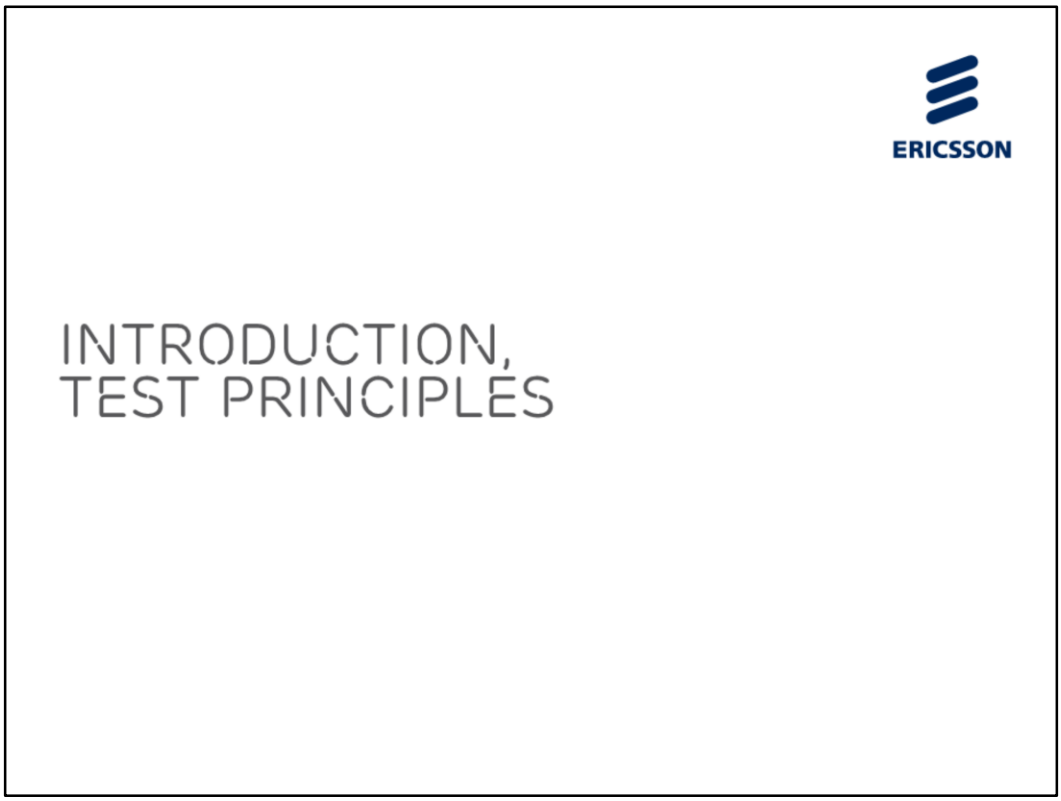
# AGENDA

- Introduction, test principles
- Black-box testing
- White-box testing
- Control flow testing
- Path testing
- Coverage testing
- Data flow testing
- Code review
- Unit testing
- Instrumentation
- Take aways



# GOALS OF TESTING

- To show the *presence* of bugs (Dijkstra, 1987)
- If tests do detect failures, we **cannot conclude** that software is **defect-free**
- Still, we need to do testing
  - driven by sound and systematic principles
- Should help **isolate errors**
  - to facilitate debugging
- Should be **repeatable**
  - repeating the same experiment, we should get the same results
    - this may not be true because of the effect of execution environment on testing
    - because of non-determinism



# SOFTWARE TESTING PRINCIPLES

Davids suggests a set of testing principles:

- All tests should be **traceable** to customer requirements.
- Tests should be **planned long before testing** begins.
- The **Pareto principle** applies to software testing.
  - 80% of all errors uncovered during testing will likely be traceable to 20% of all program modules.
- Testing should **begin** “in the **small**” and progress toward testing “in the **large**”.
- **Exhaustive** testing is **not possible**.

# CHARACTERISTICS OF TESTABLE SOFTWARE

- **Operable**
  - The better it works (i.e., better quality), the easier it is to test
- **Observable**
  - Incorrect output is easily identified; internal errors are automatically detected
- **Controllable**
  - The states and variables of the software can be controlled directly by the tester
- **Decomposable**
  - The software is built from independent modules that can be tested independently
- **Simple**
  - The program should exhibit functional, structural, and code simplicity
- **Stable**
  - Changes to the software during testing are infrequent and do not invalidate existing tests
- **Understandable**
  - The architectural design is well understood; documentation is available and organized

# TEST CHARACTERISTICS

- A good test has a high probability of finding an error
  - The tester **must understand the software** and how it might fail
- A good test is not redundant
  - Testing **time is limited**; one test should not serve the same purpose as another test
- A good test should be "best of breed"
  - Tests that have the highest likelihood of uncovering a whole class of errors should be used
- A good test should be neither too simple nor too complex
  - Each test should be **executed separately**; combining a series of tests could cause side effects and mask certain errors



# TEST HYPOTHESES

- **Regularity:** the number of **control states** in the implementation is **limited**. This allows to **limit testing to a finite set** of behaviours for systems exhibiting an infinite behavioural variety.
- **Uniformity:** it is sufficient to **test only one** out of several equivalent behaviours.
- **Independency:** Faults in one of the modules of the tested system **do not affect** the other modules, i.e., testing of the system is equivalent to testing of its modules separately.
- **Fairness:** In a nondeterministic system, the different execution paths tested cover all paths **relevant for detecting possible faults**.

# ISTQB GLOSSARY

**Black-box testing**

› **Synonyms:** specification-based testing

› Testing, either functional or non-functional, without reference to the internal structure of the component or system.

**White-box testing**

› **Synonyms:** clear-box testing, code-based testing, glass-box testing, logic-coverage testing, logic-driven testing, structural testing, structure-based testing

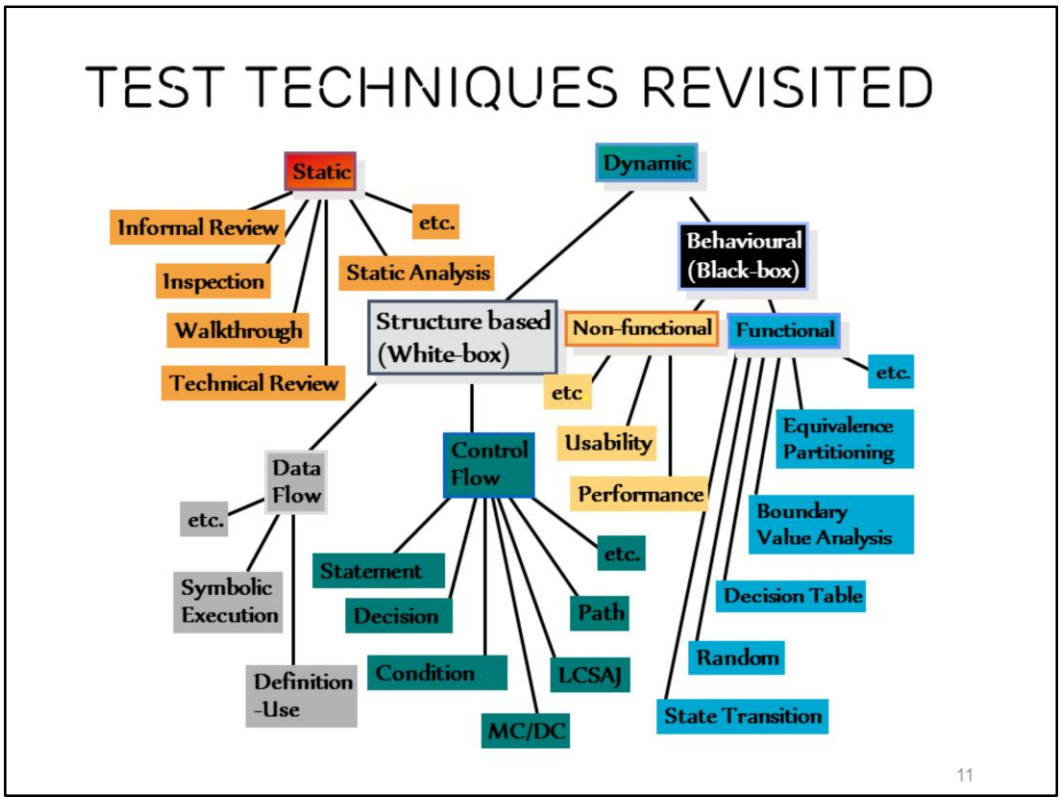
› Testing based on an analysis of the internal structure of the component or system.

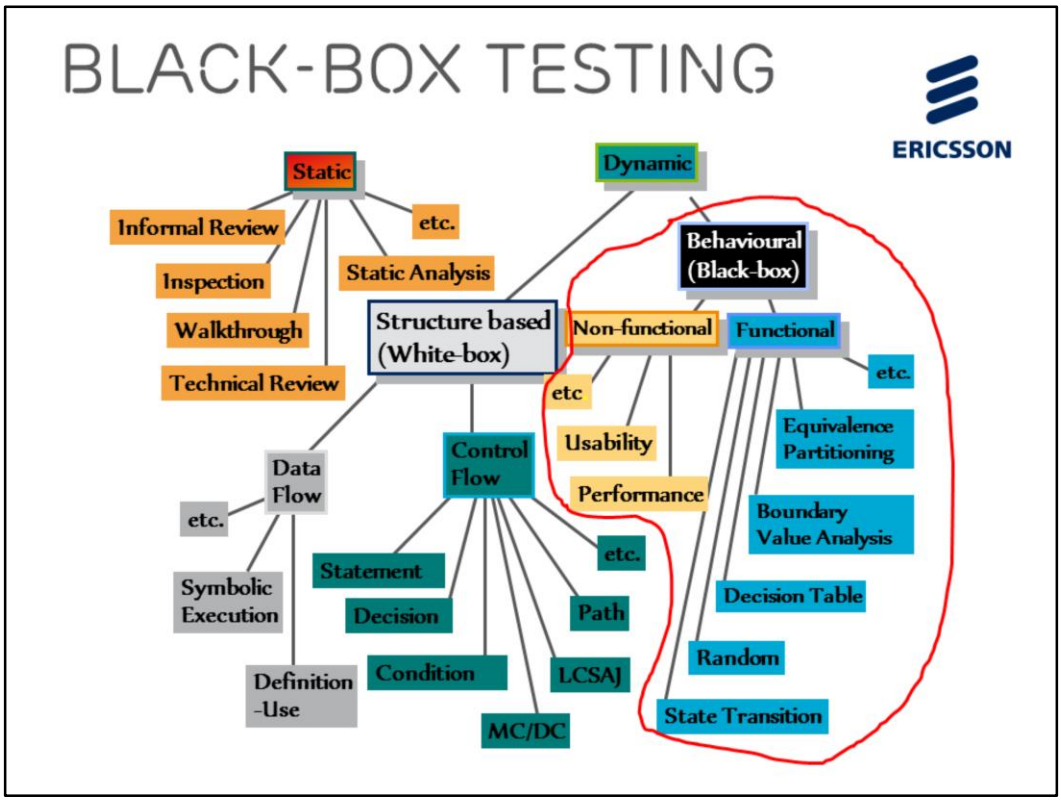
# MAJOR FOCUS

- **Specification based (Black-box) testing**: *functions, operations, external interfaces, external data and information*
- **Structure based (White-box) testing**: *internal structures, logic paths, control flows, data flows internal data structures, conditions, loops, etc.*




TEST TECHNIQUES REVISITED
Static
Informal Review
Inspection
Walkthrough
Technical Review
etc.
Static Analysis
Dynamic
Behavioural (Black-box)
Structure based (White-box)
Non-functional
Functional
etc
Usability
Performance
etc.
Equivalence Partitioning
Boundary Value Analysis
Decision Table
Random
State Transition
Data Flow
etc.
Symbolic Execution
Definition -Use
Control Flow
Statement
Decision
Condition
MC/DC
Path
LCSAJ
etc.
11



# BLACK BOX TESTING



- Black box testing
  - Implementation/System/Software Under Test
  - Point of Control and Observation
- We can
  - send **input** (action),
  - receive **output** (observation) and
  - measure the **time**

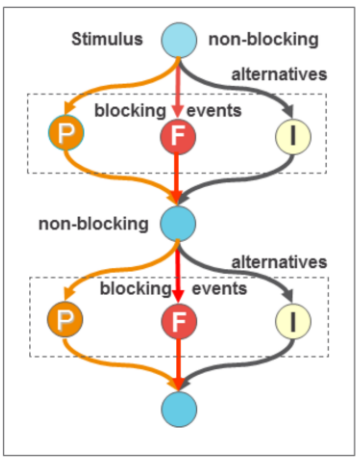

**Verdict: pass, fail,**



# TEST CASES IN BLACK-BOX TEST

- Focus on a single **requirement**
- Returns **verdict** (pass, fail, inconclusive)
- Typically a sequence of action-observation-verdict update:
  - **Action** (stimulus): non-blocking (e.g. transmit PDU, start timer)
  - **Observation** (event): takes care of multiple alternative events (e.g. expected PDU, unexpected PDU, timeout)



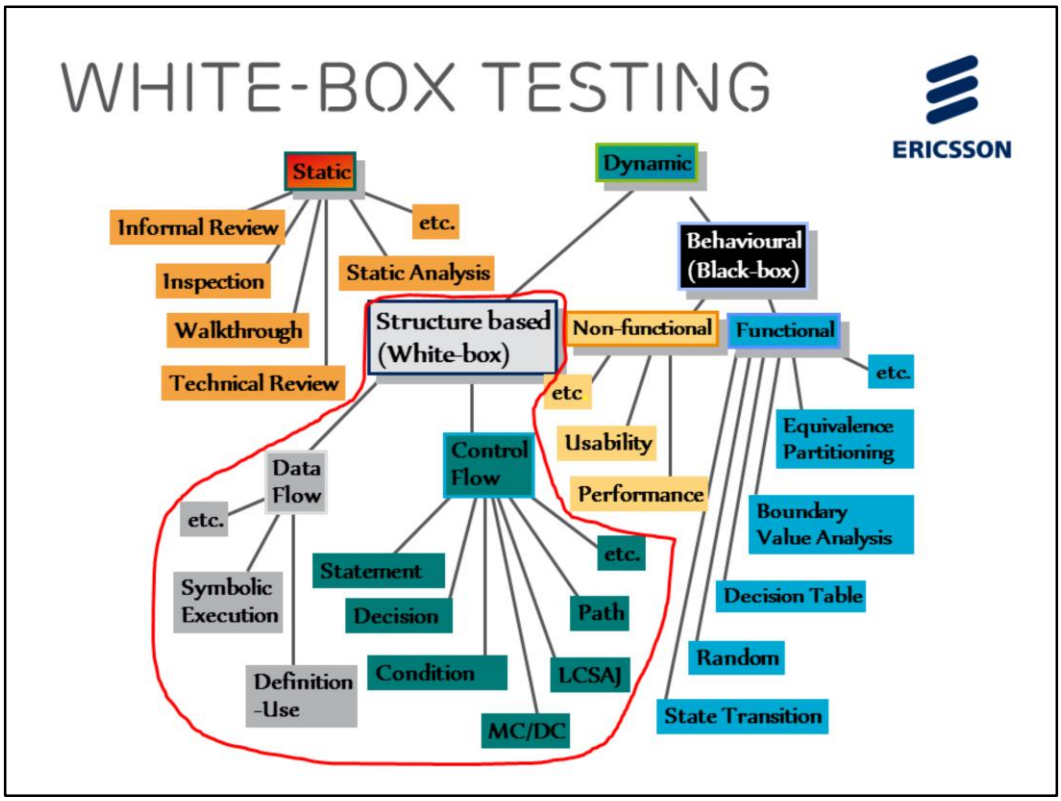

White-box testing | Ericsson Internal | © Ericsson AB 2017 | 2017-04-19 | Page 14

**Black-box** testing means that the internal structure of the tested software product is not known: the only way to test it is to send a message ("stimulus") to the system and to analyse the received response. The latter is compared to the due response determined beforehand using the reference specification. If the comparison ("pattern matching") between the real and the expected response fails, the test case is considered as "failed" otherwise "passed".

The test script language must have means to match the expected and the received messages even if the message elements arrive in different order, or some of them (the optional ones) are missing. Usually, there are more than one possible responses; all of them must be accepted.

Once the match is determined, the next stimulus is constructed taking into consideration the data having received in the response, and so on.

The test script language must be prepared to determine that the expected response is not received within the specified time frame: it must handle timing ("temporal") requirements.



# NEED FOR WHITE BOX TESTING

- Most of the functional and performance issues arise due to **bad coding**
- White box tests do the same by getting into the **internals** of every program
- **Every developer** is by default a white box tester
- If your application must scale to a **very large extent**, white box tests are inevitable

## BLACK BOX CANNOT TEST THESE

- What **path** the program took to achieve the end result
- Is there any **dead** and **unused code**
- Is there any **extra code** that is not needed
- Is the code compliant to **coding standards**
- Code **coverage**

# THE GENERAL WHITE BOX TESTING PROCESS

1. The SUT's implementation is **analyzed**
2. **Paths through** the SUT are identified
3. **Inputs are chosen** to cause the SUT to execute selected paths
4. **Expected results** for those inputs are determined
5. The **tests** are **run**
6. Actual **outputs are compared** with the expected outputs
7. A **determination** is made as to the proper functioning of the SUT



# APPLICABILITY

- White box testing can be applied at **all levels** of system development
  —unit, integration, and system.
- White box testing is more than code testing—it is not only **path** testing.
- We can apply the same techniques to test paths between **modules within subsystems**, between **subsystems within systems**, and even **between entire systems**.



# ADVANTAGES

- The tester can be sure that **every path** through the software under test has been **identified and tested**
- Side effects of having the **knowledge of the source code** is beneficial to thorough testing.
- **Optimization of code** by revealing hidden errors and being able to remove these possible defects.
- White box tests are **easy to automate**.
- White box testing give clear, engineering-based, rules for **when to stop testing**.



# DISADVANTAGES

- The number of **execution paths** may be so **large** then they cannot all be tested.
- The test cases chosen may not detect **data sensitivity errors**.
  - For example: p=q/r; may execute correctly except when r=0.
- White box testing **assumes the control flow is correct** (or very close to correct). Since the tests are based on the existing paths, **non-existent paths cannot be discovered** through white box testing.
- The tester must have the **programming skills** to understand and evaluate the software under test.



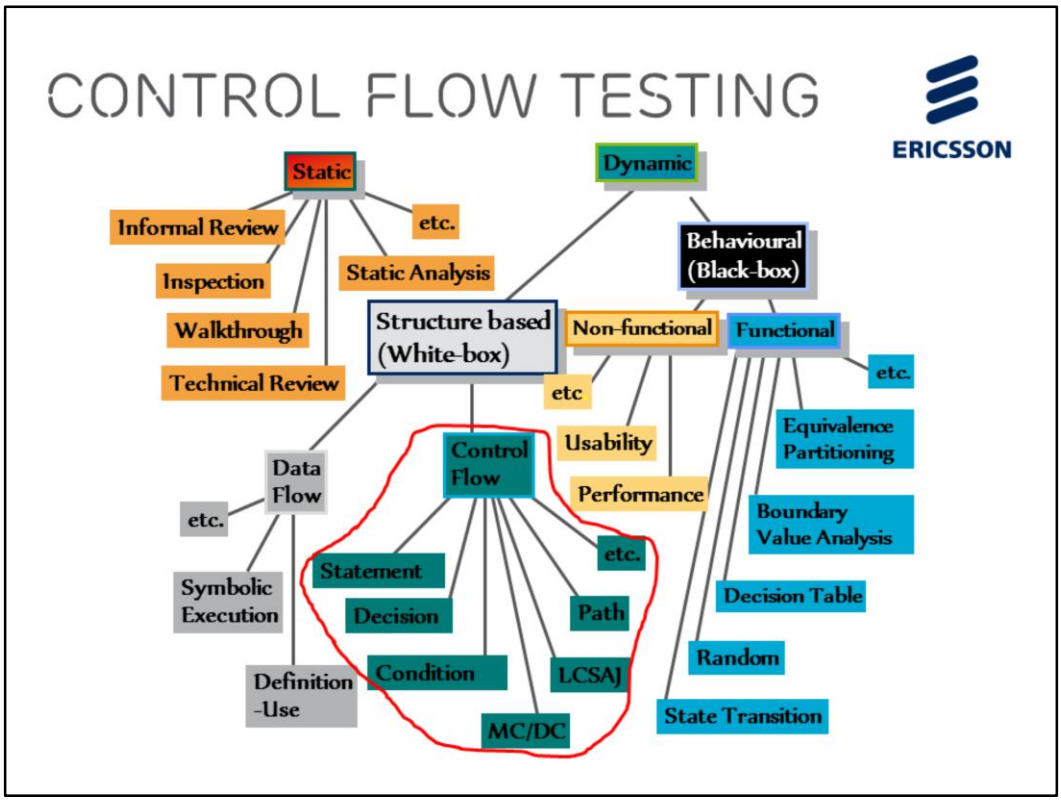
 

# WHITE BOX TESTING TECHNIQUES

- Control Flow Testing
  - Path Testing
  - Coverage Testing
- Data Flow Testing

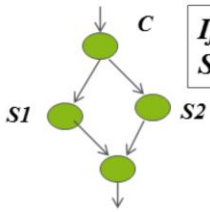


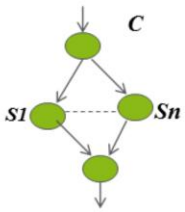

CONTROL FLOW TESTING
ERICSSON
Static
Informal Review
Inspection
Walkthrough
Technical Review
etc.
Static Analysis
Dynamic
Behavioural (Black-box)
Structure based (White-box)
Non-functional
Functional
etc
etc.
Usability
Performance
Equivalence Partitioning
Boundary Value Analysis
Decision Table
Random
State Transition
Data Flow
etc.
Symbolic Execution
Definition -Use
Control Flow
Statement
Decision
Condition
MC/DC
Path
LCSAJ
etc.

# CONTROL FLOW TESTING ELEMENTS

*Sequential statement block*




*If C Then S1 else S2;*




*Case C of*

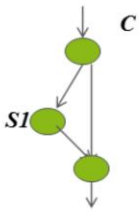
*L1: S1;*

*L2: S2;*

*…*

*Ln: Sn;*

*end;*

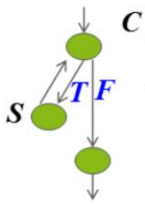



*If C Then S1;*

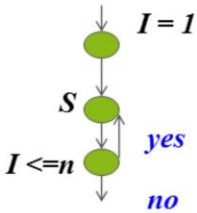


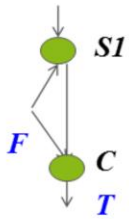
# CONTROL FLOW TESTING ELEMENTS




*While C do S;*




***For** loop:*
*for I = 1 to n do S;*




*Do loop:*
*do S1 until C;*



# FLOW GRAPH EXAMPLE

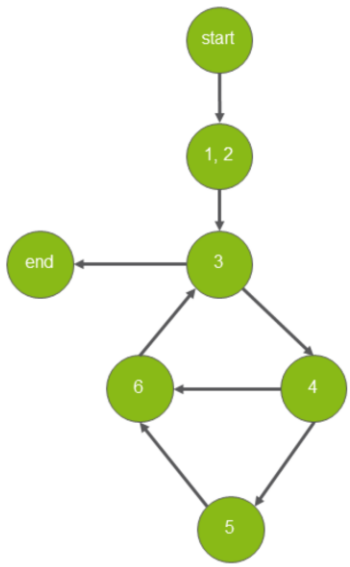
G = (V, E) where
- V is the set of basic blocks
- E is the set of control branches

Example:

```
1.  a = Read(b)
2.  c = 0
3.  while (a > 1)
4.    If (a^2 > c)
5.      c = c + a
6.    a = a - 2
```

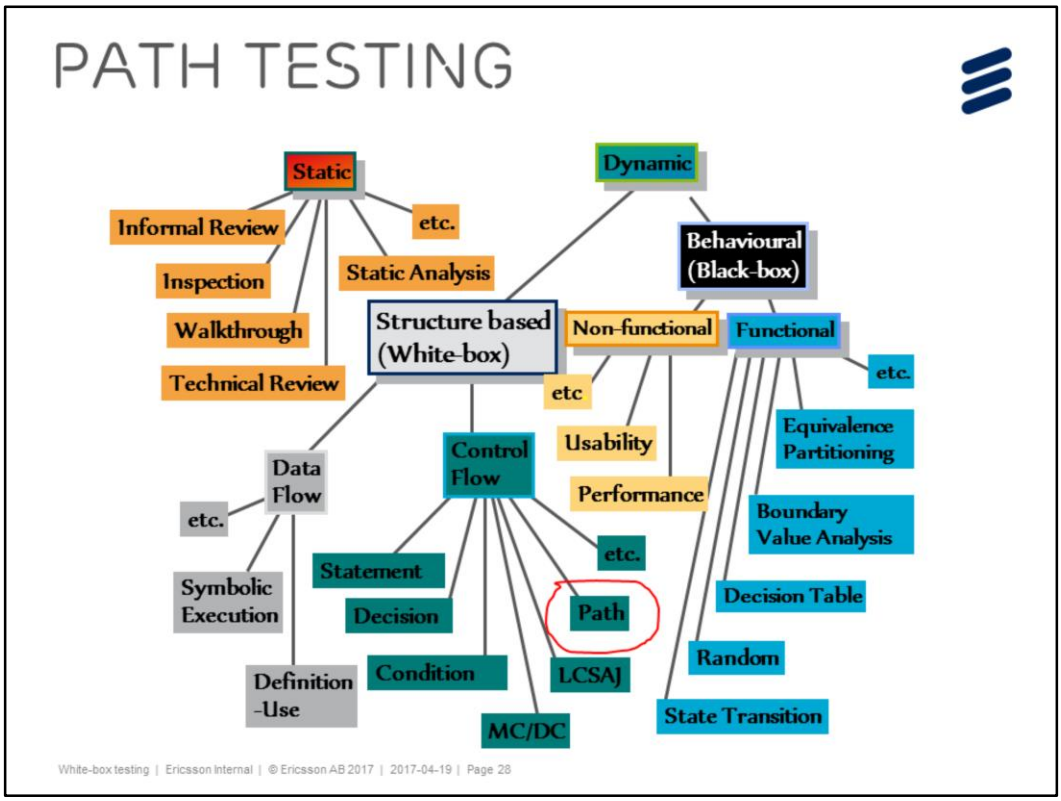
Input:
b = 2

Output:
a = 0, c = 2






PATH TESTING
Static
Informal Review
Inspection
Walkthrough
Technical Review
etc.
Static Analysis
Dynamic
Behavioural (Black-box)
Structure based (White-box)
Non-functional
Functional
etc
etc.
Usability
Performance
Equivalence Partitioning
Boundary Value Analysis
Decision Table
Random
State Transition
Data Flow
etc.
Symbolic Execution
Definition -Use
Control Flow
Statement
Decision
Condition
MC/DC
Path
LCSAJ
etc.
White-box testing | Ericsson Internal | © Ericsson AB 2017 | 2017-04-19 | Page 28

# PATH TESTING

- Identifies the ***execution paths*** through a module of program code
- Creates and executes test cases to cover those paths.
  - ***Path***: A sequence of statement execution that begins at an entry and ends at an exit.
  - An element of an ***Independent path set*** is any path through the code that introduces at least one new set of processing statements or a new condition.
  - ***Basis Path Testing*** is a means for ensuring that all independent paths through a code module have been tested



# CYCLOMATIC COMPLEXITY

- Provides a quantitative measure of the **logical complexity** of a program
- Defines the **number of independent paths** in the basis set
- Provides an **upper bound** for the number of tests that must be conducted to ensure **all statements** have been executed **at least once**
- Can be computed **three** ways
  - The number of **regions**
  - **V(G) = E – N + 2**, where E is the number of edges and N is the number of nodes in graph G
  - **V(G) = P + 1**, where P is the number of predicate nodes in the flow graph G



Regions are the faces in a planar graph.

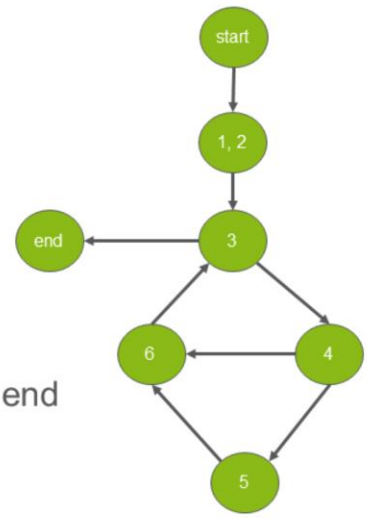
# DERIVING THE BASIS SET AND TEST CASES

1. Using the design or code as a foundation, **draw a** corresponding **flow graph**
2. Determine the **cyclomatic complexity** of the resultant flow graph
3. Determine a **basis set of linearly independent paths**
4. Prepare **test cases** that will force execution of **each path in the basis set**



# BASIC PATH COVERAGE



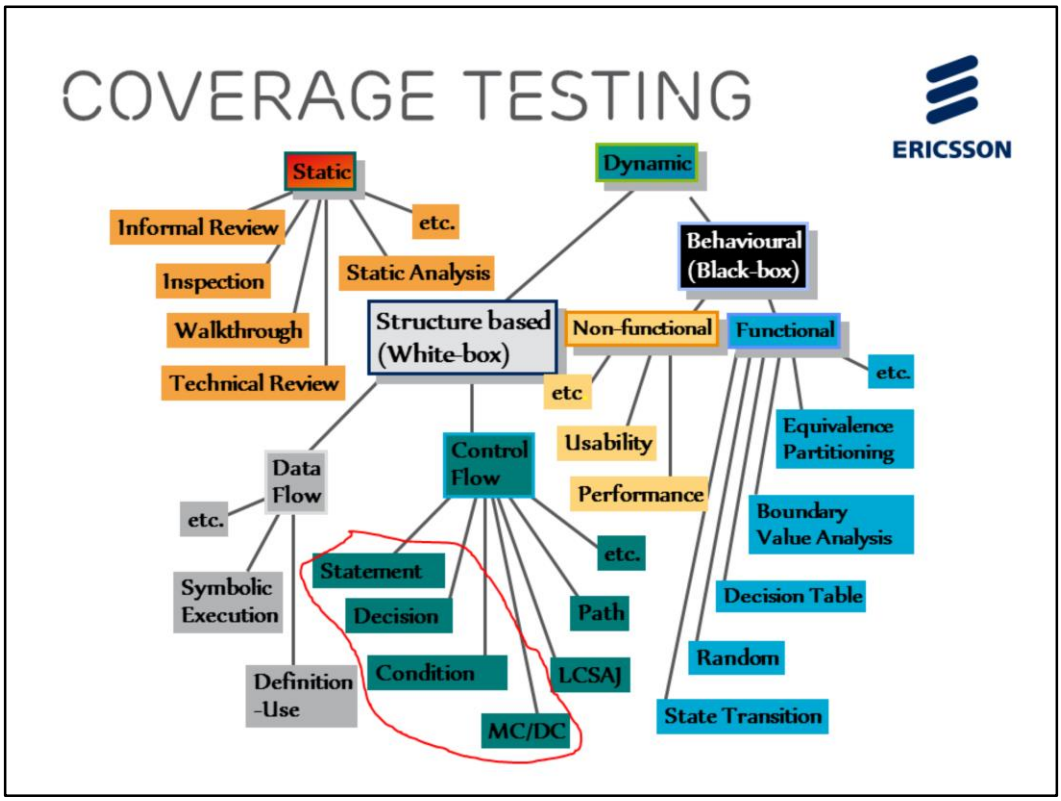

- The number of Basic paths is
  *E – N + 2*

- **Example**
  p1 = start – 1,2 – 3 – end
  p2 = start – 1,2 – 3 – 4 – 6 – 3 – end
  p3 = start – 1,2 – 3 – 4 – 5 – 6 – 3 – end

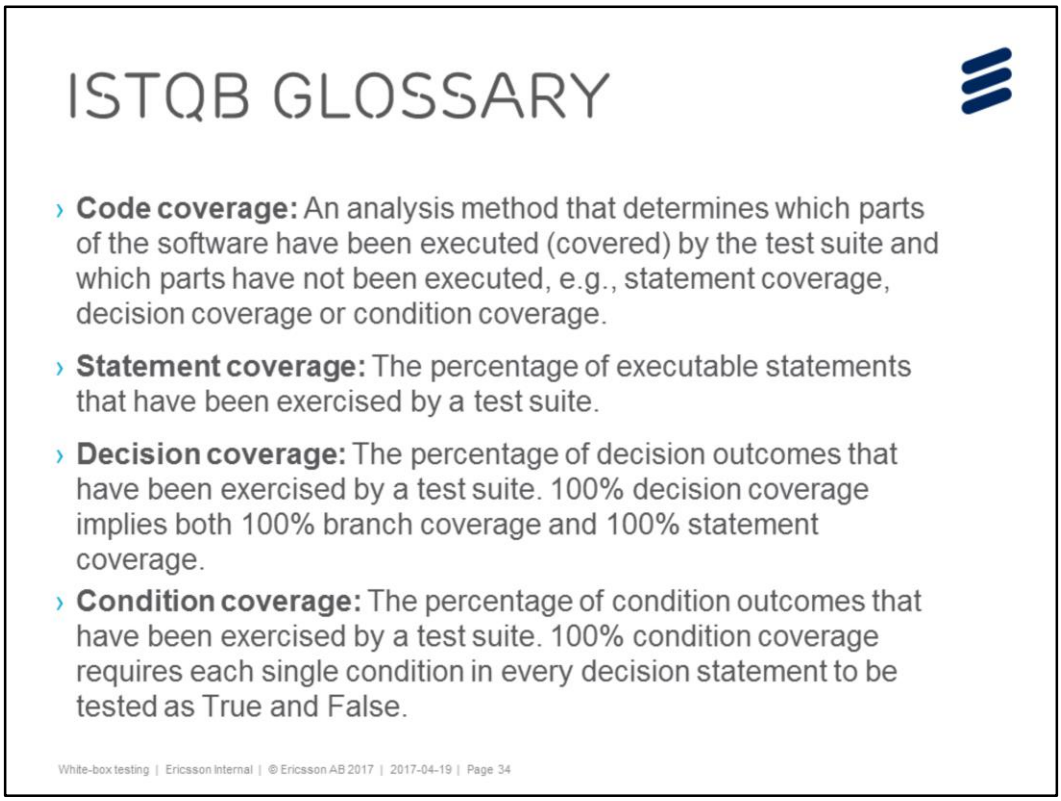
  E – N + 2 = 8 – 7 + 2 = 3








100% implication is true if there is no early evaluation

# COVERAGE OUTPUT

Example:

| Line | Code | Time |
|---|---|---|
| 47 | myproc (p1, p2) | 7181 |
| 48 | if (p1 < p2) | 2 |
| 49 | call proc1(); | 388 |
| 50 | if (p1 > p2) | 0 |
| 51 | call proc2(); | 0 |
| 52 | if (p1 == p2) | 2 |
| 53 | call proc3(); | 6789 |



# COVERAGE GOALS

- Coverage must have 100% at the end of all tests for every function (new projects)
- There must be 1 test for **every relational operation in conditions**
- There must be 1 test for **every loop termination**
- There must be 1 test for **every exception**
- There must be 1 test for **every test exit point** (if method has multiple return points)

# EXAMPLE: SAMPLE CODE FOR COVERAGE ANALYSIS

```
1   float foo (int a, int b, int c, int d, float e)  {
2       float e;
3       if (a == 0)  {
4           return 0;
5       }
6       int x = 0;
7       if ((a==b) OR ((c == d) AND bug(a) )) {
8           x=1;
9       }
10        e = 1/x;
11      return e;
12  }
```

37



# STATEMENT COVERAGE

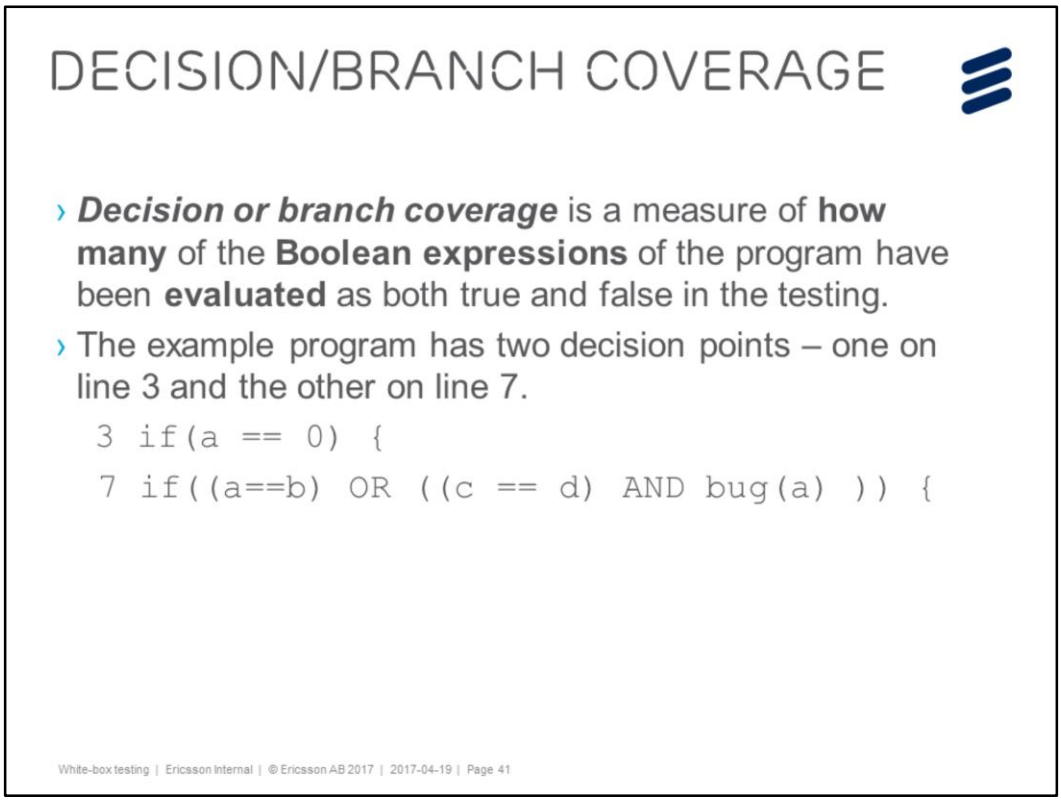
- ***Statement coverage*** is a measure of the **percentage of program statements** that are run when your tests are executed.
- The objective should be to achieve **100% statement coverage** through your testing.
- Identify the **cyclomatic number** and executing this minimum set of test cases will make this statement coverage achievable.
- ***Test Case 1***: call the method **foo(0, 0, 0, 0, 0.)**, expected return value of 0.
- In ***Test Case 1***, we executed the program statements on lines 1-5 out of 12 lines of code - a 42% (5/12) statement coverage.

# STATEMENT COVERAGE(CONT.)

- To attain 100% statement coverage, one should execute an additional test case.
- ***Test Case 2***: the method call **foo(1, 1, 1, 1, 1.)**, expected return value of 1.
- This executes the program statements on lines 6-12 - a 100% statement coverage.






Decision and branch coverage are the same in ISTQB terminology. In other literature they are differs.

# DECISION/BRANCH COVERAGE(CONT.)

- For decision/branch coverage, evaluate an entire Boolean expression as one true-or-false predicate even if it contains multiple logical-and or logical-or operators.
- We need to ensure that each of these predicates (compound or single) is tested as both true and false.

Decision Coverage

| Line # | Predicate | True | False |
|---|---|---|---|
| 3 | (a == 0) | Test Case 1 | Test Case 2 |
| | | **foo(0, 0, 0, 0, 0)** | **foo(1, 1, 1, 1, 1)** |
| | | **return 0** | **return 1** |
| 7 | ((a==b) OR ((c == d) AND bug(a) )) | Test Case 2 | |
| | | **foo(1, 1, 1, 1, 1)** | |
| | | **return 1** | |



# DECISION/BRANCH COVERAGE(CONT.)

- Three of the four necessary conditions - 75% branch coverage.
- We add ***Test Case 3***: **foo(1, 2, 1, 2, 1)** to bring us to 100% branch coverage( making the Boolean expression False).
- The objective is to **achieve 100% branch coverage** in your testing.
- In **large systems** only **75%-85%** is practical.
- Only **50%** branch coverage is practical in **very large systems** of 10 million source lines of code or more.



# CONDITION COVERAGE

- ***Condition coverage*** reports the true or false outcome of *each* Boolean sub-expression of a compound predicate.
- In line 7 there are three sub-Boolean expressions to the larger statement (a==b), (c==d), and bug(a).
- Condition coverage measures the outcome of each of these **sub-expressions independently** of each other.
- With condition coverage, you ensure that each of these sub-expressions has independently been tested as both true and false.



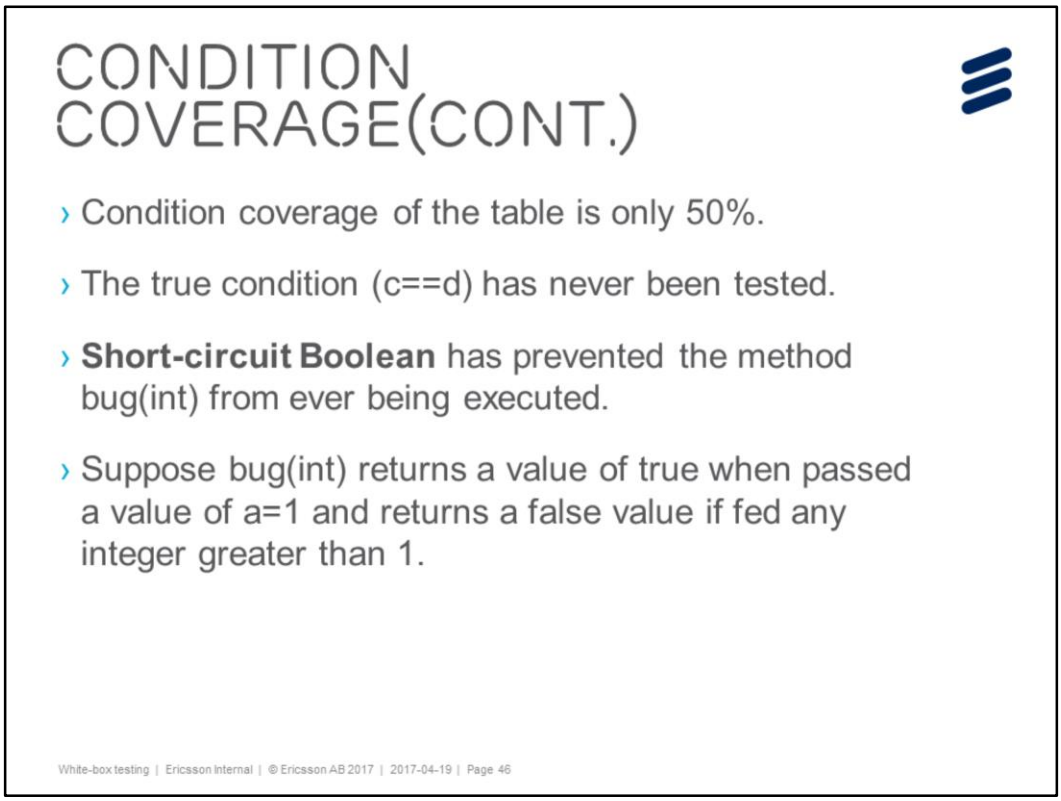
# CONDITION COVERAGE(CONT.)

Condition coverage

| Predicate | True | False |
|---|---|---|
| (a==b) | Test Case 2 **foo(1, 1, x, x, 1) return value 0** | Test Case 3 **foo(1, 2, 1, 2, 1) division by zero!** |
| (c==d) | | Test Case 3 **foo(1, 2, 1, 2, 1) division by zero!** |
| bug(a) | | |






Short-circuiting is where an expression is stopped being evaluated as soon as its outcome is determined. So for instance:

```
if (a == b || c == d || e == f) { // Do something }
```

If a == b is true, then c == d and e == f are **never evaluated at all**, because the expression's outcome has already been determined.

# CONDITION COVERAGE(CONT.)

- ***Test Case 4*** address test (c==d) as true: **foo(1, 2, 1, 1, 1)**, expected return value 1.
- When we run the test case, the function bug(a) actually returns false, which causes our actual return value (division by zero) to not match our expected return value.
- This allows us to detect an error in the bug method. Without the addition of condition coverage, this error would not have been revealed.
- To finalize our condition coverage, we must force bug(a) to be false.



# CONDITION COVERAGE(CONT.)

- ***Test Case 5***, **foo(3, 2, 1, 1, 1)**, expected return value "division by zero error".
- The condition coverage thus far is shown in the Table.

**Condition Coverage Continued**

| Predicate | True | False |
|---|---|---|
| (a==b) | Test Case 2 **foo(1, 1, x, x, 1) return value 0** | Test Case 3 **foo(1, 2, 1, 2, 1) division by zero!** |
| (c==d) | Test Case 4 **foo(1, 2, 1, 1, 1)** return value 1 | Test Case 3 **foo(1, 2, 1, 2, 1) division by zero!** |
| bug(a) | Test Case 4 **foo(1, 2, 1, 1, 1)** return value 1 | Test Case 5 **foo(3, 2, 1, 1, 1) division by zero!** |



# CONDITION COVERAGE(CONT.)

- There are **no industry standard** objectives for condition coverage, but we suggest that you keep condition coverage in mind as you develop your test cases.
- Our condition coverage revealed that some additional test cases were needed.

# EXHAUSTIVE TESTING DRAWBACKS

- The number of paths could be huge and thus untestable within a **reasonable amount of time**.
  - Every decision doubles the number of paths and
  - Every loop multiplies the paths by the number of iterations through the loop.
- For example:

```
for (i=1; i<=1000; i++)
     for (j=1; j<=1000; j++)
       for (k=1; k<=1000; k++)
         doSomethingWith(i,j,k);
```

executes doSomethingWith() one billion times (1000×1000×1000).



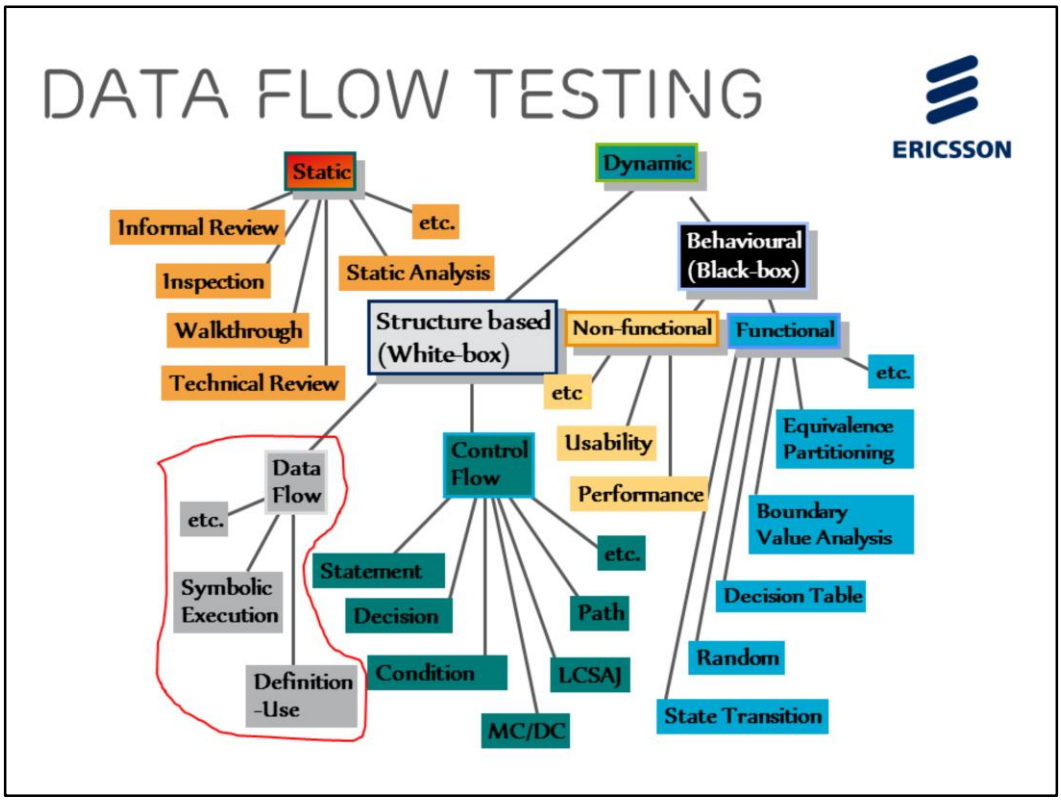
# EXHAUSTIVE TESTING DRAWBACKS(CONT)

› Paths called for in the specification may simply be missing from the module. Any testing approach based on implemented paths **will never find paths that were not implemented**.

```
if (a>0) doIsGreater();
if (a==0) dolsEqual();
// missing statement - if (a<0) dolsLess();
```



Negative path


DATA FLOW TESTING
ERICSSON
Static
Dynamic
Informal Review
etc.
Inspection
Static Analysis
Behavioural (Black-box)
Walkthrough
Structure based (White-box)
Non-functional
Functional
Technical Review
etc
etc.
Usability
Equivalence Partitioning
Data Flow
Control Flow
Performance
Boundary Value Analysis
etc.
etc.
Statement
Symbolic Execution
Decision
Path
Decision Table
Random
Definition -Use
Condition
LCSAJ
State Transition
MC/DC

# DATA FLOW TESTING

› Data flow testing is a powerful tool to detect improper use of data values due to coding errors.

```
main() {
          int x;
          if (x==42){...}      // Data error!
        }                      // x is undefined
```

# DATA FLOW TESTING

- Variables that contain data values have a defined life cycle. They are **created**, they are **used**, and they are **killed** (destroyed) - Scope

```
{               // begin outer block
  int x;        // x is declared as an integer within this outer block
  …;            // x can be accessed here
  {             // begin inner block
    int y;      // y is declared within this inner block
    …;          // both x and y can be accessed here
  }             // y is automatically destroyed at the end of this block
  …;            // x can still be accessed, but y is gone
}               // x is automatically destroyed
```



# DATA FLOW TESTING

- Variables can be used
  - in expression
  - in conditionals
- Possibilities for the first occurrence of a variable through a program path
  - ~d the variable does not exist, then it is **defined** (d)
  - ~u the variable does not exist, then it is **used** (u)
  - ~k the variable does not exist, then it is **killed** or **destroyed** (k)



# DATA FLOW TESTING

› Examine **time-sequenced pairs** of defined (d), used (u), and killed (k):

dd - not invalid but suspicious, probably a programming error

du - perfectly correct, the normal case

dk - not invalid but probably a programming error

ud - acceptable

uu - acceptable

uk - acceptable

kd - acceptable.

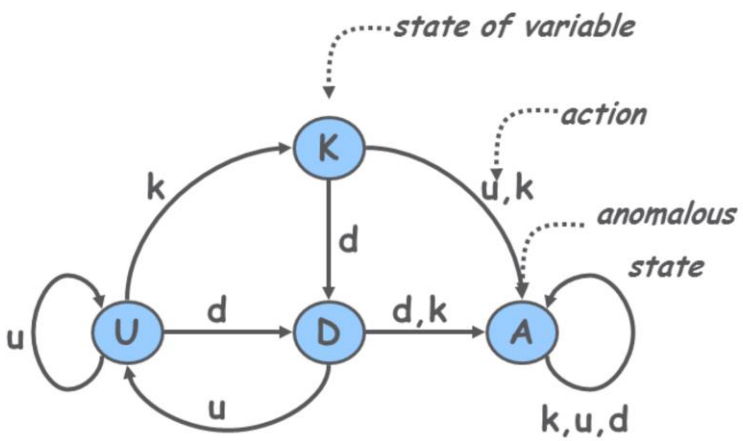
ku - a serious defect. Using a variable that does not exist or is undefined is always an error.

kk - probably a programming error.



# DATA FLOW ANOMALY



# STATIC DATA FLOW TESTING

- **Static testing cannot find all errors**

  Examples:

  - Arrays are collections of data elements that share the same name and type. For example:

```
int test[100]; //defines an array named test
               // consisting of 100 integer elements,
               // named test[0], test[1], etc.
```

  - Arrays are defined and destroyed **as a unit** but **specific elements** of the array are used individually.
  - Static analysis **cannot determine** whether the define-use-kill rules have been followed properly unless each element is considered individually.



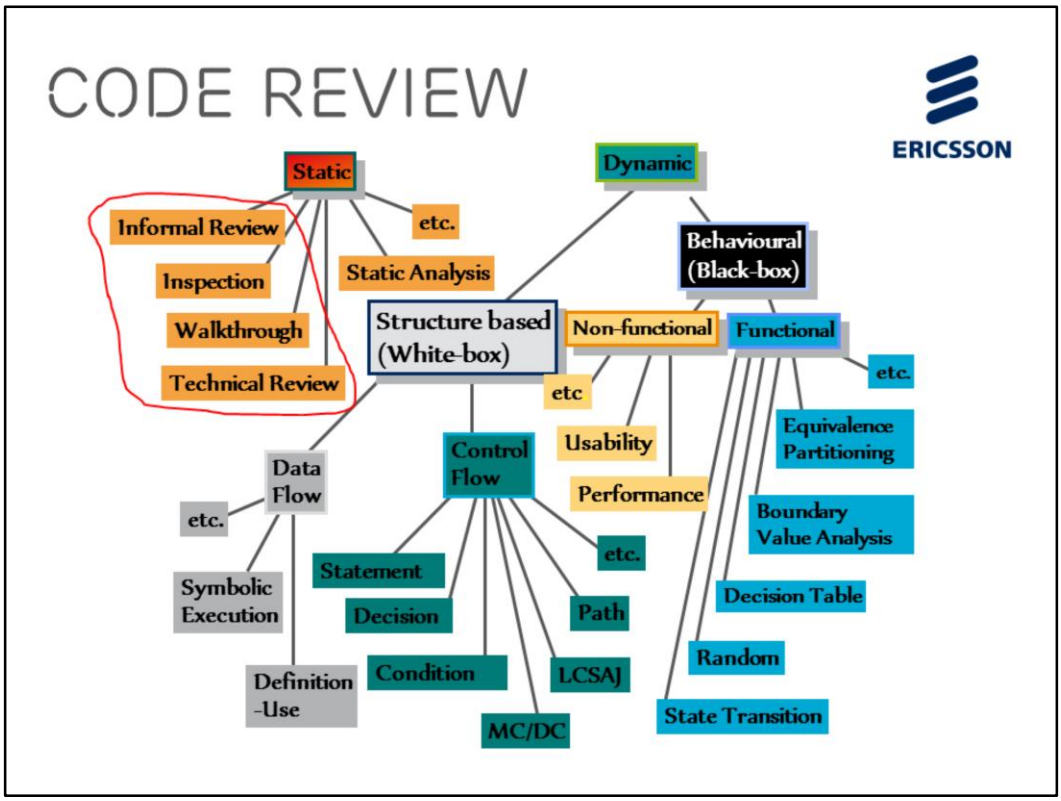
 

# DYNAMIC DATA FLOW TESTING

- Data flow testing is based on a module's control flow, it assumes that the **control flow is basically correct**.
- The data flow testing process is to choose enough test cases so that:
  - Every "define" is traced to each of its "uses"
  - Every "use" is traced from its corresponding "define"
  - To do this,
    1. Enumerate the paths through the module.
    2. Begin at the module's entry point, take the leftmost path through the module to its exit.
    3. Return to the beginning and vary the first branching condition. Follow that path to the exit.
    4. Repeat until all the paths are listed.
    5. For every variable, create at least one test case to cover every define-use pair.




CODE REVIEW
ERICSSON
Static
Dynamic
Informal Review
Inspection
Walkthrough
Technical Review
etc.
Static Analysis
Behavioural (Black-box)
Structure based (White-box)
Non-functional
Functional
etc
etc.
Usability
Performance
Equivalence Partitioning
Boundary Value Analysis
Decision Table
Random
State Transition
Data Flow
Control Flow
etc.
Symbolic Execution
Definition -Use
Statement
Decision
Condition
MC/DC
Path
LCSAJ
etc.

# CODE REVIEW

- Code review can **save upto 30%** of testing cost at a later point in time
- Usual problems happen due to **cut and paste** of code where it is not required
- When there is **a transition from one developer to another**, code goes thru challenges
- **Quick fixes** on big projects cause enormous amount of turbulence on production systems
- Ideally **senior people** must review code
- Spend at least **10% of coding** time on code review



# CODE REVIEW CHECKLIST

Just try these 9 critical points in daily life

1. No hard coding
2. Ensure loop termination conditions
3. No object creation inside loops
4. Close every object that you open
5. Give comments to every code block
6. Use database connection only when you need
7. Remove unused variables and code portions
8. Follow a consistent naming convention
9. Try to reduce overloading methods very often

# UNIT TESTING IN GENERAL

- Scope: one component from the design
  - Unit can be a **page** or a **function** or a **method** within a **class** or a **whole program** itself
- Responsibility of the **developer**
  - Not the job of an independent testing group
- Both **white-box** and **black-box** techniques are used for unit testing
- Maybe necessary to create **stubs**:
  - If modules not yet implemented or not yet tested



# UNIT TESTING

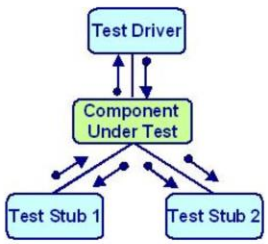
- Unit test must focus on
  - **data type** of every parameter
  - **data format** of every parameter
  - **boundary values** of every parameter
- For any given code, what deliverable do we provide as part of unit testing?
  - Most of the times it is just the **trust on the developer**



# STUBS AND DRIVERS

- When the program under test needs some **pre-built data or state**, we need to mimic the same
- This is a short-cut approach, but it serves the purpose
- **Stub** is the one that **mimics a called function**
- **Driver** is the one that **mimics a caller function**
- When we use stubs and drivers, it is a must that we ensure that the data created by them are destroyed soon after the test

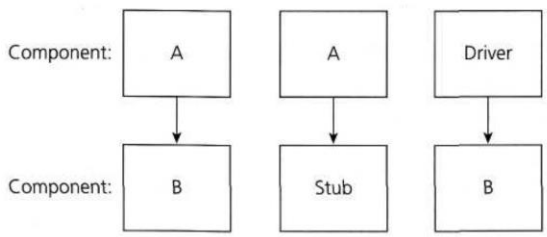




# UNIT TESTS

- Mocking:
  - substitutes its own object (the “mock object”) for an object that talks to the outside world
  - checks that it is called correctly and provides a pre-scripted response
- Stubs and Drivers




White-box testing | Ericsson Internal | © Ericsson AB 2017 | 2017-04-19 | Page 67

*Mock objects* are a popular tool for isolating classes for unit testing. When using mock objects, your test

substitutes its own object (the “mock object”) for an object that talks to the outside world. The mock object

checks that it is called correctly and provides a pre-scripted response. In doing so, it avoids time-consuming

communication to a database, network socket, or other outside entity.

Beware of mock objects. They add complexity and tie your test to the implementation of your code. When

you’re tempted to use a mock object, ask yourself if there’s a way you could improve the design of your code

so that a mock object isn’t necessary. Can you decouple your code from the external dependency more cleanly?

Can you provide the data it needs—in the constructor, perhaps—rather than having it get the data itself?

Mock objects are a useful technique, and sometimes they’re the best way to test your code. Before you assume

that a mock object is appropriate for your situation, however, take a second look at your design. You might

have an opportunity for improvement.

# BASIC STRATEGY FOR UNIT TESTING

1. Create **black-box** tests
   - Based on the specification of the unit (as determined during design)
2. Evaluate the tests using **white-box techniques** (test adequacy criteria)
   - How well did the tests **cover statements**, **branches**, **paths**, **DU-pairs** (Def-Use), etc.?
   - Many possible criteria; at the very least need **100% branch** coverage
3. Create **more tests when needed**: e.g., to increase coverage of DU-pairs



# UNIT TESTING CHECKLIST

1. Test case for custom exceptions.
2. Test case for system exceptions.
3. Test case for “body” of an if condition.
4. Test case for “body” of an else condition.
5. Test case for every loop termination.
6. Test case for every recursion termination.
7. Test case for pointers release (for memory leaks).
8. Test case for every procedure entry and exit.
9. Test case for every parameter validation for procedures.
10. Test case for resource release (Closing DB connections, releasing objects etc.)

# AUTOMATE UNIT TESTS

- To reduce time, we need **to write a program**, that calls the program under test and pass parameters
- Ideally the test program must be in the **same language** as the program being tested
  - This will help in maintaining the homogeneity of the data type of parameters passed
- Since developers know the programming language, it is **easy** for them **to write the test program** as well
- The **investment** on the unit test automation **is one time** as long as changes are minimal
- The person who automates test for a program **must know the program well**



# TESTING APIS

- APIs do **exchange of data** from one piece to another
- APIs are usually consumed by **many independent programs**
- APIs in general are meant to provide a window of information **to 3rd parties**
- One **change** in API **will affect** all consumers at once
- Most of the current day apps provide APIs
- Some APIs do require **authentication** mechanism
- APIs fall under **external interfaces**



# INSTRUMENTATION

- White box tools and profiler tools use one single concept called Instrumentation
- **Source code** is **compiled and built** as binary
- These tools understand the format of these binary files – where **function entry points** are made, how they **exit**, where **stack** is maintained etc.
- Before **profiling**, they **instrument the binary** – inject their own log codes in the binary locations
- During the execution, the instrumented code is run and **every detail is logged** using these injected code (such as entered function x, executed loop y, exited function y, destroyed object o etc.)
- Using these logs, the tools provide **internal details**



# INSTRUMENTATION OVERHEADS

- When profilers run along with the instrumented programs, it adds **load to the machine**
- This is a **necessary overhead** we need to take
- The performance **overheads are negligible** when compared to actual object size
- The statistics that are collected out of the programs will vary from time to time; but if the variance is **beyond 5%,** then the tool has **real issues**
- Also, do not run **2 competing tools** on the same machine. The effects are unpredictable

# MEMORY LEAKS

- When a memory pointer is **declared and initialized** and it is not released, it **is a leak**
- Even in managed code sections, the leaks are there for a time till the garbage collector acts
- Leaked memory locations are **useless** and **vulnerable**
- One program leaking **10 bytes of memory per call** – can **bring down** the system **in a day** when 1000s of users use the same for 100s of transaction
- This is very important when we deal with device drivers and embedded systems



# MEMORY PROFILING

- This **happens at runtime**
- Profiling is the basic block of **performance engineering**
- Concept of instrumentation and tracing happens
- Call graph will tell what is the path the program touched in terms of functions
- **Statistical profilers** talk about memory and cpu
- Memory consumed at that instance by variables and functions
- Memory released at any given point of time

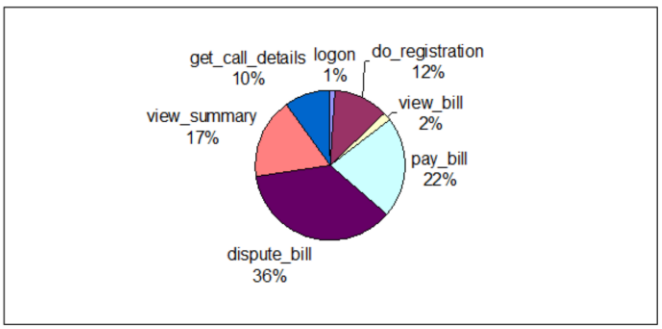
# MEMORY QUANTIFICATION

- What is the consumption of memory and cpu is quantified
- Usual quantification happens in **basic unit of measurement**
- UOM can be **KB** or **machine cycles**
- **How many cpu cycles** are spent in this method when this is **running**
- **How many cpus cycles** are spent to initialize the class



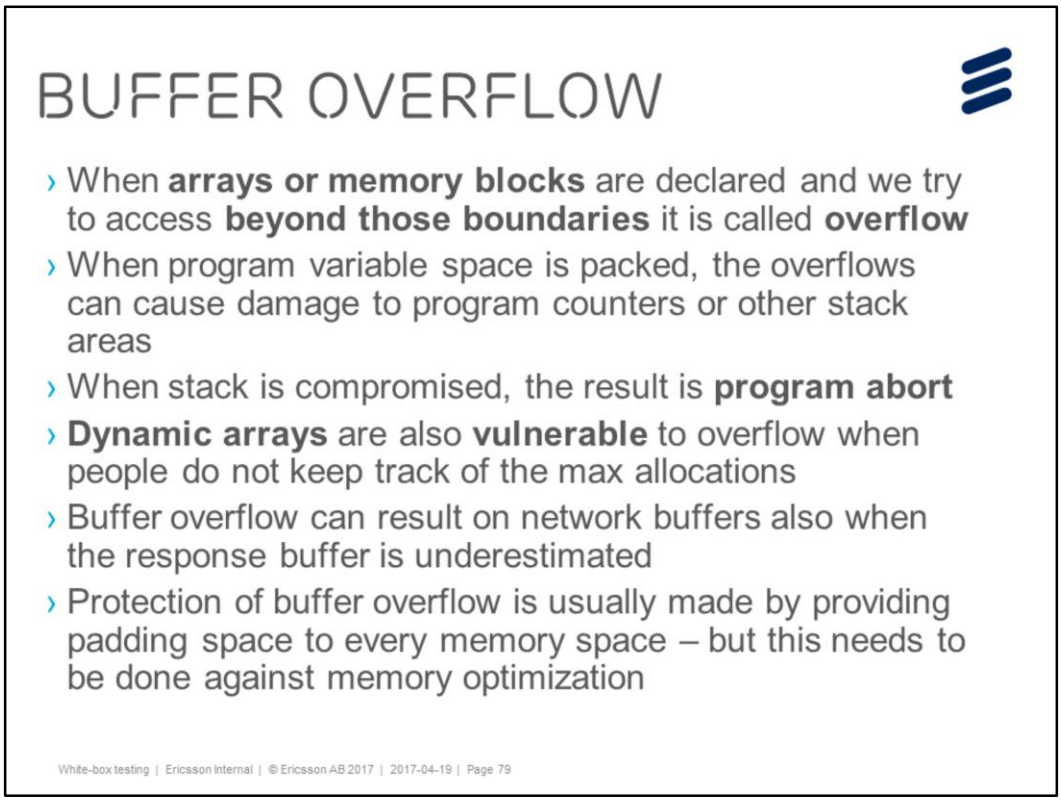
# MEMORY USAGE




| Method | Kbytes |
|---|---|
| logon | 138 |
| do_registration | 1607 |
| view_bill | 254 |
| pay_bill | 2897 |
| dispute_bill | 4876 |
| view_summary | 2346 |
| get_call_details | 1345 |






This is an important topic from security point of view.

# OBJECT ISSUES

- Objects require space and the methods need cpu cycles
- When objects are **declared and not destroyed**, they also cause **leaks**
- **Unused objects** are vulnerable areas for attack
- **Closure and nullifying objects** are essential for optimized use of memory
- Creating a series of objects in a multi user environment can cause spiral problems

# FUNCTION-WISE QUANTIFICATION

- Every function has **variables and statements**
- Parameters and variables take space from **symbol table** as well as stack
- Statements do take **cpu cycles** based on **how many times** they get **executed**
- When one function calls another, there is a huge amount of context switch happening at heap and program registry levels
- How many times these **context switches happen** will also determine the **complexity** of the programs

# METRICS COLLECTION

- **Average number of unit test** cases per page/component/any specific unit in the project
- Total **number of unit test** cases
- Total **number of unit test cases failed** in test pass-1
- Total **number of unit test cases failed** in test pass-2
- Total **number of unit test cases failed** in test pass-n
- Total number of **memory leaks found**
- Percentage **code coverage** as per the tool statistics
- **Execution time** for each test run
- Total test execution time for all units
- **Ratio** of # of **defects** found **in unit testing** to # of **defects** found **in code review**
- **Defect distribution** across units (asp related, db related, parameter related etc.)



# METRICS ANALYSIS

- **Trend analysis** helps in seeing whether the **error rate moves up or down**
- See pattern of failures and find out the **corrective action**
- Metrics **vary** a lot from **developer to developer**; hence find out error rate pattern across developers
- Analyze failure pattern and time of failure
- It is not easy to catch **all errors in one round of testing**; hence do **more rounds; more often**



# CONFIGURATION MANAGEMENT

- Since we develop test code, we need to **check in these code sections** to VSS or subversion or PVCS
- Maintain a uniform x.y version code to all test cases and suites
- Check in the test data along with test code as well
- Have **one single person who can oversee** all the **CM operations**
- **Never allow** people to have **local copies** of test code



# UNIT TEST PRACTICES

- Keep adding more tests **every day**
- **More data** preparation is the key
- Test often - even **twice a day**
- Run unit tests on a **clean test bed**
- **Review** unit test code as well
- Run regression unit tests **daily**
- Run unit tests by a **different developer**
- Make unit test code **simple**
- **Never** try to have **if then else** in a test code
- Do **not** have **loops** in a test code
- Run **critical tests first**



# DON'TS

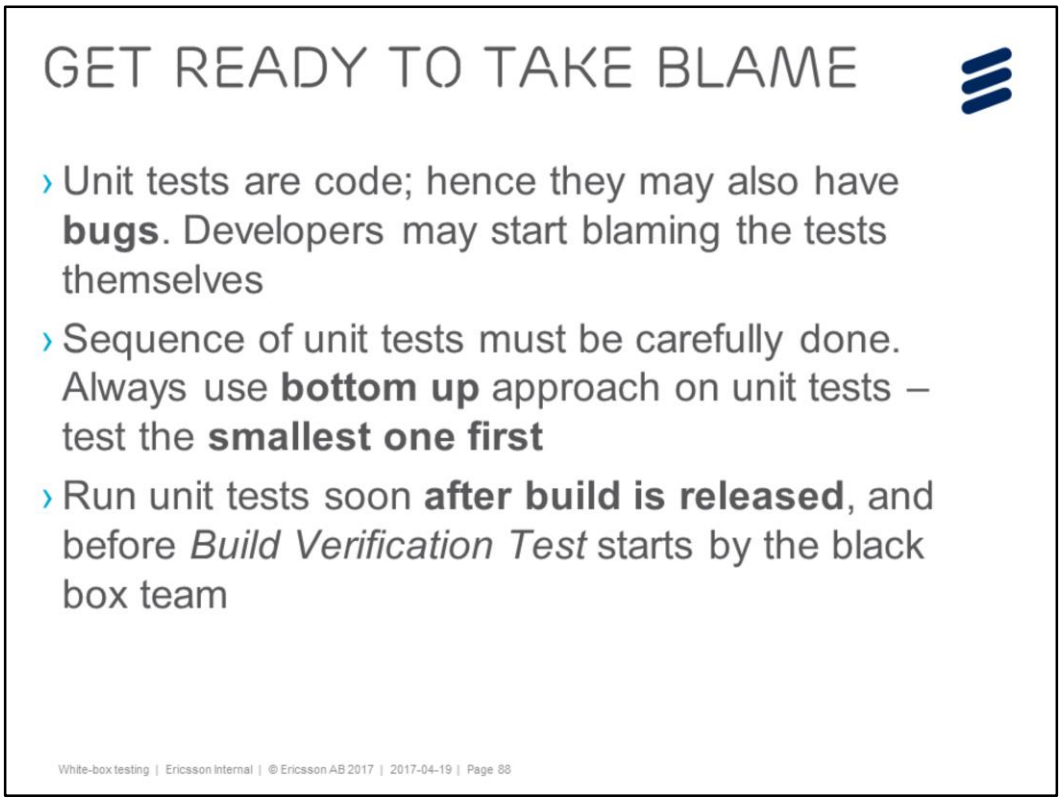
- Do not test **entire functionality** in one test
- **One test** must satisfy just **one condition** (focus on one requirement)
- Never miss the **null value test** for any parameter
- Never miss any **negative value tests** for database connection or query tests
- Never miss trying to **access an object** or **record that does not exist**, because these are the places where exceptions get triggered



# HOW TO SPEED-UP

- Let a **senior create** a sample test for every function
- For a few tests, senior and junior developers must do **pair test automation**
- Then **junior** takes care of **rest of the unit tests** for automation
- **Senior reviews** the unit test code
- Lead runs all tests **every day**




**build verification test (BVT)**

**See Also:** regression testing, smoke test

A set of automated tests which validates the integrity of each new build and verifies its key/core

functionality, stability and testability. It is an industry practice when a high frequency of build

releases occurs (e.g., Agile projects) and it is run on every new build before the build is released

for further testing.

# CHALLENGES

- White box **testers** must be **developers**
- Companies use them to **do core development** work and ignore writing white box tests
- White box tests need **tools** and sometimes they are **expensive**
- Main challenge comes in **not documenting design**. Since design is the basis, and if that is not present or incomplete, white box tests do not yield good results
- Project planning **never** considers the **time for white box testing**
- When changes are made to the programs being tested, the **changes are not documented** well and hence unit tests go out of gear
- Results can be seen only **after 1 or 2 quarters**. Many management teams do not have patience to wait for such a long time

# NEVER GIVE UP

- Teams **give up** white box tests **after a quarter** or so
  - This is because, they do not see visible returns
- Most of the times, **great systems** are brought **down** due to **missing white box tests** and not black box functionality
- It may **take years** to find out, one great **bug** – but it is essential for long term products
- If you are in product company, **be patient** and white box will pay back



# SUMMARY: VIDEO

WHAT IS WHITE BOX TESTING

https://www.youtube.com/watch?v=3bJcvBLJViQ