L = mid;
        }
        else if(T[mid] < key)
            bott = mid + 1;
        else top = mid - 1;
    }//while
}//binary search
```

Example of Path Testing cont. …

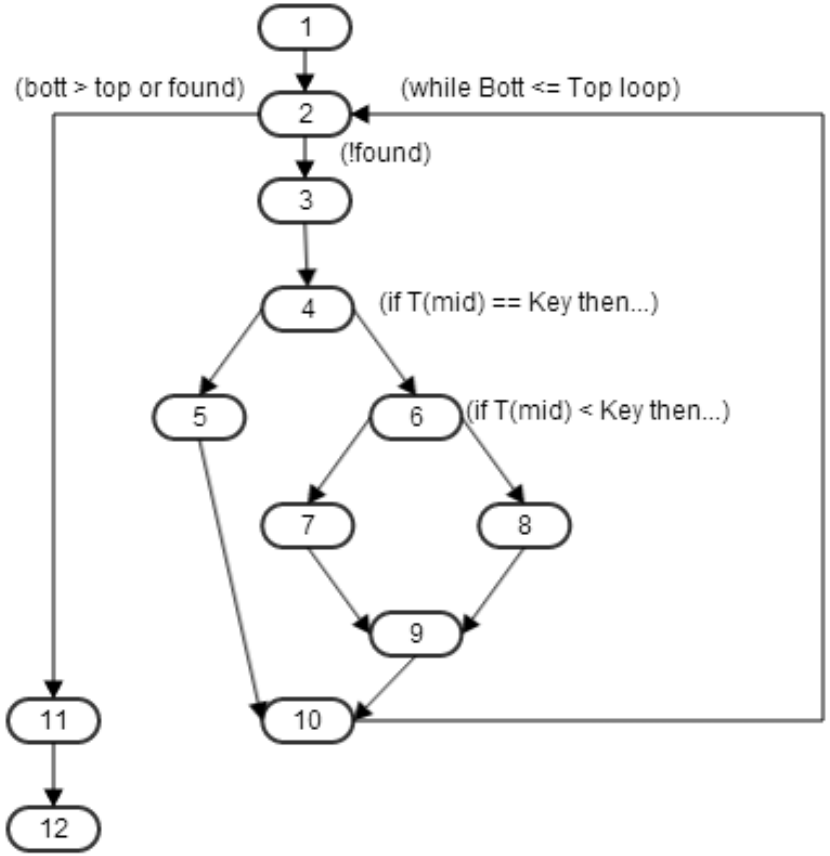
```
void Binary_search(elem key,elem* T, int size,
    boolean &found,int &L)
{
    int bott, top, mid;
    bott = 0;
    top = size - 1;
    found = false;

    while(bott <=top && !found)
    {
        mid = (top + bott)/2;
        if(T[mid] == key)
        {
            found = true;
            L = mid;
        }
        else if(T[mid] < key)
            bott = mid + 1;
        else top = mid - 1;
    }//while
}//binary search
```



Flow graph for a binary search function

Example of Path Testing cont. …



The independent paths are:

a. 1, 2, 11, 12
b. 1, 2, 3, 4, 5, 10, 2, 11, 12
c. 1, 2, 3, 4, 6, 7, 9, 10, 2, 11, 12
d. 1, 2, 3, 4, 6, 8, 9, 10, 2, 11, 12

Example of Path Testing cont. …

The independent paths are:

a. 1, 2, 11, 12
b. 1, 2, 3, 4, 5, 10, 2, 11, 12
c. 1, 2, 3, 4, 6, 7, 9, 10, 2, 11, 12
d. 1, 2, 3, 4, 6, 8, 9, 10, 2, 11, 12

If all of these paths are executed, we can be sure that:

- every statement in the function has been executed at lease once
- every branch has been exercised for true and false conditions

Example of Path Testing cont. …

The independent paths are:

a. 1, 2, 11, 12
b. 1, 2, 3, 4, 5, 10, 2, 11, 12
c. 1, 2, 3, 4, 6, 7, 9, 10, 2, 11, 12
d. 1, 2, 3, 4, 6, 8, 9, 10, 2, 11, 12

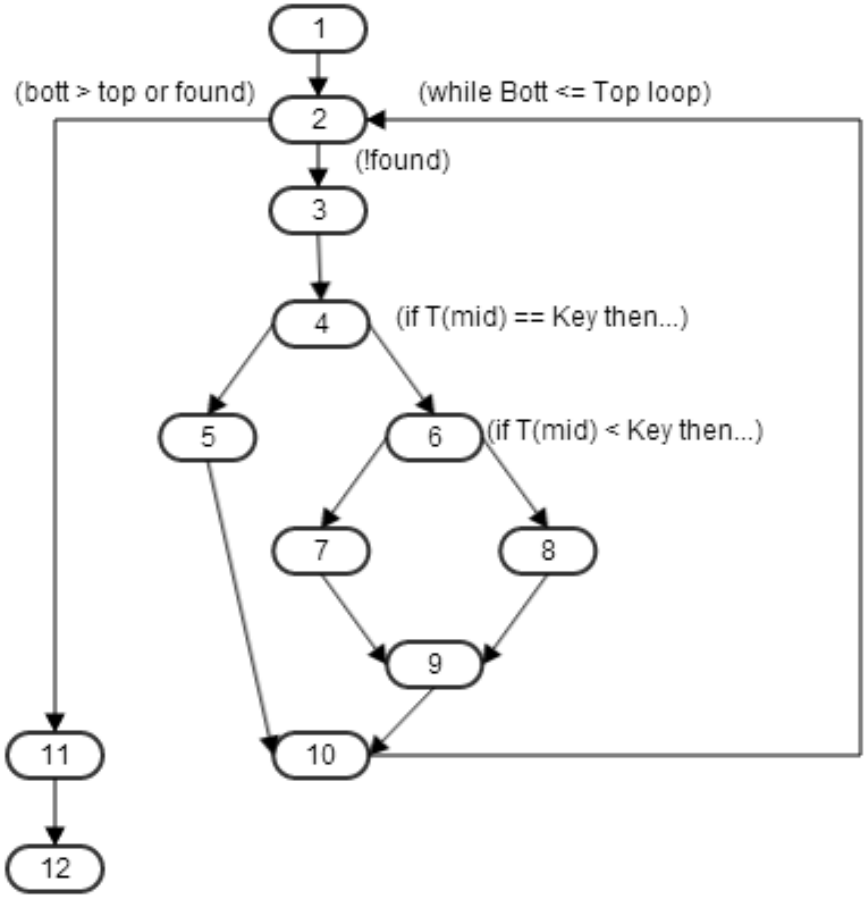
Prepare test cases that will force execution of each independent path.

- Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested.
- Each test case is executed and compared to expected results.

Test cases for binary search function

Input array: T  0

size =0



Independent Path	Value to be searched (key)	Expected Output (Found, L)
1, 2, 11, 12	15	false, *null*

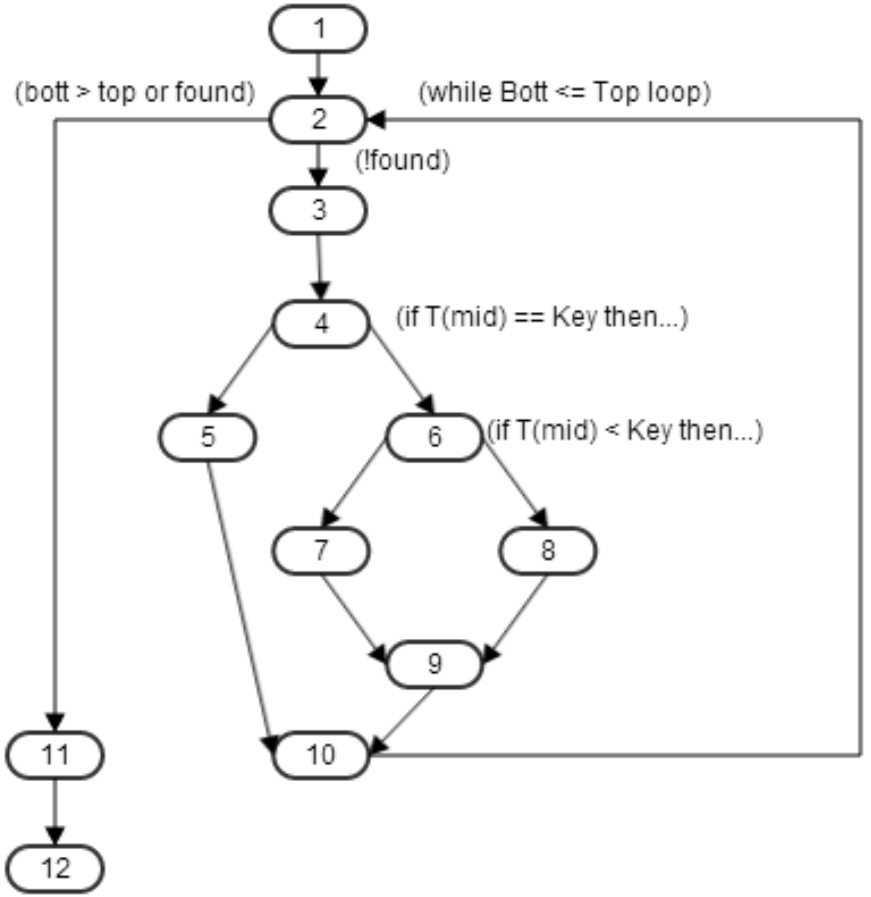
Binary Search Code

Test cases for binary search function

Input array: T

1	5	7	10	14	16	17	25	26	30
0	1	2	3	4	5	6	7	8	9

size =10



Independent Path	Value to be searched (key)	Expected Output (Found, L)
1, 2, 3, 4, 5, 10, 2, 11, 12	14	true, 4
1, 2, 3, 4, 6, 7, 9, 10, 2, 11, 12	25	true, 7
1, 2, 3, 4, 6, 8, 9, 10, 2, 11, 12	5	true, 1

Binary Search Code

Test Plan

Test Plan

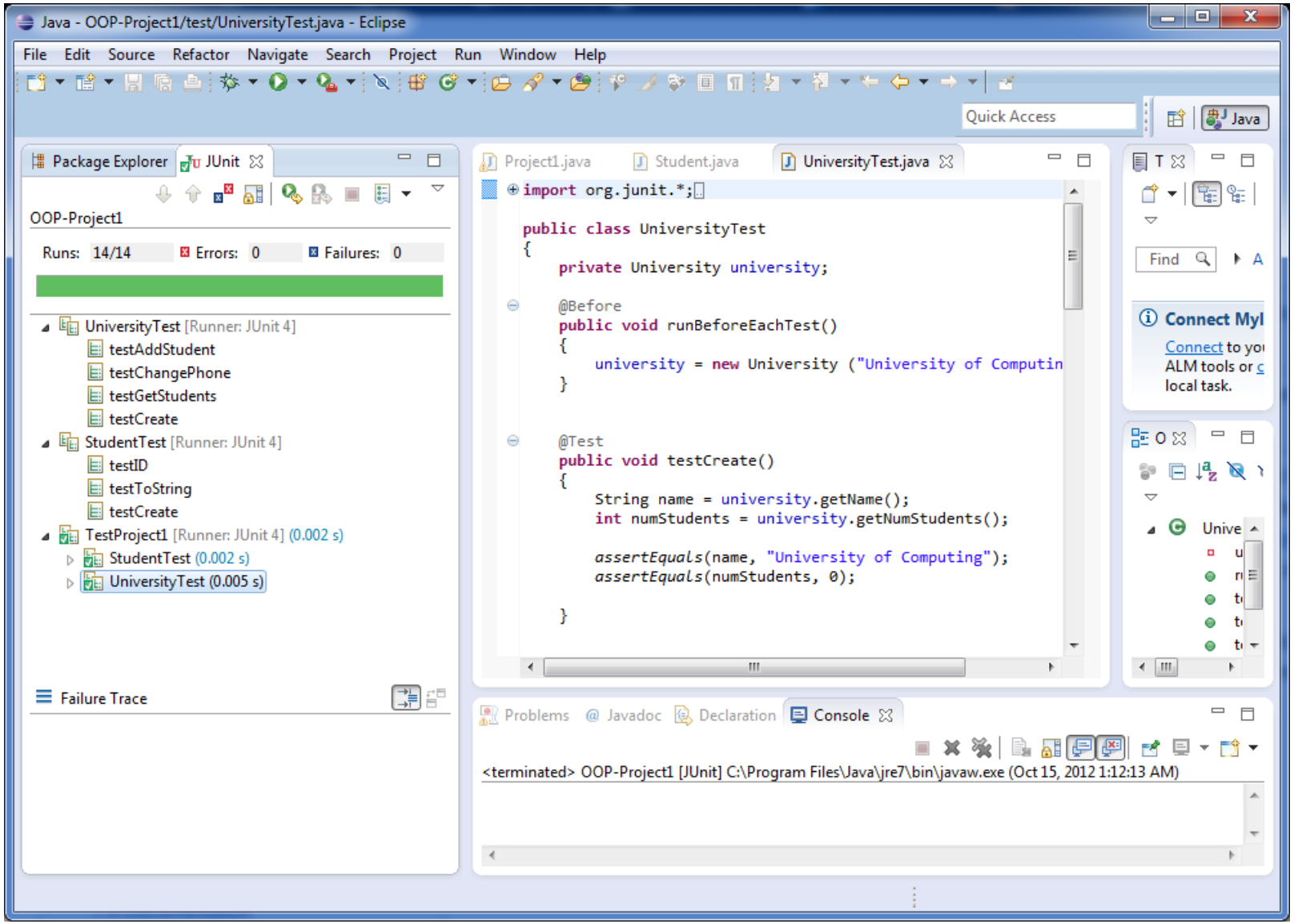
- a document describing the scope, approach, resources and schedule of intended test activities
- identifies amongst other test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice, and any risks requiring contingency planning

Some Unit Testing Frameworks

JUnit

Image from:

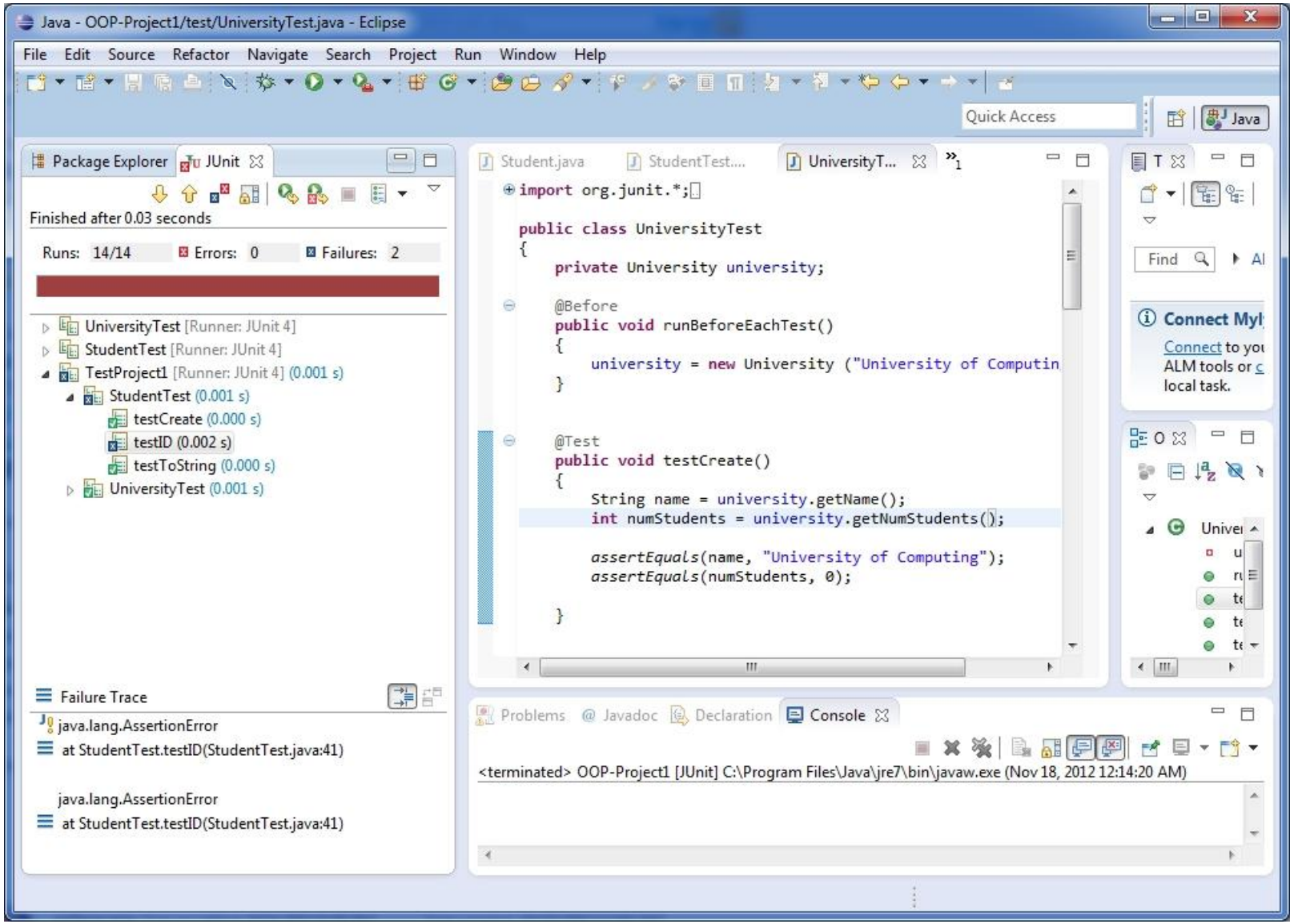
http://oopbook.com/junit-testing/junit-testing-in-eclipse/



JUnit

Image from:

http://oopbook.com/junit-testing/junit-testing-in-eclipse/



JUnit

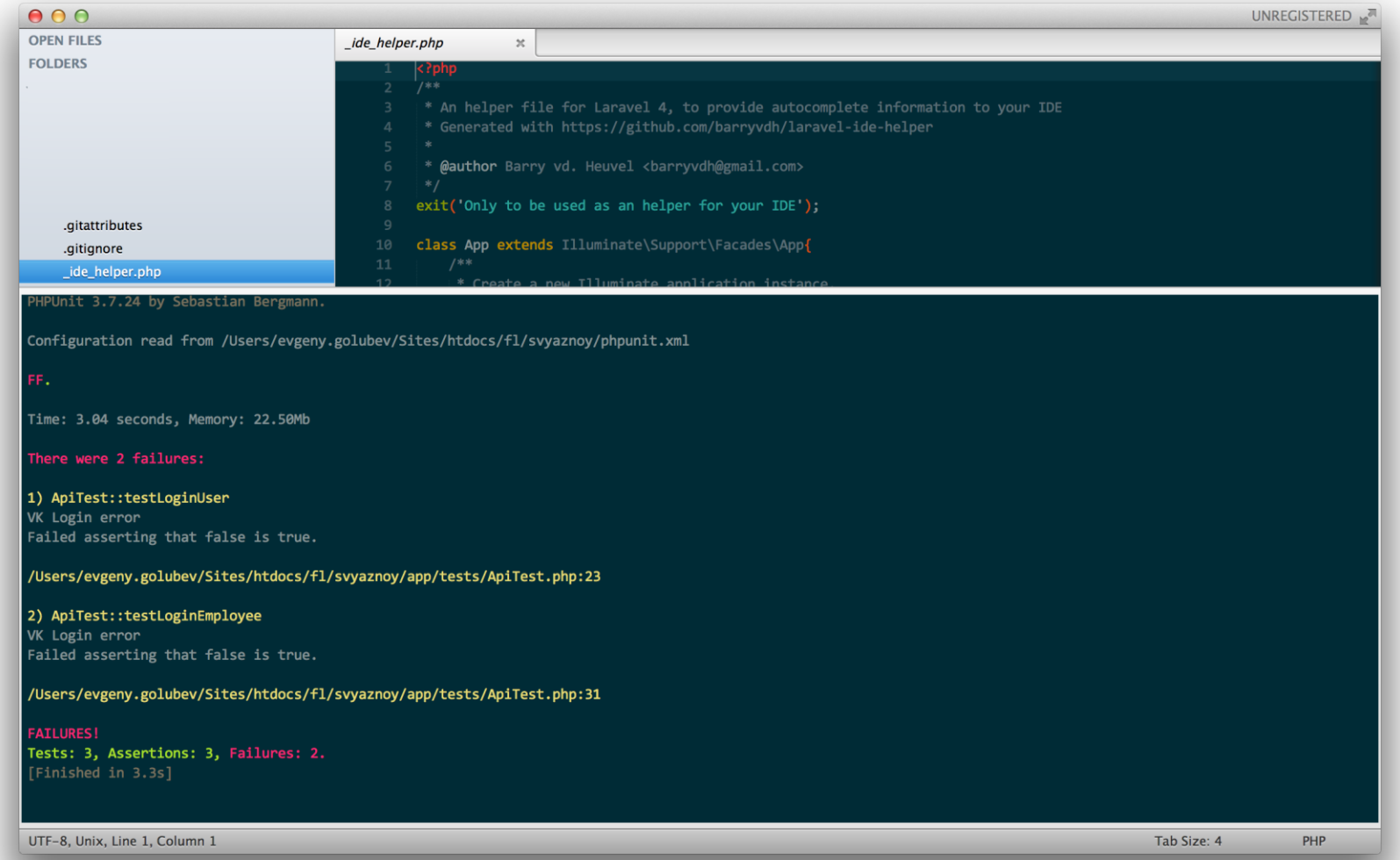
Tutorials and References

- http://www.tutorialspoint.com/junit/junit_test_framework.htm
- https://courses.cs.washington.edu/courses/cse143/11wi/eclipse-tutorial/junit.shtml
- http://junit.org/javadoc/latest/

PHPUnit

Image from:

https://github.com/evgeny-golubev/SimplePHPUnit-for-Sublime-Text



PHPUnit

Tutorials and References

- https://phpunit.de/getting-started.html
- http://www.sitepoint.com/tutorial-introduction-to-unit-testing-in-php-with-phpunit/
- http://devzone.zend.com/1115/an-introduction-to-the-art-of-unit-testing-in-php/

References

- Sommerville, I. (1995). *Software Engineering.* 5th Edition. Essex, England: Addison-Wesley
- Satzinger, J. W., Jackson, R. B., and Burd, S. D. (2007). *Systems Analysis and Design in a Changing World*. 4th Edition. Cengage Learning.
- Pressman, R. S. (2010). *Software Engineering a Practitioner's Approach (Alternate Edition)*. 7th Edition. New York, NY: McGraw-Hill.
- *Test Plan.* (2011). Retrieved February 23, 2015, from: http://softwaretestingfundamentals.com/test-plan/ .