



*Cycle Ingénieur – 2<sup>ème</sup> année*  
*Département Informatique*

# Verification and Validation

## Part IV : System Test I

Burkhart Wolff

Département Informatique  
Université Paris-Sud / Orsay

# Difference between Validation and Verification

---

- ❑ Validation :
  - Does the system meet the clients requirements ?
  - Will the performance be sufficient ?
  - Will the usability be sufficient ?

# Difference between Validation and Verification

---

- ❑ Validation :
  - Does the system meet the clients requirements ?
  - Will the performance be sufficient ?
  - Will the usability be sufficient ?

*Do we build the right system ?*

# Difference between Validation and Verification

---

- ❑ Validation :
  - Does the system meet the clients requirements ?
  - Will the performance be sufficient ?
  - Will the usability be sufficient ?

*Do we build the right system ?*

- ❑ Verification: Does the system meet the specification ?

# Difference between Validation and Verification

---

- ❑ Validation :
  - Does the system meet the clients requirements ?
  - Will the performance be sufficient ?
  - Will the usability be sufficient ?

*Do we build the right system ?*

- ❑ Verification: Does the system meet the specification ?

*Do we build the system right ?*

*Is it « correct » ?*

# How to do Validation ?

---

- ❑ Measuring customer satisfaction ...  
(well, that's afterwards, and its difficult)
- ❑ Interviews, inspections (again post-hoc)
- ❑ How to validate a system early?
  - early prototypes, including performance analysis ...
  - mock-ups (fonctionnality, ergonomics,...)
  - **Test and Animation** on the basis of formal specifications  
(e.g., à la OCL !)

# How to do Verification ?

---

- **Test and Proof** on the basis of formal specifications (e.g., à la OCL !) against programs ...

# How to do Verification ?

---

- Test and Proof on the basis of formal specifications (e.g., à la OCL !) against programs ...

In the sequel, we concentrate on Testing and Proof Techniques ...

# A Philosophical Position Statement : Test vs. Proof

---

## □ Note:

Some researcher consider test as **opposite** to formal proof! Reasons:

- “A test can only reveal the presence of bugs, but not their absence” (Dijkstra, v. Dalen)
- ... these researchers referred to unsystematic tests ... (which are, admittedly, still quite common in SE practice)

# A Philosophical Position Statement : Test vs. Proof

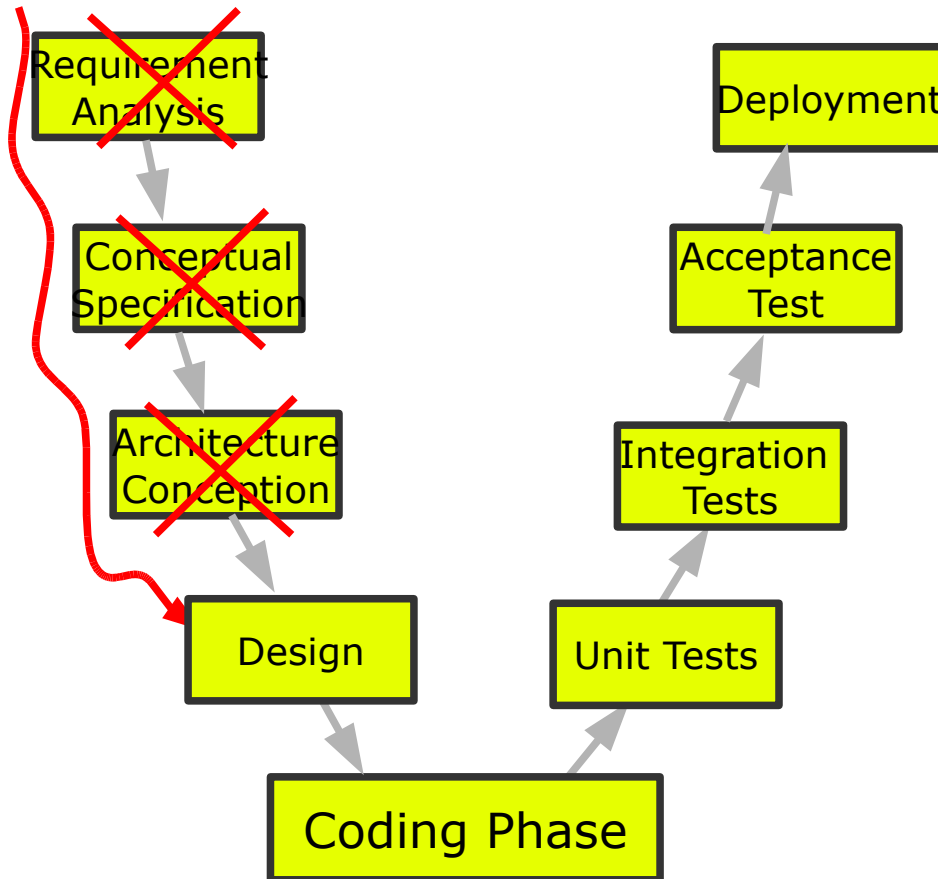
---

## □ Note:

We consider (systematic!) test more as an **approximation** to formal proof. Reasons:

- The nature of the approximation can be made formally precise (via explicit test-hypothesis ...)
- both techniques, model-based tests and formal verification, share a lot of technologies ...
- even full-blown proof attempts may profit from testing, since it can help to debug specs early and cost-effectively

# Test in the SE Process

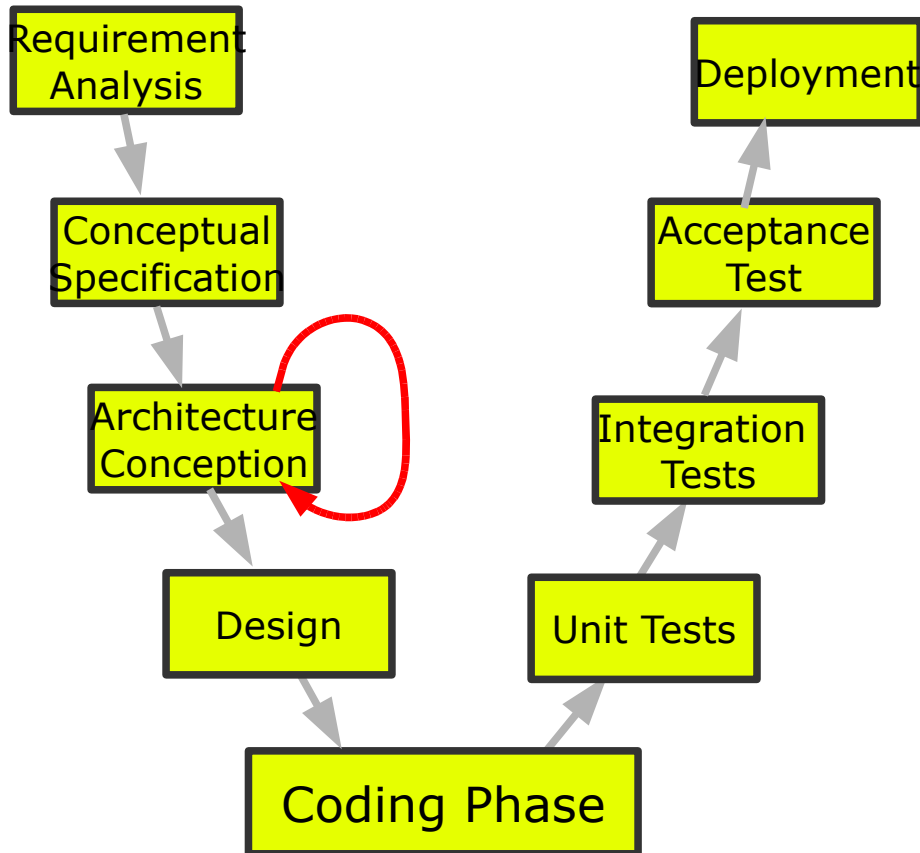


## Where to integrate Tests in the SE-Process:

- On the methodological level, à la “Extreme Programming” (XP) ?  
No specs, instead writing test scenarios and test cases from the beginning ...

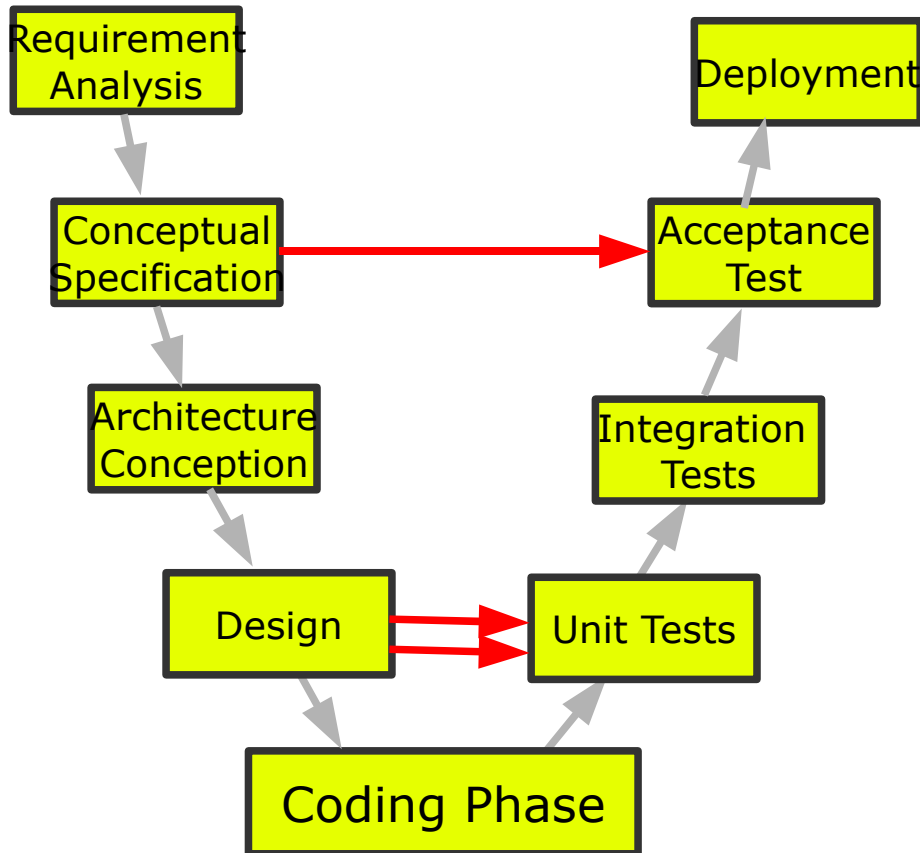
# Test in the SE Process

---



- Where to integrate Tests in the SE-Process:
  - On the methodological level, à la “Extreme Programming” (XP) ?  
No specs, instead writing test scenarios and test cases from the beginning ...
  - On the **specification level** for validation ...

# Test in the SE Process



- Where to integrate Tests in the SE-Process:
  - On the methodological level, à la “Extreme Programming” (XP) ?  
No specs, instead writing test scenarios and test cases from the beginning ...
  - On the specification level for validation ...
  - On the **specification level against code**

# Test in the SE Process

---

- General questions for verification in a process:
  - How to select test-data ? To which purpose ?>
  - How to focus verification activities?  
Where to verify formally, and  
where to test, and when did we test enough?
  - Note: The quality of a test does not increase necessarily by the number of test-cases !**
  - Automation ? Tools ?

# Some empirical data ...

---

- ❑ Size of Software ?
  - Peugeot 607 : 2 Mb embedded software
  - Windows 90: 10 Mb. LOC source, Win2000: 30 Mb.
  - Kernel Hyper V: 50000 LOC. (Highly complex, concurrent C)
  - Noyau RedHat 7.1 (2002) : ~2.4 M. LOC, XWindow ~1.8, Mozilla ~2.1 M.
  - Space Shuttle (and its environment) : ~50 MLOC
  
- ❑ Reminder: Development Cost ?
  - Percentage of «Coding» ? 15 - 20 %  
Trend: Code is more and more generated (*CASE Tools*)
  - Proportion of Validation et Verification ? ~20% / ~20%

# Verification Costs

---

- ❑ costs ?                      *35 - 50 % of the global effort ?*
- ❑ all “real” (large) software has remaining bugs ...
- ❑ The cost of bug ?
  - the cost to reveal and fix it ...  
or:
    - the cost of a legal battle it may cause...
    - or the potential damage to the image  
(difficult to evaluate, but veeeery real)
    - or costs as a result to come later on the market
  - *on the other side – you can't test infinitely, and verification is again 10 times more costly than thoroughly testing !*

# Verification Costs

---

- ❑ Conclusion:
  - verification is vitally important, and also critical in the development
  - to do it cost-effectively, it requires
    - ❑ a lot of expertise on products and process
    - ❑ a lot of knowledge over methods, tools, and tool chains ...

# Overview on the part on « Test »

---

- ❑ WHAT IS TESTING ?
- ❑ A taxonomy on types of tests
  - Static Test / Dynamic (*Runtime*) Test
  - Structural Test / Functional Test
  - Statistic Tests
- ❑ Functional Test; Link to UML/OCL
  - Dynamic Unit Tests, Static Unit Tests,
  - Coverage Criteria
- ❑ Structural Tests
  - Control Flow and Data Flow Graphs
  - Tests and executed paths. Undecidability.
  - Coverage Criteria

# What is testing ?

---

- ❑ It is an approximation to verification
- ❑ Main emphasis: finding bugs early,
  - either in the model
  - or in the program
  - or in both
- ❑ A **systematic** test is:
  - process programs and specifications and to compute a set of test-cases under controlled conditions.
  - ***ideally***: testing is complete if a certain criteria, the adequacy criteria is reached.

# Limits of testing ?

---

- ❑ We said, test is an approximation to verification, usually easier (and less expensive)
  
- ❑ Note: Sometimes it is easier to verify than to test. In particular:
  - low-level OS implementations:  
memory allocation, garbage collection  
memory virtualization, ...  
crypt-algorithms, ...
  
  - non-deterministic programs with  
no control over the non-determinism.

# Taxonomy: Static / Dynamic Tests

---

- ❑ **static:** running a program before deployment on data carefully constructed by the analyst (in a testing environ.)
  - analyse the result on the basis of all components
  - working on some classes of executions symbolically = representing infinitely many executions
  
- ❑ **dynamic:** running the programme (or component) after deployment, on “real data” as imposed by the application domain
  - experiment with the real behaviour
  - essentially used for post-hoc analysis and debugging

# Taxonomy: Unit / Sequence / Reactive Tests

---

- ❑ **unit**: testing of a local component (function, module), typically only one step of the underlying state. (In functional programs, that's essentially all what you have to do!)
- ❑ **sequence**: testing of a local component (function, module), but typically sequences of executions, which typically depend on internal state
- ❑ **reactive sequence**: testing components by sequences of steps, but these sequences represent communication where later parts in the sequence depend on what has been earlier communicated

# Taxonomy: Functional / Structural Test

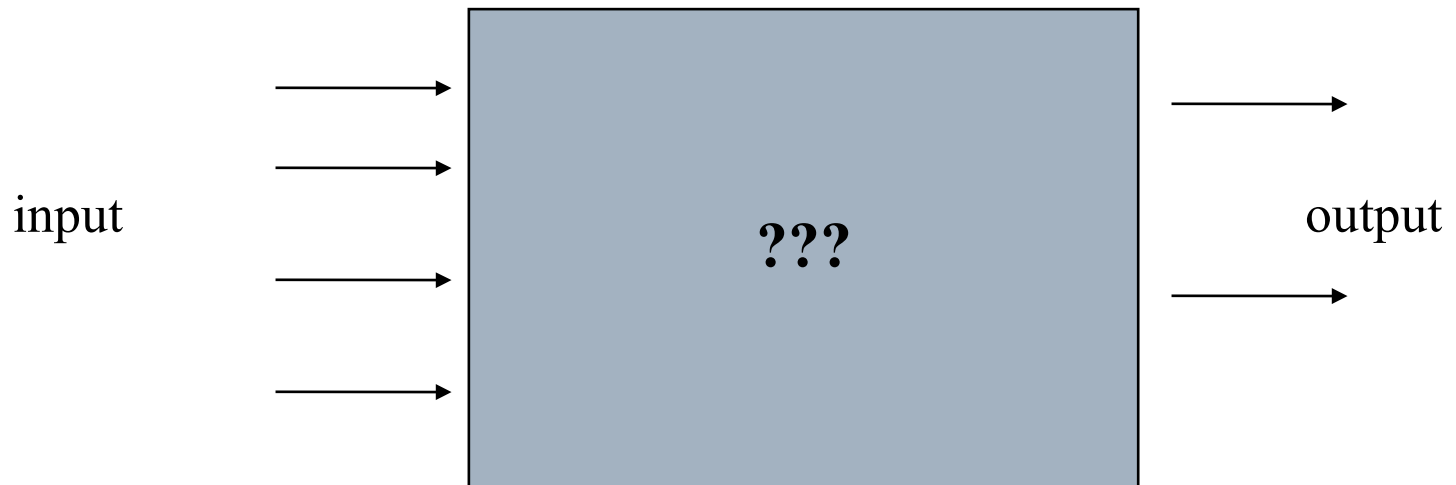
---

- ❑ **functional:** (also: black-box tests). Tests were generated on a specification of the component, the test focusses on input output behaviour.
- ❑ **structural:** (also: white-box tests). Tests were generated on the basis of the structure or the program, i.e. using control-flow, data-flow paths or by using symbolic executions.
- ❑ **both:** (also: grey-box testing).

# Functional Dynamic Unit Test

---

- ❑ We got the spec, but not the program, which is considered a black box:



we focus on what the program *should* do !!!

# Functional Dynamic Unit Test : an example

---

The (informal) specification:

*Read a "Triangle Object" (with three sides of integral type), and test if it is isoscele, equilateral, or (default) arbitrary.*

*Each length should be strictly positive.*

Give a specification, and develop a test set ...

# Functional Dynamic Unit Test : an example

---

The specification in UML/OCL (Classes in USE Notation):

```
class Triangles inherits_from Shapes
  attributes
    a : Integer
    b : Integer
    c : Integer

  operations
    mk(Integer,Integer,Integer):Triangle
    is_Triangle(): triangle
end
```

# Functional Dynamic Unit Test : an example

---

The specification in UML/OCL (Classes in USE Notation):

**context** Triangles:

**inv** def : a.oclIsDefined() and b.oclIsDefined()...

**inv** pos :  $0 < a$  and  $0 < b$  and  $0 < c$

**inv** triangle :  $a + b > c$  and  $b + c > a$  and  $c + a > b$

**context** Triangle::isTriangle()

**post** equi :  $a = b$  and  $b = c$  implies result=equilateral

**post** iso :  $((a <> b$  or  $b <> c)$  and  
 $(a = b$  or  $b = c$  or  $a = c))$  implies result=isosceles

**post** default:  $(a <> b$  or  $b <> c)$  and  
 $(a <> b$  and  $b <> c$  and  $a <> c)$   
implies result=arbitrary

# Functional Dynamic Unit Test : an example

---

The specification in UML/OCL (Classes in USE Notation):  
Recall implicit consequences due to strictness of all operations

```
context Triangle::isTriangle()  
...  
post res_strict: self.oclIsUndefined implies  
                    result.oclIsUndefined  
post res_total: result.oclIsDefined implies  
                    self.oclIsDefined
```

# Functional Dynamic Unit Test : an example

---

*How to perform Runtime-Test?*

*Well, compile:*

**context** X:

**inv**  $l_1 : C_1, \dots,$

**inv**  $l_n : C_n$

*to some checking code (with assert as in Junit, VCC, Boogie, ...)*

```
check_X() = assert(C1); ... ; assert(Cn)
```

# Functional Dynamic Unit Test : an example

---

*How to perform Runtime-Test?*

*Moreover, compile:*

```
context C::m(a1:C1, ..., an:Cn)  
pre      : P(self, a1, ..., an)  
post     : Q(self, a1, ..., an, result)
```

*to some checking code (with assert as in Junit, VCC, Boogie, ...)*

```
check_C(); check_C1(); ... ; check_Cn();  
assert(P(self, a1, ..., an));  
result=run_m(self, a1, ..., an);  
assert(Q(self, a1, ..., an, result));
```

# Functional Dynamic Unit Test : Problems

---

- ❑ Thus, any violation of an invariant, a pre-condition or a post-condition is detected.
- ❑ If a violation occurs within an execution of a method, the error is precisely reported.
- ❑ On the other hand – it is **post-hoc**. Only when a problem occurred, we know where. And we need **complete** program.
- ❑ Inefficiencies can be partly overcome by optimized compilations.