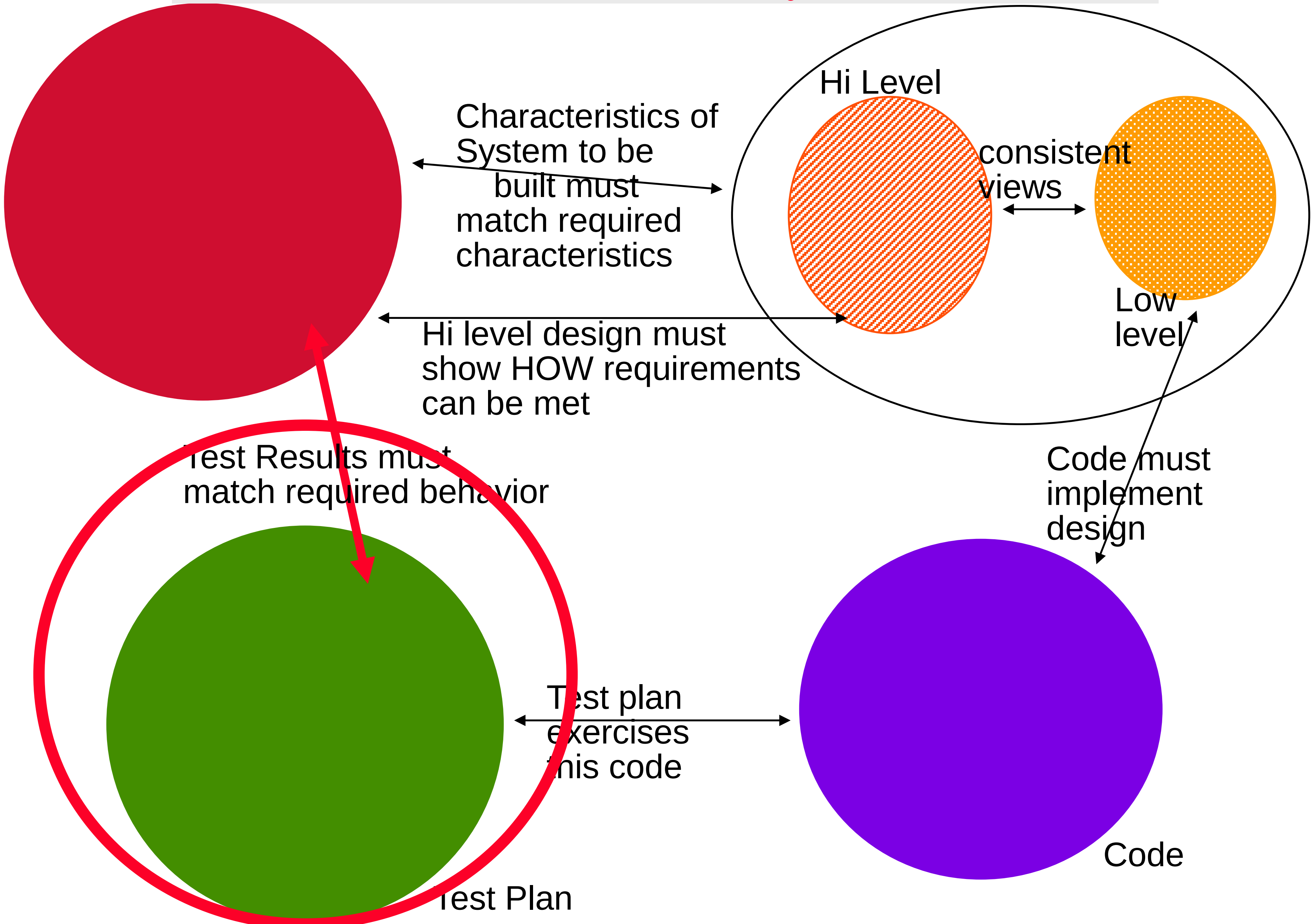
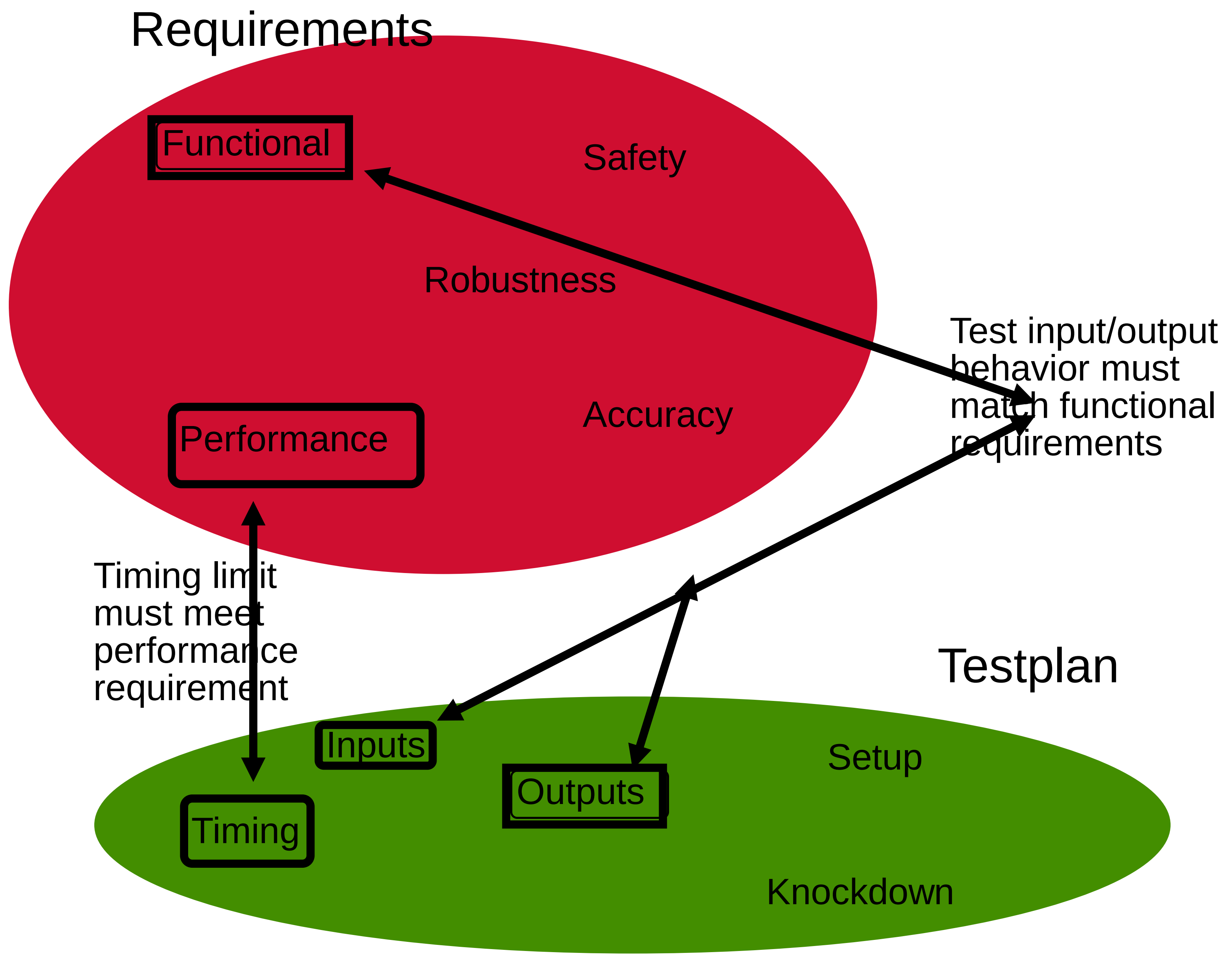
Testing

Reda Bendraou
reda.bendraou{{@}}Lip6.fr
http://pagesperso-systeme.lip6.fr/Reda.Bendraou/

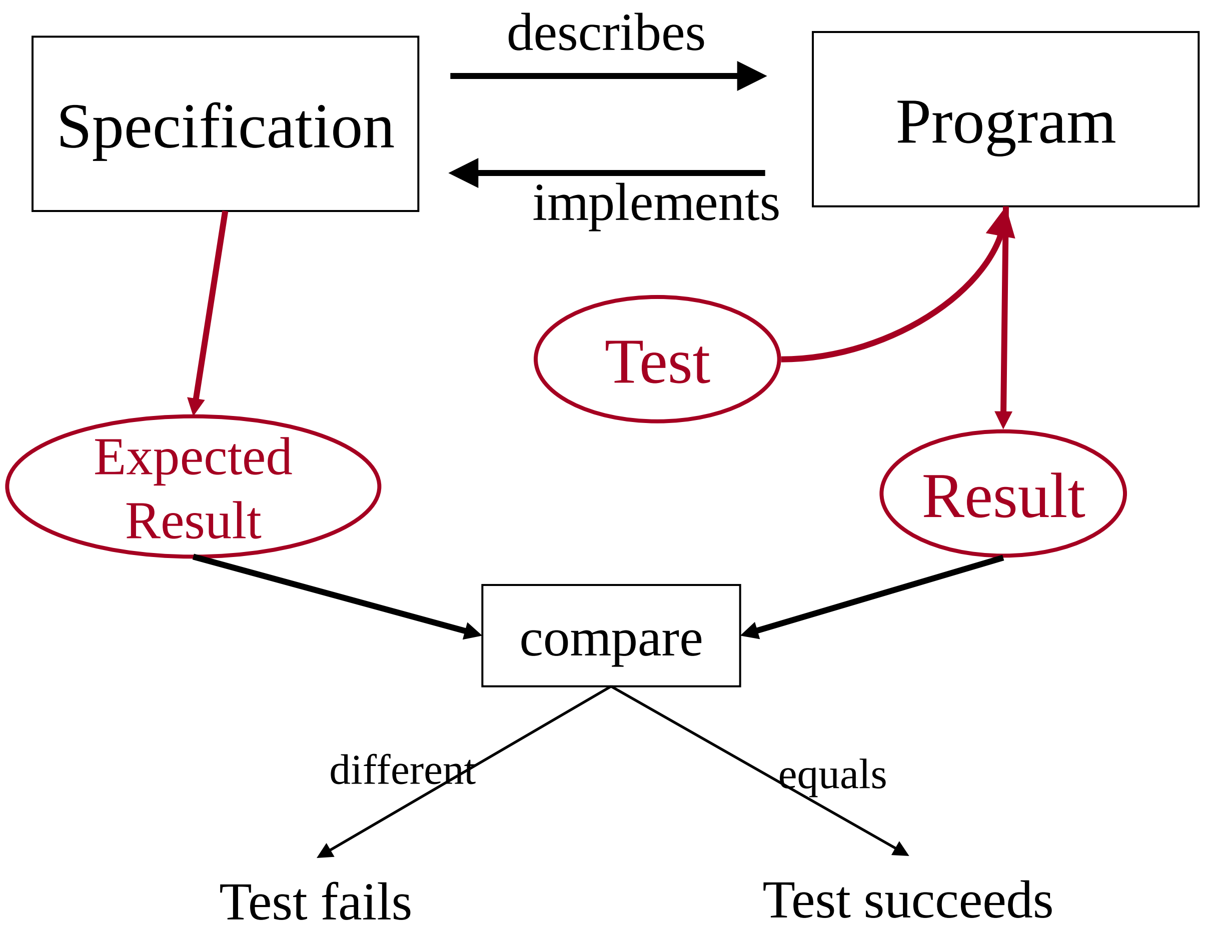
Some slides were adapted from L. Osterweil, B. Meyer, and P. Müller material

Focus on "How do you know"


Requirem
Characteristics of System to be built must match required characteristics
Hi Level
consistent views
Low level
Hi level design must show HOW requirements can be met
Test Results must match required behavior
Code must implement design
Test plan exercises this code
Test Plan
Code


Requirements
Functional
Safety
Robustness
Performance
Accuracy
Test input/output behavior must match functional requirements
Timing limit must meet performance requirement
Testplan
Inputs
Outputs
Setup
Timing
Knockdown

Testing


Specification
describes
Program
implements
Test
Expected Result
Result
compare
different
equals
Test fails
Test succeeds

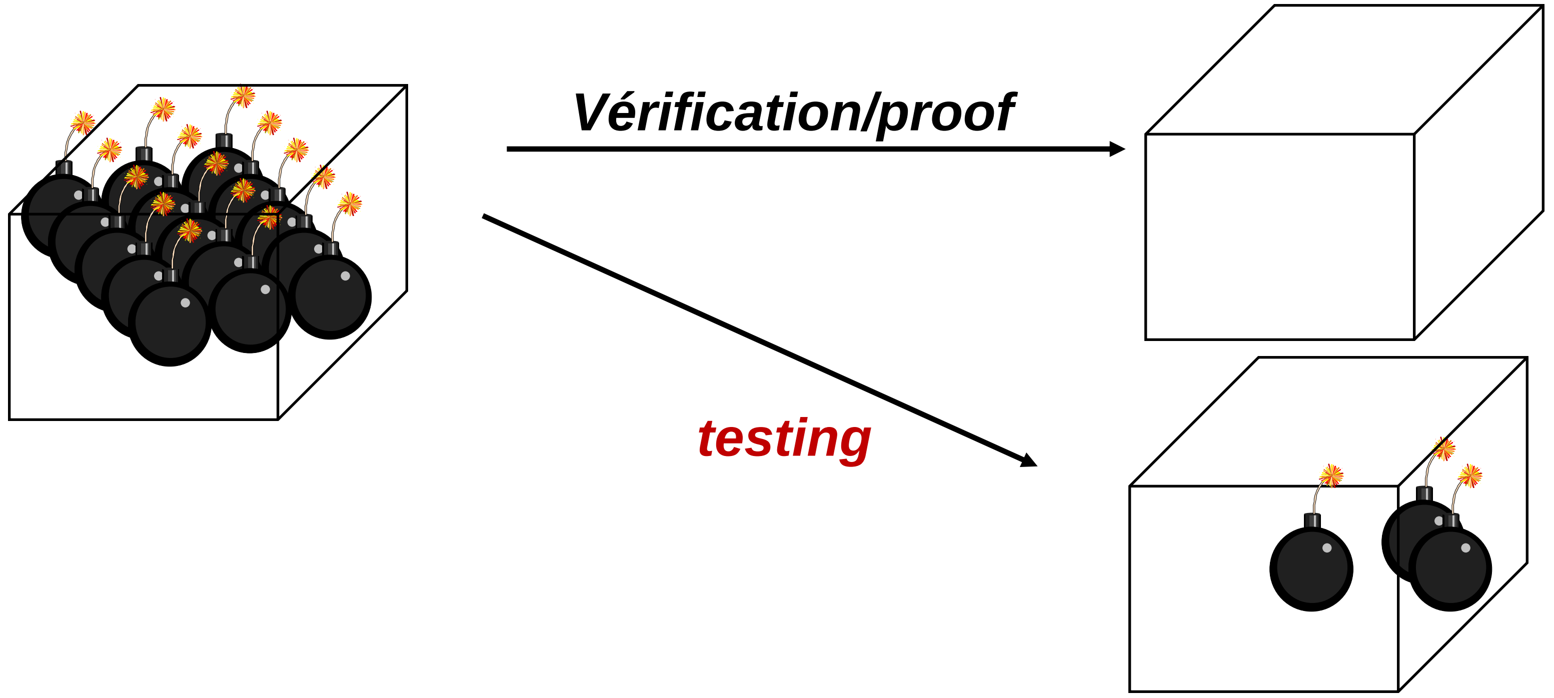
Testing issues

Testing : Validation

“Testing can only show the presence of errors never their absence!”

E.W. Dijkstra, 1970, Structured Programming and a few other places

validation $\longrightarrow$ ***does the product meet the spec?***

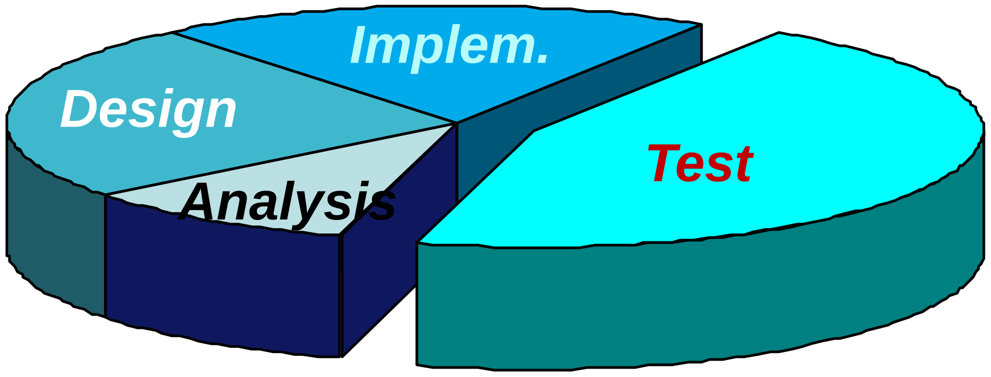


Testing : problem

- We can’t afford to test everything and at anytime
  - We have to make choices about what to test and with which input values
- Proving that a program is bug free is an undecidable (indeterminate) problem
  - Need to define some realistic heuristics
- To test a software is to try to make it fail
  - Software fails => Test successful!

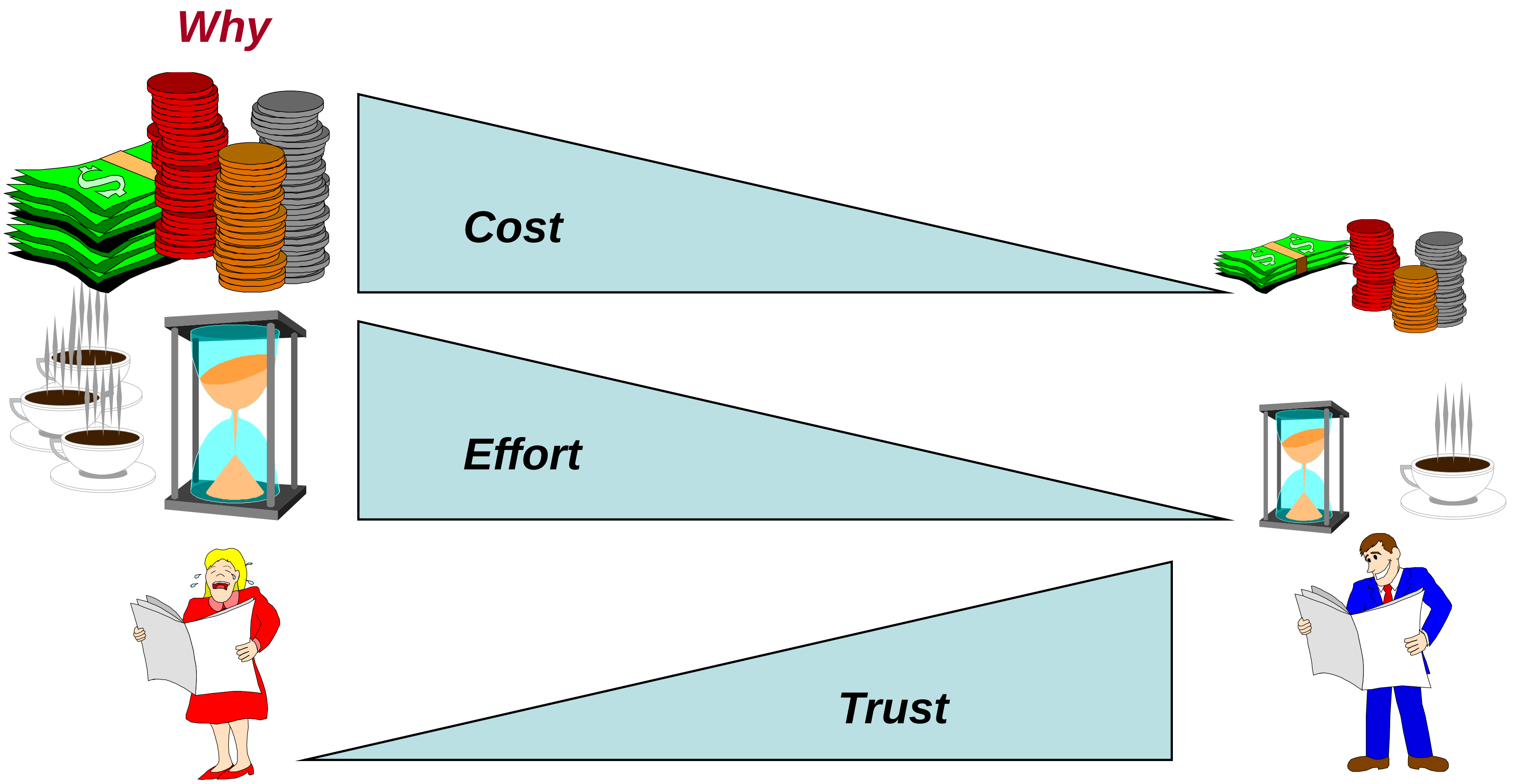
Testing : costs

Testing cost in development processes



+ maintenance = 80 % of global development cost!!!

Testing : costs



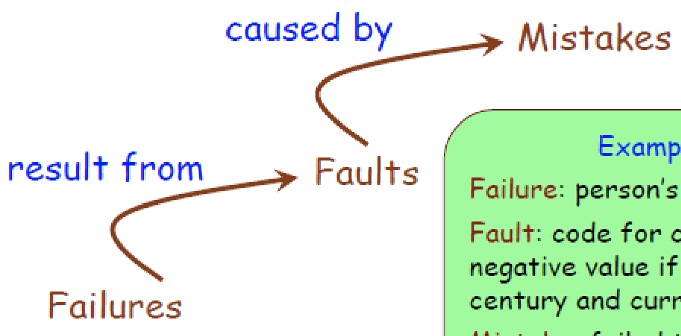
Testing

- 30% students first job/experience is about testing
- 50% of start-ups collapse because they deliver software with a lot of bugs
  - Bad test campaign
  - No “no regression” testing
  - Bad maintenance

Failure?

- A failure=>any event of system's execution that violates a stated quality objective
- Quality is the absence of "deficiencies " (bugs)

More precise terminology (IEEE):



Also: Error
In the case of a failure, extent of deviation from expected result

Example: A Y2K issue

Failure: person's age appears as negative!

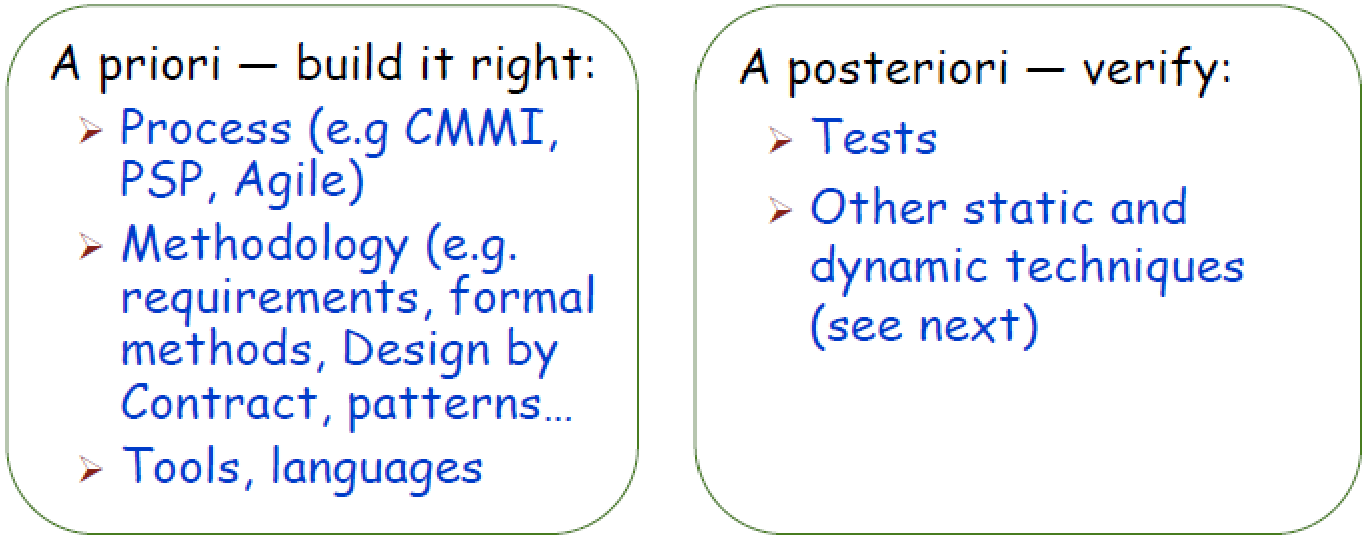
Fault: code for computing age yields negative value if birthdate is in 20th century and current date in 21st

Mistake: failed to account for dates beyond 20th century

From P. Müller

When and how Quality can be achieved

At any time



A priori — build it right:

- Process (e.g CMMI, PSP, Agile)
- Methodology (e.g. requirements, formal methods, Design by Contract, patterns…
- Tools, languages

A posteriori — verify:

- Tests
- Other static and dynamic techniques (see next)

When and how Quality can be achieved

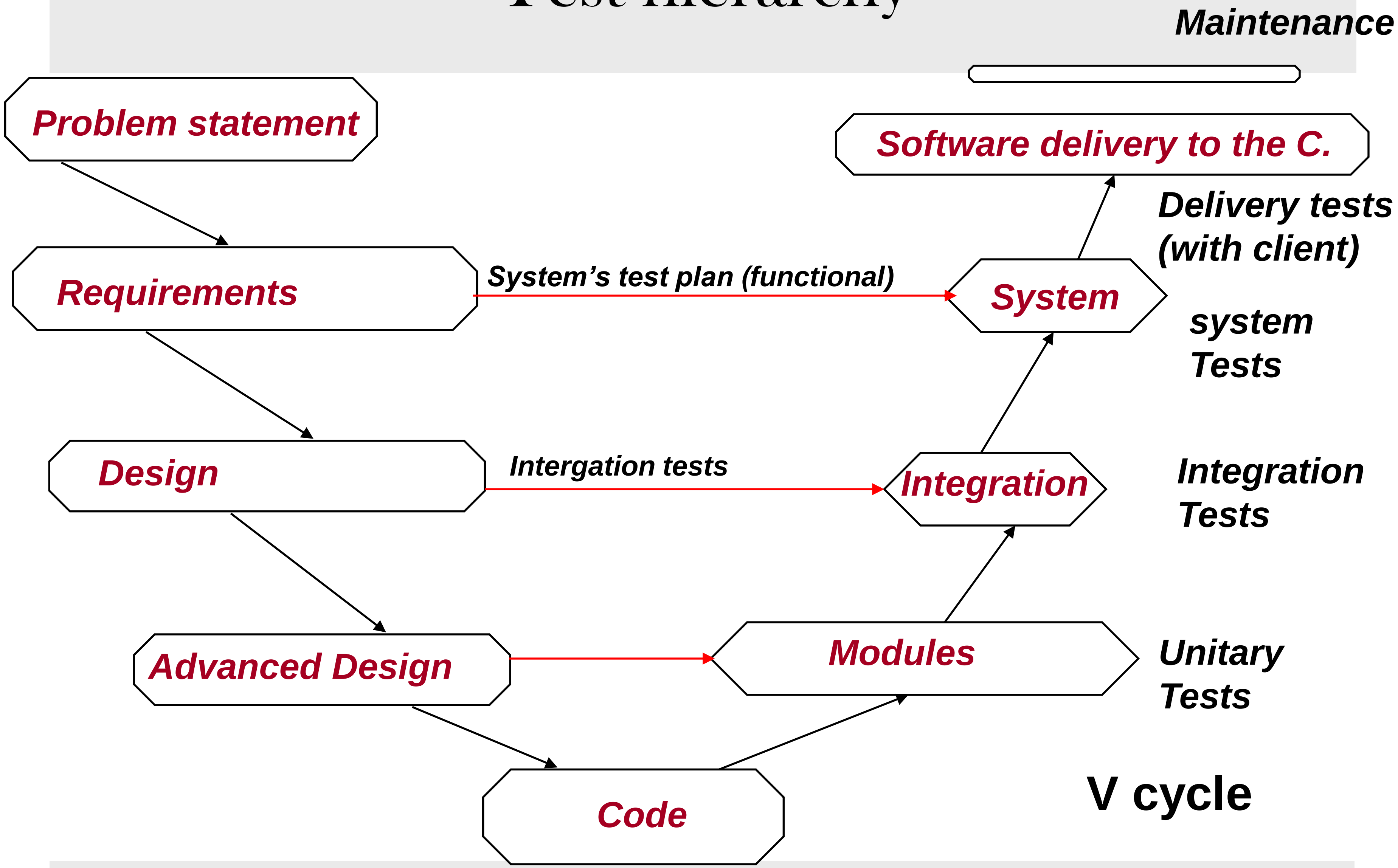
- How (for a posteriori)

Static (no execution):
- Reviews (human)
- Type checking & enforcement of other reliability-friendly programming language traits
- Static analysis
- Proofs

In-between but mostly static:
- Model checking
- Abstract interpretation
- Symbolic execution

Dynamic (must execute):
- Tests

Test hierarchy



Test hierarchy

Unit/Module
- Comparing a code unit or module with design specifications.
- planned during coding: done after coding

Integration
- Systematic combination of software components and modules
- planned during design: done after unit/module V&V

Software System
- Comparing entire software system with requirements
- planned during requirements: done after integration

System
- Comparing integrated hardware/software system to requirements
- planned during informal requirements: after SW System

Testing: basic concepts

Implementation Under Test (IUT)

The software (& possibly hardware) elements to be tested

Test case

Precise specification of one execution intended to uncover a possible fault:

- Required state & environment of IUT before execution
- Inputs

Test run

One execution of a test case

Test suite

A collection of test cases

Testing: basic concepts

Expected results (for a test case)
Precise specification of what the test is expected to yield in the absence of a fault

fault:

- Returned values
- Exceptions
- Resulting state of program & environment
- Non-functional characteristics (time, memory…)
- Messages

Test oracle
A mechanism to determine whether a test run satisfies the expected results
□ Output is generally just “pass” or “fail”.

Testing: basic concepts

Test driver

A program, or program element (e.g. class), used to apply test cases to an IUT

Stub

A temporary implementation of a software element, replacing its actual implementation during testing of other elements relying on it.
Generally doesn't satisfy the element's full specification.
May serve as placeholder for:

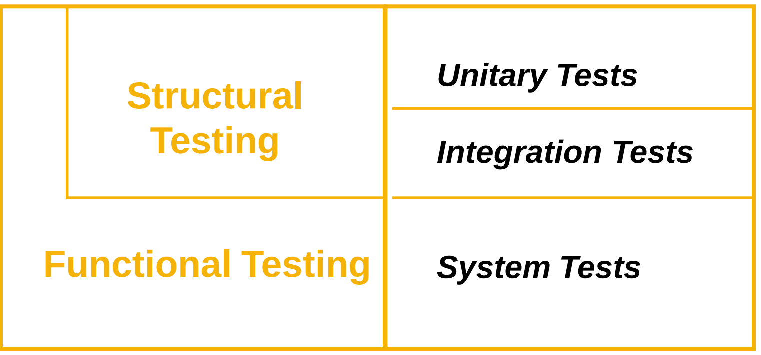
- A software element that has not yet been written
- External software that cannot be run for the test (e.g.
- because it requires access to hardware or a live database)
- A software element that takes too much time or memory to
- run, and whose results can be simulated for testing purposes

Test Classification

By Goal

- Functional test
- Performance test
- Stress (or “load”) test

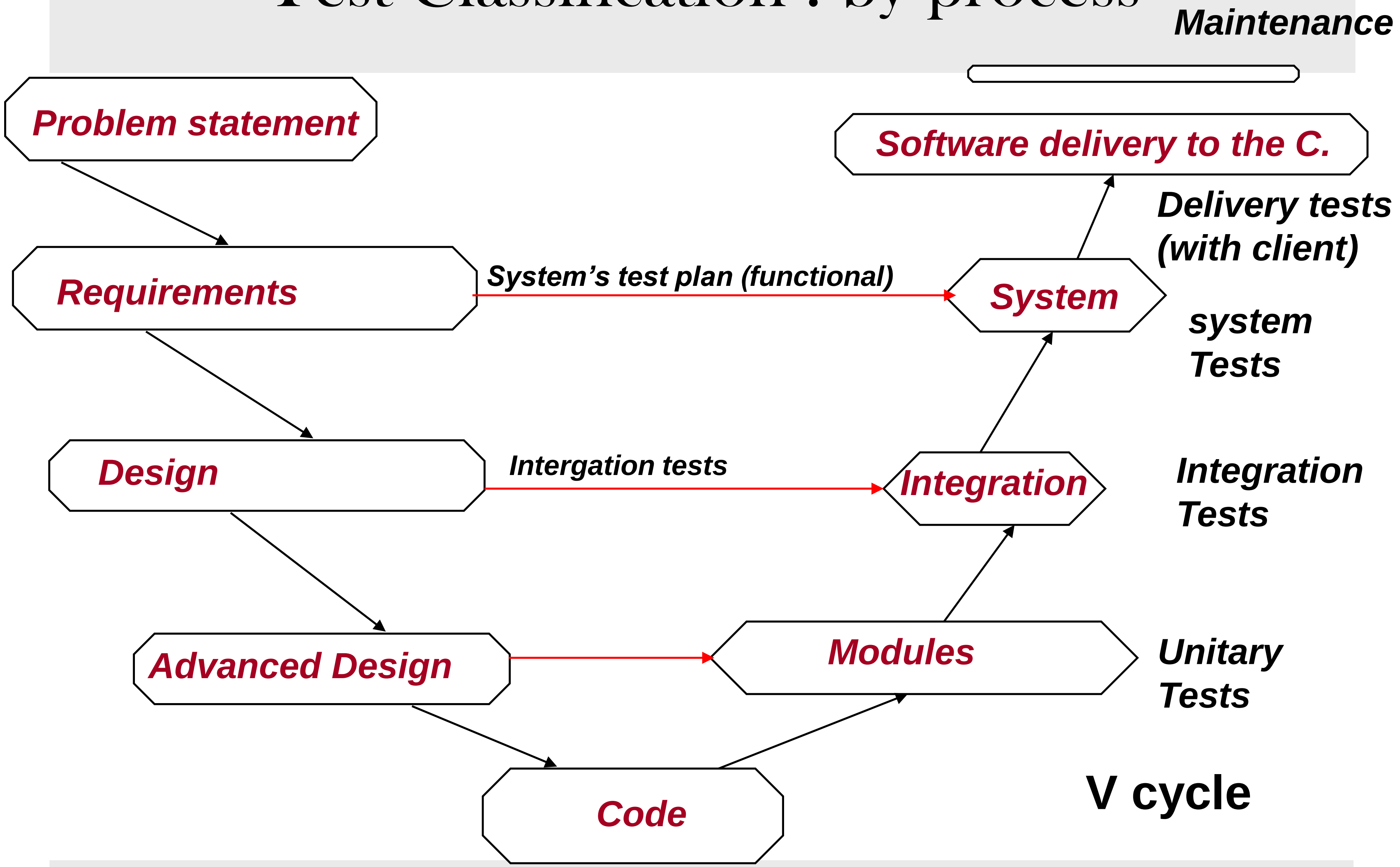
By scope



By intent

Fault-directed testing
Goal: reveal faults through failures
□ Unit and integration testing
Conformance-directed testing
Goal: assess conformance to required capabilities
□ System testing
Acceptance testing
Goal: enable customer to decide whether to accept a product
Regression testing
Goal: Retest previously tested element after changes, to assess whether they have re-introduced faults or uncovered new ones.
Mutation testing
Goal: Introduce faults to assess test case quality

Test Classification : by process



Test Classification : Black Vs white box

White-Box
- Source code (IUT) is used to define test cases

Black-box
- Properties of IUT available only through specification , interfaces, etc.
- Typical use : integration

Testing: Process

- Identify parts of the software to be tested
- Identify interesting input values (next slide)
- Identify expected results (functional) and execution characteristics (non-functional)
- Run the software on the input values
- Compare results & execution characteristics to expectations

How to identify interesting inputs?

- Need for realistic inputs, would not be feasible to test all values !!

- **Partition testing**: select elements from a partition of the input set, i.e. a set of subsets that is
  - Complete: union of subsets covers entire domain
  - Pairwise disjoint: no two subsets intersect

- Purpose (or hope!):
  - For any input value that produces a failure, some other in the same subset produces a similar failure

- Common abuse of language: "a partition" for "one of the subsets in the partition"
  - Better called "**equivalence class (ec)**"

- Each ec should at least be used by one test case

How to identify interesting inputs?

Ideas for equivalence classes:

- Set of values so that if any is processed correctly (incorrectly) then any other will be processed correctly (incorrectly)
- Values at the center of a range, e.g. 0, 1, -1 for integers
- Boundary values, e.g. MAXINT
- Values known to be particularly relevant
- Values that must trigger an error message (“invalid”)
- Intervals dividing up range, e.g. for integers

Test Quality: notion of Coverage?

Goal: to assess the effectiveness of a test suite,
=>Measure how many instructions in your program are reached (executed) by your test.

How it works:

- Choose a kind of program element, e.g. instructions (instruction coverage) or paths (path coverage)
- Count how many are executed at least once
- Report as percentage

100% coverage achieved by a test suite => every instruction in the element tested has been executed at least by one test

Known tools: Emma, Jcoverage, Ncover..etc

Coverage criteria

Instruction (or: statement) coverage:

Measure instructions executed

Disadvantage: insensitive to some control structures

Branch coverage:

Measure conditionals whose paths are both executed

Condition coverage:

Count how many atomic boolean expressions evaluates to both true and false

Path coverage:

Count how many of the possible paths are taken

(Path: sequence of branches from routine entry to exit)

Test Quality: Mutation testing

Creation of tests still poses the question whether the tests are correct and sufficiently cover the requirements that have originated the implementation.

- "Quis custodiet ipsos custodes?" ["Who will guard the guards?"].)
- In our case, who tests the tester?

Mutation testing !

Idea: make small changes to the program source code (so that the modified versions still compile) and see if your test cases fail for the modified versions

Purpose: estimate the quality of your test suite

What it is Mutation testing?

if a mutant is introduced without the behavior (generally output) of the program being affected this indicates:

- The code that had been mutated was never executed (dead code)
- The test suite was unable to locate the faults represented by the mutant.

Mutants are based on well-defined mutation operators

- mimic typical programming errors (ex. using the wrong operator or variable name)
- force the creation of valuable tests (ex. dividing each expression by zero).

Disadvantage

- A large number of mutants usually are introduced into a large program,
- Requires the compilation and execution of an extremely large number of copies of the program.

Mutants: example

Original program:

```
if (a < b)
        b := b - a;
else
        b := 0;
```

Mutants:

```
if (a < b)
if (a <= b)
if (a > b)
if (c < b)
        b := b - a;
        b := b + a;
        b := x - a;
else
        b := 0;
        b := 1;
        a := 0;
```

Mutation: some concepts

A mutant is the result of an application of a mutation operator
Faulty versions of the program = mutants

A mutant is Killed if at least one test case detects the fault injected into the mutant, Alive otherwise

Mutation operator: a rule that specifies a syntactic variation of the program text so that the modified program still compiles

The quality of the mutation operators determines the quality of the mutation testing process.

Mutation operator coverage (MOC): For each mutation operator, create a mutant using that mutation operator.

Mutation Operators: Examples

- Replace arithmetic operator by another
- Replace relational operator by another
- Replace logical operator by another
- Replace a variable by another
- Replace a variable (in use position) by a constant
- Replace a constant by another
- Replace “while… do…” by “repeat… until…”
- Replace condition of test by negation
- Replace call to a routine by call to another

Testing: How to proceed?

- ❑ Defining the process
    - ❑ **Test plan (next slides)**
    - ❑ Input and output documents
- ❑ Who is testing?
    - ❑ Developers / special testing teams / customer
- ❑ What test levels do we need?
    - ❑ Unit, integration, system, acceptance, regression
- ❑ Order of tests
    - ❑ Top-down, bottom-up, combination
- ❑ Running the tests
    - ❑ Manually
    - ❑ Use of tools
    - ❑ Automatically

Test Plans and Test Planning

- Goal: Determine if the product satisfies the requirements that spawned its development
- *Testing* is done after the product is built
- *Test planning* commences during requirements
  - Testing can be an elaborate process
    - Best to plan it out
  - Testing can require elaborate harnesses
  - Testing difficulties can shape product requirements
  - Testing effort can be saved by static analysis
  - Make sure your requirements are Testable !!

Design of a Test Plan

- Structure
  - Possibly a tree-like hierarchy of Test plan Element Specifications (TES)
  - One TES for each function, or functional aspect to be tested
    - Function identification
    - Testing goals
- Summary of Resources required
  - Time
  - People
  - Equipment
  - Other software systems
    - Available or to-be-built
- Evaluation approach: Is this worth the cost?

Example Structure of a Testplan Element Specification

- Goals/Requirements for this test case
- Requirements element (e.g. function) or aspect (e.g. security) being tested
- Needed resources (e.g. databases, users, computers, software)
- Setup procedure
- Input data, which may be
  - fixed, randomly selected, selected from a list
- Output results required
  - Speed required
  - Definition of what is "correct" output
    - fixed number, range, formula
- Response to failure(s)
- Cleanup/knockdown
- Evaluation: turning this data into information, maybe into knowledge.

Some Kinds of Test Element Goals

- Is the functionality correct?
- Does the software execute fast enough?
- Is the software easy enough to use to satisfy a particular class of stakeholders?
- Does the software “fail safe” under certain specific circumstances?

Some test plan elements can potentially test more than one of these

What kind of properties to test?

- Functional
- Security/Safety
- Usability
- Consistency
- Performance
- Robustness

Who Tests?

Any significant project should have a separate Quality Analysis team

Unit tests: the developers

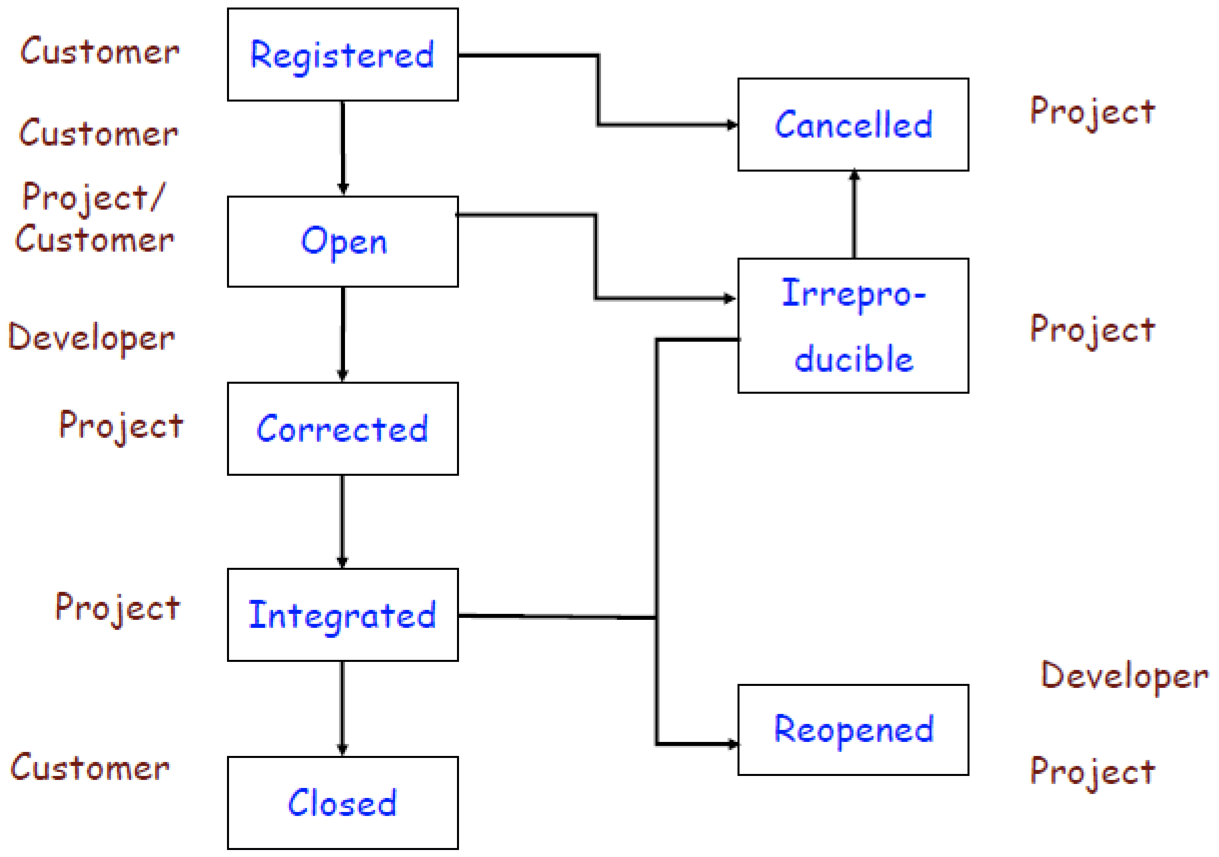
Suggestion: pair each developer with another who serves as "personal tester"

Integration test: developer or QA team

System test: QA team

Acceptance test: customer & QA team

Error life cycle: example



From B. Meyer and P. Muller

What about Standards? IEEE 829

IEEE Standard for Software Test Documentation, 1998

Can be found at: **http://tinyurl.com/35pcp6**
(shortcut for: http://www.ruleworks.co.uk/testguide/documents/
IEEE%20Standard%20for%20Software%20Test%20Documentation..pdf)

Specifies a set of test documents and their form

For an overview, see the Wikipedia entry

IEEE-829: Test elements

Test plan:

“Prescribes scope, approach, resources, & schedule of testing. Identifies items & features to test, tasks to perform, personnel responsible for each task, and risks associated with plan”

Test specification documents:

- Test design specification: identifies features to be covered by tests, constraints on test process
- Test case specification: describes the test suite
- Test procedure specification: defines testing steps

Test reporting documents:

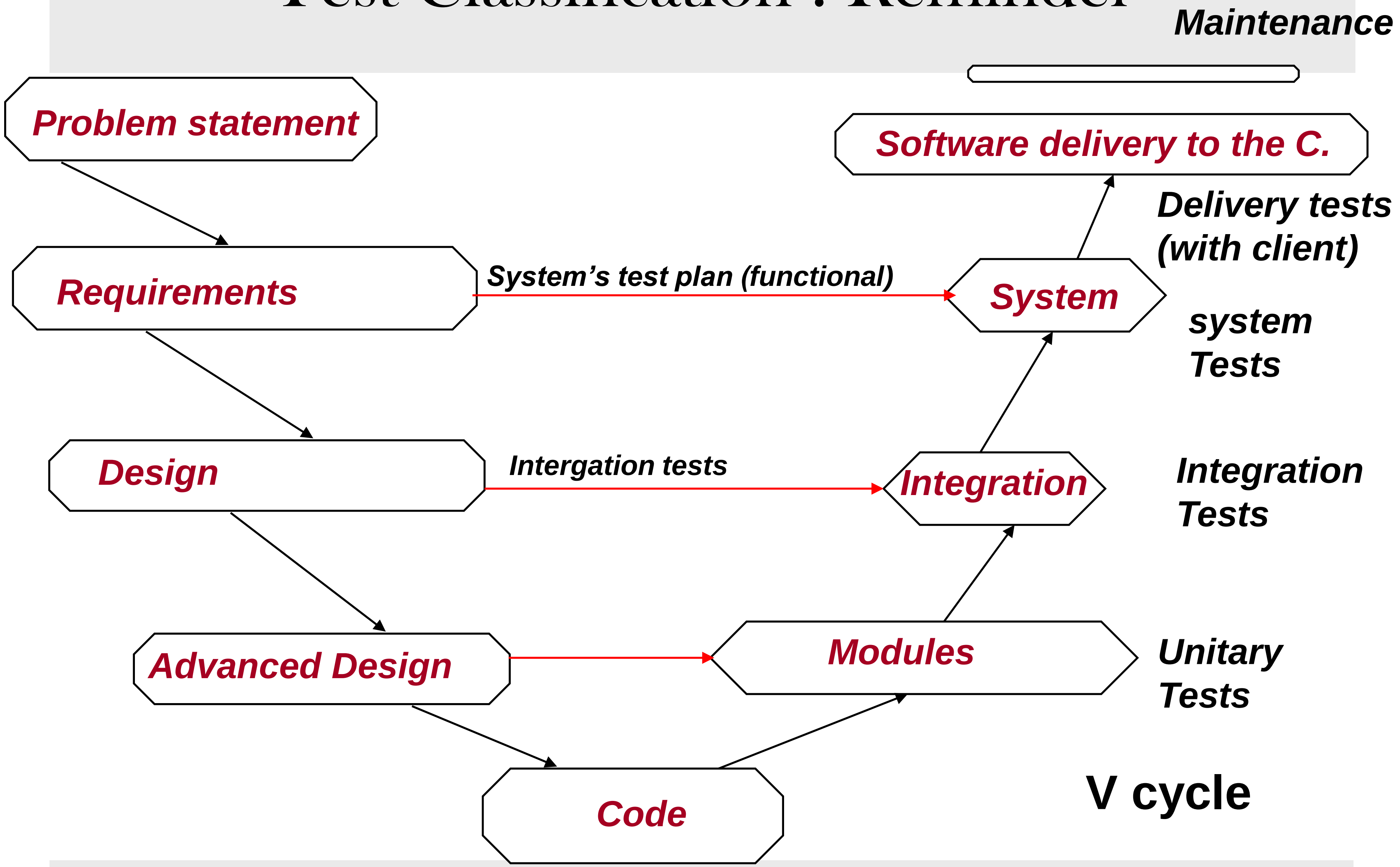
- Test item transmittal report
- Test log
- Test incident report
- Test summary report

IEEE 829: Test plan structure

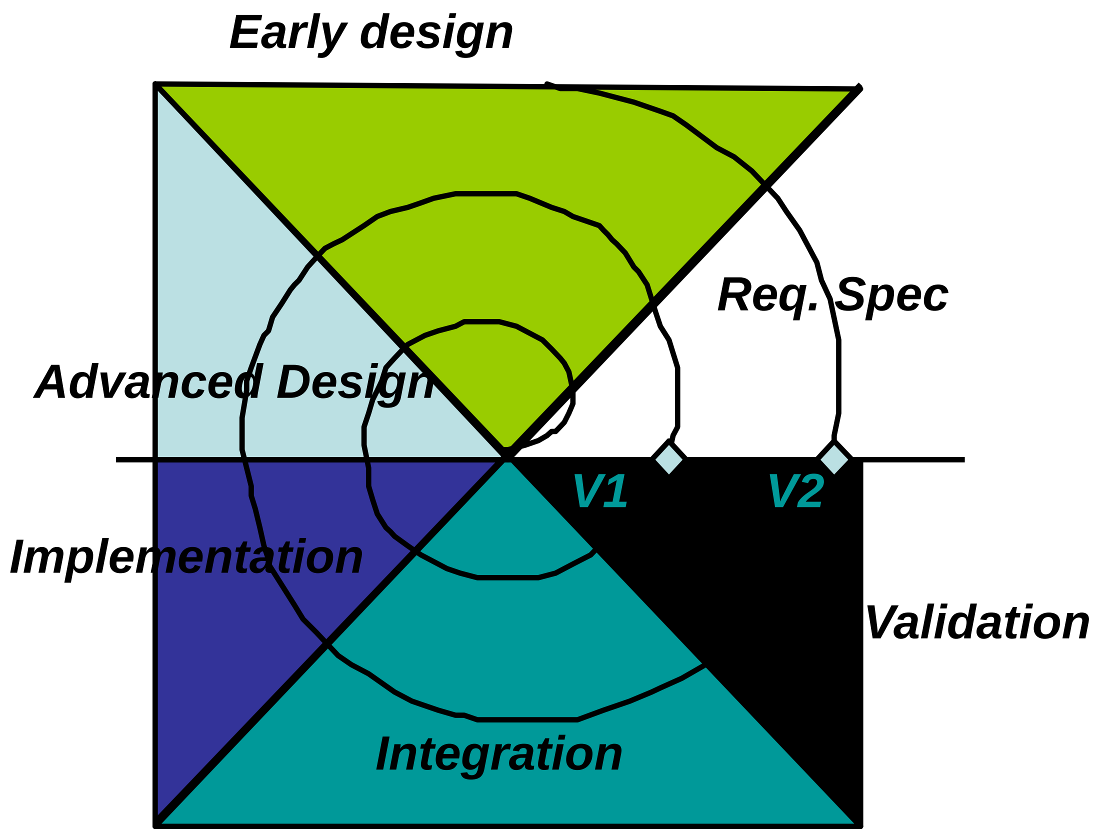
a) Test plan identifier
b) Introduction
c) Test items
d) Features to be tested
e) Features not to be tested
f) Approach
g) Item pass/fail criteria
h) Suspension criteria and resumption requirements
i) Test deliverables
j) Testing tasks
k) Environmental needs
l) Responsibilities
m) Staffing and training needs
n) Schedule
o) Risks and contingencies
p) Approvals

Let’s zoom in the different kinds of tests

Test Classification : Reminder



Or the « spiral » life cycle

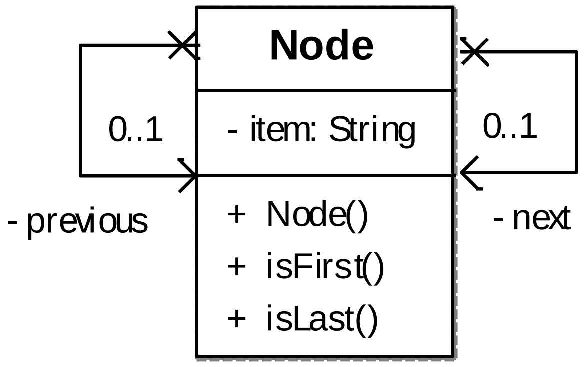


Unit Testing

- **To validate a module independently of the others**
- To validates the different functions/operations intensively
- Are the functions /operations ok?
- Is the code readable? maintainable ?

Unit Testing

- **In OO programming**
  - Unit= class



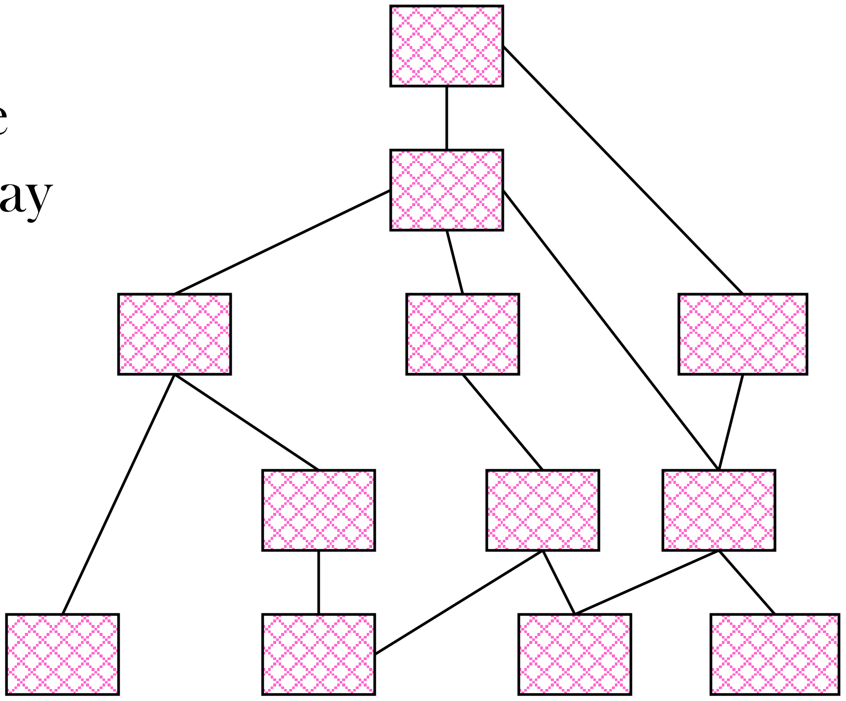
Integration Testing

- Continuously testing the behavior of the system every time a new module is integrated
- Issue: choosing the right integration order!

Integration Testing

- **Simple case: no cycles in modules dependencies**

- Dependencies form a Tree and the integration is done in a bottom-up way

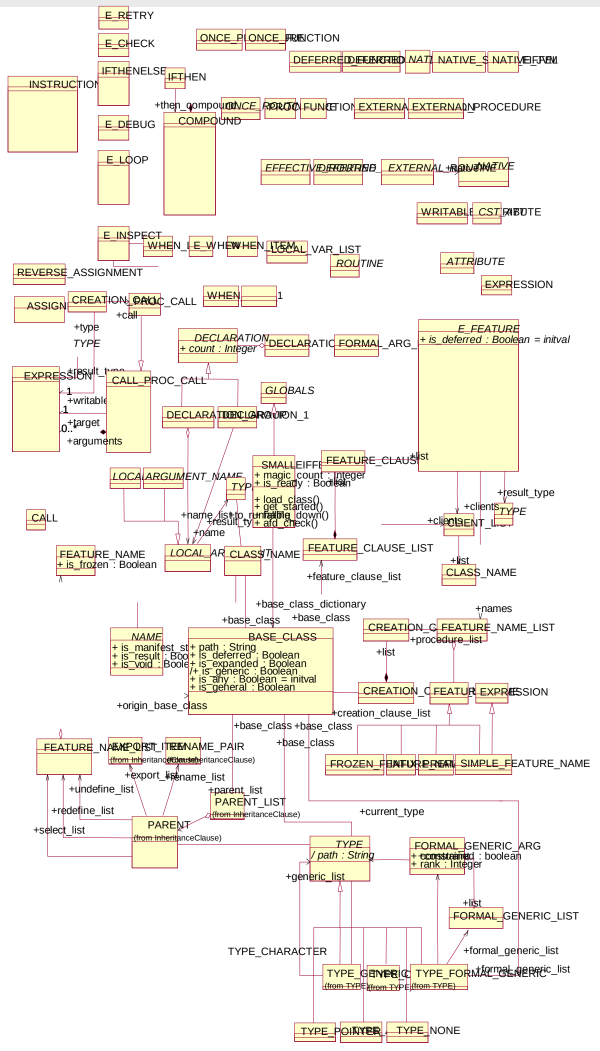


Integration Testing

- Worst case: presence of cycles between modules dependencies

- Often the case in OO design

- Need to define heuristics to find the best integration order



System Testing

- To validate the global behavior of the system
  - Check the that functionalities meet the requirements spec.
  - Check in the interfacing with other systems
  - Check the GUI

No-regression testing

- The purpose of non-regression testing is to verify whether, after introducing or updating a given software application, the change has had the intended effect
  - What worked before, still works fine!
- Usually in maintenance phase
  - After refactoring, adding/deleting a functionality
- After the correction of an error/bug

Unit Testing

Unit Testing in OO

- Testing a unit independently of the others

  A unit=> Class

- It's a Client's point of view code
  - Test Plans call methods to be tested from the outside (external client)
  - Only public methods can be tested (or use reflection)
    - Hints for testing private methods: http://www.artima.com/suiterunner/privateP.html
  - Testing a class, is done by an external class
- At least one test by public method
- Need to define a test order!!
  - What are the dependencies between the methods?

Unit test case

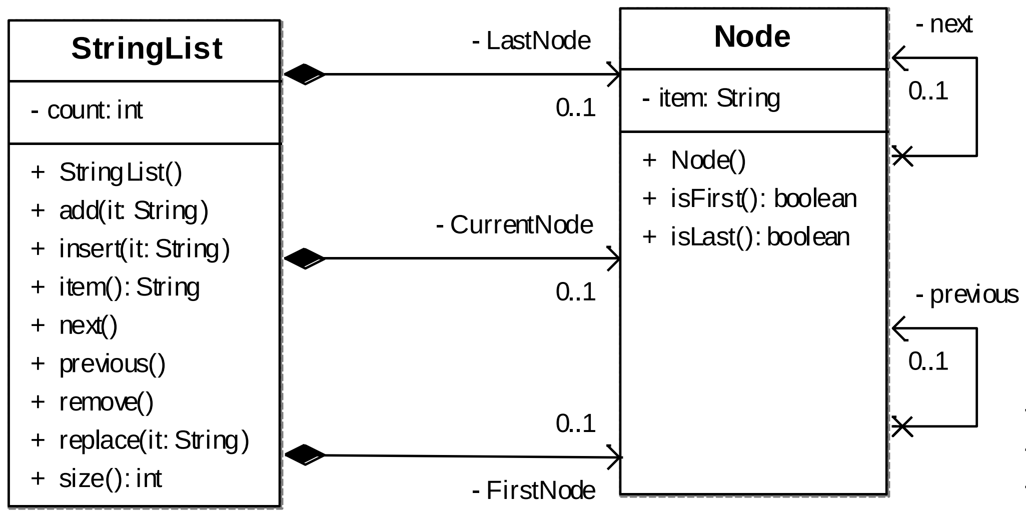
- **Test case= a method**

- Body of the method
  - Initial configuration
  - Test data
    - Parameters used to test the method
  - An oracle
    - Specify the expected result
    - Or check some properties on the expected result

- One test class for each class to be tested
  - Regroup all test cases for the class

Example

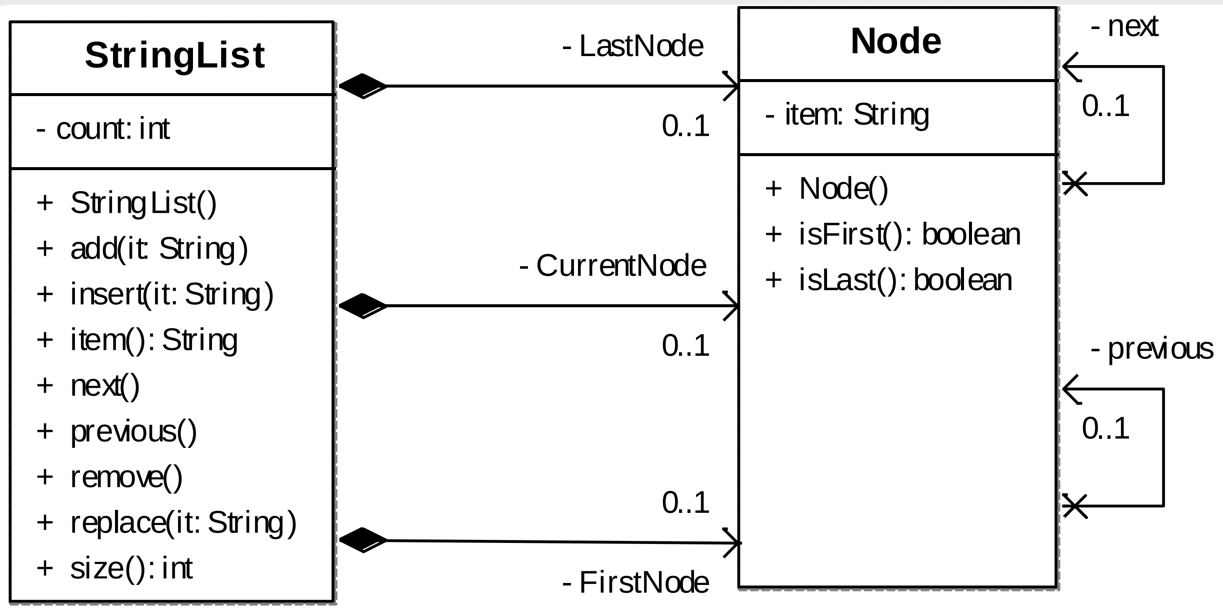
How to test StringList?



- •Test adding an element to an empty list
- •Test deleting an element from a list
- •Test deleting the last element of a list
- •Etc….

How to write these tests?
How to execute them?
Are they good?
Is it enough tested?
...

Example : testing the StringList class



- Create a test class to manipulate StringList class's instances

- One test case by each public method

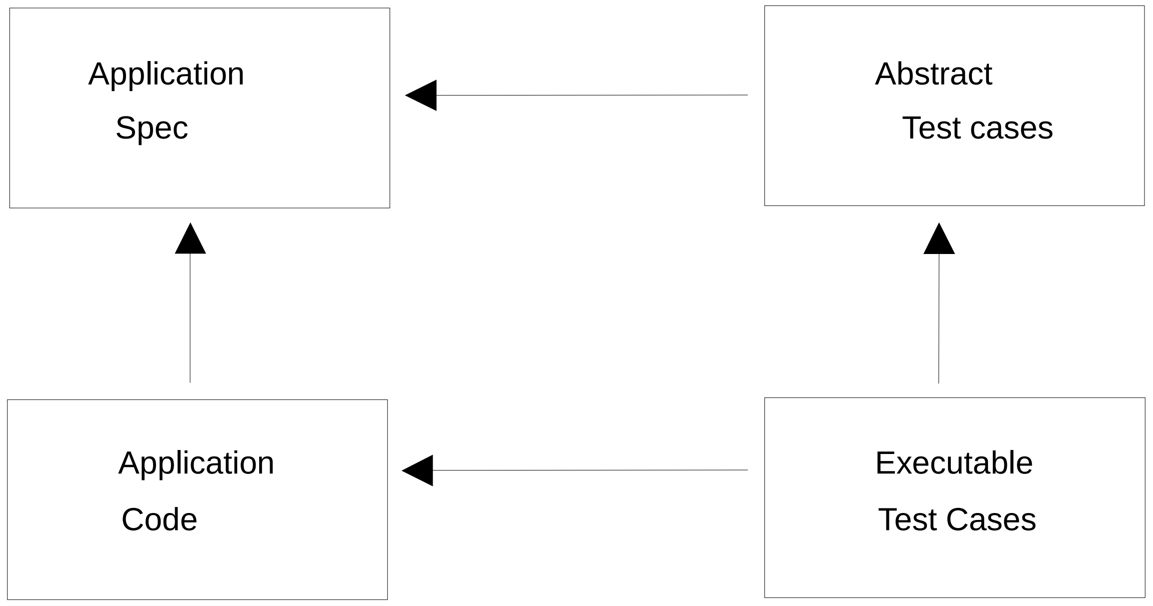
Tests: abstraction levels

- **Abstract test cases:** you can use UML Sequence diagrams
  - **Goal:** to start reasoning about tests in earlier development phases

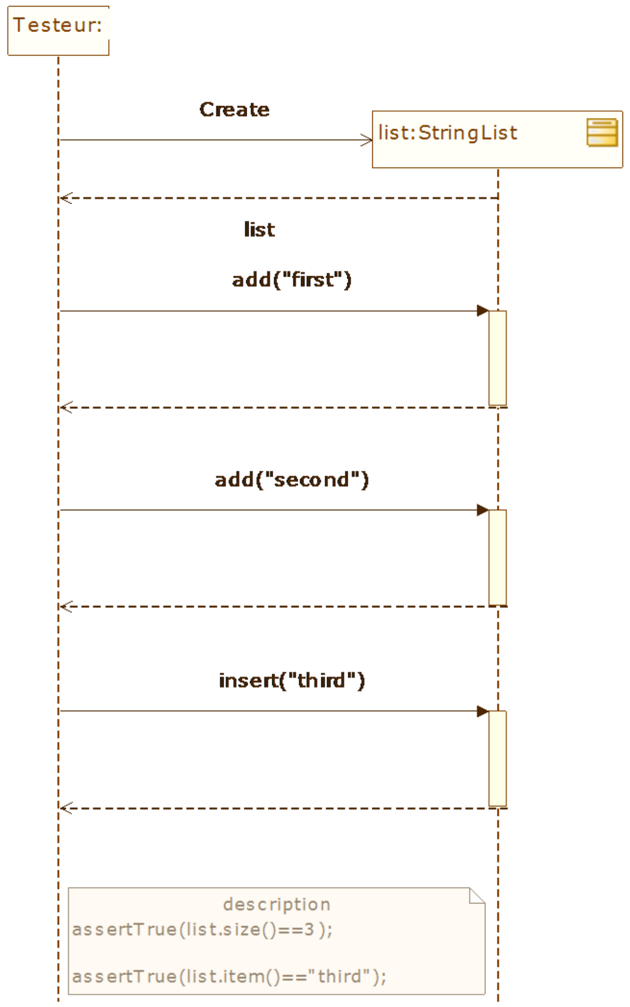
- **Executable test cases** : use of x-unit framework to concretely execute tests

Abstract Test Cases

Application & Test



Abtract Test Case: Example
test of StringList- the insert() method



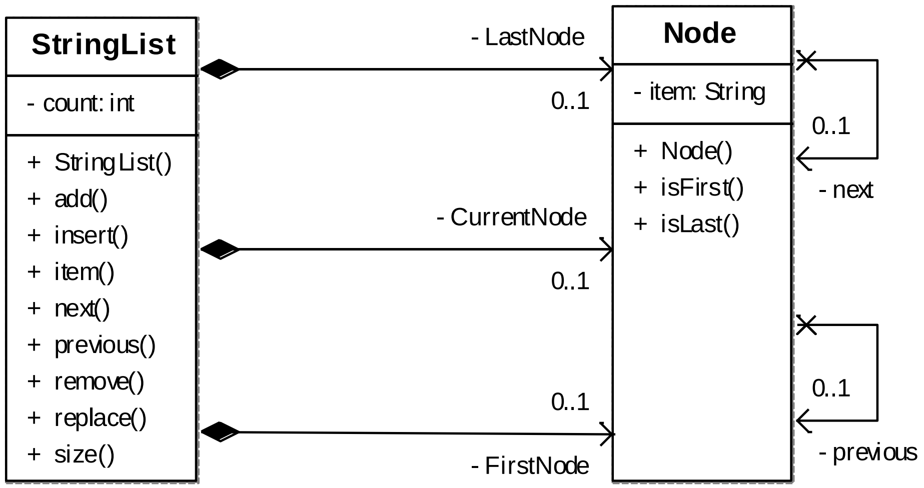
Executable test cases

JUnit

- Origin
  - Xtreme programming (test-first development)
  - Java framework by E. Gamma et K. Beck
  - open source: www.junit.org

- Goals
  - Testing Java applications
  - Non-Regression testing

Executable test case
Testing StringList- the insert() method



Intention { **Test case specification**

```
//first test for insert: call insert and see if
//current element is the one that's been inserted
public void testInsert1(){
  StringList list = new StringList();
  list.add("first");
  list.add("second");
  list.insert("third");
  assertTrue(list.size()==3);
  assertTrue(list.item()=="third");
}
```

initialisation { `StringList list = new StringList(); list.add("first"); list.add("second");`

Call with a data { `list.insert("third");`

oracle { `assertTrue(list.size()==3); assertTrue(list.item()=="third");`

Conclusion

- Its a domain by its own! Very important field of research
- Very important in the development life cycle
- Each phase has its own tests
- Unit testing
  - Unfortunately doesn't cover all the possibilities
  - Increase customer's trust in the application
  - Even if you cover 99% , the 1% left can be source a damageable bug!!!

Lectures

- Software Engineering,
  - Ian Sommerville, Addison Wesley; 8 edition (15 Jun 2006), ISBN-10: 0321313798
- The Mythical Man-Month
  - Frederick P. Brooks JR., Addison-Wesley, 1995
- Cours de Software Engineering du Prof. Bertrand Meyer à cette @:
  - http://se.ethz.ch/teaching/ss2007/252-0204-00/lecture.html
- Cours d'Antoine Beugnard à cette @:
  - http://public.enst-bretagne.fr/~beugnard/

- UML Distilled 3rd édition, a brief guide to the standard object modeling language
  - Martin Fowler, Addison-Wesley Object Technology Series, 2003, ISBN-10: 0321193687
- UML2 pour les développeurs, cours avec exercices et corrigés
  - Xavier Blanc, Isabelle Mounier et Cédric Besse, Edition Eyrolles, 2006, ISBN-2-212-12029-X
- UML 2 par la pratique, études de cas et exercices corrigés,
  - Pascal Roques, 6ème édition, Edition Eyrolles, 2008
- Cours très intéressant du Prof. Jean-Marc Jézéquel à cette @:
  - http://www.irisa.fr/prive/jezequel/enseignement/PolyUML/poly.pdf
- La page de l'OMG dédiée à UML: http://www.uml.org/

- Design patterns. Catalogue des modèles de conception réutilisables
  - Richard Helm (Auteur), Ralph Johnson (Auteur), John Vlissides (Auteur), Eric Gamma (Auteur), Vuibert informatique (5 juillet 1999), ISBN-10: 2711786447
- **Cours sur les tests est basé sur les Cours très complets et bien faits de Yves le Traon et Benoit Baudry**