V&V

Main Phases

Development

Requirements definition

Requirement document

Req. inspection

Design

Design document

Des. inspection

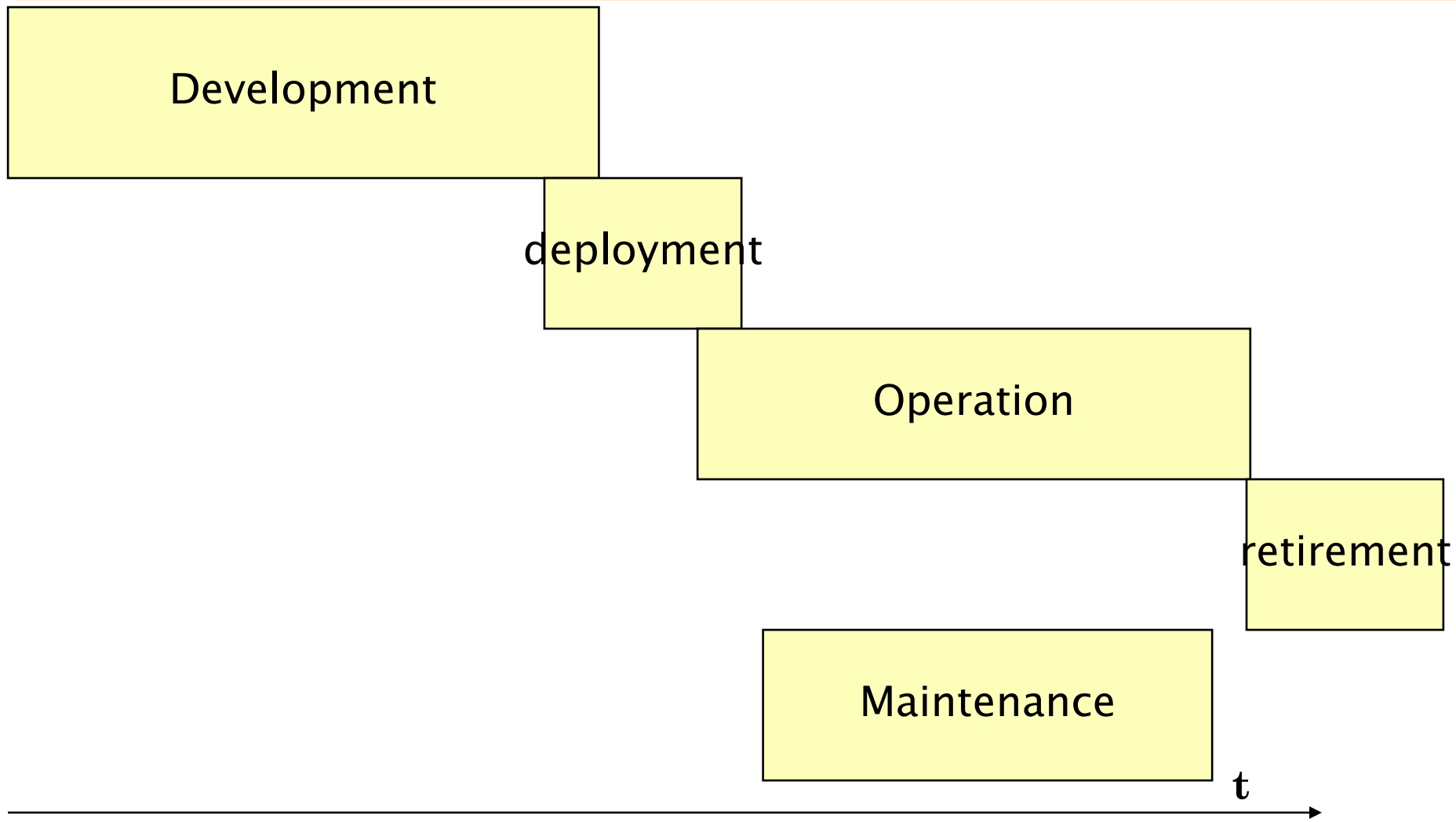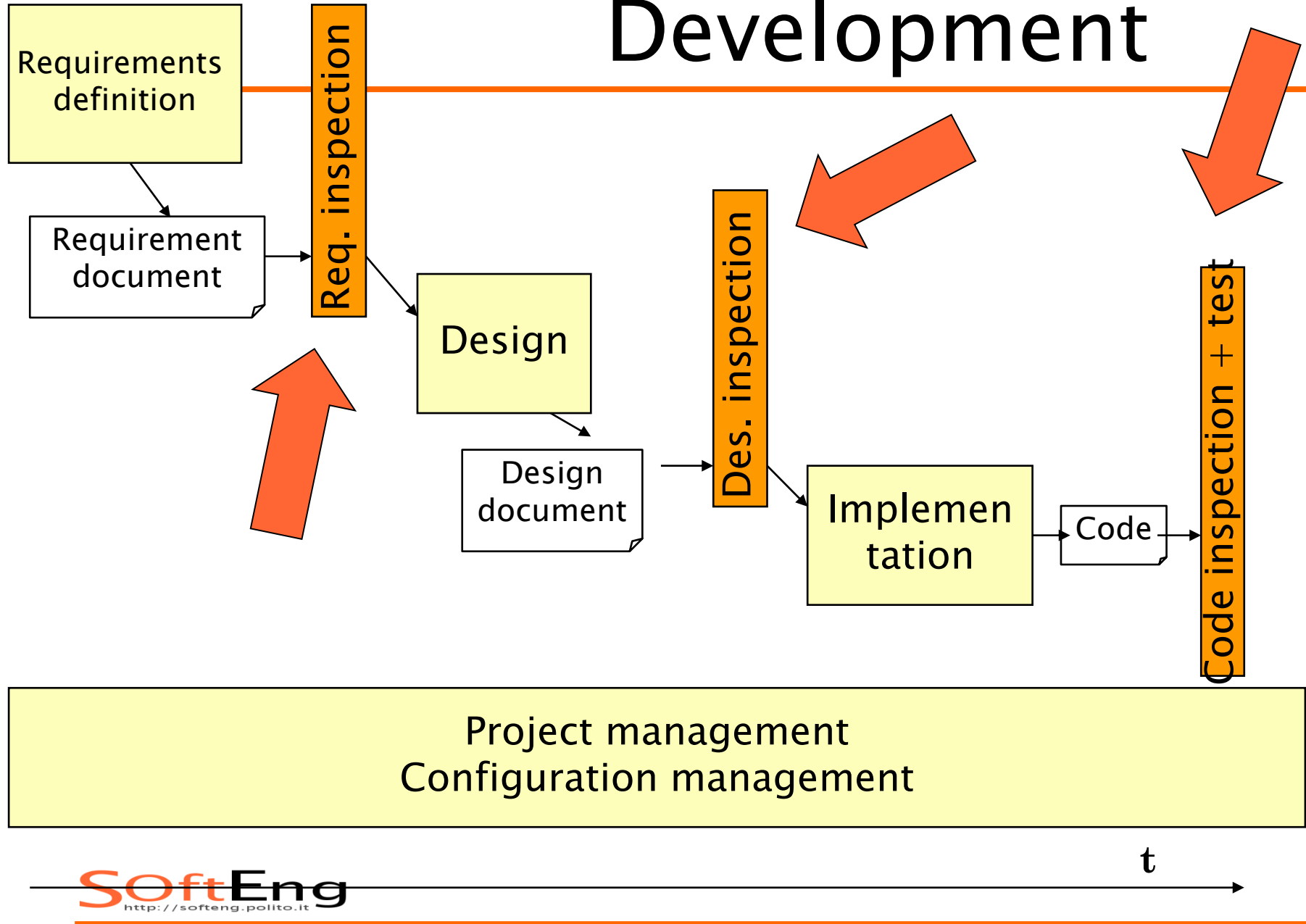Implementation

Code

Code inspection + test

Project management
Configuration management
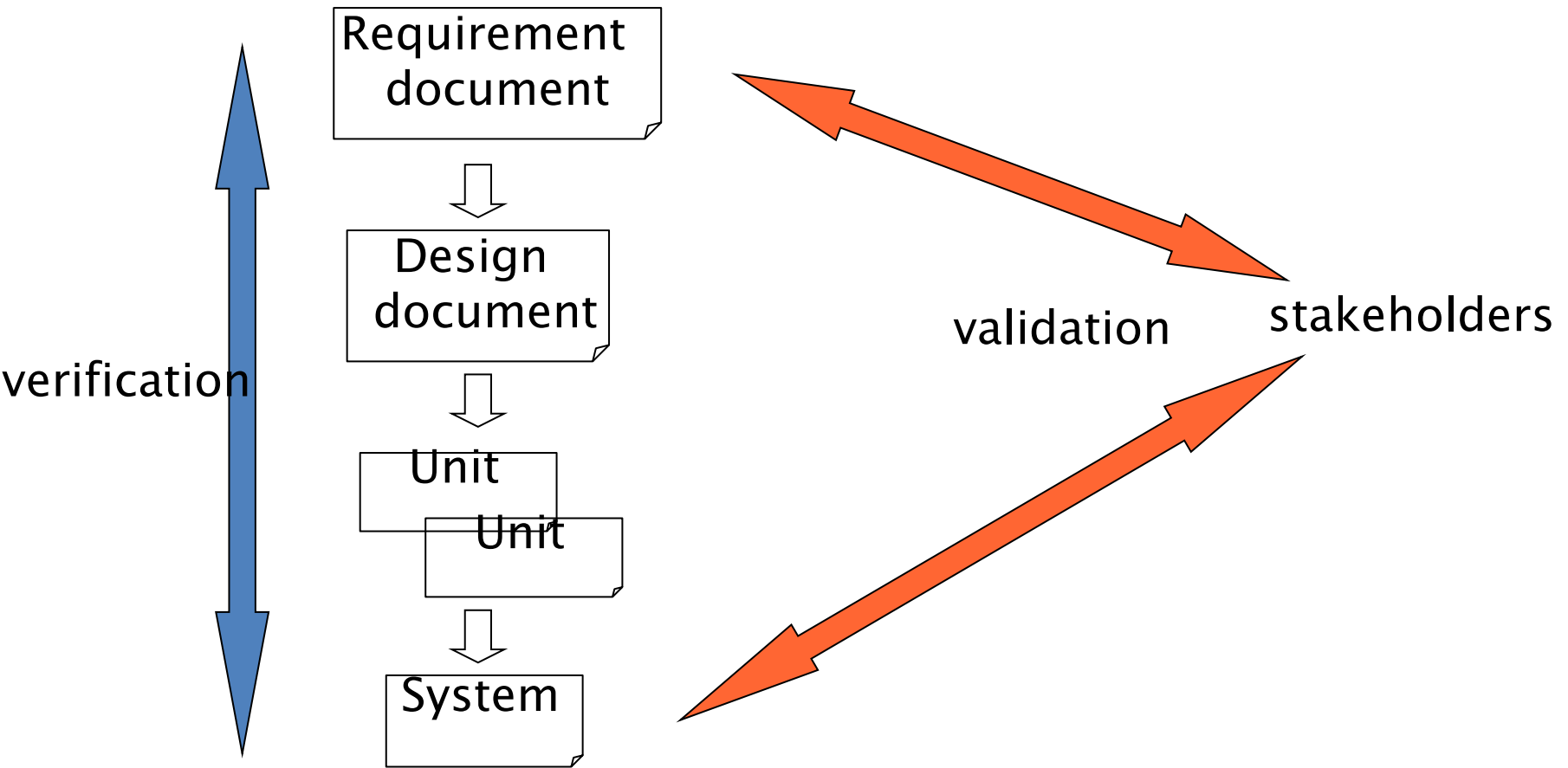
t

V&V

- Validation
  - is it the right software system?
  - effectiveness
  - external (vs user)
  - reliability
- Verification
  - is the software system right?
  - efficiency
  - internal (correctness of vertical transformations)
  - correctness

V & V

Scenario1 in a dev process

- **Stakeholder**
  - ◆ Real need: big car /6 seats

- **Developers**
  - ◆ R1: compact car (4 seats)

  - ◆ Result : compact car (4 seats)
    - – Verification: passed
    - – Validation : not passed

SOftEng
http://softeng.polito.it

Scenario2 in a dev process

- **Stakeholder**
  - ◆ Real need: big car /6 seats

- **Developers**
  - ◆ R1: big car (6 seats)

  - ◆ Result : big car (6 seats)
    - – Verification: passed
    - – Validation : passed

SOftEng
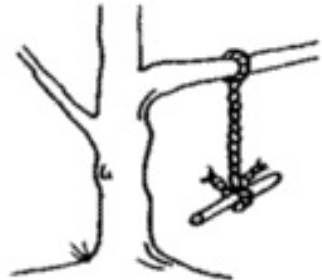http://softeng.polito.it

Scenario3 in a dev process

- Stakeholder
  - Real need: big car /6 seats

- Developers
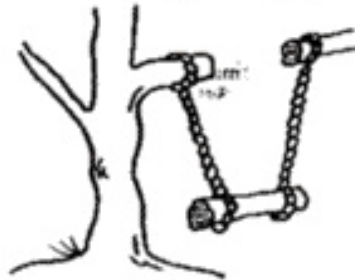 - R1: big car (6 seats)

  - Result : compact car (4 seats)
    – Verification: not passed
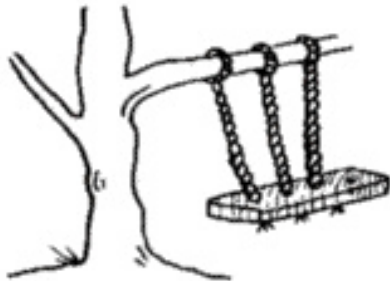   – Validation : passed on req doc, not passed on delivered product

Requirements

- The root…

What the user asked for
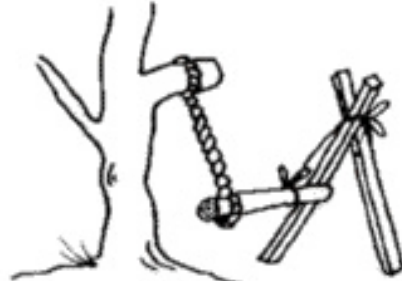
How the analyst saw it

How the system was designed

As the programmer wrote it
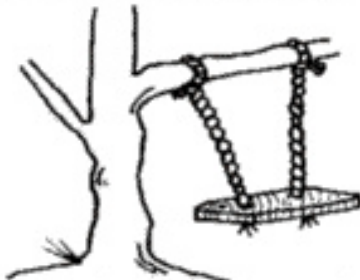
What the user really wanted

How it actually works

…of all evils

Traceability

if(i = j){

...

Requirement

Design Item

Code Fragment

Use case

Test case

Test result

Failure

V & V vs. cost of fixing defect

Requirement document → Design document → Unit / Unit → System
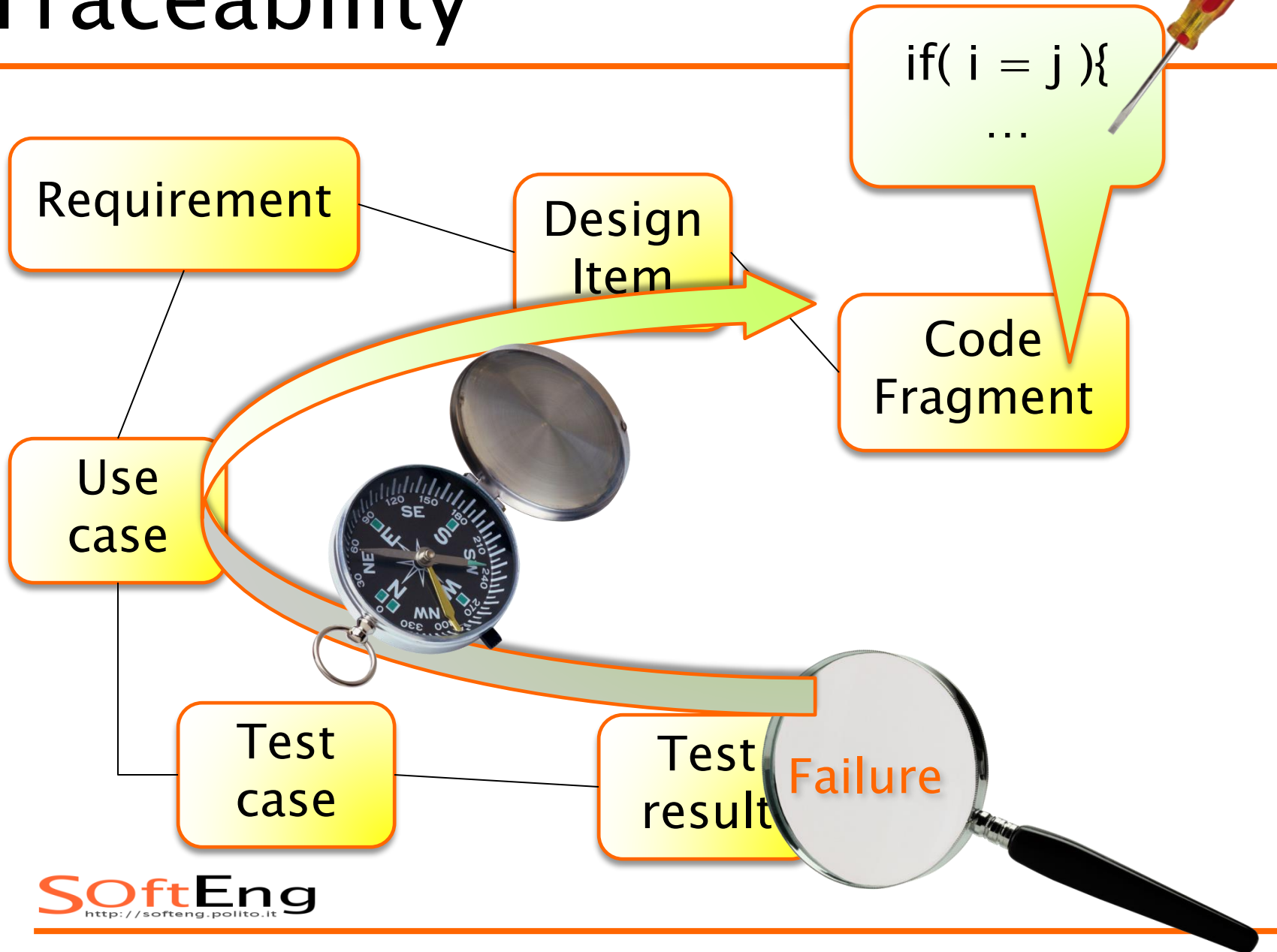
Cost of fixing defect
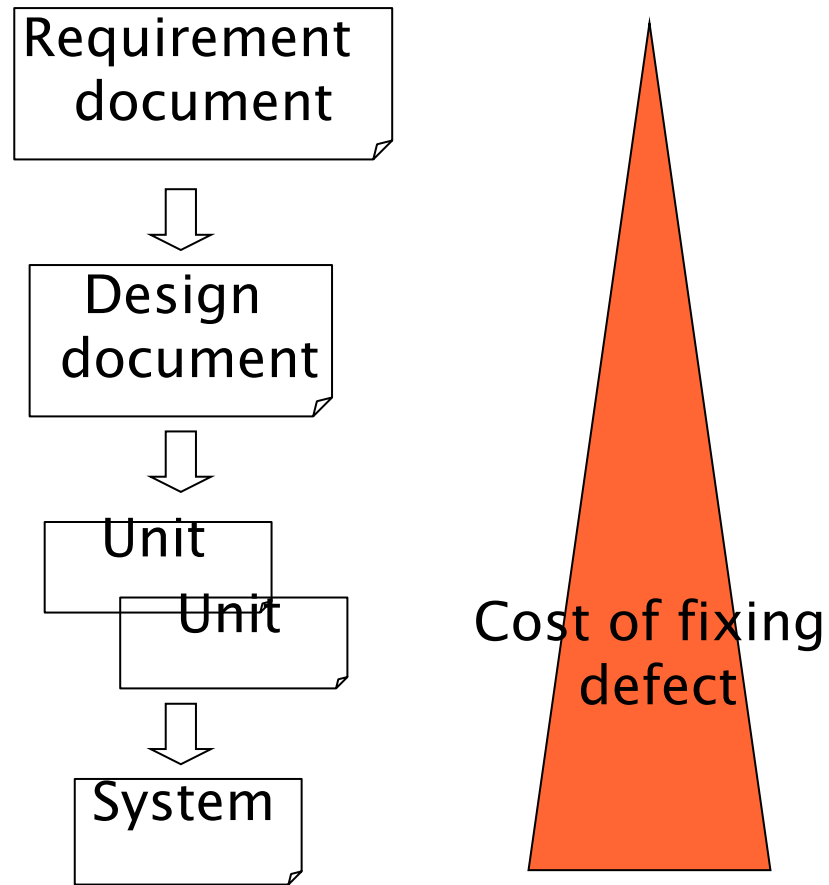
Failure, fault, defect

- Failure
  - An execution event where the software behaves in an unexpected way
- Fault
  - The feature of software that causes a failure
  - May be due to:
    - An error in software
    - Incomplete/incorrect requirements
- Defect
  - Failure or fault

Bicycle

- Failure
  - User falls down
  - R1 not satisfied

- Why?

Requirements

R1 transport person

Bicycle

- Fault
  - Handlebar ill mounted? (implementation pb)
  - Saddle too high? (configuration pb)
  - Wheels too far apart? (design problem)

  - (where the defect comes from?)

Failure fault defect
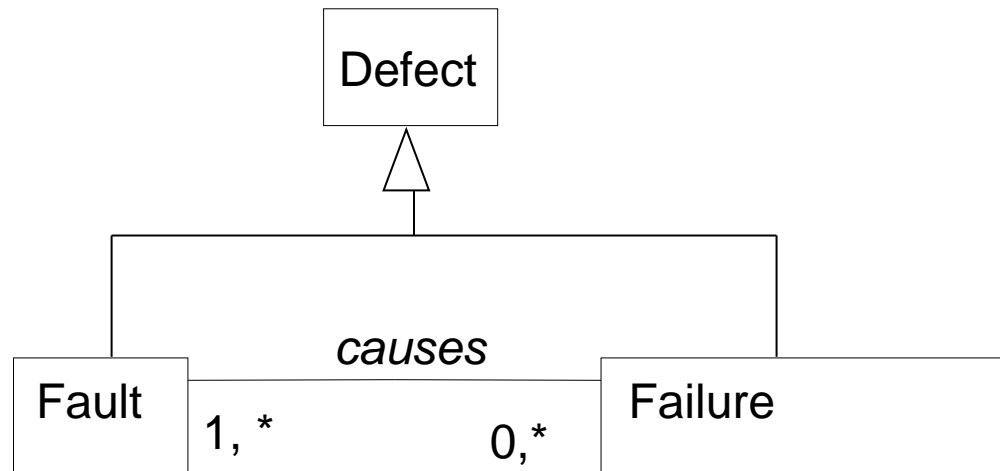
Insertion / removal

- Defect is characterized by
  - Insertion activity (phase)
  - Removal activity (phase)

insertion

discover
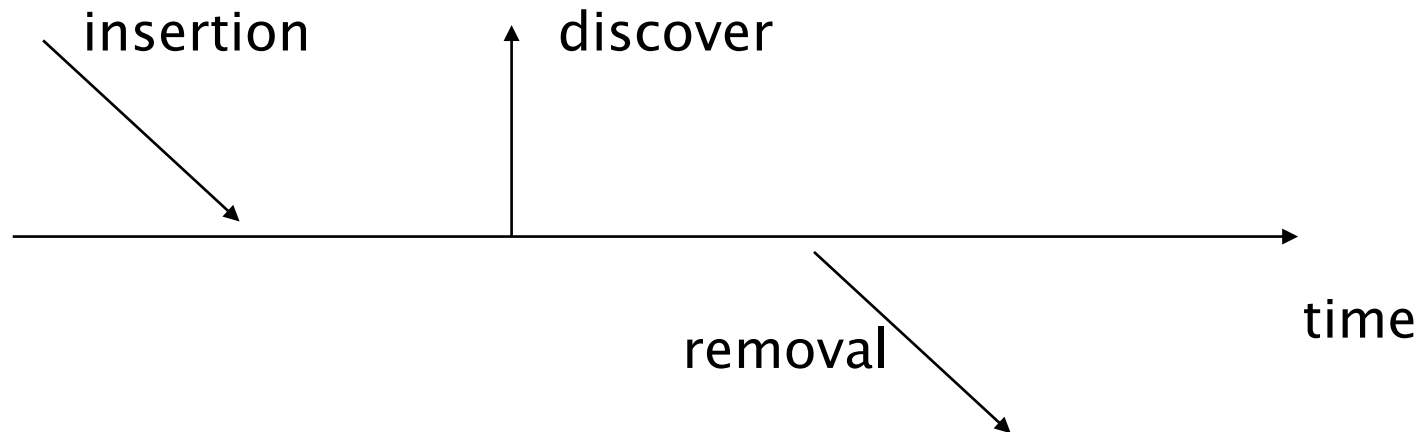
removal

time

Basic goal of VV

- Minimize number of defects inserted
  - Cannot be zero due to inherent complexity of software
- Maximize number of defects discovered and removed
- Minimize time span between insertion and discover and removal

SOftEng
http://softeng.polito.it

Insertion/removal by phase – typical scenario

Defects/KLOC

A bar chart showing Injected (blue) and Discovered (red) defects per KLOC across phases:

Phase	Injected	Discovered
Requirements	8	3
Design	16	8
Code	14	11
Unit test	7.8	10
Integration test	2	7.6
System test		4.2
Operation		3

SOftEng
http://softeng.polito.it
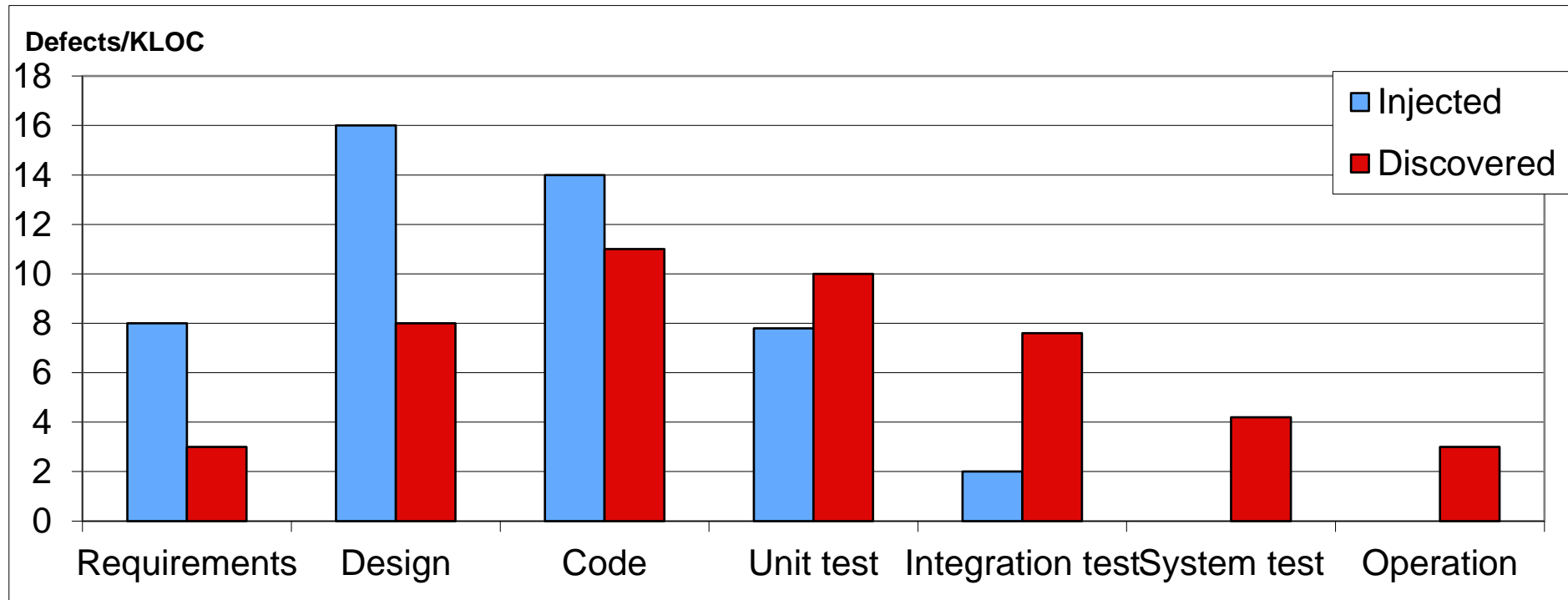
Rework Problem

- The longer the delay insert-remove, the higher the cost of removing defect
- Avoidable rework accounts for 40-50% of development [Boehm, 1987; Boehm&Basili, 2001]
  - More recent data available at www.cebase.org
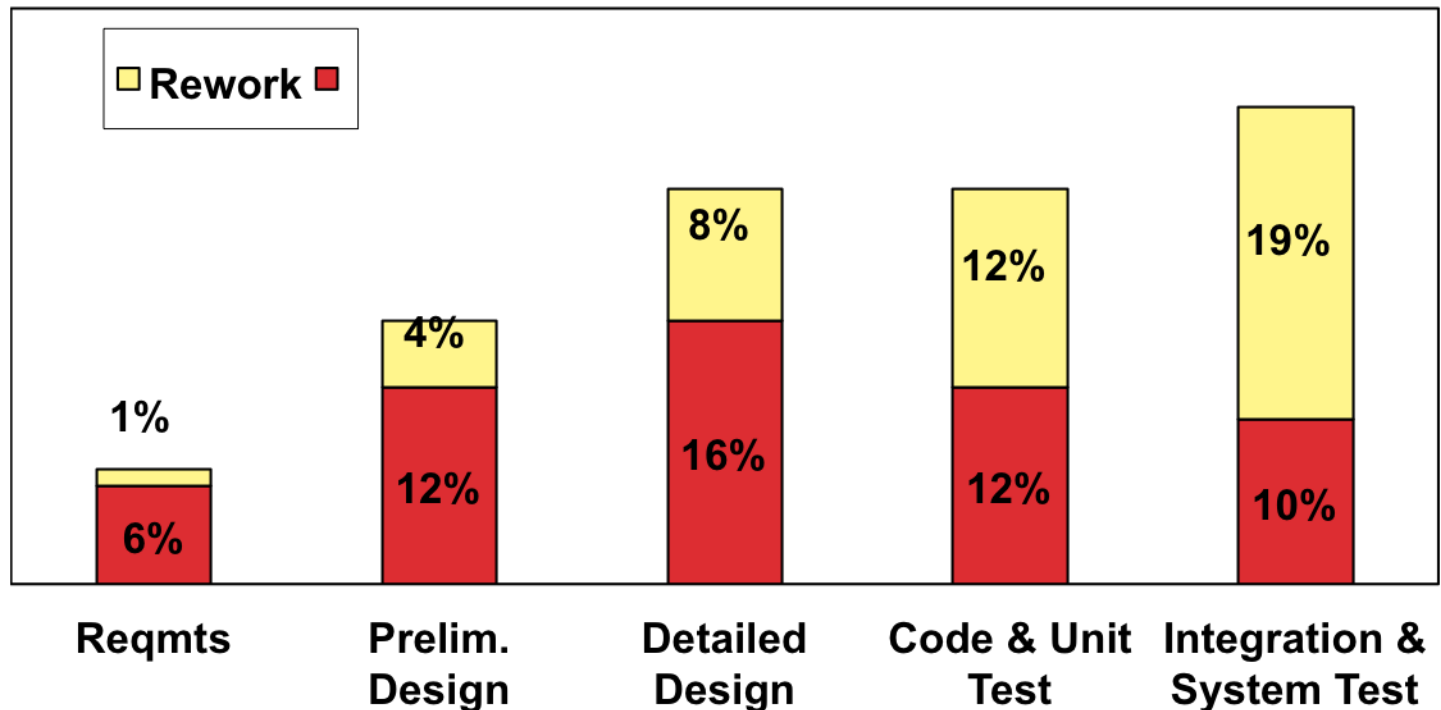
V&V techniques – types

- **Static**
  - ◆ inspections
  - ◆ source code analysis
- **Dynamic**
  - ◆ testing

V&V techniques – per activity

- Requirements
  - Inspection of requirement document (static)
  - GUI prototype (static)
  - Product prototype (dynamic)
- Design
  - Inspection of design document (static)
- Implementation
  - Test (unit, integration, application) (dyn)
  - Inspection of source code (static)

SOftEng
http://softeng.polito.it

Inspections

Inspections

- Static
  - ◆ inspections ⬅
  - ◆ source code analysis
- Dynamic
  - ◆ testing

Development

Requirements
definition

Req. inspection

Requirement
document

Design

Des. inspection

Design
document

Implemen
tation

Code

Code inspection + test

Project management
Configuration management

t

SoftEng
http://softeng.polito.it

Inspections

Requirements engineering	Requirement document	VV requirements	Requirement document
Design .	Design document	VV design	Design document
Implement unit / Implement unit	Unit / Unit	VV unit / VV unit	Unit / Unit
Integrate units	System	VV system	System

Project management
Configuration management
Quality assurance

Inspection

- Consists in
  - reading documents/code
  - By a group of people (3+, group dynamics)
  - With goal of finding defects (no correction)
- Variants of inspections
  - Reading techniques, walkthroughs, reviews
- Can find many defects
  - Test concentrates one defect at a time

Inspection

- **Advantages**
  - ◆ Can be applied to documents
    - – Requirements, design, test cases, ..
  - ◆ Can be applied to code
    - – Does not require execution environment
  - ◆ Is very effective
    - – Reuses experience and know of people on domain and technologies
    - – Has more global view (vs test: one defect at a time)
    - – Uses group dynamics
- **Limits**
  - ◆ More suitable for functional aspects
  - ◆ Requires time (effort and calendar time)

SOftEng
http://softeng.polito.it

Benefits

- Early defect detection improves product quality and reduces avoidable rework (down to 10–20%)
    - Data from industry averages [Capers Jones, 1991]
        - more data available at www.cebase.org

Requirements **Insp.** 5 (20)

Design **Insp.** 8 (40)

Defects per KLOC passed with inspection

Code **Insp.** 15 (100)

Defects per KLOC passed without inspection

Unit Test **Insp.** 7 (50)

Integration Test **Insp.** 3 (20)

System Test **Insp.** 1 (10)

Rework Costs
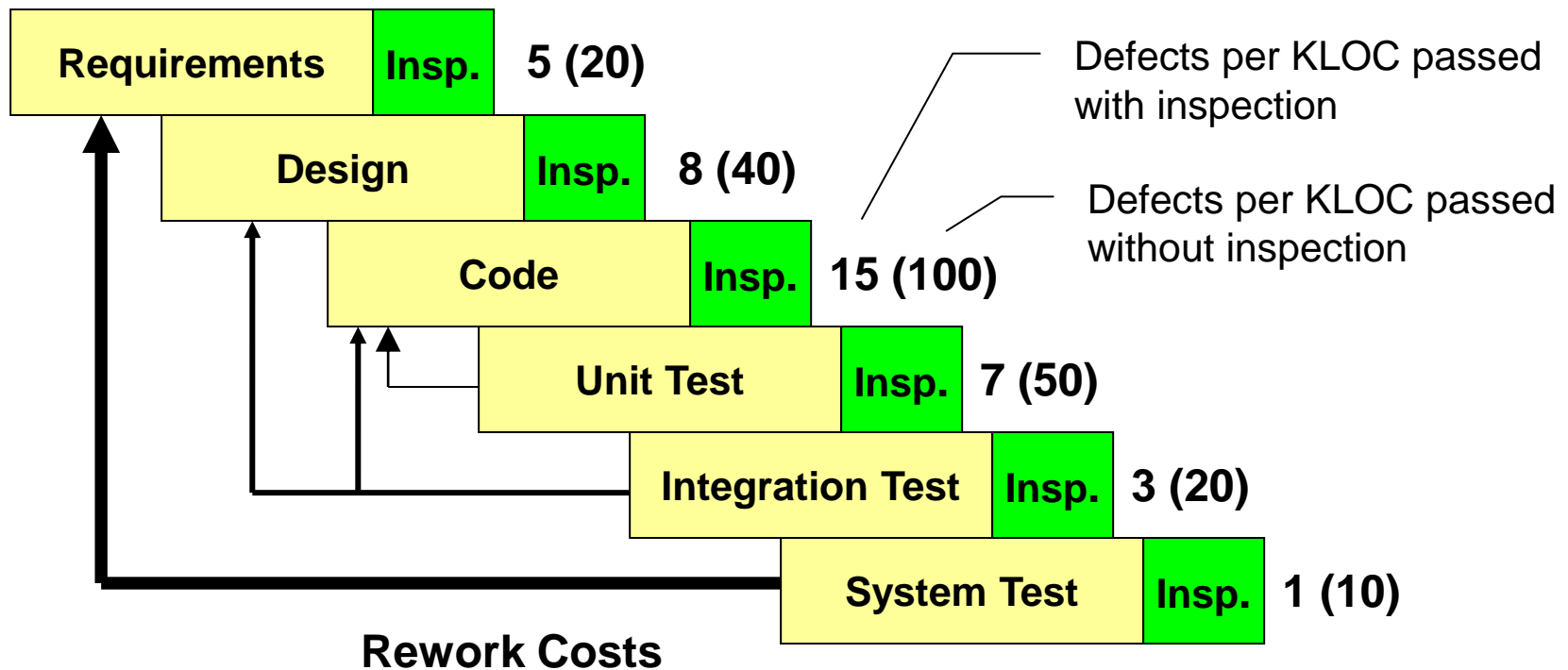
Inspection vs. testing

- Complementary techniques
  - ◆ Both should be used in V and V

Roles in group

- Moderator:
  - Leads inspection process and notably inspection meeting
    - Selects participants, prepares material
    - Usually not from project that produces document to be inspected
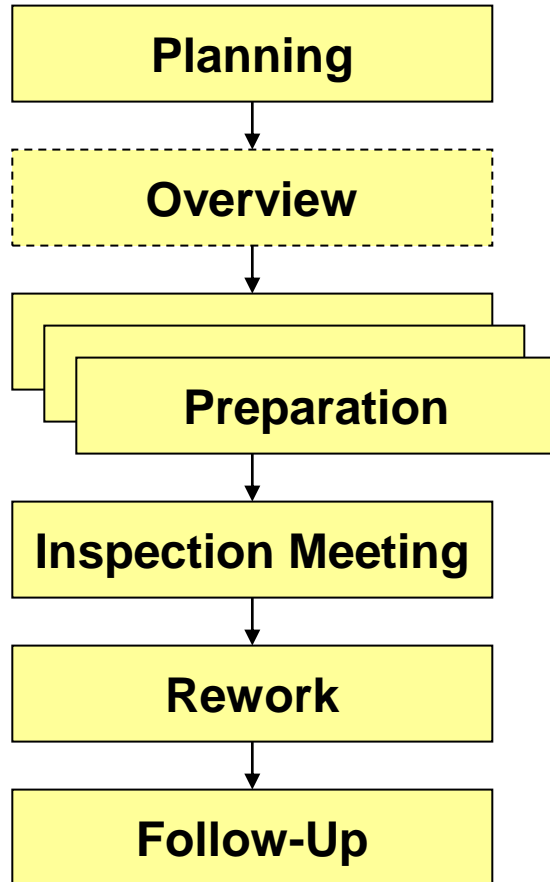- Readers:
  - Read document to be inspected
- Author
  - Answers to questions that arise
- Scribe
  - Writes inspection log

Fagan Inspection Process

Planning

Overview

Preparation

Inspection Meeting

Rework

Follow-Up

- Select team, arrange materials, schedule dates

- Present process, product

- *Become familiar with the product*

- **Team analysis to find defects**

- Correct defects

- Verify fixes, collect data

Inspectors are often unprepared. How to make visible the quality of preparation?

Process

- ◆ Overview
  - – Quickly present inspection to group goals and document to be inspected
- ◆ Preparation
  - – Read individually (applying inspection technique)
- ◆ Meeting
  - – Group reads, discusses issues, agrees on problems. Scribe logs problems. Moderator keeps focus, keeps pace, stops (long) discussions
- ◆ Rework
  - – Author fixes defects/problems
- ◆ Follow up
  - – Repeat inspection or close and pass to next phase

Prerequisites for successful inspections

- **Commitment from management**
  - Effort invested upfront, that "does not produce anything"
- **Find defects, not fix them**
- **Document under inspection meets quality standards**
- **Results not used to evaluate people (and notably author)**
- **Constructive approach**
  - Group aims to produce best possible document
    - No "kill the author" game
    - No "relax and chat" meetings

Techniques vs. document

- Ad hoc (code, requirements, design)
  - Just read it
- Defect taxonomy (code, requirements, design)
  - Categories of common defects
- Checklist (code, requirements, design)
  - Questions/controls to be applied
- Code
  - Author or reader 'executes' code
  - Reader reconstructs goal of code from code
  - Reader defines and applies some test cases
- Requirements
  - Scenario based reading
    - Defect based
    - Perspective based
- Design
  - Traceability matrix
  - Scenario execution

Defect Taxonomies for Requirements

One level
[Basili et al., 1996]

- Omission
- Incorrect Fact
- Inconsistency
- Ambiguity
- Extraneous Information

Two levels
[Porter et al., 1995]

- Omission
  - Missing Functionality
  - Missing Performance
  - Missing Environment
  - Missing Interface
- Commission
  - Ambiguous Information
  - Inconsistent Information
  - Incorrect or Extra Functionality
  - Wrong Section

SOftEng
http://softeng.polito.it

Checklists for Requirements

- Based on past defect information
- Questions refine a defect taxonomy

[Ackerman et al., 1989]

- Completeness
  1. Are all sources of input identified?

  ...

  12. For each type of run, is an output value specified for each input value?

  ...

- Ambiguity
  18. Are all special terms clearly defined?

  ...

- Consistency
  ...

SOftEng
http://softeng.polito.it

Checklists for code

- Depends on programming language
- Depends on previous results of inspections

Fault class	Inspection check
Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the lower bound of arrays be 0, 1, or something else? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a delimiter explicitly assigned?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	Do all function and procedure calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

Inspection checks

SOftEng
http://softeng.polito.it

Rates (code inspections)

- 500 LOC/hour (overview)
- 125 LOC/hour (preparation)
- 90–125 LOC/hour (meeting)
  - Ex. 500 LOCs, 4 people, 40 person hours
    - Overview 1hr X 4= 4person hours
    - Preparation 4hr X 4 = 16 person hours
    - Meeting 5hr X 4 = 20 person hours

Scenario based reading

- ◆ Ask inspectors to create an appropriate abstraction
  - – Help to understand the product
- ◆ Ask inspectors to answer a series of questions tailored to the abstraction Inspectors follow different scenarios each focusing on specific issues

Defect-Based Reading

[Porter et al., 1995]

- A scenario-based reading technique to detect defects in requirements expressed in a formal notation (SCR)

- Each scenario focuses on a specific class of defects
  - data type inconsistencies
  - incorrect functionality
  - ambiguity/missing functionality

Excerpt from incorrect functionality scenario

1. *For each functional requirement identify all input/output data objects: questions …*

2. *For each functional requirement identify all specified system events:*

   (a) Is the specification of these events consistent with their intended interpretation?

3. *Develop an invariant for each system mode:*

   questions …

Perspective-Based Reading

[Basili et al., 1996]

- A scenario-based reading technique to detect defects in requirements expressed in natural language
  - extended later for design and source code
- Each scenario focuses on reviewing the document from the point of view of a specific stakeholder
  - User (abstraction required: user tasks descriptions)
  - Designer (abstraction required: design)
  - Tester (abstraction required: test suite)

    For each requirement/functional specification, generate a test or set of tests that allow you to ensure that an implementation of the system satisfies the requirement/functional specification. Use your standard test approach and technique, and incorporate test criteria in the test suite. In doing so, ask yourself the following questions for each test:

    questions ...

TESTING

Testing

- Static
  - inspections
  - source code analysis
- Dynamic
  - testing

Testing

- Dynamic technique, requires execution of executable system or executable unit
  - system test
  - unit test

Testing

- The process of <u>operating</u> a system or component under specified conditions observing or recording the results to detect the <u>differences</u> between <u>actual</u> and <u>required</u> behavior (= failures)
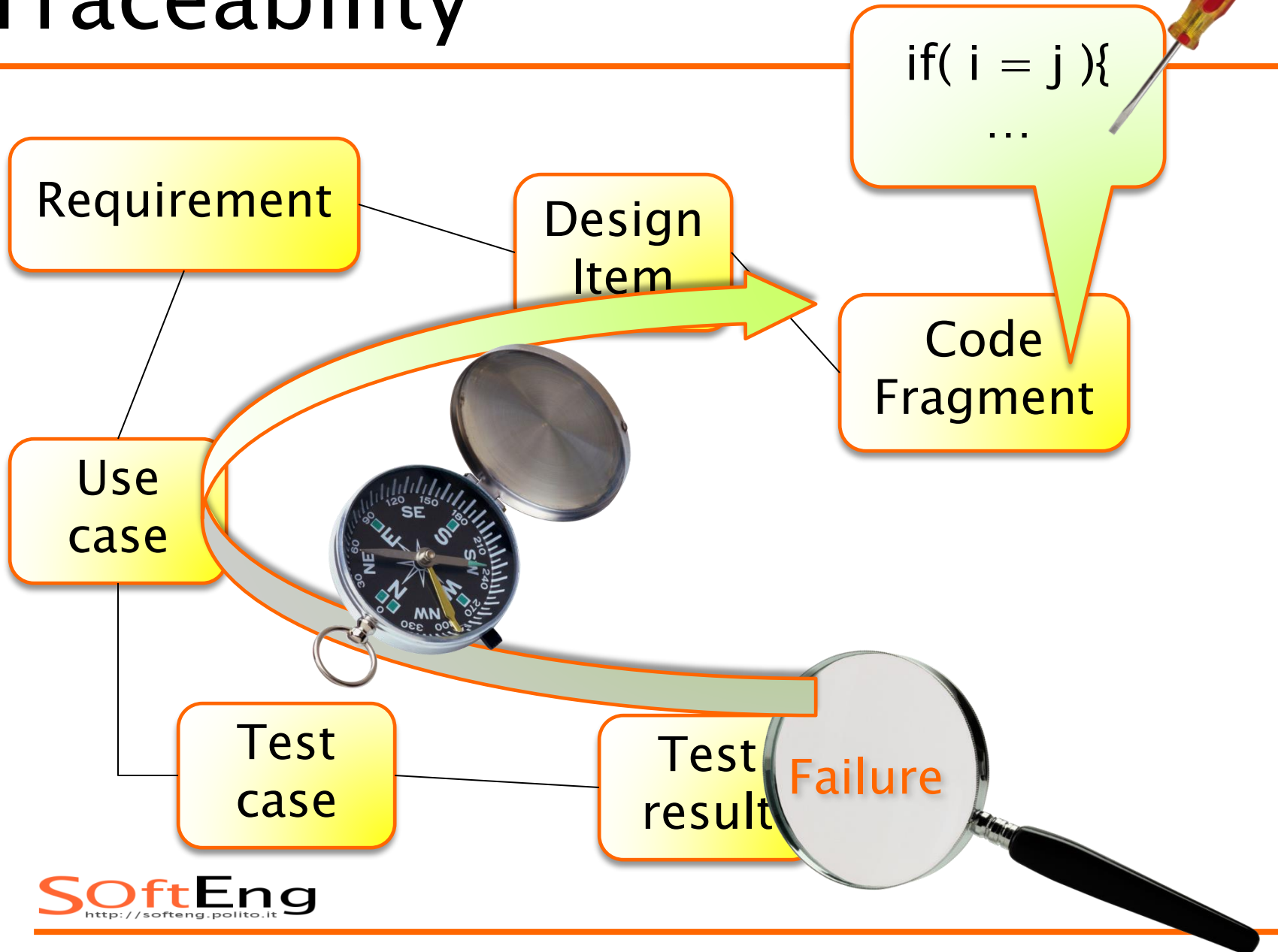
Purpose of test

- The purpose of testing process is to find defects in the software products
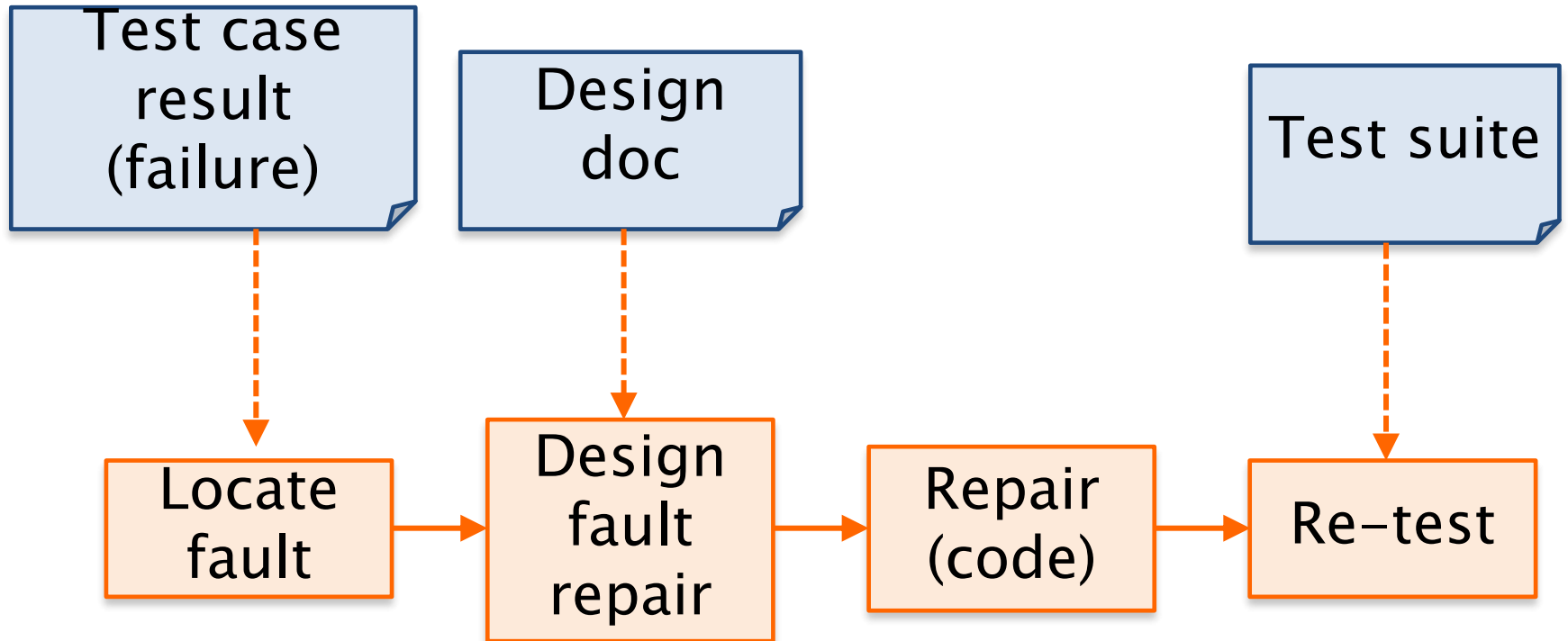 - A test is successful if it reveals a defect

Testing vs. debugging

- Defect testing and debugging are different activities

  - May be performed by different roles in different times

- Testing tries to find failures

- Debugging searches for and removes the corresponding fault(s)

Traceability

Requirement

Use case

Test case

Design Item

Code Fragment

```
if( i = j ){
…
```

Test result

Failure

Debugging

Test case

- Certain stimulus applied to executable (system or unit), composed of
  - name
  - input (or sequence of)
  - expected output

- With defined constraints/context
  - ex. version and type of OS, DBMS, GUI ..

Test suite

- Set of (related) test cases

Test case log

- Test case ref. +
  - Time and date of application
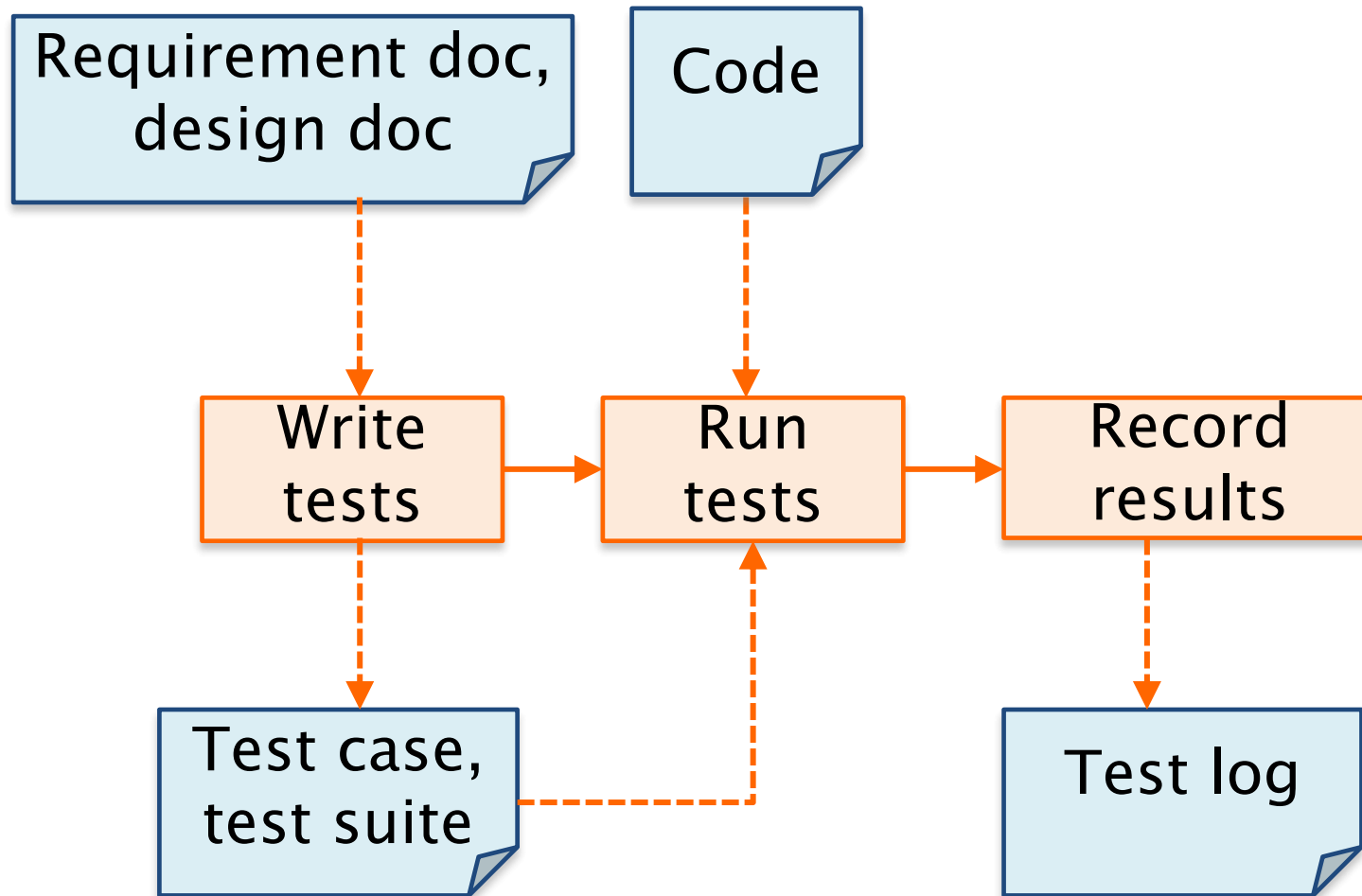 - Actual output
  - Result (pass / no pass)

Ex.

- Function **`add(int x, int y)`**
- Test case:
  - T1(1,1; 2)
  - T2(3,5; 8)
- Test suite
  - TS1{T1, T2}
- Test log
  - T1, 16-3-2013 9:31, result 2, success
  - T2, 16-3-2013 9:32, result 9, fail
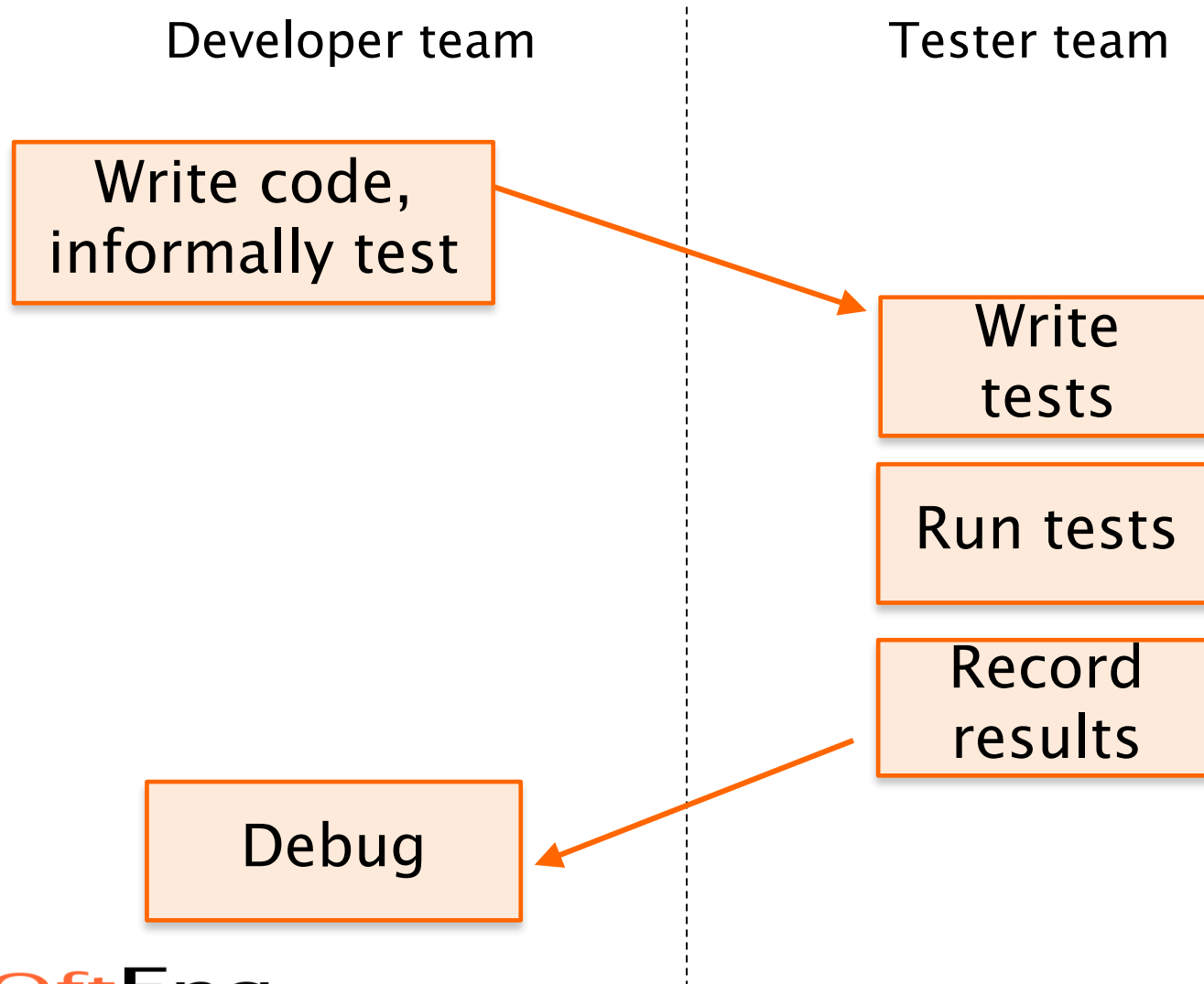
Test activities

- **Write test cases**
  - Test case, test suite
- **Run test case (test suite)**
- **Record results**
  - Test case log

Test activities

Possible scenario

Developer team | Tester team

Write code, informally test → Write tests

Run tests

Record results

Debug ←

Scenario 2

Developer team

Write code, informally test

Write tests

Run tests

Record results

Debug

Scenario 3

Developer team

Tester team

Tester team
(3rd party)

Write code, informally test

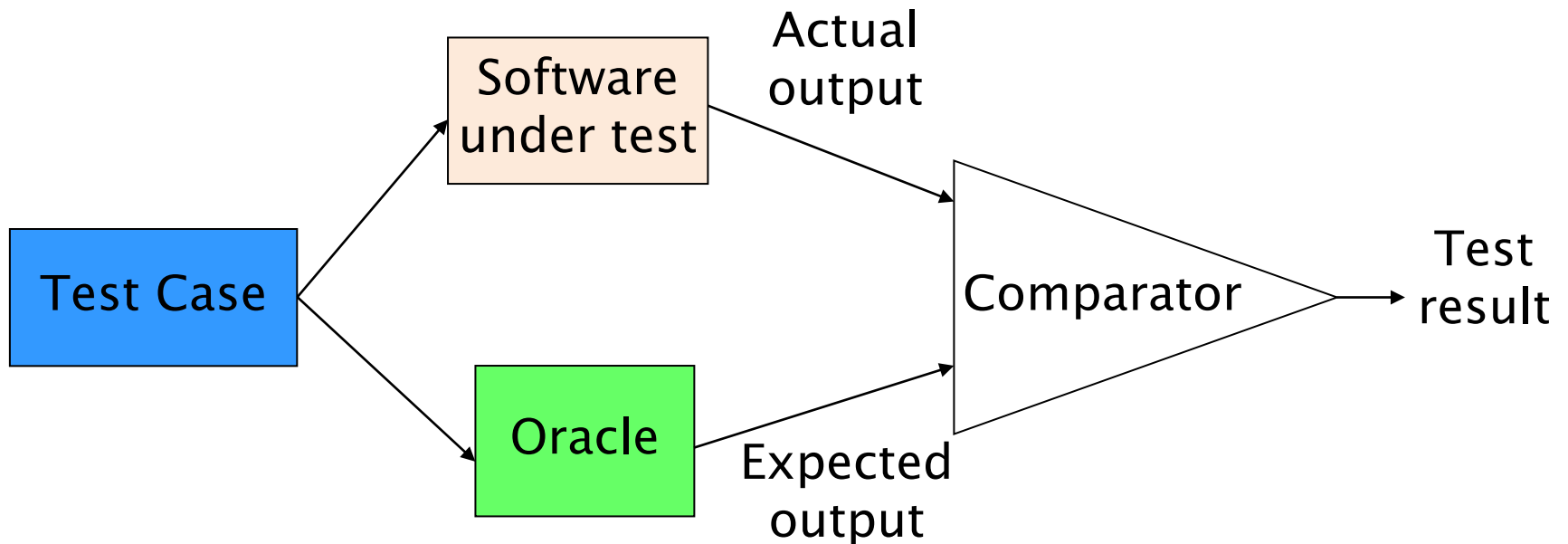Write tests

Run tests

Record results

Debug

Write tests

Run tests

Record results
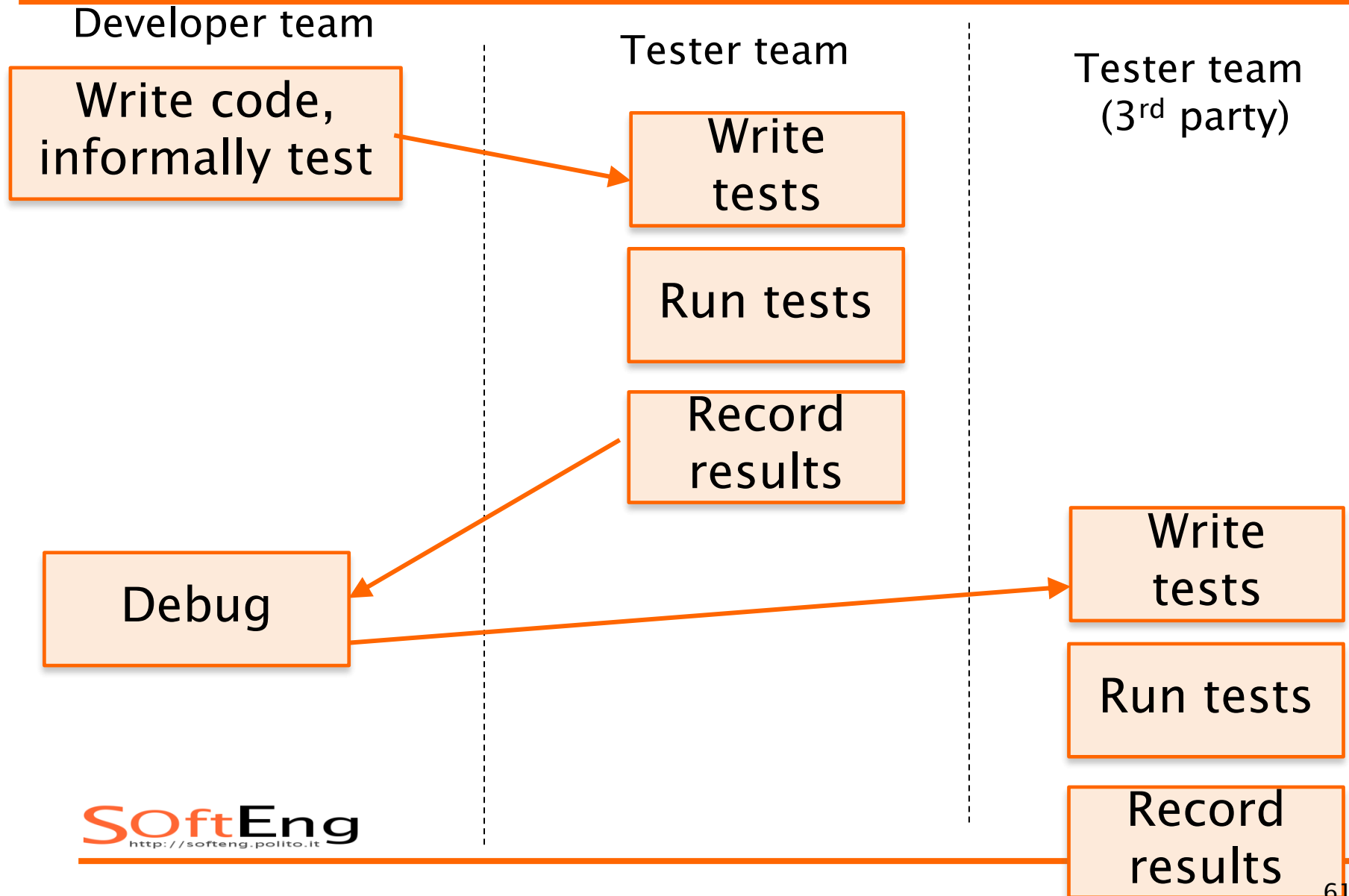
Oracle

Oracle

- The ideal condition would be to have an automatic oracle and an automatic comparator

  - The former is very difficult to have
  - The latter is not always available (ex GUI)

- A human oracle is subject to errors

- The oracle is based on the program specifications (which can be wrong)

Oracle

- Necessary condition to perform testing:
  - Know the expected behavior of a program for a given test case (oracle)
- Human oracle
  - Based on req. specification or judgment
- Automatic oracle
  - Generated from (formal) req. specification
  - Same software developed by other parties
  - Previous version of the program (**regression**)

Theory and constraints

Correctness

- Correct output for all possible inputs

- Requires exhaustive testing

Exhaustive testing

- Write and run all possible test cases
 (== execute all possible inputs)

Exhaustive test – example

- function: Y = A + B

- A and B integers, 32 bit

- Total number of test cases : $2^{32} * 2^{32} = 2^{64} \approx 10^{20}$

- 1 ms to run a test case $\Rightarrow$ 3 billion years
  (and the time for writing test cases is not considered)

Pentium case – 1994

- Error in division function in the FPU (missing entries in a lookup table)

- 1 case in 9 billion

- Cost 474 million USD to replace the processors

Exhaustive test
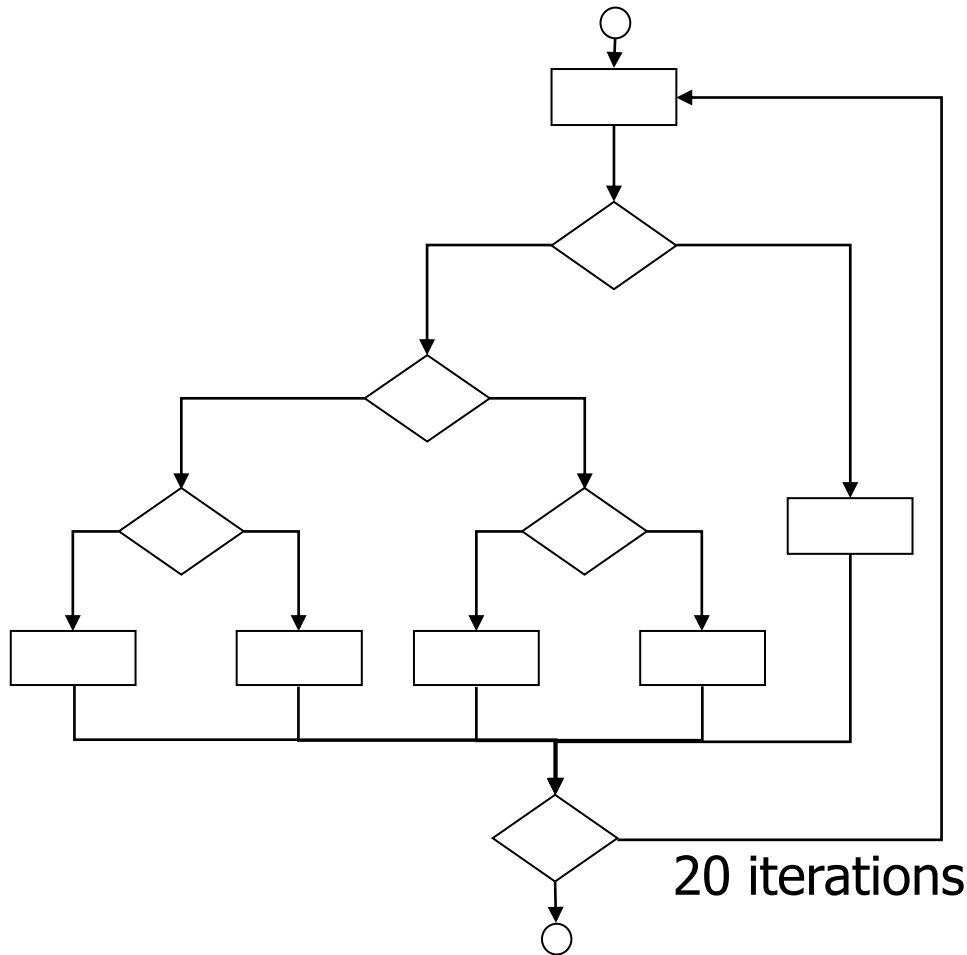
- 5 possible paths per iteration

- 20 iterations

- Overall: $5^{20} \approx 10^{14}$ possible paths

- 1 ms/test $\Rightarrow$ 3170 years

20 iterations

Exhaustive test

- Exhaustive test is not possible
- So, goal of test is finding defects, not demonstrating that system is defect free
- Goal of test (and VV in general) is assuring a *good enough* level of confidence

Dijkstra thesis

- *Testing can only reveal the presence of errors, never their absence*

E. W. Dijkstra. Notes on Structured Programming.
In *Structured Programming,* O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. Academic, New York, 1972, pp. 1-81.

How to select test cases?

- Exhaustive testing impossible (except in trivial cases)

- Key point is in how to select test cases

- Criterion to select test cases is evaluated by
  - Reliability
  - Validity

SOftEng
http://softeng.polito.it

Basic concepts

- D: program domain (input)
- $d \in D$, P(d) is the program output
- OK(d) $\Leftrightarrow$ P(d) corresponds to oracle
- Test: $T \subseteq D$
- SUCC(T) $\Leftrightarrow$ $\forall\ t \in T$, OK(t)

J. B. Goodenough and S. L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering,* June 1975, pp. 26-37.

Criteria

- Tests can be selected by means of criteria

- C : selection criterion for tests

- COMPLETE(T,C): T is selected by C

Validity

- $\exists d \in D, \neg OK(d) \Rightarrow (\exists T \subseteq D)$ COMPLETE(C,T) $\wedge$ $\neg$SUCC(P,T))

- *Valid Criterion:*
  - ◆ *C* is *valid* if and only if whenever *P* is incorrect *C* selects at least one test set *T* which is not successful for *P*.

Reliability

- $\forall T1, \forall T2 \subseteq D,$
  $COMPLETE(C,T1) \wedge COMPLETE(C,T2))$
  $\Rightarrow SUCC(T1) \Leftrightarrow SUCC(T2)$

- *Reliable Criterion:*

  - *C* is *reliable* if and only if either every test selected by *C* is successful or no test selected is successful.
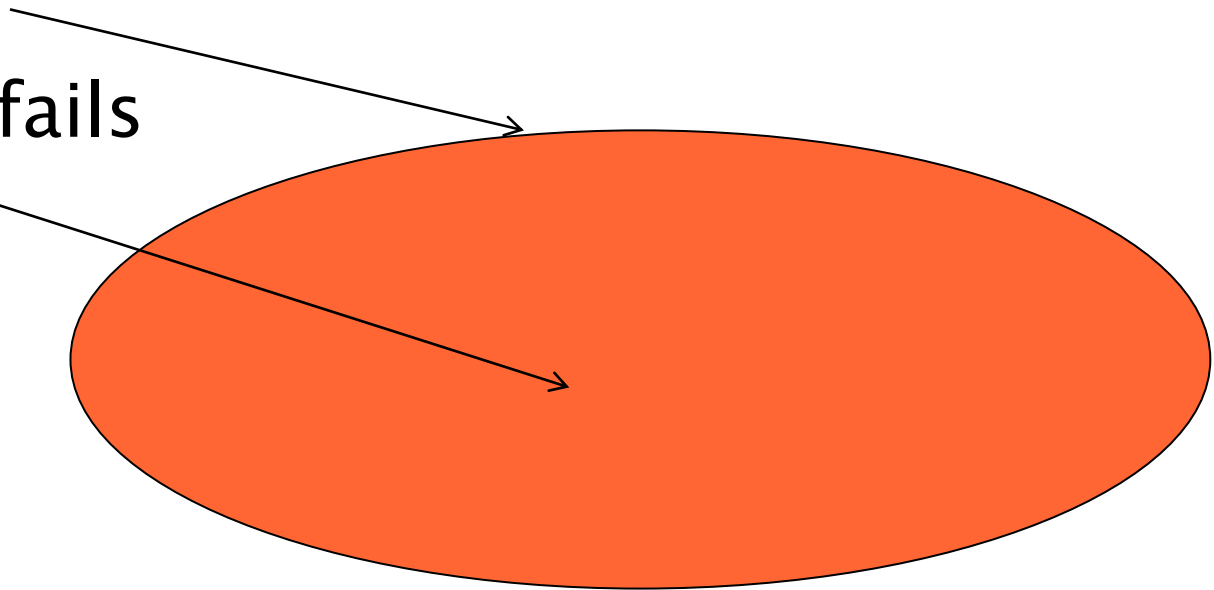
Ex: Pentium case

- Reliable: all test cases succeed in I−F, all test cases fail in F

- Valid: selects both F and I

I: input space

F $\subseteq$ I: input that fails

I−F : input that does not fail

Fundamental theory

- Theorem

$(\exists T \subseteq D)(COMPLETE(T,C) \wedge RELIABLE(C) \wedge$
$VALID(C) \wedge SUCC(T)) \Rightarrow$
$(\forall d \in D)OK(d)$

- The success of a test T selected by a reliable and valid criterion implies the correctness of T

Uniformity

- Criterion uniformity focuses on program specification
  - Not only program
- A criterion C is uniformly valid and uniformly reliable if and only if C selects only the single test set $T = D$

E. J. Weyuker and T. J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering,* May 1980, pp. 236–246.

Howden theorem

- For an arbitrary program P it is impossible to find an algorithm that is able to generate a finite ideal test (that is selected by a valid and reliable criterion)

W.Howden. Reliability of the Path Analysis Testing Strategy. IEEE Transactions of Software Engineering, 2(3), September 1976, pp. 208-215

SOftEng
http://softeng.polito.it

Brainerd Landweber

- Given two programs the problem of deciding whether they compute the same function is indecidible

- Therefore even if we have access to the archetype program we cannot demonstrate the equivalence of a new program

Weinberg's law

- A developer is unsuitable to test his/her own code

  - A developer is not emotionally willing to find defects

  - If a developer misunderstands a problem, he cannot find such error

- Testing should be performed by

  - A separate QA team

  - Peers

Pareto-Zipf law

- Approximately 80% of defects come from 20% of modules

- It is better to concentrate on the faulty modules
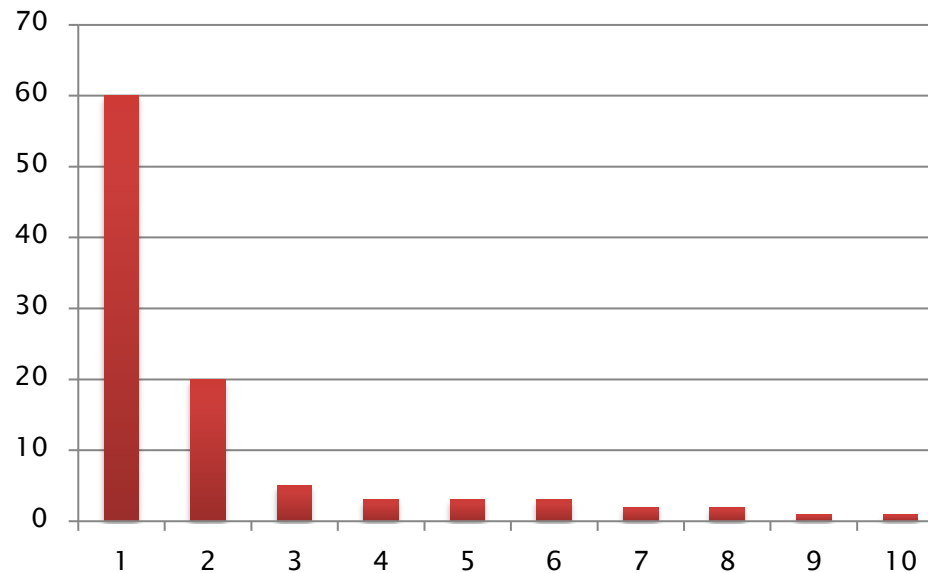
Summary

- ## From correctness point of view

  - ### Use criteria to define test cases that are
    - Reliable
    - Valid

- ## From psychological point of view

  - ### Use a policeman mindset
    - Any part of a software (including tests) is guilty in principle (may contain defects)

- ## From risk management point of view

  - ### Test in function of the risk
    - Safety critical software
    - Mission critical software

Test classification

- Per item under test
  - Unit, integration, system
  - Regression
- Per approach
  - Black box (functional)
  - White box (structural)
  - Reliability assessment/prediction
  - Risk based (safety security)
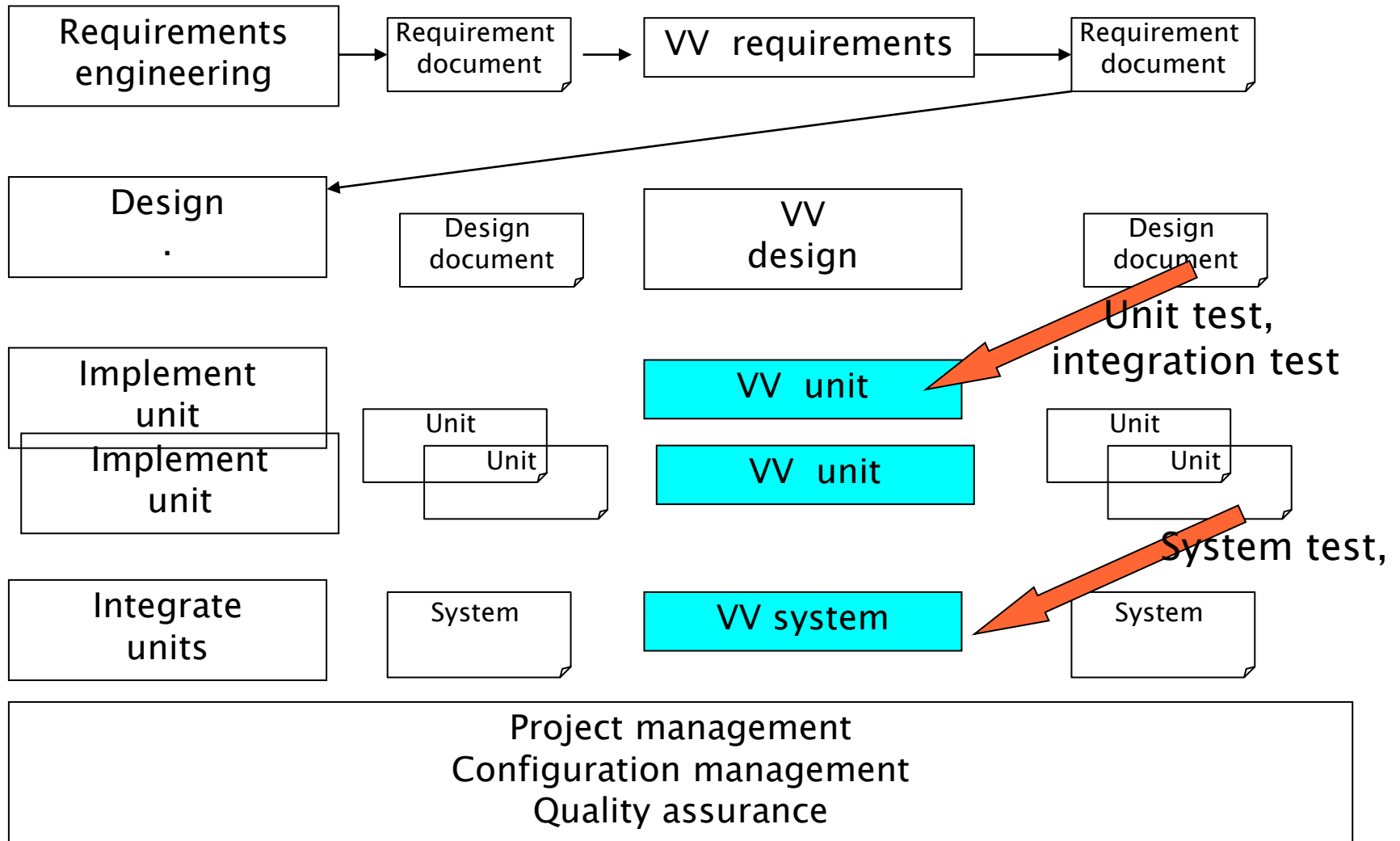- Per formality
  - Exploratory / informal
  - formal

Test per item under test

- **Unit tests**
  - Individual modules
- **Integration tests**
  - Some modules working together (partial system)
- **System tests**
  - All modules together (complete system)
    - API level
    - GUI level

Test per formality

- **Informal testing / exploratory testing**
  - ◆ Tests are not formalized (just applied, test case and test result are lost)
  - ◆ Technique used to generate the test is not explicit
- **(Formal) Testing**
  - ◆ Tests are formalized
  - ◆ A technique is used (and documented) to generate test cases

Test per item

Requirements engineering	Requirement document	VV requirements	Requirement document

Design
.

Design document	VV design	Design document

Unit test, integration test

Implement unit
Implement unit

Unit		VV unit		Unit	
	Unit	VV unit			Unit

System test,

Integrate units	System	VV system	System

Project management
Configuration management
Quality assurance

Test per approach

- **Given an item to test, approach can be**
  - ◆ **Requirements driven**
    - – Are the requirements of the object satisfied?
  - ◆ **Structure**
    - – Is the object built as it should?
  - ◆ **Reliability / statistic**
    - – Does it satisfy the customer need? (use most common operational scenarios)
  - ◆ **Risk**
    - – Is it vulnerable to most likely risks?

Testing classification (2)

	Item		
	Unit	Integration	System
Functional / black box	X	X	X
Structural / white box	X		
Reliability			X
Risks			X

Test classification and coverage

Approach	Item tested		
	Unit	Integration	System
Requirements-driven	100% unit requirements	100% product requirements	100% system requirements
Structure-driven	85% logic paths	100% modules	100% components
Statistics-driven			90-100% of usage profiles
Risk-driven	As required	As required	100% if required

Coverage

- Entities considered by at least one test case / total entities

- 'Entity' depends on type of test
  - Test cases
  - Requirement
  - Function
  - Structural element
    - Statement, decision, module

SOftEng
http://softeng.polito.it

Coverage

Item tested	Coverage	
Unit (class)	methods	Consider methods of the class >= one test case per method
	Black box, partitions	Consider a method Define partitions >= One test case per partition
	Black box, boundary	Define partitions >= One test case per boundary
	Black box, random	Generate randomly test cases

Coverage

Object tested	Coverage	
Integration (some classes)	dependencies	At least one test case per dependency

Coverage

Object tested	Coverage	
System (all classes)	Functional requirements	At least one test case per requirement (may correspond to testing methods of 'main' class at end of integration testing)
	Scenarios	At least one test case per scenario
	Non functional requirements	At least one test case per non functional requirement (efficiency)

Regression testing

UNIT TEST

Unit Test

Requirements engineering	→	Requirement document	→	VV requirements	→	Requirement document

Design .		Design document		VV design		Design document

Implement unit		Unit		VV unit		Unit
Implement unit		Unit		VV unit		Unit

Integrate units		System		VV system		System

Project management
Configuration management
Quality assurance
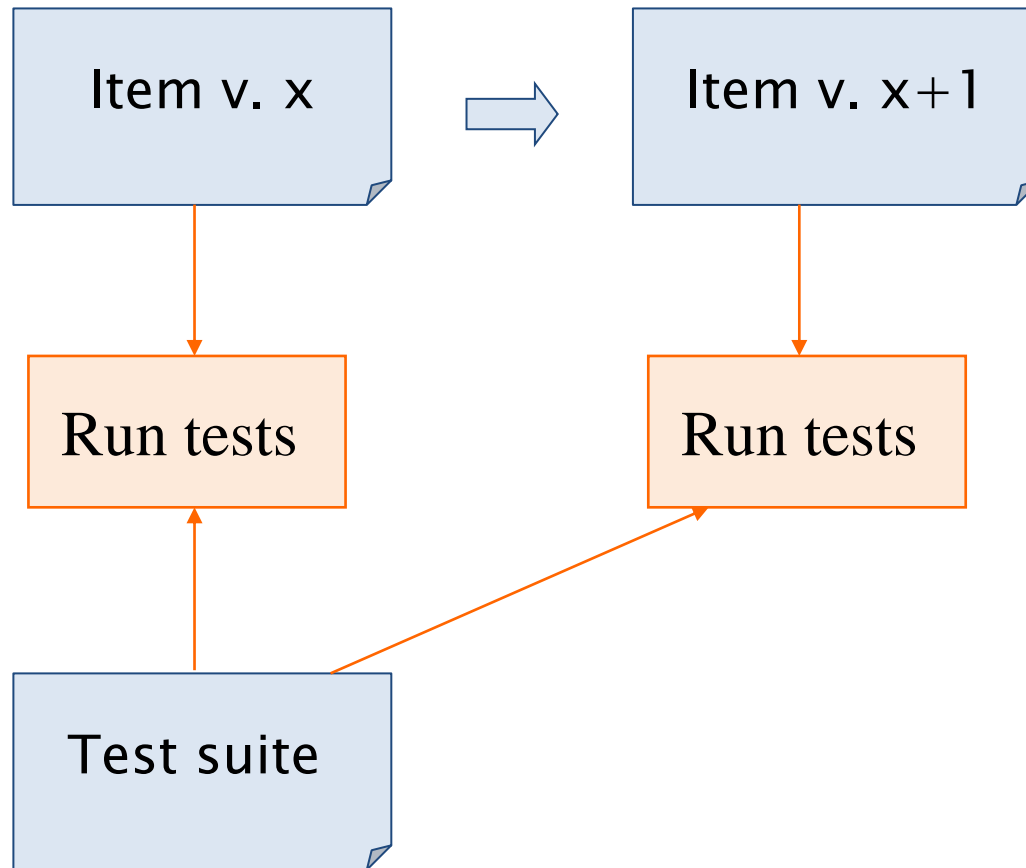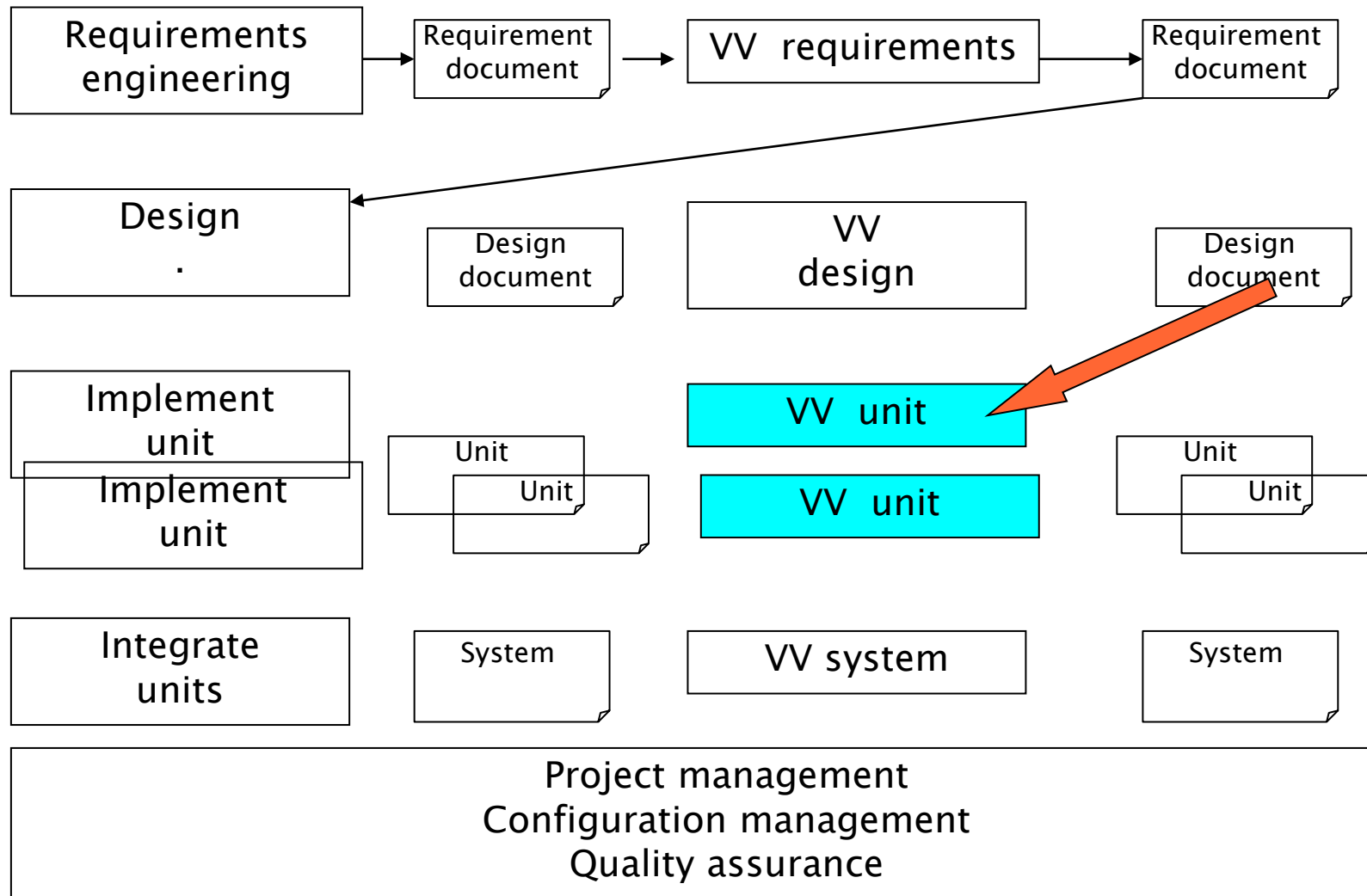
Unit test

- ## Test of one <u>independent</u> unit
  - ◆ Unit:
    - – function (procedural languages)
    - – class and its methods (oo languages)

Unit test

- **Black box (functional)**
  - ◆ Random
  - ◆ Equivalence classes partitioning
  - ◆ Boundary conditions
- **White Box (structural)**
  - ◆ Coverage of structural elements

Unit test – black box

Random

- **Function `double squareRoot(double x)`**
  - Extract randomly x
    - T1 (3.0 ; $\sqrt{3}$)
    - T2 (1000.8 ; $\sqrt{1000.8}$)
    - T3 (–1223.7; error)

- **Function `double invert(double x)`**
  - Extract randomly x
    - T1 (1.0 ; 1.0)
    - T2 (–2.0 ; –0.5)

Random

- Pros
  - Indipendent of requirements
- Cons
  - Requires many test cases (easy to define the inputs, requires Oracle to compute the expected output)

Equivalence classes partitioning

- Divide input space in partitions
  - that have similar behavior from point of view of (requirements for) unit
  - Take one / two test cases per partition
- Boundary conditions
  - Boundary between partitions
  - Take test cases on the boundary

Equivalence classes
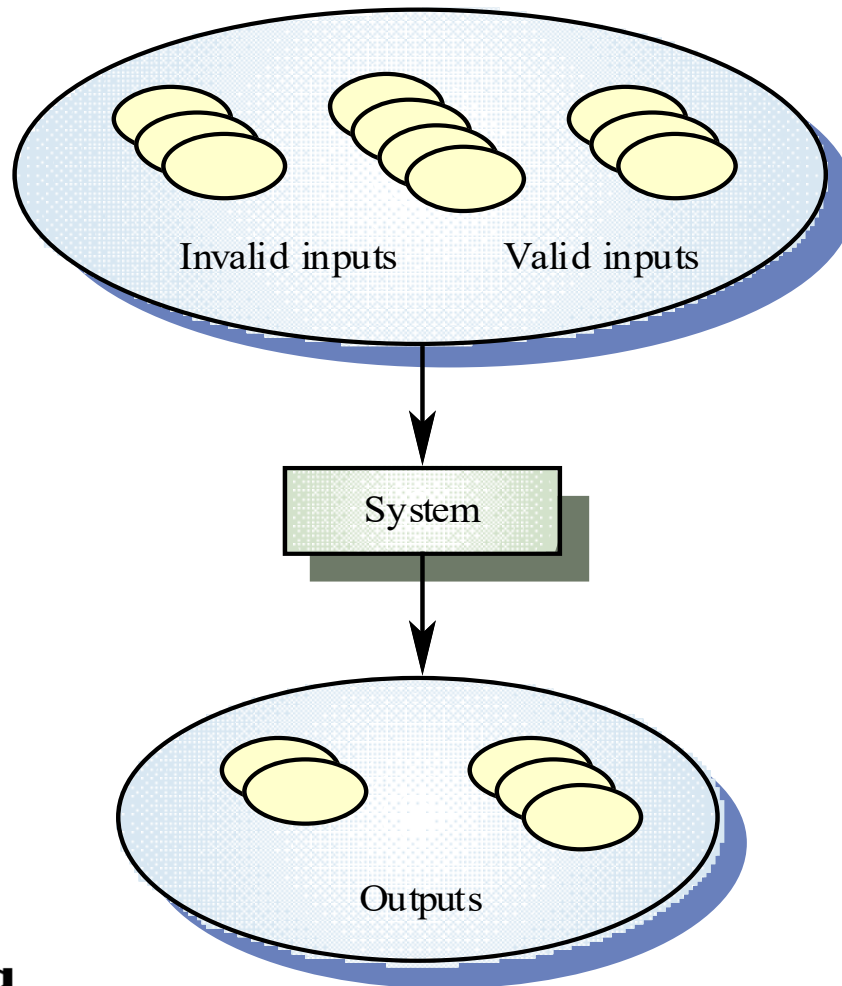
Equivalence classes

- A class corresponds to set of valid or invalid inputs for a *predicate* on the input variables
  - ◆ If a test in a class has not success the other tests in the same class may have the same behavior

  - ◆ Valid: acceptable
  - ◆ Invalid: not acceptable (produces exception, error)

SOftEng
http://softeng.polito.it

Criteria, predicates, partitions

- Criterion: attribute

- Predicate: condition on attribute, defines a partition ($==$ eq class)

- Ex

  - Criterion: age
  - Predicate: age $>$ 100

    age $<$ 25

Predicates

- Common predicates:
  - ◆ Single value: age = 33
  - ◆ Interval: age between 0 and 200
  - ◆ Boolean: married = true or false
  - ◆ Discrete set: marital status = single, married, divorced

Predicates and eq classes

Predicates	Classes	Example
Single value	Valid value, invalid values < value Invalid values > value	Age = 33 Age < 33 Age > 33
Interval	Inside interval, Outside one side Outside, other side	Age > 0 and age <200 Age > 200 Age < 0
Boolean	True false	Married = true Married = false
Discrete set	Each value in set One value outside set	Status = single Status = married Status = divorced Status = jksfhj

Selection of test cases

- Every equivalence class must be covered by one test case at least
  - A test case for each invalid input class
  - Each test case for valid input classes must cover as many (remaining) valid classes as possible

Equivalence classes

- Function double `squareRoot(double x);`
  - ◆ Criterion: sign of x
  - ◆ Partitions
    - – Sign is positive   T1 (1 ; $\sqrt{1}$)
    - – Sign is negative   T2 (–1 ; error)
  - ◆ Boundary: zero, infinite
    - – Zero and close
      T3 (0; $\sqrt{0}$) T4(0.01; $\sqrt{0.01}$) T5(–0.01; error)
    - – 'Infinite' and close
      T6 (maxdouble; $\sqrt{\text{maxdouble}}$) T7 (maxdouble+0.01; err)
      T8 (mindouble; $\sqrt{\text{mindouble}}$) T7 (mindouble–0.01; err)

- Previous examples considered only one criterion

- What if many?

- Need to consider all combinations of predicates

Equivalence classes

`int convert(String s)`
converts a sequence of chars (max 6) into an integer number. Negative numbers start with a '-'

Criterion to define the class	Equivalence classes and test cases	
String represents a well formed integer	Yes T1("123"; 123)	No T2("1d3" ; error)
Sign of number	Positive T1("123" ; 123)	Negative T3("-123" ; -123)
Number of characters	<=6 T1("123" ; 123)	>6 T4("1234567"; err)

Equiv. classes– combinatorial

WF integer	sign	N char	
yes	Pos	<=6	T1("123"; 123)
		>6	T4("1234567"; err)
	Neg	<=6	T3("–123"; –123)
		>6	T5("–123456"; err)
no	Pos	<=6	T2("1d3"; err)
		>6	T6("1sed345"; err)
	Neg	<=6	T7("–1ed"; err)
		>6	T8("–1ed234"; err)

Boundary – combinatorial

WF integer	sign	N char	
yes	Pos	<=6	"0" "999999"
		>6	"1000000" "9999999"
	Neg	<=6	"-0" "-99999"
		>6	"-999999"
no	Pos	<=6	""
		>6	" " (7 blanks)
	Neg	<=6	"-"
		>6	"- "

Equiv classes and state

- When a module has state
  - the state has to be considered to define the partitions
  - State may be difficult to read/create
  - Requires a sequence of calls

Equiv classes and state

- **`double Ave3(int i)`**
  - ◆ Computes average of last three numbers passed, excluding the negative ones

  - ◆ Criteria
    - – state: n elements received
    - – int i: positive, negative

Equiv classes and state

N elements	i	Test
0	NA	
1	Pos	T1(10; 10)
	Neg	T2(–10; ?)
2	Pos	T3(10,20 ; 15)
	Neg	T4(–10,–20; ?)
3	Pos	T5(10,2,6; 6)
	Neg	T6(–10,–2,–6; ?)
>3	Pos	T7(1,2,3,4; 2.5)
	Neg	T8(–1,–2,–3,–4; ?)

Test of OO classes

- Have state
- Many functions to be tested

- Identify criteria and equivalence classes
- Apply them to each function

Test of OO classes

```
//  receives and stores events, with time tag
public class EventsQueue{
  public void reset();      // cancels all events
  public void push(int timeTag)
                         throws InvalidTag
   // discards events with negative or zero time tag
   // and with time tag already existing
  public int pop() throws EmptyQueue
   // returns and cancels event with lower time tag
   // raises exception if queue empty
}
```

Test of OO classes

- Function push()

Empty	Repeated elements	Sign > 0	Test case
yes	yes	Yes	Reset(); Push(10); Push(10); Pop() → 10; Pop(); → EmptyQueue
		No	Reset();Push(−10); Push(−10); Pop(); → EmptyQueue
	no	Yes	
		No	
no	yes	Yes	
		No	
	no	Yes	
		No	

Test of OO classes

- Function reset()

Empty	Repeated elements	Sign > 0	Test cases
yes	yes	Yes	NA
		No	NA
	no	Yes	reset(); pop() → EmptyQueue
		No	NA
no	yes	Yes	NA
		No	NA
	no	Yes	NA
		No	NA

Unit test black box – summary

- Functional test of units (functions, classes) generates test cases starting from the specification of the unit

- Key techniques are

  - Random
  - Equivalence classes partitioning
  - Boundary conditions

UNIT TEST – WHITE BOX

Unit test

- **Black box (functional)**
  - ◆ Random
  - ◆ Equivalence classes partitioning
- **White Box (structural)**
  - ◆ Coverage of structural elements
    - – Statement
    - – Decision, condition (simple, multiple)
    - – Path
    - – Loop

SOftEng
http://softeng.polito.it

Statement coverage

double abs(double x){

  if (x>=0) then return x;

         else return -x;

}

statements

Two test cases to cover all statements T1(1; 1)

T2(-1; 1)

Statement coverage

Try to execute all statements in the program

Measure:

statement coverage =

#statements covered / #statements

Problem: statement?

```
double abs(double x){
   if (x>=0) then return x;
                 else return -x;
}
double abs(double x){
   if (x>=0) then return x;
                 else return -x;
}
double abs(double x){
   if (x>=0) then return x; else return -x;
}
```

Transform program in control flow

- ## Node:
  - ◆ atomic instruction
  - ◆ decision
- ## Edge: transfer of control

  - ◆ and basic blocks
    - – Nodes can be collapsed in basic blocks
    - – A basic block has only one entry point at initial instruction and one exit point at final instruction

Control flow graph

```
double abs(double x){
if (x>=0) then return x;
        else return –x;
            }
```
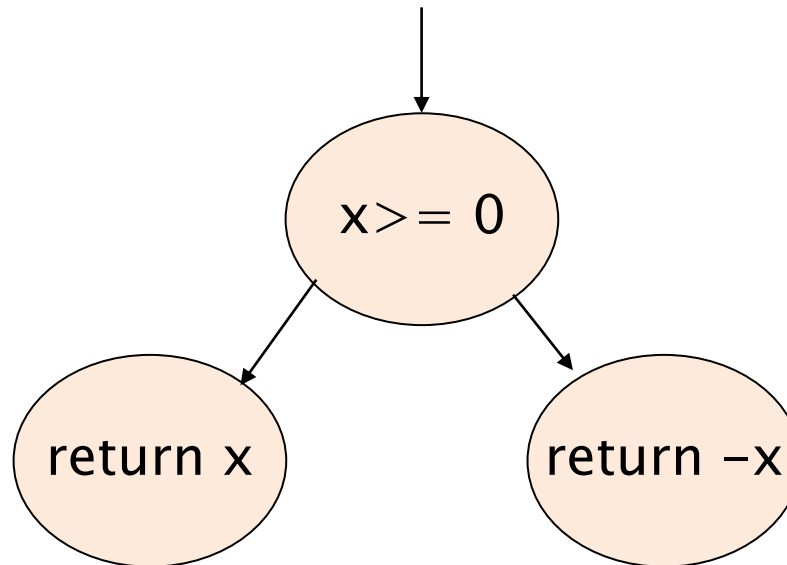
Control flow graph

```
float homeworkAverage(float[] scores) {
    float min = 99999;
    float total = 0;
    for (int i = 0; i < scores.length; i++){
        if (scores[i] < min)
            min = scores[i];
        total += scores[i];
    }
    total = total – min;
    return total / (scores.length – 1);
}
```

T1: input: scores[10.0]
    Exp Output: 10.0

min=9999

total=0

i=0

i<scores.length

true

scores[i]<min

true

total=total–min

i++

min=scores[i]

total+=scores[i]
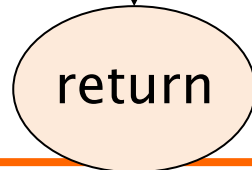
return

Basic blocks

```
float homeworkAverage(float[] scores) {
    float min = 99999;
    float total = 0;
    for (int i = 0; i < scores.length; i++){
        if (scores[i] < min)
            min = scores[i];
        total += scores[i];
    }
    total = total - min;
    return total / (scores.length - 1);
}
```

(scores[10.0] ; 10.0)

100% node coverage

min = 99999
total= 0
i=0

i<scores.length

true

scores[i]<min

true

i++

min=scores[i]

total+=scores[i]

total=total-min
return
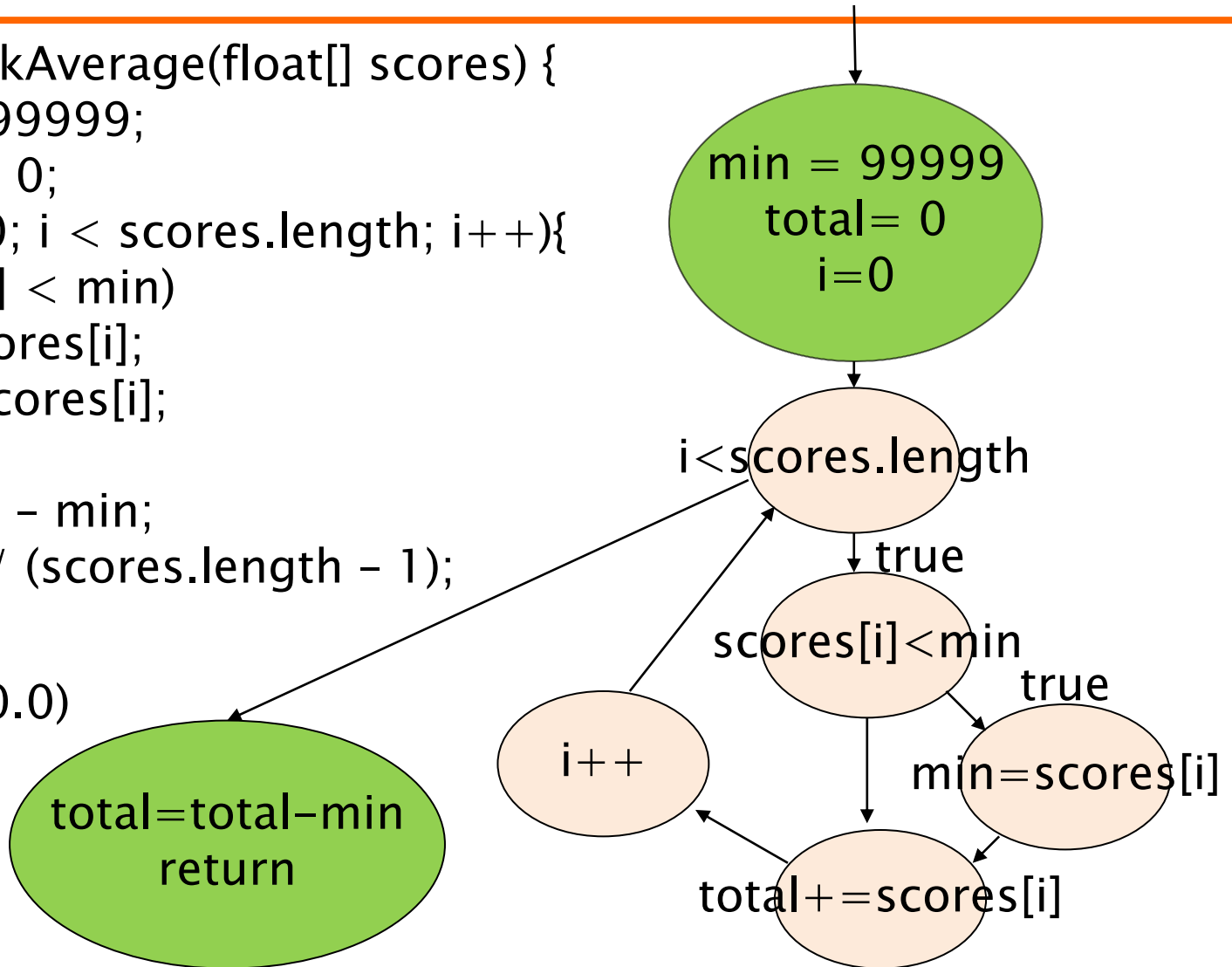
Measure: Node coverage

- Node coverage = number of nodes executed / total number of nodes

- For each test
- Cumulative: for a test suite

Node coverage

```
double abs(double x){
    if (x>=0) then return x;
             else return -x;
                 }
```
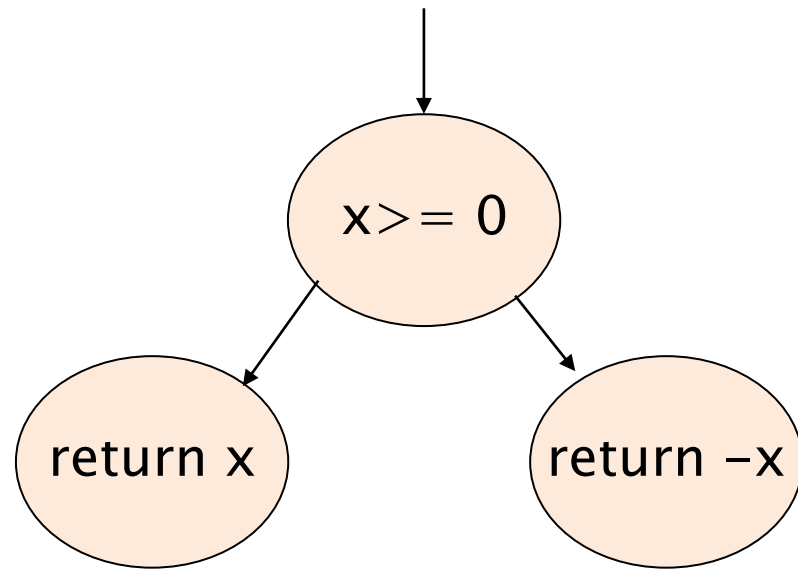

x>= 0

return x   return -x

T1 (1; 1)
T2 (-1; 1)

Statement coverage = node coverage

Statement coverage

- Node coverage $\Leftrightarrow$ Statement coverage
- Basic block cov $\Leftrightarrow$ Statement coverage

Decision coverage

Try to cover all decisions in the program with true and false

   decision: boolean expression

Measure:

decision coverage =

#decisions covered/ #decisions

Decision coverage

- Edge coverage ⇔ Decision coverage

Edge coverage

```
double abs(double x){
    if (x < 0) then x = -x;
    return x;
}
```

T1 (1 ;1)
T2 (-1; 1)
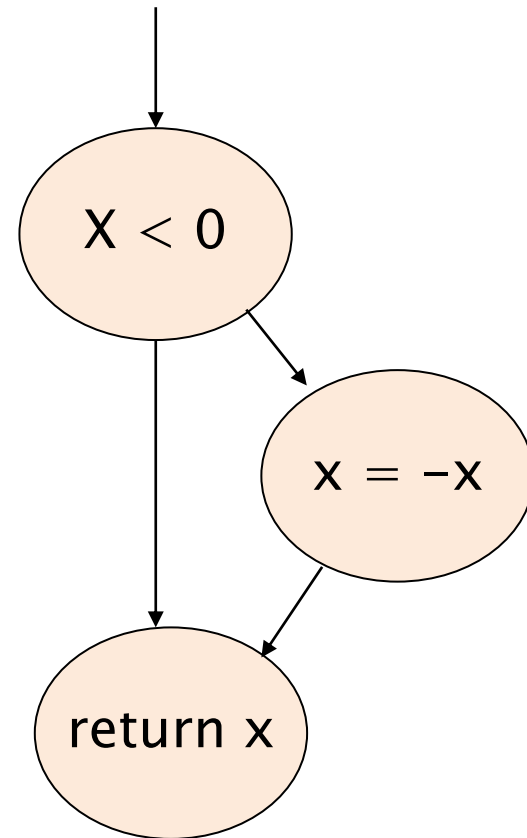
```
float homeworkAverage(float[] scores) {
   float min = 99999;
   float total = 0;
   for (int i = 0; i < scores.length; i++){
     if (scores[i] < min)
       min = scores[i];
     total += scores[i];
   }
   total = total - min;
   return total / (scores.length - 1);
}
```

T1({1}; 1)

T2({1,2}; 1.5)

min=9999

total=0

i=0

i<scores.length

true

scores[i]<min

tru
e

min=scores[i]

i++

total=total-min

total+=scores[i]

return
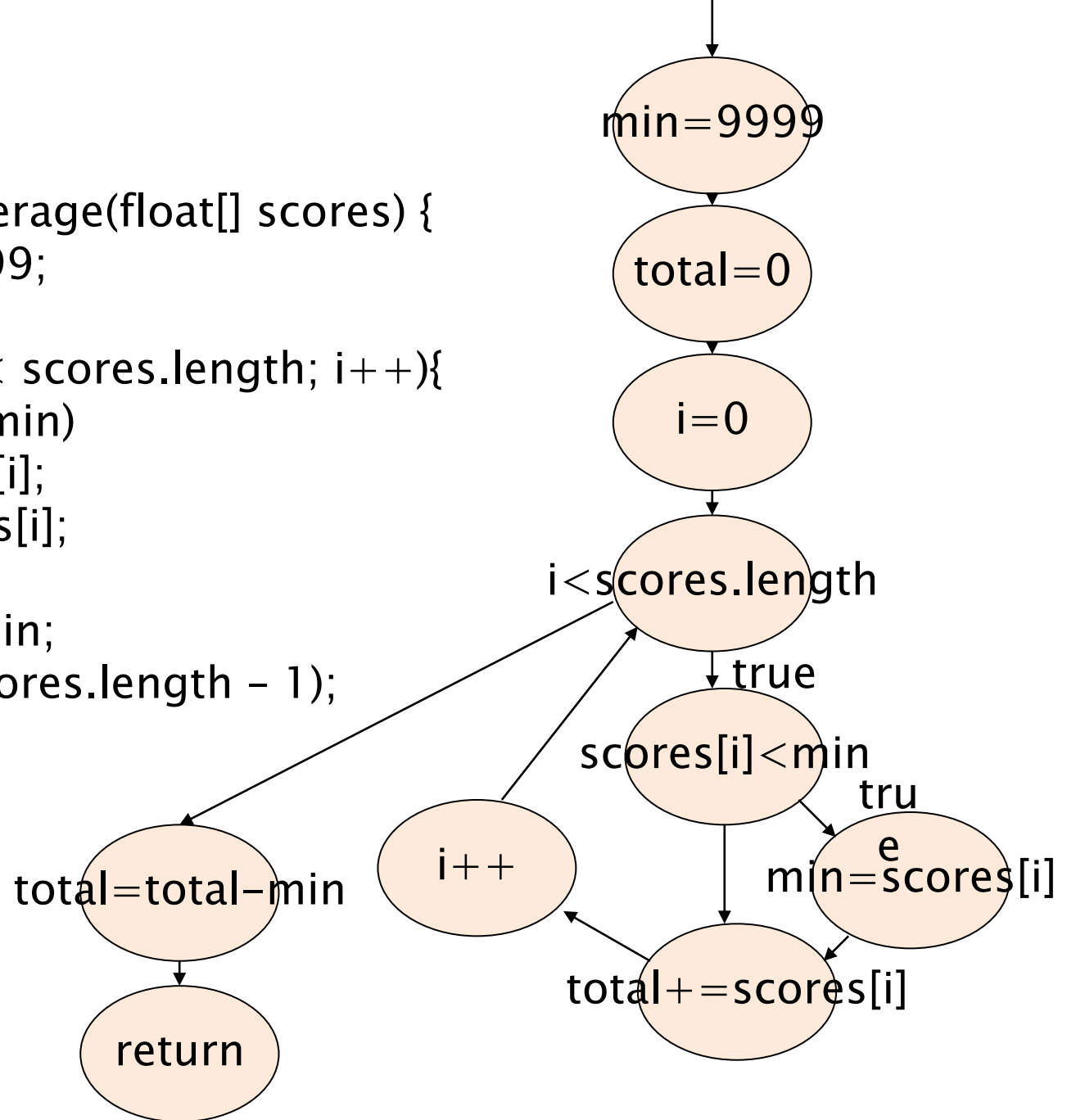
Relations

Edge coverage implies node coverage

   not viceversa

Condition coverage

A decision can be made of a combination of terms (== conditions)

Ex:

boolean isMarried;
 boolean isRetired;
 int age;
 if (age>60 and isRetired or isMarried)

Condition coverage

- **Simple condition coverage**
  - Each condition set at least once to T and F

- **Multiple condition coverage**
  - All combinations T/F are tried

Simple condition coverage

```
{
  boolean isMarried;
  boolean isRetired;
  int age;
  if (age>60 and isRetired or isMarried)
         discountRate = 30;
  else  discountRate = 10;
}
```

Test case	age>60	isRetired	isMarried	Decision
T1	T	T	T	T
T2	F	F	F	F

Multiple condition coverage

```
{
    boolean isMarried;
    boolean isRetired;
    int age;
    if (age>60 and isRetired or isMarried)
            discountRate = 30;
    else  discountRate = 10;
}
```

Test case	age>60	isRetired	isMarried	Decisio
T1	T	T	T	T
T2	T	T	F	T
T3	T	F	T	T
T4	T	F	F	F
T5	F	T	T	T
T6	F	T	F	F
T7	F	F	T	T
T8	F	F	F	F

146

Relations

Multiple condition coverage
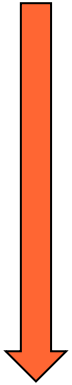
Simple condition coverage
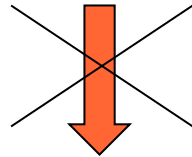
Decision coverage
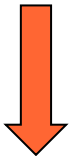
Statement coverage

Test case	age>60	isRetired	isMarried	Decision
T1	T	T	T	T
T2	T	T	F	T
T3	T	F	T	T
T4	T	F	F	F
T5	F	T	T	T
T6	F	T	F	F
T7	F	F	T	T
T8	F	F	F	F

- T2 and T7 provide simple condition coverage, but no decision coverage

Path coverage

- Path = sequence of nodes in a graph
- select test cases such that every path in the graph is visited

Path coverage

- Ex. 4 paths in this simple graph
  - 1,2,3,4,5
  - 1,3,5
  - 1,3,4,5
  - 1,2,3,5
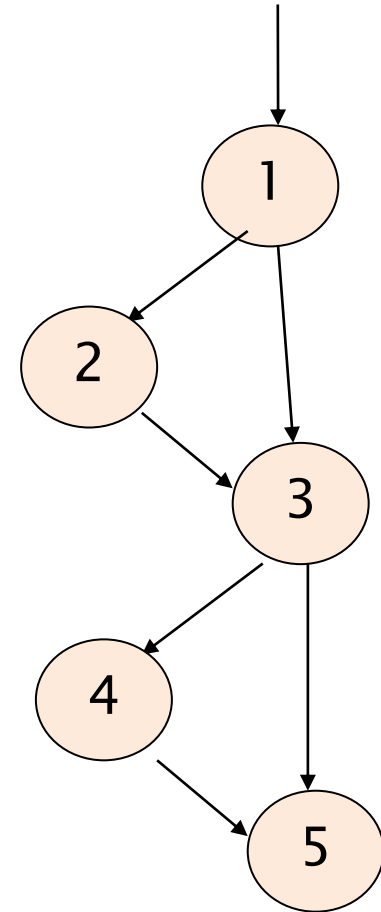
Path coverage

- **Combinatorial explosion with cycle**
  - ◆ 1,3,5
  - ◆ 1,3,5,1,3,5
  - ◆ 1,3,5,1,3,5,1,3,5
  - ◆ Etc ..
  - ◆ Npaths $= 4^{nloops}$
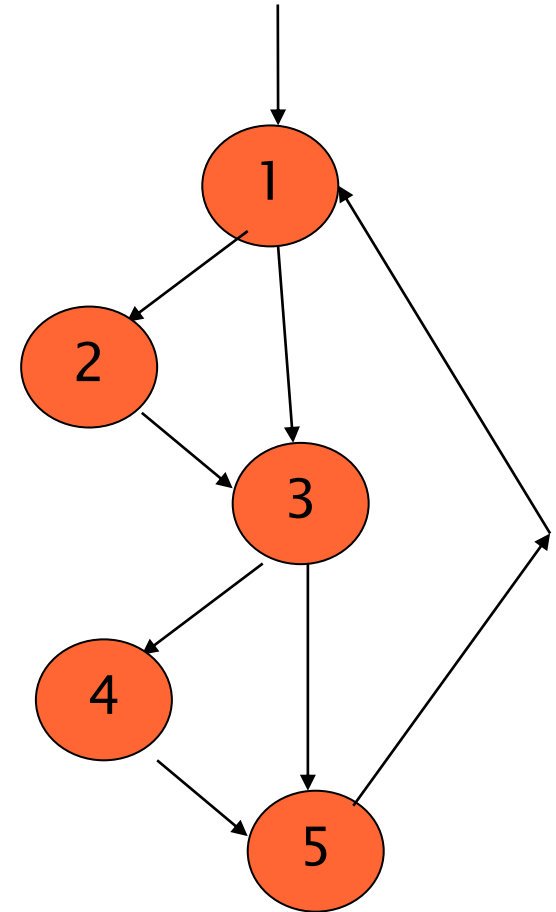
Path coverage

- In most cases unfeasible if graph is cyclic

- Approximations
  - Path-n
    - Path-4 == loop 0 to 4 times in each loop
  - Loop coverage
    - In each loop cycle 0, 1 , >1 times

Loop coverage

- select test cases such that every loop boundary and interior is tested
  - Boundary: 0 iterations
  - Interior: 1 iteration and > 1 iterations

  - Coverage formula: $x/3$

Loop coverage

- Consider each loop (for, while) separately

- Write 3 test cases
  - No enter the loop
  - Cycle once in the loop
  - Cycle more than once

Loop coverage

```
float homeworkAverage(float[] scores) {
    float min = 99999;
    float total = 0;
    for (int i = 0; i < scores.length; i++){
        if (scores[i] < min)
            min = scores[i];
        total += scores[i];
    }
    total = total - min;
    return total / (scores.length - 1);
}
```

T1({1}; 1) loops 1

T2({1,2,3}; 2) loops 2

T3({}; ?) loops 0

min=9999

total=0

i=0

i<scores.length

true

scores[i]<min

true

i++

min=scores[i]

total=total-min

total+=scores[i]

return
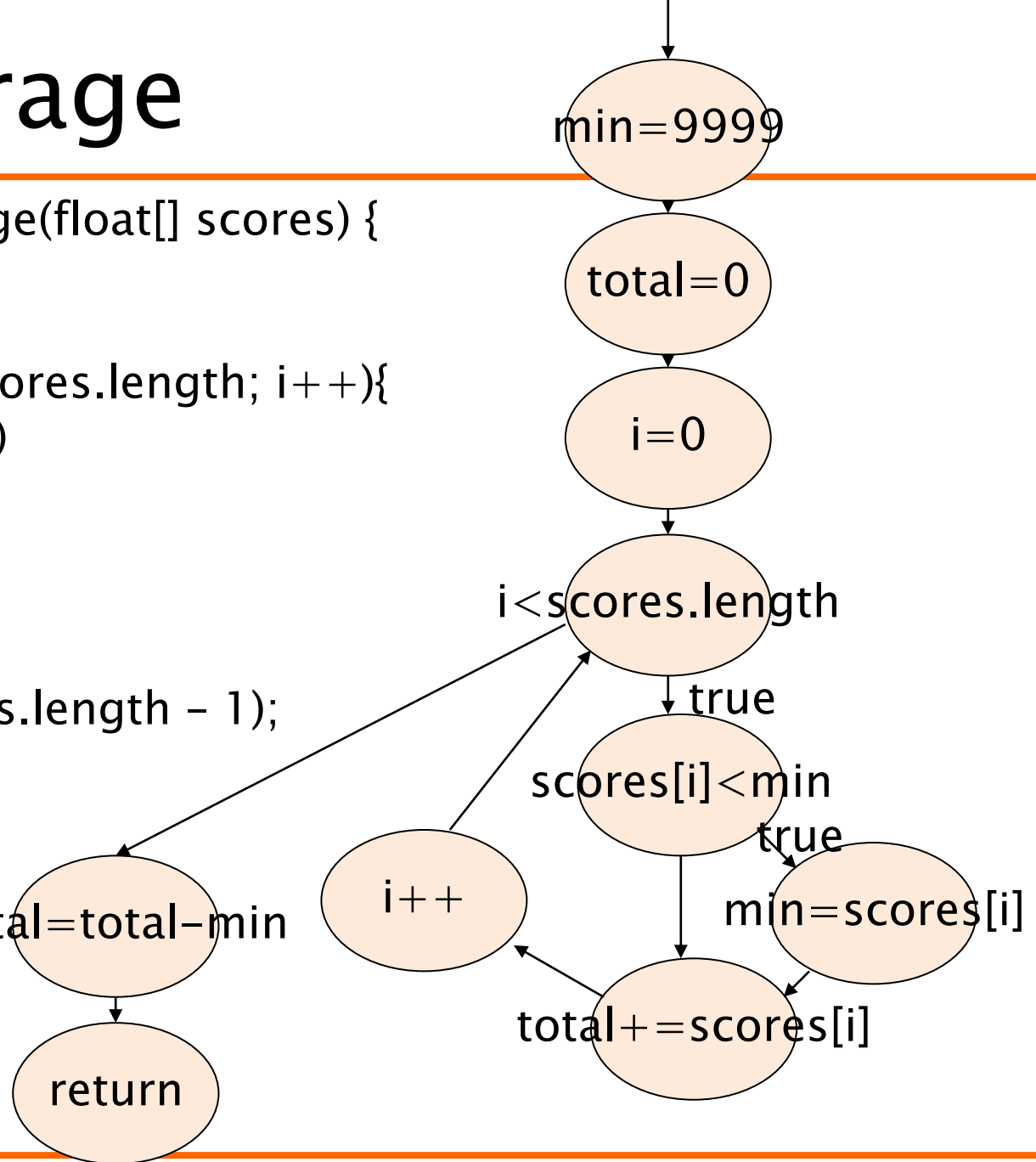
Tools

- To write and run test cases
  - ◆ Ex JUnit
- To compute coverage
  - ◆ Ex. Cobertura, Clover, Jacoco

JUnit


```java
import junit.framework.TestCase;


public class TAverage extends TestCase {

    public void testT1(){
        float scores[] = {1};
        float ave = Average.homeworkAverage(scores);
        assertEquals(1, ave);
    }

    public void testT2(){
        float scores[] = {1,2 };
        float ave = Average.homeworkAverage(scores);
        assertEquals(1.5, ave);
    }

    public void testT3(){
        float scores[] = {};
        float ave = Average.homeworkAverage(scores);
        assertEquals(0, ave);
    }
}
```

157

Cobertura

Summary

- Structural / white box testing starts from the code, and uses several coverage objectives

  - Statements
  - Decisions
  - Conditions (simple, multiple)
  - Path
  - Loop

Summary

- White box testing is typically made in the development environment and supported by tools to compute coverage

Integration test

Integration test

Requirements engineering	Requirement document	VV requirements	Requirement document

Design .	Design document	VV design	Design document

Implement unit / Implement unit	Unit / Unit	VV unit / VV unit	Unit / Unit

Integrate units	System	VV system	System

Project management
Configuration management
Quality assurance
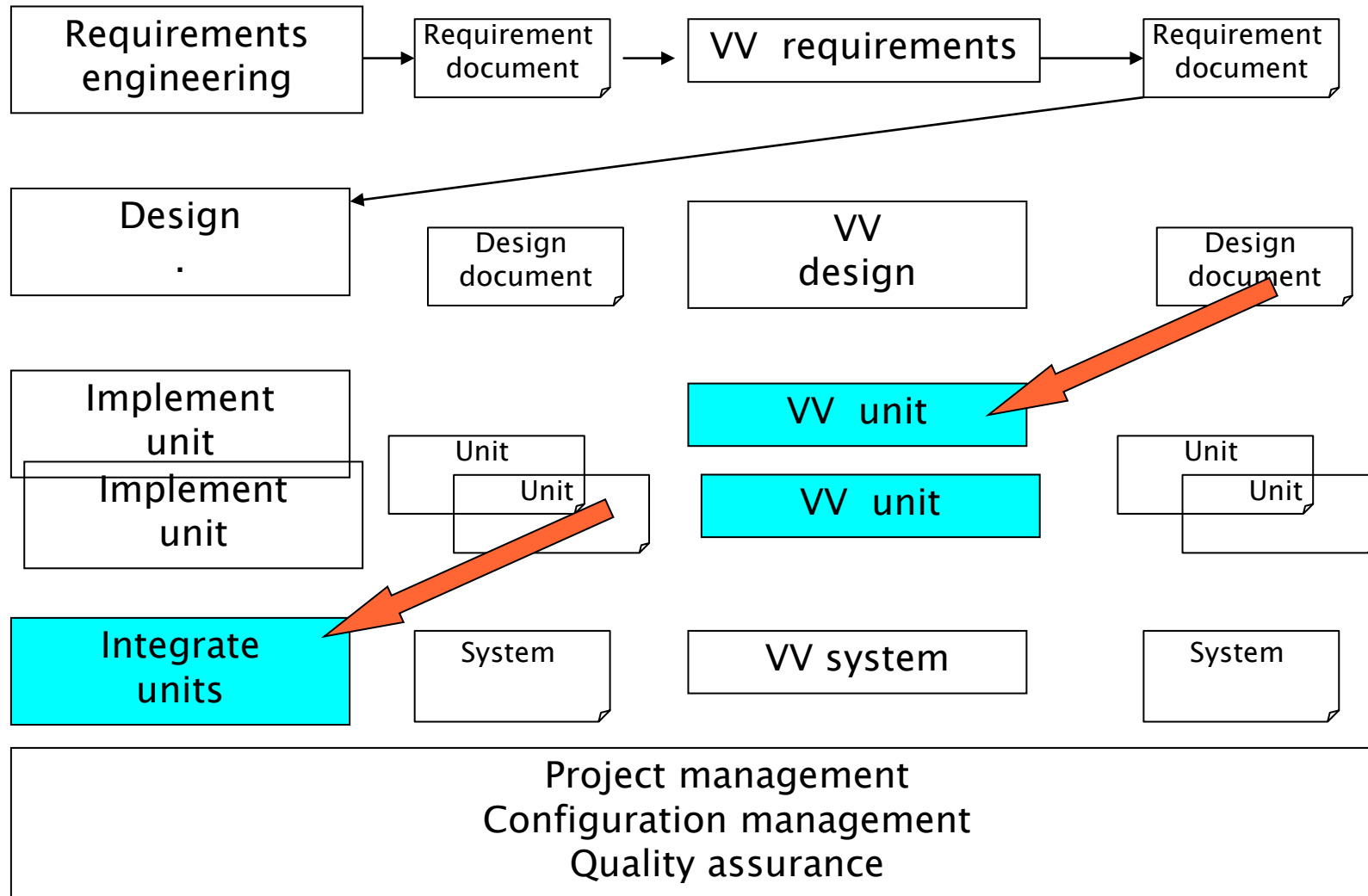
Integration test

- Test of some <u>dependent</u> units
  - ◆ Unit:
    - – function (procedural languages)
    - – class and its methods (oo languages)

The problem

- Some units need others. How to test them?

Person

+name
+surname
+birthdate
+birthcity
+sex
+SSN

SSNGenerator

+computeSSN()

CityCodeGenerator

+cityNameToCityCode()
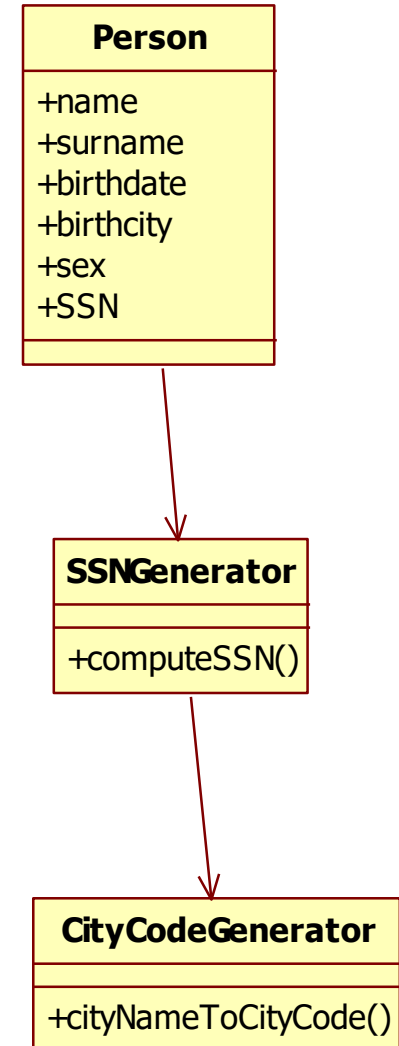
The dependency graph

function1() { //some code
           call function2();
           call function3();
           // some other code}
function2() { //some code
           call function4();}

```
function1
```

```
function2      function3
```

```
function4
```
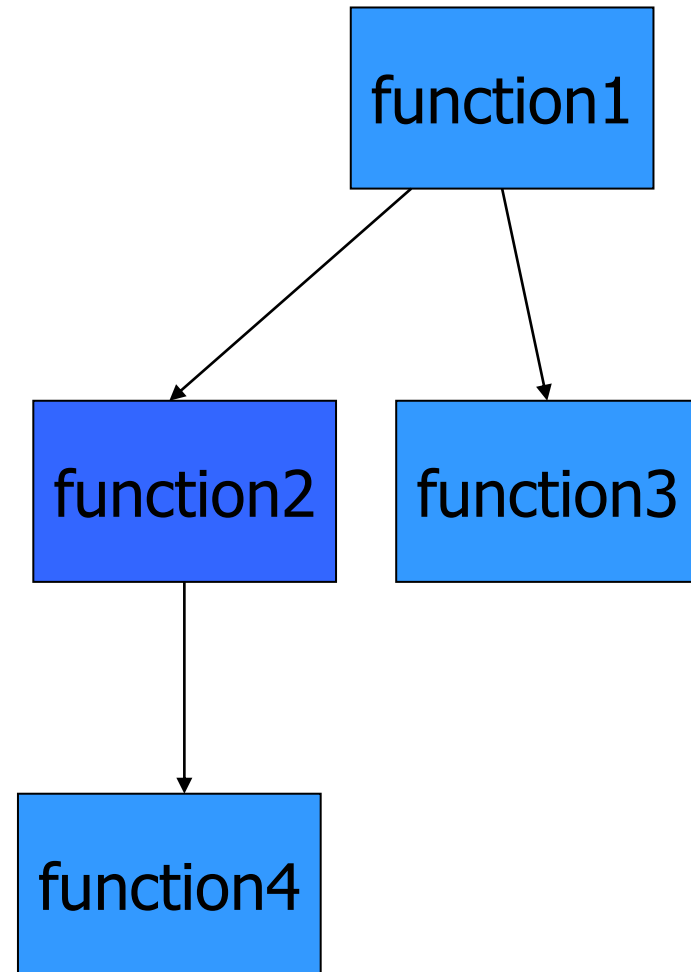
The problem -2

- Function3, function4 have no dependency
  - ◆ Unit test techniques can be applied
- Function1, function2 have dependencies, how to test them?
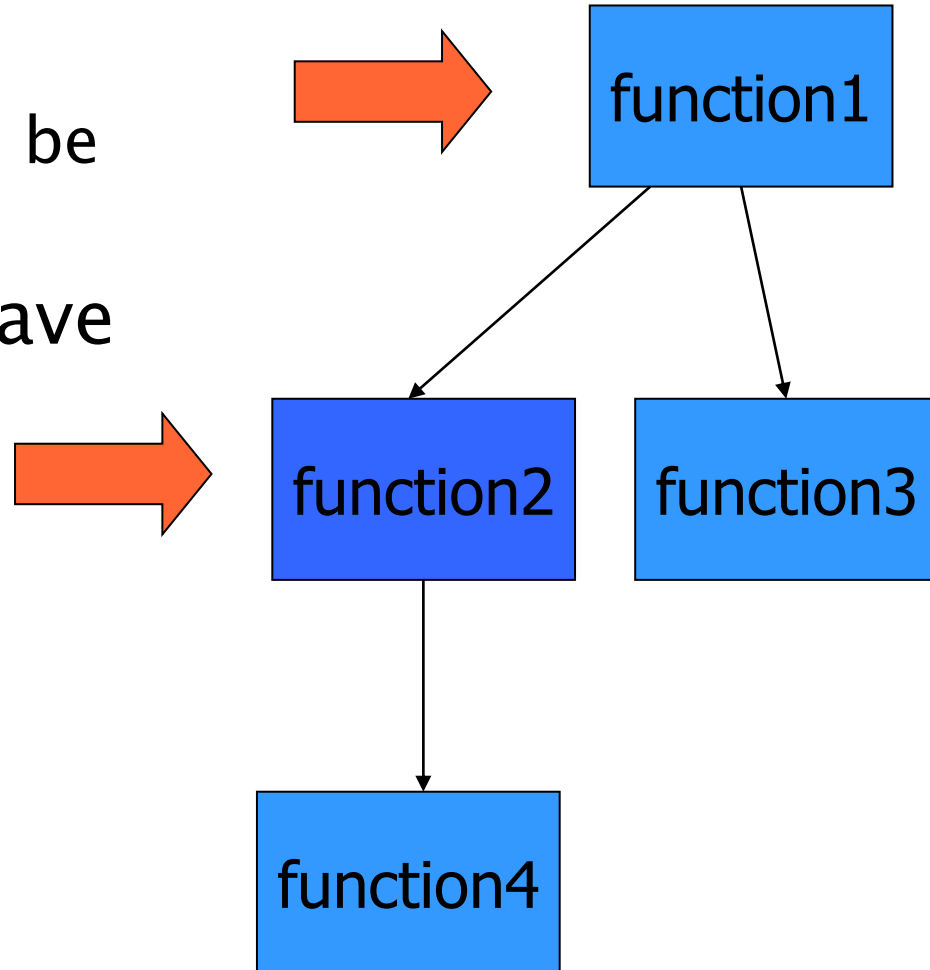
In fact two (related) problems

- How to test 'independently' a function that has dependencies?
  - Ex Function1()

- How to test the dependency?

Test independently a function with dependencies

Technique

- Stubs

- Try to eliminate the dependency using substitutes

Goal: test function2
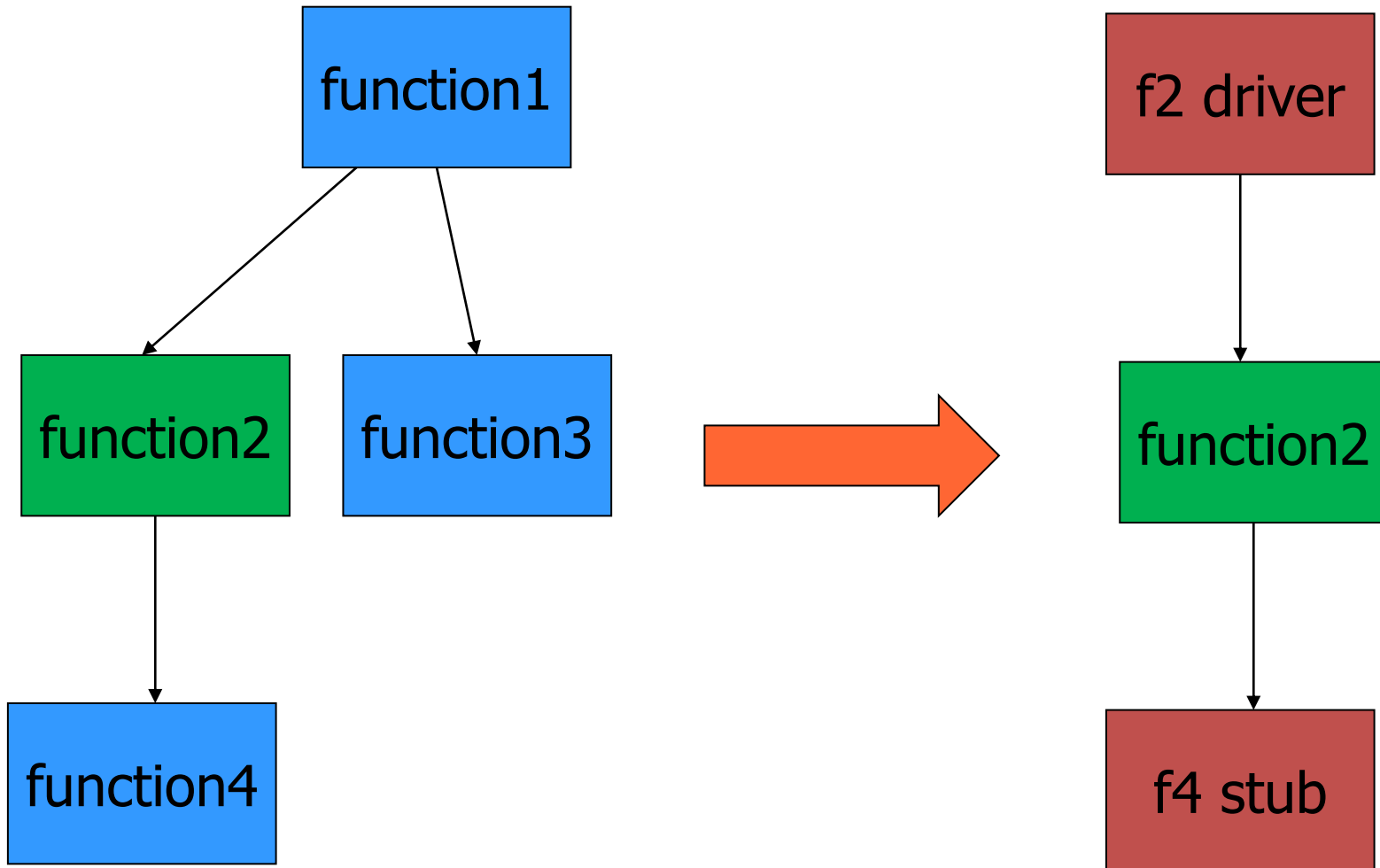
Stub, driver

- Driver
  - Unit (function or class) developed to pilot another unit
- Stub
  - Unit developed to substitute another unit (fake unit)

- Also called mock ups, mocks

Ex. driver (JUnit)

```java
public class ConverterTest extends TestCase {

    public void testOne(){
        Converter c = new Converter();
        char [] str = {'1', '2','3','4','5','6', '7'};
        try {
                c.convert(str);
                fail();
        } catch (Exception e) {
          assertTrue(true);
        }
    }
}
```

```java
public class Converter {

    public int convert(char[] str) throws Exception {
        if (str.length > 6)
                throw new Exception();
        int number = 0;
        int digit;
        int i = 0;
        if (str[0] == '-')
                i = 1;
        for (; i < str.length; i++) {
                digit = str[i] - '0';
                number = number * 10 + digit;
        }
        if (str[0] == '-')
                number = -number;
        if (number > 32767 || number < -32768)
                throw new Exception();
        return number;
    }

}
```

172

Stub

- Must be simpler than unit substituted (trade off between simplicity and functionality)

  - Ex. unit = function to compute social security number from name/family name etc.

  - Stub = returns always same ssn

  - Ex. unit = catalog of products, contains thousands of them

  - Stub. Contains 3 products, returns one of them
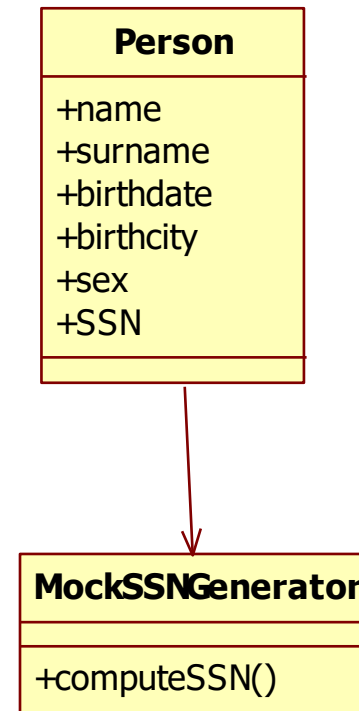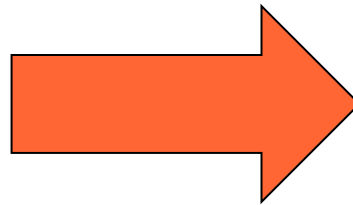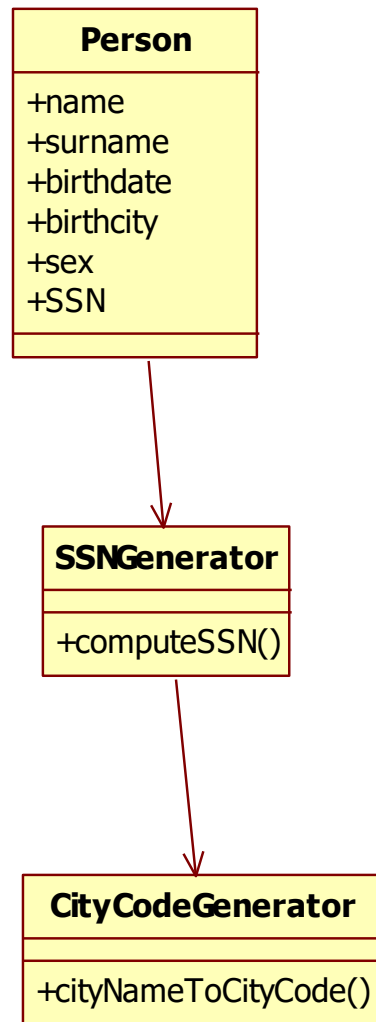
Stub, embedded systems

- Substitues sensors / actuators with pure software units

  - Ex, heating control system
  - Temperature sensor stub, rain sensor stub etc
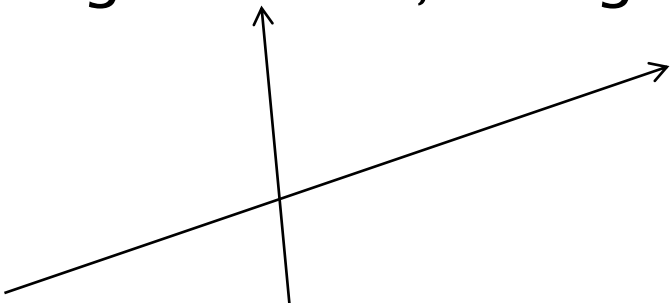
Test dependencies between functions

Dependency defect

- Two units work perfectly when isolated
- Defect happens when connected

Ex. Dependency defect

Class Person{

    public Person(String surname, String name){}

    }

main() {

    new Person("John", "Wright")

}

Ex. Dependency defect

triangleSurface(float height, float base){}

main() {
 triangleSurface(10, 4); // int instead of float
 triangleSurface(3.1, 4.2); // exchanged base (3.1)
                                // and height (4.2)

}

Ex Dependency defect

- ## The Mars polar lander case (1999)

  ```
  double function( double metricMeasure){}

  main() {
      double imperialMeasure;

          function( imperialMeasure);

   }
  ```

- ## UK team develops unit assuming metric input

- ## US team calls unit assuming imperial input

- ## Difference is small, noticed only at 'landing'

Technique

- Two units are tested in isolation first
- Integrate them, test again
  - Focus on the dependency

- In case of more units, integrate incrementally

- In any case, avoid BIG BANG integration

Goal: test dependency function2-function4
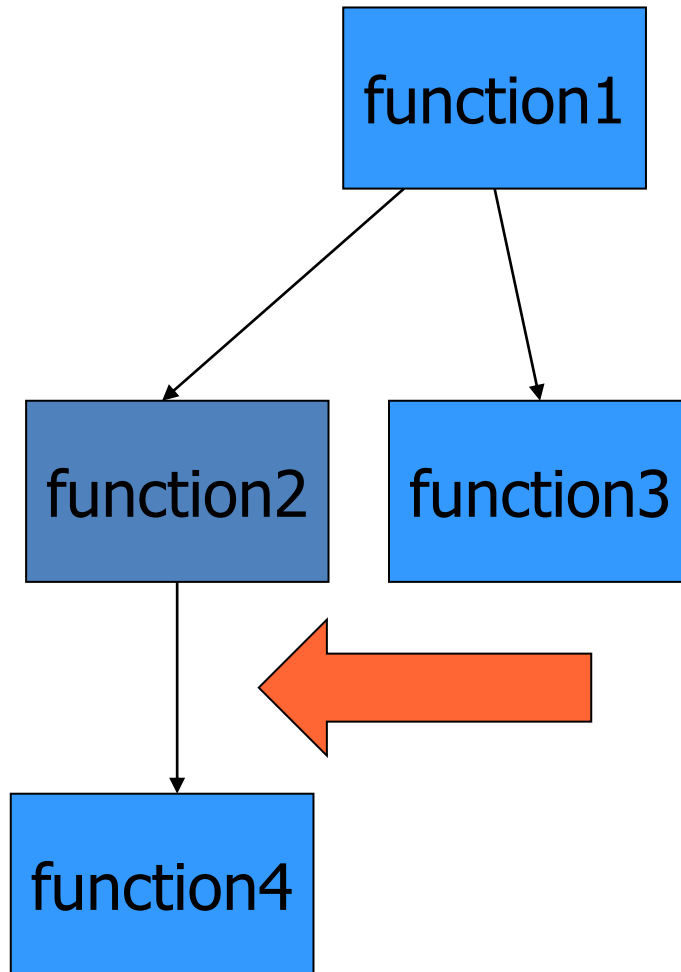
1 test function4 (unit test)

2 test function 2, with stub for function4
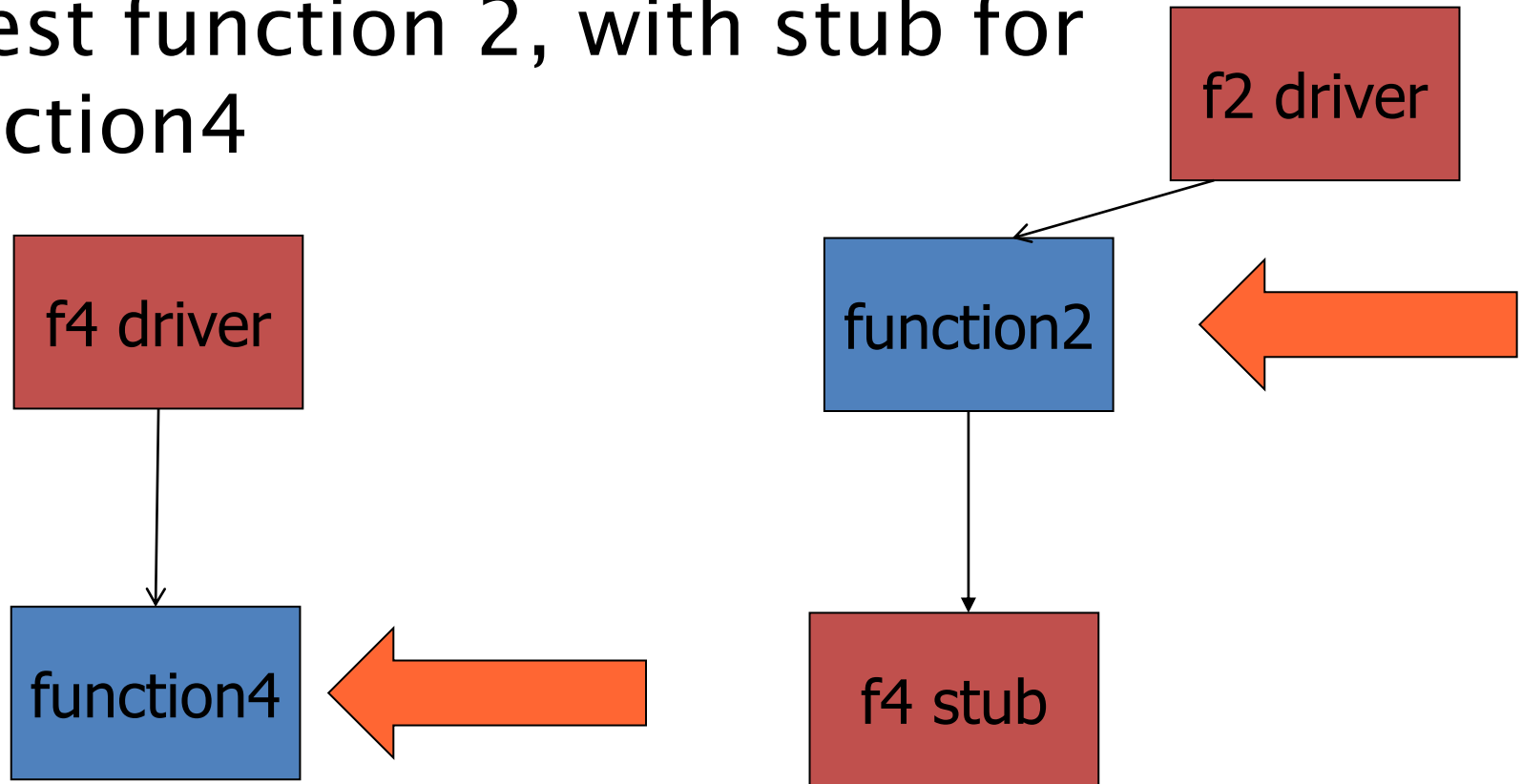
3 integrate function 2 and 4, test the dependency

f2 driver

function2
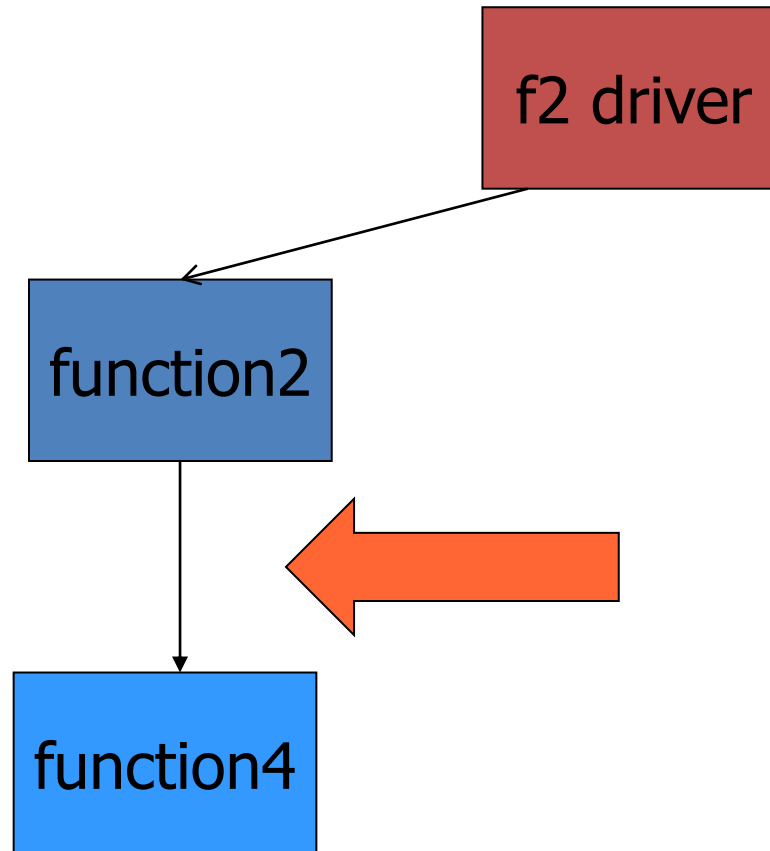
function4

- ◆ Other simpler option
  - ▪ 1 test function4 (unit test)
  - ▪ 2 integrate function2, test both

f2 driver

f4 driver

function2

function4

function4
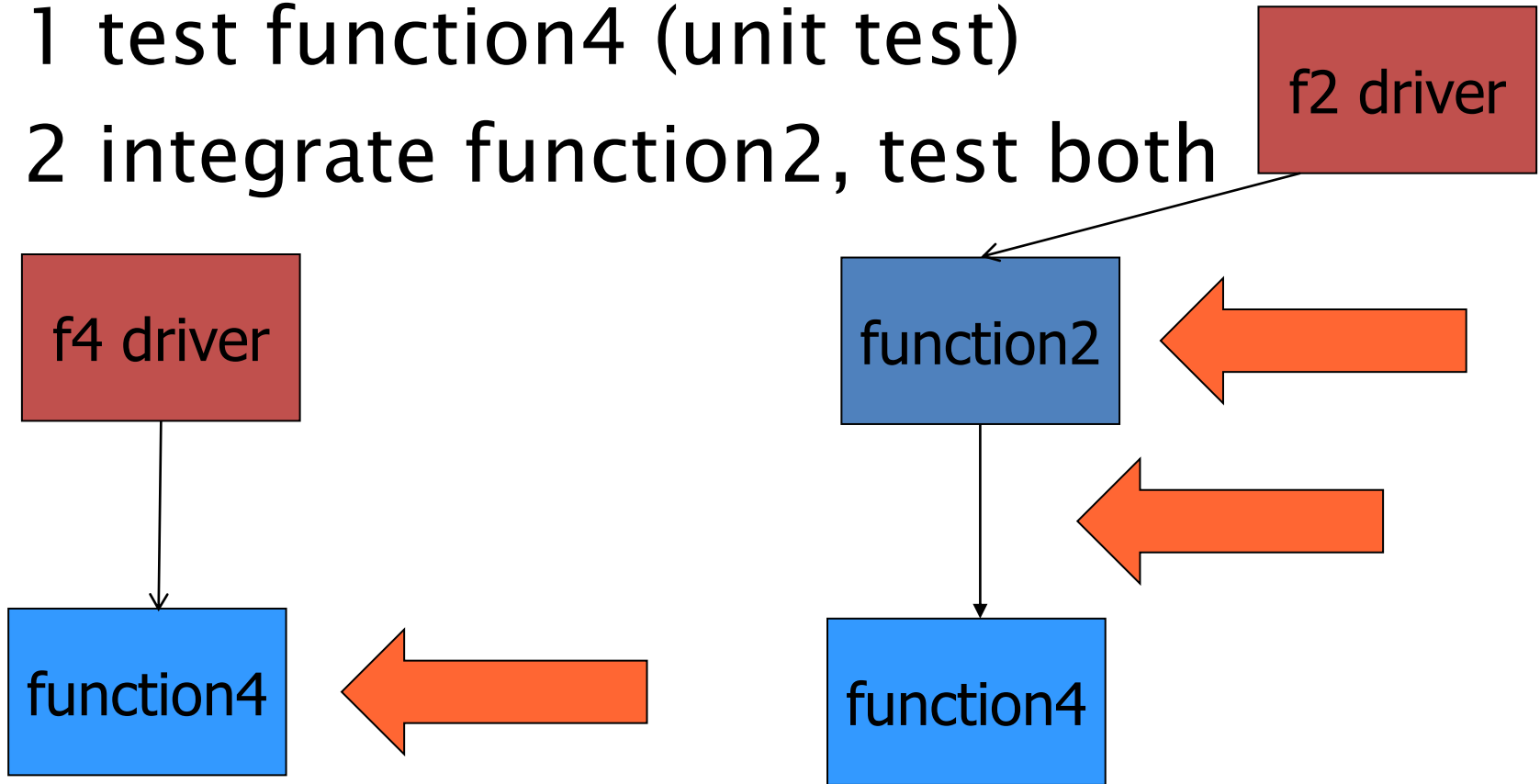
Incremental integration

- Goal:
  - Add one unit at a time, test the partial aggregate
- Pro:
  - Defects found, most likely, come by last unit/interaction added
- Con:
  - More tests to write, stubs/drivers to write

Incremental integration

- **Top down**

- **Bottom up**
  - Defined relatively to the dependency graph
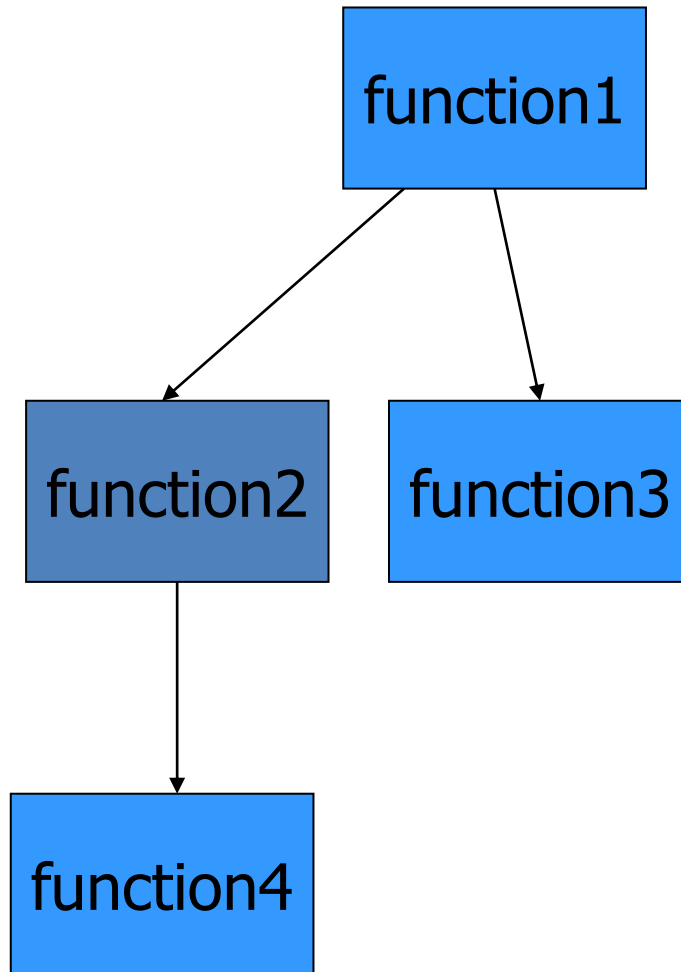
Bottom up

- Step 1
  - function4, function3 have no dependencies. Test them in isolation (unit test)

Test cases function3 → function3

Test cases function4 → function4
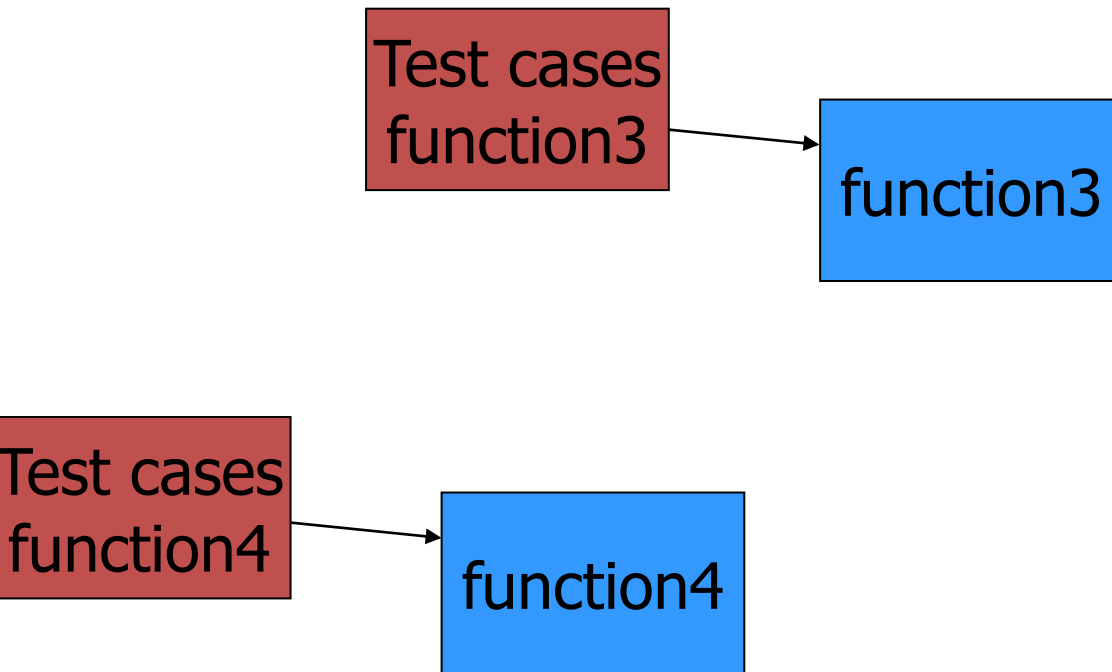
Step 2

- Test function2+function4 as-if they were one unit
- if defect is found, it should come
  - from function2 or
  - from interaction function1-function4
  - Not from function 4, that is now 'trusted'

Test cases
Function2+4

function2
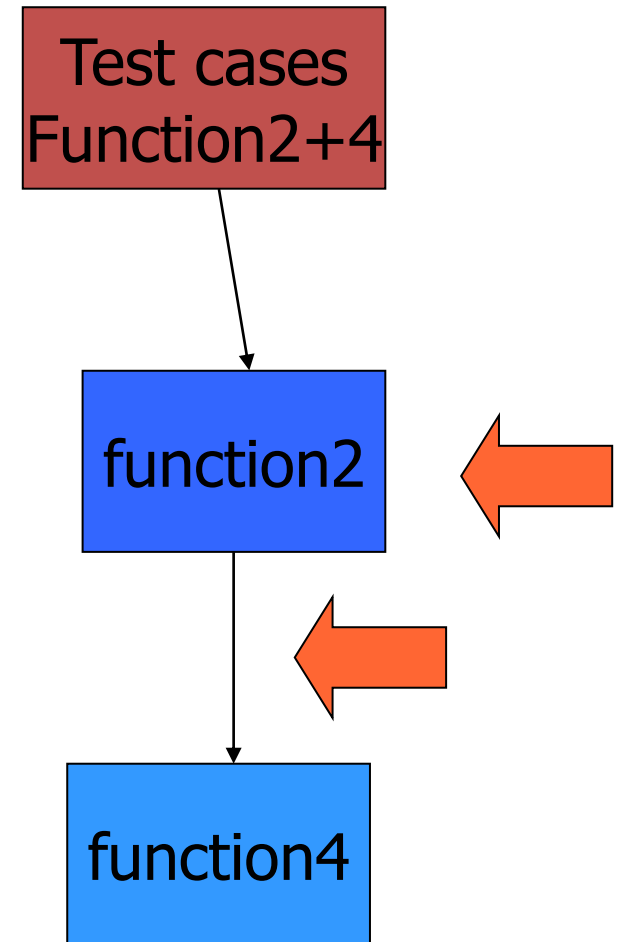
function4

Step 3

- ◆ Test all
- ◆ if defect is found, it should come
  - – from function1 or
  - – from interaction function1-function2
  - – from interaction function1-function3
  - – Not from function 2+4, and function3, that are now 'trusted'

Test cases

function1

function2   function3

function4

Top down

- Step 1

Test cases → function1 → f2 stub, f3 stub
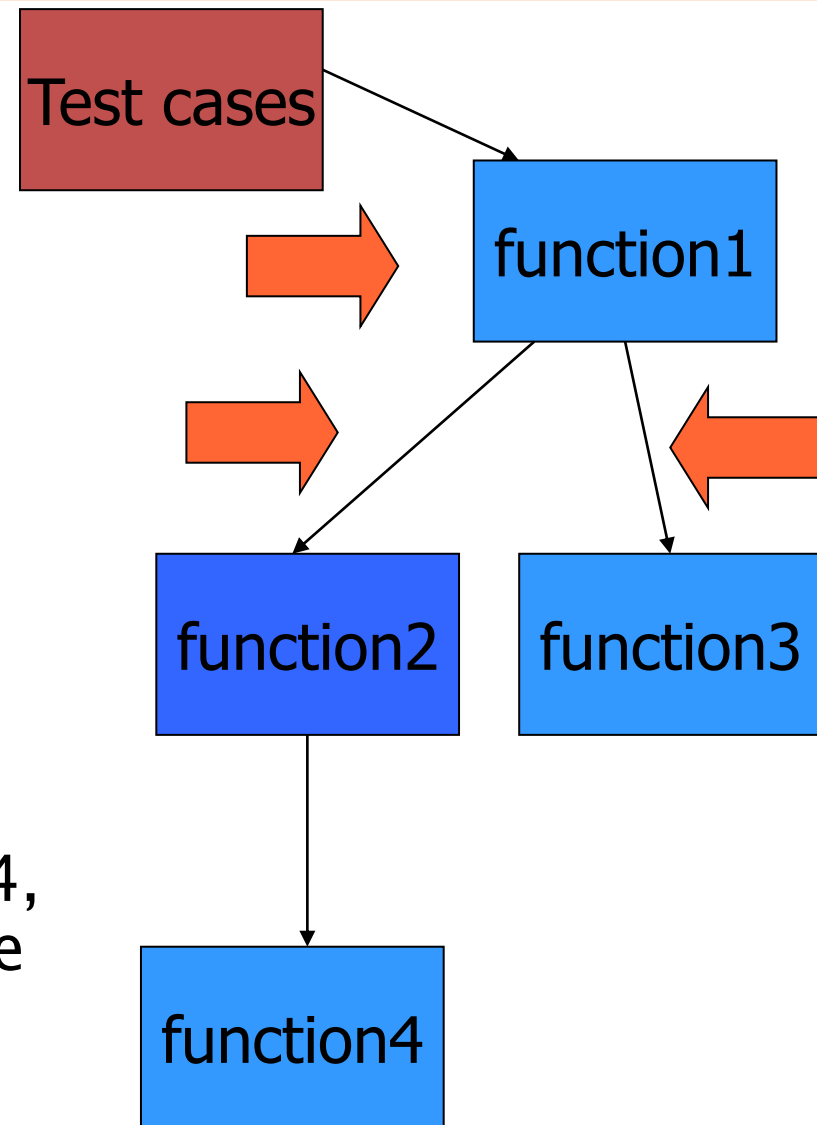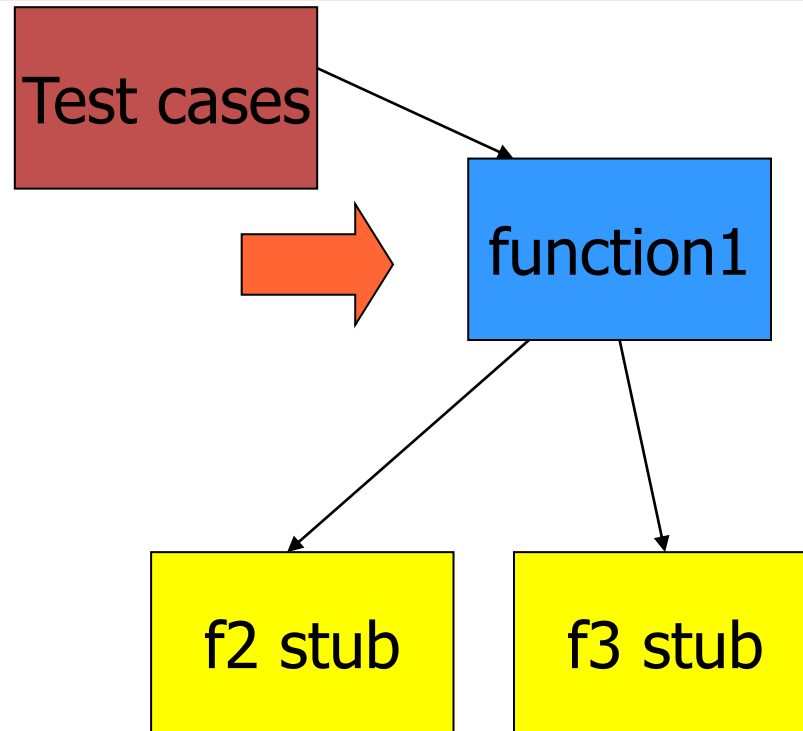
Top down

- Step 2

Top down

- Step 3

Test cases

function1

function2   function3

function4
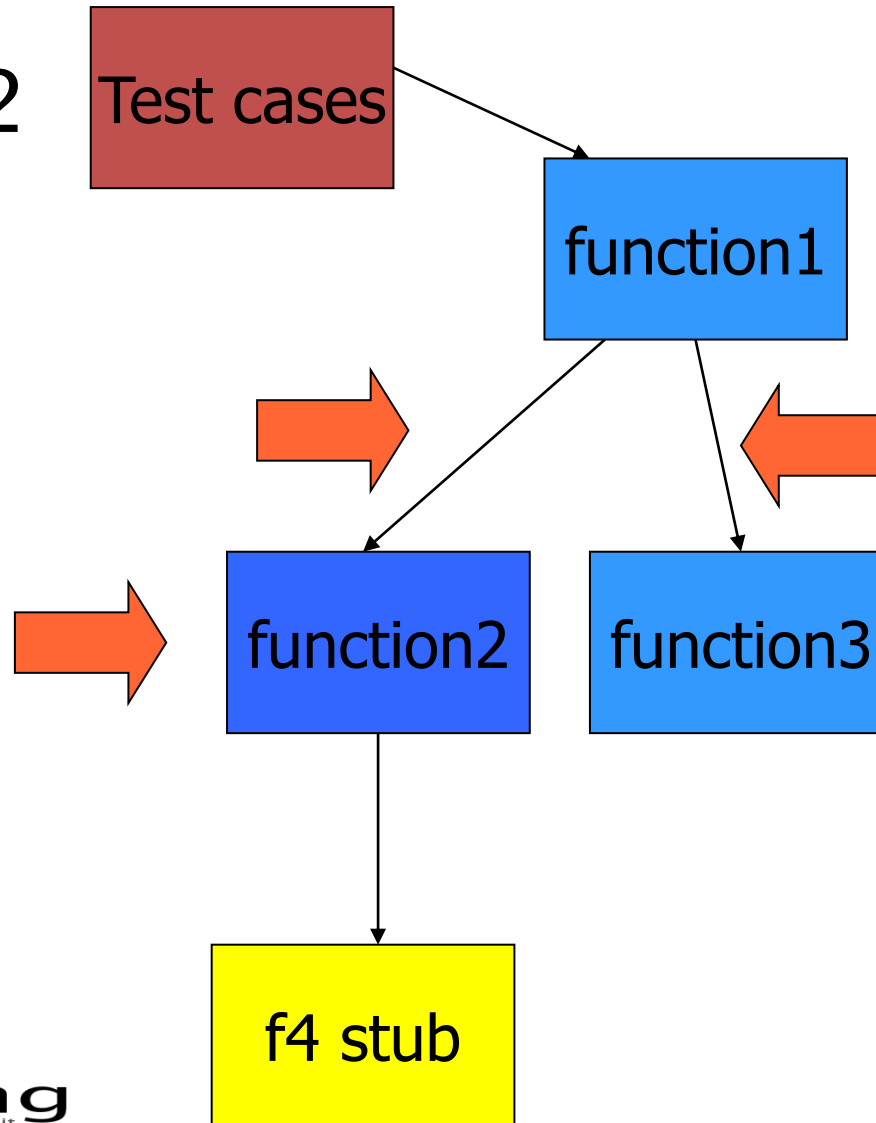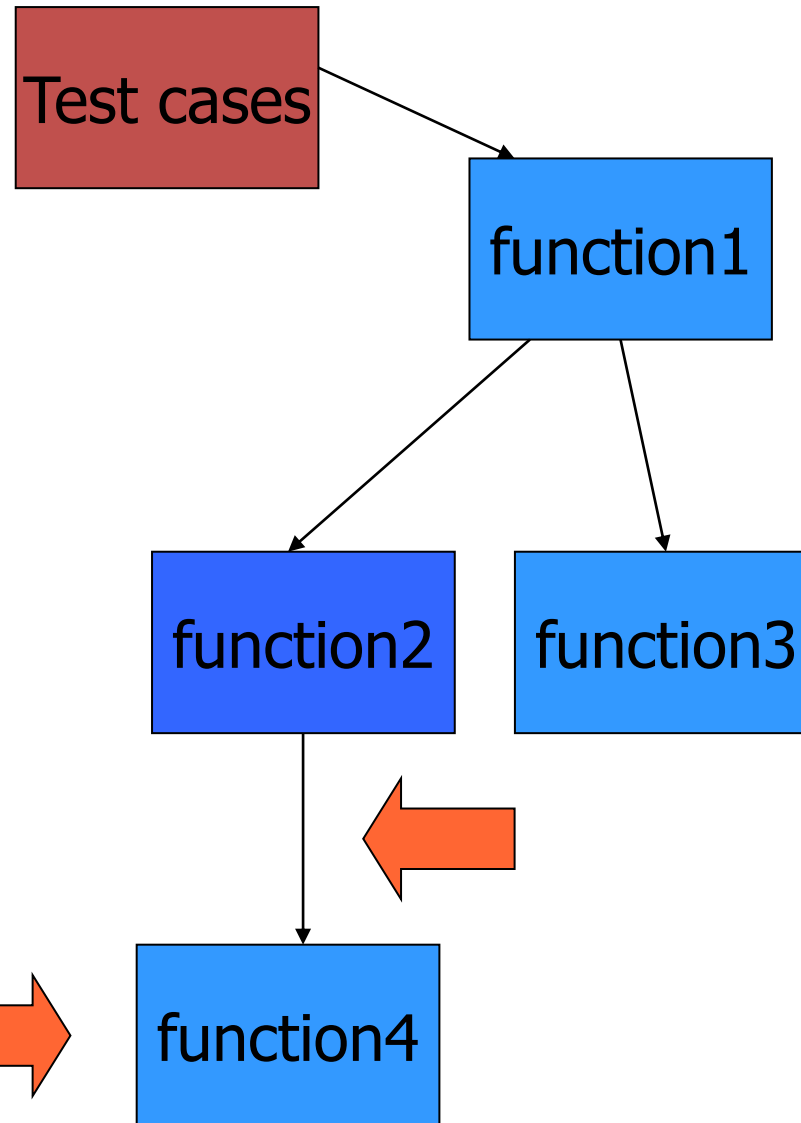
Top-down

- Pros
  - Allows early detection of architectural flaws
  - A limited working system is available early
- Cons
  - Requires the definition of stubs for all lower level units
  - Suitable only for top-down development
 - Lower levels not directly observable

SOftEng
http://softeng.polito.it

Bottom-up

- Pros
  - ◆ Testing can start early in the development process
  - ◆ Lower levels are directly observable
- Cons
  - ◆ Requires the definition of drivers for all lower level units

In practice

- Incremental integration is a mix of top down and bottom up, trying to
  - Minimize stubs creation
  - Compromise with availability of units
  - (units are developed in an order suitable to the integration sequence decided)

Big bang integration

- **All functions are developed**
- **Test is applied directly to the aggregate as-if it were a single unit**
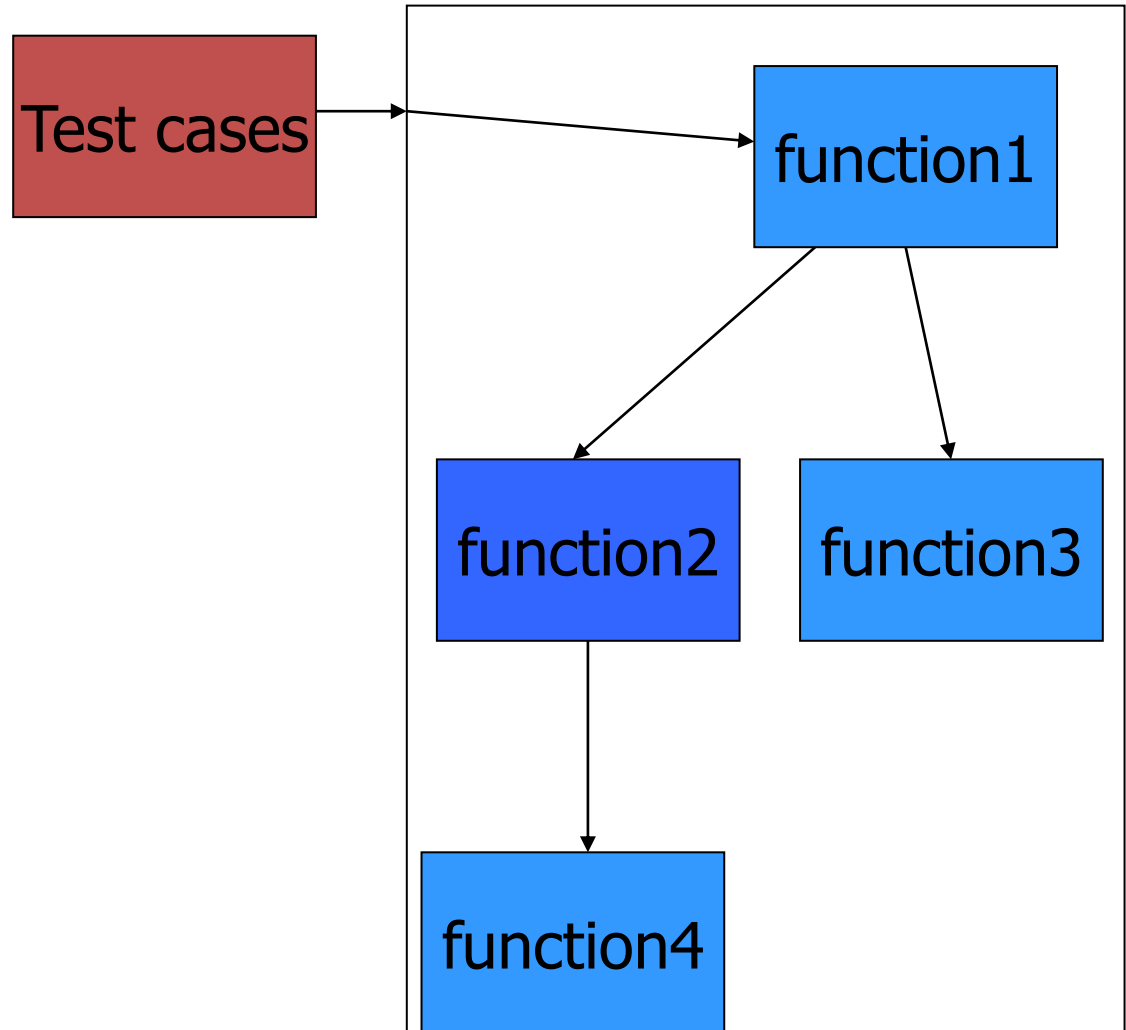
Problems

- When a defect is found, how to locate it?

  - Could be generated
    - by any function
      - function1, 2, 3, 4
    - by any interaction
      - function1 to function2, function1 to function3, function2 to function4
   - Interaction problems: bad parameter passed, parameter passed in wrong order, in wrong timing

Example

- A company has employees and departments, each employee belongs to a department
- Payroll
  - salary(ID) – returns salary given employee ID
  - bonus(amount) – gives bonus 'amount' to employees with higher sales record
- Employee
  - name, surname, ID, salary
- Department
  - name, id, salesPerYear
  - addEmployee
- Employees
  - add employee
  - search employee, returns employee given ID
- Departments
  - add department
  - search department, returns department given ID
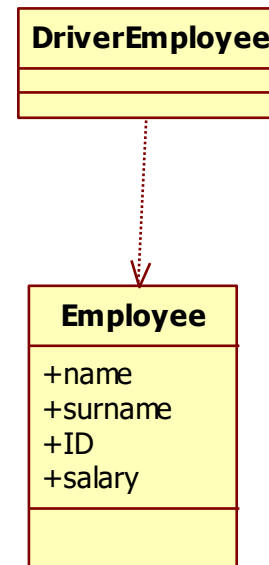
Class diagram (1)
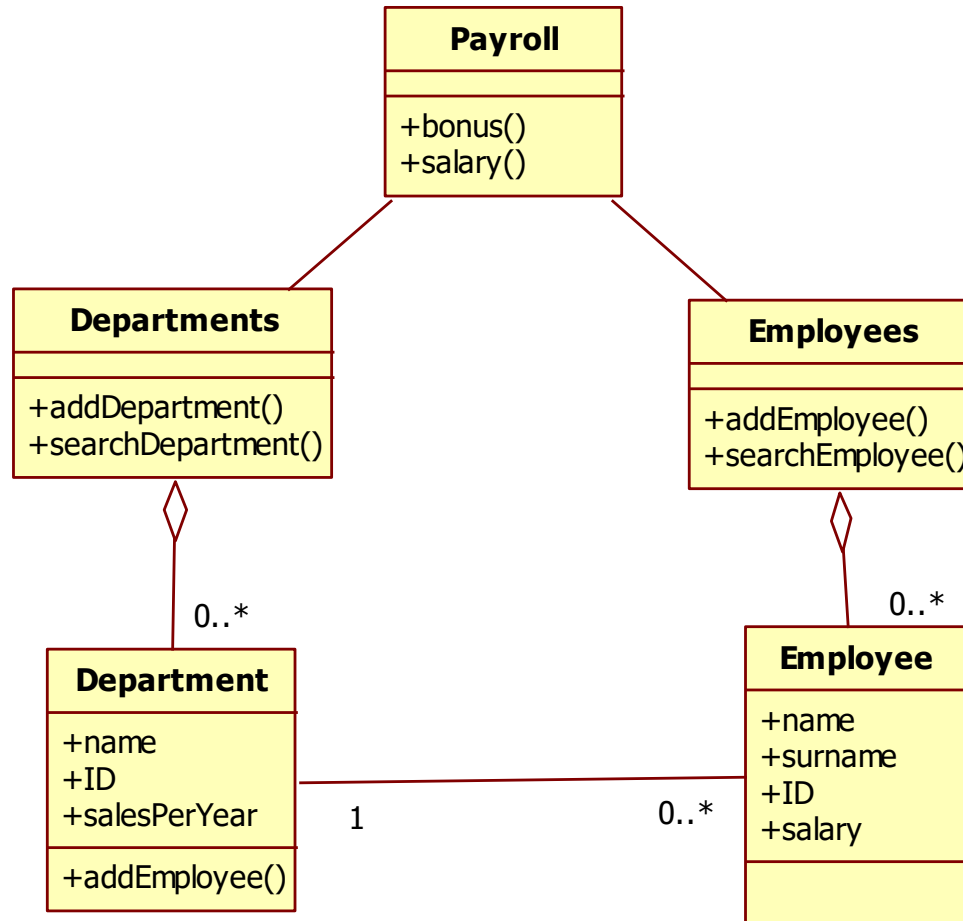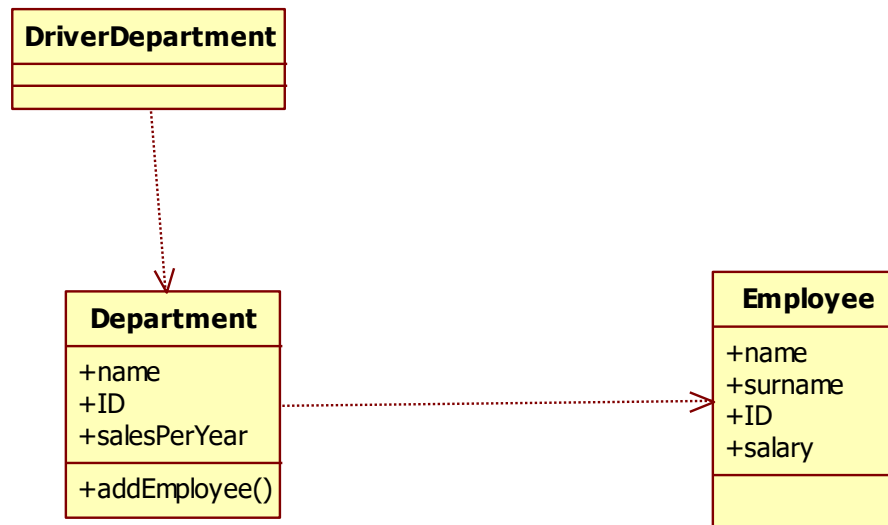
Dependencies (1)
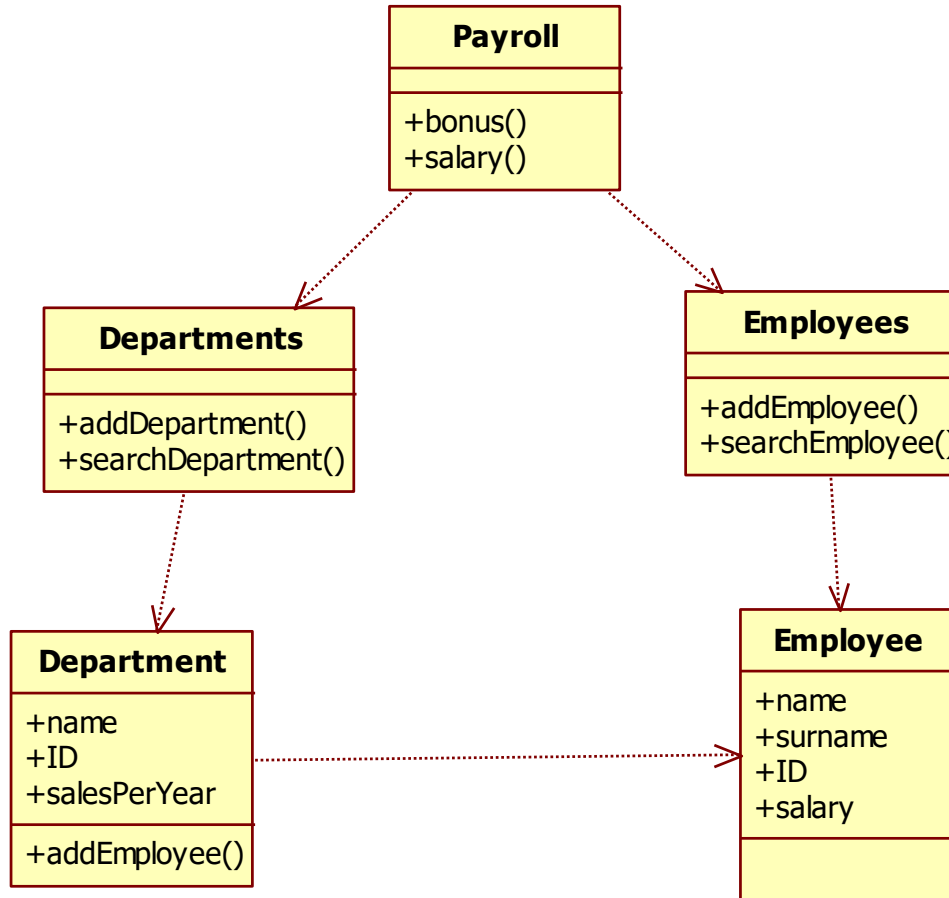
Integration

- Bottom up
  - Employee
  - Employees   Department
  - Departments
  - Payroll

- Top Down
  - Payroll
  - Departments, Employees
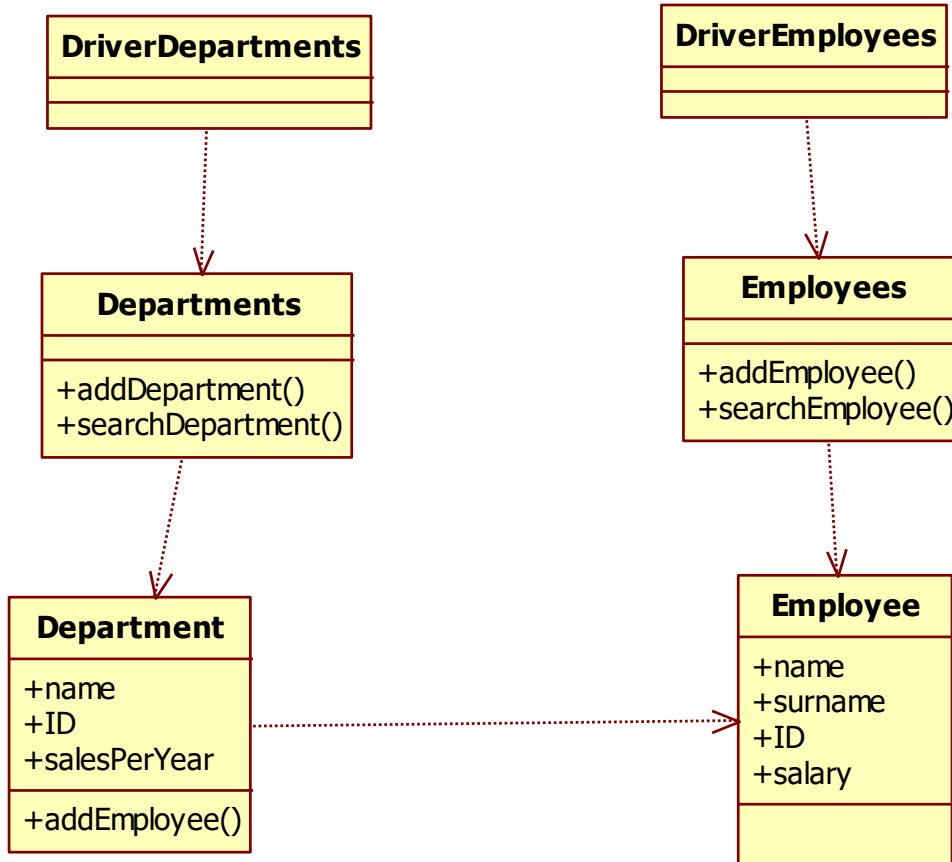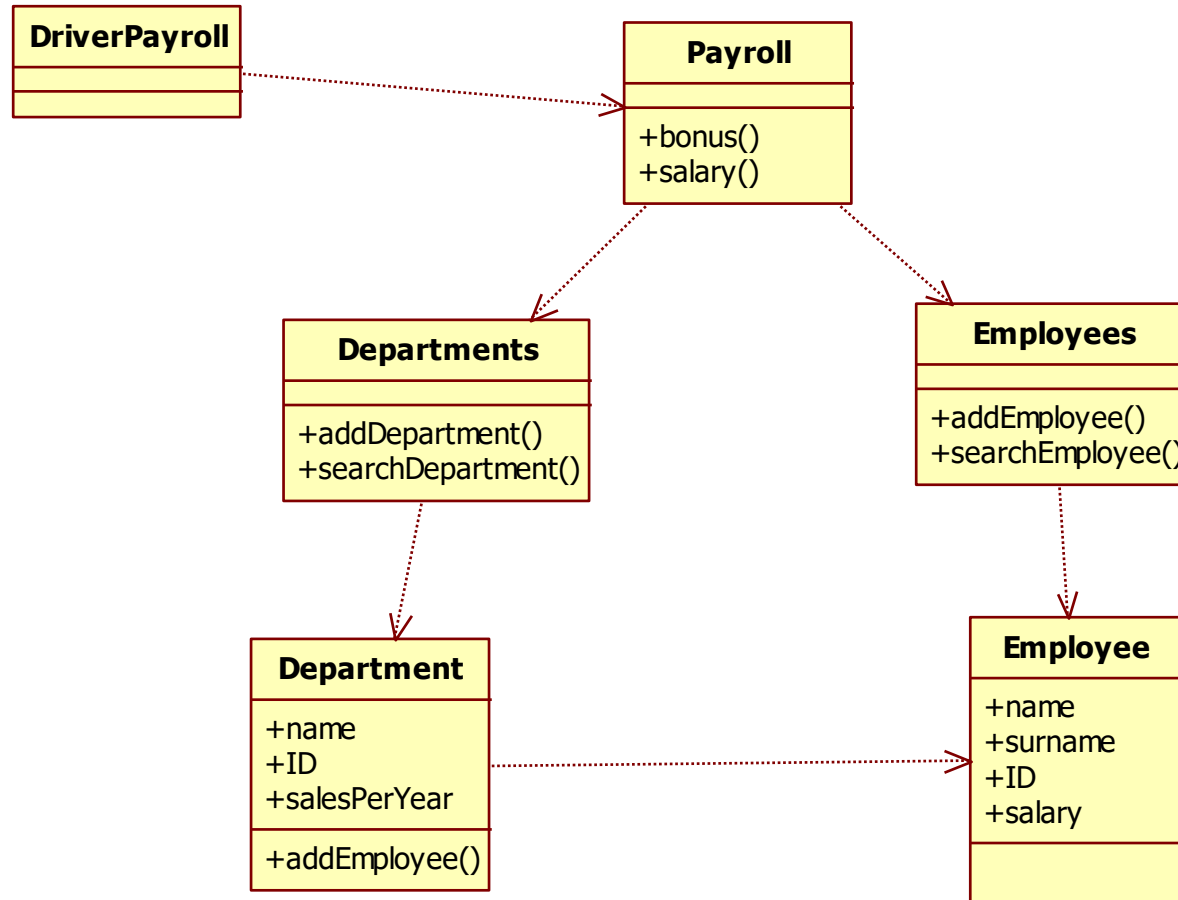
BU – 1

BU – 2
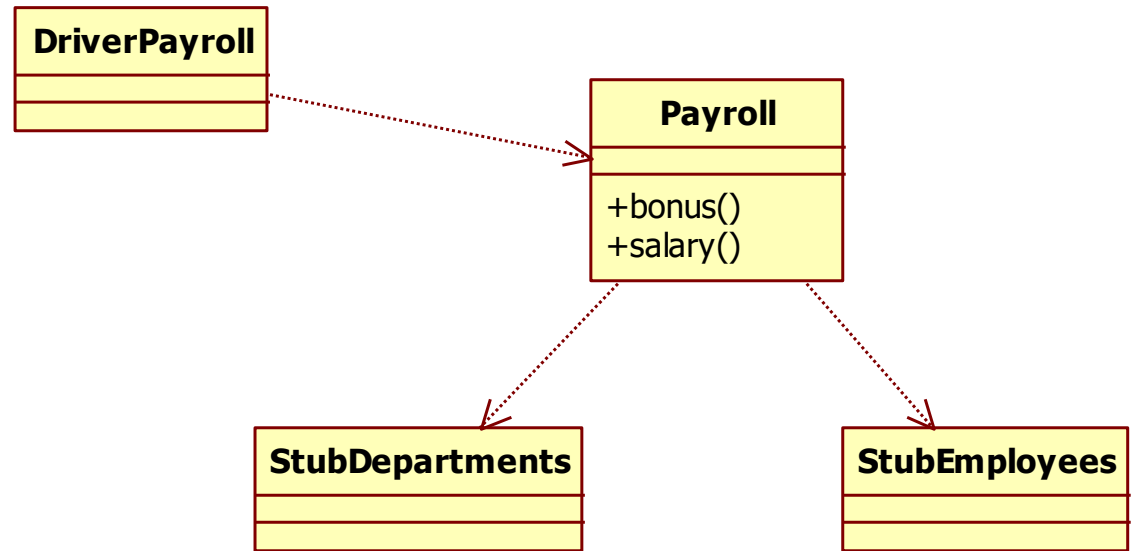
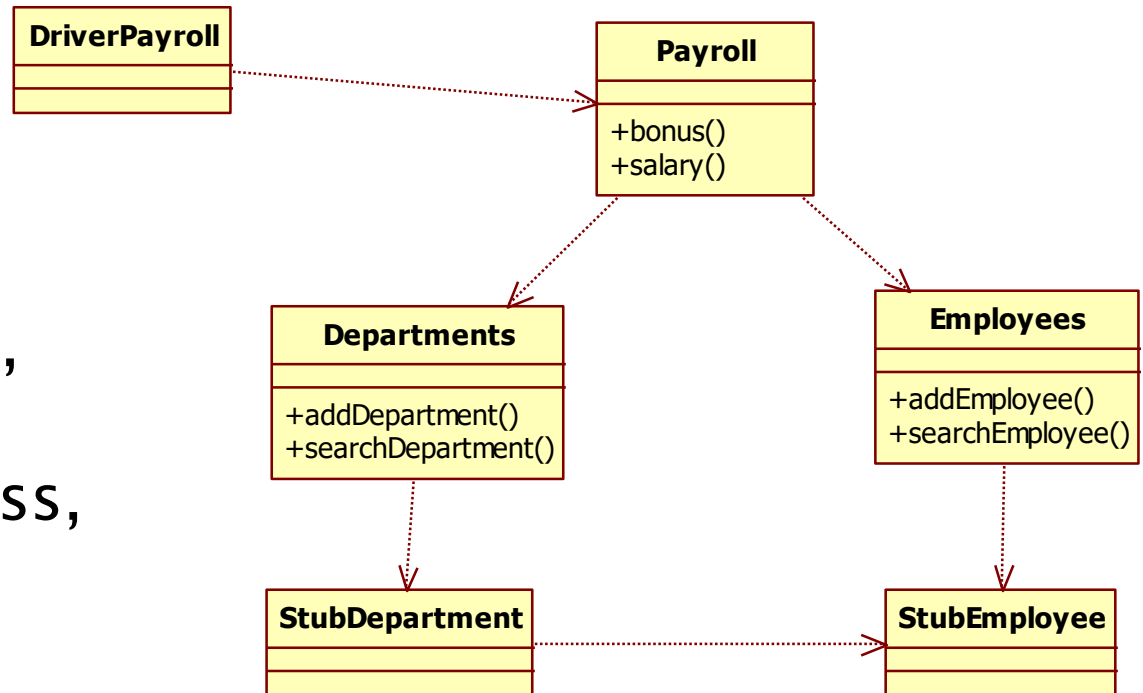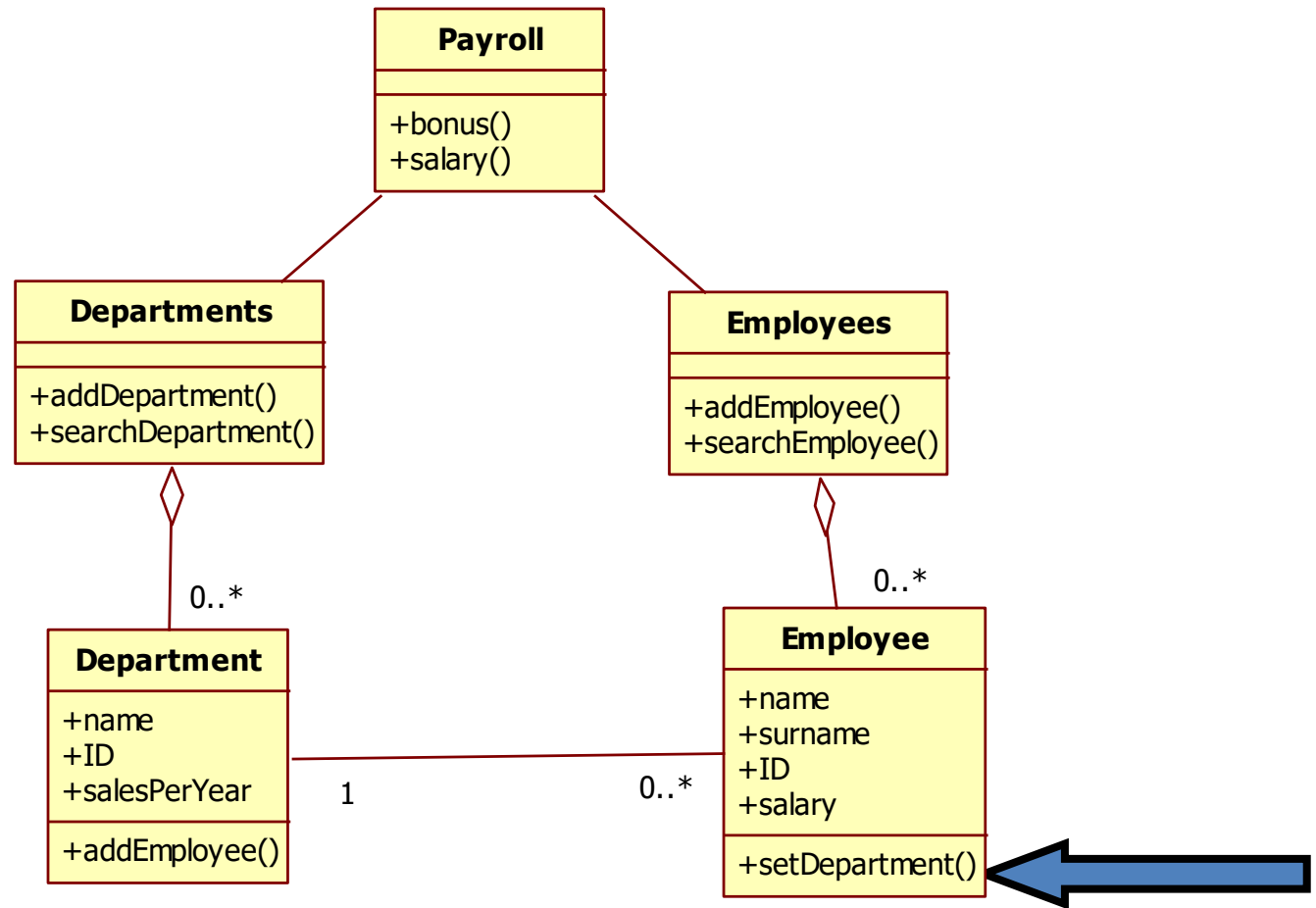# BU – 3

BU – 4

TD – 1

TD – 2

StubDepartment, StubEmployee probably useless, have similar complexity of Department and Employee

Class Diagram (2)

Dependencies (2)
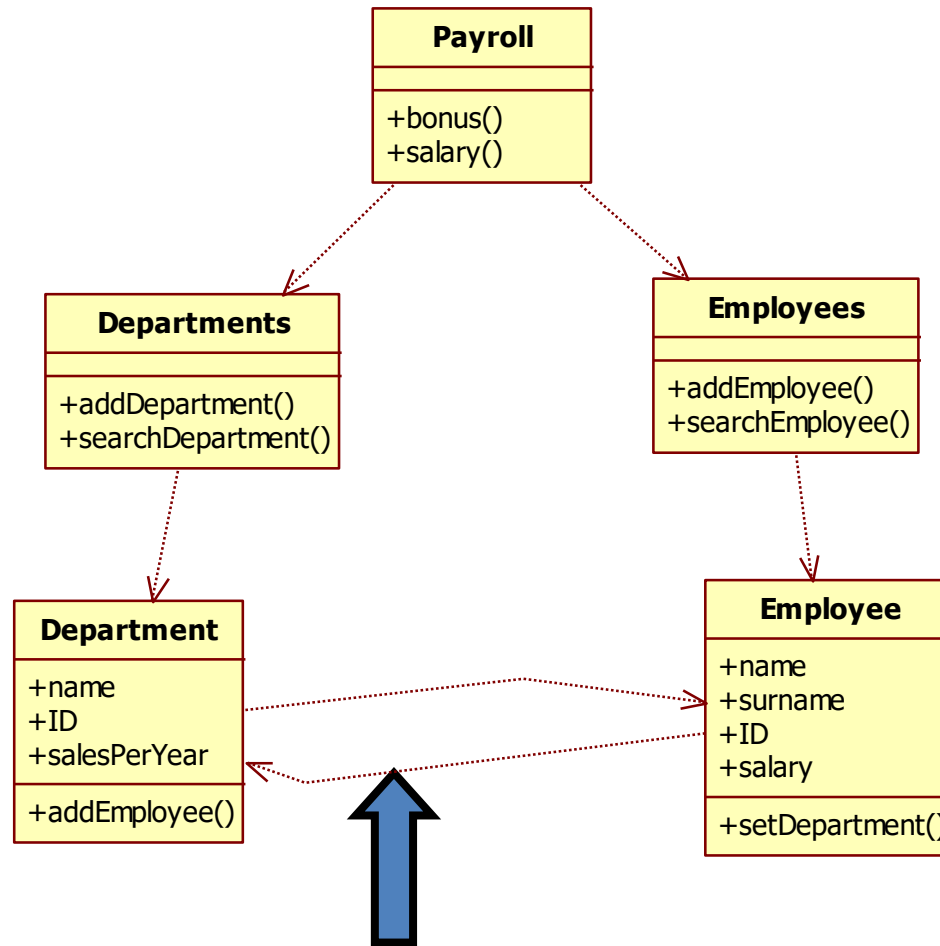
Integration

1. Consider classes with dependency loop as single class

   Not feasible if large/complex classes

2. Change design, split loop
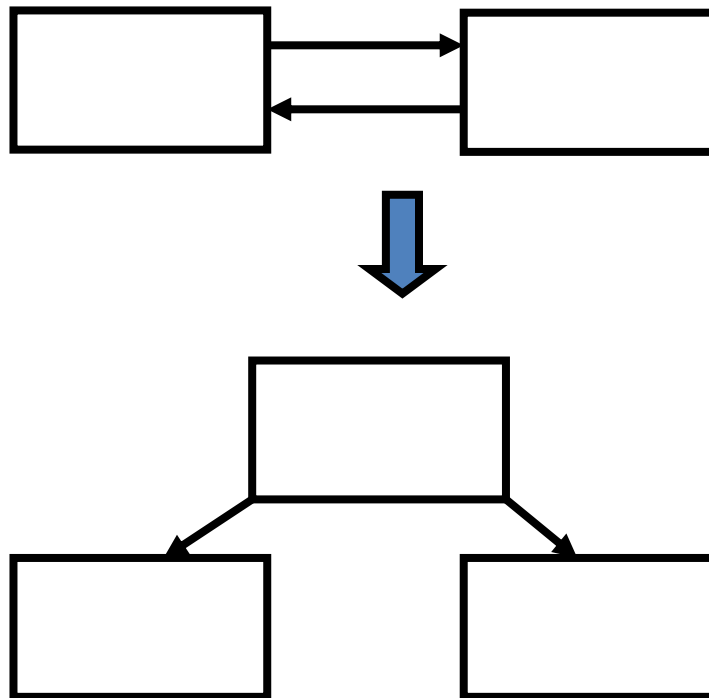
Case 1 – BU
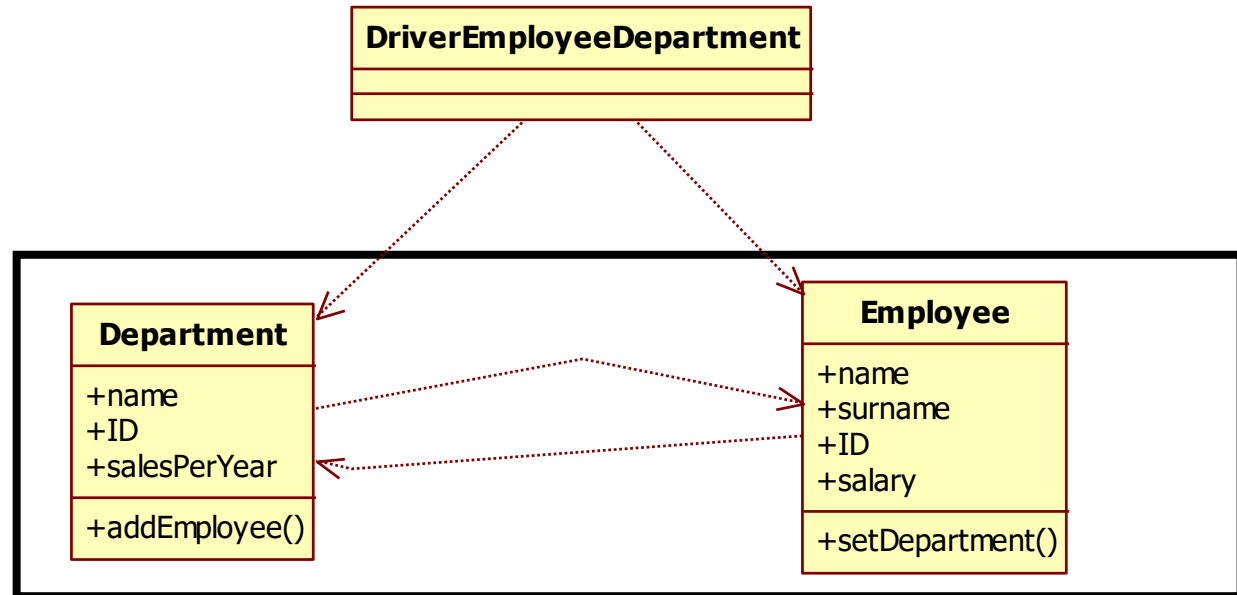
◆ Next steps as for BU case

Hw sw integration (embedded sw)

1. Test software units with stubs replacing hardware (actuators, sensors)

   ex: sensor temperature → sw stub

   development environment (ex MS Windows on Intel cpu)

2. Integrate software + hardware

   target environment (ex Android on ARM device)

SOftEng
http://softeng.polito.it

- Usually writing stubs for sensors / actuators implies defining a simulated / emulated model of the context of the sw application (as defined by context diagram and system design)

System test

Acceptance test

System test

Integration test

Unit test

System test

| Requirements engineering | → | Requirement document | → | VV requirements | → | Requirement document |

| Design . | | Design document | | VV design | | Design document |

| Implement unit | | Unit | | VV unit | | Unit |
| Implement unit | | Unit | | VV unit | | Unit |

| Integrate units | | System | | **VV system** | | System |

Project management
Configuration management
Quality assurance
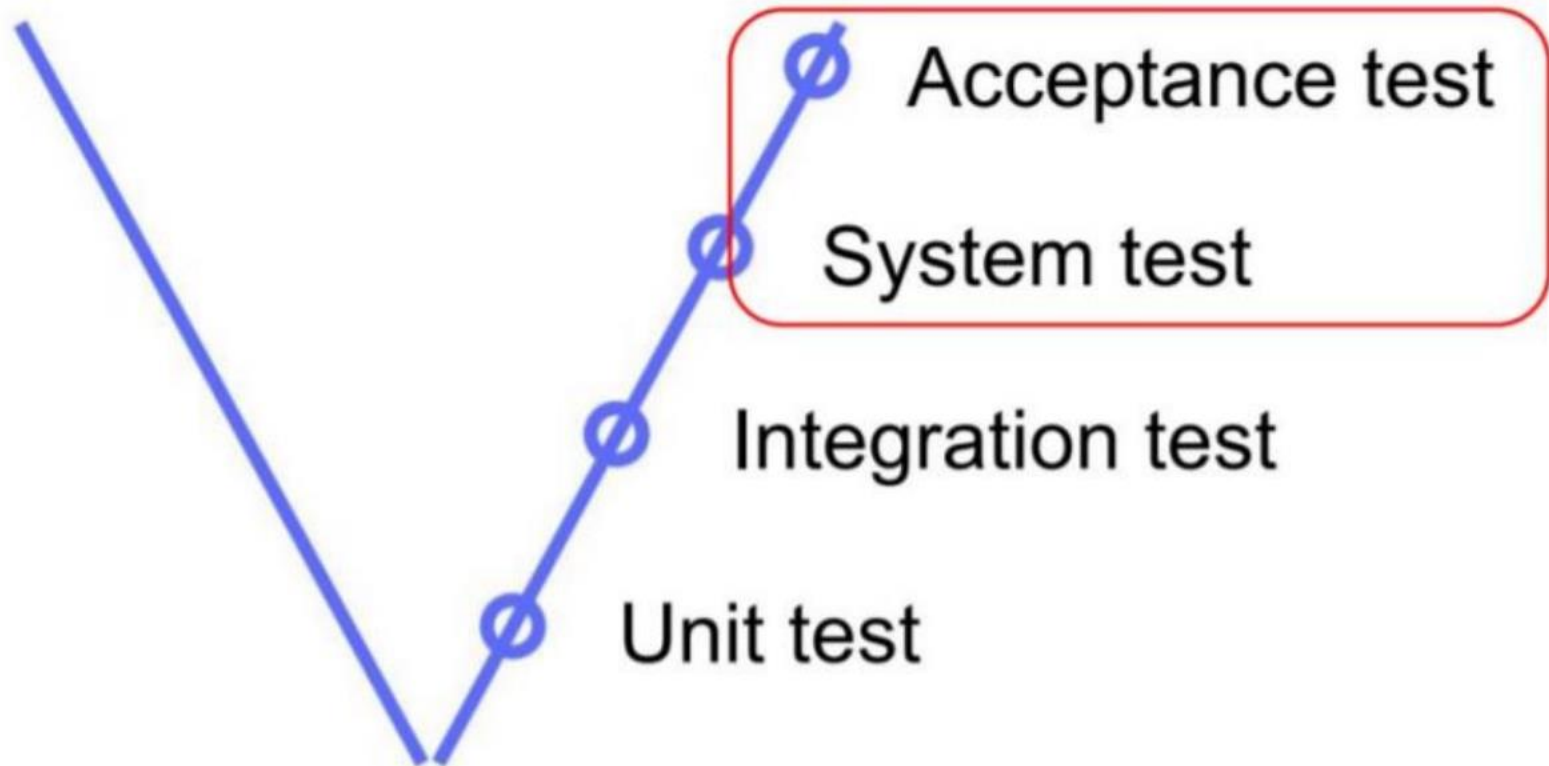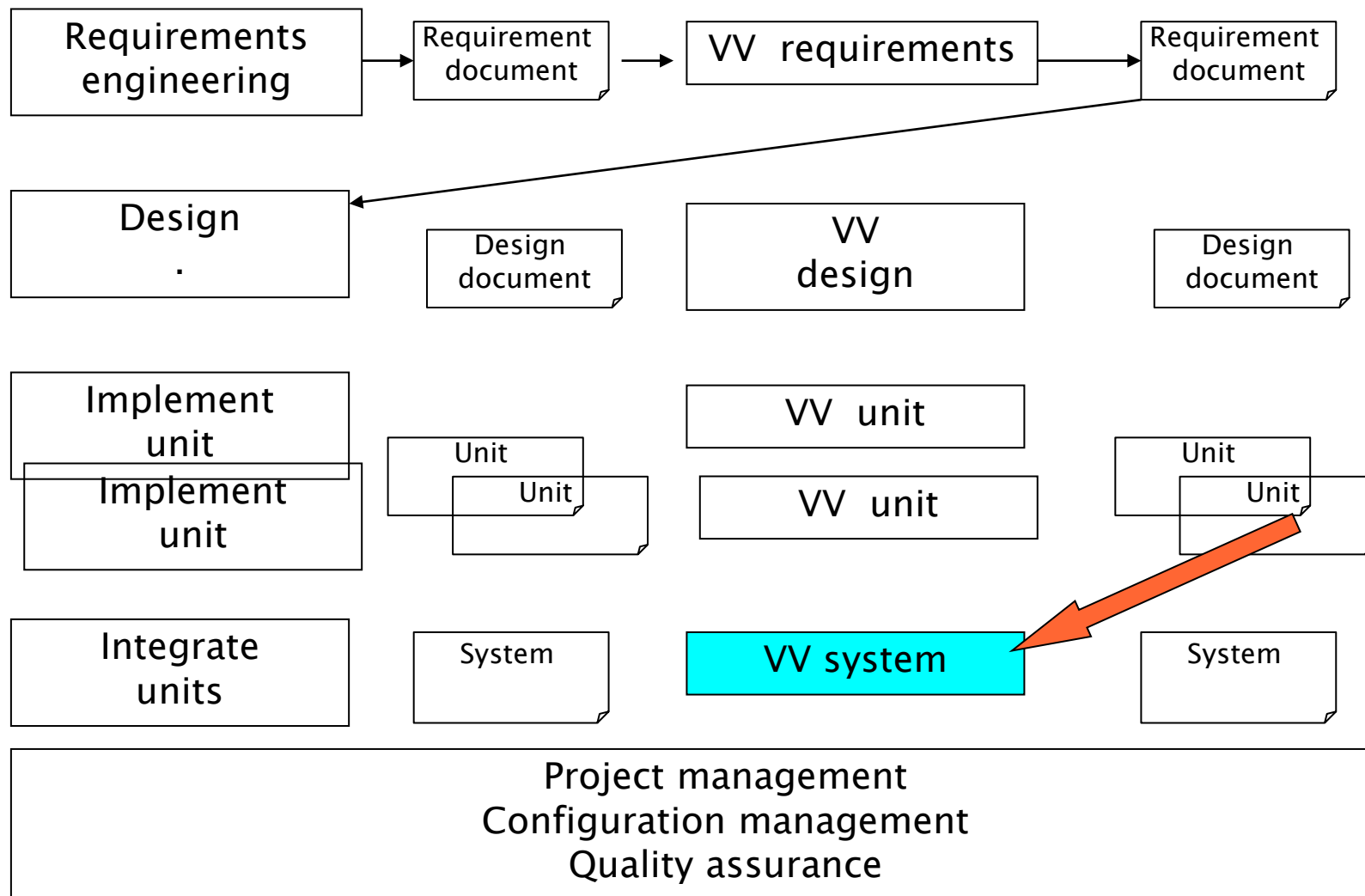
System test

- Test of <u>all</u> units composing an application
  - ◆ Unit:
    - – function (procedural languages)
    - – class and its methods (oo languages)

System test

- Focuses on
  - Functional properties (functional test)
    - Aka last stage of integration testing
  - Non functional properties (..ilities)
- Considers different platforms
  - Development
  - Production
- Considers different players
  - Developers / testers
  - End users

System test and platforms

The platform

- Environment where an application runs, defined by
  - ◆ Operating system
  - ◆ Database
  - ◆ Network
  - ◆ Memory
  - ◆ CPU
  - ◆ Libraries
  - ◆ Other applications installed
  - ◆ Other users
  - ◆ …

Platform and test

- An element (system, unit, ..) can be tested on
  - ◆ Target platform
    - – Where the element will run for day by day use
    - – **Cannot** be used for production
      - – Risk of corrupting data
      - – Availability
  - ◆ Production platform
    - – Where the element is produced
    - – Cannot be (in most cases) equal to the target platform

Platforms, examples

- **Embedded system**
  - ◆ ABS for car, heating control system, mobile phone
    - – Production platform is typically PC, external devices simulated/emulated
- **Information system**
  - ◆ Bank account management, student enrollment management
    - – Production platform is PC or workstation, database replicated in simplified form

What can change in system testing

- Who performs it
  - End user (beta testing)
  - Developer
- On what platform
  - Development
  - Target

System test and players

System test and players

- Developer, production platform
- Developer, target platform

- End user, developer platform
- End user, target platform

- **System test performed by end users is also called**
  - ◆ Acceptance testing
    - – Context: custom development, acquirer formally accepts the system, if test is positive project ends, payments can proceed
    - – Test cases: written by acquirer
  - ◆ Beta testing
    - – Context: mass market products, subset of end users tries the product before mass market release
    - – Test cases: informal

System test and test types

System test

- Test of functional properties
- Test of non functional properties

System test

- Test of functional properties (or functional requirements)

- Starting point is requirements document
  - Coverage of functional requirements
  - Coverage of use cases / scenarios

  - Consider usage profile (the most common, typical ways of using the system)
    - Cfr. Unit and integration test, goal is coverage, using all functions, all code.

Usage profiles

- Not all functions / use cases are equal
  - In many commercial products end users use 5-10% of all available functions

- Test more / test first the functions that are used more

Usage profile

Function / scenario	Frequency of usage
F1	10
F2	10
F3	80

User profiles

Advanced user profile	Function / scenario	Frequency of usage
	F1	20
	F2	20
	F3	60

Beginner user profile	Function / scenario	Frequency of usage
	F1	0
	F2	10
	F3	90

System test

- **Test of non functional requirements**
  - ◆ Non functional properties are usually system, (emerging) properties. In many cases only testable when system is available
    - – See efficiency, reliability

Non functional properties

- **Usability, reliability, portability, maintainability, efficiency (see ISO 9126)**
- **Configuration**: the commands and mechanisms to change the system
- **Recovery**: the capability of the system to react to catastrophic events
- **Stress**: reliability of the system under limit conditions
- **Security**: resilience to non authorized accesses

System test – variants

- Acceptance testing
  - Data and test cases provided by the customer, on target platform
- Beta-testing
  - Selected group of potential customers

Test, in summary

	Functional/ non functional	Who tests	Platform	Techniques
Unit test	Functional / structural	Developer or test group	Production	BB, WB
Integration test	Functional	Developer or test group	Production	Incremental TD or BU
System test	Functional + non functional	Developer or test group or user	Production, target	Requirement coverage Use cases coverage

Testing classification (2)

	Phase		
	Unit	Integration	System
Functional / black box	X	X	X
Structural / white box	X		
Reliability			X
Risks			X

Coverage

- Entities considered by at least one test case / total entities

- 'Entity' depends on type of test
  - Test cases
  - Requirement
  - Function
  - Structural element
    - Statement, decision, module

SOftEng
http://softeng.polito.it

Coverage

Object tested	Coverage	
Unit (class)	methods	Consider methods of the class At least one test case per method
	Black box, partitions	Consider a method Define partitions One test case per partition
	Black box, boundary	Define partitions One test case per boundary
	Black box, random	Generate randomly test cases

Coverage

Object tested	Coverage	
Unit (class)	White box	Consider a method Statement coverage Decision coverage Consider a loop in a method Loop coverage

Coverage

Object tested	Coverage	
Integration (some classes)	dependencies	At least one test case per dependency

Coverage

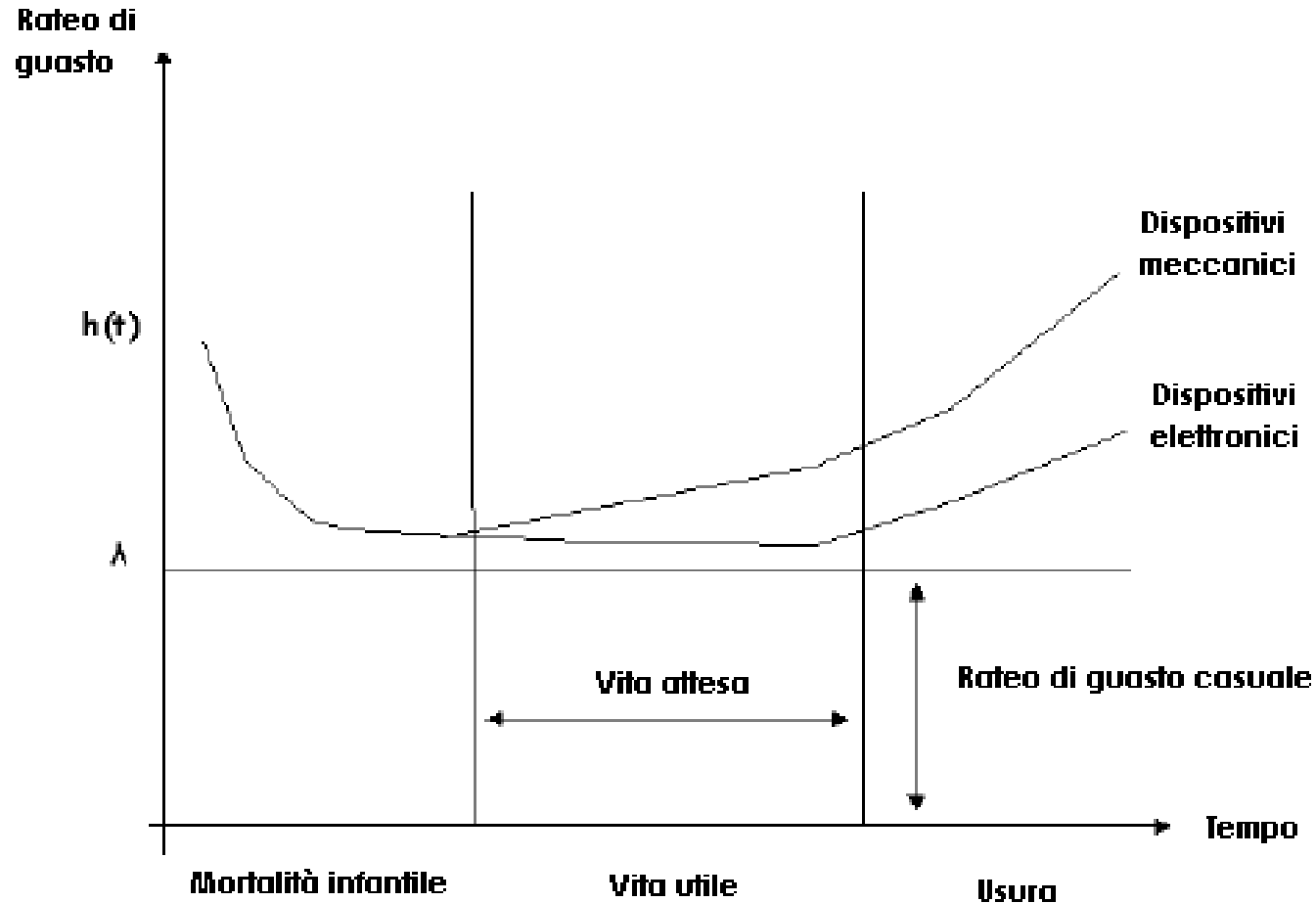| Object tested	Coverage	
System (all classes)	Functional requirements	At least one test case per requirement (may correspond to testing methods of 'main' class at end of integration testing)
	Scenarios	At least one test case per scenario
	Non functional requirements	At least one test case (Junit) per non functional requirement (efficiency)
		Meaningful tests for usability, portability, etc

Reliability testing

- Aims at providing estimate of reliability
  $$= P(\text{failure over period of time})$$

- Other measures of reliability:
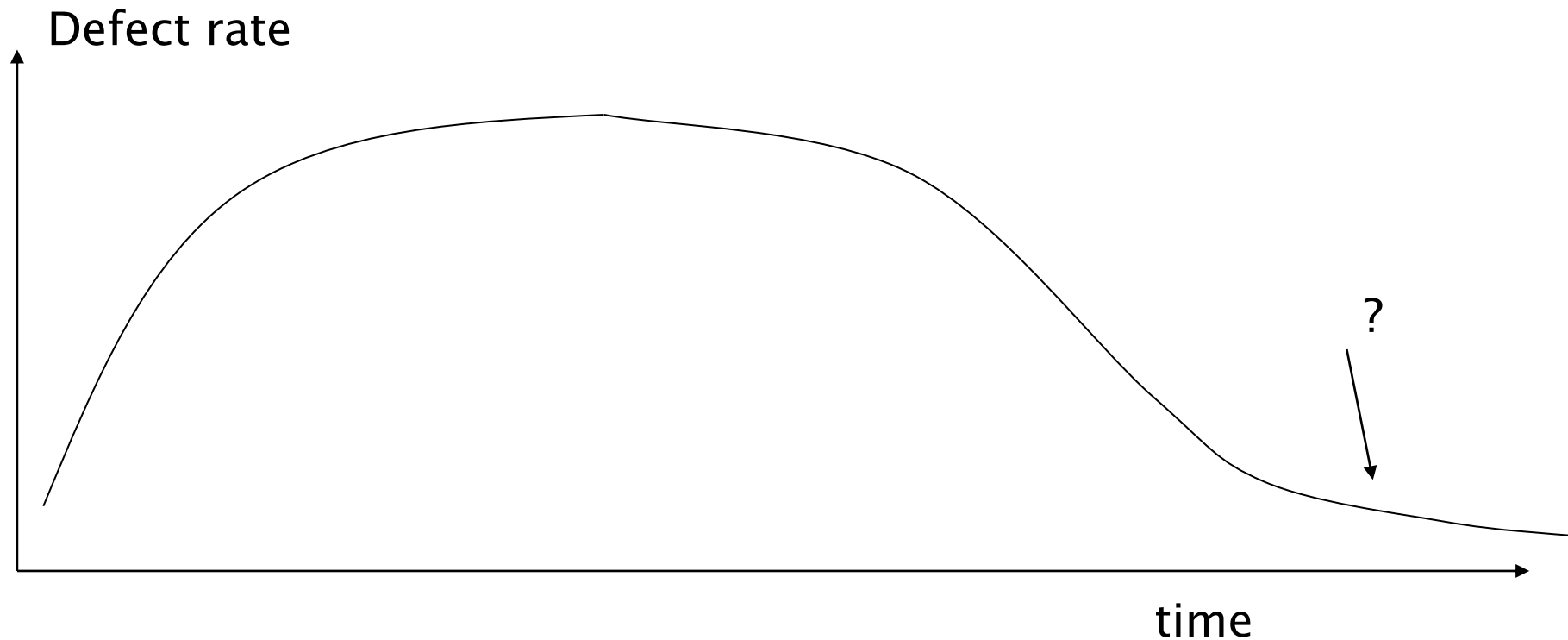 - defect rate = defect/time
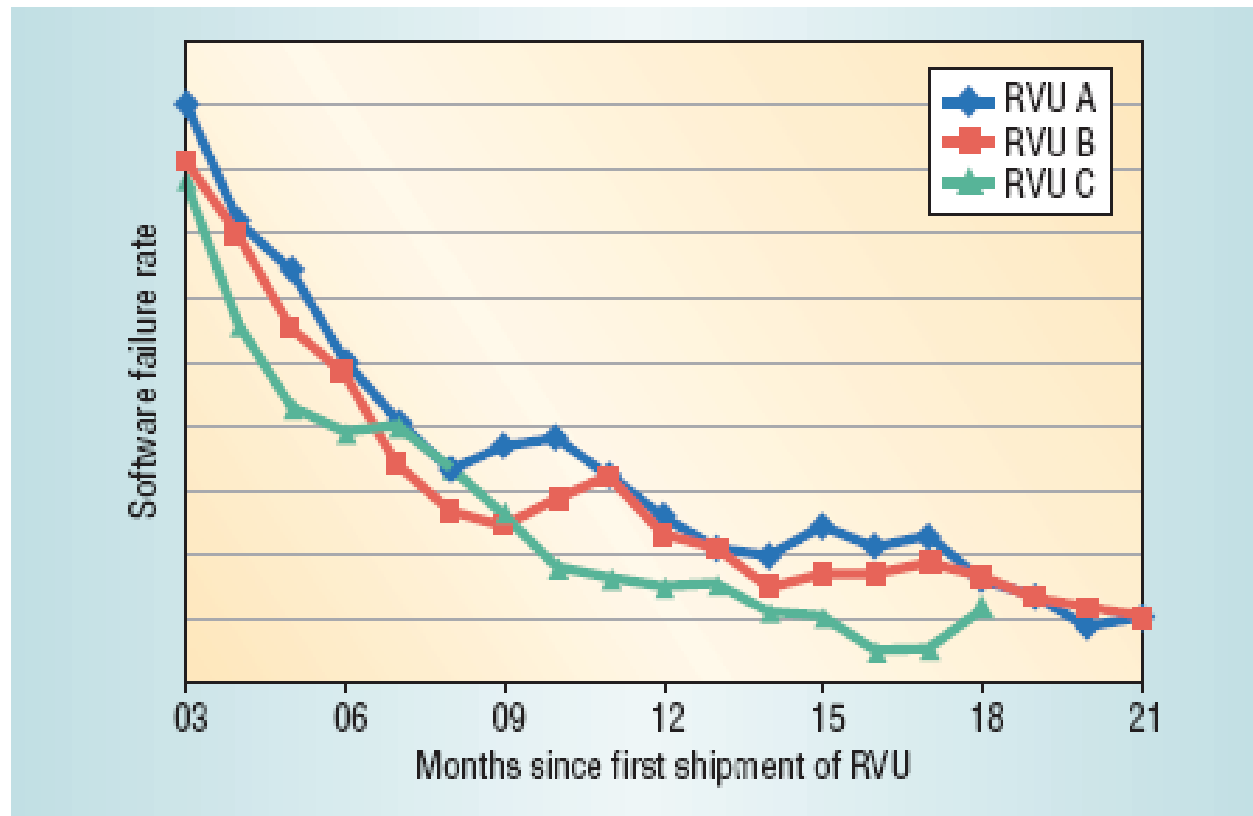 - MTBF = time between defects

Constraints

- Large number of test cases
- Independent
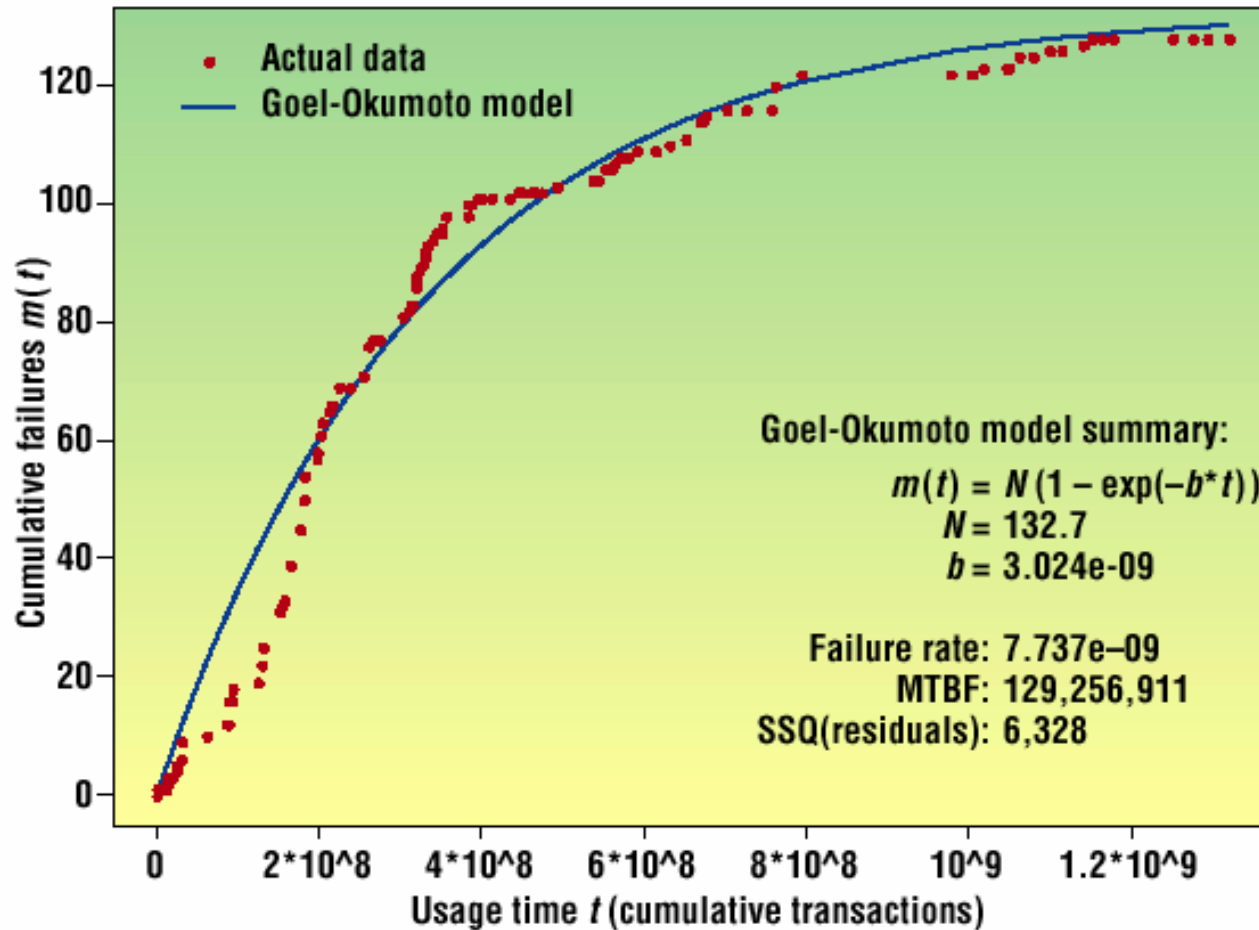- Defect fix does not introduce other defect

Hw reliability

Sw reliability

Defect rate

time

?

HP

IBM application

Goel-Okumoto model summary:
$$m(t) = N(1 - \exp(-b*t))$$
$$N = 132.7$$
$$b = 3.024\text{e-}09$$

Failure rate: 7.737e−09
MTBF: 129,256,911
SSQ(residuals): 6,328

Risk based testing / safety

- Identify risks
- Characterize risks: probability, effect
- Rank risks
- Handle risks

Ex. ABS (Anti Lock Brake)

- Risks:
  - No brake when push pedal
  - Brake when no push on pedal
  - Push pedal and wheels block

- Rank risks

- Develop tests to cover more risky situations

User profiles based testing

- Variant of risk based testing

- Identify user profiles, and usage profiles
- Rank them by frequency of usage
- Test more more used profiles

Ex. MS Word

- 5% of functions are used by 90% of users

- Focus test effort on those functions first

  - Ex allocate 90% of test effort on 5% of functions

Regression testing

- Regression testing
  - Tests previously defined are repeated after a change
  - To assure that the change has not introduced defects
    - Time0
      - Element (unit, system) in v0, test set t0 is defined and applied, all tests pass
    - Time1
      - Element is changed to v1
      - Test set t0 is re-applied, do all tests still pass?

Test, documentation and automation

The problem

- **Test cases should be not only documented**
  - So that they are not lost, and can be reapplied
    - Cfr. Test cases are just invented and applied
- **But also automated**
  - So that application of test cases is fast and error free
    - Cfr manual application of test cases

- Informal testing
  - Test cases are not documented
- Formal testing
  - Non operational (Word, excel..)
  - Operational (JUnit CUnit etc)

Representing test cases

- **Not operational**
  - i.e. Word document
- **Operational**
  - Wiki + translator to programming language
    - Fitnesse, FIT or SLIM
  - Scripting or Programming language
    - Java, Eclipse + JUnit
    - (similar for C, C#, http, perl,..)

Good test case design

- A good test case satisfies the following criteria:

    - Reasonable probability of catching an error
    - Does interesting things
    - Doesn't do unnecessary things
    - Neither too simple nor too complex
    - Makes failures obvious
    - Mutually Exclusive, Collectively Exhaustive

SOftEng
http://softeng.polito.it

Mutation testing

Mutation testing

- Are our test cases 'good'?

- Idea:
  1. write test cases
  2. inject errors in program (single small change)
  3. verify if test cases catch the errors injected

Mutation Testing

- Mutation Testing (a.k.a., Mutation analysis, Program mutation)
  - Introduced in early 1970s
  - Used in software/hardware

Terms

- *Mutant*: program with one change
- *Killable mutant*: non functionally equivalent. A test case can kill it
- *Equivalent mutant*: functionally equivalent to program. No test case can kill it.
- *Mutation score*:
  - ◆ property of a test suite (goal: 100%)
  - ◆ Killed non equivalent mutants / all non equivalent mutants

Mutations

- **Common mutations**
  - ◆ Delete a statement
  - ◆ Swap two statements
  - ◆ Replace arithmetic operation
  - ◆ Replace boolean relation
  - ◆ Replace a variable
  - ◆ Replace boolean subexpression with constant value

Mutation examples

public int sum(int a, int b)
{
return a + b;
}

public int sum(int a, int b)
{
return a + b--;
}

public int sum(int a, int b)
{
return a * b;
}

public int sum(int a, int b)
{
return a++ + b;
}

...

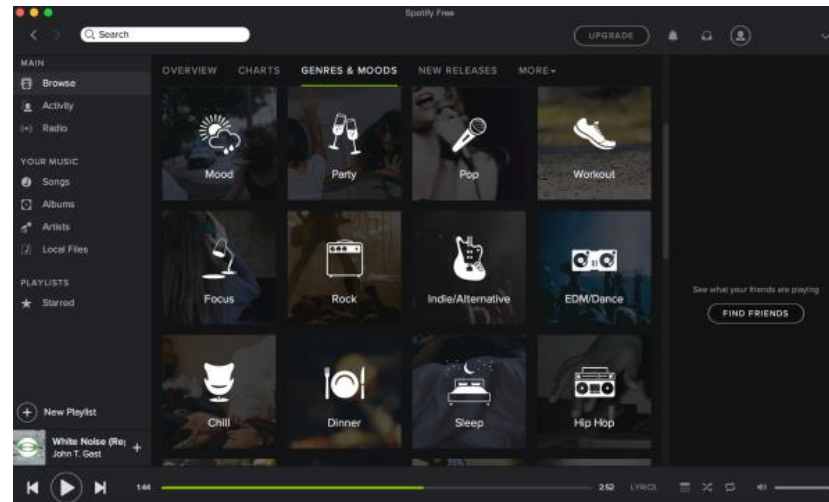# Mutations in Java

- PITest

Testing tools

- Unit / integration testing
  - JUnit, Mockito, ..
- GUI testing
- Table based
- Coverage
  - Eclemma, Clover, ..
- Profiling
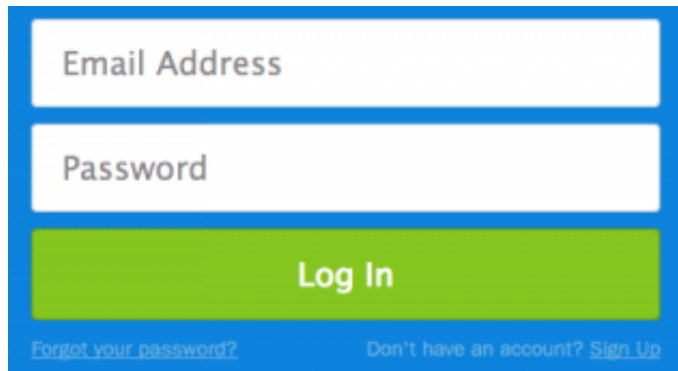
GUI Test tools

What is GUI Testing

GUI Software: any software provided with a GUI (Graphical User Interface).

GUI Testing is the practice of testing a software application through its user interface.

What is GUI Testing for

- **Functional tests:** black box tests exercising the basic functionalities of an application through interaction with the GUI, without knowing the source code.

Example:
Test that the log-in functionalities behave properly.

What is GUI Testing for

- **Look & Feel verification:** GUI should appear as defined in the requirements document, or in the mockups provided by the stakeholders.

- **Compatibility testing:** testing that the application is deployed and behaving properly on different devices, screens and layouts (device diversity).

Approaches to GUI Testing

Manual

Scripted

Capture & Replay

Model-based

Visual / Image Recognition

Manual GUI Testing

Manual execution of test scenarios (informal testing).

- Easy to setup, does not require tools;

- Error prone, hardly reproducible, expensive.

Scripted GUI Testing

Development of test scripts using dedicated scripting languages.

- Scripts can be automatically executed and used for regression testing;
- Test scripts can be difficult to write, and hard to maintain during the evolution of the software.

Scripted GUI Testing

Examples:

Selenium WebDriver
(web applications),

Appium
Espresso (mobile).

```java
@Test
public void testPurchase() throws Exception {
    // open the page and login
    dynaTrace.setTestStepName("doLogin");
    EasyTravelHomePage homePage = EasyTravelHomePage.open(driver);
    EasyTravelLoginDialog loginDialog = homePage.loginDialog();
    loginDialog.login("hainer", "hainer");

    // search for New York and verify that there are results
    dynaTrace.setTestStepName("doSearch");
    EasyTravelSearchResult result = homePage.search("New York");
    int resultCount = result.getSearchResultCount();
    if(resultCount == 0)
        throw new Exception("No Search Results for New York");

    // click on a random link and click on buy
    dynaTrace.setTestStepName("selectAndBuy");
    int randomIndex = new java.util.Random().nextInt(resultCount);
    result.clickOnByLink(randomIndex);

    // now purchase the flight
    result.completeBuy("4111111111111111");

    // time to logout
    dynaTrace.setTestStepName("doLogout");
    homePage.logout();
}
```

Capture & Replay

User inputs are given once to the user interface (CAPTURE), and then codified into a repeatable script (REPLAY).

- Faster and easier to obtain test scripts with respect to pure scripted techniques;

- Very fragile to the evolution of the user interface;

- Scripts must be enriched manually to perform complex operations.

Capture & Replay

Examples:

SilkTest, HP UFT

(desktop).
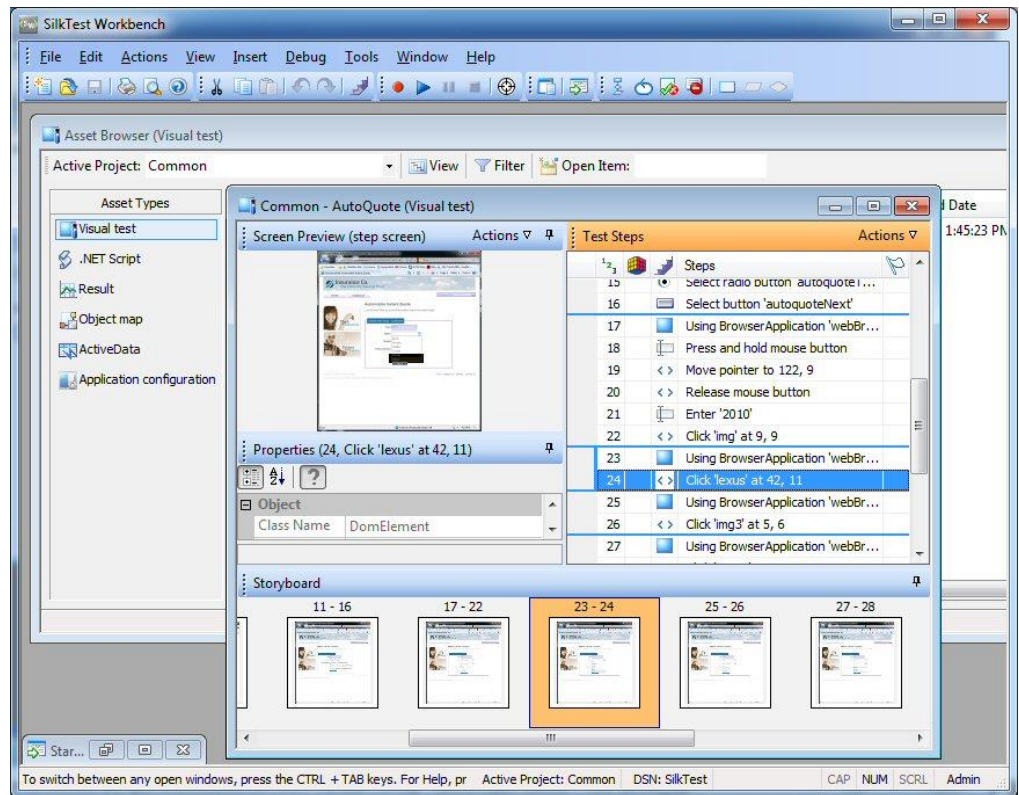
Model-Based GUI Testing

Test cases are obtained based on models of the user interface (e.g., oriented graphs or finite state machines).
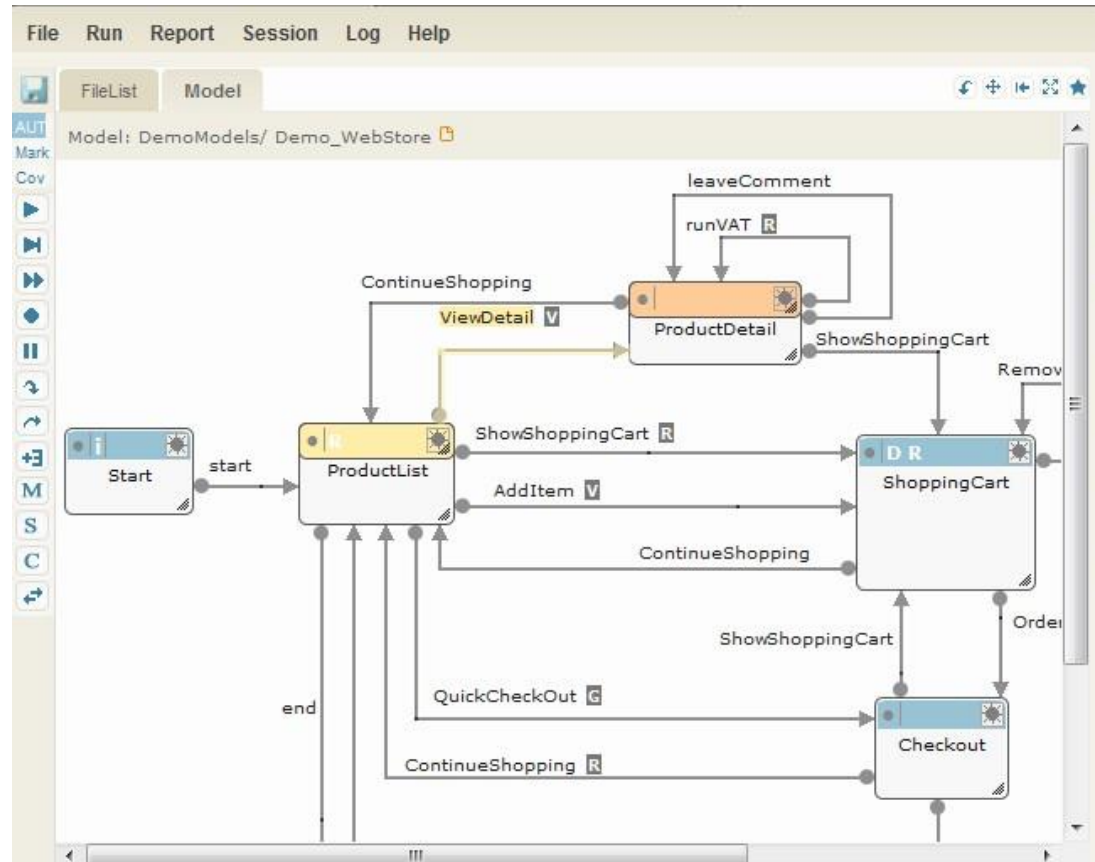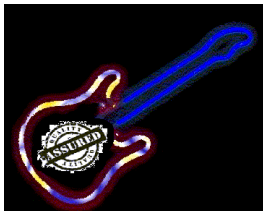
- Allow automated execution and generation of use cases;

- Very high coverage of use cases and functionalities can be obtainable once a model is available;

- Manual effort required in defining and tailoring the GUI model.

Model-Based Testing

Tools:

TestOptimal (web),

MobiGUITAR (mobile).

Visual Testing

Image recognition techniques are used to identify elements of the user interface to interact with.

- Easy definition of test cases with no need of technical knowledge – only screen captures are needed;
- Can be applied seamlessly to any kind of software provided with an (emulated) user interface;
- Very high fragility to even minor changes in the GUI;
- Difficult in-depth testing of application functionalities.

Visual Testing

Examples:
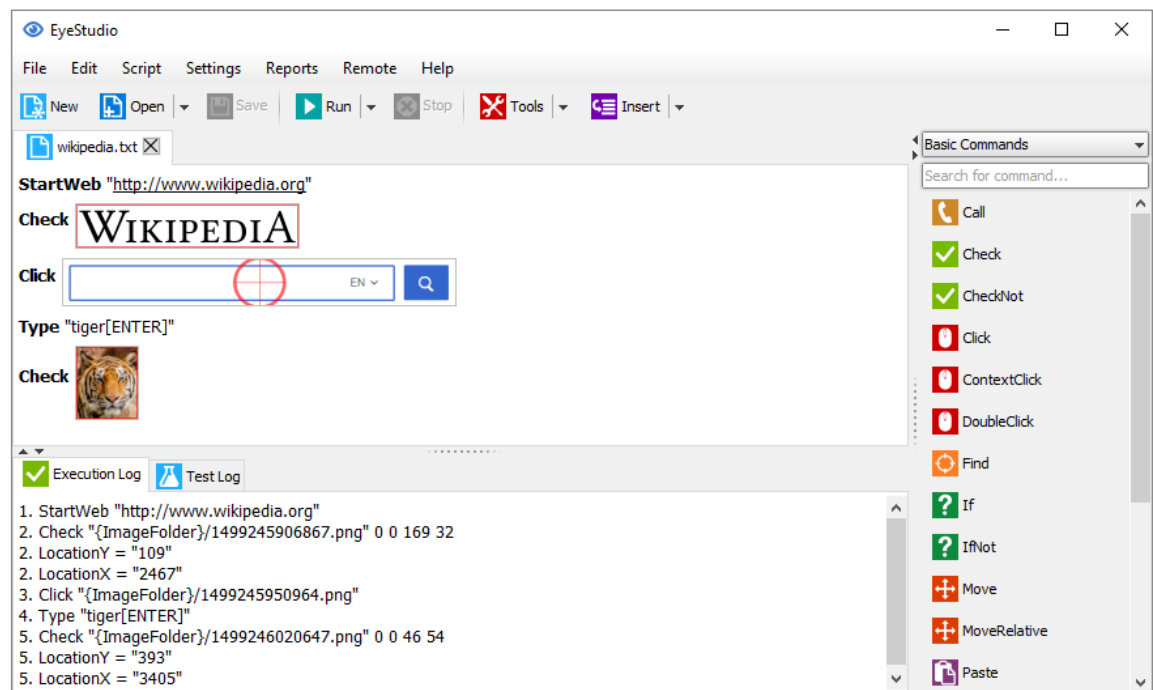
Sikuli, EyeAutomate

Table based

Table based testing

- Test cases are written as tables on a wiki, and linked to the application to be tested
- Pro:
  - Allows end users to write tests (especially acceptance tests, black box tests)
  - Independent of GUI
  - Allows automation
- Cons
  - Requires fixtures

inputs

output

FIT Framework for Integrated Test

- Open source implementation of table based testing
  - User specifies tests in HTML tables
  - Developers defines *fixtures* to parse tables and execute tests
  - Fit compares tables and actual results

FITnesse

- Standalone wiki that's hooked to FIT
- Allows group to easily edit test files without worrying about ensuring the correct versions propagate out to all locations
- http://fitnesse.org

Coverage

- Show graphically and numerically coverage (statements, branches, conditions) on source code

- Ex., Clover, Jcoverage, Cobertura, Eclemma, JaCoCo

Profilers

- Trace time spent per function, given specific test execution
  - Performance test

Testing – certifications

- ISTQB

  - www.istqb.org
  - Int software testing qualifications board
  - Delegations in most countries
  - In italy ita-istqb

ISTQB

- Publishes Syllabus
  - Available free
    - Foundation, advanced, expert levels
    - Core, agile, specialist tracks

- Provides Certifications, via exam
  - See foundation level

Static analysis

- Static
  - inspections
  - source code analysis ⬅
- Dynamic
  - testing

Static analysis techniques

- Compilation static analysis
- Control flow analysis
- Data flow analysis
- Symbolic execution
- Reverse documentation of design

Compilation analysis

- Compilers analyze the code checking for
  - Syntax correctness
  - Types correctness
  - Semantic correctness
- The errors detected by a compiler strongly depend on the language
  - Loose vs. strongly typed languages
  - Static vs. dynamic visibility

MISRA-C

- MISRA: Motor Industry Software Reliability Association

- Issues Misra-C, guidelines for C programs
  - Issue1, 1998
    - 127 rules, 93 compulsory
  - Issue2, 2004
    - 141 rules, 121 compulsory

Rules, examples

5 Use only characters in the source character set. This excludes the characters $ and @, among others.

22 Declarations of identifiers denoting objects should have the narrowest block scope unless a wider scope is necessary.

34 The operands of the && and || operators shall be enclosed in parenthesis unless they are single identifiers.

67 Identifiers modified within the increment expression of a loop header shall not be modified inside the block controlled by that loop header.

103 Relational operators shall not be applied to objects of pointer type except where both operands are of the same type and both point into the same object.

SOftEng
http://softeng.polito.it

Rule 5

signed char dollar = '$';

- not accepted

signed char esc_m = '\m';

- not accepted (what would be the associated behaviour to this escape sequence?)

Rule 34

if ((var++) || (num == 11)){...} /* OK */
if (var++ || num == 11){...} /* NOT OK */

if ((vect[num]==4) && (num == 11)){...} /* OK */
if (vect[num] == 4 && (num == 11)){...} /* NOT OK */

if ((structure.field != 0) && (num < 11)){...} /* OK */

Rule 67

```
for (int i = 0; i< max; i++){


  i=i+1;  // NO
}
```

Misra static analyzers

- Parse source code and check if rules are violated
  - QA-C by Programming Research, is a full featured MISRA C1 and C2 validator.
  - Testbed by LDRA, offers a static and dynamic analysis.
  - PC-Lint by Gimpel, is one of the fastest and least expensive validtors.
  - DAC by Ristan-CASE, provides a reverse engineering, documentation and code analyzer.

SOftEng
http://softeng.polito.it

Bad Smells (Fowler)

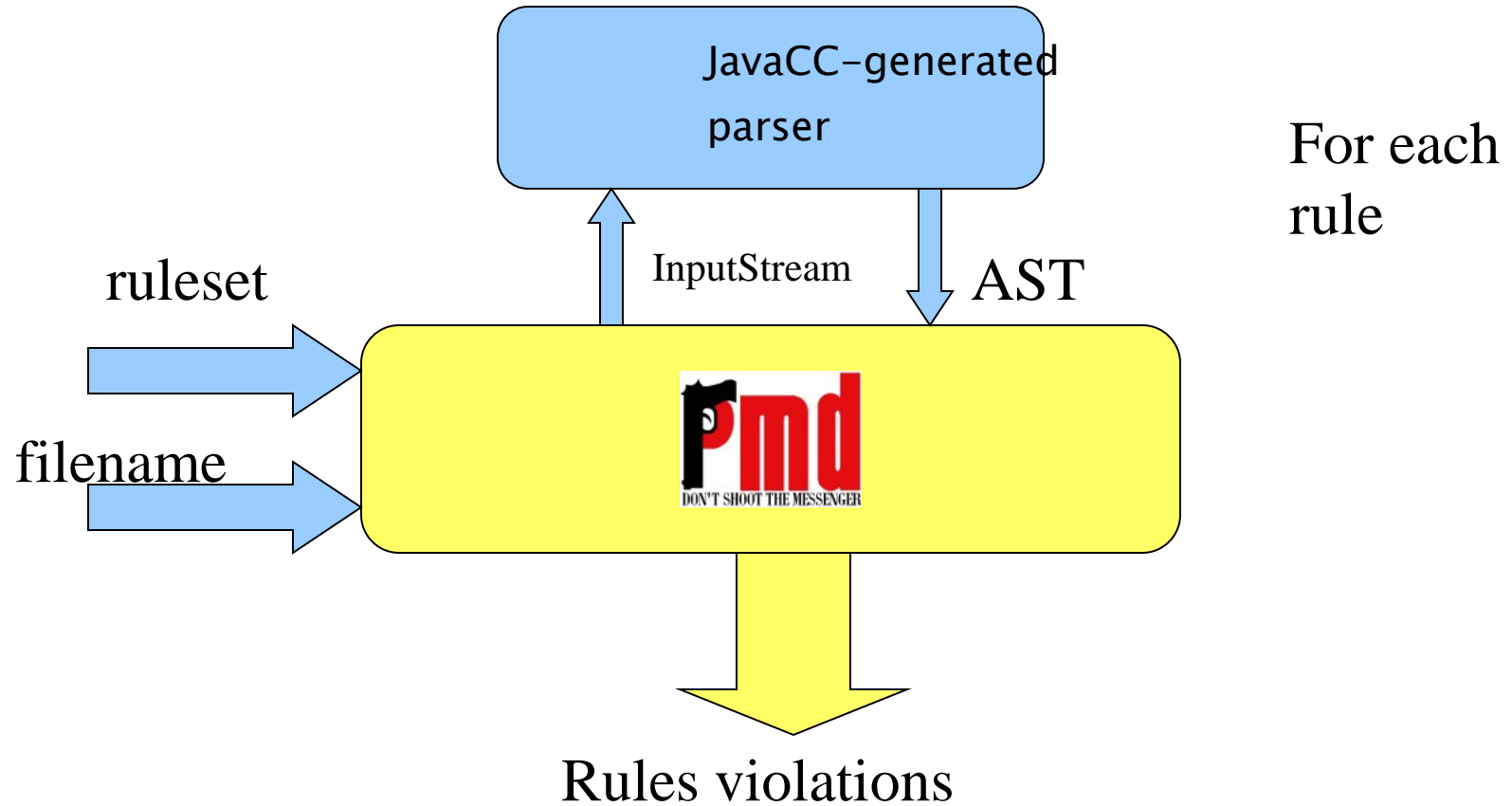- Fowler et al., Refactoring, Improving quality of existing code

Bad smells

- Duplicated code
- Long method
- Large class
- Long parameter list
- Divergent change
- Shotgun surgery
- Feature envy
- Data clumps
- Primitive obsession
- Switch statements
- Parallel inheritance hierarchies
- Lazy class
- Speculative generality
- Temporary field
- Message chain
- Middle man
- Inappropriate intimacy
- Alternative classes with different interfaces
- Incomplete Library class
- Data class
- Refused bequest
- Comments

Java analyzers

- PMD
  - pmd.sourceforge.net
- Findbug
  - findbug.sourceforge.net

PMD

JavaCC-generated parser

For each rule

ruleset

InputStream

AST

filename

pmd
DON'T SHOOT THE MESSENGER

Rules violations

PMD, rules

- empty try/catch/finally/switch statements...
- Dead code – unused local variables, parameters and private methods...
- Suboptimal code – wasteful String/StringBuffer usage...
- Overcomplicated expressions – unnecessary if statements, for loops that could be while loops...
- Duplicate code – copied/pasted code means copied/pasted bugs

Findbug, rules

- ◆ **Correctness bug:**
  - – Probable bug – an apparent coding mistake resulting in code that was probably not what the developer intended.
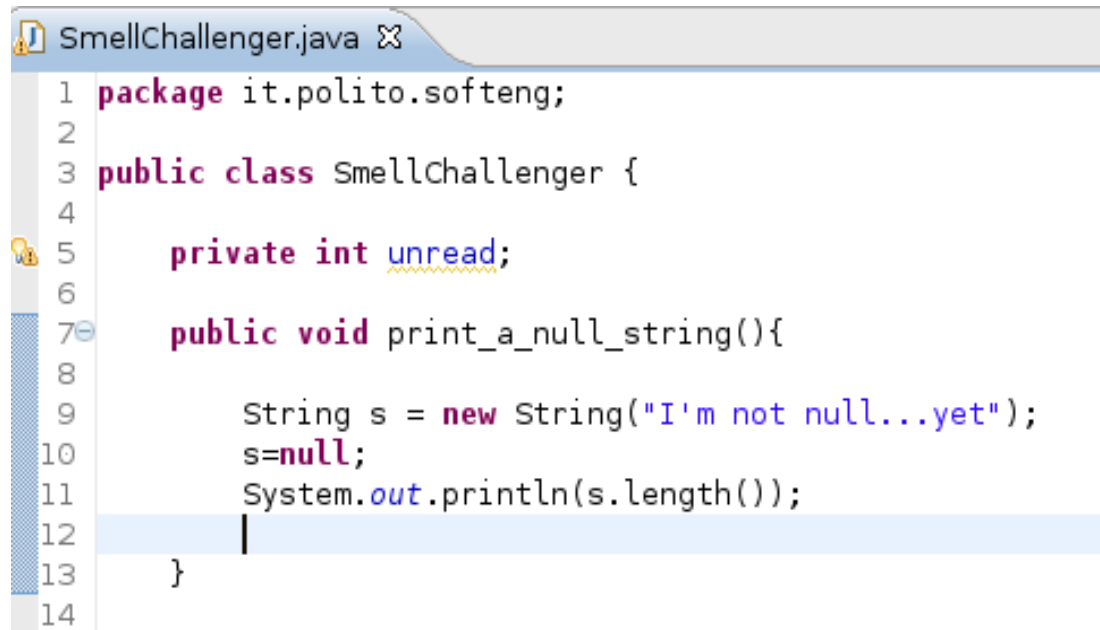
- ◆ **Bad Practice**
  - – Violations of recommended and essential coding practice. Examples include hash code and equals problems, cloneable idiom, dropped exceptions, serializable problems, and misuse of finalize.

- ◆ **Dodgy**
  - – Code that is confusing, anomalous, or written in a way that leads itself to errors. Examples include dead local stores, switch fall through, unconfirmed casts, and redundant null check of value known to be null.

PMD and FIND BUGS: let's try!

- Let's run the plugins with the following piece of code

```java
package it.polito.softeng;

public class SmellChallenger {

    private int unread;

    public void print_a_null_string(){

        String s = new String("I'm not null...yet");
        s=null;
        System.out.println(s.length());

    }
}
```

Results comparison

Avoid unused private fields → `private int unread;` ← Unused field

Methods names should not contain underscores →
```
public void
print_a_null_string(){
```

Avoid instantiating String objects: it's usually unnecessary

Avoid variables with short names

```
String s = new String
("I'm not null...yet");
```

Dead store to variable s

Invokes inefficient new String(String) constructor

Results comparison

Anomaly: A recently defined variable is redefined. This is ominous but don't have to be a bug.

Assigning an Object to null is a code smell. Consider refactoring.

```
s=null;

System.out.println
(s.length());

}
```

System.(out|err).print is used, consider using a logger.

Null point deference of s

Load of known null value

Data flow analysis

- Analyzes the values of variables during execution to find out anomalies

- Looks like dynamic but some information can be collected statically

- Three operations on variables
  - Definition: – write– a new value is assigned
  - Use: – read– the value of the variable is read
  - Nullification: the variable has no significant value

Data flow analysis

- Correct sequences
- D U
  - The use of a variable must be always preceded by a definition of the same variable
- Suspect (forbidden) sequences
- D D
- N U
  - A use of a variable not preceded by a definition corresponds to the use of an undefined value

Data flow analysis

- Tools recover the sequence and recognize suspect ones

	x1	x2	x
void swap(float*x1, float* x2){	D	D	–
int float x;	–	–	N
*x2 = x;	–	D	U
*x2 = *x1;	U	D	–
*x1 = x;	D	–	U
}			

x = *x2;

Symbolic execution

- The program is executed with symbolic values instead of actual values

- Output variables are expressed as symbolic formulas of input variables

- Symbolic execution may be extremely complex even for simple programs

Symbolic execution

1 integer product (int x, int y, int z){

2   int tmp1, tmp2;

3   tmp1 = x*y;

4   tmp2 = y*z;

5   return tmp1 * tmp2 / y;

6 }

Stmt	x	y	z	tmp1	tmp2	product
2	X	Y	Z	?	?	?
3	X	Y	Z	X*Y	?	?
4	X	Y	Z	X*Y	Y*Z	?
5	X	Y	Z	X*Y	Y*Z	X*Y*Y*Z / Y

Reverse documentation of design / code

- Over time the code changes and may or may not correspond to the initial design

- How to reconstruct the actual, current design?

Code base → [Reverse design tool] → Design

Tools

- Structure 101