

Software Testing



Presentation designed and written by human based on:

I. Sommerville - Software Engineering 8, Ch. 4 - Software Processes,
Ch. 23 - Software Testing
R. Pressman - Software Engineering 5th Ed. Ch.17 - Software Testing
Techniques, Ch.18 - Software Testing Strategies

Dr. Petru Florin Mihancea

V20260323

Two perspectives

Verification (do not confuse with formal verification)

Trying to ensure that the built software **correctly** implements a specific function

Defect testing

a **successful test** is one that **exposes a defect** that causes the system to perform incorrectly

Are we building the product right ?

Validation

Trying to ensure that the built software conforms to the **user real needs**

Validation/acceptance testing

a **successful test** shows that the system operates as intended

Are we building the right product ?

Jr. Petru Florin Mihancea

Testing “Limitation”

Exhaustive testing, where every possible program execution sequence is tested is **impossible**

Testing can only show the presence of errors, not their absence

E.Dijkstra

NO precise answer

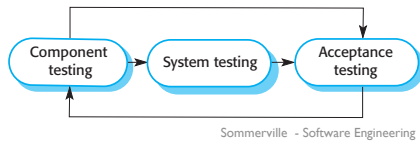
“you’re done testing when you run out of time/money”

“you’re never done testing, the burden simply shifts from you to your customer” i.e. every time the user executes a program, it is being tested

“statistical models can be used to predict total testing time required to achieve a particular low failure intensity”

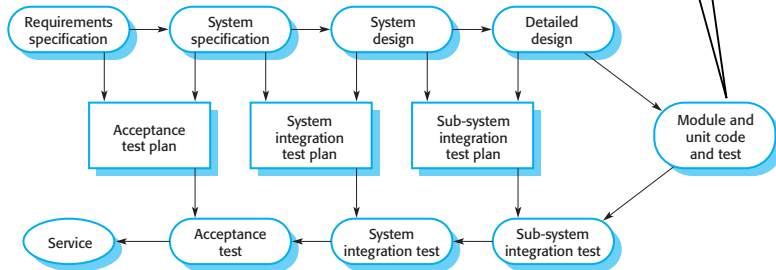
In general, we have to design tests that have the highest likelihood of finding the most errors

Testing Phases



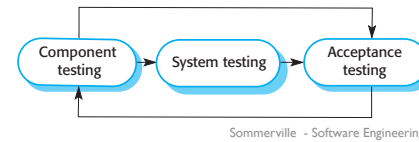
Component/Unit Testing

- components are functions, object classes, or small coherent groupings of these elements and are tested in isolation
- programmers make use of their own data and incrementally test their own code during development



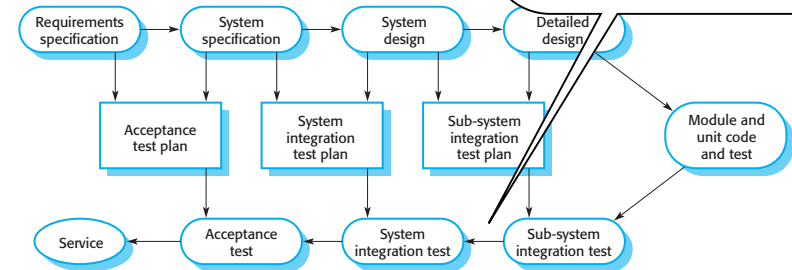
Jr. Petru Florin Mihances

Testing Phases



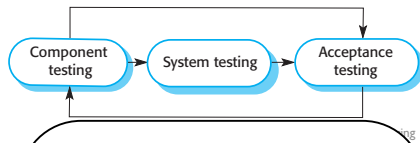
Integration Testing

- finding defects due to unanticipated interactions between components, interfacing problems, etc.
- verifying that system/sub-systems meet their functional requirements expected by the developers
- performed by an independent team (especially in the last integration steps)



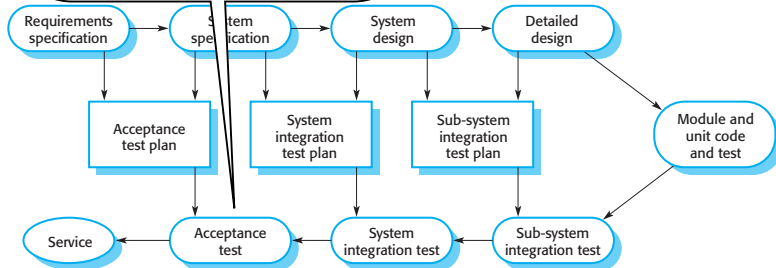
Jr. Petru Florin Mihances

Testing Phases



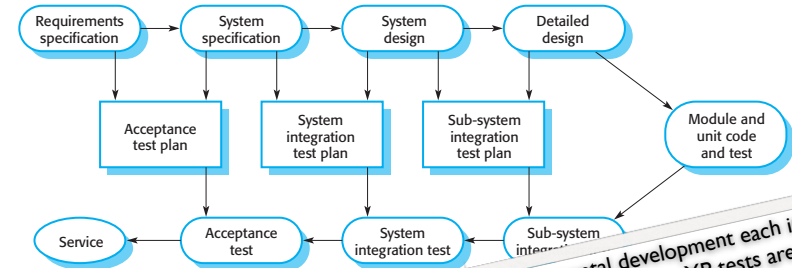
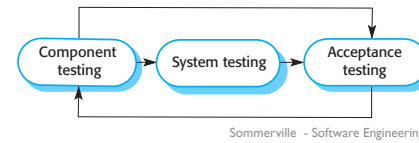
Validation/Acceptance Testing

- entire system testing with data provided by the customers



Jr. Petru Florin Mihances

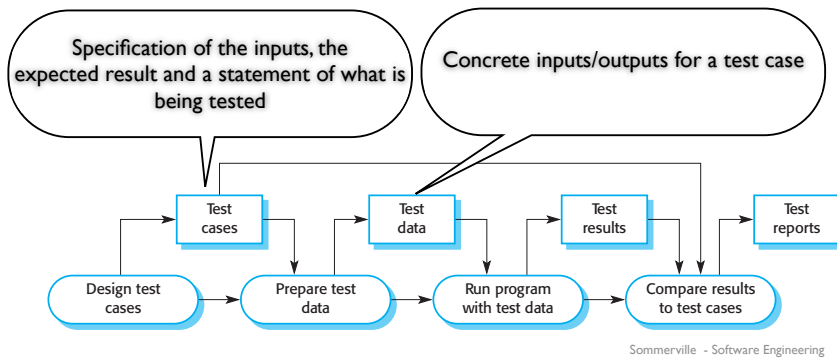
Testing Phases



In incremental development each increment is tested as developed; in XP tests are created ever before development starts

Jr. Petru Florin Mihances

Generic Model for Testing Process



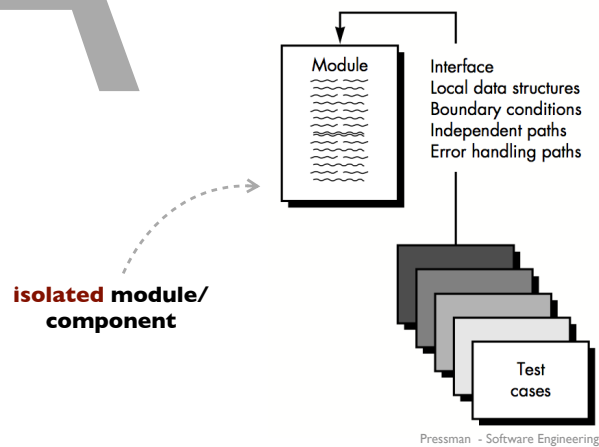
Dr. Petru Florin Mihances

1

Approaching Software Testing

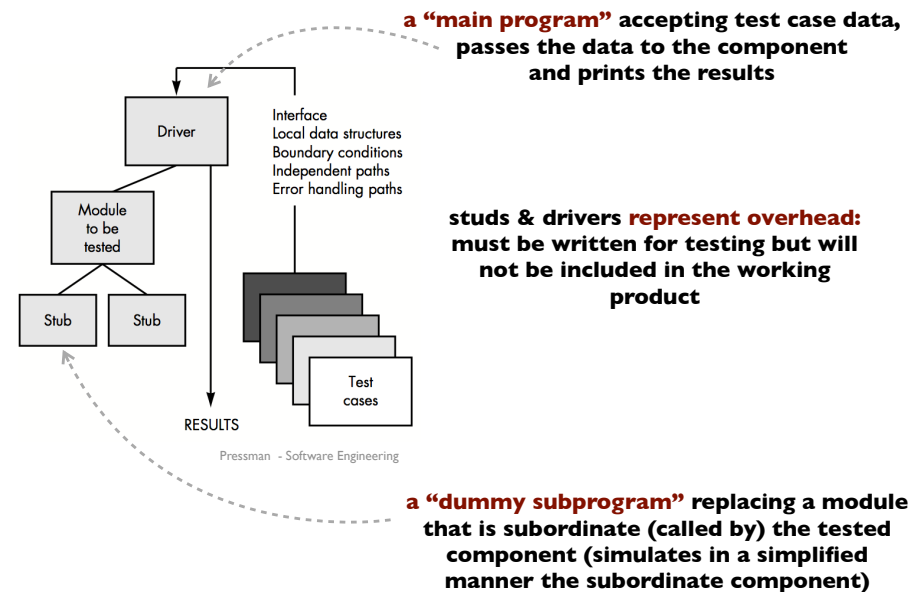
Dr. Petru Florin Mihances

Unit Testing



Dr. Petru Florin Mihances

Unit Testing Procedure



Dr. Petru Florin Mihances

3

Integration Testing

If all components **work individually**, why do we **doubt that they'll work together** ?

Interfacing problems

- data lost across an interface
- adverse effect by a module on another one
- accepted individual sub-functions may not provide the desired combined major function

Major approaches

- Big-Bang** integration
- Incremental** integration

Jr. Petru Florin Mihaices

“Big-Bang” Integration Testing

All components are integrated in advance and the **entire program** is tested as a whole

Major Disadvantage Chaos

very difficult to isolate causes due to the **vast expanse of the entire program**
when **some errors are solved**, **other errors** occur
and the process looks like an endless loop

Jr. Petru Florin Mihaices

Incremental Integration Testing

The program is **constructed and tested in small increments**

Advantages

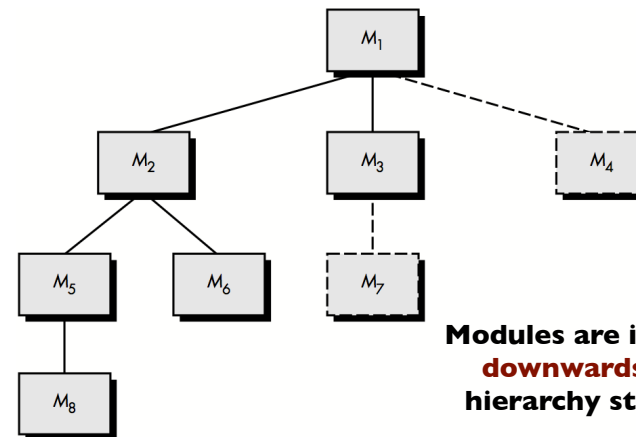
- errors are **easier to isolate**
- interfaces are **more likely** to be tested completely

Two integration approaches

- Top-Down**
- Bottom-Up**

Jr. Petru Florin Mihaices

Top-Down



Pressman - Software Engineering

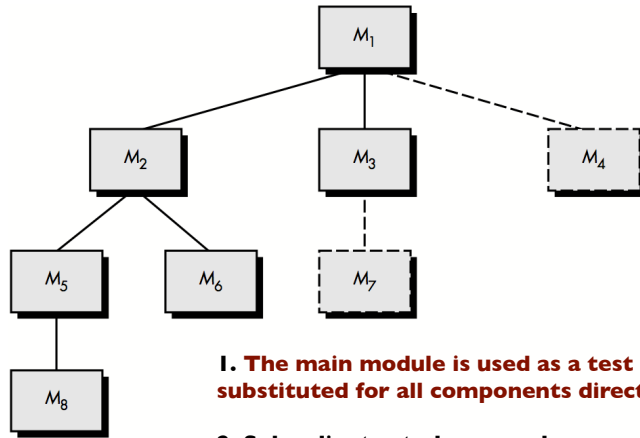
Modules are integrated by **moving downwards** through the control hierarchy starting from the main control module

Submodules are successively integrated in a **depth-first** or **breadth-first** manner

- Depth-First:** incrementally integrates all components on a major control path
- Breadth-First:** incrementally integrates all components directly subordinate

Jr. Petru Florin Mihaices

Top-Down



1. The main module is used as a test driver and stubs are substituted for all components directly subordinate
2. Subordinate stubs are replaced one at a time with the real component (in a depth/breadth first manner)

3. As each component is integrated, tests are conducted
4. On completion of each set of tests, another component is integrated
5. Regression testing may be applied to ensure that no new errors have been introduced

Pressman - Software Engineering

Dr. Petru Florin Mihailescu

Top-Down Pros & Cons

Pros

Checks **major control points** early in the testing process
In the depth-first way, a **complete function can be tested giving confidence**

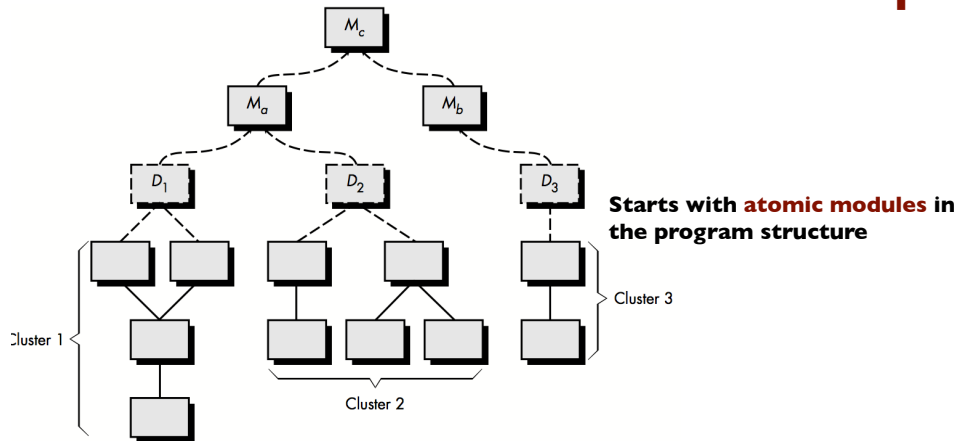
Cons

Stubs are needed for both not built/untested components
In practice, proper testing of high level modules may require complicated processing from the low level modules - **alternatives:**

- the tester may wait until she can replace the stub with the real module (tends to violate the idea of top-down integration)
- the tester may create a more complex stub (with significant overhead)

Dr. Petru Florin Mihailescu

Bottom-Up



Starts with **atomic modules** in the program structure

Pressman - Software Engineering

1. **Low-level components** are combined into **clusters** that perform a specific subfunction
2. A **driver** is written to coordinate test case inputs and outputs
3. The cluster is tested
4. **Drivers are removed and clusters are combined moving upwards** in the program structure

Dr. Petru Florin Mihailescu

Bottom-Up Pros & Cons

Pros

Checks **low-level data processing** early in the testing process
No need for stubs

Cons

Drivers are needed to test lower-level modules
More testing is required later when the upper-level modules are available because the drivers are incomplete

Hybrid approaches are possible: some high-level modules are integrated top-down while low-level modules are integrated bottom-up

Dr. Petru Florin Mihailescu

Smoke Testing

An integration testing approach

a pacing mechanism for time-critical projects
the team can assess the project on a regular basis



<http://www.grokdotcom.com>

1. Coded components are **integrated into a “build”**
2. Tests are designed to expose problems with **the highest likelihood of throwing the project behind the schedule (i.e., show-stopper errors)**
3. The build is integrated with other builds and **it is smoke tested daily**

Jr. Petru Florin Mihances

Benefits of Smoke Testing

Integration risk is **minimised**

incompatibilities and blocking errors may be uncovered earlier in the integration testing process

Error diagnosis and correction is **simplified**

the “new increment” added to the build is probably responsible for a new error

Progress is easier to assess

each day more code is integrated and more has been demonstrated to work

Jr. Petru Florin Mihances

Validation Testing

Are we building the right product ?

Acceptance testing - for custom software

Acceptance tests conducted by the **customer/end-user** to validate all requirements
Important to establish with the customer formal **validation criteria** during requirements engineering

Alpha & Beta testing - software for open market

Alpha testing
done in the **presence of developer** at her site

Beta testing

potential customers are selected to use the product in their environment and problems (real or imagined) are reported to the developers at regular intervals

Jr. Petru Florin Mihances

Regression Testing

Adding/changing a module **changes**
the software ...

These changes may **cause problems** with
functions that **previously worked fine** :(

Regression testing means the re-execution of some tests that have already been conducted to ensure that changes have not propagated unintended side effects

Especially in an integration context, re-executing all the tests can easily become impractical

A **regression test suite** should contain:

A sample of tests that will exercise all software functions
Tests focusing functions that are likely to be affected by the change
Tests that focus the changed component

Jr. Petru Florin Mihances

Intermediate I

Live Quiz



© Petru Florin Mihances

2

Test Case Design

Objective - design tests that have the highest likelihood of finding the most errors with a minimum amount of time and effort; thus we need a systematic approach

© Petru Florin Mihances

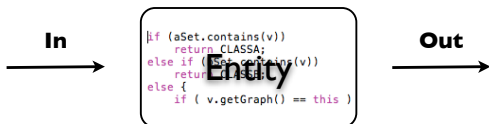
In practice, usually for validation and integration testing

Categories

Black-Box



White-Box (glass-box)



In practice, usually for unit-testing and some integration testing

© Petru Florin Mihances

A

(Basis) Path Testing White-Box Category

1. Identify a set of **linearly independent** paths

2. Write the corresponding **test case to exercise each** of these paths

ensures that each statement has been executed at least one time and each conditional statement is exercised for both true and false

Based on the notions of flow graphs, cyclomatic complexity and independent path

© Petru Florin Mihances

```
class BinSearch {
```

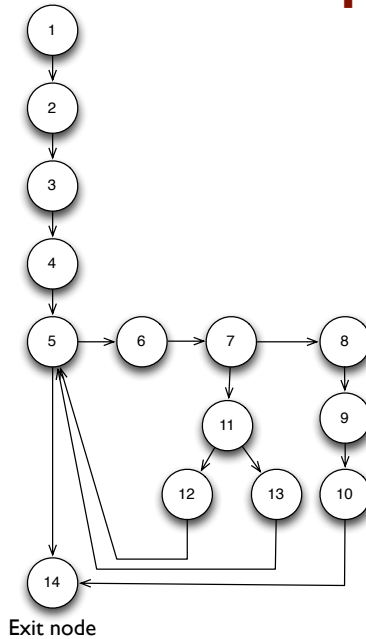
```
// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types
by
// reference to a function and so return two values
// the key is -1 if the element is not found
```

```
public static void search ( int key, int [] elemArray, Result r )
{
1.  int bottom = 0 ;
2.  int top = elemArray.length - 1 ;
3.  int mid ;
4.  r.found = false ;
5.  r.index = -1 ;
6.  while ( bottom <= top )
7.  {
8.      mid = (top + bottom) / 2 ;
9.      if (elemArray [mid] == key)
10.     {
11.         r.index = mid ;
12.         r.found = true ;
13.         return ;
14.     } // if part
15.     else
16.     {
17.         if (elemArray [mid] < key)
18.             bottom = mid + 1 ;
19.         else
20.             top = mid - 1 ;
21.     }
22. } //while loop
23. } // search
24. } //BinSearch
```

Sommerville - Software Engineering

Dr. Petru Florin Mihances

Flow Graph



```
class BinSearch {
```

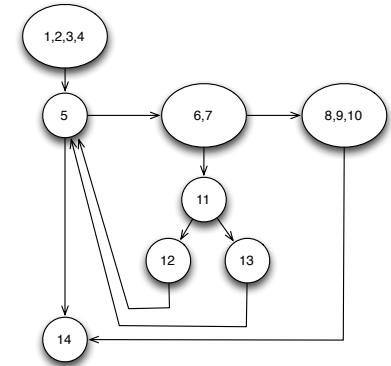
```
// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types
by
// reference to a function and so return two values
// the key is -1 if the element is not found
```

```
public static void search ( int key, int [] elemArray, Result r )
{
1.  int bottom = 0 ;
2.  int top = elemArray.length - 1 ;
3.  int mid ;
4.  r.found = false ;
5.  r.index = -1 ;
6.  while ( bottom <= top )
7.  {
8.      mid = (top + bottom) / 2 ;
9.      if (elemArray [mid] == key)
10.     {
11.         r.index = mid ;
12.         r.found = true ;
13.         return ;
14.     } // if part
15.     else
16.     {
17.         if (elemArray [mid] < key)
18.             bottom = mid + 1 ;
19.         else
20.             top = mid - 1 ;
21.     }
22. } //while loop
23. } // search
24. } //BinSearch
```

Sommerville - Software Engineering

Dr. Petru Florin Mihances

Flow Graph



```
class BinSearch {
```

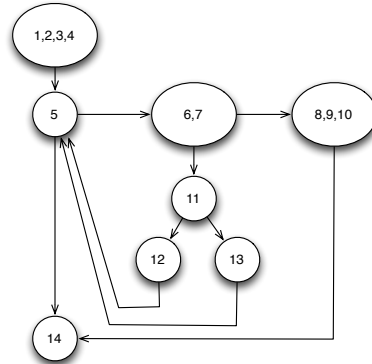
```
// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types
by
// reference to a function and so return two values
// the key is -1 if the element is not found
```

```
public static void search ( int key, int [] elemArray, Result r )
{
1.  int bottom = 0 ;
2.  int top = elemArray.length - 1 ;
3.  int mid ;
4.  r.found = false ;
5.  r.index = -1 ;
6.  while ( bottom <= top )
7.  {
8.      mid = (top + bottom) / 2 ;
9.      if (elemArray [mid] == key)
10.     {
11.         r.index = mid ;
12.         r.found = true ;
13.         return ;
14.     } // if part
15.     else
16.     {
17.         if (elemArray [mid] < key)
18.             bottom = mid + 1 ;
19.         else
20.             top = mid - 1 ;
21.     }
22. } //while loop
23. } // search
24. } //BinSearch
```

Sommerville - Software Engineering

Dr. Petru Florin Mihances

Cyclomatic Complexity



$$V(G) = E - N + 2$$

E - number of edges

N - number of nodes

```
class BinSearch {
```

```
// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types
by
// reference to a function and so return two values
// the key is -1 if the element is not found
```

```
public static void search ( int key, int [] elemArray, Result r )
{
1.  int bottom = 0 ;
2.  int top = elemArray.length - 1 ;
3.  int mid ;
4.  r.found = false ;
5.  r.index = -1 ;
6.  while ( bottom <= top )
7.  {
8.      mid = (top + bottom) / 2 ;
9.      if (elemArray [mid] == key)
10.     {
11.         r.index = mid ;
12.         r.found = true ;
13.         return ;
14.     } // if part
15.     else
16.     {
17.         if (elemArray [mid] < key)
18.             bottom = mid + 1 ;
19.         else
20.             top = mid - 1 ;
21.     }
22. } //while loop
23. } // search
24. } //BinSearch
```

Sommerville - Software Engineering

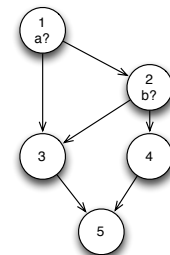
Dr. Petru Florin Mihances

Cyclomatic Complexity

Another way to compute the metric:
the number of **simple** branching conditions + 1

For **compound short-circuiting** conditions
(short-circuiting and, or, etc.) you must count
each simple condition

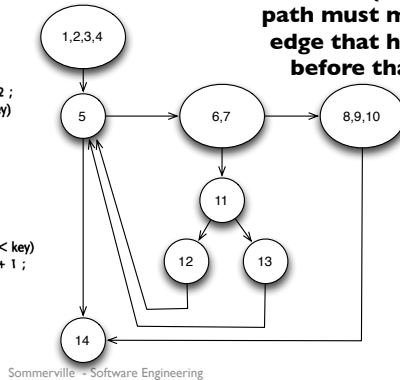
```
1,2:if(a || b) {
3: ...
} else {
4: ...
}
5:
```




```
class BinSearch {
```

```
// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types
// by
// reference to a function and so return two values
// the key is -1 if the element is not found
```

```
public static void search ( int key, int [] elemArray, Result r )
{
    1. int bottom = 0 ;
    2. int top = elemArray.length - 1 ;
    3. int mid ;
    4. r.found = false ;
    5. while ( bottom <= top )
    {
        6. mid = (top + bottom) / 2 ;
        7. if (elemArray [mid] == key)
        {
            8. r.index = mid ;
            9. r.found = true ;
            10. return ;
        } // if part
        else
        {
            11. if (elemArray [mid] < key)
            12. bottom = mid + 1 ;
            else
            13. top = mid - 1 ;
        }
    } //while loop
    14. } // search
} //BinSearch
```



Sommerville - Software Engineering

The Basis Set

Cyclomatic complexity indicates the number of independent paths in the basis set

An independent path is any path that introduces at least one new processing statement or a new condition (in graph, an independent path must move along at least one edge that has not been traversed before that path was defined)

- 1) 1,2,3,4,5,14
- 2) 1,2,3,4,5,6,7,8,9,10,14
- 3) 1,2,3,4,5,6,7,11,12,5,...
- 4) 1,2,3,4,5,6,7,11,13,5,...

Finally, we have to derive test cases for each of these paths

Dr. Petru Florin Mihances

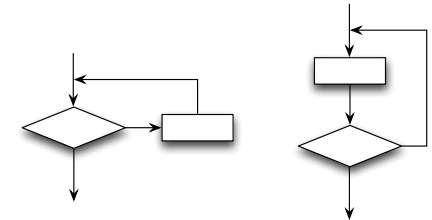


Simple loops

Write test cases to:

1. Skip the loop
2. One pass through the loop
3. Two passes through the loop
4. M passes where $M < N$
5. $N-1$, N , $N+1$ (forcing) passes

N - maximum number of allowable passes



Dr. Petru Florin Mihances

Loop Testing White-Box Category

Nested loops

1. Simple loop testing for innermost loop keeping the outer loops at their minimum iteration parameter

2. Move out one loop to apply first step for it; keep the inner loops at their typical value for their iterations

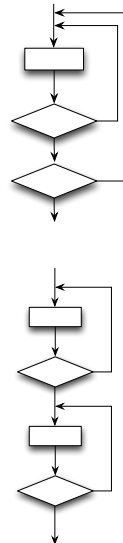
Repeat until the outermost loop has been tested

Concatenated loops

If independent, separately test each one as for simple loops

Otherwise, apply a similar approach as for nested loops

Loop Testing White-Box Category



Dr. Petru Florin Mihances

Equivalence Partitioning Black-Box Category

Divides the input domain of the tested entity into classes (partitions) of equivalent data

Rationale - an entity should behave in the same way for all members of a partition

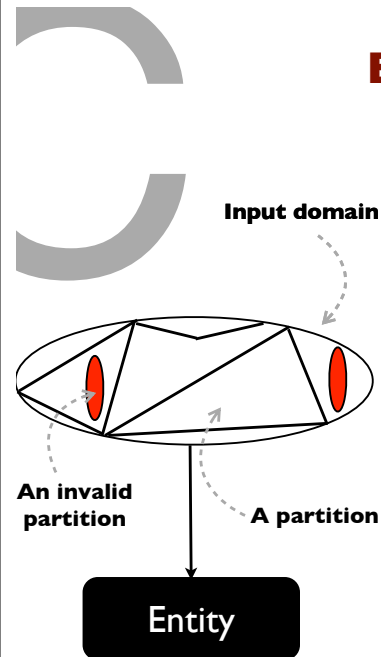
Division is based on evaluating input conditions (for each input data):

a. If an input condition specifies a range, we have one valid and two invalid partitions

b. If it requires a specific numerical value, we have one valid and two invalid partitions

c. If it requires a member from a set, we have one valid and one invalid classes

etc.



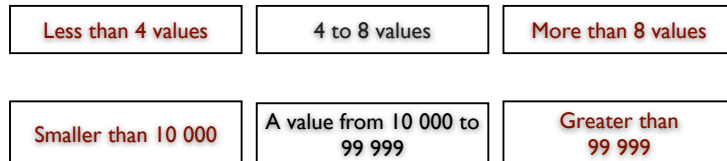
Dr. Petru Florin Mihances

Equivalence Partitioning

Black-Box Category

Example

The entity accepts 4 to 8 values that are five digit integers greater (or equal) to 10 000



Jr. Petru Florin Mihances

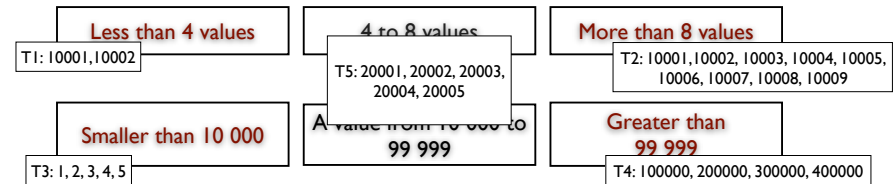
Equivalence Partitioning

Black-Box Category

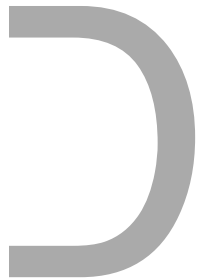
Producing test cases

- Write a **distinct** test case **for each invalid** partition
- Write as many test cases as necessary to cover **all** the **valid partitions** (try to cover as many valid partitions as possible in a test case)

Example



Jr. Petru Florin Mihances



Boundary Value Analysis

Black-Box Category

Errors tend to occur at the boundaries of the input domain thus, **write test cases that exercise bounding values**

Complements equivalence partitioning

- If an input condition specify a range bounded by **a** and **b**, test cases should be designed with values **a** and **b** and **just above** and **just below a** and **b**
 - If a number of values is specified, test cases should be developed that **exercise the minimum and maximum numbers**. **Just above** and **bellow** minimum and maximum must also be tested
- etc.

Jr. Petru Florin Mihances

Boundary Value Analysis

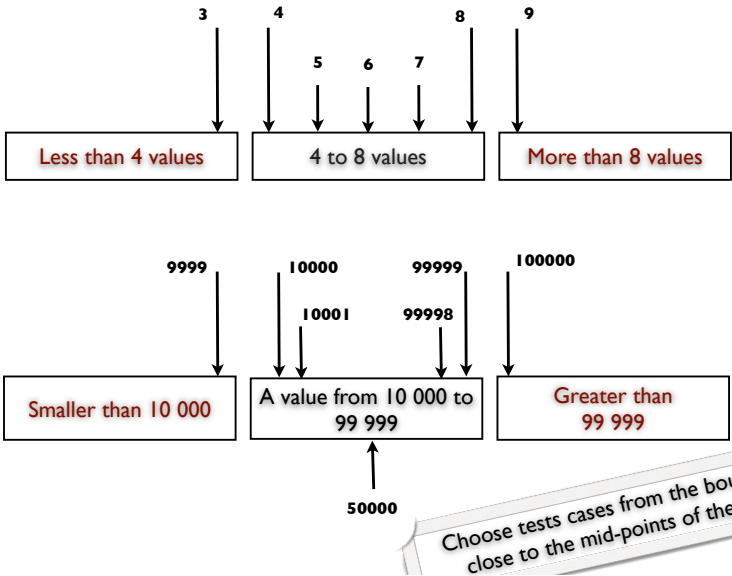
Black-Box Category

For sequences (collections)

- Test with sequences having **zero** and **one** element
- Ensure in distinct tests that the **first**, the **middle** and the **last** elements are accessed
- (More general guideline) Use **different** sequences with **different lengths** in different tests

Jr. Petru Florin Mihances

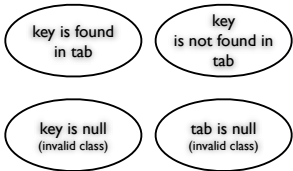
Example I : Partitioning + Boundary Value Analysis



Example (II)

boolean search(Integer key, int [] tab)

Returns true when the key is found in tab;
otherwise it always returns false



+

Sequence testing
guidelines



key	tab	result
null	[3]	FALSE
5	null	FALSE
77	[]	FALSE
7	[7]	TRUE
0	[7]	FALSE
17	[17,30,23,2]	TRUE
35	[40,198,9,19,38,6,35]	TRUE
23	[17,18,21,23,29,41,38]	TRUE
285	[12,253,89,13,30]	FALSE

End of Chapter VI

Live Quiz

