

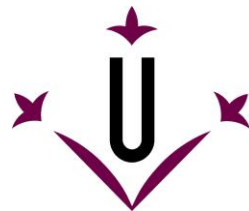
Quality Management and Improvement

Chapter 4

Testing and Metrics

2020/21

Juan Enrique Garrido Navarro
juanenrique.garrido@udl.cat



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

- Software Testing
- Metrics

- **Software Testing**
 - Overview
 - Activities
 - Strategies and Classification
 - Questions
 - Testing based on checklists and partitions
 - Additional types of testing
- **Metrics**

- **Software Testing**
 - **Overview**
 - Activities
 - Strategies and Classification
 - Questions
 - Testing based on checklists and partitions
 - Additional types of testing
- Metrics

- *Formal process carried out by specialized testing team in which a software unit, several integrated software units or an entire software package are examined by running the programs on a computer.*
- All the associated test are performed according to **approved test procedures** on **approved test cases**.

OBJECTIVES

Testing
Objectives

OBJECTIVES

Testing
Objectives

```
graph TD; A[Testing Objectives] --> B[To demonstrate quality or proper behaviour]; A --> C[To detect and fix problems];
```

To demonstrate quality
or proper behaviour

To detect and
fix problems

- ***DIRECT OBJECTIVES:***

- **INDIRECT OBJECTIVES:**

OBJECTIVES

- ***DIRECT OBJECTIVES:***

To **identify** and **reveal** as many **errors** as possible.

To **bring** the tested software, after correction of the required errors and retesting, to an **acceptable level of quality**.

To perform the required tests **efficiently** and **effectively**, within budgetary and scheduling limitations.

- ***INDIRECT OBJECTIVES:***

OBJECTIVES

- ***DIRECT OBJECTIVES:***

To **identify** and **reveal** as many **errors** as possible.

To **bring** the tested software, after correction of the required errors and retesting, to an **acceptable level of quality**.

To perform the required tests **efficiently** and **effectively**, within budgetary and scheduling limitations.

- ***INDIRECT OBJECTIVES:***

To **compile** a record of software **errors** for use in error prevention.

OBJECTIVES

“If your goal is to show the absence of errors you won’t discover many.

If your goal is to show the presence of errors, you will discover a large percentage of them”

[Myers, 1976]

Myers, G.J.. (1976): “Software Reliability – Principles and Practices”. John Wiley & Sons, New York.

- Basic idea of testing **involves** ...

- Basic idea of testing **involves** ... the execution of software and the observation of its behaviour or outcome.

- Basic idea of testing **involves** ... the execution of software and the observation of its behaviour or outcome.
- If a failure is observed, the execution record is analysed to locate and fix the fault/s that caused the failure.

- Basic idea of testing **involves** ... the execution of software and the observation of its behaviour or outcome.
- If a failure is observed, the execution record is analysed to locate and fix the fault/s that caused the failure.
- **Purpose** of software testing is to ensure that the software systems would work as expected when they are used by their target customers and users.

- Basic idea of testing **involves** ... the execution of software and the observation of its behaviour or outcome.
- If a failure is observed, the execution record is analysed to locate and fix the fault/s that caused the failure.
- **Purpose** of software testing is to ensure that the software systems would work as expected when they are used by their target customers and users.
- The most natural way to show this fulfilment of expectations is to demonstrate their operation through some “dry-runs” or controlled experimentation in laboratory settings before the products are released or delivered.

- Basic idea of testing **involves** ... the execution of software and the observation of its behaviour or outcome.
- If a failure is observed, the execution record is analysed to locate and fix the fault/s that caused the failure.
- **Purpose** of software testing is to ensure that the software systems would work as expected when they are used by their target customers and users.
- The most natural way to show this fulfilment of expectations is to demonstrate their operation through some “dry-runs” or controlled experimentation in laboratory settings before the products are released or delivered. In the case of software products, such **controlled experimentation through program execution** is generally called ***testing***.

Testing as a part of SQA

Testing as a part of SQA

- Testing is an integral part of the SQA activities.

Testing as a part of SQA

- Testing is an integral part of the SQA activities.
- Testing falls into the category of defect reduction alternatives that also include inspection and various static and dynamic analyses.
- Unlike inspection, testing **detects faults indirectly** through the execution of software instead of critical examination used in inspection.
- However, testing and inspection often finds different kinds of problems, and may be more effective under different circumstances.

Testing as a part of SQA

- Testing is an integral part of the SQA activities.
- Testing falls into the category of defect reduction alternatives that also include inspection and various static and dynamic analyses.
- Unlike inspection, testing **detects faults indirectly** through the execution of software instead of critical examination used in inspection.
- However, testing and inspection often finds different kinds of problems, and may be more effective under different circumstances.
- **Therefore, inspection and testing should be viewed more as complementary QA alternatives instead of competing ones.**

- **Software Testing**
 - Overview
 - **Activities**
 - Strategies and Classification
 - Questions
 - Testing based on checklists and partitions
 - Additional types of testing
- Metrics

Major Activities

- The basic concepts of testing can be best described in the context of the major activities involved in testing.

Major Activities

- The basic concepts of testing can be best described in the context of the major activities involved in testing.

Test planning and preparation

Set the goals for testing, select an overall testing strategy, and prepare specific test cases and the general test procedure.

Major Activities

- The basic concepts of testing can be best described in the context of the major activities involved in testing.

Test planning and preparation

Set the goals for testing, select an overall testing strategy, and prepare specific test cases and the general test procedure.

Test execution and related activities

Include related observation and measurement of product behaviour.

Major Activities

- The basic concepts of testing can be best described in the context of the major activities involved in testing.

Test planning and preparation

Set the goals for testing, select an overall testing strategy, and prepare specific test cases and the general test procedure.

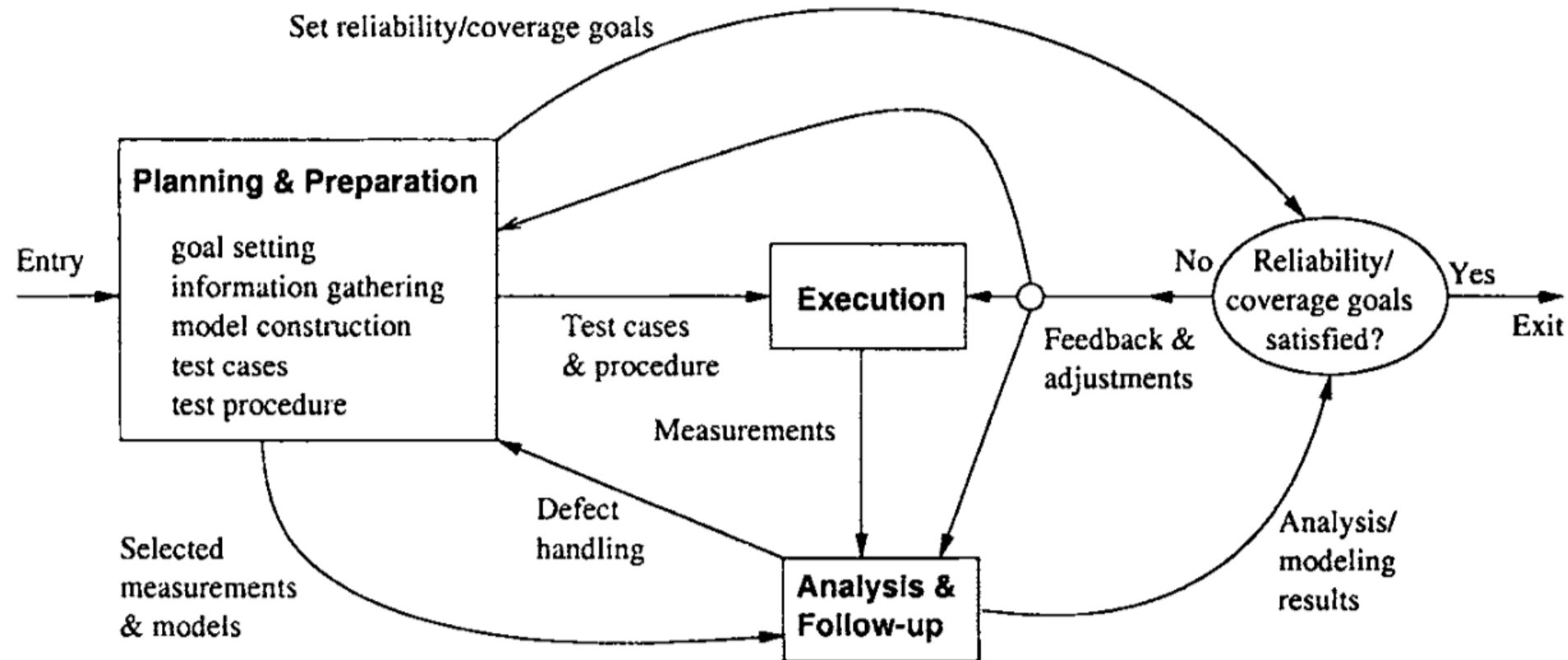
Test execution and related activities

Include related observation and measurement of product behaviour.

Analysis and follow-up

Include result checking and analysis to determine if a failure has been observed, and if so, follow-up activities are initiated and monitored to ensure removal of the underlying causes, or faults, that led to the observed failures in the first place.

Major Activities - Scheme



Activities - Preparation

- **Test planning and preparation** -> the most important activity in the generic testing process for systematic testing based on formal models. Most of the key decisions about testing are made during this stage.

Activities - Preparation

- **Test planning and preparation** -> the most important activity in the generic testing process for systematic testing based on formal models. Most of the key decisions about testing are made during this stage.
- The high-level task for test planning is to set goals and to determine a general testing strategy.

- We set an **overall testing strategy** by making the following decisions:
 - Overall objectives and goals: which can be refined into specific goals for specific testing. Some specific goals include *reliability* for usage-based statistical testing or *coverage* for various traditional testing techniques.
 - Objects to be tested and the specific focus: functional testing views the software product as a black-box and focuses on testing the external functional behaviour; while structural testing views the software product or component as a (transparent) white-box and focuses on testing the internal implementation details.

Activities - Preparation

- In addition, we need to define and distinguish the test cases and the test runs:

Activities - Preparation

- In addition, we need to define and distinguish the test cases and the test runs:
 - A **test case** *is a collection of entities and related information that allows a test to be executed or a test run to be performed.*
 - A **test run** *is a dynamic unit of specific test activities in the overall testing sequence on a selected testing object.*

Activities - Preparation

- In addition, we need to define and distinguish the test cases and the test runs:
 - A **test case** *is a collection of entities and related information that allows a test to be executed or a test run to be performed.*
 - A **test run** *is a dynamic unit of specific test activities in the overall testing sequence on a selected testing object.*
- Each time a static test case is invoked, we have an individual dynamic test run.
- Therefore, each test case can correspond to multiple test runs.

Activities - Preparation

- The collection of individual test cases that will be run in a test sequence until some stopping criteria are satisfied is called a **test suite**.
- Test suite preparation involves the construction and allocation of individual test cases in some systematic way based on the specific testing techniques used.
- In general, all the test cases should form an integrated suite, regardless of their origins, how they are derived, and what models are used to derive them.

Activities - Preparation

- To prepare individual test cases and the overall test suite, the test procedure also needs to be prepared for effective testing.

Activities - Preparation

- To prepare individual test cases and the overall test suite, the test procedure also needs to be prepared for effective testing.
- Concerns to consider:

Activities - Preparation

- To prepare individual test cases and the overall test suite, the test procedure also needs to be prepared for effective testing.
- Concerns to consider:
 - **Dependencies** among individual test cases. Some test cases can only be run after others because one is used to set up another.
 - **Defect detection** related sequencing. Many problems can only be effectively detected after others have been discovered and fixed.
 - **Sequences to avoid accidents.** For some systems, possibly severe problems and damages may incur during testing if certain areas were not checked through related test runs prior to the current test run.
 - **Problem diagnosis** related sequencing. Some execution problems observed during testing may involve complicated scenarios and many possible sources of problems.
 - **Natural grouping** of test cases, such as by functional and structural areas or by usage frequencies, can also be used for test sequencing and to manage parallel testing.

Activities - *Execution*

- Steps in test execution:

Activities - *Execution*

- Steps in test execution:
 1. Allocating test time and resources.

Activities - *Execution*

- Steps in test execution:
 1. Allocating test time and resources.
 2. Invoking and running tests, and collecting execution information and measurements.

Activities - *Execution*

- Steps in test execution:
 1. Allocating test time and resources.
 2. Invoking and running tests, and collecting execution information and measurements.
 3. Checking testing results and identifying system failures.

Activities - Analysis and Follow-up

- The third group of major test activities is analysis and follow-up after test execution.
- The measurement data collected during test execution, together with other data about the testing and the overall environment, form the data input to these analyses, which, in turn, provide valuable feedback to test execution and other testing and development activities.
- **Follow-up** *includes defect fixing and making other management decisions, such as product release and transition from one development phase or sub-phase to another.*
- Analysis of individual test runs includes result checking and failure identification.

Activities - Analysis and Follow-up – Failure identification

Activities - Analysis and Follow-up – Failure identification

- When failures are identified, additional analyses are normally performed by developers to diagnose the problem and locate the faults that caused the failures for defect removal.
- This activity may involve the following steps:
 - **Understanding** the problem.
 - Being able to **recreate** the same problem scenario and observe the same problem.
 - **Problem diagnosis** to examine what kind of problem it is, where, when, and possible causes.
 - **Fault locating**, to identify the exact location/s of fault/s.
 - **Defect fixing**, to fix the located fault(s) by adding, removing, or correcting certain parts of the code.

Activities - Test Automation

- Test automation *aims to automate some manual tasks with the use of some software tools.*
- The demand for test automation is strong.
- Purely manual testing from start to finish can be tedious and error-prone.
- Some level of automation for individual activities is possible ... not the whole process.

Activities - Test Automation

- It can be supported by various commercial tools or tools developed within large organizations.
- The **key in the use of test automation** to relieve people of tedious and repetitive tasks and to improve overall testing productivity is to first examine what is possible, feasible, and economical, and then to set the right expectations and goals.

Activities - Test Automation

- It can be supported by various commercial tools or tools developed within large organizations.
- The **key in the use of test automation** to relieve people of tedious and repetitive tasks and to improve overall testing productivity is to first examine what is possible, feasible, and economical, and then to set the right expectations and goals.



Examples...



<http://www.testingtools.com/test-automation/>

- **Software Testing**
 - Overview
 - Activities
 - **Strategies and Classification**
 - Questions
 - Testing based on checklists and partitions
 - Additional types of testing
- Metrics

STRATEGIES

SOFTWARE TESTING STRATEGIES

- To test the software in its entirety: **big bang testing**
- To test in an incremental way (**incremental testing**): *Example*
 - To test the software piecemeal, in modules, as they are completed: **unit tests**.
 - To test groups of tested modules integrated with newly completed modules: **integration tests**.
 - Once **integration tests** are completed, the entire package is tested as a whole: **system test**.

CLASSIFICATIONS

TESTING CLASSESOpposing
concepts**BLACK BOX**
(FUNCTIONALITY)
TESTING

Identifies bugs only according to software malfunctioning as they are revealed in its erroneous outputs.

Disregards the internal path of calculations and processing performed.

WHITE BOX
(STRUCTURAL)
TESTING

Examines internal calculations paths in order to identify bugs.

In many cases, both concepts are applicable, although for some SQA requirements only one class of test is suitable.

Due to cost considerations . . . Most of the testing carried out currently is black box testing, which is relatively less costly.

BLACK BOX

- *It allows us to perform output correctness test.*
 - Apply the concept of **test cases**.
 - Improved choice of test cases can be achieved by the efficient use of **equivalence classes (EC) partitioning**. It is a black box method aimed at increasing the efficiency of testing.
 - *An EC is a set of input variable values that produce the same results or that are processed identically.*
 - An EC that contains only valid states is defined as a “valid EC”, whereas an EC that contains only invalid states is defined as an “invalid EC”.
 - EC boundaries defined by a numeric or alphabetic value.
 - If a program has as input several variables, valid and invalid ECs should be defined for each of these variables.
 - Test cases are defined so that each valid EC and each invalid EC are included in at least one test case.
 - Test cases are defined separately for valid and invalid ECs.

BLACK BOX

- *It allows us to perform output correctness test.*
 - In comparison with random sample of test cases . . .
 - . . . Equivalence classes save resources as eliminate duplication of test cases defined for each EC.

BLACK BOX – Output Correctness test - *Example*

Reservation of Paddle courts

Monday , Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

Client (Visitor, Member)	V	V	V	V	M	M	M	M
Hour	8:00 – 12:00	12:01 – 14:00	14:01 – 20:00	20:01 - 23:59	8:00 – 12:00	12:01 – 14:00	14:01 – 20:00	20:01 - 23:59
Price	20	25	20	30	7	10	7	15

BLACK BOX – Output Correctness test - *Example*

Reservation of Paddle courts

Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

Client (Visitor, Member)	V	V	V	V	M	M	M	M
Hour	8:00 – 12:00	12:01 – 14:00	14:01 – 20:00	20:01 - 23:59	8:00 – 12:00	12:01 – 14:00	14:01 – 20:00	20:01 - 23:59
Price	20	25	20	30	7	10	7	15

Equivalence Classes

BLACK BOX – Output Correctness test - *Example*

Reservation of Paddle courts

Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday								
Client (Visitor, Member)	V	V	V	V	M	M	M	M
Hour	8:00 – 12:00	12:01 – 14:00	14:01 – 20:00	20:01 - 23:59	8:00 – 12:00	12:01 – 14:00	14:01 – 20:00	20:01 - 23:59
Price	20	25	20	30	7	10	7	15

Equivalence Classes

Variable	Valid equivalence classes	Values for valid ECs	Boundary values	Invalid equivalence classes	Representing values for invalid ECs
Day	(1) Mon, Tue, Wed, Thurs, Fri, Sat, Sun	Mon		(1) Not a day	Hi
Client	(1) V	V		(1) Not V or M	Z
	(2) M	M			
Hour	(1) 8:00 -12:00	9:00	8:00 and 12:00	(1) < 8:00 (2) Not time	4:55 Blue
	(2) 12:01 – 14:00	13:00	12:01 and 14:00		
	(3) 14:01 – 20:00	15:00	14:01 and 20:00		
	(4) 20:01 – 23:59	21:00	20:01 and 23:59		

BLACK BOX – Output Correctness test - *Example*

Reservation of Paddle courts

Test cases

Type	Number	Day	Client	Hour	Test case result
Valid ECs	1	Mon	V	9:00	20 €
	2	Mon	M	13:00	10 €
	3	Mon	M	15:00	7 €
	4	Mon	M	21:00	15 €
	5	Mon	V	8:00	20 €
	6	Mon	M	12:00	7 €
	7	Mon	M	12:01	10 €
	8	Mon	M	14:00	10 €
	9	Mon	M	14:01	7 €
	10	Mon	M	20:00	7 €
	11	Mon	M	20:01	15 €
	12	Mon	M	23:59	15 €
Invalid ECs	13	Hi	V	9:00	Invalid day
	14	Mon	Z	9:00	Invalid visitor
	15	Mon	V	4:55	Invalid hour
	16	Mon	V	Blue	Invalid hour

BLACK BOX

- Apart from output correctness tests, other black box testing classes:

Quality requirement factor	Test class
Correctness	Availability tests: reaction time, that is, the time needed to obtain the requested information.
Reliability	Reliability tests.
Efficiency	Stress tests (load or durability tests): e.g. under maximal operational load.
Integrity	Software system security tests: e.g. detection of unauthorized access.
Usability	Training and operational usability tests.

WHITE BOX

- White box testing is based on checking the data processing for each case . . .
- ... immediately raises the question of coverage of a vast number of possible processing paths and the multitudes of lines of code.

WHITE BOX

Path coverage

To cover all the possible paths, where coverage is measured by percentage of paths covered.

Motivated by the aspiration to achieve complete coverage of a program by testing all its possible paths.

Impractical in most cases because the vast resources required for its performance.

Paths? -> IF-THEN-ELSE, WHILE ... conditional statements

Example

10 conditional statements, each allowing for only two options.

The simple model contains 1024 different paths.

A full coverage ... one test case for each possible path ... 1024 test cases.

WHITE BOX

Path coverage

To cover all the possible paths, where coverage is measured by percentage of paths covered.

Motivated by the aspiration to achieve complete coverage of a program by testing all its possible paths.

Impractical in most cases because the vast resources required for its performance.

Paths? -> IF-THEN-ELSE, WHILE ... conditional statements

Example

10 conditional statements, each allowing for only two options.

The simple model contains 1024 different paths.

A full coverage ... one test case for each possible path ... 1024 test cases.

100 modules of similar complexity



102.400 test cases

WHITE BOX

Path coverage

This situation has encourage development of an alternative . . .



Example

10 conditional statements, each allowing for only two options.

The simple model contains 1024 different paths.

A full coverage ... one test case for each possible path ... 1024 test cases.

100 modules of similar complexity

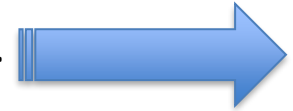
102.400 test cases

WHITE BOX

Path coverage

This situation has encourage development of an alternative . . .

lines



Example

10 conditional statements, each allowing for only two options.

The simple model contains 1024 different paths.

A full coverage ... one test case for each possible path ... 1024 test cases.

100 modules of similar complexity

102.400 test cases

WHITE BOX

Line coverage

To cover all program code lines, where coverage is measured by percentage of lines covered.

Requires that, for full line coverage, every line of code may be executed at least once during the process of testing.

WHITE BOX

Line coverage

To cover all program code lines, where coverage is measured by percentage of lines covered.

Requires that, for full line coverage, every line of code may be executed at least once during the process of testing.

Flow chart and flow graph (program) Can be helpful



WHITE BOX

Line coverage

To cover all program code lines, where coverage is measured by percentage of lines covered.

Requires that, for full line coverage, every line of code may be executed at least once during the process of testing.

Flow chart and flow graph (program) . . .

. . . Can be helpful



WHITE BOX

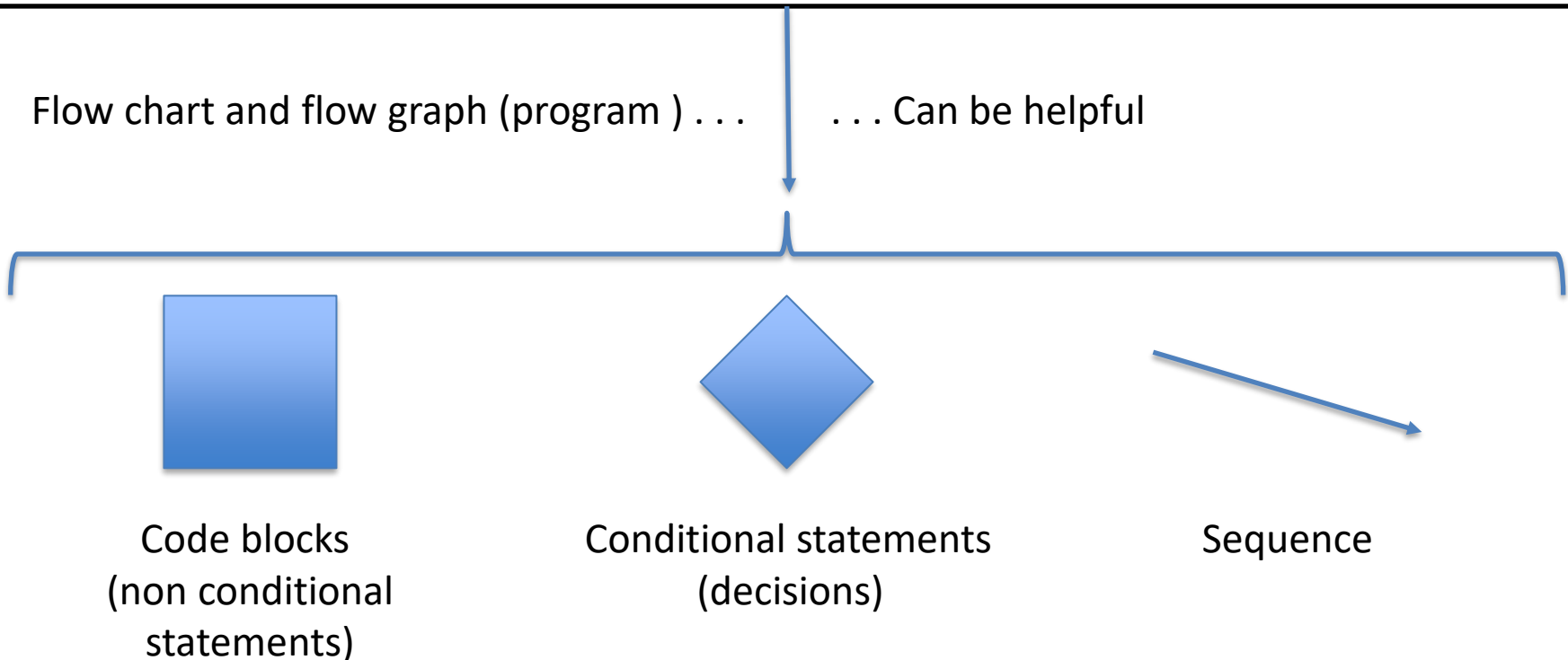
Line coverage

To cover all program code lines, where coverage is measured by percentage of lines covered.

Requires that, for full line coverage, every line of code may be executed at least once during the process of testing.

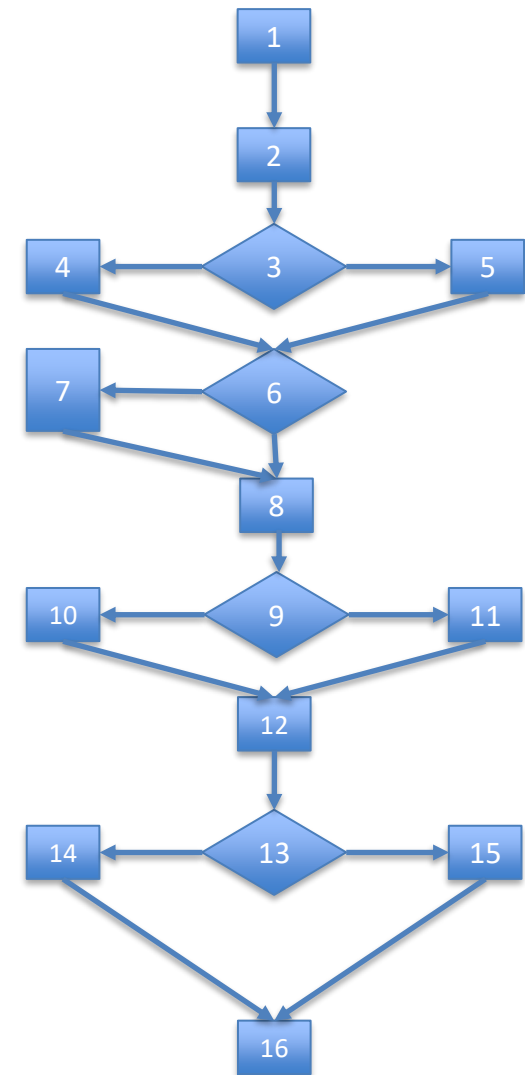
Flow chart and flow graph (program) . . .

. . . Can be helpful



WHITE BOX - Example

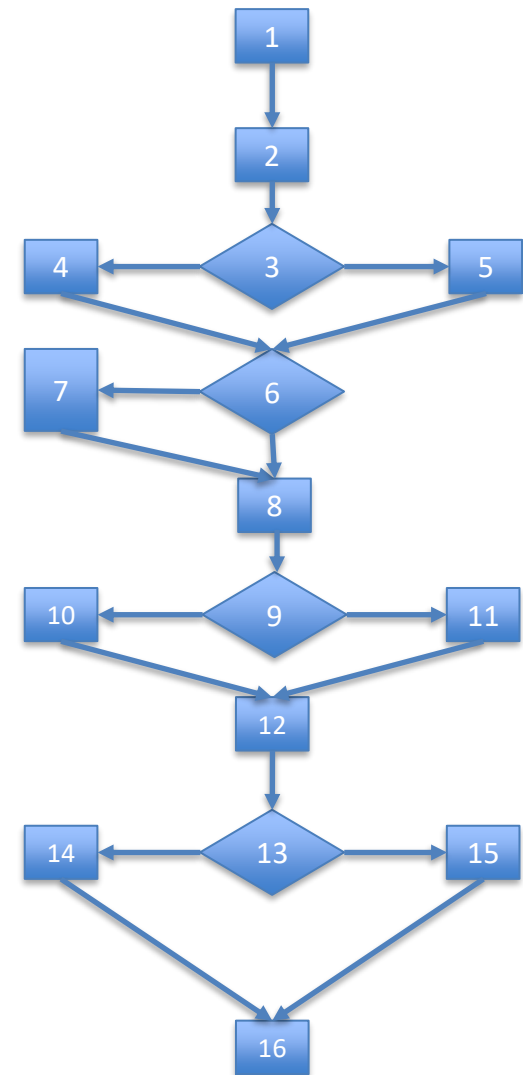
```
(1)  print("Start");
(2)  num = obtain_num();
(3)  if((num % 2) == 0)
(4)      total_p += num;
(5)  else
(6)      total_i += num;
(7)
(8)  if(total_p + total_i > max)
(9)      print("Full");
(10)
(11) let = obtain_let();
(12)
(13) if(is_capital(let))
(14)     insert_cap(let);
(15) else
(16)     insert_low(let);
(17)
(18) sym = obtain_sym();
(19)
(20) if(is_separator(sym))
(21)     insert_sep(sym);
(22) else
(23)     insert_other(sym);
(24)
(25) print("Stop");
```



WHITE BOX - Example

```
(1) print("Start");
(2) num = obtain_num();
(3) if((num % 2) == 0)
(4)     total_p += num;
(5) else
(6)     total_i += num;
(7) if(total_p + total_i > max)
(8)     print("Full");
(9) let = obtain_let();
(10) if(is_capital(let))
(11)     insert_cap(let);
(12) else
(13)     insert_low(let);
(14) sym = obtain_sym();
(15) if(is_separator(sym))
(16)     insert_sep(sym);
(17) else
(18)     insert_other(sym);
(19) print("Stop");
```

How many test cases are required by full path coverage? All the possible paths should be executed at least once.

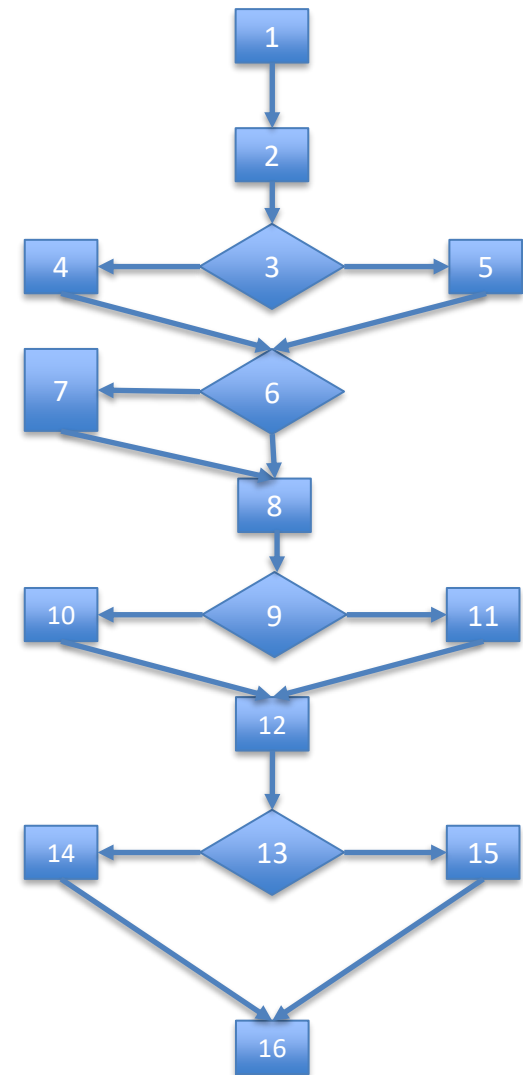


WHITE BOX - Example

```
(1)  print("Start");
(2)  num = obtain_num();
(3)  if((num % 2) == 0)
(4)      total_p += num;
(5)  else
(6)      total_i += num;
(7)
(8)  if(total_p + total_i > max)
(9)      print("Full");
(10)
(11) let = obtain_let();
(12)
(13) if(is_capital(let))
(14)     insert_cap(let);
(15) else
(16)     insert_low(let);
(17)
(18) sym = obtain_sym();
(19)
(20) if(is_separator(sym))
(21)     insert_sep(sym);
(22) else
(23)     insert_other(sym);
(24)
(25) print("Stop");
```

How many test cases are required by full path coverage? All the possible paths should be executed at least once.

We can know the minimum number of paths to be inspected to test the system by full line coverage.



WHITE BOX - Example

```

(1)  print("Start");
(2)  num = obtain_num();
(3)  if((num % 2) == 0)
(4)      total_p += num;
(5)  else
(6)      total_i += num;
(7)
(8)  if(total_p + total_i > max)
(9)      print("Full");
(10)
(11) let = obtain_let();
(12)
(13) if(is_capital(let))
(14)     insert_cap(let);
(15) else
(16)     insert_low(let);
(17)
(18) sym = obtain_sym();
(19)
(20) if(is_separator(sym))
(21)     insert_sep(sym);
(22) else
(23)     insert_other(sym);
(24)
(25) print("Stop");

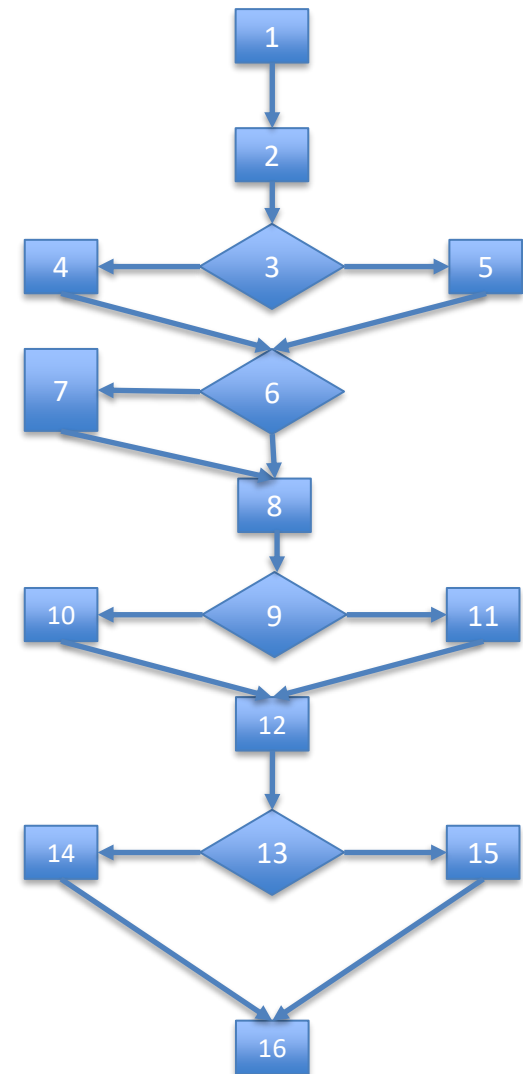
```

How many test cases are required by full path coverage? All the possible paths should be executed at least once.

We can know the minimum number of paths to be inspected to test the system by full line coverage.

Path 1: 1,2,3,4,6,7,8,9,10,12,13,14,16

Path 2: 1,2,3,5,6,8,9,11,12,13,15,16



WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- It **measures the complexity** of a program.

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- McCabe's cyclomatic complexity metrics provides a support for the basis path testing strategy. It serves to give an upper limit to the number of test cases needed for full coverage.

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- It **measures** the **complexity** of a program.
- At the same time, it **determines** the maximum number of independent paths needed to achieve full line coverage of the program.

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- It **measures** the **complexity** of a program.
- At the same time, it **determines** the maximum number of independent paths needed to achieve full line coverage of the program.
- Based on . . .

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- It **measures** the **complexity** of a program.
- At the same time, it **determines** the maximum number of independent paths needed to achieve full line coverage of the program.
- Based on . . . Graphs theory . . . It is calculated according to the program characteristics as captured by its program flow graph.

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- It **measures** the **complexity** of a program.
- At the same time, it **determines** the maximum number of independent paths needed to achieve full line coverage of the program.
- Based on . . . Graphs theory . . . It is calculated according to the program characteristics as captured by its program flow graph.

Any path on the program flow graph that includes at least one edge that is not included in any of the former independent paths.

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- The cyclomatic metric **$V(G)$** is expressed in three different ways:

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- The cyclomatic metric **$V(G)$** is expressed in three different ways:
 - $V(G) = R$
 - $V(G) = E - N + 2$
 - $V(G) = P - 1$

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- The cyclomatic metric **$V(G)$** is expressed in three different ways:

- $V(G) = R$
- $V(G) = E - N + 2$
- $V(G) = P - 1$

Number of regions in the program flow graph:

- Any enclosed area in the graph.
- The area around the graph not enclosed. An additional one.

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- The cyclomatic metric **$V(G)$** is expressed in three different ways:

- $V(G) = R$
- $V(G) = E - N + 2$
- $V(G) = P - 1$

Number of regions in the program flow graph:

- Any enclosed area in the graph.
- The area around the graph not enclosed. An additional one.

Number of edges in the program flow graph.

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- The cyclomatic metric $V(G)$ is expressed in three different ways:

- $V(G) = R$
- $V(G) = E - N + 2$
- $V(G) = P - 1$

Number of regions in the program flow graph:

- Any enclosed area in the graph.
- The area around the graph not enclosed. An additional one.

Number of nodes in the program flow graph.

Number of edges in the program flow graph.

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- The cyclomatic metric **$V(G)$** is expressed in three different ways:

- $V(G) = R$
- $V(G) = E - N + 2$
- $V(G) = P - 1$

Number of regions in the program flow graph:

- Any enclosed area in the graph.
- The area around the graph not enclosed. An additional one.

Number of nodes in the program flow graph.

Number of decisions contained in the graph.

Number of edges in the program flow graph.

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- The cyclomatic metric **$V(G)$** is expressed in three different ways:
 - $V(G) = R$
 - $V(G) = E - N + 2$
 - $V(G) = P + 1$
- Example ($R = 5$, $E = 19$, $N = 16$, $P = 4$)
 - $V(G) = 5$
 - $V(G) = 19 - 16 + 2 = 5$
 - $V(G) = 4 + 1 = 5$

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- The cyclomatic metric **$V(G)$** is expressed in three different ways:
 - $V(G) = R$
 - $V(G) = E - N + 2$
 - $V(G) = P + 1$
- Example ($R = 5$, $E = 19$, $N = 16$, $P = 4$)
 - $V(G) = 5$
 - $V(G) = 19 - 16 + 2 = 5$
 - $V(G) = 4 + 1 = 5$
 - The maximum number of independent paths in the example is 5.

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- Example ($R = 5$, $E = 19$, $N = 16$, $P = 4$)
 - $V(G) = 5$
 - $V(G) = 19 - 16 + 2 = 5$
 - $V(G) = 4 + 1 = 5$
 - The maximum number of independent paths in the example is 5.

Path number	Path	Edges added	Number of edges added
1	1,2,3,4,6,8,9,10,12,13,14,16	1,2,3,4,6,8,9,10,12,13,14,16	12
2	1,2,3,5,6,8,9,10,12,13,14,16	5	1
3	1,2,3,5,6,7,8,9,10,12,13,14,16	7	1
4	1,2,3,5,6,8,9,11,12,13,14,16	11	1
5	1,2,3,5,6,8,9,10,12,13,15,16	15	1

McCabe, T.J. (1976): "A software complexity measure". IEEE Transactions on Software Engineering, 2(6), 308-320.

WHITE BOX – McCabe's cyclomatic complexity metrics [McCabe, 1976]

- Several studies of the relationship “cyclomatic complexity metrics and quality and testability” have been carried out.
- [Jones, 1996]:
 - Less than 5 -> simple and easy to understand.
 - 10 or less -> not too difficult.
 - 20 or more -> high complexity.
 - + 50 -> Untestable.

McCabe, T.J. (1976): “A software complexity measure”. IEEE Transactions on Software Engineering, 2(6), 308-320.

Jones, C. (1996): “ Applied Software Measurement – Assuring Productivity and Quality”. 2nd edn, McGraw-Hill, New York.

- **Software Testing**
 - Overview
 - Activities
 - Strategies and Classification
 - **Questions**
 - Testing based on checklists and partitions
 - Additional types of testing
- Metrics

- What artefacts are tested?
- What to test, and what kind of faults is found?
- When, or at what defect level, to stop testing?
- When can specific test activities be performed?

- **What artefacts are tested?**

- **What artefacts are tested?**

- The primary types of objects or software artefacts to be tested are software programs or code written in different programming languages.
- Program code is the focus of our testing effort and related testing techniques and activities.

- **What to test, and what kind of faults is found?**

- **What to test, and what kind of faults is found?**

- Black-box (or functional) testing verifies the correct handling of the functions provided or supported by the software, or whether the observed behaviour conforms to user expectations or product specifications.
- White-box (or structural) testing verifies the correct implementation of internal units, structures, and relations among them.
- When black-box testing is performed, failures related to specific functions can be observed, leading to corresponding faults being detected and removed. The emphasis is on reducing the chances of encountering functional problems by target customers.
- On the other hand, when white-box testing is performed, failures related to internal implementations can be observed, leading to corresponding faults being detected and removed.

- **When, or at what defect level, to stop testing?**

- For most of the testing situations, the answer depends on the completion of some pre-planned activities, coverage of certain entities, or whether a pre-set goal has been achieved.
- Without a formal assessment for decision making, decision to stop testing can usually be made in two general forms:

- *Resource-based criteria*, where decision is made based on resource consumptions. The most commonly used such stopping criteria are:

- “Stop when you run out of time.”
- “Stop when you run out of money.”

Such criteria are irresponsible, as far as product quality is concerned, although they may be employed if product schedule or cost are the dominant concerns for the product in question.

- *Activity-based criteria*, commonly in the form:
 - “Stop when you complete planned test activities.”

This criterion implicitly assumes the effectiveness of the test activities in ensuring the quality of the software product. However, this assumption could be questionable without strong historical evidence based on actual data from the project concerned.

- **When, or at what defect level, to stop testing?**
 - Because of these shortcomings, informal decisions without using formal assessments have very limited use in managing the testing process and activities for large software systems.
 - More formal ...

- **When, or at what defect level, to stop testing?**
 - **Usage-based testing (UBT):**
 - It uses reliability criterion whose basic idea is to set a reliability goal in the quality **planning** activity during product planning and requirement analysis, and later on to **compare** the reliability assessment based on testing data to see if this pre-set goal has been reached. If so, the product can be released. Otherwise, testing needs to continue and product release needs to be deferred.
 - One important implication of using this criterion for stopping testing is that the **reliability assessments should be close to what actual users would expect**, which requires that the testing right before product release resembles actual usages by target customers.

- **When, or at what defect level, to stop testing?**
 - **Coverage-based testing (CBT):**
 - It involves coverage of some specific entities, such as components, execution paths, statements, etc. The use of coverage criteria is associated with defining appropriate coverage for different testing techniques, linking what was tested with what was covered in some formal assessments.

- **When, or at what defect level, to stop testing?**
 - *Usage-based testing is generally applicable to the final stage of testing.*
 - For earlier sub-phases of testing, or for stopping criteria related to localized test activities, reliability definition based on customer usage scenarios and frequencies may not be meaningful.
 - For example, many of the internal components are never directly used by actual users, and some components associated with low usage frequencies may be critical for various situations. Under these situations, the use of reliability criterion may not be meaningful.

- **When, or at what defect level, to stop testing?**

Comparisson

- UBT views the objects of testing from a user's perspective and focuses on the usage scenarios, sequences, patterns, and associated frequencies or probabilities; while CBT views the objects from a developer's perspective and focuses covering functional or implementation units and related entities.
- **UBT use product reliability goals** as the exit criterion; and **CBT** using **coverage goals** - surrogates or approximations of reliability goals - as *the exit criterion*.

- **When, or at what defect level, to stop testing?**

Comparisson

- If the software is complete or nearly complete, then *the correctness-centered quality goals can be directly translated into reliability goals*, which, in turn, requires us to use usage-based testing.

- When, or at what defect level, to stop testing?

Comparisson

- If the software is complete or nearly complete, then the correctness-centered quality goals can be directly translated into reliability goals, which, in turn, requires us to use usage-based testing.
- Sometimes, *these quality goals can be translated indirectly into coverage goals*, which can be achieved by **black-box** testing for the whole system. However, if only individual units and pieces are available, we might choose to focus on the implementation details and perform coverage-based **white-box** testing.

- **When can specific test activities be performed?**

- Because testing is an execution-based QA activity, a prerequisite to actual testing is the existence of the implemented software units, components, or system to be tested, although **preparation** for testing can be carried out in **earlier** phases of software development.
- As a result, actual testing of large software systems is typically organized and divided into **various sub-phases** starting from the coding phase up to post-release product support.

- **Software Testing**
 - Overview
 - Activities
 - Strategies and Classification
 - Questions
 - **Testing based on checklists and partitions**
 - Additional types of testing
- Metrics

- The simplest form of testing is to **start running the software and make observations**, in the hope that it is easy to distinguish between **expected** and **unexpected** behaviour.

- The simplest form of testing is to **start running the software and make observations**, in the hope that it is easy to distinguish between **expected** and **unexpected** behaviour.
- We call these forms of simple and informal testing ad hoc testing.

- The simplest form of testing is to **start running the software and make observations**, in the hope that it is easy to distinguish between **expected** and **unexpected** behaviour.
- We call these forms of simple and informal testing ad hoc testing.
- When ad hoc testing is used repeatedly to test a software system, the testers then need to keep track of what has been done, in order **to avoid wasting** their time repeating the same tests.

- The simplest form of testing is to **start running the software and make observations**, in the hope that it is easy to distinguish between **expected** and **unexpected** behaviour.
- We call these forms of simple and informal testing ad hoc testing.
- When ad hoc testing is used repeatedly to test a software system, the testers then need to keep track of what has been done, in order **to avoid wasting** their time repeating the same tests.
- In addition, an informal “to-do” list is commonly used to track what needs to be done. ?????

- The simplest form of testing is to **start running the software and make observations**, in the hope that it is easy to distinguish between **expected** and **unexpected** behaviour.
- We call these forms of simple and informal testing ad hoc testing.
- When ad hoc testing is used repeatedly to test a software system, the testers then need to keep track of what has been done, in order **to avoid wasting** their time repeating the same tests.
- In addition, an informal “to-do” list is commonly used to track what needs to be done. **????? -> checklists**

- The simplest form of testing is to **start running the software and make observations**, in the hope that it is easy to distinguish between **expected** and **unexpected** behaviour.
- We call these forms of simple and informal testing ad hoc testing.
- When ad hoc testing is used repeatedly to test a software system, the testers then need to keep track of what has been done, in order **to avoid wasting** their time repeating the same tests.
- In addition, an informal “to-do” list is commonly used to track what needs to be done. **?????** -> **checklists** -> each item can be checked off when corresponding testing was performed ... until every item on the lists is checked off.

Examples

- Checklist based on product specifications with each major specification item as a checklist item, can be used to perform black-box testing.

Examples

- Checklist based on product specifications with each major specification item as a checklist item, can be used to perform black-box testing.
- Checklists of expected programming features that are supposed to be implemented in a software product, or coding standards that are supposed to be followed in implementation, are examples of white-box checklists.

Examples

- Checklist based on product specifications with each major specification item as a checklist item, can be used to perform black-box testing.
- Checklists of expected programming features that are supposed to be implemented in a software product, or coding standards that are supposed to be followed in implementation, are examples of white-box checklists.
- Checklists of various structures or features that cut through different elements, such as list of call-pairs, consumers-and-producers for certain resources, modules sharing some common data, etc.

Examples

- Checklist based on product specifications with each major specification item as a checklist item, can be used to perform black-box testing.
- Checklists of expected programming features that are supposed to be implemented in a software product, or coding standards that are supposed to be followed in implementation, are examples of white-box checklists.
- Checklists of various structures or features that cut through different elements, such as list of call-pairs, consumers-and-producers for certain resources, modules sharing some common data, etc.
- Checklists about certain properties.

Examples

- Checklist based on product specifications with each major specification item as a checklist item, can be used to perform black-box testing.
- Checklists of expected programming features that are supposed to be implemented in a software product, or coding standards that are supposed to be followed in implementation, are examples of white-box checklists.
- Checklists of various structures or features that cut through different elements, such as list of call-pairs, consumers-and-producers for certain resources, modules sharing some common data, etc.
- Checklists about certain properties.
- ...

- One of the important characteristics of these checklists is that **they are typically not very specific.**
- For example, a high-level functional checklist typically lists the major functions to be performed by a software product, but the list items are hardly detailed enough for testers to start a specific test run.

- One of the important characteristics of these checklists is that **they are typically not very specific.**
- For example, a high-level functional checklist typically lists the major functions to be performed by a software product, but the list items are hardly detailed enough for testers to start a specific test run.
- It usually involves experienced testers **setting up the system and testing environment** to execute specific test cases.
- In addition, repetition of the same test case in a later test run can only be guaranteed with this additional information about setup and environment, but *not deduced from the check list item itself.*

- This would lead to **difficulties** when we try to **rerun** the failing execution to recreate the failure scenario for problem diagnosis, or when we need to **re-verify** the problem fixes.
- Therefore, additional information needs to be kept for these purposes.

Partitions

- Some problems can be resolved if we can derive a special type of checklists, called ***partitions***, that can both achieve full coverage of some specifically defined entities or relationships and avoid overlaps.

Partitions

- Some problems can be resolved if we can derive a special type of checklists, called ***partitions***, that can both achieve full coverage of some specifically defined entities or relationships and avoid overlaps.
- Then, *partitions are a special subclass of checklists*.

Partitions

Types of partitions

Types of partitions

Partitions of some product entities, such as external functions (black-box view), system components (white-box view), etc. The definition of such partitions are typically a simple “member” relation in sets.

Partitions

Types of partitions

Partitions of some product entities, such as external functions (black-box view), system components (white-box view), etc. The definition of such partitions are typically a simple “member” relation in sets.

Partitions based on certain properties, relations, and related logical conditions.

Partitions

Types of partitions

Partitions of some product entities, such as external functions (black-box view), system components (white-box view), etc. The definition of such partitions are typically a simple “member” relation in sets.

Partitions based on certain properties, relations, and related logical conditions.

Combinations of the above basic types are also commonly used in decision making

- **Software Testing**
 - Overview
 - Activities
 - Strategies and Classification
 - Questions
 - Testing based on checklists and partitions
 - **Additional types of testing**
- Metrics

- **Diagnosis testing**
 - To recreate and diagnose the problems when problems are reported by customers during operational use.
- **Beta testing**
 - Controlled product release and operational use by limited customers.
 - It often directly precedes operational use or is carried out at the very beginning of it.
- **Acceptance testing**
 - It is attached to the end of system testing because it is typically performed right after system testing to determine if the product should be released.
- **Unit testing**
 - It tests a small software unit (e.g. function or procedure) at a time, which is typically performed by the individual programmer who implemented the unit.

- **Defect-based testing**
 - It is based on the discovered defects or potential defects.
- **Performance testing**
 - It focuses on the performance of the software system in realistic operational environments.
- **Stress testing**
 - It is a special form of performance testing, where software system performance under stress is tested. This type of testing is also closely related to **capacity testing**, where the maximal system capacity is assessed.
- **Usability testing**
 - It assesses the overall usability of software systems, particularly for those systems where user interfaces play an important role.
- ...

- Software Testing
 - Overview
 - Activities
 - Strategies and Classification
 - Questions
 - Testing based on checklists and partitions
 - Additional types of testing
- **Metrics**

- Measuring software . . .

- Measuring software . . .
- . . . concerned with **delivering a value for an attribute of a software component/system/process . . . comparing** with other and to the standards (applied by the organization) . . . We are able to obtain **conclusions about the quality of software.**

- Measuring software . . .
- . . . concerned with **delivering a value for an attribute of a software component/system/process . . . comparing** with other and to the standards (applied by the organization) . . . We are able to obtain **conclusions about the quality of software.**
- **Software metric**: *characteristic of a software system, system documentation or development process that can be objectively measured.*

- Measuring software . . .
- . . . concerned with **delivering a value for an attribute of a software component/system/process . . . comparing** with other and to the standards (applied by the organization) . . . We are able to obtain **conclusions about the quality of software.**
- **Software metric**: *characteristic of a software system, system documentation or development process that can be objectively measured.*



Control metrics

To support process management (e.g. time required to repair reported defects).

Predictor metrics

To help you predict characteristics of the software (e.g. cyclomatic complexity of a module). These are used to measure internal attributes of a sw system.

- In addition . . . metrics can be classified into three categories:

- In addition . . . metrics can be classified into three categories:

Product metrics:

Describe the characteristics of the product (e.g. size, complexity, design features, quality level, etc.).

Process metrics:

Can be used to improve software development and maintenance (e.g. effectiveness of the defect removal during development or the response to the fix process).

Project metrics:

Describe the project characteristics and execution (e.g. number of software developers, cost or schedule).

- In addition . . . metrics can be classified into three categories:

Product metrics:

Describe the characteristics of the product (e.g. size, complexity, design features, quality level, etc.).

Process metrics:

Can be used to improve software development and maintenance (e.g. effectiveness of the defect removal during development or the response to the fix process).

Project metrics:

Describe the project characteristics and execution (e.g. number of software developers, cost or schedule).

Software quality metrics are a subset of software metrics that focus on quality aspects of the product, process and project.

Metrics

- In addition . . . metrics can be classified into three categories:

Product metrics:

Describe the characteristics of the product (e.g. size, complexity, design features, quality level, etc.).

Process metrics:

Can be used to improve software development and maintenance (e.g. effectiveness of the defect removal during development or the response to the fix process).

Project metrics:

Describe the project characteristics and execution (e.g. number of software developers, cost or schedule).

Software quality metrics are a subset of software metrics that focus on quality aspects of the product, process and project.

End-product quality metrics

In-process quality metrics

Metrics

- In addition . . . metrics can be classified into three categories:

Product metrics:

Describe the characteristics of the product (e.g. size, complexity, design features, quality level, etc.).

Process metrics:

Can be used to improve software development and maintenance (e.g. effectiveness of the defect removal during development or the response to the fix process).

Project metrics:

Describe the project characteristics and execution (e.g. number of software developers, cost or schedule).

Software quality metrics are a subset of software metrics that focus on quality aspects of the product, process and project.

End-product quality metrics

In-process quality metrics

Note: we should view quality from the entire software life-cycle perspective . . . then . . . we should include metrics that measure the quality level of maintenance process.

- Product metrics are predictor metrics.

- Product metrics are predictor metrics.
- Classes:
 - **Dynamic metrics**: collected by measurements made of a program in execution. These metric can be collected during system testing or after the system has gone into use.
 - Help to assess the efficiency and reliability of a program.
 - **Static metrics**: collected by measurements made of representations of the system, such as the design, program or documentation.
 - Help to assess the complexity, understandability and maintainability of a software system or system components.
- These metrics are related to **different quality attributes**.

Metrics

Product Metrics – *Static Software Product Metrics*

Software metric	Description
Fan-in/Fan-out	<p><u>Fan-in</u> is a measure of the number of functions or methods that call another function or method (say X).</p> <p><u>Fan-out</u> is the number of functions that are called by function X.</p> <p>A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects.</p> <p>A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.</p>
Length of code	<p>This is a measure of the size of a program.</p> <p>Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be.</p> <p>Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.</p>

Metrics

Product Metrics – *Static Software Product Metrics*

Software metric	Description
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability.
Length of identifiers	This measure of the average length of identifiers (name for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

Metrics

Product Metrics – *Dynamic Software Product Metrics*

Software metric	Description
Defect Density Metric	<p>Defect Density is the number of defects which are detected and confirmed in a software product during a concrete time. (A <u>defect</u> is an anomaly in a product).</p> <p>It is usually expressed in thousand lines of code (KLOC). It is anything but simple. Many variations: count only executable lines, plus definitions, plus comments . . .</p>
Customer Problems Metric	<p>This metric measures the problems customers encounter when using a product.</p> <p>This is usually expressed in terms of problems per user month (<i>PUM</i>) = <i>(total problems that customers reported for a period of time) / (Total number of license-months of the sw during the period)</i>. <i>The number of license-months = (number of install licenses of the sw) x (Number of months in the calculation period)</i>.</p>
Customer Satisfaction Metric	<p>Customer satisfaction is (often) measured by customer survey data using the scale: very satisfied, satisfied, neutral, dissatisfied and very dissatisfied.</p> <p>Based on the customer survey data, several metrics can be constructed: percent of completely satisfied customer, percent of satisfied customer . . .</p>

- Aforementioned metrics are applicable to any program but more specific metrics have also been proposed.

- Aforementioned metrics are applicable to any program but more specific metrics have also been proposed.
- For example . . . object-oriented (OO).

Metrics

Product Metrics – *The CK Object-Oriented Metrics Suite [Chidamber, 1994]*

Software metric	Description
Weighted methods per class (WMC)	<p>This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value.</p> <p>The larger the value for this metric, the more complex the object class.</p> <p>Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.</p>
Depth of inheritance tree (DIT)	<p><i>This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses.</i></p> <p>The deeper the inheritance tree, the more complex the design.</p> <p>Many classes may have to be understood to understand the object classes at the leaves of the tree.</p>
Number of children (NOC)	<p>This is a measure of the number of immediate subclasses in a class.</p> <p>It measures the breadth of a class hierarchy, whereas DIT measures its depth.</p> <p>A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.</p>

Chidamber, S, Kemerer, C. (1994): "A Metrics Suite for Object-Oriented Design". IEEE Trans. On Software Eng, 20(6), 476-493.

Metrics

Product Metrics – *The CK Object-Oriented Metrics Suite [Chidamber, 1994]*

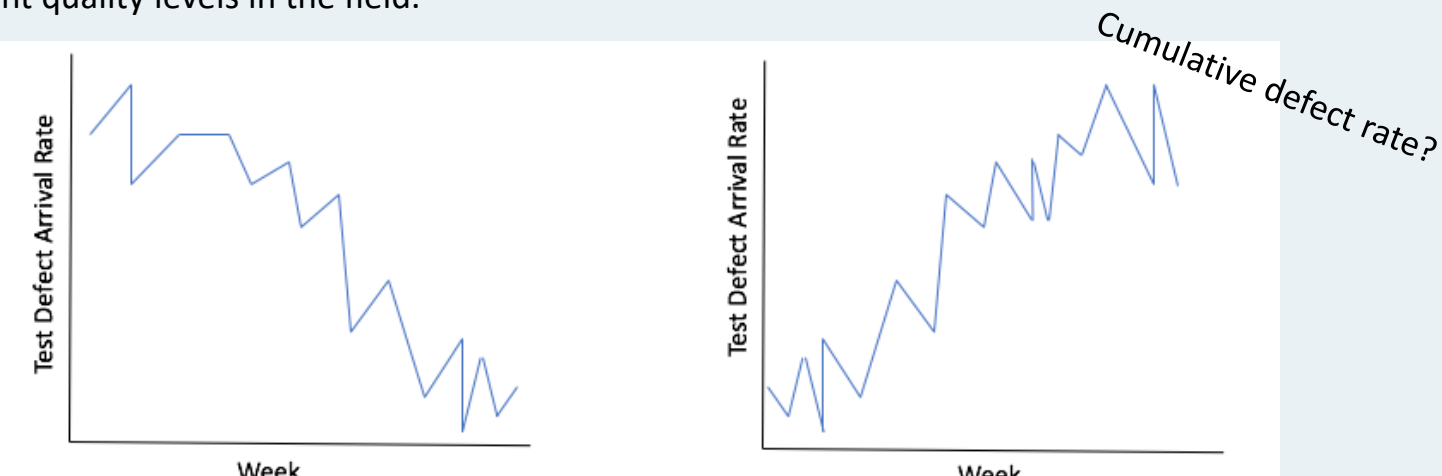
Software metric	Description
Coupling between object classes (CBO)	<p>Classes are coupled when methods in one class use methods or instance variables defined in a different class.</p> <p>CBO is measure of how much coupling exists.</p> <p>A high value of CBO means that classes are highly dependent and then, it is more likely that changing one class will affect other classes in the program.</p>
Response for a class (RFC)	<p>RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class.</p> <p>The higher value for RFC, the more complex a class and hence, the more likely is that it will include errors.</p>
Lack of cohesion in methods (LCOM)	<p>LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of methods pairs without shared attributes and the number of method pairs with shared attributes.</p> <p>The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional or useful information over and above that provided by other metrics.</p>

Chidamber, S, Kemerer, C. (1994): “ A Metrics Suite for Object-Oriented Design”. IEEE Trans. On Software Eng, 20(6), 476-493.

- In-process quality metrics play an important role . . . because our goal is to understand the programming process and to learn to engineer quality into the process.
- In-process quality metrics are less formal than end-product metrics and their practices differ among developers.

Metrics Metrics Tools

In-Process Metrics

Metric	Description
Defect Density During Machine Testing	Higher defect rate found during testing indicates that the software has experienced higher error injection during its development process, unless the higher testing defect rate is due to an extraordinary testing effort (e.g. additional or new testing approach that was deemed more effective in detecting defects).
Defect Arrival Pattern During Machine Testing	<p>The pattern of defect (or for that matter, time between failures) gives more information that overall defect density (as it is a summary indicator) during testing.</p> <p>Even with the same overall defect rate during testing, different patterns of defects arrivals indicate different quality levels in the field.</p> <div><p>The figure contains two line graphs side-by-side, both with 'Test Defect Arrival Rate' on the y-axis and 'Week' on the x-axis. The left graph shows a line that starts at a medium level, fluctuates, and then shows a general downward trend over about 10 weeks. The right graph shows a line that starts at a low level and shows a general upward trend over about 10 weeks. A diagonal line is drawn across the right graph from the bottom-left towards the top-right, with the text 'Cumulative defect rate?' written along it.</p></div>

Metrics

In-Process Metrics

Metric	Description
Defect Density During Machine Testing	Higher defect rate found during testing indicates that the software has experienced higher error injection during its development process, unless the higher testing defect rate is due to an extraordinary testing effort (e.g. additional or new testing approach that was deemed more effective in detecting defects).
Defect Arrival Pattern During Machine Testing	<p>The pattern of defect (or for that matter, time between failures) gives more information than overall defect density (as it is a summary indicator) during testing.</p> <p>Even with the same overall defect rate during testing, different patterns of defects arrivals indicate different quality levels in the field.</p> <p>The objective is always to look for defect arrivals that stabilize at a very low level, or times between failures that are far apart, before ending the testing effort and releasing the software to the field.</p>

Metrics

In-Process Metrics

Metric

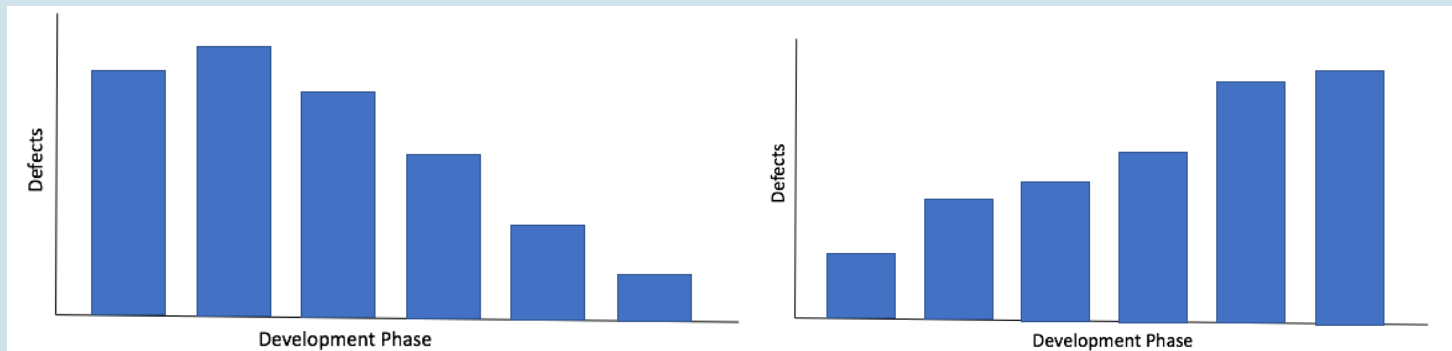
Description

Phase-Based Defect Removal Pattern

This is an extension of the test defect density metric. In addition to testing, it **requires the tracking of defects at all phases of the development cycle**, including reviews, code inspections and formal verifications before testing.

Because a large percentage of programming defects is related to design problems, conducting formal reviews or functional verifications to enhance the defect removal capacity of the process at the front end reduces error injection.

The pattern of phase-based defect removal **reflects the overall defect removal ability of the development process**.



Defect Removal Effectiveness (DRE)

$DRE = (\text{Defects removed during a development phase} / \text{Defects latent in the product}) \times 100\%$

The number of latent defects in the product at any phase is not known . . . it is approximated:

Defects remove during the phase + defect found later

- When various metrics are indicating a consistent negative message . . . The product will not be good enough to ship.
- When all metrics are positive, there is a good chance that the product quality will be positive.
- Problems? Questions?
 - When some of the metrics are positive and some are not.
- Then . . .
 - The point is that after going through all metrics and models, measurements and data, and qualitative indicators, **the team needs to step back and take a big-picture view, and subject all information to its experience base in order to come to a final analysis.**
 - Metrics aids decisions making, but do not replace it.

Bibliography

References

- **Mario G. Piattini, Félix O. García, Ignacio García.** Calidad de Sistemas de Información. 3rd edition. Ra-Ma Editorial. 2015.
- **Daniel Galin.** Software Quality: Concepts and Practice, First Edition. 2018 the IEEE Computer Society, Inc. Published 2018 by John Wiley & Sons, Inc.
- **Ian Sommerville.** Software Engineering. Ninth version. Addison-Wesley, 2011.
- **Jeff Tian.** Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement. IEEE Computer Society, 2005.
- **Daniel Galin.** Software Quality Assurance. From theory to implementation. Pearson Education Limited, 2004.
- **Stephen Kan.** Metrics and Models in Software Quality Engineering. 2nd Edition. Addison-Wesley, 2002.

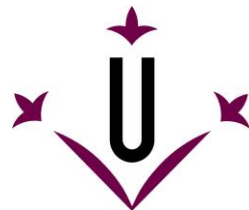
Quality Management and Improvement

Chapter 4

Testing and Metrics

2020/21

Juan Enrique Garrido Navarro
juanenrique.garrido@udl.cat



Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial