TQSO - Teste e Qualidade de Software (*Software Testing and Quality*)

Software Quality Concepts

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

It works!

- “When a developer says ‘it works’, he really means ‘it appears to fulfill some requirement to some degree.’” (One or more successes)

James Bach

- “When you hear someone say, ‘It works,’ immediately translate that into, ‘We haven't tried very hard to make it fail, and we haven't been running it very long or under very diverse conditions, but so far we haven't seen any failures, though we haven't been looking too closely, either.’ (Zero or more successes)

Jerry Weinberg

What is software (product) quality?

- Quality
  1. The degree to which a system, component, or process meets **specified** requirements.
  2. The degree to which a system, component, or process meets customer or user needs or **expectations**.

     [Source: IEEE Standard Glossary of Software Engineering Terminology (Std 610, 12-1990)]
  3. the totality of characteristics of an entity that bear on its ability to satisfy **stated** and **implied** needs

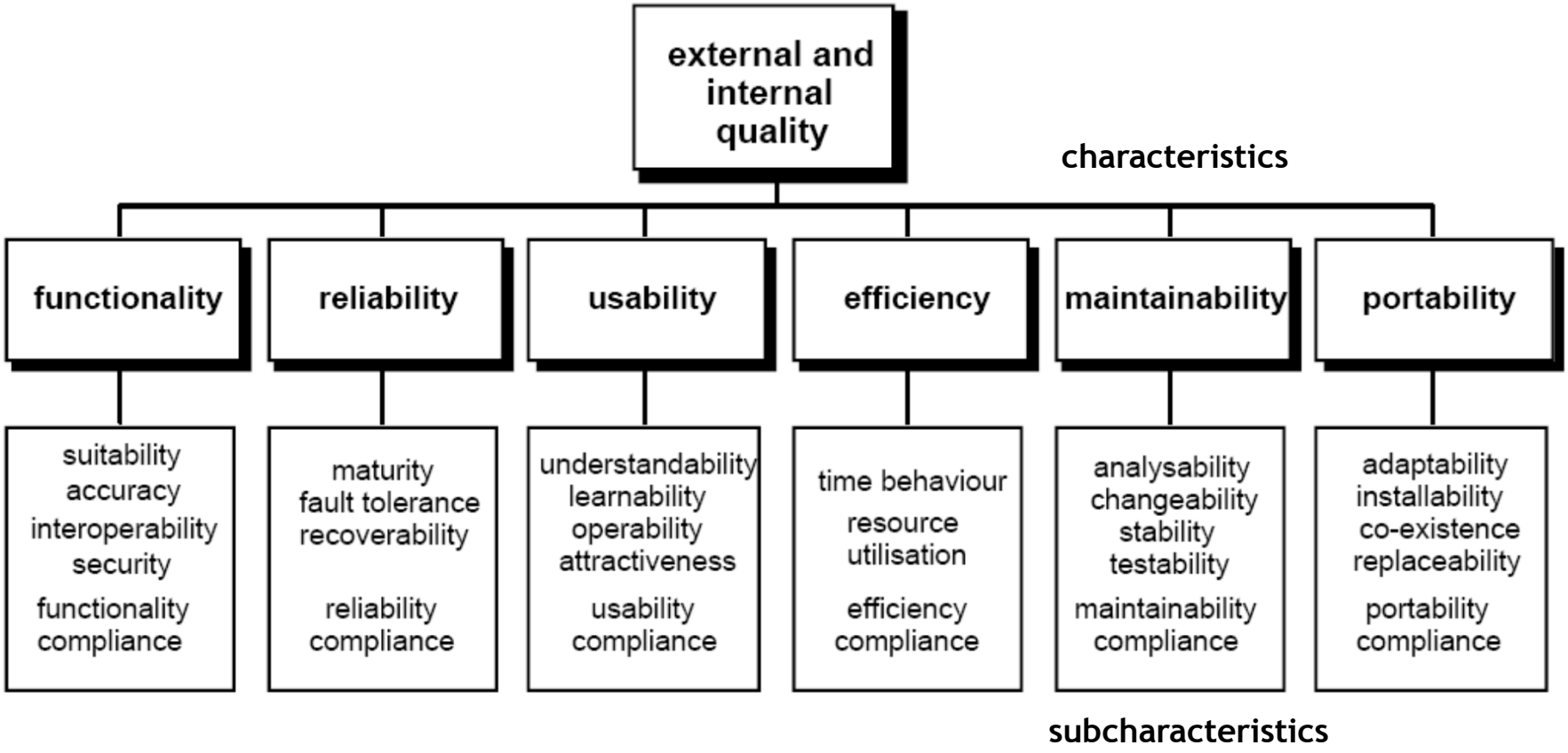
     [Source: ISO 8402:1994, Quality management and quality assurance – Vocabulary]
- Quality = **fitness for purpose**

It is not enough to meet specifications, because they are imperfect

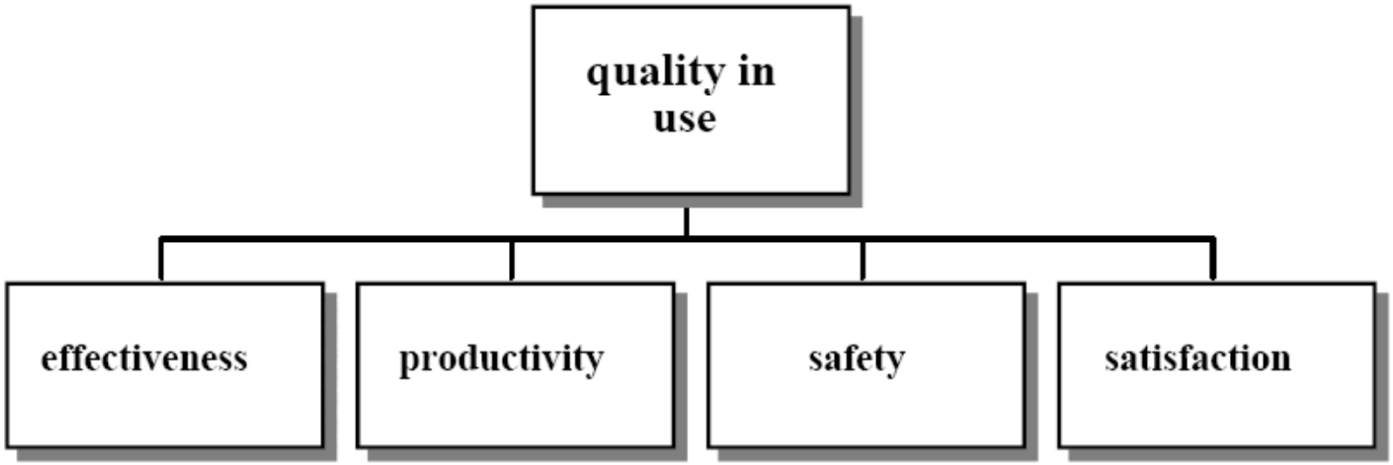
Software quality attributes according to the ISO/IEC 9126 standard

- Distinguishes three views of software product quality:
  - ***Internal Quality***
    - is the totality of characteristics of the software product from an internal view during its development or maintenance (e.g., code, architecture)
  - ***External Quality***
    - is the totality of characteristics of the software product from an external view during its execution
  - ***Quality in Use***
    - is the user's view of the quality of the software product when it is used in a specific environment and a specific context of use. It measures the extent to which users can achieve their goals in a particular environment, rather than measuring the properties of the software itself (e.g., usability)
- Ideally, the internal quality determines the external quality and external quality determines quality in use
- Standards:
  - ISO/IEC 9126-1:2001 Software engineering -- Product quality -- Part 1: Quality model
  - ISO/IEC TR 9126-2:2003 Software engineering -- Product quality -- Part 2: External metrics
  - ISO/IEC TR 9126-3:2003 Software engineering -- Product quality -- Part 3: Internal metrics
  - ISO/IEC TR 9126-4:2004 Software engineering -- Product quality -- Part 4: Quality in use metrics

ISO 9126-1:2001- Quality Model for External and Internal Quality



ISO 9126-1:2001- Quality model for quality in use



The capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specified context of use.

The capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use.

The capability of the software product to achieve acceptable levels of risk of harm to people, business, software, property or the environment in a specified context of use.

The capability of the software product to satisfy users in a specified context of use.

Quality attributes of critical systems

- Types of critical systems:
  - ***Safety-critical system***
    - a system whose failure may result in injury, loss of life or major environment damage, e.g., an insulin delivery system
  - ***Mission-critical system***
    - a system whose failure may result in the failure of some goal-directed activity, e.g., a navigational system for a space aircraft
  - ***Business-critical system***
    - a system whose failure may result in the failure of the business using the system, e.g., a customer account system in a bank, e.g., the web site of Amazon
- Sometimes also called high-integrity systems

Quality attributes of critical systems

- **Main quality attributes required for critical systems are usually grouped under the term "dependability"**
- **Dimensions of dependability:**
  - ***Reliability*** - The probability of failure-free system operation over a specified time in a given environment for a given purpose
  - ***Availability*** - The probability that a system, at a point in time, will be operational and able to deliver the requested services
    - It's possibly to have high availability with low reliability if failures are repaired quickly
  - ***Safety*** - The system's ability to operate, normally or abnormally, without danger of causing human injury or death and without damage to the system's environment
  - ***Security*** – The system's ability to protect itself from accidental or deliberate external attack
  - Several authors also include maintainability
- RAMS = Reliability, Availability, Maintainability and Safety

Type of defects
Orthogonal Defect Classification (ODC)

- **Function** - capability not implemented or implemented incorrectly
- **Interface** - problems in the way two separate pieces of logic communicate
- **Checking** - validate data/values before use (incorrectly)
- **Assignment** - initialization
- **Timing/serialization** - management of shared/real-time resources
- **Build/package/merge** - problems with the use of libraries
- **Documentation –** problems (e.g., inconsistencies, incompleteness) with documents
- **Algorithm –** such that does not imply changes in the architecture

[source: Metrics and Models in Software Quality Engineering]

Frequency of the main types of bugs?

IBM defect data:

Classificação ODC (*)	Description	Frequency
Algorithm	"execução incorrecta ou em falta que pode ser corrigida sem ser necessário introduzir alterações arquitecturais no software"	43.4 %
Assignment	"valores incorrectamente atribuídos ou não atribuídos"	22.0 %
Checking	"validação de dados incorrecta ou expressões condicionais incorrectas"	17.5 %
Function	"falha que afecta uma quantidade considerável de código e refere-se a uma capacidade do software que está em falta ou construída incorrectamente"	8.7 %
Interface	"interacção incorrecta entre módulos/componentes"	8.2 %

(*) Orthogonal Defect Classification (ODC)

Useful for review check-lists and fault based testing!

[source:Henrique Madeira, Universidade de Coimbra]

What are the main sources of bugs?

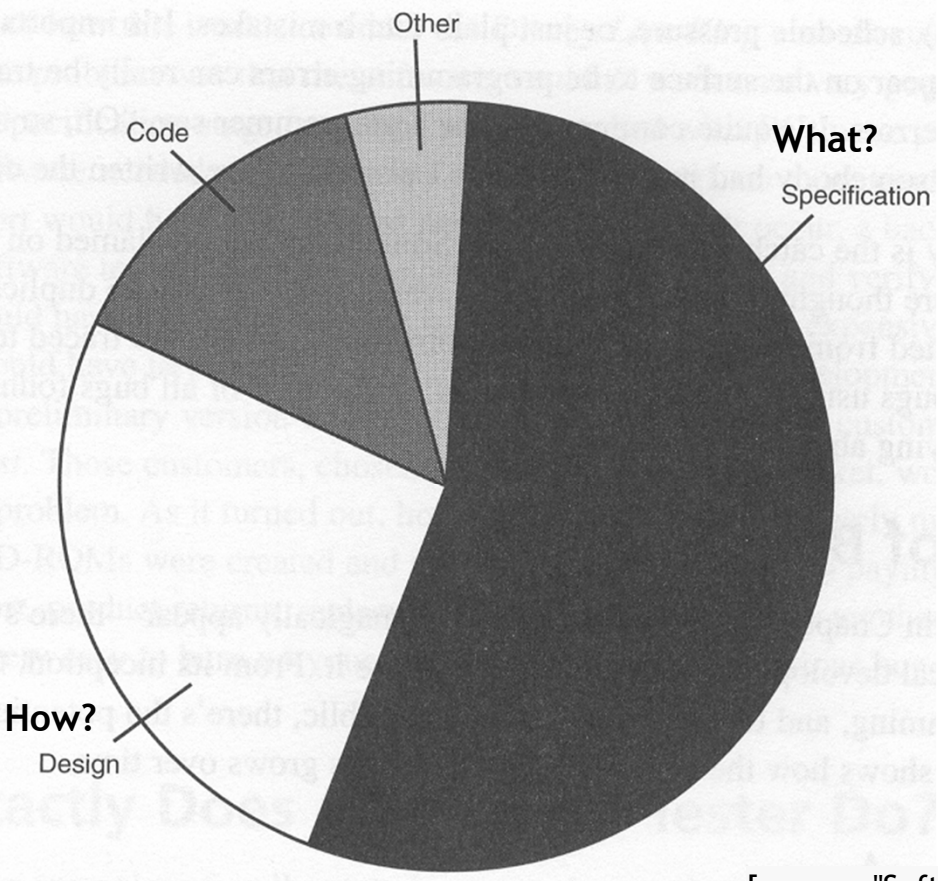


FIGURE 1.1
Bugs are caused for numerous reasons, but the main cause can be traced to the specification.

Take more time to get and understand specifications!

[source: "Software Testing", Ron Patton]

The current status of software quality (1)

- Microsoft Windows XP End-User License Agreement:

  11. LIMITED WARRANTY FOR PRODUCT ACQUIRED IN THE US AND CANADA.

  Microsoft warrants that the Product will perform substantially in accordance with the accompanying materials for a period of ninety days from the date of receipt.
  (…)

  YOUR EXCLUSIVE REMEDY. Microsoft's and its suppliers' entire liability and your exclusive remedy shall be, at Microsoft's option from time to time exercised subject to applicable law, (a) **return of the price paid** (if any) for the Product, or (b) **repair or replacement** of the Product, that does not meet this Limited Warranty and that is returned to Microsoft with a copy of your receipt.
  (..)

The current status of software quality (2)

	India	Japan	US	Europe & other	Total
Number of projects	24	27	31	22	104
Median output[1]	209	469	270	436	374
Median defect rate[2]	.263	.020	.400	.225	.150

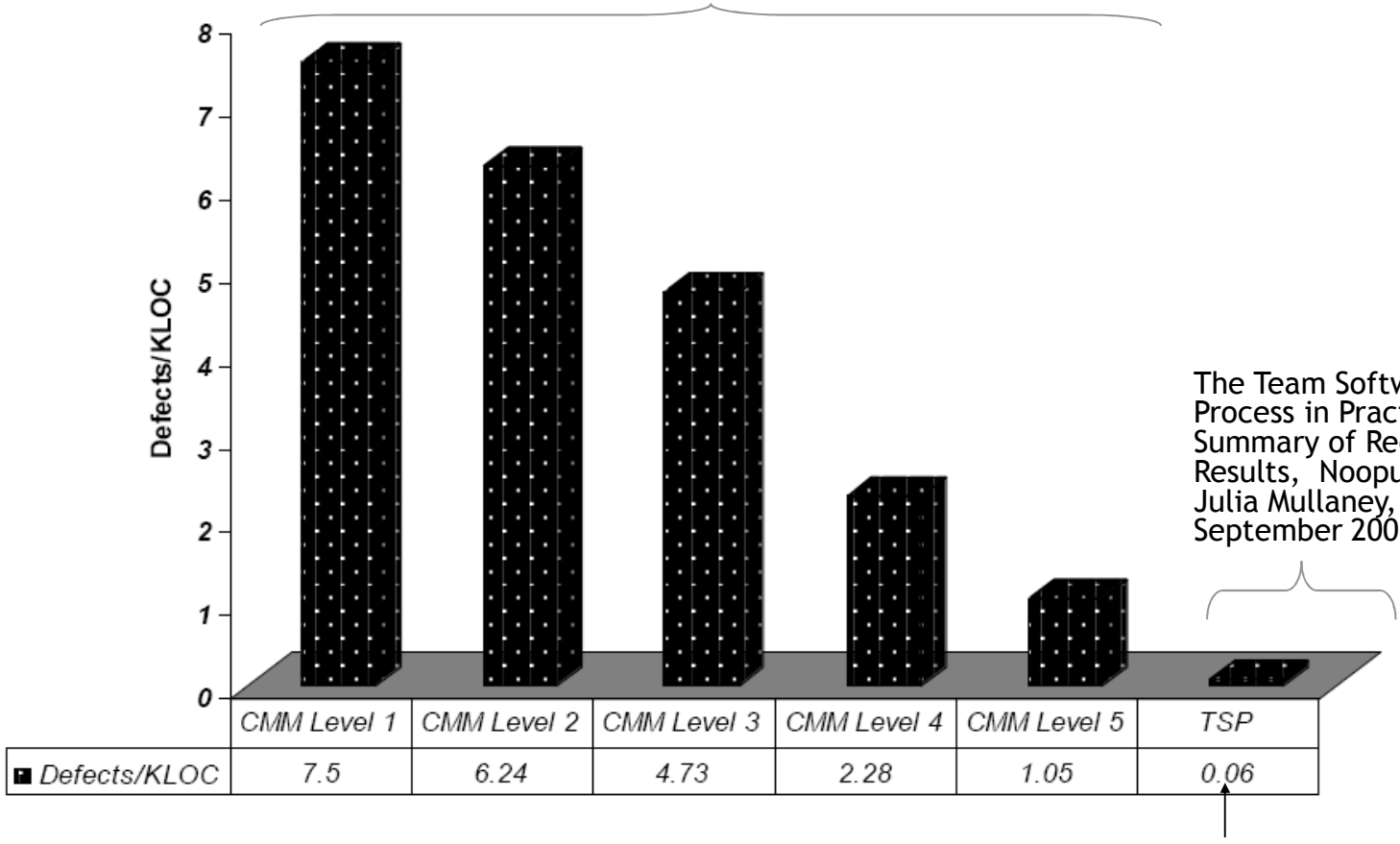
[1] (new?) LOC / programmer-month (considering the whole life cycle)

[2] Number of defects/ KLOC reported by customers in the first year post delivery

[source: "Software Development Worldwide: The State of the Practice", M. Cusumano (MIT), A. MacCormack (Harvard Univ.), C. F. Kemerer (Pittsburgh Univ.), B. Crandall (HP), IEEE SOFTWARE, 2003]

The current status of software quality (3)

Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Reading, MA: Addison-Wesley, 2000.



The Team Software Process in Practice: A Summary of Recent Results, Noopur Davis, Julia Mullaney, SEI, September 2003

Figure 17: Average Defect Density of Delivered Software

(~ 1 defect / 330 pages of 50 lines each)

The importance of software quality (1)

We depend more and more on software …

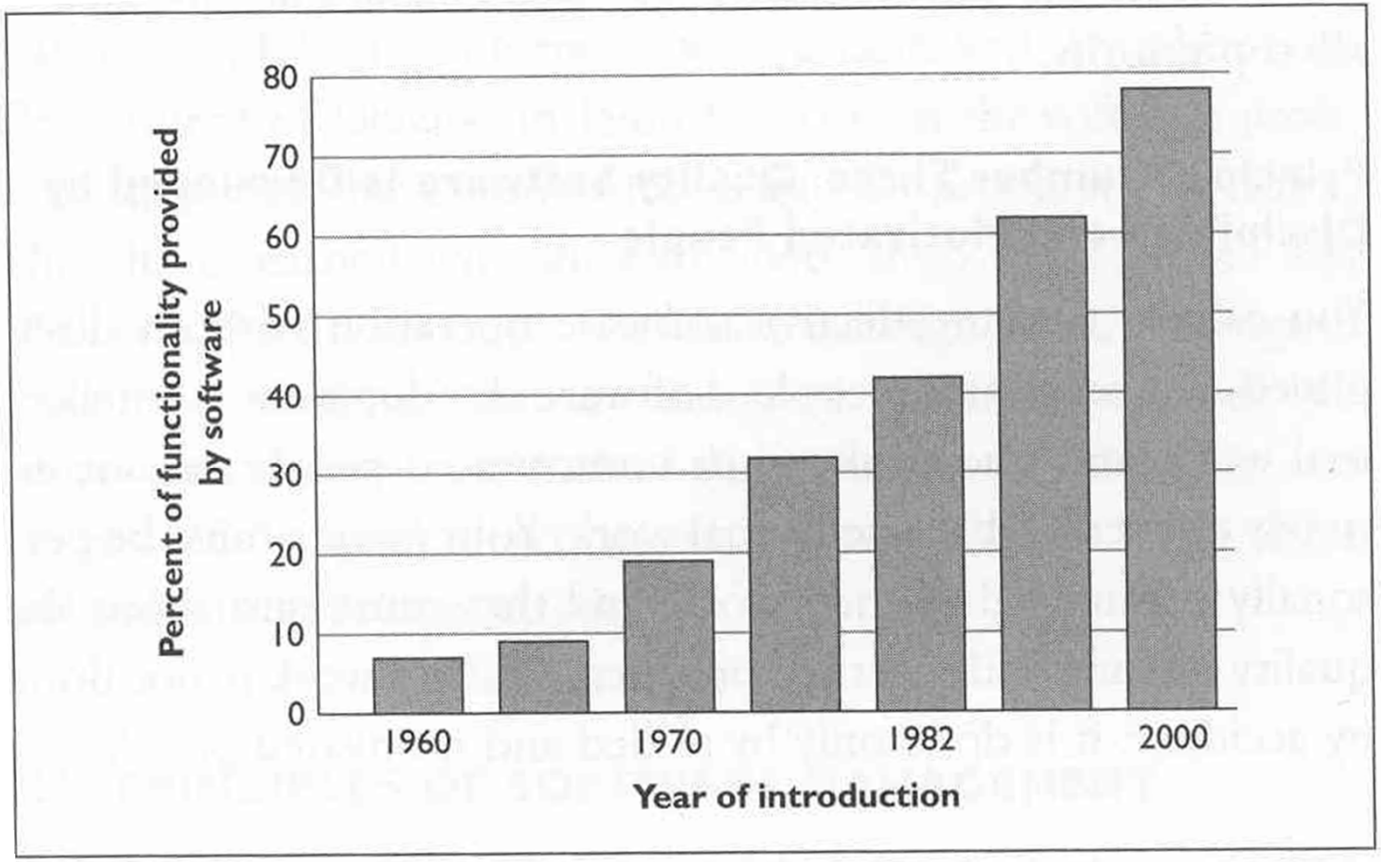


Figure 1.1 Software functionality in military aircraft

[source: Watts Humphrey, "Winning with Software", 2002]

The importance of software quality (2)

- Software size increases by a factor of 10 every 10 years …
  - 50 KLOC - Word 3.0 for DOS
  - 1 MLOC - Unix, System V, Release 4, 1990
  - 10 MLOC – Linux, 2000
  - 50 MLOC - Windows Vista, 2007

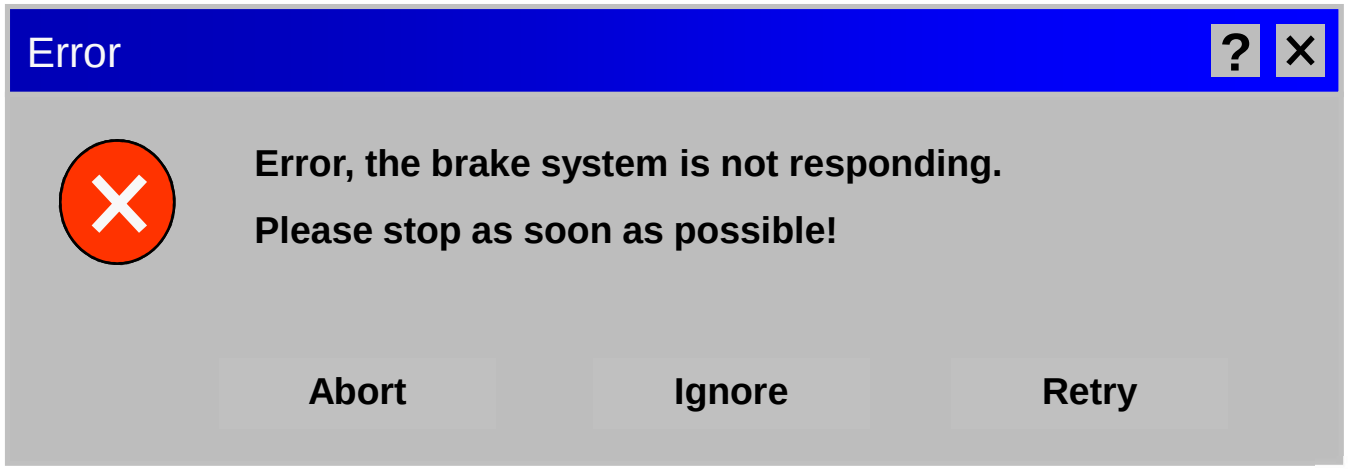
- 50 MLOC * 1 defect/KLOC = 50.000 defects in Windows Vista?

- Ideally: 1 defect / MLOC

KLOC = 1000 lines of code
MLOC = 1000000 lines of code

The importance of software quality (3)

Impact on life and environment …



Tim Davis, Ford Motor Company, 27th International Conference on Software Engineering, 2005

Economic impact …

- According to the National Institute of Standards and Technology (NIST), USA, direct costs of software error represent 0,6 % of GNP (PIB) in the USA

  [source: "The Economic Impacts of Inadequate Infrastructure for Software Testing", NIST, May 2002]

Bad examples

- Intel gastou $475 m na correcção do erro da virgula flutuante do Pentium em 1994 (Computer Science, Springer Verlag – 1995)
- PrimeCo Personal Communications cancelou contrato de $500M com Motorola por causa de falhas (Wall Street Journal – 24/02/98)
- Time Warner Communications gastou $1B em sistema de informação para tentar entrar no negócio residencial da rede telefónica (Computerworld – 05/05/97)
- National Bank of Australia perdeu $1,75B devido a erro não detectado durante 2 anos (New York Times – Nov/01)
- Ariane 5 (10 anos de desenvolvimento no valor de $7B) com uma carga de $500M, explodiu 40 segundos após lançamento. Módulo de software gerou evento não tratado (ESA – 1996)
- Therac-25 ministrou doses incorrectas de Raios X em pacientes entre 1985 e 1987 – 6 mortes (IEEE Computer – 07/07/93)
- More and constant updates at **http://www.risks.org**

Why quality pays

- Poor-quality software can be life-threatening
  - Or mission/business/environment/economy-threatening …
- Quality work saves time and money
  - E.g., with the PSP/TSP, defect density decreases by a factor of 10 while productivity increases
- Quality work is more predictable
  - The testing and repair effort of a bad quality product is unpredictable
  - See TSP data

[Source: Watts Humphrey, “Winning with Software”, 2002]

Quality costs

- Costs of **conformance**
  - All costs associated with planning and running tests (and revisions) just one time
- Costs of **nonconformance**
  - Costs due to **internal** failures (before release)
    - Cost of isolating, reporting and regression testing bugs (found before the product is released) to assure that they're fixed (left-hand side of fig. 1.2)
  - Costs due to **external** failures (after release)
    - If bugs are missed and make it through to the customers, the result will be costly product support calls, possibly fixing, retesting, and releasing the software, and – in a worst case-scenario – a product recall or lawsuits (right-hand side of fig. 1.2)

[source: "Software Testing", Ron Patton]

Quality costs
Costs of nonconformance (1)

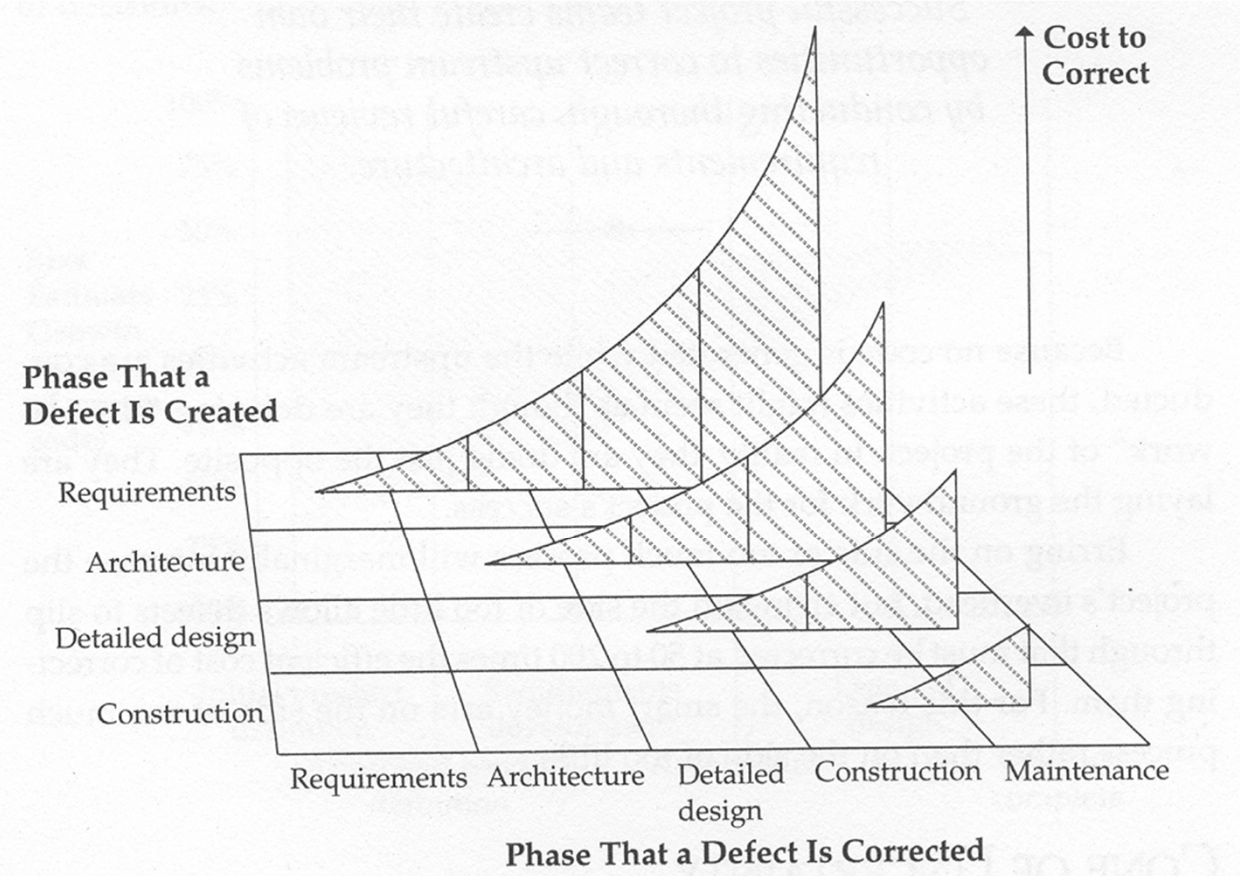


FIGURE 3-5 *Increase in defect cost as time between defect creation and defect correction increases. Effective projects practice "phase containment"—the detection and correction of defects in the same phase in which they are created.*

[source: "Software Project Survival Guide", Steve McConnell]

Quality costs
Quality is free!?

- In his book "Quality is Free: The Art of Making Quality Certain", Philip Crosby argues that the costs of conformance plus the costs of nonconformance due to internal failures is (usually) less than the costs of nonconformance due to external failures

costs of conformance
+
costs of nonconformance due to internal failures

<

costs of nonconformance due to external failures

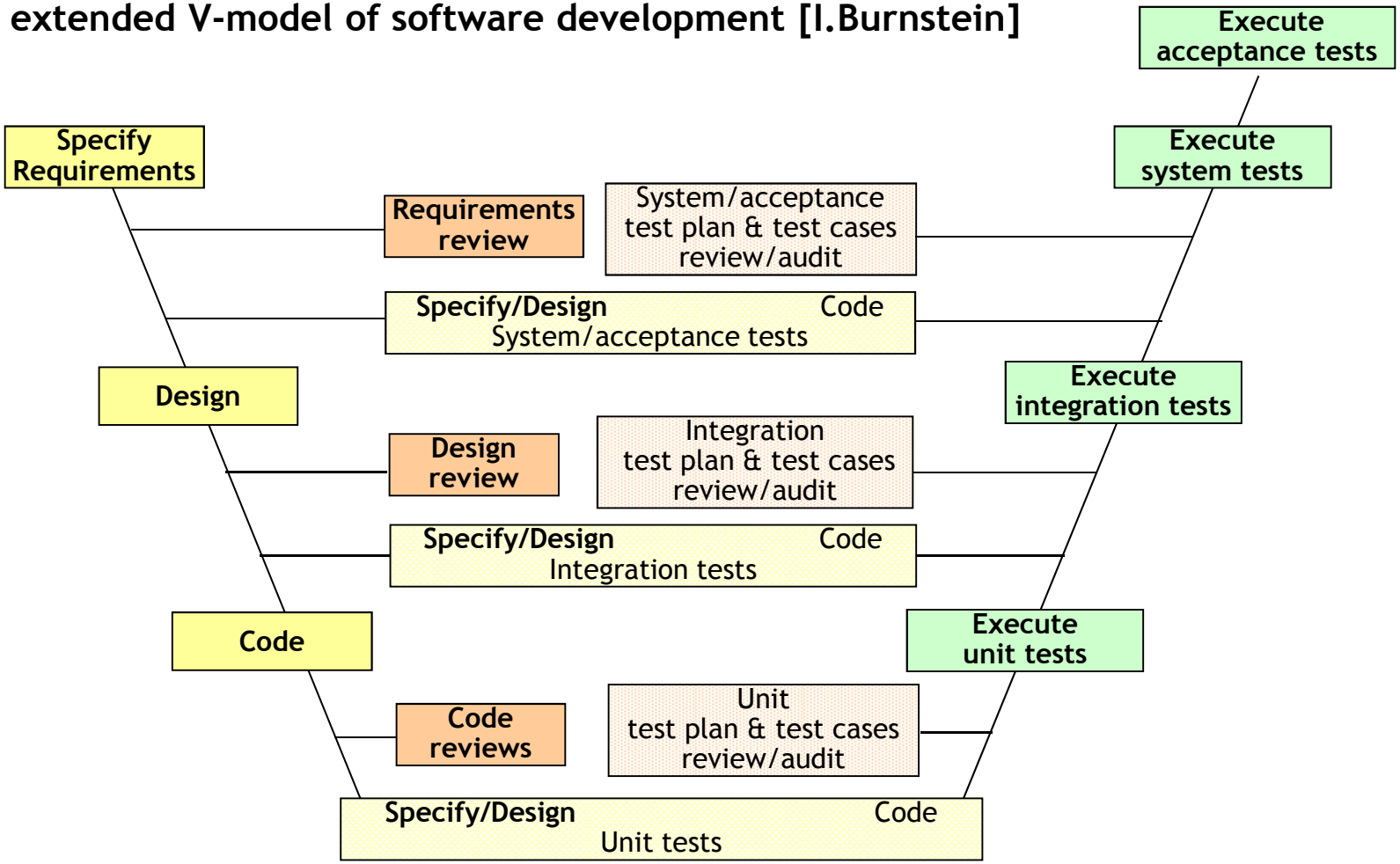
[source: Ron Patton, "Software testing“]

Software tests

- **Dynamic** V&V technique, concerned with exercising the system under test with defined **test cases** and observing its behaviour to discover **defects** (discrepancies between observed and expected behaviour)
  - Since exhaustive testing is usually impossible, “program testing can be used to show the presence of bugs, but never to show their absence” [Dijkstra, 1972]
  - Defect testing – find defects, using test data/test cases that have higher probability of finding defects
- A secondarily goal is to increase the confidence on the software correctness and to evaluate product quality
  - Statistical testing – estimate the value of a software quality metric (efficiency, availability, reliability, …), using representative test cases / test data
- Advantages: automation, ultimate validation technique, evaluation of external quality attributes, …

Tests and reviews along the software life cycle (2)

The extended V-model of software development [I.Burnstein]



TQSO - Teste e Qualidade de Software (*Software Testing and Quality*)

Unit Testing

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Unit testing - definition (1)

- **Unit testing**: Testing of individual hardware or software units or groups of related units [IEEE 90].

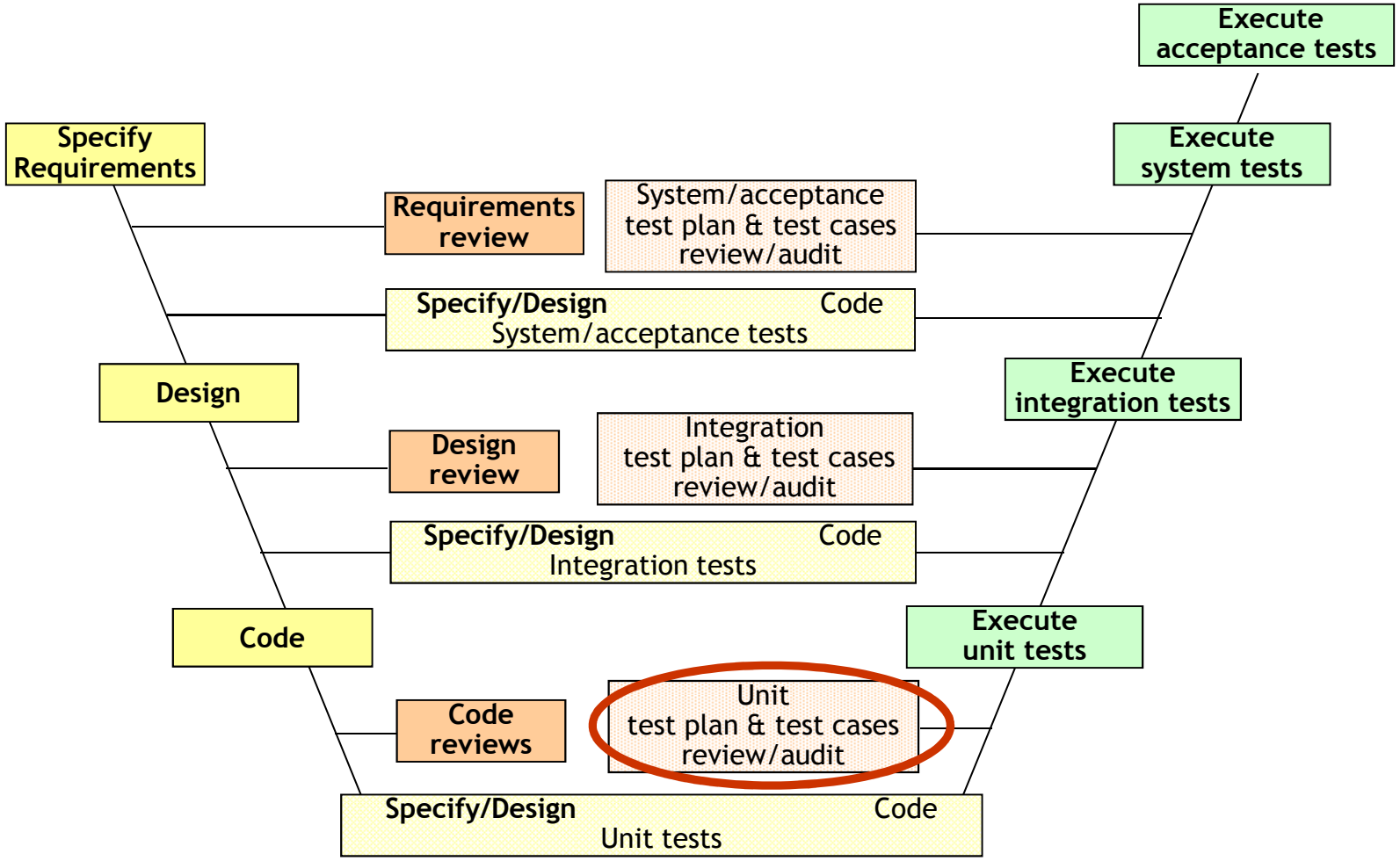
- **Unit testing** is a development procedure where programmers create tests as they develop software. The tests are simple short tests that test functionality of a particular unit or module of their code, such as a class or function.

Unit testing - definition (2)

- **Unit testing** is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. This testing mode is a component of **Extreme Programming (XP)**, a pragmatic method of software development that takes a meticulous approach to building a product by means of continual testing and revision.

- A **unit test** is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A “unit” is a method or function.

Unit testing



The extended V-model of software development [I.Burnstein]

Benefits of unit testing

- Facilitates change
  - Allows refactor code at a later date and make sure the module still works correctly
- Simplifies integration
  - Helps to eliminate uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier
- Documentation
  - Provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit API

“Good” unit tests (1)

- What properties a unit test should have?
  - It is automated and repeatable
  - It is easy to implement
  - Once it is written, it stays on for the future
  - Anyone can run it
  - It runs at the push of a button
  - It runs quickly
- Unit tests are code …
  - Maintanability
  - Readability
  - Correctness
  - Documentation
  - …

“Good” unit tests (2)

- **Runs fast**, runs fast, runs fast. If the tests are slow, they will not be run often.
- **Separates or simulates environmental dependencies** such as databases, file systems, networks, queues, and so on. Tests that exercise these will not run fast, and a failure does not give meaningful feedback about what the problem actually is.
- **Is very limited in scope**. If the test fails, it is obvious where to look for the problem. Use few Assert calls so that the offending code is obvious. It is important to only test one thing in a single test.
- **Runs and passes in isolation**. If the tests require special environmental setup or fail unexpectedly, then they are not good unit tests. Change them for simplicity and reliability. Tests should run and pass on any machine. The "works on my box" excuse doesn't work.
- **Often uses stubs and mock objects**. If the code being tested typically calls out to a database or file system, these dependencies must be simulated, or mocked. These dependencies will ordinarily be abstracted away by using interfaces.
- **Clearly reveals its intention**. Another developer can look at the test and understand what is expected of the production code.

[http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3]

Separation of interface from implementation

- Because some classes may have references to other classes, testing a class can frequently spill over into testing another class.
  - Ex.: classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database.
- This is a mistake
  - unit test should never go outside of its own class boundary.
- Abstract an interface around database connection and implement it with your own **mock objects**
  - the independent unit can be more thoroughly tested
  - this results in a higher quality unit that is also more maintainable.

Mock objects

- **Mock objects** are simulated objects that mimic the behavior of real objects in controlled ways.
- Useful when a real object is impractical or impossible to incorporate into a unit test. If an object has any of the following characteristics, it may be useful to use a mock object in its place:
  - supplies non-deterministic results (e.g., the current time or the current temperature);
  - has states that are difficult to create or reproduce (e.g., a network error);
  - is slow (e.g., a complete database, which would have to be initialized before the test);
  - does not yet exist or may change behavior;
  - would have to include information and methods exclusively for testing purposes (and not for its actual task).

Mock objects (example)

- An alarm clock program which causes a bell to ring at a certain time might get the current time from the outside world. To test this, the **test must wait** until the alarm time to know whether it has rung the bell correctly. If a mock object is used in place of the real object, it can be programmed to provide the bell-ringing time (whether it is actually that time or not) so that the alarm clock program can be tested in isolation.

Some definitions

- **Test driver** – A software module or application used to invoke a test item and, often, provide test inputs (data), control and monitor execution. A test driver automates the execution of test procedures.
- **Test Harness** – A system of test drivers and other tools to support test execution (e.g., stubs, executable test cases and test drivers). The tool that actually executes the tests.
- **Test Stubs** – stubs simulate collaboration (an implementation of an interface that returns hard-coded values) while mocks test collaboration (an object which, in addition to implementing the interface and returning meaningful values, allows for verification that the correct calls were made upon the object, perhaps in the correct order; to verify the interaction between two classes).

Six rules of unit testing

- Write the test first
- Never write a test that succeeds the first time
- Start with the null case, or something that doesn't work
- Don't be afraid of doing something trivial to make the test work
- Loose coupling and testability go hand in hand
- Use mock objects

[http://radio.weblogs.com/0100190/stories/2002/07/25/sixRulesOfUnitTesting.html]

Limitations of unit testing (1)

- Testing, in general, cannot be expected to catch every error in the program. Unit tests can only show the presence of errors; it cannot show the absence of errors.
- It only tests the functionality of the units themselves.
- It may not catch integration errors, performance problems, or other system-wide issues.
- Unit testing is more effective if it is used in conjunction with other software testing activities.

Limitations of unit testing (2)

- Software testing is a combinatorial problem.
  - For example, every **boolean decision** statement requires at least **two tests**: one with an outcome of "true" and one with an outcome of "false". As a result, for **every line of code** written, programmers often need **3 to 5 lines of test code**. Therefore, it is unrealistic to test all possible input combinations for any non-trivial piece of software without an automated characterization test generation tool such as JUnit Factory used with Java code or many of the tools listed in List of **unit testing frameworks**.

Limitations of unit testing (3)

- To obtain the intended benefits from unit testing use of a version control system is essential
  - If a later version of the unit fails a particular test that it had previously passed, the version-control software can provide a list of the source code changes (if any) that have been applied to the unit since that time.
- It is also essential to implement a sustainable process for ensuring that test case failures are reviewed daily and addressed immediately
  - If such a process is not implemented and ingrained into the team's workflow, the application will evolve out of sync with the unit test suite—- increasing false positives and reducing the effectiveness of the test suite.

Unit testing frameworks

- Help simplify the process of unit testing
  - Log test cases that fail
  - Automatically flag and report in a summary these failed test cases.
  - Depending upon the severity of a failure, the framework may halt subsequent testing.
- Examples
  - JUnit
  - JTest
  - NUnit
  - MbUnit
  - …

[http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks]

Test fixture

- **Test fixture** refers to the fixed state used as a baseline for running tests in software testing. The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable. Some people call this the test context.
- Examples of fixtures:
  - loading a database with a specific, known set of data
  - erasing a hard disk and installing a known clean operating system installation
  - copying a specific known set of files
  - preparation of input data and setup/creation of fake or mock objects

Setup and teardown (1)

- You want to avoid duplicated code when several tests share the same initialization and cleanup code.
- … before and after each test method inside a test class
  - **JUnit**: Use the setUp( ) and tearDown( ) methods. Both of these methods are part of the junit.framework.TestCase class.
  - **NUnit**: build methods and annotate them with SetUp and TearDown tags

Setup and teardown (2)

- …. One-time set ut and tear down
  - You want to run some setup code one time and then run several tests. You only want to run your cleanup code after all of the tests are finished, ex.: establish a database connection.
  - **JUnit**: Use the junit.extensions.TestSetup class to define **test suites**
    - Pass a TestSuite to the TestSetup constructor. This means that TestSetup's setUp( ) method is called once before the entire suite, and tearDown( ) is called once afterwards.

      TestSetup setup = new TestSetup(new TestSuite(TestPerson.class)) {…}
  - **NUnit**: methods annotated with TestFixtureSetup and TestFixtureTearDown attributes.

TQS - Teste e Qualidade de Software (*Software Testing and Quality*)

Mutation Testing

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Mutation Testing

- Mutation testing is a **fault-based testing technique** that is based on the assumption that a program is well tested if all simple faults are predicted and removed; complex faults are coupled with simple faults and are thus detected by tests that detect simple faults.
- Mutation testing is used to test the quality of your test suite. This is done by mutating certain statements in your source code and checking if your test code is able to find the errors.
  - How do you know that you can trust your unit tests?
  - How do you know that they're really telling you the truth?
  - If they don't find a bug, does that really mean that there aren't any?
  - What if you could test your tests?

Mutation testing

- **Mutation testing** is a method of software testing, which involves modifying program's source code or byte code in small ways. In short, any tests which pass after code has been mutated are defective. These, so-called ***mutations***, are based on well-defined ***mutation operators*** that either mimic typical programming errors (such as using the wrong operator or variable name) or force the creation of valuable tests (such as driving each expression to zero). The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

Mutation testing

- **Mutation Testing** is the process of generating tests to improve the mutation analysis score.
- **Mutation Analysis** is the process of measuring how good a test set is (how many mutations it kills).
- The **mutation score** is the ratio of dead mutants over the total number of non-equivalent mutants. Thus, the tester's goal is to raise the mutation score to 1.00, indicating that all mutants have been detected. A test set that kills all the mutants is said to be adequate relative to the mutants. If (as is likely) mutants are still alive, the tester can enhance the set of test cases by supplying new inputs.

Mutation testing

- A **mutation operator** is a rule that is applied to a program to create mutants. Typical mutation operators, for example, replace each operand by every other syntactically legal operand, or modify expressions by replacing operators and inserting new operators, or delete entire statements.
- Examples of traditional mutation operators
  - Statement deletion.
  - Replace each Boolean sub-expression with true and false.
  - Replace each arithmetic operation with another one, e.g. + with *, - and /.
  - Replace each Boolean relation with another one, e.g. > with >=, == and <=.
  - Replace each variable with another variable declared in the same scope (variable types should be the same).

Method-level Mutation Operators

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

Table 1: 11 Method-level Mutation Operators for Java

Mutation operators

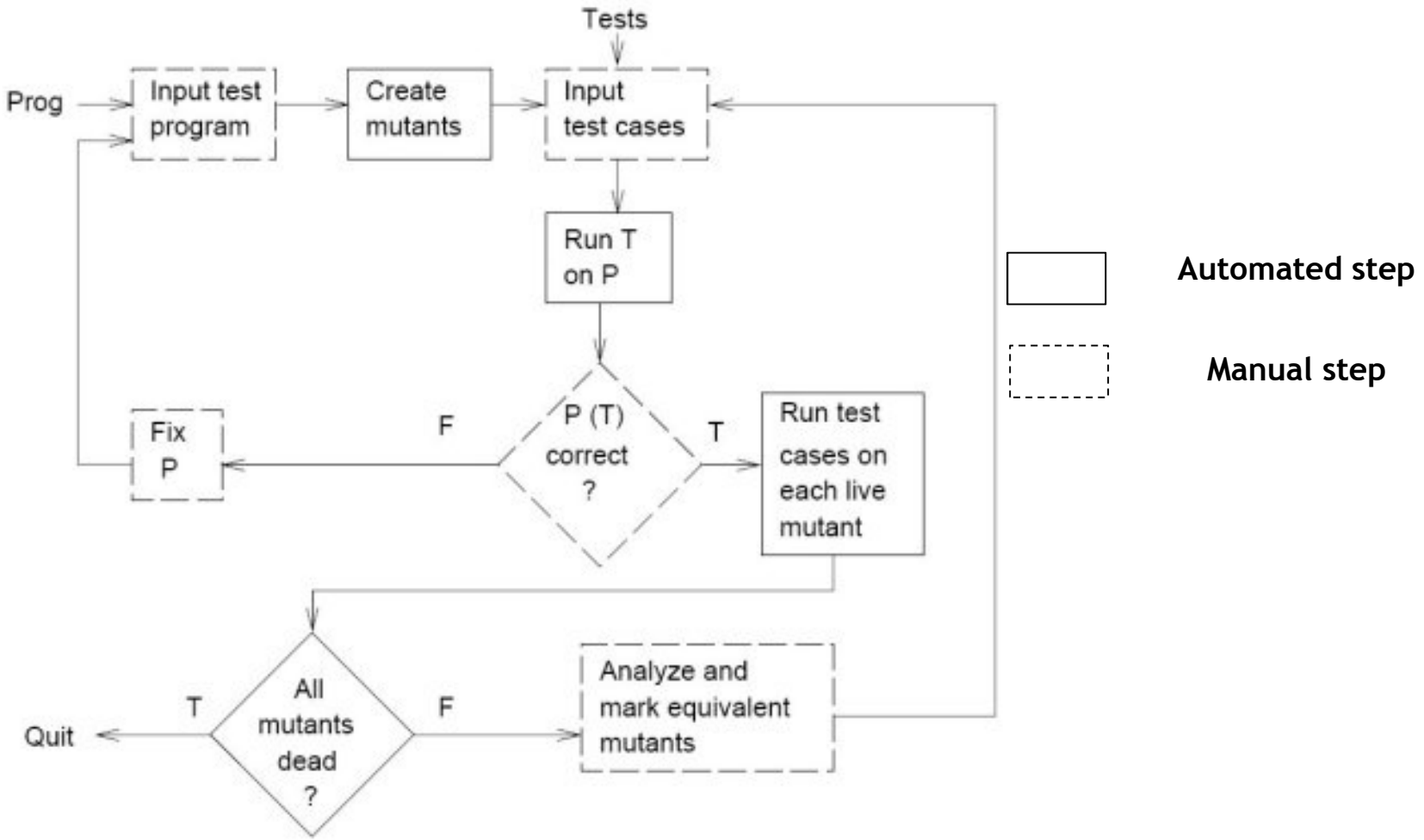
- Beside this, there are mutation operators for object-oriented languages, for concurrent constructions, complex objects like containers etc. They are called class-level mutation operators.
- For example, muJava classifies class mutation operators into four groups: Encapsulation, Inheritance, Polymorphism and Java-Specific Features.

Inter-Class Mutation Operators

Language Feature	Operator	Description
Encapsulation	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	super keyword insertion
	ISD	super keyword deletion
	IPC	Explicit call to a parent's constructor deletion
Polymorphism	PNC	new method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Cast type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
Java-Specific Features	JTI	this keyword insertion
	JTD	this keyword deletion
	JSI	static modifier insertion
	JSD	static modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Acessor method change
	EMM	Modifier method change

Table 1: Mutation Operators for Inter-Class Testing

Mutation testing process



Mutation testing process

- Starts with a code component and its associated test cases (in a state such that the code passes all test cases)
- The original code component is modified in a simple way (replace operators, constants, etc.) to provide a set of similar components that are called mutants, based on typical errors
- The original test cases are run with each mutant
- Live mutants cannot be distinguished from the original program (parent)
- Distinguishing a mutant from its parent is referred to as killing such mutant

Mutation testing process

- If a mutant remains live (passes all the test cases), then either the mutant is equivalent to the parent (and is ignored), or it is not equivalent, in which case additional test cases should be developed in order to kill such mutant
- The rate of mutants "killed" (after removing mutants that are equivalent to the original code) gives an indication of the rate of undetected defects that may exist in the original code

Weak/Strong mutation testing

```
if (a && b) c=1; else c=0;
```

- The condition mutation operator would replace '&&' with '||' and produce the following mutant:

```
if (a || b) c=1; else c=0;
```

- Now, for the test to kill this mutant, the following condition should be met:
  - (1) Test input data should cause different program states for the mutant and the original program. For example, a test with a=1 and b=0 would do this.
  - (2) The value of 'c' should be propagated to the program's output and checked by the testing.

Weak/Strong mutation testing

- ***Weak mutation testing*** (or *weak mutation coverage*) requires that only the first condition is satisfied. ***Strong mutation testing*** requires that both conditions are satisfied. Strong mutation is more powerful, since it ensures that the test suite can really catch the problems. Weak mutation is closely related to code coverage methods. It requires much less computing power to ensure that the test suite satisfies weak mutation testing than strong mutation testing.
- **Weak Mutation Testing** is a coverage measure - i.e. tells you about the code that is run by your tests.
- **Strong Mutation Testing** measures whether your code needs to be like it is to pass the tests.

Equivalent mutants

- Many mutation operators can produce equivalent mutants. For example, boolean relation mutation operator will replace “==“ with ">=" and produce the following mutant:

original

```
int index=0;
while (…) { …;
   index++;
if (index==10)
   break;
}
```

mutant

```
int index=0;
while (…) { …;
   index++;
if (index>=10)
   break;
}
```

- However, it is not possible to find a test case which could kill this mutant. The resulting program is equivalent to the original one. Such mutants are called equivalent mutants.

Equivalent mutants

- Equivalent mutants detection is one of biggest obstacles for practical usage of mutation testing. The effort, needed to check if mutants are equivalent or not, can be very high even for small programs.

Computational effort

- One of the barriers to the practical use of mutation testing is the unacceptable computational expense of generating and running vast numbers of mutant programs against the test cases. The number of mutants generated for a software unit is proportional to the product of the number of data references and the number of data objects. Typically, this is a large number for even small software units. Because each mutant program must be executed against at least one, and potentially many, test cases, mutation analysis requires large amounts of computation.

Computational effort

- Approaches to reduce this computational expense usually follow one of three strategies: do fewer, do smarter, or do faster.
  - The **do fewer** approaches seek ways of running fewer mutant programs without incurring intolerable information loss.
  - The **do smarter** approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs or seek to avoid complete execution.
  - The **do faster** approaches focus on ways of generating and running each mutant program as quickly as possible.

Tools

- Mothra
- Proteum
- muJava
- MuClipse
- Jumble
- JesTer (for Java)
- PesTer (for Python)
- SQLMutation
- Nester (for C#)
- …

Exercises

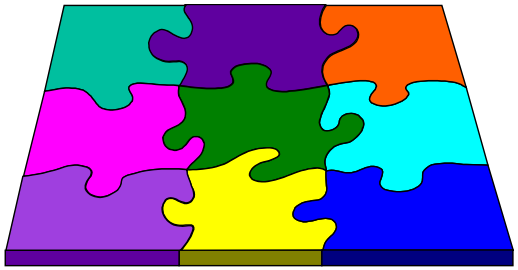
```
1: Bool A,B,C; Int V;
2: read(A);read(B);read(C);read(V);
3: while (V<20)  {
4:        if ( A /\ ( B \/ C ) )
5:            V:= V+5;
6:        if (A /\ ~B)
7:            V:=V+10;
8:  }
9: print V;
```

- Conceba casos de teste (com grau de cobertura de 100%) para o programa considerando os seguintes mutantes. Indique qual o valor retornado pelo programa em cada um desses casos de teste.

Id	Mutação
1	3: while (V<=20) A=T; B=T; C=T; V=20; R=20 ;R'=25
2	4: if (A \/ (B \/ C)) A=F; B=F; C=T ;V=15; R="loop" ;R'=20
3	6: if (A \/ ~B) A=F; B=F; C=T ;V=15; R="loop" ;R'=25

TQS - Teste e Qualidade de Software (*Software Testing and Quality*)

Integration testing



Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Integration testing

- Testing of groups of components integrated to create a sub-system. Components should be tested previously.
- Usually the responsibility of an independent testing team (except sometimes in small projects)
- Integration testing should be black-box testing with tests derived from the technical specification
- A principal goal is to detect defects that occur on the interfaces of units
- Main difficulty is localising errors
- *Incremental* integration testing (as opposed to *big-bang* integration testing) reduces this difficulty

Interfaces types

- Parameter interfaces
  - Data passed from one procedure to another
- Shared memory interfaces
  - Block of memory is shared between procedures
- Procedural interfaces
  - Sub-system encapsulates a set of procedures to be called by other sub-systems
- Message passing interfaces
  - Sub-systems request services from other sub-systems

Interface errors

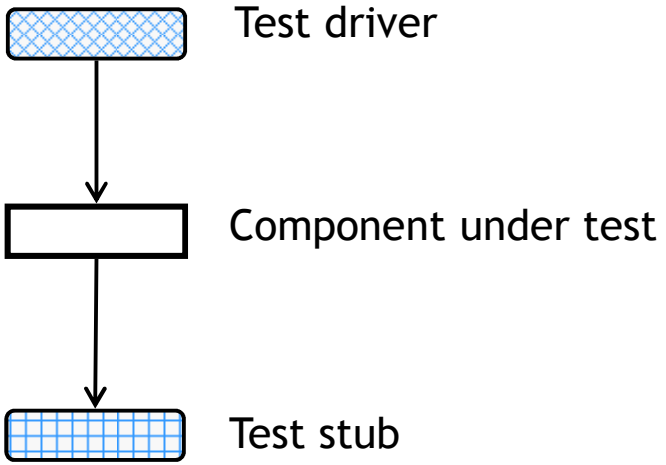
- Interface misuse
  - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order
- Interface misunderstanding
  - A calling component embeds assumptions about the behaviour of the called component which are incorrect
- Timing errors
  - The called and the calling component operate at different speeds and out-of-date information is accessed

Interface testing guidelines

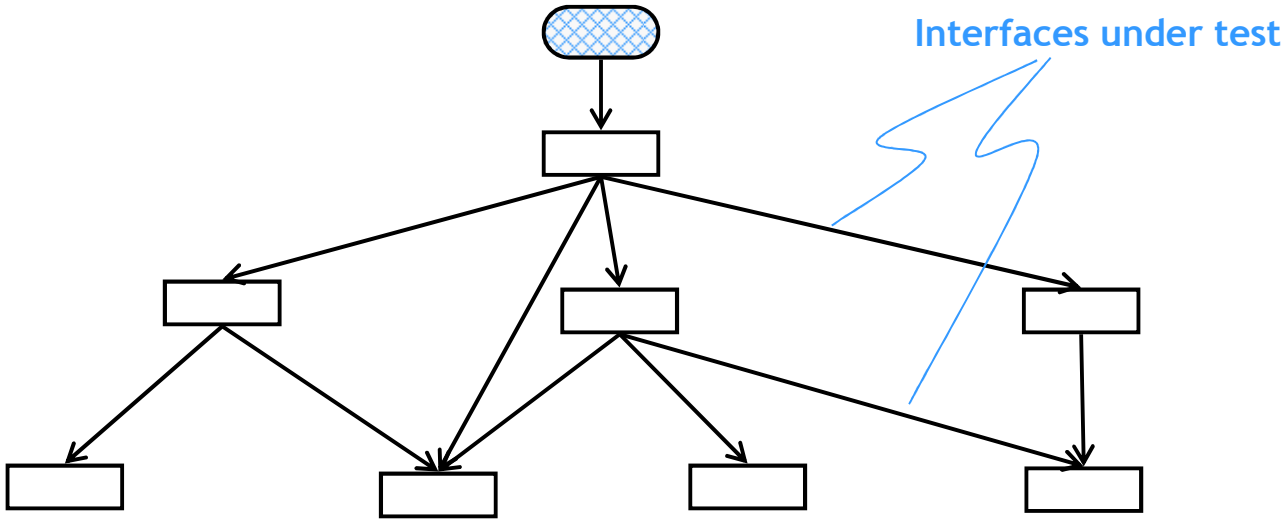
- Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- Always test pointer parameters with null pointers
- Design tests which cause the component to fail
- Use stress testing in message passing systems
- In shared memory systems, vary the order in which components are activated

Test harness: drivers and stubs

- Test harness: auxiliary code developed to support testing
- Test drivers
  - Call the target code, simulating calling units or a user
  - In automatic testing: implementation of test cases and procedures
- Test stubs
  - Simulate modules/units/systems called by the target code
  - Mock objects can be used for this purpose



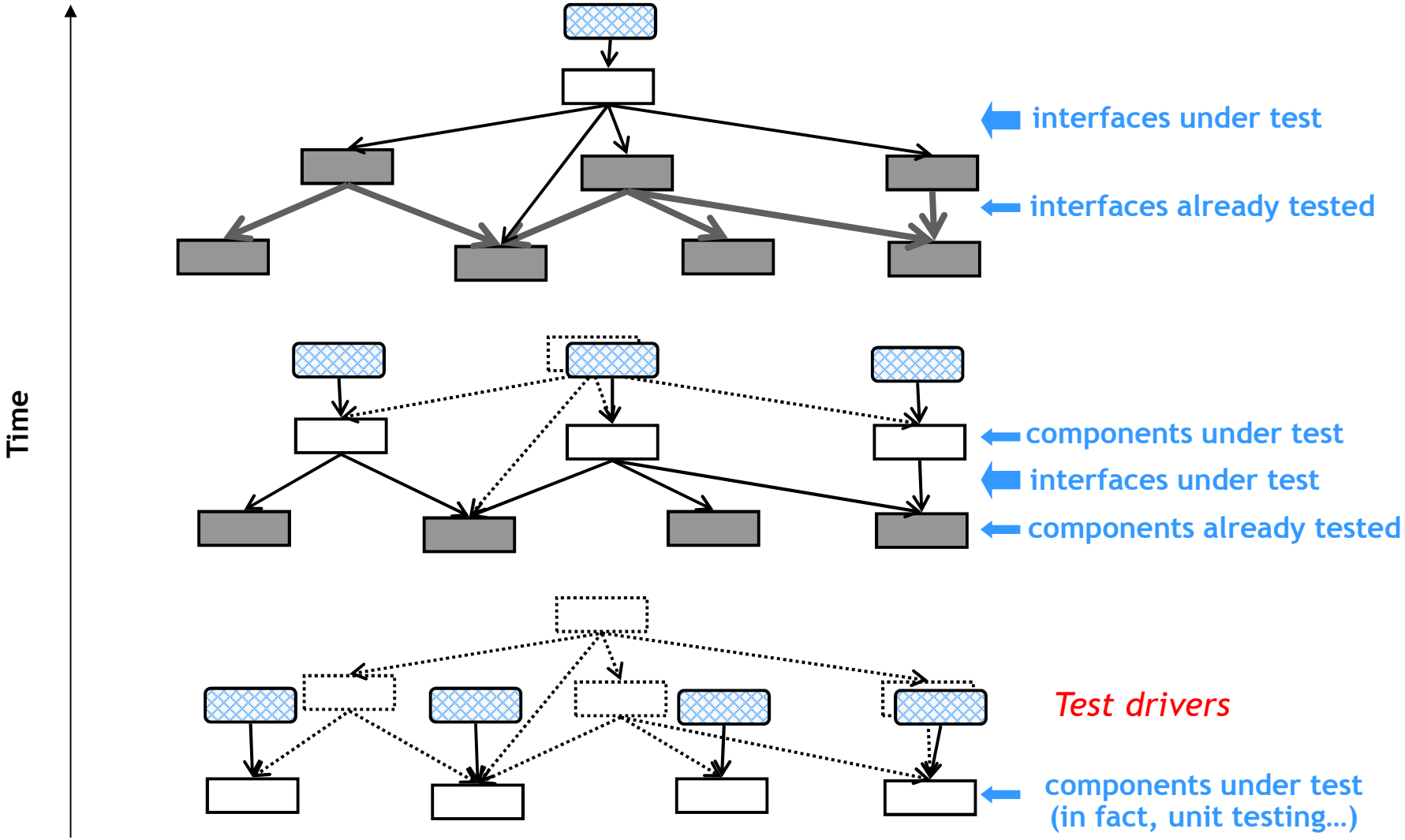
Big-bang integration testing



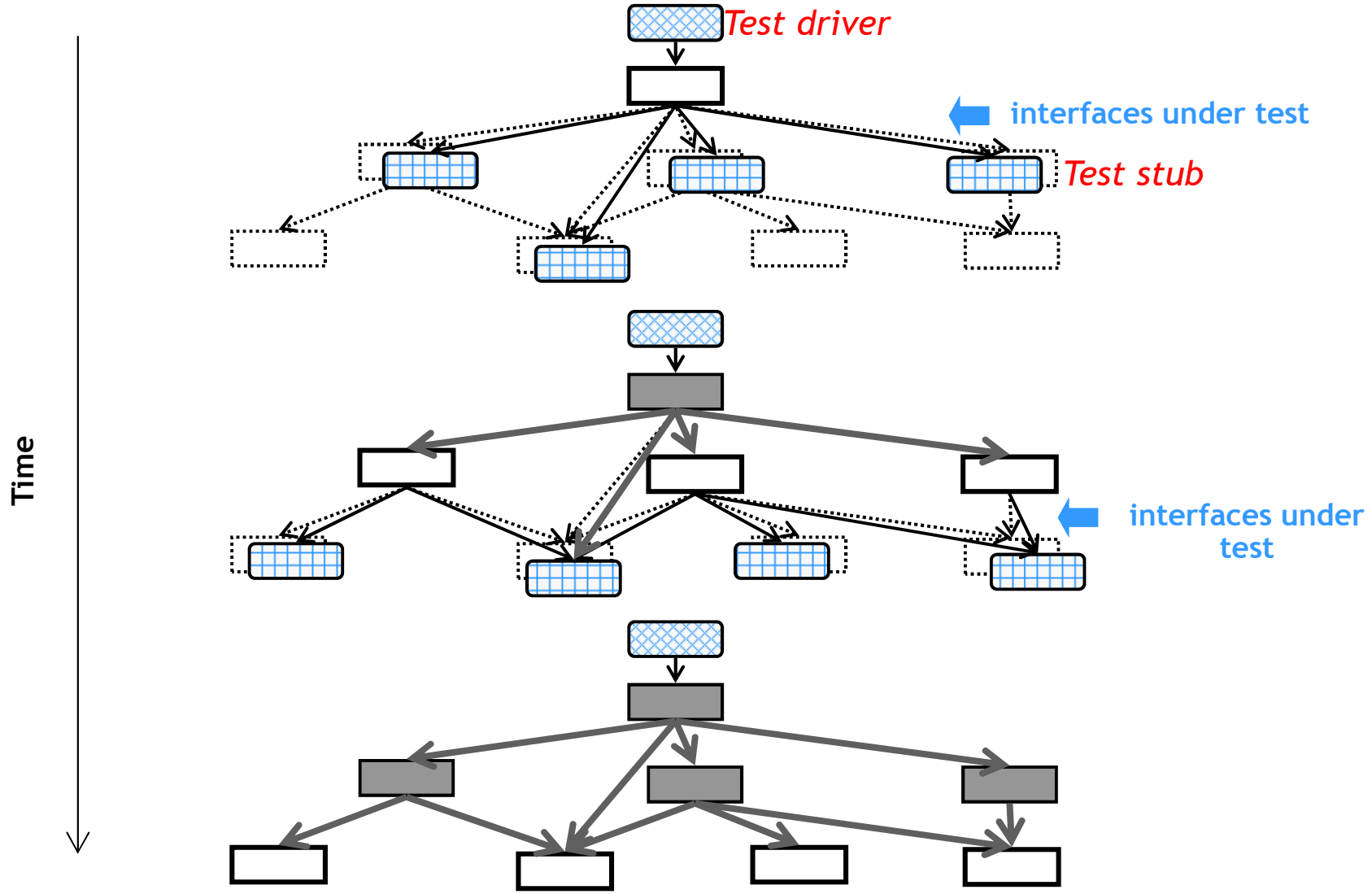
Incremental integration testing

- Top-down integration testing
  - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- Bottom-up integration testing
  - Integrate individual components in levels until the complete system is created
- Sandwich Testing
  - Combination of Top-down with Bottom-up testing
- Collaboration integration testing
  - Appropriate for iterative development strategies where software components are created and fatten as new use cases are implemented (through a collaboration of objects and components)
  - Scenario based testing
- The integration testing strategy must follow the software construction strategy

Bottom-up integration testing



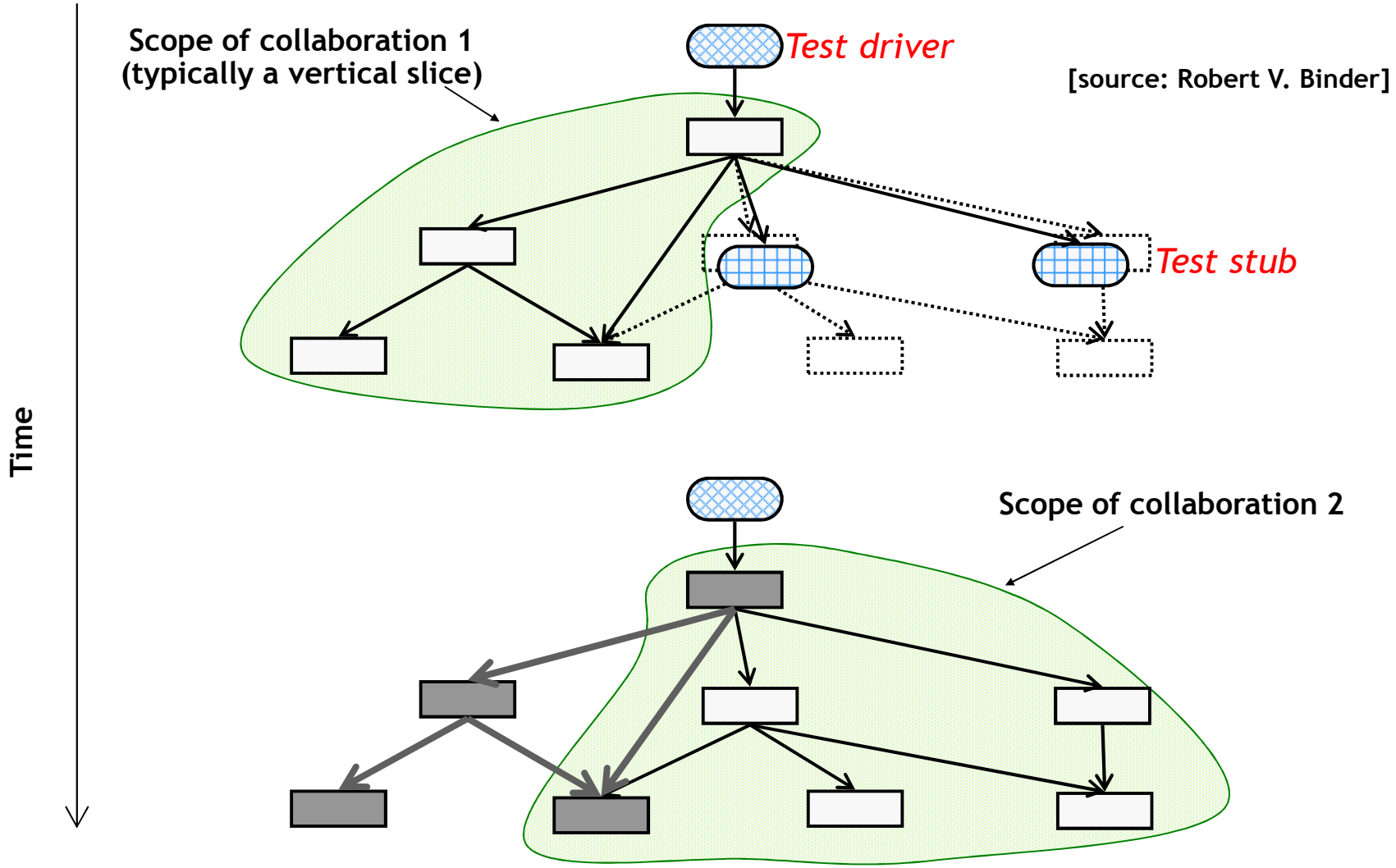
Top-down integration testing



Top-down versus bottom-up

- Architectural validation
  - Top-down integration testing is better at discovering errors in the system architecture
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development
- Test implementation
  - Top-down integration requires the development of complex stubs to drive significant data upward while bottom-up integration requires drivers. Often a combination of approaches known as sandwich
- Test observation
  - Problems with both approaches. Extra code may be required to observe tests

Collaboration integration testing



TQS - Teste e Qualidade de Software (*Software Testing and Quality*)

System testing

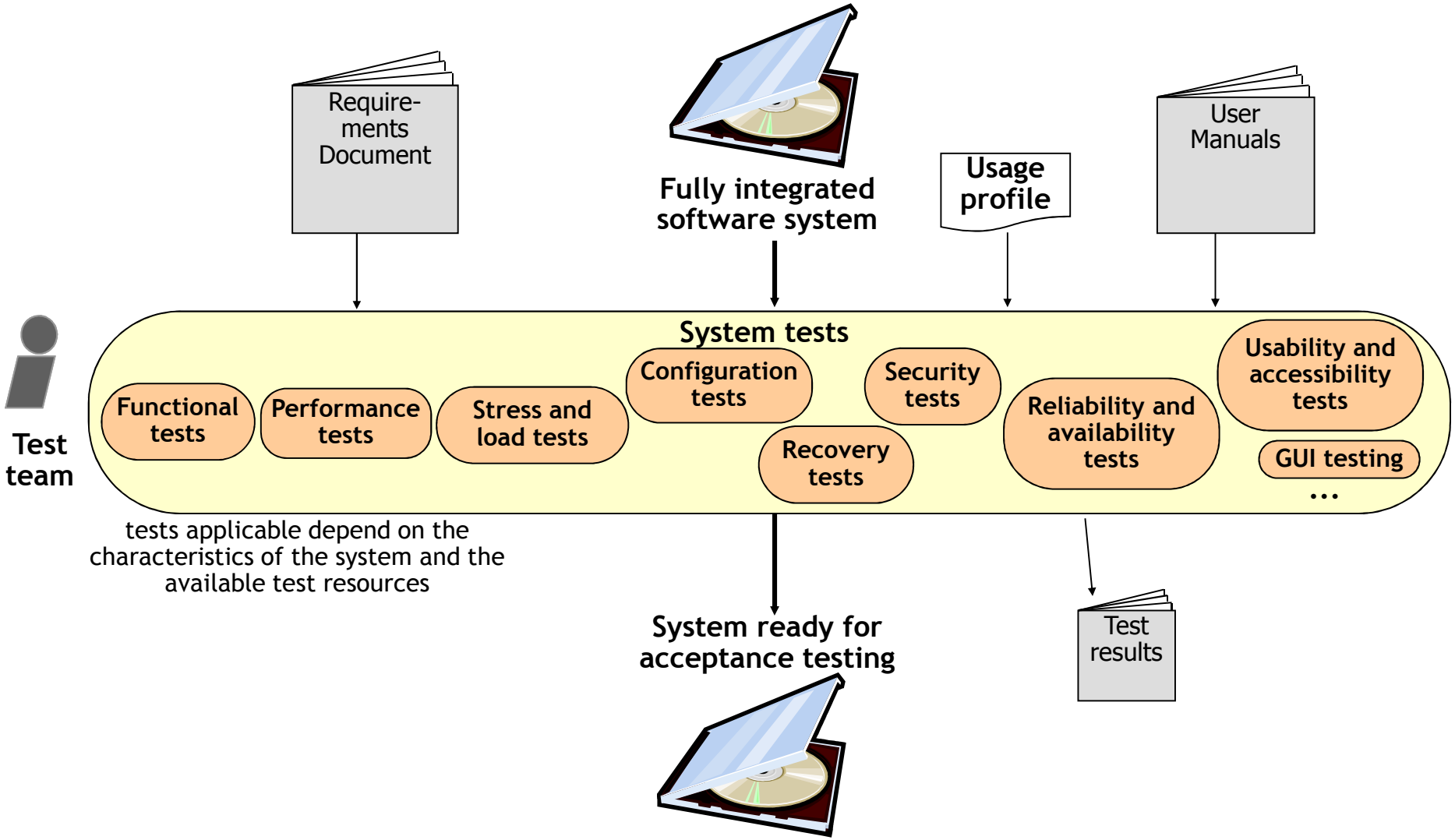
Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

System testing

- Testing the system as a whole by an independent testing team
- Often requires many resources: laboratory equipment, long test times, etc.
- Usually based on a requirements document, specifying both functional and non-functional (quality) requirements
- Preparation should begin at the requirements phase with the development of a master test plan and requirements-based tests (black-box tests)
- The goal is to ensure that the system performs according to its requirements, by evaluating both functional behavior and quality requirements such as reliability, usability, performance and security
- Especially useful for detecting external hardware and software interface defects, for example, those causing race conditions, deadlocks, problems with interrupts and exception handling, and ineffective memory usage
- Tests implemented on the parts and subsystems may be reused/repeated, and additional tests for the system as a whole may be designed

Types of system testing



Functional testing

- Ensure that the behavior of the system adheres to the requirements specification
- Black-box in nature
- Equivalence class partitioning, boundary-value analysis and state-based testing are valuable techniques
- Document and track test coverage with a (tests to requirements) traceability matrix
- A defined and documented form should be used for recording test results from functional and other system tests
- Failures should be reported in test incident reports
  - Useful for developers (together with test logs)
  - Useful for managers for progress tracking and quality assurance purposes

Performance testing

- Goals:
  - See if the software meets the performance requirements
  - See whether there are any hardware or software factors that impact on the system's performance
  - Provide valuable information to tune the system
  - Predict the system's future performance levels
- Results of performance tests should be quantified, and the corresponding environmental conditions should be recorded
- Resources usually needed
  - a source of transactions to drive the experiments, typically a **load generator**
  - an experimental **test bed** that includes hardware and software the system under test interacts with
  - instrumentation of **probes** that help to collect the performance data (event logging, counting, sampling, memory allocation counters, etc.)
  - a set of **tools** to collect, store, process and interpret data from probes

Stress and load testing

- **Load testing** – maximize the load imposed on the system (volume of data, number of users, ...)
  - Examples:
    - a system is required to handle 10 interrupts / second and the load causes 20 interrupts/second
    - a suitcase being tested for strength and endurance is stomped by a multi tonne elephant
    - testing a word processor by editing a very large document
- **Stress testing** – minimize the resources available to the system (processor, memory, disk space, ...). Stress testing often uncovers race conditions, deadlocks, depletion of resources in unusual or unplanned patterns, and upsets in normal operation that are not revealed under normal testing conditions
  - Examples:
    - run processes that consume resources (CPU, memory, disk, network) on the Web and database servers
    - take the database offline, then restart it
- The goal is to try to break the system, find the circumstances under which it will crash, and provide confidence that the system will continue to operate correctly (possibly with bad performance but with correct functional behavior) under conditions of stress
- Supported by many of the resources used for performance testing

Configuration testing

- Configuration testing checks for failure of the system to perform under all of the combinations of hw and sw configurations. Typical sw systems interact with multiple hw devices such as disc drives, tape drives, and printers.
- Objectives [Beizer]:
  - show that all the configuration changing commands and menus work properly
  - show that all interchangeable devices are really interchangeable, and that they each enter the proper states for the specified conditions
  - show that the systems' performance level is maintained when the devices are interchanged, or when they fail
- Types of test to be performed:
  - rotate and permute the positions of devices to ensure physical/logical device permutations work for each device
  - induce malfunctions in each device, to see if the system properly handles the malfunction
  - induce multiple device malfunctions to see how the system reacts

Security testing

- Evaluates system characteristics that relate to the availability, integrity and confidentiality of system data and services
- Computer software and data can be compromised by
  - **criminals intent** on doing damage, stealing data and information, causing denial of service, invading privacy
  - **errors** on the part of honest developers/maintainers (and users?) who modify, destroy, or compromise data because of misinformation, misunderstandings, and/or lack of knowledge
- Both can be perpetuated by those inside and outside on an organization
- Areas to focus: password checking, legal and illegal entry with passwords, password expiration, encryption, browsing, trap doors, viruses, ...
- Usually the responsibility of a security specialist
- See Segurança em Sistemas Informáticos (http://www.fe.up.pt/si/Disciplinas_geral.FormView?P_ANO_LECTIVO=2003/2004&P_CAD_CODIGO=EI1202&P_PERIODO=2S)

Recovery testing

- Drive a system to losses of resources in order to determine if it can recover properly from these losses
- Especially important for transaction systems
- Example: loss of a device during a transaction
- Tests would determine if the system could return to a well-known state, and that no transactions have been compromised
  - Systems with automated recovery are designed for this purpose
- Areas to focus [Beizer]:
  - **Restart** – the ability of the system to restart properly on the last checkpoint after a loss of a device
  - **Switchover** – the ability of the system to switch to a new processor, as a result of a command or a detection of a faulty processor by a monitor

Reliability and availability testing

- **Software reliability** is the probability that a software system will operate without failure under given conditions for a given interval
  - May be measured by the mean time between failures (MTBF)
  - MTBF = MTTF (mean time to failure) + MTTR (mean time to repair)
- **Software availability** is the probability that a software system will be available for use
  - May be measured by the percentage of time the system is on or **uptime** (example: 99,9%)
  - A = MTTR / MTBF
- Low reliability is compatible with high availability in case of low MTTR
- Requires statistical testing based on usage characteristics/profile
  - During testing, the system is loaded according to the usage profile
- More information: Ilene Burnstein, section 12.5
- Usually evaluated only by high maturity organizations

Usability and accessibility testing

- Usability tests are concerned with external Properties: The user's perspective:
  - **Satisfaction**: subjective view (pleasant, comfortable, intuitive, consistent)
  - **Reliable**: refers to the errors a user can do when using the system
  - **Learnability**: time taken to learn how to use the system
  - **Efficiency**: how efficient a user can be when using the system

- See also Interacção Pessoa Computador

http://www.fe.up.pt/si/Disciplinas_geral.FormView?P_ANO_LECTIVO=2003/2004&P_CAD_CODIGO=EI1108&P_PERIODO=1S

Usability and accessibility testing

- Accessibility testing is the technique of making sure that your product is accessibility compliant.
- Typical accessibility problems can be classified into following four groups, each of them with different access difficulties and issues:
  - **Visual impairments** such as blindness, low or restricted vision, or color blindness. User with visual impairments uses assistive technology software that reads content loud. User with weak vision can also make text larger with browser setting or magnificent setting of operating system
  - **Motor skills** such as the inability to use a keyboard or mouse, or to make fine movements
  - **Hearing impairments** such as reduced or total loss of hearing
  - **Cognitive abilities** such as reading difficulties, dyslexia or memory loss

GUI testing: manual techniques

- Heuristic Methods
  - A group of specialists studies the interface in order to find problems that they can identify.
- Guidelines
  - Recommendations about user interfaces. E.g.: how to organize the display and the menu structure.
- Cognitive walkthrough
  - The developers walk through the interface in the context of core tasks a typical user will need to accomplish. The actions and the feedback of the interface are compared to the user's goals and knowledge, and discrepancies between user's expectations and the steps required by the interface are noted.
- Usability tests
  - The interface is studied under real-world or controlled conditions (real users), with evaluators gathering data on problems that arise during its use.

Automated GUI testing approaches

- Capture-Replay tools
  - WinRunner, Rational Robot, Android
- Random input testing tools
  - Rational's TestFactory uses dumb monkey method

- Unit testing frameworks
  - JUnit, NUnit
- Model-based testing tools
  - Spec Explorer (API testing)
  - Spec Explorer with GUI testing extensions
  - Guitar (Atif Memon)

TQS - Teste e Qualidade de Software (*Software Testing and Quality*)

Acceptance testing

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Acceptance testing

- (1) Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system. (2) Formal testing conducted to enable a user, customer, or other authorized entity to determine whether to accept a system or component.

  [IEEE Standard Glossary of Software Engineering Terminology 610.12-1990]

- Usually the responsibility of the customer -> costumer tests
- Tests are usually based on a requirements document or a user manual
- A principal goal is to check if customer requirements and expectations are met

TQS - Teste e Qualidade de Software (*Software Testing and Quality*)

Regression testing

Ana Paiva

apaiva@fe.up.pt www.fe.up.pt/~apaiva

Regression testing

- Regression testing is not a level of testing, but it is the *retesting of software* that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes **[Source: Burstein]**
- Regression tests are especially important when multiple software releases are developed **[Source: Burstein]**
- Sometimes the execution of all tests is not feasible so there is the need to select a subset of those tests in order to reduce the time for regression testing. Some techniques to support such selection are execution trace, execution slice and test prioritization **[source: Aditya P. Mathur]**

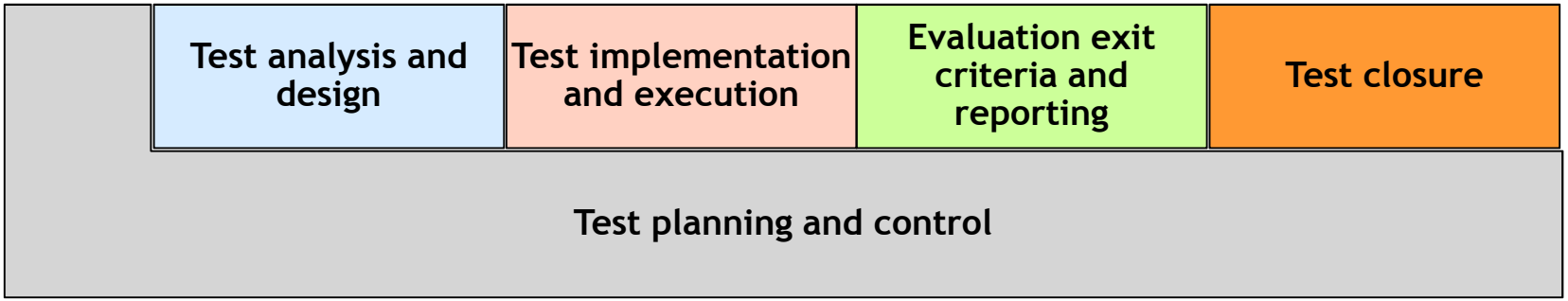
TQS - Teste e Qualidade de Software (*Software Testing and Quality*)

Test process

Ana Paiva

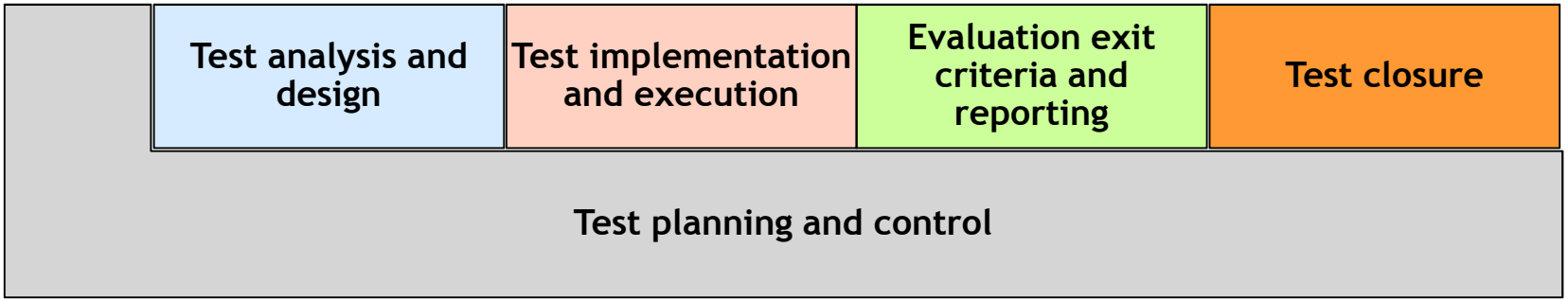
apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Test process



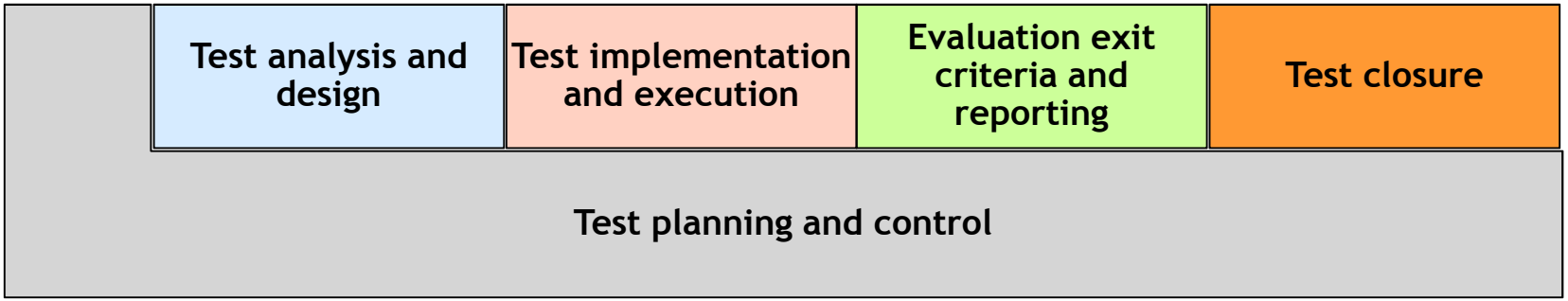
- Test planning and control
  - Verifying the mission of testing, defining the objectives of testing and the specification of test activities in order to meet the objectives and mission
- Test analysis and design
  - Activity where general testing objectives are transformed into tangible test conditions and test cases

Test process



- Test implementation and execution
  - Activity where test procedures or scripts are specified by combining the test cases in a particular order and including any other information needed for test execution, the environment is set up and tests are run
- Evaluation exit criteria and reporting
  - Activity where test execution is assessed against the defined objectives. Includes: checking test logs against exit criteria specified in test planning; assessing if more tests are needed or the exit criteria should be changed; write a summary report for stakeholders

Test process



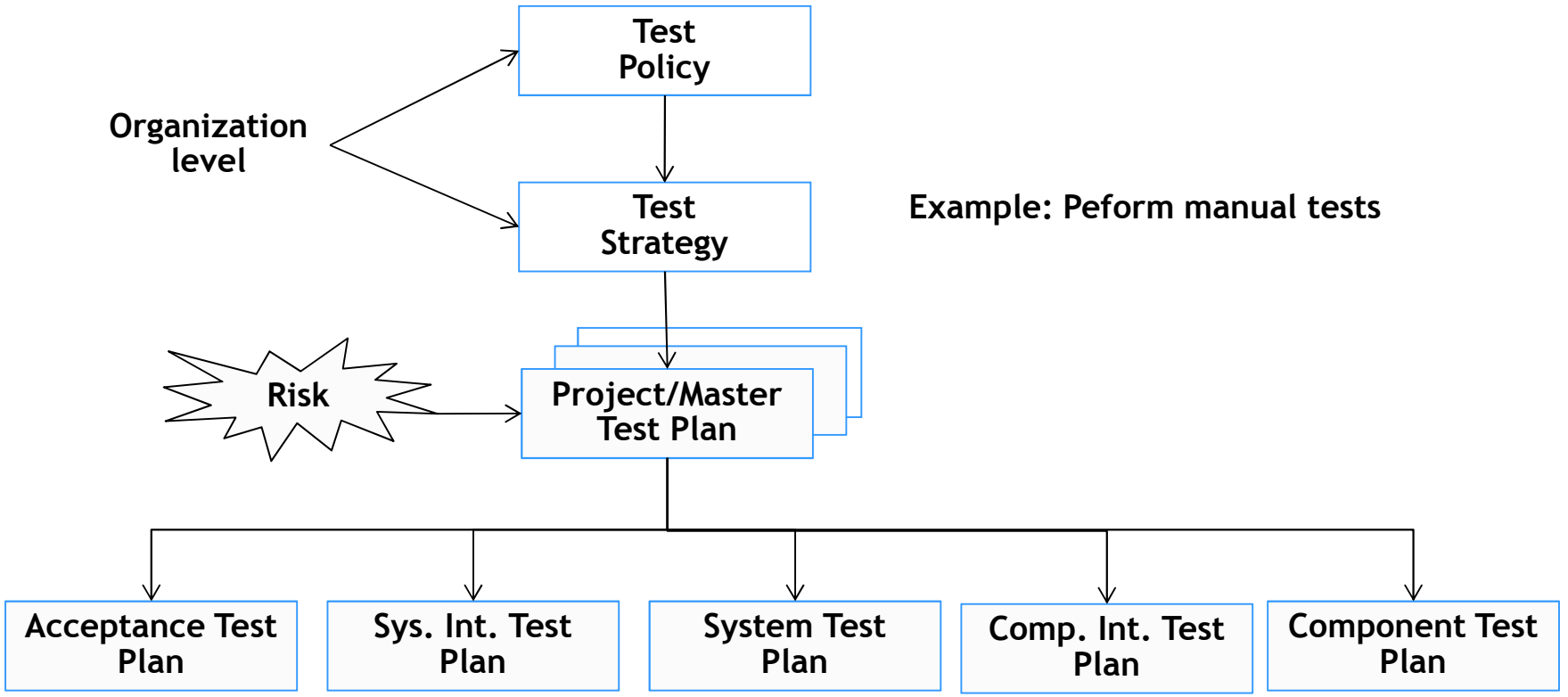
- Test closures activities
  - Collect data from completed test activities to consolidate experience, testware, facts and numbers. Include:
    - Checking which planned deliverables have been delivered, the closure of incident reports or raising of change records for any that remain open, and the documentation of the acceptance of the system
    - Finalizing and archiving testware, the test environment and the test infrastructure for later reuse
    - Handover of testware to the maintenance organization
    - Analyzing lessons learned for future releases and projects, and the improvement of test maturity

Test planning

- One of the key ways of improving the chances of a successful UAT (User acceptance testing) project is to have a good test plan in place. The IEEE 829 document standard outlines the sixteen key sections that should be in any test plan.

- **User Acceptance Testing** (UAT) is a process to obtain **confirmation by a** Subject Matter Expert (SME), preferably the owner or client of the object under test, through trial or review, that the modification or addition meets **mutually agreed-upon requirements. In software development,** UAT is **one of the final stages of a project and often occurs before a client or customer accepts the new system.**

Test planning



Test Policy

- The test policy **describes** the **organization's philosophy** toward testing (and possibly quality assurance). It is set down, either in writing or by management direction, laying out the overall objectives about testing that the organization wants to achieve. This policy may be developed by the Information Technology, Research and Development, or Product Development department, but should reflect the organizational values and goals as they relate to testing.

Test Policy

- Where a written test policy exists, it may be a short, high-level document that:
  - Provides a definition of testing, such as building confidence that the system works as intended and detecting defects.
  - Lays out a fundamental test process, e.g., test planning and control, test analysis and design, test implementation and execution, evaluating of test exit criteria and test reporting, and, test closure activities.
  - Describes how to evaluate the effectiveness and efficiency of testing, e.g., the percentage of defects to be detected (Defect Detection Percentage) and the relative cost of defects detected in testing as opposed to after release.
  - Defines desired quality targets, such as reliability (e.g., measured in term of failure rate) or usability.
  - Specifies activities for test process improvement, e.g., application of the Test Maturity Model or Test Process Improvement model, or implementation of recommendations from project retrospectives.

Test Strategy

- The test strategy describes the organization's methods of testing, including product and project risk management, the division of testing into levels, or phases, and the high-level activities associated with testing. The test strategy, and the process and activities described in it, should be consistent with the test policy. It should provide the generic test requirements for the organization or for one or more projects.
- Test strategies (also called test approaches) may be classified based on when test design begins:
  - Preventative strategies design tests early to prevent defects
  - Reactive strategies where test design comes after the software or system has been produced.

Test Strategy

- Typical strategies (or approaches) include:
  - Analytical strategies, such as risk-based testing
  - Model-based strategies, such as operational profiling
  - Methodical strategies, such as quality-characteristic based
  - Process- or standard-compliant strategies, such as IEEE 829-based
  - Dynamic and heuristic strategies, such as using bug-based attacks
  - Consultative strategies, such as user-directed testing
  - Regression testing strategies, such as extensive automation.

Project/Master Test Plan

- The master test plan describes the **application** of the test strategy **for a particular project**, including the particular levels to be carried out and the relationship among those levels. The master test plan should be consistent with the test policy and strategy, and, in specific areas where it is not, should explain those deviations and exceptions. The master test plan should complement the project plan or operations guide in that it should describe the testing effort that is part of the larger project or operation.

Project/Master Test Plan

- Typical topics for a master test plan include:
  - Items to be tested and not to be tested
  - Quality attributes to be tested and not to be tested
  - Testing schedule and budget (which should be aligned with the project or operational budget)
  - Test execution cycles and their relationship to the software release plan
  - Business justification for and value of testing
  - Relationships and deliverables among testing and other people or departments
  - Definition of what test items are in scope and out of scope for each level described
  - Specific entry criteria, continuation (suspension/resumption) criteria, and exit criteria for each level and the relationships among the levels
  - Test project risk.

Risk analysis

- **What if there isn't enough time for thorough testing?**
  Use risk analysis, along with discussion with project stakeholders, to determine where testing should be focused. Since it's rarely possible to test every possible aspect of an application, every possible combination of events, every dependency, or everything that could go wrong, risk analysis is appropriate to most software development projects. This requires judgment skills, common sense, and experience. (If warranted, formal methods are also available.)

Risk analysis

- Risk analysis considerations can include:
  - Which functionality is most important to the project's intended purpose?
  - Which functionality is most visible to the user?
  - Which functionality has the largest safety impact?
  - Which functionality has the largest financial impact on users?
  - Which aspects of the application are most important to the customer?
  - Which aspects of the application can be tested early in the development cycle?
  - Which parts of the code are most complex, and thus most subject to errors?
  - Which parts of the application were developed in rush or panic mode?
  - Which aspects of similar/related previous projects caused problems?
  - Which aspects of similar/related previous projects had large maintenance expenses?
  - Which parts of the requirements and design are unclear or poorly thought out?
  - What do the developers think are the highest-risk aspects of the application?
  - What kinds of problems would cause the worst publicity?
  - What kinds of problems would cause the most customer service complaints?
  - What kinds of tests could easily cover multiple functionalities?
  - Which tests will have the best high-risk-coverage to time-required ratio?

Test planning

- Test planning is influenced by
  - The test policy of the organization
  - The scope of the testing
  - Test objectives
  - Risk
  - Constraints
  - Criticality
  - Testability
  - Availability of resources
- Test planning is a continuous activity hence we should update the plan to reflect:
  - Changes in requirements
  - Change in risk
  - Increased knowledge of original requirements and risk
  - Results from the testing carried out

Exit criteria

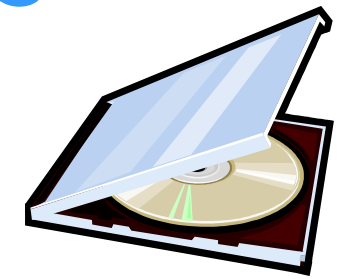
- The purpose of exit criteria is to define when to stop testing. Typically, exit test criteria may consist of:
  - Thoroughness measures, such as coverage of code, functionality or risk
  - Estimates of defect density or reliability measures
  - Cost
  - Residual risk, such as defects not fixed or lack of test coverage in certain areas
  - Schedules such as those based on time to market

Test planning documents (IEEE 829)

- Outlines of test planning documents in the "Standard for Software Test Documentation" is IEEE 829 – 1998, include templates for:
  - At the test **planning** stage the deliverable is:
    - Test Plan
  - At test **specification** stage the deliverables are:
    - Test design specifications
    - Test case specification
    - Test procedures specification
    - Test item transmittal report
  - At test **execution** stage the deliverables are:
    - Test log
    - Test incident report
    - Test summary report

TQS - Teste e Qualidade de Software (*Software Testing and Quality*)

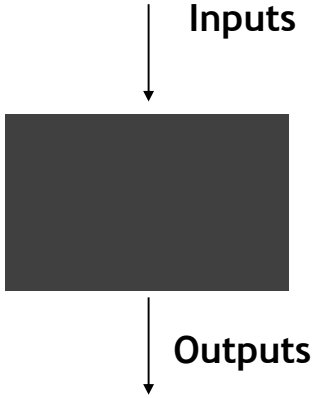
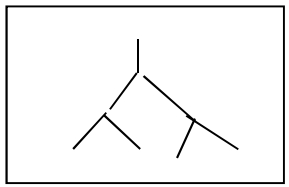
Test case design strategies



Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Test case design strategies and techniques

Strategy	Tester's View	Knowledge sources	Techniques / Methods
Black-box testing (not code-based) (sometimes called functional testing)		Requirements document Specifications User manual Models Domain knowledge Defect analysis data Intuition Experience	Equivalence class partitioning Boundary value analysis Cause effect graphing Error guessing Random testing State-transition testing Scenario-based testing
White-box testing (also called code-based or structural testing)		Program code Control flow graphs Data flow graphs Cyclomatic complexity High-level design Detailed design	Control flow testing/coverage: •Statement coverage •Branch (or decision) coverage •Condition coverage •Branch and condition coverage •Modified condition/decision coverage •Multiple condition coverage •Independent path coverage •Path coverage Data flow testing/coverage Class testing/coverage Mutation testing

(adapted from: I. Burnstein, pg.65)

TQS - Teste e Qualidade de Software (*Software Testing and Quality*)

Black box

Ana Paiva

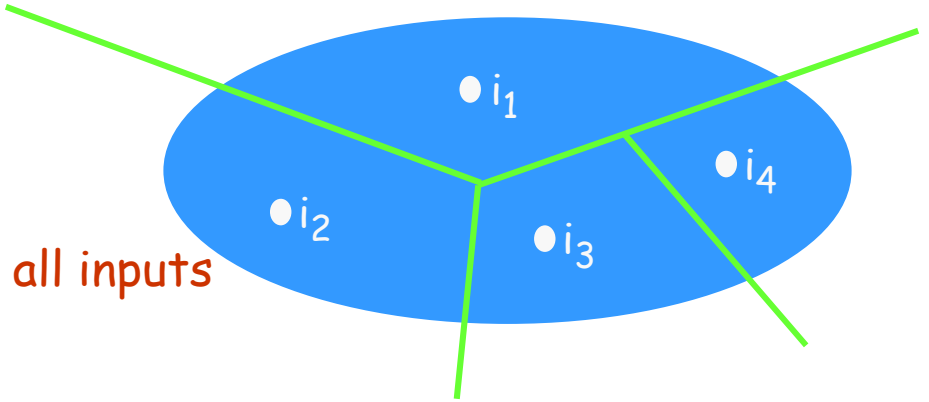
apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

Equivalence class partitioning

Divide all possible inputs into classes (partitions) such that :

- There is a finite number of input equivalence classes
- You may reasonably assume that
  - the program behaves analogously for inputs in the same class
  - one test with a representative value from a class is sufficient
  - if the representative detects a defect
    then other class members would detect the same defect

(Can also be applied to outputs)

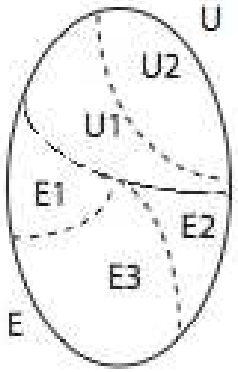


Equivalence class partitioning

Faults targeted

- The entire set of inputs to any application can be divided into at least two subsets: one containing all the expected, or legal, inputs (E) and the other containing all unexpected, or illegal, inputs (U).

- Each of the two subsets, can be further subdivided into subsets on which the application is required to behave differently (e.g., E1, E2, E3, and U1, U2).



Sistematic procedure for equivalence partitioning (1)

- **1. Identify the input domain:** Read the requirements carefully and identify all input and output variables, their types, and any conditions associated with their use.
  - Environment variables, such as class variables used in the method under test and environment variables in Unix, Windows, and other operating systems, also serve as input variables. Given the set of values each variable can assume, an approximation to the input domain is the product of these sets.
- **2. Equivalence classing:** Partition the set of values of each variable into disjoint subsets. Each subset is an equivalence class. Together, the equivalence classes based on an input variable partition the input domain. Partitioning the input domain using values of one variable, is done based on the expected behavior of the program.
  - Values for which the program is expected to behave in the "same way" are grouped together. Note that "same way" needs to be defined by the tester.

Sistematic procedure for equivalence partitioning (2)

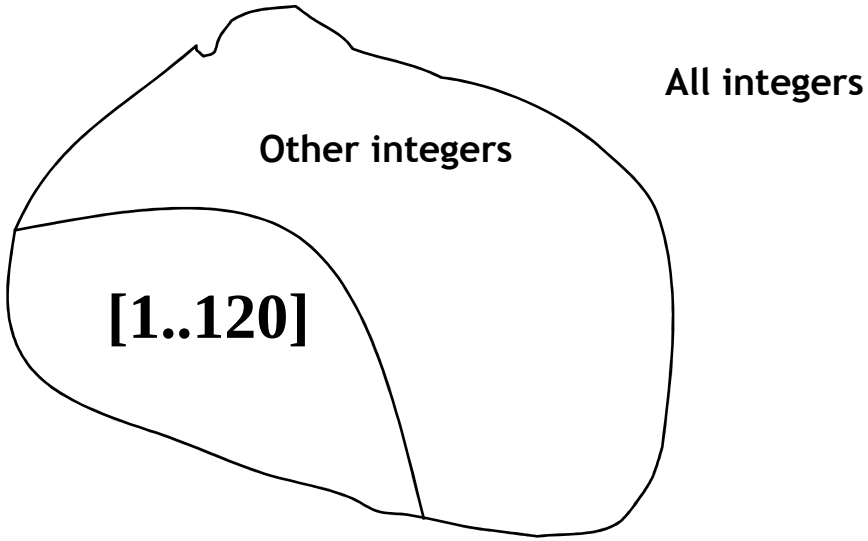
- **3. Combine equivalence classes:** This step is usually omitted and the equivalence classes defined for each variable are directly used to select test cases. However, by not combining the equivalence classes, one misses the opportunity to generate useful tests.
  - The equivalence classes are combined using the multidimensional partitioning approach described earlier.
- **4. Identify infeasible equivalence classes:** An infeasible equivalence class is one that contains a combination of input data that cannot be generated during test. Such an equivalence class might arise due to several reasons.
  - For example, suppose that an application is tested via its GUI, i.e., data is input using commands available in the GUI. The GUI might disallow invalid inputs by offering a palette of valid inputs only. There might also be constraints in the requirements that render certain equivalence infeasible.

Equivalence class partitioning

- Identify input equivalence classes
  - Based on conditions on inputs/outputs in specification/description
  - Both valid and invalid input equivalence classes
  - Based on heuristics and experience, E.g.,:
    - "input x in [1..10]" $\rightarrow$ classes: $x < 1$, $1 \leq x \leq 10$, $x > 10$
    - "enumeration A, B, C" $\rightarrow$ classes: A, B, C, not$\{A,B,C\}$
    - "input integer n" $\rightarrow$ classes: n not an integer, $n<\min$, $\min \leq n < 0$, $0 \leq n \leq \max$, $n > \max$
- Define one (or a couple of) test cases for each class
  - Test cases that cover valid classes (1 test case for 1 or more valid classes)
  - Test cases that cover at most one invalid class (1 test case for 1 invalid class)
  - Usually useful to test for 0/null/empty and other special cases
- Combine equivalent classes
  - Combine valid with invalid values to increase the capability of detecting missing code
- Identify infeasible classes

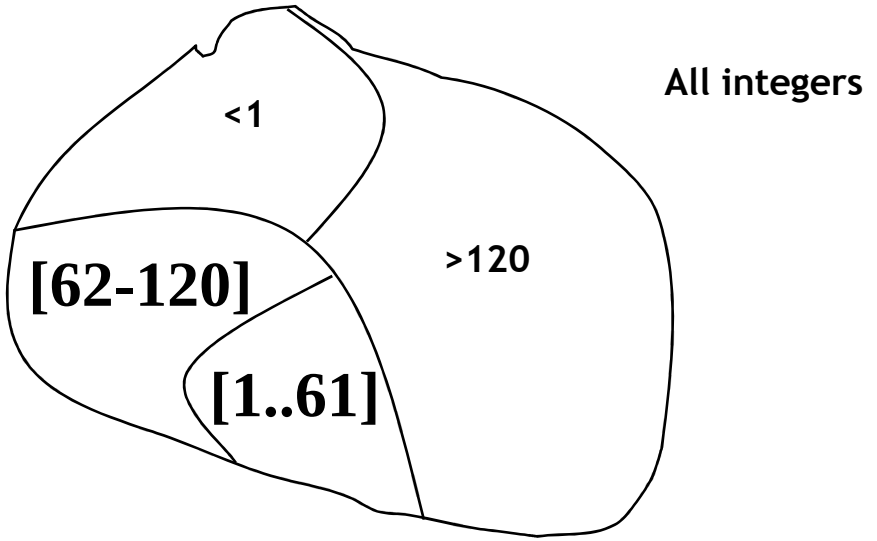
Equivalence class partitioning - Example 1

- Consider an application A that takes an integer denoted by age as input. Let us suppose that the only legal values of age are in the range [1..120]. The set of input values is now divided into a set E containing all integers in the range [1..120] and a set U containing the remaining integers.



Equivalence class partitioning - Example 1

- Further, assume that the application is required to process all values in the range [1..61] in accordance with requirement R1 and those in the range [62..120] according to requirement R2. Thus E is further subdivided into two regions depending on the expected behavior.
- Similarly, it is expected that all invalid inputs less than or equal to 1 are to be treated in one way while all greater than 120 are to be treated differently. This leads to a subdivision of U into two categories.



Equivalence class partitioning - Example 1

- Tests selected using the equivalence partitioning technique aim at targeting faults in the application under test with respect to inputs in any of the four regions, i.e., two regions containing expected inputs and two regions containing the unexpected inputs.

- It is expected that any single test selected from the range [1..61] will reveal any fault with respect to R1. Similarly, any test selected from the region [62..120] will reveal any fault with respect to R2. A similar expectation applies to the two regions containing the unexpected inputs.

Equivalence class partitioning - Example 2

Test a function that calculates the absolute value of an integer *x*

- Equivalence classes :

Criteria	Valid eq. classes	Invalid eq. classes
nr of inputs	1 (1)	0 (2) , > 1 (3)
input type	integer (4)	non-integer (5)
particular x	< 0 (6) ,	>= 0 (7)

- Test cases :

x = -10 (1,4,6)

x = 100 (1,4,7)

x = (2)

x = 10 20 (3)

x = "XYZ" (5)

Equivalence class partitioning - Example 3

- Test a program that computes the *sum* of the first *N* integers as long as this *sum* is less than *maxint*. Otherwise an error should be reported. If *N* is negative, then it takes the absolute value *N*.
- Formally:
  Given integer inputs *N* and *maxint* compute result :

$$\text{result} = \sum_{K=0}^{|N|} k \quad \text{if this <= maxint, error otherwise}$$

Equivalence class partitioning - Example 3

- Equivalence classes:

Condition	Valid eq. classes	Invalid eq. classes
nr of inputs	2	< 2, > 2
type of input	int int	int no-int, no-int int, no-int no-int
abs(*N*)	$N < 0$, $N \geq 0$	
maxint	$\sum k \leq$ maxint, $\sum k >$ maxint	

- Test Cases :

	maxint	*N*	result
Valid	100	10	55
	100	-10	55
	10	10	error
Invalid	10	-	error
	10 20	30	error
	"XYZ"	10	error
	100	9.1E4	error

Equivalence classes based on program output

- In some cases the equivalence classes are based on the output generated by the program. For example, suppose that a program outputs an integer.
- It is worth asking: “Does the program ever generate a 0? What are the maximum and minimum possible values of the output?”
- These two questions lead to the two following equivalence classes based on outputs:
  - E1: Output value v is 0.
  - E2: Output value v is the maximum possible.
  - E3: Output value v is the minimum possible.
  - E4: All other output values.
- Based on the output equivalence classes one may now derive equivalence classes for the inputs. Thus each of the four classes given above might lead to one equivalence class consisting of inputs.

Equivalence classes for variables: range

Eq. Classes	Example	
	Constraints	Classes
One class with values inside the range and two with values outside the range.	**speed ∈[60..90]**	**{50}, {75}, {92}**
	area: float area≥0.0	**{{-1.0}, {15.52}}**
	age: int	**{{-1}, {56}, {132}}**
	letter: char	**{{J}, {3}}**

Equivalence classes for variables: strings

Eq. Classes	Example	
	Constraints	**Classes**
At least one containing all legal strings and one all illegal strings based on any constraints.	firstname: string	{{ε}, {Sue}, {Loooong Name}}

Equivalence classes for variables: enumeration

Eq. Classes	Example	
	Constraints	**Classes**
Each value in a separate class	autocolor:{red, blue, green}	{{red,} {blue}, {green}}
	up:boolean	{{true}, {false}}

Equivalence classes for variables: arrays

Eq. Classes	Example	
	Constraints	**Classes**
One class containing all legal arrays, one containing the empty array, and one containing a larger than expected array.	int [ ] aName: new int[3];	{[ ]}, {[-10, 20]}, {[-9, 0, 12, 15]}

Equivalence classes for variables: compound data types

- Arrays in Java and records, or structures, in C++, are compound types. Such input types may arise while testing components of an application such as a function or an object.
- While generating equivalence classes for such inputs, one must consider legal and illegal values for each component of the structure.

```
struct transcript{
        string fName; // First name.
        string lName; //  Last name.
        string cTitle [200]; // Course titles.
        char grades [200]; // Letter grades corresponding
                           to course titles.
}
```

Partitioning

- Unidimensional partitioning
  - One way to partition the input domain is to consider one input variable at a time. Thus each input variable leads to a partition of the input domain. We refer to this style of partitioning as unidimensional equivalence partitioning or simply unidimensional partitioning.

  This type of partitioning is commonly used.

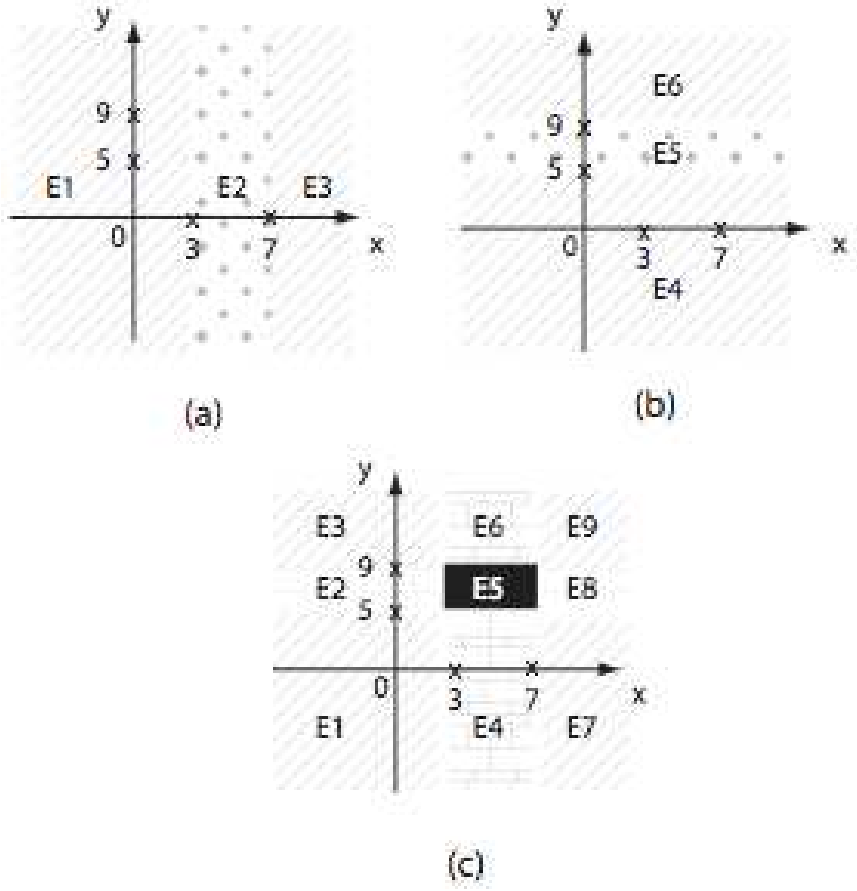
- Multidimensional partitioning
  - Another way is to consider the input domain *I* as the set product of the input variables and define a relation on *I*. This procedure creates one partition consisting of several equivalence classes. We refer to this method as multidimensional equivalence partitioning or simply multidimensional partitioning.
  - Multidimensional partitioning leads to a large number of equivalence classes that are difficult to manage manually. Many classes so created might be infeasible. Nevertheless, equivalence classes so created offer an increased variety of tests as is illustrated in the next section.

Partitioning Example (1)

- Consider an application that requires two integer inputs *x* and *y*. Each of these inputs is expected to lie in the following ranges: $3 \leq x \leq 7$ and $5 \leq y \leq 9$.
- For unidimensional partitioning we apply the partitioning guidelines to *x* and *y* individually. This leads to the following six equivalence classes.
- E1: $x<3$ E2: $3 \leq x \leq 7$ E3: $x>7$ (y ignored)
- E4: $y<5$ E5: $5 \leq y \leq 9$ E6: $y>9$ (x ignored)
- For multidimensional partitioning we consider the input domain to be the set product X x Y. This leads to 9 equivalence classes.

Partitioning Example (2)

- 9 equivalent classes
  - E1: x<3, y<5
  - E3: x<3, y>9
  - E2: x<3, 5≤y≤9
  - E4: 3≤x≤7, y<5
  - E5: 3≤x≤7, 5≤y≤9
  - E6: 3≤x≤7, y>9
  - E7: x>7, y<5
  - E8: x>7, 5≤y≤9
  - E9: x>7, y>9



Boundary value analysis

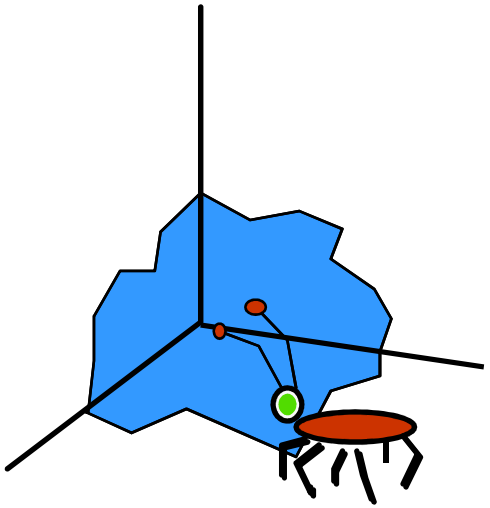
Based on experience / heuristics :

- Testing boundary conditions of equivalence classes is more effective, i.e., values directly on, above, and beneath edges of classes
  - If a system behaves correctly at boundary values, than it probably will work correctly at "middle" values
- Choose input boundary values as tests in input classes instead of, or additional to arbitrary values
- Choose also inputs that invoke output boundary values (values on the boundary of output classes)
- Example strategy as extension of equivalence class partitioning:
  - choose one (or more) arbitrary value(s) in each equivalent class
  - choose values exactly on lower and upper boundaries of equivalent classes
  - choose values immediately below and above each boundary (if applicable)

Boundary value analysis

“Bugs lurk in corners and congregate at boundaries.”

[Boris Beizer, "Software testing techniques"]



Boundary value analysis
Example 1

Test a function for calculation of absolute value of an integer

- Valid equivalence classes :

Condition	Valid eq. classes	Invalid eq. Classes
particular abs	< 0, >= 0	

- Test cases :

  class x < 0, arbitrary value: x = -10

  class x >= 0, arbitrary value x = 100

  classes x < 0, x >= 0, on boundary : x = 0

  classes x < 0, x >= 0, below and above: x = -1, x = 1

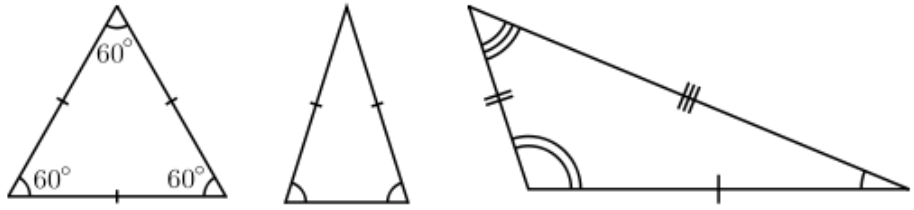
Boundary value analysis
A self-assessment test 1 [Myers]

- "A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene (all lengths are different), isosceles (two lengths are equal), or equilateral (all lengths are equal)."

Write a set of test cases to test this program.

Inputs: l_1, l_2, l_3 , integer, $l_i > 0$, $l_i < l_j + l_k$

Output: error, scalene, isosceles or equilateral

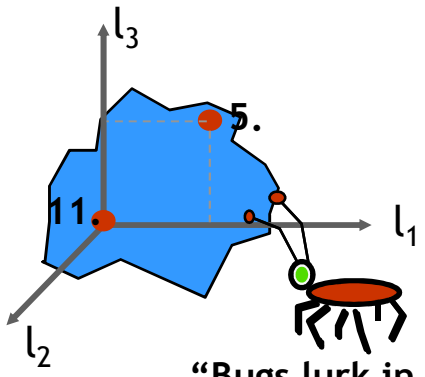


Boundary value analysis
A self-assessment test 1 [Myers]

Test cases for:

valid inputs:

1. valid scalene triangle ?
2. valid equilateral triangle ?
3. valid isosceles triangle ?
4. 3 permutations of previous ?



"Bugs lurk in corners and congregate at boundaries."

invalid inputs:

5. side = 0 ? (boundary "plane")
6. negative side ?
7. one side is sum of others ? (boundary)
8. 3 permutations of previous ?
9. one side larger than sum of others ?
10. 3 permutations of previous ?
11. all sides = 0 ? (boundary "corner")
12. non-integer input ?
13. wrong number of values ?

Boundary value analysis
Example 2

- Given inputs maxint and *N* compute result :

$$\text{result} = \sum_{K=0}^{|N|} k \quad \text{if this <= maxint, error otherwise}$$

- Valid equivalence classes :

condition	valid eq. classes	boundary values.
abs(*N*)	$N < 0$, $N \geq 0$	N = (-2), -1, 0, 1
maxint	$\sum k \leq$ maxint,	$\sum k$ = maxint-1,
	$\sum k >$ maxint	maxint,
		maxint+1,
		(maxint+2)

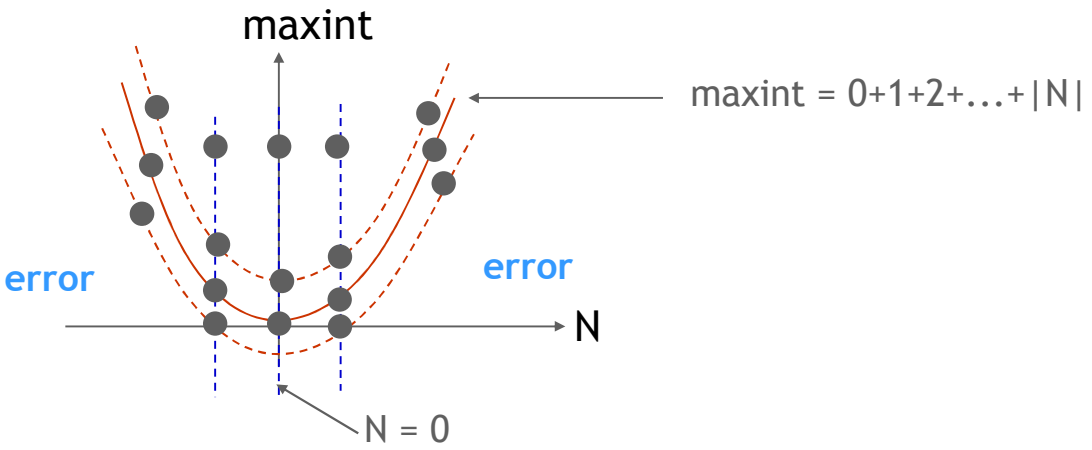
Boundary value analysis
Example 2

- Test Cases :

maxint	*N*	result
55	10	55
54	10	error
56	10	55
0	0	0

maxint	*N*	result
100	0	0
100	-1	1
100	1	1
…	…	…

- How to combine the boundary conditions of different inputs ?
  Take all possible boundary combinations ? This may blow up ……



Boundary value analysis
Example 3: search routine specification

procedure Search (Key : ELEM ; T: ELEM_ARRAY;
Found : **out** BOOLEAN; L: **out** ELEM_INDEX) ;

Pre-condition

-- the array has at least one element
T'FIRST <= T'LAST

Post-condition

-- the element is found and is referenced by L
(Found and T (L) = Key)

or

-- the element is not in the array
(**not** Found **and**
not (**exists** i, T'FIRST >= i <= T'LAST, T (i) = Key))

(source: Ian Sommerville)

Boundary value analysis
Example 3 - input partitions

- P1 - Inputs which conform to the pre-conditions (valid)
  - array with 1 value (boundary)
  - array with more than one value (different size from test case to test case)
- P2 - Inputs where a pre-condition does not hold (invalid)
  - array with zero length
- P3 - Inputs where the key element is a member of the array
  - first, last and middle positions in different test cases
- P4 - Inputs where the key element is not a member of the array

Boundary value analysis

Example 3 - test cases (valid cases only)

Array	**Element**
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

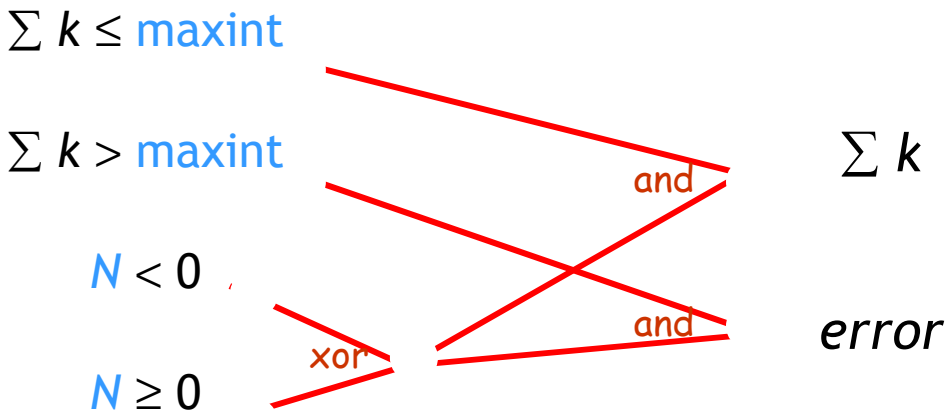
Cause-effect graphing

- Black-box technique to analyze combinations of input conditions
- Identify causes and effects in specification

  causes → inputs / initial state conditions

  effects → outputs / final state conditions
- Make Boolean Graph linking causes and effects
- Annotate impossible combinations of causes and effects
- Develop decision table from graph with in each column a particular combination of inputs and outputs
- Transform each column into test case

Cause-effect graphing
Example 2



Decision table

("truth table")

Each entry in de decision table is a 0 or a 1 depending on whether or not the corresponding condition is false or true

causes	∑ k ≤ maxint	1	1	0	0
(inputs)	∑ k > maxint	0	0	1	1
	N < 0	1	0	1	0
	N ≥ 0	0	1	0	1
effects	∑ k	1	1	0	0
(outputs)	*error*	0	0	1	1

Cause-effect graphing

- Systematic method for generating test cases representing combinations of conditions
- Differently from eq. class partitioning, we define a test case for each possible combination of conditions
- Combinatorial explosion of number of possible combinations
  - In the worst case, if *n* causes are related to an effect *e*, then the maximum number of combinations that bring *e* to a *1-state* is 2^n.

Error guessing

- Just ‘guess’ where the errors are ……
- Intuition and experience of tester
- Ad hoc, not really a technique
- But can be quite effective
- Strategy:
  - Make a list of possible errors or error-prone situations (often related to boundary conditions)
  - Write test cases based on this list

Risk based testing

- **Risk-based testing** (RBT) is a type of software testing that prioritizes the features and functions to be tested based on priority/importance and likelihood or impact of failure. More sophisticated 'error guessing'
- Try to identify critical parts of program (high risk code sections):
  - parts with unclear specifications
  - developed by junior programmer while his wife was pregnant ……
  - complex code :
    measure code complexity - tools available (McGabe, Logiscope,…)
- High-risk code will be more thoroughly tested
  (or be rewritten immediately ……)

Testing for race conditions

- Also called bad timing and concurrency problems
- Problems that occur in multitasking systems (with multiple threads or processes)
- A kind of boundary analysis related to the dynamic views of a system (state-transition view and process-communication view)
- Examples of situations that may expose race conditions:
  - problems with shared resources:
    - saving and loading the same document at the same time with different programs
    - sharing the same printer, communications port or other peripheral
    - using different programs (or instances of a program) to simultaneously access a common database
  - problems with interruptions:
    - pressing keys or sending mouse clicks while the software is loading or changing states
  - other problems:
    - shutting down or starting two or more instances of the software at the same time
- Knowledge used: dynamic models (state-transition models, process models)

[source: Ron Patton]

Random testing

- Input values are randomly generated
- Do you know that a monkey using a piano keyboard could play a Vivaldi opera? Could the same monkey, using your application, discovery defects?
- Two kinds of tools
  - **Dumb monkeys** – low IQ; they can't recognize an error when they see one
  - **Smart monkeys** – generate inputs with some knowledge to reflect expected usage; get knowledge from state table or model of the AUT.
- Microsoft says that 10 to 20% of the bugs in Microsoft projects are found by these tools

Random testing

- Advantages
  - Good for finding system crashes
  - Particularly adequate for performance testing (it's not necessary to check the correctness of outputs)
  - No effort in generating test cases
  - Independent of updates
  - Increase confidence on the software when running several hours without finding errors
  - “Easy” to implement
- Disadvantages
  - Not good for finding other kinds of errors
  - Difficult to reproduce the errors (repeat test cases / sequence of inputs)
  - Unpredictable
  - May not cover special cases that are discovered by "manual" techniques

Deriving test cases from requirements and use cases

- Particularly adequate for system and acceptance testing
- From requirements:
  - You have a list of requirements
  - Define at least one test case for each requirement
  - Build and maintain a (tests to requirements) traceability matrix
- From use cases:
  - You have use cases that capture functional requirements
  - Each use case is described by one or more *normal* flow of events and zero or more *exceptional* flow of events
  - Define at least one test case for each flow of events (also called *scenario*)
  - Build and maintain a (tests to use cases) traceability matrix

State-transition testing

- Construct a state-transition model (state machine view) of the item to be tested (from the perspective of a user/client). E.g., with a state diagram in UML
- Define test cases to exercise all states and all transitions between states
  - Usually, not all possible paths (sequences of states and transitions), because of combinatorial explosion
  - Each test case describes a sequence of inputs and outputs (including input and output states), and may cover several states and transitions
  - Also test to fail – with unexpected inputs for a particular state
- We will talk about this techniques in more detail in the following lectures

Black box testing: Which One ?

- Black box testing techniques :
  - Equivalence partitioning
  - Boundary value analysis
  - Cause-effect graphing
  - Error guessing
  - ……
- Which one to use ?
  - None of them is complete
  - All are based on some kind of heuristics
  - They are complementary

Black box testing: which one ?

- Always use a combination of techniques
  - When a formal specification is available try to use it
  - Identify valid and invalid input equivalence classes
  - Identify output equivalence classes
  - Apply boundary value analysis on valid equivalence classes
  - Guess about possible errors
  - Cause-effect graphing for linking inputs and outputs

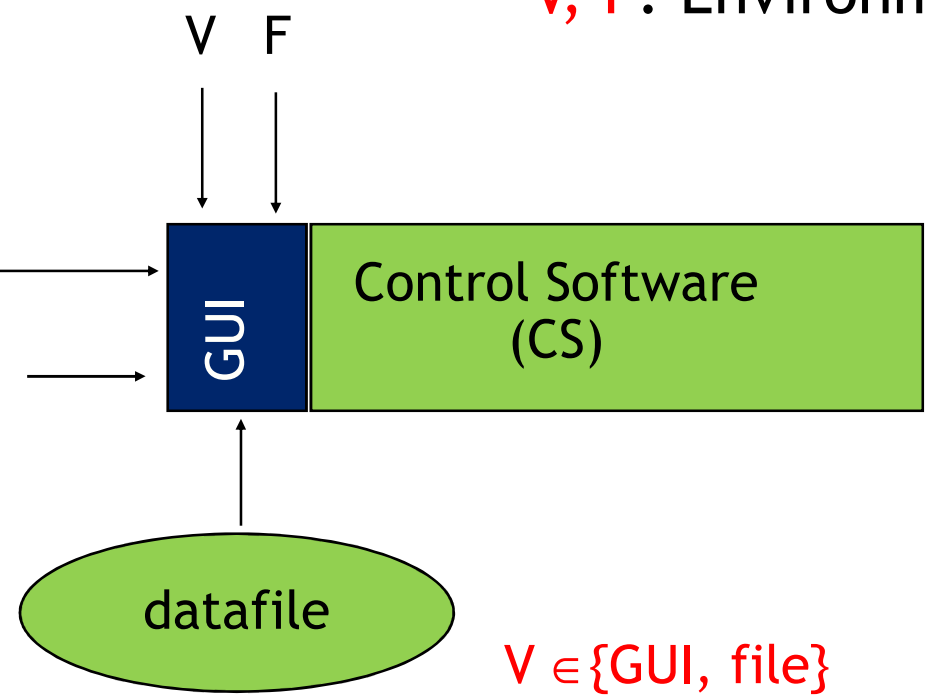
Exercise

- The control software of BCS, abbreviated as CS, allows a human operator to give one of three commands (cmd): change the boiler temperature (temp), shut down the boiler (shut), and cancel the request (cancel).
- Command temp causes CS to ask the operator to enter the amount by which the temperature is to be changed (tempch). Values of tempch are in the range [-10..10] in increments of 5 degrees Fahrenheit. An temperature change of 0 is not an option.
- BCS examines variable V. If V is set to GUI, the operator is asked to enter one of the three commands via a GUI. However, if V is set to file, BCS obtains the command from a command file.
- The command file may contain any one of the three commands, together with the value of the temperature to be changed if the command is temp. The file name is obtained from variable F.

Exercise

V, F: Environment variables

cmd: command
(temp, shut, cancel)



tempch: desired
temperature change
(-10..10)

V ∈{GUI, file}

F: file name if V is set to “file.”

Exercise

- Identify input domain
- Identify equivalence classes
- Combine equivalence classes
- Discard infeasible equivalence classes
- Generate sample tests for BCS from the remaining feasible equivalence classes

Identify input domain

Variable	Kind	Type	Value(s)
V	Environment	Enumerated	{GUI, file}
F	Environment	String	A file name
cmd	Input via GUI or file	Enumerated	{temp,cancel,shut}
tempch	Input via GUI or file	Enumerated	{-10,-5,5,10}

S = V x F x cmd x tempch

Equivalence class partition

Variable	Partition
V	{{GUI},{file},{undefined}}
F	f_valid, f_invalid
cmd	{{temp},{cancel},{shut},{c_invalid}}
tempch	{{-10,-5,5,10},{t_invalid}}

Combine equivalence classes

- Variables V, F, cmd and tempch have been partitioned into 3, 2, 4 and 2 susets, respectively
- Set products of these four variables leads to a total of 3x2x4x5=120

Discard infeasable classes

- The amount by which the boiler temperature is to be changed is needed only when the operator selects temp for cmd, Thus all equivalent classes that match the following template are infeasable
  - {(V,F,{cancel,shut,c_invalid},t_valid U t_invalid)} = 3x2x3x5 = 90
- GUI does not allow invalid values of temperature change to be input. Two more equivalente classes infeasable
  - {(GUI,f_valid,temp,t_invalid)} and {(GUI,f_invalid,temp,t_invalid)}
- Carefully designed application might not ask for the values of cmd and tempch when V=file and F contains a file name that does not exist
  - {(file, f_invalid, temp, t_valid U t_invalid)}
- Application will not allow values of cmd and tempch to e input when V is undefined
  - {(undefined, -, temp, t_valid U t_invalid)}
- Discard 90+2+5+5=102 equivalence classes

Result

{(GUI,f_valid,temp,t_valid)} = 4

{(GUI,f_invalid,temp,t_valid)} = 4

{(GUI, -, cancel, NA)} = 2

{(file, f_valid,temp,t_valid U t_invalid)} = 5

{(file,f_valid,shut,NA)} = 1

{(file,f_invalid,NA,NA)} = 1

{(undefined,NA,NA,NA)} = 1

total = 18

- means that data can be input but is not used by the software

NA means that data cannot be input to the control software

TQS - Teste e Qualidade de Software (*Software Testing and Quality*)

White box

Ana Paiva

apaiva@fe.up.pt **www.fe.up.pt/~apaiva**

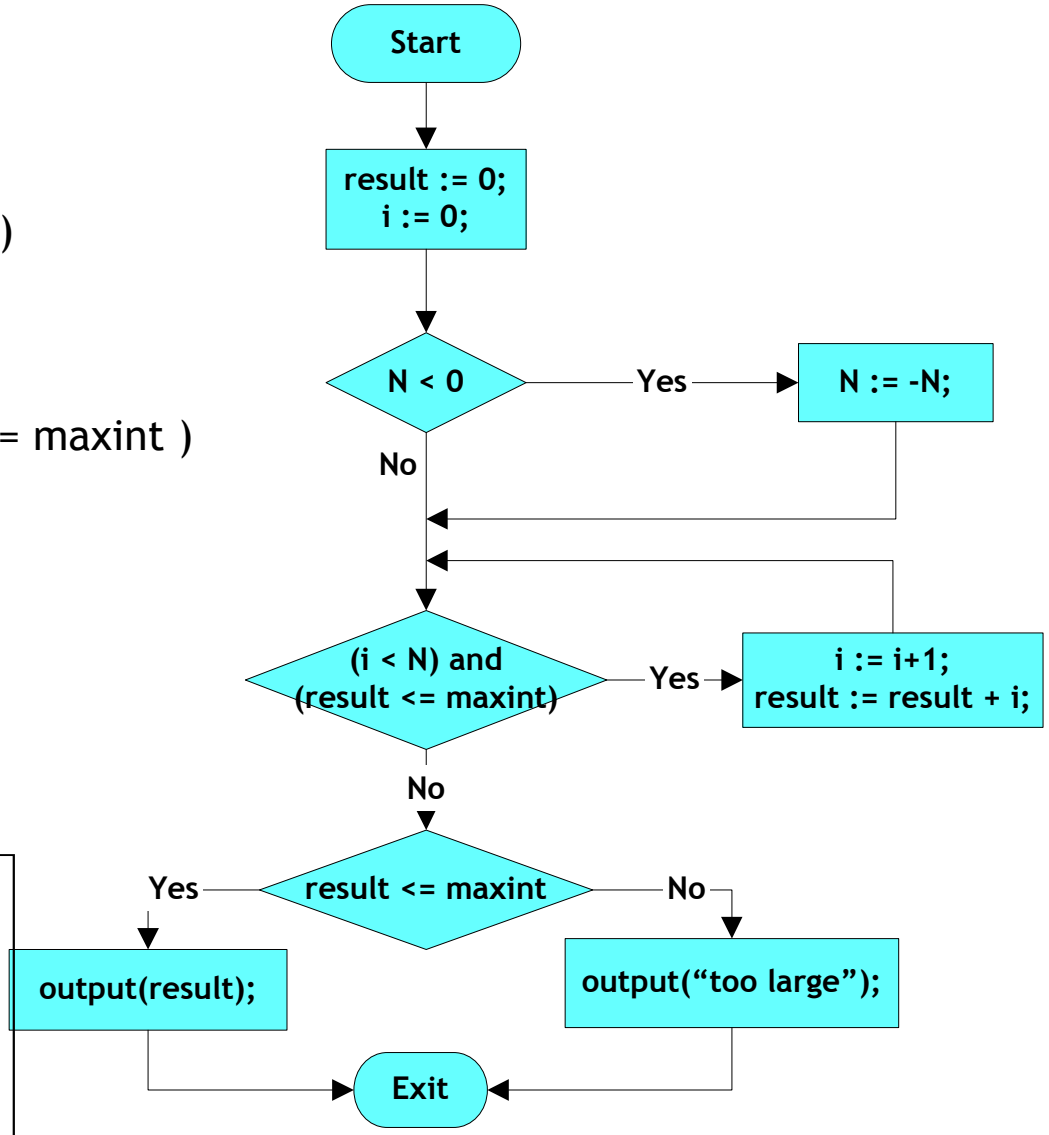
White-box testing

- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases
- Used mainly for unit testing
- Programming language dependent
- Extent to which (source) code is executed, i.e., covered
- Different kind of coverage :
  - based on control flow analysis - statement, decision, condition, decision and condition, MC/DC, path, ...
  - based on data flow analysis

Control flow analysis
Example

```
1   PROGRAM  sum ( maxint, N : INT )
2     INT    result := 0 ;   i := 0 ;
3     IF   N < 0
4     THEN   N  :=  - N ;
5     WHILE   ( i < N )  AND   ( result <= maxint )
6     DO   i  :=  i + 1 ;
7            result  :=  result + i ;
8     OD;
9     IF   result <= maxint
10    THEN   OUTPUT ( result )
11    ELSE   OUTPUT ( “too large” )
12  END.
```

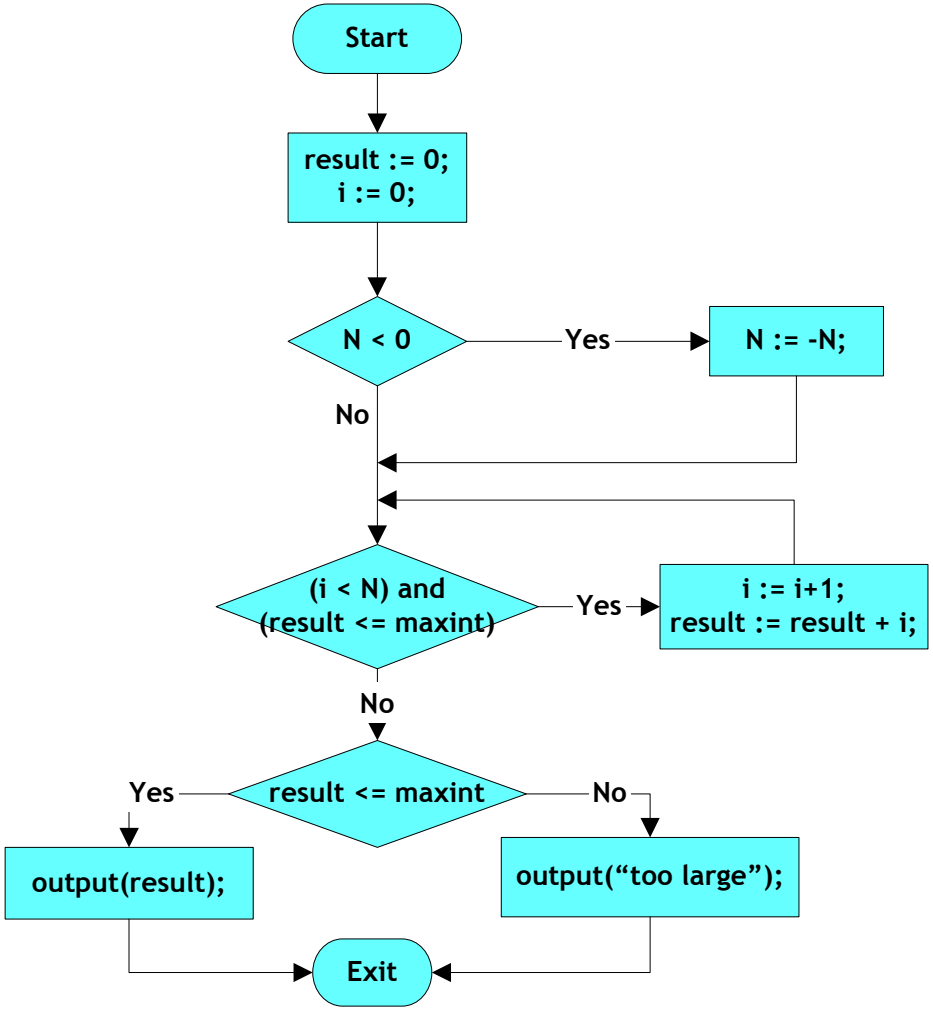
$$result = \begin{cases} \sum_{K=0}^{|N|} k & \text{, if this <= maxint} \\ error, & \text{otherwise} \end{cases}$$



Statement coverage

- Execute (exercise) every statement of a program
  - Generate a set of test cases such that each statement of the program is executed at least once
- Weakest white-box criterion
- Analysis supported by many commercial and freeware tools (test coverage or code coverage tools)
  - Standard Unix tool: tcov
  - A listing indicates how often each statement was executed and the percentage of statements executed
- Note: in case of unreachable statements, statement coverage is not possible

Example : statement coverage



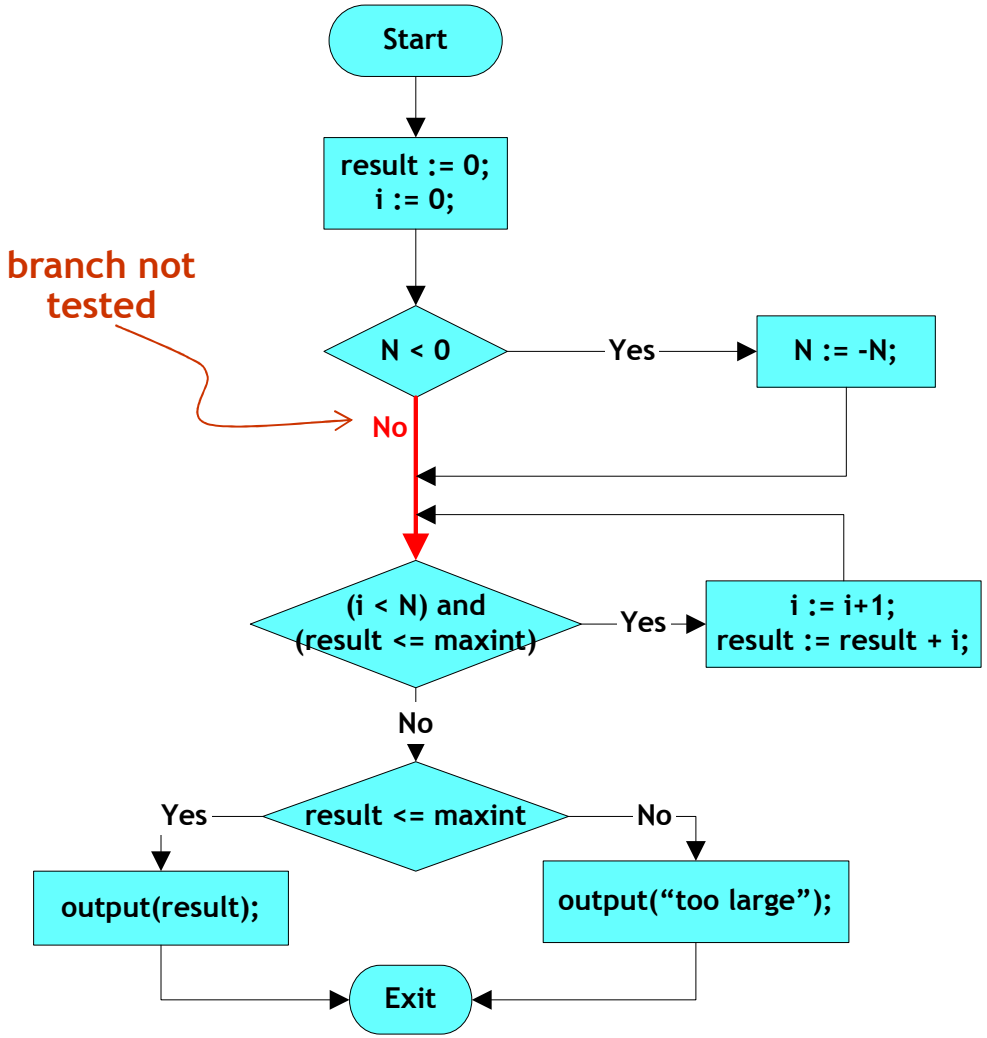
Tests for complete statement (node) coverage:

inputs		outputs
maxint	N	result
10	-1	1
0	-1	too large

Decision (or branch) coverage

- **Execute every branch of a program :**
  each possible outcome of each decision occurs at least once
- Example:
  - simple decision: IF b THEN s1 ELSE s2
    - b should be tested for true and false
  - multiple decision:
    CASE x OF
    1 : ….
    2 : ….
    3 : ….
- Stronger than statement coverage
  - IF THEN without ELSE – if the condition is always true all the statements are executed, but branch coverage is not achieved (infeasibility)

Example : decision (or branch) coverage



Tests for complete statement (node) coverage:

inputs		outputs
maxint	N	result
10	-1	1
0	-1	too large

are not sufficient for decision (branch) coverage!

Take:

inputs		outputs
maxint	N	result
10	3	6
0	-1	too large

for complete decision (branch) coverage

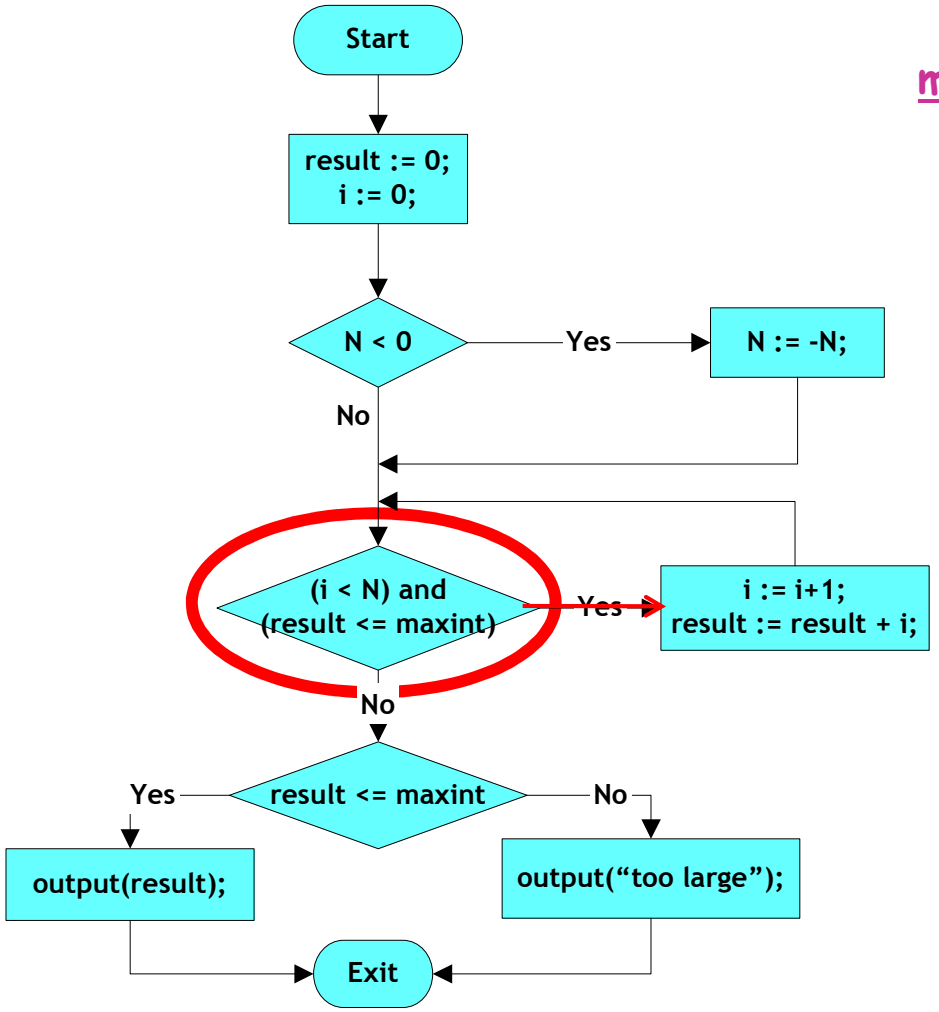
Condition coverage

- Design test cases such that **each possible outcome** of each condition in a decision (composite condition) **occurs at least once**

- Example:
  - **decision** (i < N) AND (result <= maxint)
    consists of **two** conditions : (i < N) , (result <= maxint)
  - test cases should be designed such that each condition gets value **true** and **false** at least once

- Last test cases of previous slides already guarantee condition (and branch) coverage

Condition and decision (or condition / decision) coverage



Test cases:

maxint	N	i	result	i<N	result<=maxint
-1	1	0	0	true	false
1	0	0	0	false	true

give condition coverage for all conditions

But don't preserve branch coverage

⇓

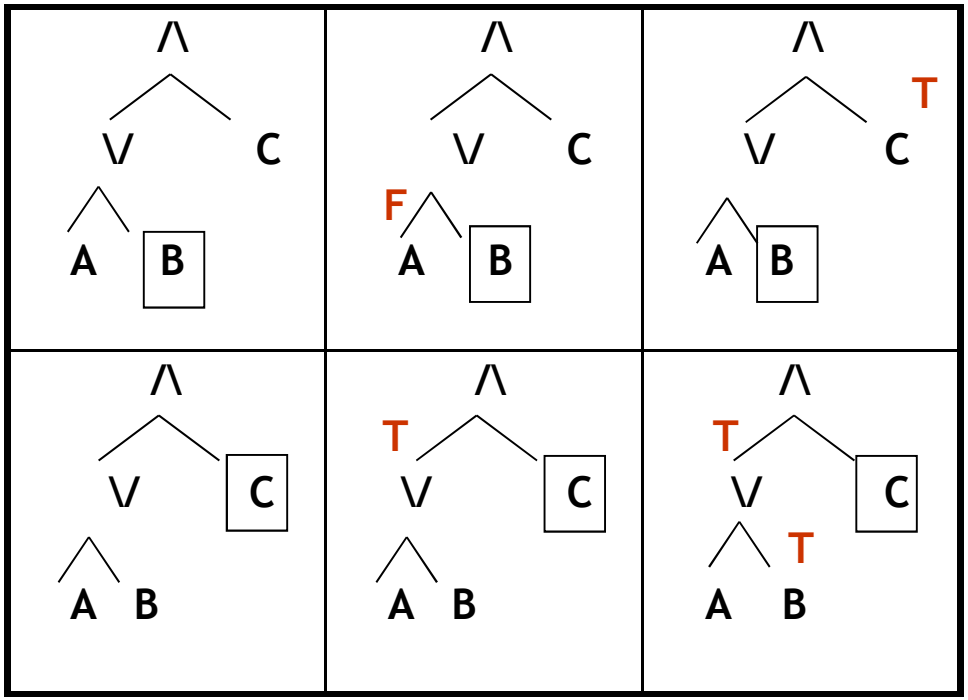
always take care that condition coverage preserves branch coverage : condition and decision coverage

Modified Condition/Decision Coverage

- Also known as MC/DC or MCDC
- Design test cases such that
  - every decision in the program has taken **all possible outcomes at least once** (decision coverage)
  - every condition in a decision in the program has taken **all possible outcomes** at least once (condition coverage)
  - every condition in a decision has been shown to **independently affect that decision's outcome**; a condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions
    - **condition** – a Boolean expression containing no Boolean operators
    - **decision** – a Boolean expression composed of conditions and zero or more Boolean operators
- Created at Boeing, required for level A (critical) software for the Federal Aviation Administration (FAA) in the USA by RCTA/DO-178B

Modified Condition/Decision Coverage

(A \/ B) /\ C



Modified Condition/Decision Coverage

- Test cases required to meet the MC/DC criteria

Test case	Conditions		Decision
	A	B	A **and** B
1	True	True	True
2	False	True	False
3	True	False	False

Test case	Conditions		Decision
	A	B	A **or** B
1	False	False	False
2	True	False	True
3	False	True	True

Test case	Condition	Decision
	A	**not** A
1	True	False
2	False	True

Test case	Conditions		Decision
	A	B	A **xor** B
1	True	True	False
2	False	True	True
3	True	False	True

(or another combination)

Modified Condition/Decision Coverage

- Consider the following fragment of code:

```
if
    A or (B and C)
then
    do_something;
else
    do_something_else;
end if;
```

- MC/DC may be achieved with the following set of test inputs (note that there are alternative sets of test inputs, which will also achieve MC/DC):

Case	A	B	C	Outcome
1	FALSE	FALSE	TRUE	FALSE
2	TRUE	FALSE	TRUE	TRUE
3	FALSE	TRUE	TRUE	TRUE
4	FALSE	TRUE	FALSE	FALSE

are not evaluated if logical operators short-circuit

cases 2 a 4 are sufficient for branch and condition coverage, but only if logical operators do not short-circuit

- Because:
  - A is shown to independently affect the outcome of the decision condition by case 1 and case 2
  - B is shown to independently affect the outcome of the decision condition by case 1 and case 3
  - C is shown to independently affect the outcome of the decision condition by case 3 and case 4

Multiple condition coverage

- Design test cases for each combination of conditions
- Example:
  - (i < N) (result <= maxint)

    false false

    false true

    true false

    true true
- Implies decision, condition, decision and condition, modified branch/condition coverage
- But : exponential blow-up ($2^{\text{number of conditions}}$)
- Again : some combinations may be infeasible

Path coverage

- Execute every possible path of a program, i.e., every possible sequence of statements
- Strongest white-box criterion (based on control flow analysis)
- Usually impossible: infinitely many paths (in case of loops)
- So: not a realistic option
- But note : enormous reduction w.r.t. all possible test cases (each sequence of statements executed for only one value) (doesn't mean exhaustive testing)

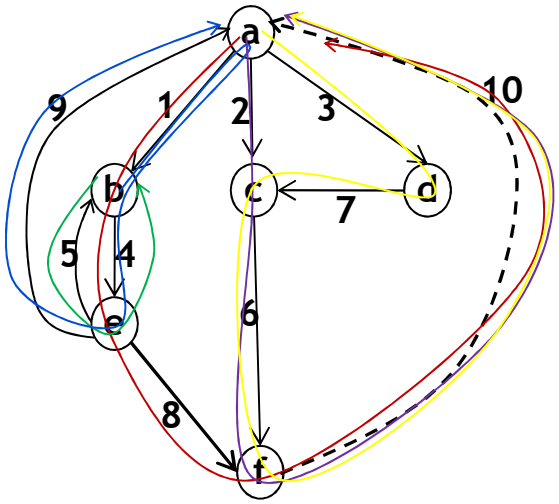
Independent path (or basis path) coverage

- Obtain a maximal set of linearly independent paths (also called a basis of independent paths)
  - If each path is represented as a vector with the number of times that each edge of the control flow graph is traversed, the paths are linearly independent if it is not possible to express one of them as a linear combination of the others
- Generate a test case for each independent path
- The number of linearly independent paths is given by the McCabe's *cyclomatic complexity* of the program
  - Number of edges - Number of nodes + 2 in the control flow graph
  - Measures the structural complexity of the program

Independent path (or basis path) coverage

- Problem: some paths may be impossible to execute
- Also called structured testing (see McCabe for details)
- McCabe's argument: this approach produces a number of test cases that is proportional to the complexity of the program (as measured by the cyclomatic complexity), which, in turn, is related to the number of defects expected
- More information:
  - http://www.mccabe.com/iq_research_metrics.htm
  - "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", Arthur H. Watson, Thomas J. McCabe, NIST Special Publication 500-235

Example: Independent path coverage

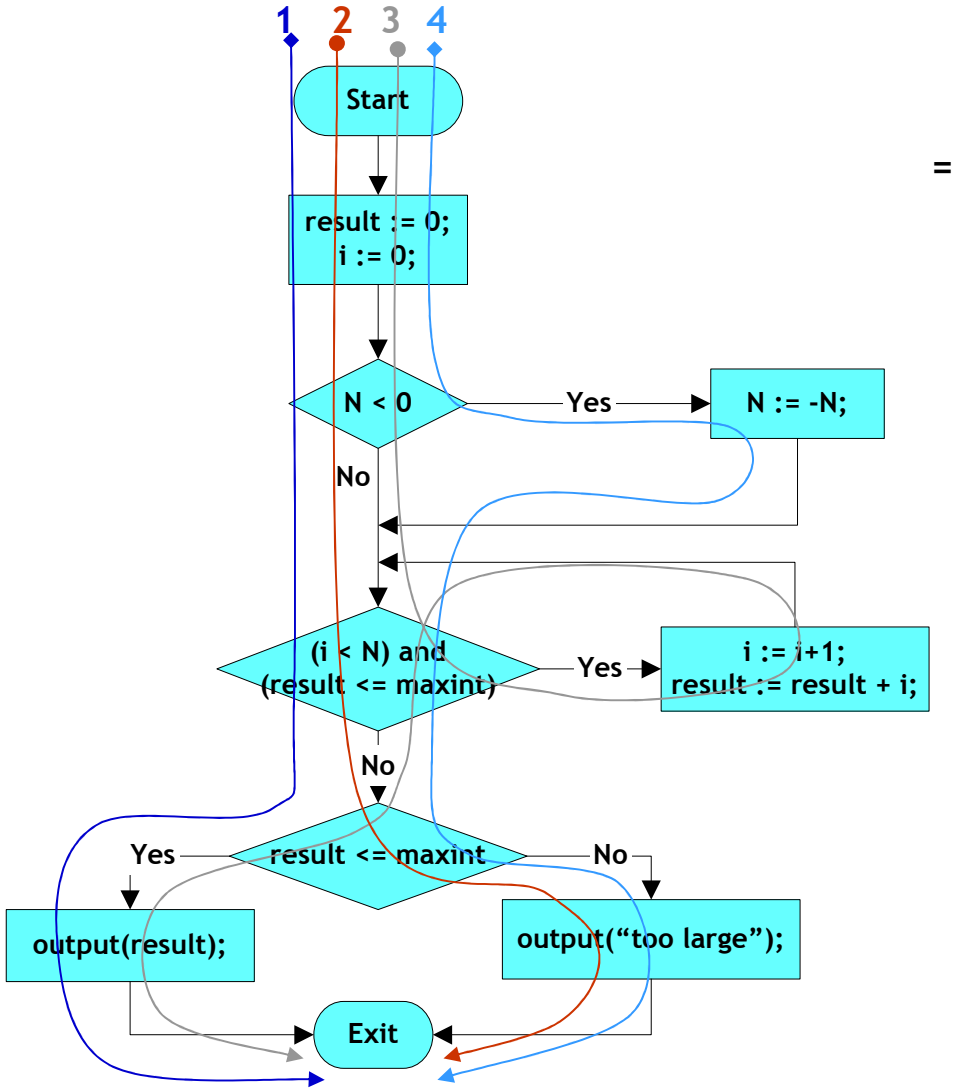


	1	2	3	4	5	6	7	8	9	10
abefa	1	0	0	1	0	0	0	1	0	1
beb	0	0	0	1	1	0	0	0	0	0
abea	1	0	0	1	0	0	0	0	1	0
acfa	0	1	0	0	0	1	0	0	0	1
adcfa	0	0	1	0	0	1	1	0	0	1

Theorem: In a strongly connected graph, G, the cyclomatic number is equal to the maximum number of linear independent circuits.

- 5 independent circuits
  - (abefa),(beb),(abea),(acfa),(adcfa)
- 5 independent paths
  - (abef), (abebef),(abeabef),(acf),(adcf)
  - This set of independent paths forms the basis for the set of all circuits in the graph.
- For instance
  - The path (abea(be)^{3}f) = 2(abebef)-(abef)

Example: Independent path coverage

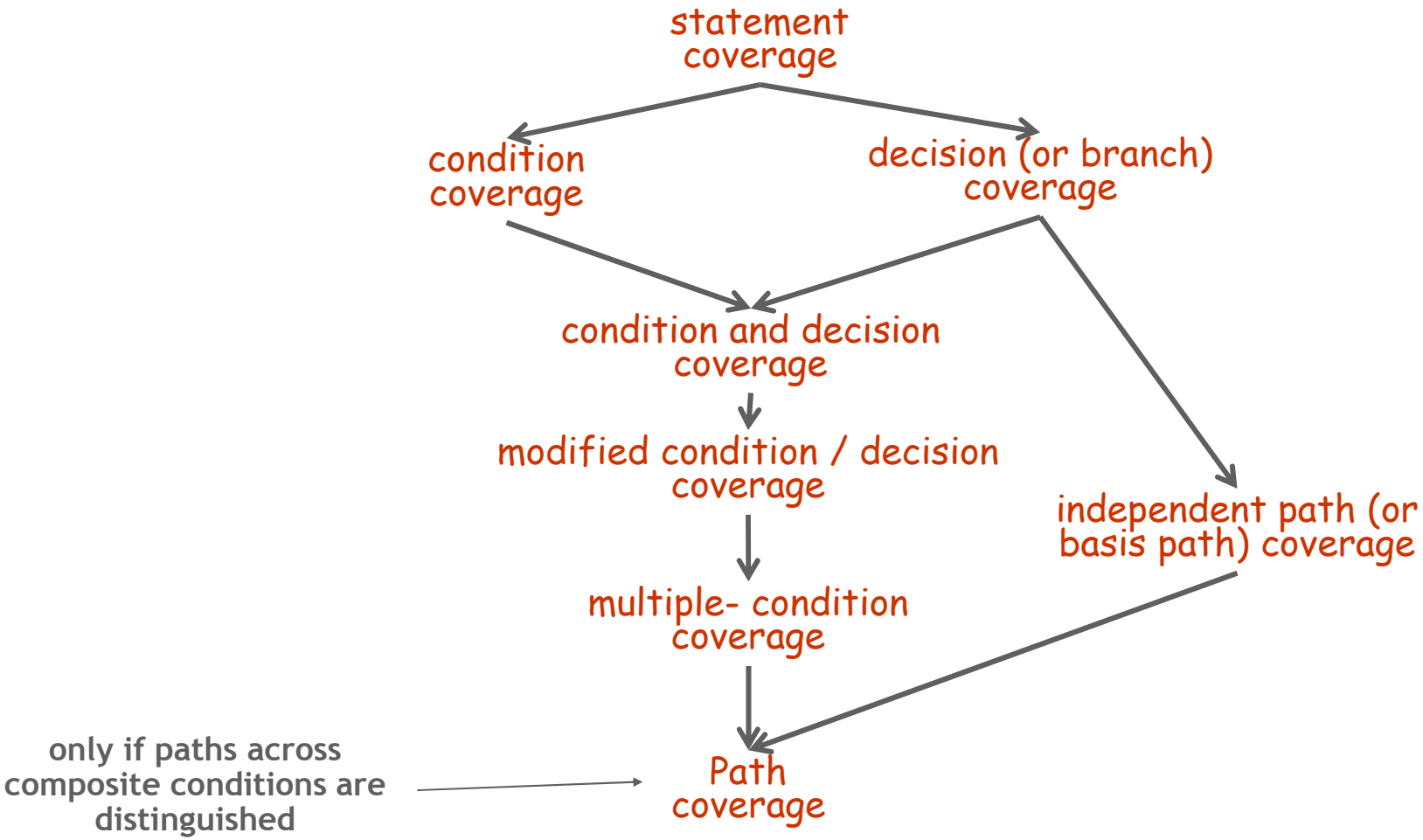


number of independent paths
≡ cyclomatic complexity
= number of edges - number of nodes + 2
= 12 - 10 + 2
= 4

Test cases

Path	inputs		outputs
	maxint	N	result
1	1	0	0
2	-1	0	too large
3	-1	-1	too large
4	10	1	1

White-Box Testing : Overview



Data flow testing

- We say a variable is **defined** in a statement when its value is assigned or changed

```
Y = 26 * X
```

  - This is indicated as a *def* for the variable Y

- We say a variable is **used** in a statement when its value is utilized in a statement. The value of the variable is not changed. Data flow roles:
  - *c-use* for variable X:

```
Y = 26 * X
```

  - *p-use* for variable X:

```
if (X  > 98)
    Y  = max
```

  - others: *undefined* or *dead*

Data flow testing

- Coverage criteria
  - All def
  - All p-uses
  - All c-uses/some p-uses
  - All p-uses/some c-uses
  - All uses
  - All def-use paths
  - … and several variants of this technique …

- The strongest of these criteria is *all def-use paths*. This includes all *p-* and *c-uses*.

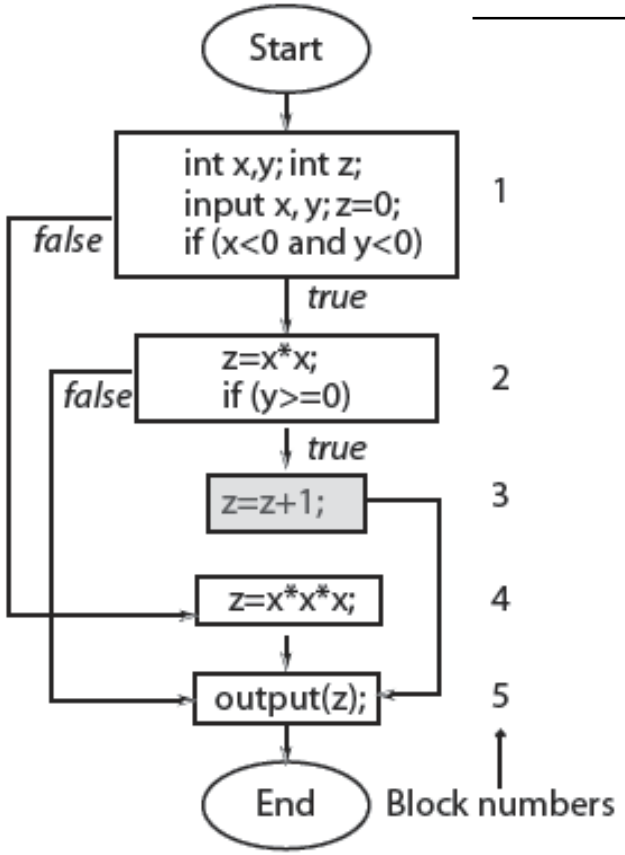
Data flow graph

- A data-flow graph of a program, also known as def-use graph, captures the flow of definitions (also known as defs) across basic blocks in a program.
- It is similar to a control flow graph of a program in that the nodes, edges, and all paths thorough the control flow graph are preserved in the data flow graph. An example follows.

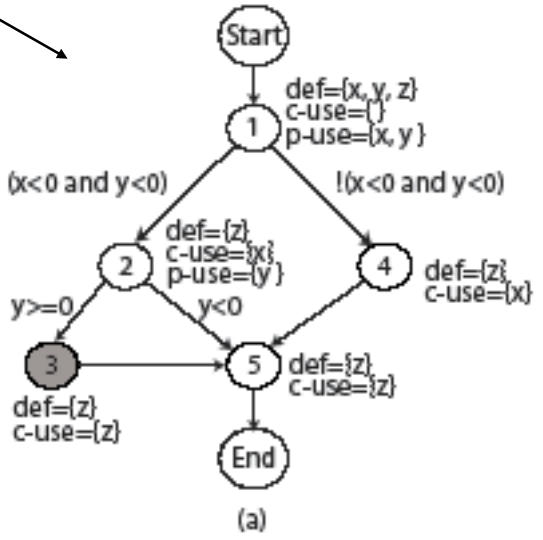
Data flow graph: Example

- Given a program, find its basic blocks, compute defs, c-uses and p-uses in each block. Each block becomes a node in the def-use graph (this is similar to the control flow graph).
- Attach defs, c-use and p-use to each node in the graph. Label each edge with the condition which when true causes the edge to be taken.
- We use $d_i(x)$ to refer to the definition of variable x at node i. Similarly, $u_i(x)$ refers to the use of variable x at node i.

Data flow graph: Example (contd.)



Node (or Block)	def	c-use	p-use
1	{x, y, z}	{ }	{x, y}
2	{z}	{x}	{y}
3	{z}	{z}	{ }
4	{z}	{x}	{ }
5	{ }	{z}	{ }



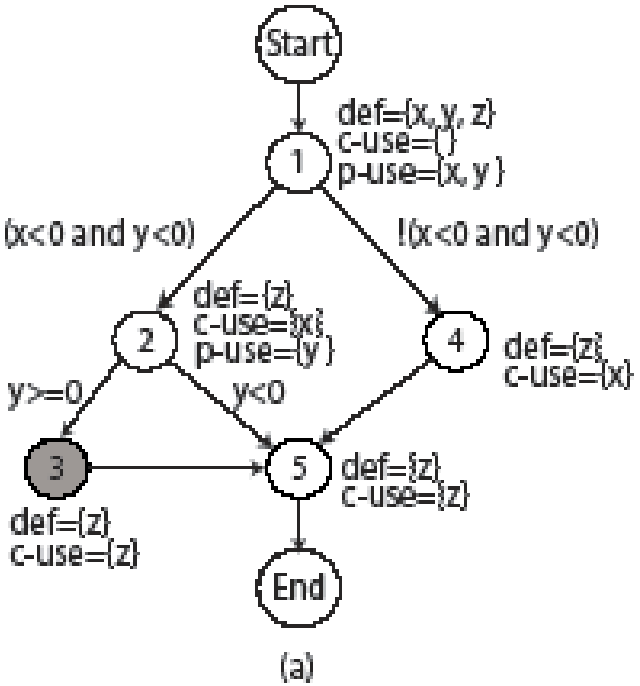
Unreachable node

Def-clear path

Any path starting from a node at which variable x is defined and ending at a node at which x is used, without redefining x anywhere else along the path, is a *def-clear* path for x.

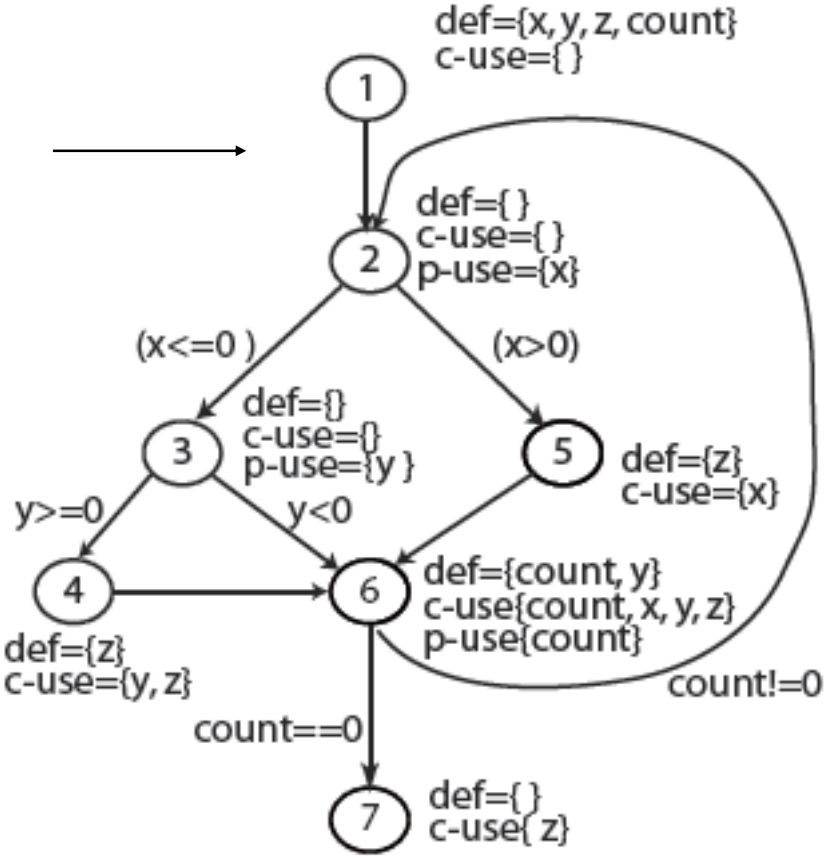
Path 2-5 is def-clear for variable z defined at node 2 and used at node 5. Path 1-2-5 is NOT def-clear for variable z defined at node 1 and used at node 5.

Thus definition of z at node 2 is live at node 5 while that at node 1 is not live at node 5.



Def-clear path (another example)

```
1    begin
2      float x, y, z=0.0;
3      int count;
4      input (x, y, count);
5      do {
6        if (x≤0) {
7          if (y≥0) {
8            z=y*z+1;
9          }
10       }
11       else{
12         z=1/x;
13       }
14       y=x*y+z
15       count=count-1
16     while (count>0)
17     output (z);
18   end
```



Node	Lines
1	1, 2, 3, 4
2	5, 6
3	7
4	8, 9, 10
5	11, 12, 13
6	14, 15, 16
7	17, 18

Find def-clear paths for defs and uses of x and z. Which definitions are live at node 4?

Mutation testing (fault injection)

- Starts with a code component and its associated test cases (in a state such that the code passes all test cases)
- The original code component is modified in a simple way (replace operators, constants, etc.) to provide a set of similar components that are called mutants, based on typical errors
- The original test cases are run with each mutant
- Live mutants cannot be distinguished from the original program (parent)
- Distinguishing a mutant from its parent is referred to as killing such mutant
- If a mutant remains live (passes all the test cases), then either the mutant is equivalent to the parent (and is ignored), or it is not equivalent, in which case **additional test cases** should be developed in order to kill such mutant
- The rate of mutants "killed" (after removing mutants that are equivalent to the original code) gives an indication of the rate of undetected defects that may exist in the original code

Class testing / coverage

- In object oriented systems, the units to be tested are typically classes, but can also be methods or clusters of related classes
- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - *Exercising the object in all possible states*
- Each test case typically exercises a possible object lifecycle, with alternating operation calls to change and query the object's state
- Encapsulation, inheritance and polymorphism complicate the design of test cases
  - Encapsulation – how to check the object's state?
  - Inheritance – how to (unit) test abstract classes, abstract methods and interfaces?
  - Polymorphism – how to test methods with *callbacks* (from super-class to sub-class)? (same problem with event handlers)

White-Box testing : How to Apply ?

- Don’t start with designing white-box test cases!
- Start with black-box test cases
  (equivalence partitioning, boundary value analysis,
  cause effect graphing, derivation with formal methods,…)
- Check white-box coverage
  (statement, branch, condition,… , coverage)
- Use a testing coverage tool
- Design additional white-box test cases for not covered code

Exercise 1

If the pseudocode below were a programming language ,how many tests are required to achieve 100% statement coverage?

```
If x=3 then
        Display_messageX;
        If y=2 then
                Display_messageY;
        Else
                Display_messageZ;
Else
        Display_messageZ;
```

- 1. Choose the correct answer

  a) 1; b) 2; c) 3; d) 4

Using the same code example as question 17,how many tests are required to achieve 100% branch/decision coverage?

- 2. Choose the correct answer

  a) 1; b) 2; c) 3; d) 4

Exercise 2

- **Given the following code, which is true about the minimum number of test cases required for full statement and branch coverage:**

```
Read P
Read Q
IF P+Q > 100 THEN
   Print “Large”
ENDIF
If P > 50 THEN
   Print “P Large”
ENDIF
```

a) 1 test for statement coverage, 3 for branch coverage

b) 1 test for statement coverage, 2 for branch coverage

c) 1 test for statement coverage, 1 for branch coverage

d) 2 tests for statement coverage, 3 for branch coverage

e) 2 tests for statement coverage, 2 for branch coverage

Exercise 3 (1)

- A Program is written to meet the following requirements:
  - R1: Given coordinate positions x, y and z, and a direction valued d, the program must invoke one of the three functions fire-1, fire-2, fire-3 as per conditions bellow:
    - R1.1: Invoke fire-1 when (x<y) AND (z * z > y) AND (prev="East").
    - R1.2: Invoke fire-2 when (x<y) AND (z * z ≤ y) OR (current="South").
    - R1.3: Invoke fire-3 when none of the two conditions above is true.
  - R2: The invocation described above must continue until an input Boolean variable becomes true.

Exercise 3 (2)

```
1  begin
2    float x, y, z;
3    direction d;
4    string prev, current;
5    bool done;

6    input(done);
7    current="North";
8    while (~done)  {                                          ← condition C1
8              input (d);
10             prev=current; current=f(d);
11             input(x,y,z);
12             if ((x<y) and (z*z > y) and (prev=="East"))     ← Condition C2
13                             fire-1(x,y);
14             else if ((x<y) and (z*z<=y) or (current=="South"))   ← Condition C3
15                             fire-2(x,y);
16             else
17                             fire-3(x,y);
17             input(done);
18   }
19   output("Firing completed.");
20 end
```

Exercise 3 (3)

- Verify that the following set T1 of four tests, executed in the given order, is adequate with respect to statement, block, and decision coverage criteria but not with respect to the condition coverage criterion.

Test set T1						
Test	Requirement	done	d	x	y	z
t1	R1.2	False	East	10	15	3
t2	R1.1	False	South	10	15	4
t3	R1.3	False	North	10	15	5
t4	R2	True	-	-	-	-

Exercise 3 (4)

Test set T2						
Test	Requirement	done	d	x	y	z
t1	R1.2	False	East	10	15	3
t2	R1.1	False	South	10	15	4
t3	R1.3	False	North	10	15	5
t5	R1.1 and R1.2	False	South	10	5	5
t4	R2	True	-	-	-	-

Test set T2 is adequate according to MC/DC?

Exercise 3 (5)

- Verify that the following set T3, obtained by adding t6, t7, t8, and t9 to T2 is adequate with respect to MC/DC coverage criterion. Note again that sequencing of tests is important in this case (especially for t1 and t7)!

Test set T3						
Test	Requirement	done	d	x	y	z
t1	R1.2	False	East	10	15	3
t6	R1	False	East	10	5	2
t7	R1	False	East	10	15	3
t2	R1.1	False	South	10	15	4
t3	R1.3	False	North	10	15	5
t5	R1.1 and R1.2	False	South	10	5	5
t8	R1	False	South	10	5	2
t9	R1	False	North	10	5	2
t4	R2	True	-	-	-	-

Exercise 4

- Suppose that condition C=C1 AND C2 AND C3 has been coded as C'=C1 AND C3. Four tests that form an MC/DC adequate set for C' are in the following table. Verify that the following set of four tests is MC/DC adequate but does not reveal the error.

	Test	C	C'	Error detected?
	C1, C2, C3	C1 and C2 and C3	C1 and C3	
t1 t2	true, true, true false, false, false	true false	true false	No
t3 t4	true, true, false false, false, true	false false	false false	No