Politecnico
di Milano

Verification

(Note: verification used in a very general sense, including validation)

Verification&validation

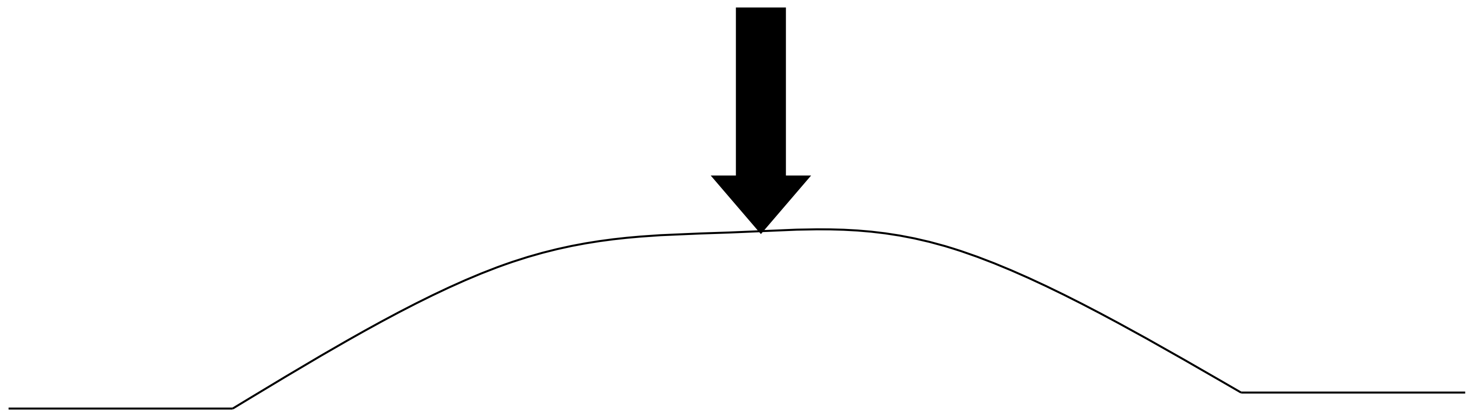
- Verification
  - *did we build the program right?*
- Validation
  - *did we build the right program?*

Why, what, where?

- Zero defect software practically impossible to achieve
- Careful and continuous verification needed
- Everything must be verified (spec. documents, design documents, test data, …)
  - even the verification must be verified!
- Verification along all development process, not just at the end

Verification in engineering

- Example of bridge design
- One test assures infinite correct situations



Verification in software engineering

- Programs do not display a “continuous” behavior
- Verifying the function in one point does not tell us anything about other points
  - Example 1

```
. . .
a =  … / (x  +20) ...
. . .
```

Any value of x is ok, except for x = -20!

V&V Terminology

Term	Description
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users
System error	Erroneous system behaviour where the behaviour of the system does not conform to its specification.
System fault	An incorrect system state i.e. a system state that is unexpected by the designers of the system.
Human error or mistake	Human behaviour that results in the introduction of faults into a system.

Faults, errors and failures

- Failures are usually a result of system errors that are derived from faults in the system
- However, faults do not necessarily result in system errors
  - The faulty system state may be transient and 'corrected' before an error arises
- Errors do not necessarily lead to system failures
  - The error can be corrected by built-in error detection and recovery
  - The failure can be protected against by built-in protection facilities. These may, for example, protect system resources from system errors

Difficulties in V&V (1)

- Checking of some qualities does not have a binary (yes/no) outcome
- Many properties are subjective
- Some are even implicitly stated

Difficulties in V&V (2)

- Qualities are not clearly stated or
- Are not reasonable (is 100% reliability a reasonable goal?)
- The relative importance of qualities and their relationships with other project objectives needs to be identified

Difficulties in V&V (3)

- It is almost impossible to develop error free software
- New approaches and technologies may introduce new errors and problems
  - E.g., transition to new language or development environment
- Challenge
  - find the right blend of verification and validation approaches for each specific software

When do v&v start?

- As soon as we decide to develop a product
- During feasibility study we consider
  - functionality, required qualities and their impact on costs

- Quality manager participates in the feasibility study
  - focuses on how to assess and control quality during development
  - influences the definition of the preliminary architecture of the system in order to ensure that it can be tested and analyzed more easily

An example

- The development of a web application
  - If the application is decomposed in three layers (UI, business and data layers) the quality assurance team can be structured accordingly
    - The human interface group is responsible for usability
    - The key quality people can be involved in checking the kernel of critical functions within the business and data layers
    - Less experienced persons can take care of the other parts
  - Some preliminary decisions about the quality assurance approach can be taken. For instance:
    - A first prototype will not go through a complete acceptance test but will be used to validate requirements and design
    - The acceptance test for the first release will be focused on usability feedback from a subset of users and will check typical security problems
    - The acceptance test for the second release will include a check of all functionalities and reliability measures

What v&v technique should be applied?

- The choice depends on quality, cost, schedule, resource constraints
- Combination of different techniques because
  - Each technique may be effective for different classes of faults
  - May be applicable at different points in a project
  - May have different purposes
  - May have different tradeoffs in cost and assurance

An example

- While developing our web application
  - A semi-formal notation is used for requirement description and system design
    - The quality manager decides to use *inspection* to check these documents
      - performed by single persons or small groups for design documents
      - performed by a larger group according to well formalized procedures for req. descriptions and specs.
  - For unit test each developer is required to produce functional test cases together with the code
    - If less than 80% code statements are executed by these test cases, other tests are identified by using a structural approach (the company has a tool to evaluate test coverage)
  - Integration and system test cases are generated by the quality team. <u>Scaffolding</u> and <u>oracles</u> are part of the system architecture

we will see these

How can we assess the readiness of a product?

- Finding all faults is nearly impossible
- Analysis and testing cannot go on forever
- … but the product should be delivered when it meets the functionality and the quality required by the market

- Examples of important measures for dependability
  - Availability: QoS in terms of running versus down time
  - Mean Time Between Failure (MTBF): QoS in terms of the length of time interval during which the service is available
  - Reliability: a fraction of all attempted system operations that completed successfully

An example

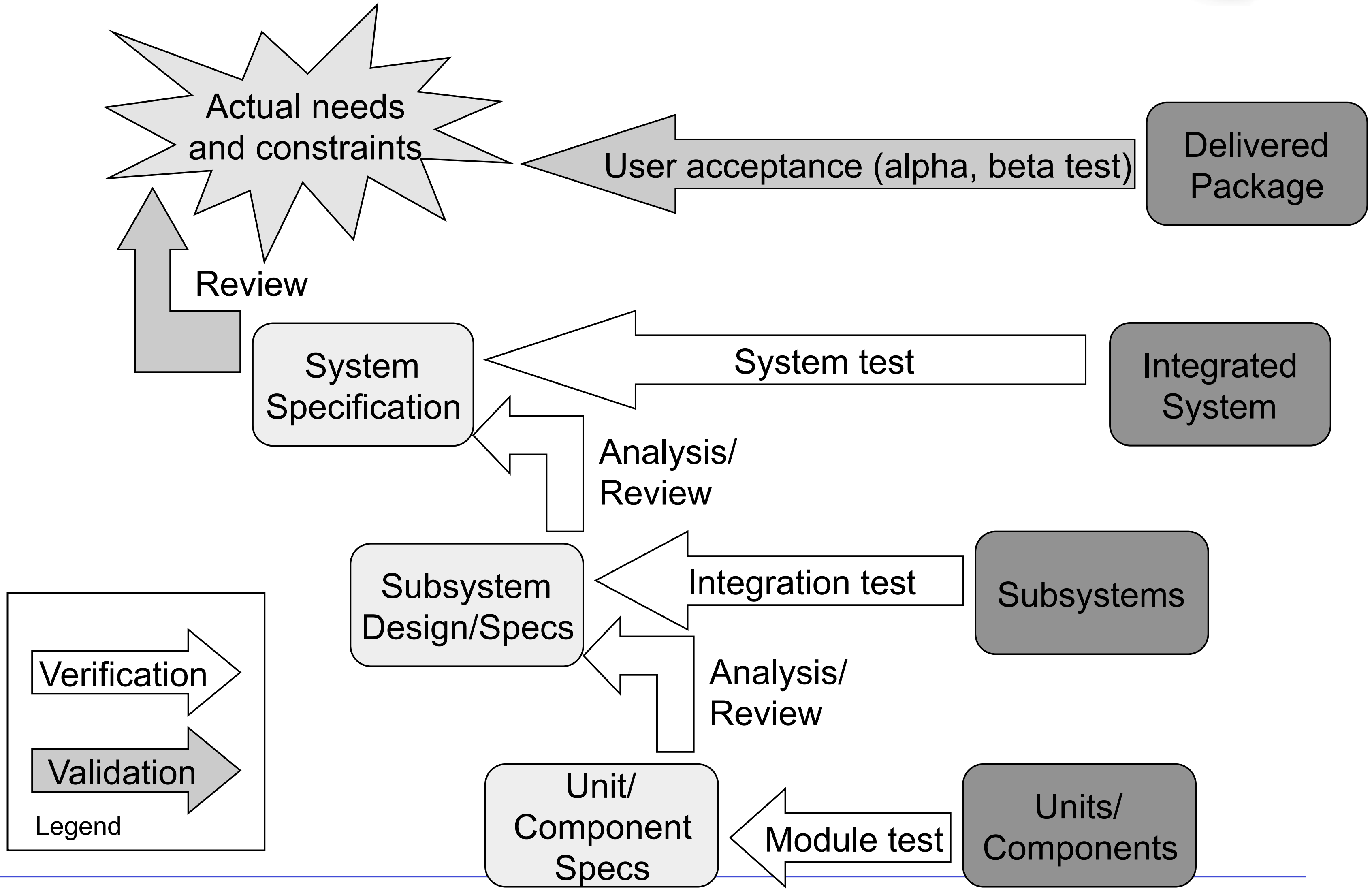
- The company developing the web application sets the following goals
  - No more than 30 min of down time per month
  - MTBF one week
  - < 1 failure per 1000 user sessions
- Measures taken during integration and system test are not representative of actual usage and failure rate
  - The associated test cases have been designed to find faults
- Specific test cases can be generated from previous operational profiles
- Alpha test and beta test are also used


WE ADDED A NEW PERFORMANCE TEST, BUT LEARNED THAT THE TEST ITSELF IS FLAWED.
NOW OUR PRODUCT FAILS OUR OWN TESTS AND OUR CUSTOMERS ARE ASKING TO SEE THE TEST RESULTS.
DO I HAVE PERMISSION TO FAKE THE TEST DATA?
I DIDN'T EVEN KNOW DATA CAN BE REAL.
Dilbert.com DilbertCartoonist@gmail.com
8-11-10 © 2010 Scott Adams, Inc./Dist. by UFS, Inc.

How can we control the quality of successive releases?

- Various new versions of a software can be produced during its life cycle
  - Patches
  - Major releases
- Tests already executed on the first release need to be executed again on the new versions (regression testing)
- Automatic test execution is desirable for speeding up the process
- New test cases are added to the regression test suite as a new version is developed

V&V activities and software artifacts (the V model)



UNIT TEST

aimed at exercising
- algorithm correctness
- functional specs
- structural coverage

conducted by the same developers

INTEGRATION TEST

aimed at exercising
- interfaces
- module interactions

SYSTEM TEST

aimed at exercising
- global behavior
- performance
- Hw/Sw integration

conducted by independent team

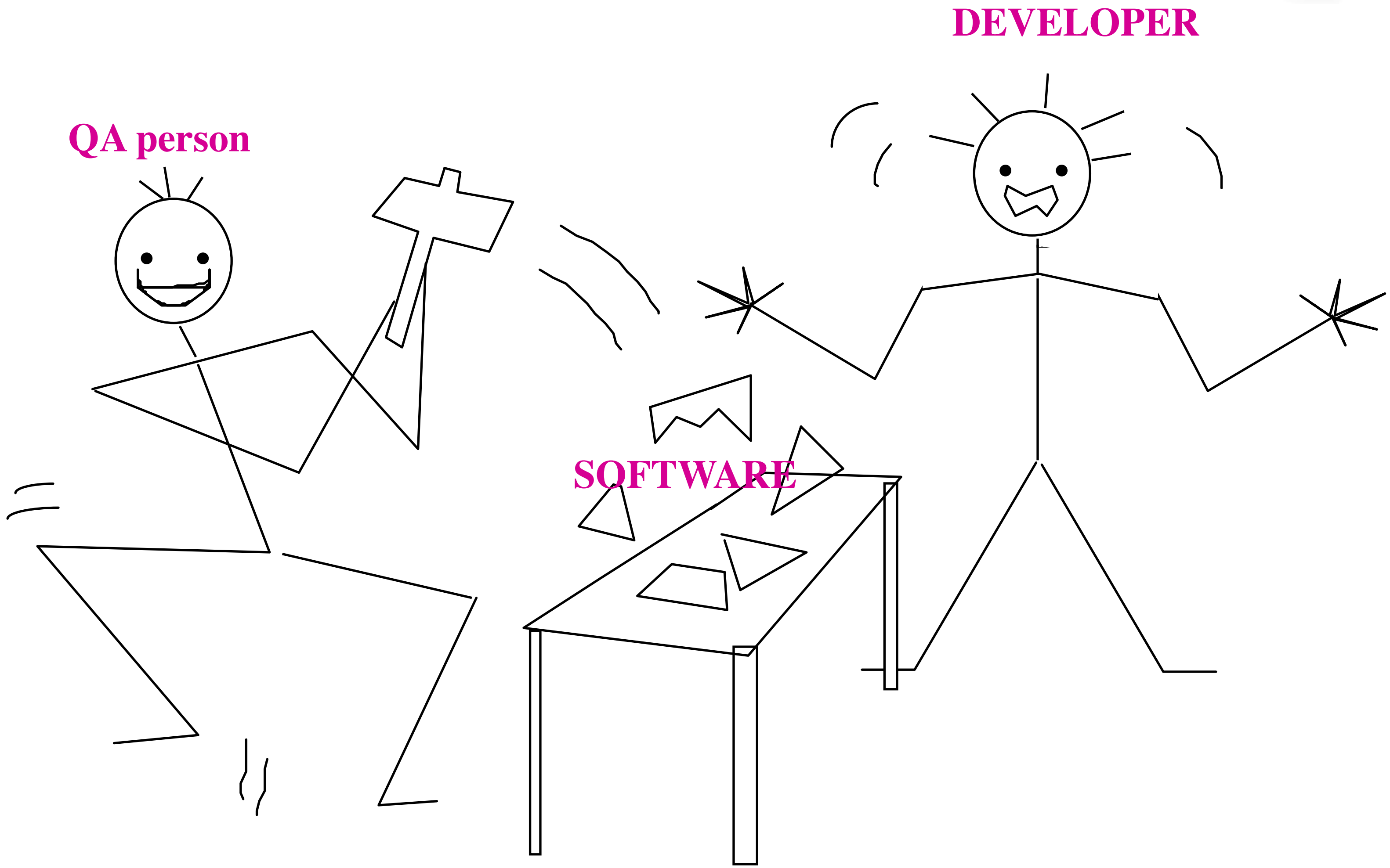
Planning and monitoring

- An *analysis and test plan* identifies
  - Objectives (quality goals) and scope
  - Documents and items that need to be available to perform the various quality assurance activities
  - Items to be tested (and features to be tested)
  - The analysis and test activities to be performed
  - The staff to be involved
- It includes
  - Constraints, pass/fail criteria, schedule, deliverables, hw and sw requirements, risks and contingencies
- Process monitoring and visibility is very important
  - Visibility on the schedule (are we on time with respect to the plan?)
  - Visibility on the achievement of the quality goals

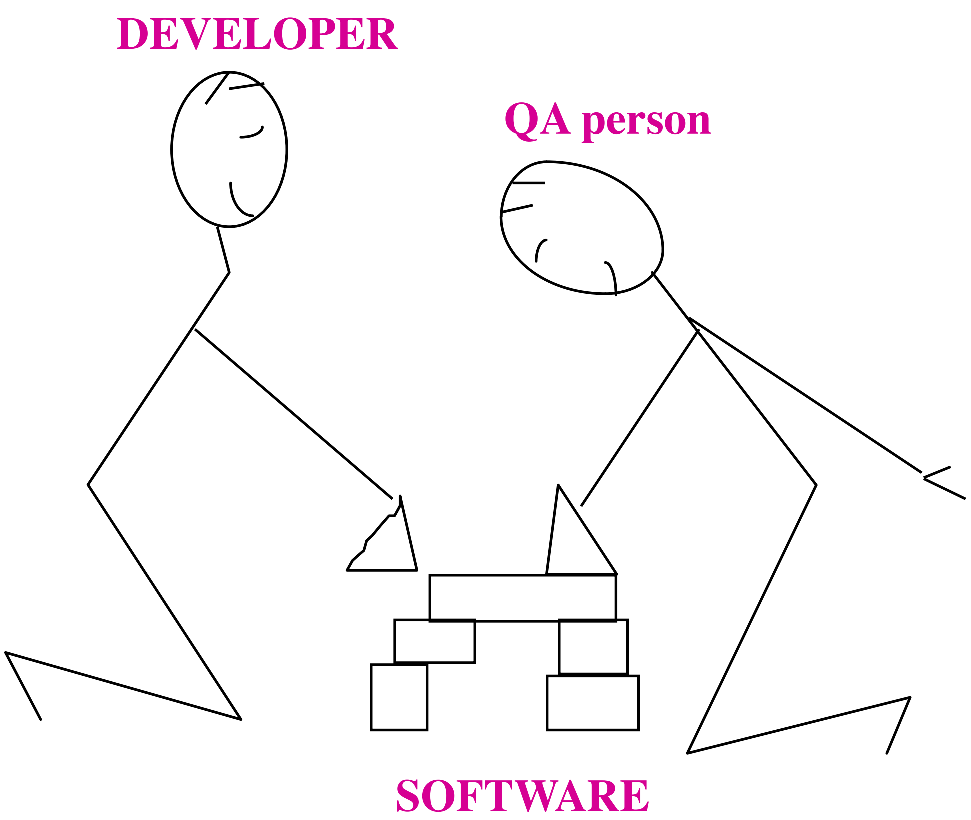
The V&V process improvement

- Should be part of the overall process improvement process
  - Team members should be properly motivated
- Based on analysis of faults detected in previous projects and on the identification of the errors that caused them
- Four phases:
  - Defining the data to be collected about faults
  - Analyzing collected data to identify fault classes
  - Analyzing selected fault classes to identify weaknesses in the development and quality measures
  - Adjusting the quality and development process

Software Verification: from this ...



… to this



Different attitudes:

DEVELOPER	QA Person
•Optimistic	•Pessimistic
•How to better design	•How to better observe
•Interpret and repair bugs	•Discover and report bugs
•Focus on how it could work	•Focus on how it could break

Complementary

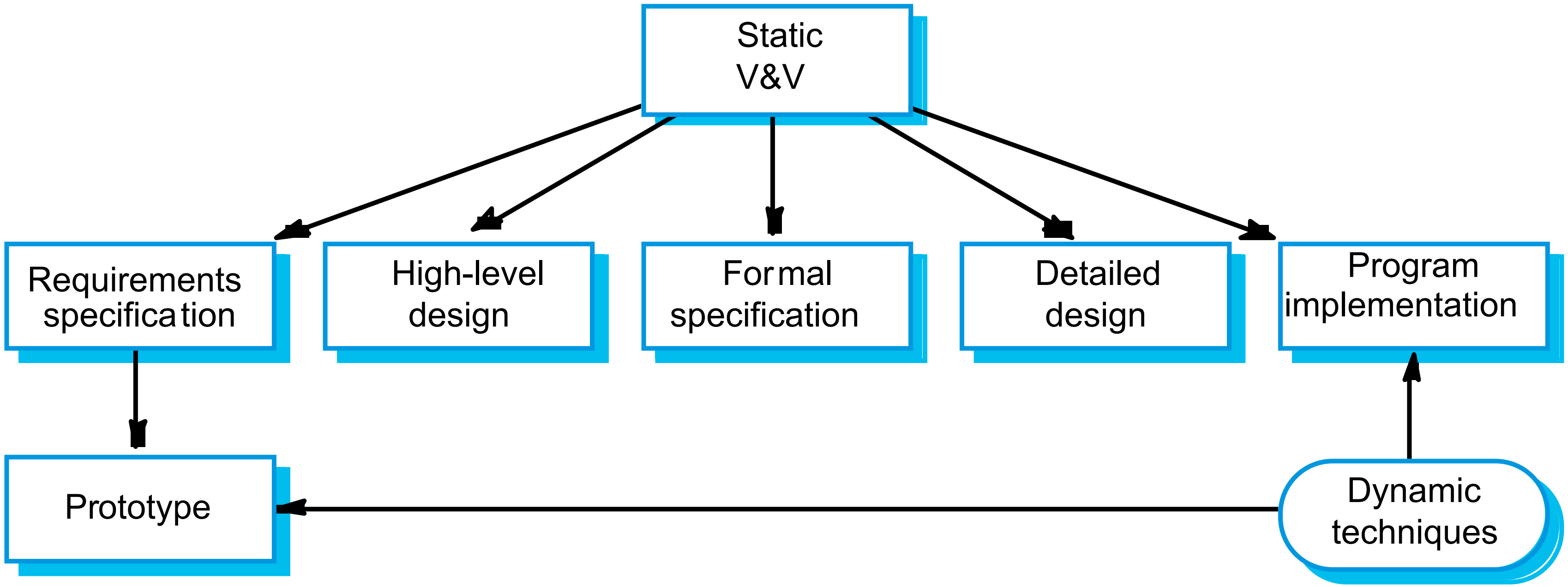
The quality improvement group should involve both developers and quality assurance people

Analysis vs testing

Main approaches to V&V

- ANALYSIS (usually, static technique)
  - analytic study of properties
- TESTING (dynamic technique)
  - experimenting with behavior of the products
  - sampling behaviors
    - GOAL: find "counterexamples"

Static and dynamic V&V



Analysis (vs. testing)

- Does not involve the actual execution of software
- Two main approaches
  - Manual inspection
  - Automated static analysis
- Can be applied at any stage of the development process
- Particularly well suited at the early stages of specification and design
  - Lack of executability makes testing impossible

Reviews, walkthrough, inspections..

- Two kinds of reviews
  - Walkthroughs
  - Inspections
- Latter more formal than first.

- Both:
  - Review is an in-depth examination of some work product by a team of reviewers.
  - Product is anything produced for the lifecycle, i.e., requirements, plans, design, code, test cases, documentation, manuals, everything!
  - No meeting is more than 2 hours.

The focus is on finding errors in the product,

- not on correcting them, and
- not in finding fault in producer of product.

- It is essential that it not be used for employee performance evaluation, either of producer or reviewers.

Differences

- Atmosphere:
  - W: informal I: formal
- Reviewers:

  W: experts in the domain

  I: trained, professional inspectors
- Subject of review:
  - W: correctness of product, as seen by experts
  - I: correctness of product, according to checklist of items to be examined
- Leader of discussion and session controller:
  - W: producer of product
  - I: official moderator of review team

Walkthroughs

- The producer presents product (if code, then also its documentation) and the reviewers comment on the correctness of the product (if code, also consistency with documentation).

Software Inspections

Fagan code inspections

Related formal review techniques

Lessons

Software inspection: history

- Inspection technique was developed by Michael E. Fagan at IBM Kingston.
- Fagan was a certified quality engineer and studied the methods of Deming and Juran.
- He used inspection on a SW project he was managing in 72–74, in effect applying industrial hardware quality methods to SW
- It was very successful!
- He reported results in a now famous 1976 paper.
- The method became very popular in IBM, although there was some resistance.

Software inspection: history

- AT&T Bell Labs started using technique in 1977.
- In 1986, a major Bell Labs SW development organization with 200 people reported its experience with inspections:
  - 14% productivity increase for a single release
  - better tracking and phasing
  - early defect density data improved 10-fold
  - staff credited inspection as an ‘‘important influence on quality and productivity’’

Definitions

ANSI/IEEE Standard 729-1983 IEEE Standard Glossary of SE Terminology defines inspection as

"... a formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems...."

Inspections

- Fagan's Code Inspections

- Roles
- Checklists
- Steps
- Users at Inspections

Software Inspection Roles

- **Inspection team members have specific roles:**
- Moderator:
  - Typically borrowed from another project. Chairs meeting, chooses participants, controls process
- Readers, Testers (inspectors):
  - Read code to group, look for flaws
- Author:
  - Passive participant; answer questions when asked
- Scribe

Software Inspection Process

- Planning
  - Moderator checks entry criteria, choose participants, schedule meeting
- Overview
  - Provide background education, assign roles
- Preparation
- Inspection (see ahead)
- Rework
  - The producer fixes the product according to list of faults in report
- Follow-up (& possible re-inspection)
  - The moderator makes sure that all faults have been fixed and calls another inspection if more than 5% of the product has been modified

In the Meeting

- Goal: Find as many faults as possible
  - max 2 x 2 hour sessions per day
  - approx. 150 source lines/hour
- Approach: Line-by-line paraphrasing
  - Reconstruct intent of code from source
  - May also "hand test"
- Find and log defects, but don't fix them
  - Moderator responsible for staying on track

Checklists — NASA example

From “Software Formal Inspections Guidebook,” Office of Safety and Mission Assurance, NASA-GB-A302 approved August 1993

- About 2.5 pages for C code, 4 for FORTRAN
  - Divided into: Functionality, Data Usage, Control, Linkage, Computation, Maintenance, Clarity
- Examples:
  - Does each module have a single function?
  - Does the code match the Detailed Design?
  - Are all constant names upper case?
  - Are pointers not typecast (except assignment of NULL)?
  - Are pointers immediately set to NULL (or 0) following the deallocation of memory?
  - Are nested “INCLUDE” files avoided?
  - Are non-standard usages isolated in subroutines and well documented?
  - Are there sufficient comments to understand the code?

Example:
Do you see the problem here?

- Code from Apache web server, version 2.0.48
- Response to normal page request on secure (https) port

```
static void ssl_io_filter_disable(ap_filter_t *f)
{   bio_filter_in_ctx_t *inctx = f->ctx;


    inctx->ssl = NULL;
    inctx->filter_ctx->pssl = NULL;
}
```

Are pointers immediately set to NULL (or 0) following the deallocation of memory?

Incentive Structure from [Fagan 86]

- Faults found in inspection are not used in personnel evaluation
  - Programmer has no incentive to hide faults
- Faults found in testing (after inspection) are used in personnel evaluation
  - Programmer has incentive to find faults in inspection, but not by inserting more

Why does inspection work?

- The evidence says it is cost-effective. Why?
  - Detailed, formal process, with record keeping
  - Check-lists; self-improving process
  - Social aspects of process, esp. for author
  - Consideration of whole input space
  - Applies to incomplete programs
- Limitations
  - Scale: Inherently a unit-level technique
  - Non-incremental; what about evolution?

Some figures

- Experiments over several projects have shown:
  - can be 4 times more effective than testing and
  - 2 times more effective than walkthroughs

  in finding errors
- Aetna Insurance Company:
  - Formal Review found 82% of errors, 25% cost reduction
- Bell-Northern Research:
  - Inspection cost: 1 hour per defect.
  - Testing cost: 2-4 hours per defect.
  - Post-release cost: 33 hours per defect
- Hewlett-Packard
  - Est. inspection savings (1993): $21,454,000

Document inspections

- Inspections can be applied on the whole documentation set build during the software lifecycle
- See the NASA document for guidelines, examples for the checklist
  - Is the design functionally cohesive?
  - Have the assumptions been documented?
  - Will the selected design or algorithm meet all of its requirements?
- http://www.hq.nasa.gov/office/codeq/doctree/NS87399.pdf

Caveat

- Team members must read product before meeting.
- Meeting must not be longer than two hours.
- Author's boss cannot be present at the meeting.
- Inspectors must not consider solutions during meeting.
- Inspectors must not attack author; they can criticize product.

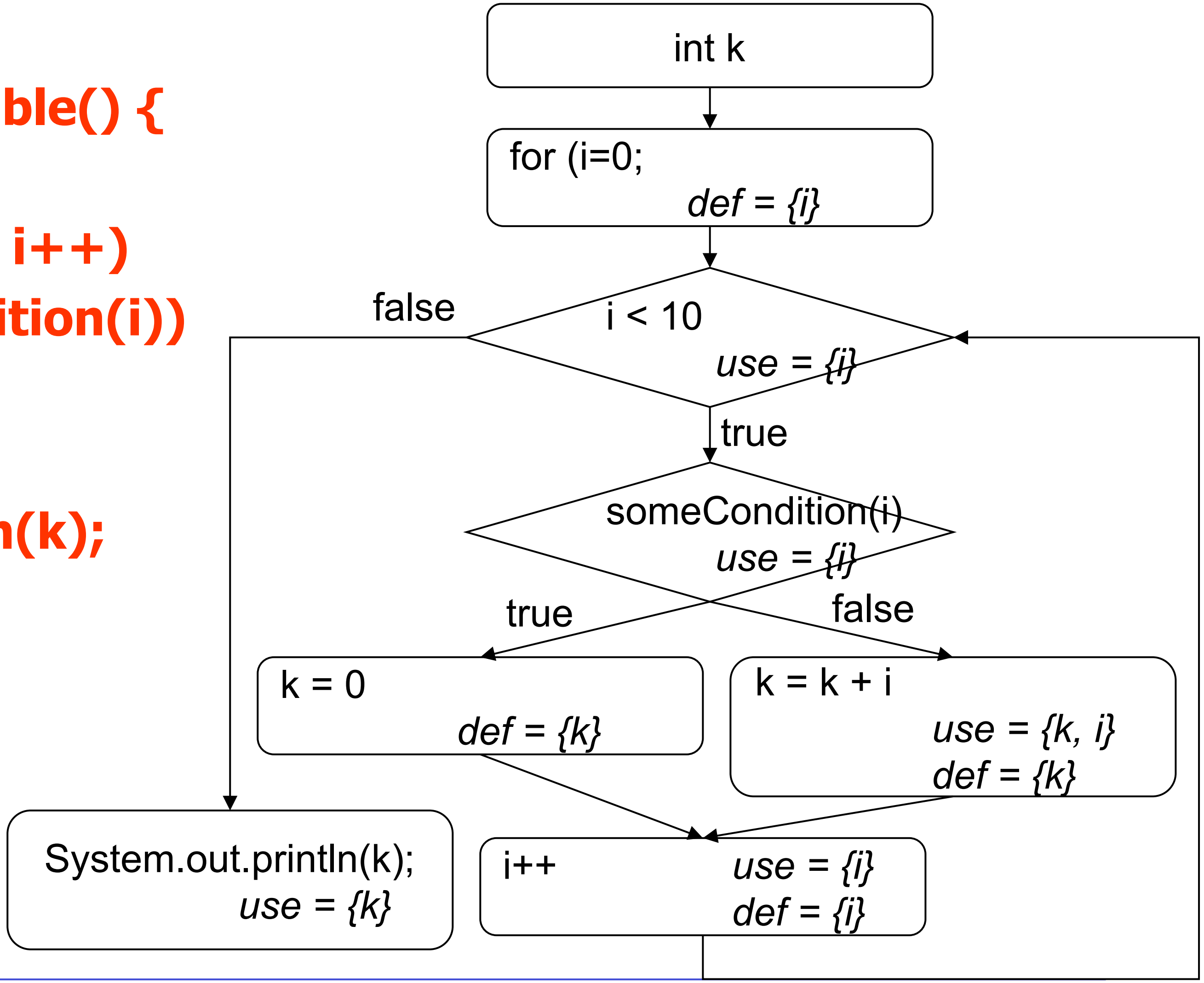
Analysis: Automated Approaches

Data flow analysis

- Based on the identification of variables definitions and use
- Typically used by compilers
  - Check for possible errors (e.g., a variable being used but not initialized)
  - Code optimization (e.g., computation results that can be used later on in the execution are saved)

Variables defs and uses

```
static void questionable() {
    int k;
    for (int i=0; i<10; i++)
        if (someCondition(i))
            k = 0;
        else k = k+i;
    System.out.println(k);
}
```



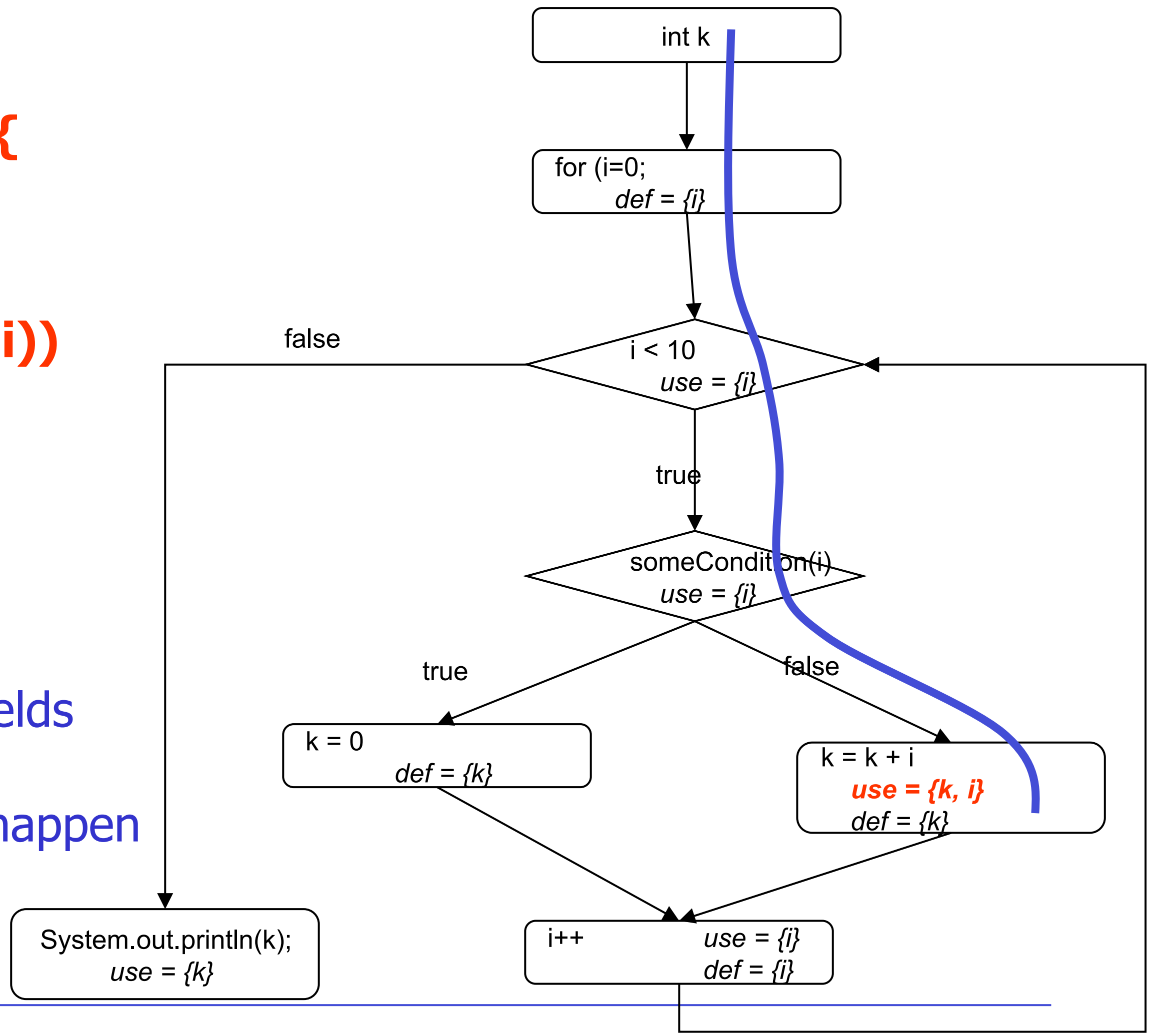
Possible checks

- Is a variable always initialized when used?
- Is a variable assigned and then never used?
- Does a variable always get a new value before being used?

Note that these are not errors, but symptoms of possible errors

Is a variable always initialized when used?

```
static void questionable() {
    int k;
    for (int i=0; i<10; i++)
        if (someCondition(i))
            k = 0;
        else k = k+i;
    System.out.println(k);
}
```



Assuming that someCondition yields false, k is not initialized

Maybe in practice this does not happen

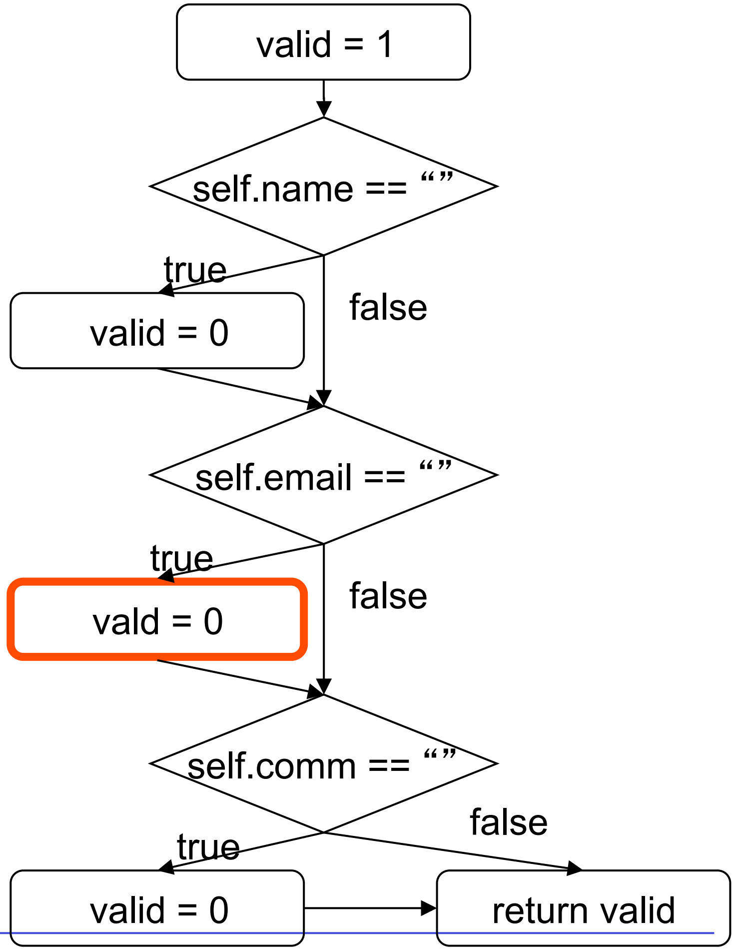
Dataflow is a “pessimistic” tool

Checking for useless definitions

CGI script

```
class SampleForm(FormData):
 fieldnames=('name', 'email',
     'comment')
 def validate(self):
   valid = 1;
   if self.name=="": valid = 0
   if self.email=="": vald = 0
   if self.comment=="": valid = 0
   return valid
```

dataflow discovers typing error as an unused variable "vald"



Data flow analysis: limitations

- It is pessimistic: it cannot distinguish between paths that can actually be executed and paths that cannot

- It cannot always determine whether two names or expressions refer to the same object

Symbolic execution

- Builds predicates that characterize
  - Conditions for executing paths
  - Effects of the execution on program state
- Symbolic state:

  <path-condition, symbolic bindings>
- Finds important applications in
  - program analysis
  - test data generation
  - formal verification (proofs) of program correctness

Symbolic state

Values are expressions over symbols

Executing statements computes new expressions, example:

```
mid = (high+low)/2
```

Execution with concrete values

before instruction

low	12
high	15
mid	-

after instruction

low	12
high	15
mid	13

Execution with symbolic values

before instruction

low	L
high	H
mid	-

after instruction

Low	L
high	H
mid	(L+H)/2

Example 1

```
read (a); read (b);
x = a + b;
write (x);
```

- Let A and B the symbolic values read for a, b
- x = A+B
- Printed result is A+B

More examples

```
read (a); read (b);
x = a + 1;
y = x *  b;
write (y);
[res: (A + 1) * B]
```

```
read (a); read (b);
x = a + 1;
x = x + b + 2;
write (x);
[res: A + B + 3]
```

Path condition

Path condition (PC) needed to record condition on input data that specifies the path followed during symbolic execution

```
1   read (y, a);
2   x = y + 2;
3   if x > a
4           a = a + 2;
5   else
6           y = x + 3;
7   x = x + a + y;
```

How can the symbolic executor determine if x > a is true or false?

Execution is performed for a specific path, for instance:

Execution path: <1, 2, 3, 5, 6, 7>

Path condition: $Y + 2 \leq A$

Execution result:

{a=A, y=Y+5, x=2Y+A+7}

In short:

<{a = A, y=Y+5, x=2*Y+A+7}<1, 2, 3, 5, 6, 7> Y + 2 ≤ A>

Path condition

- If no initial assumption on data, it is TRUE
- If code fragment has a precondition, this gives the initial value to path condition

```
int gcd (int x, y){
        pre: x,y>0
        …
}
```

Initial state symbolic interpreter

x=X   y=Y

pc: X>0 & Y>0

Uses of PC

- PC can be used to synthesize data that follow a given execution path
- If a fragment (e.g., a subprogram) is specified by a pre and a post-condition
  - check that pc implies pre initially
  - in example, symb-expressions for x and y must be positive
- You can restart execution after the fragment with PC=post & "part of old PC unaffected by fragment" and new symbolic values

Example

- Consider the following fragment of code :

```
int foo() {
1    x = input();
2   while (x > 0) {
3       y = 2 * x;
4       if (x > 10)
5         y = x - 1;
6       else
7         x = x + 2;
8      x = x - 1;
9   }
10  x = x - 1;
11 return x;
}
```

You are to accomplish the following:

1. draw the control flow graph of the program;
2. provide the use-definition information for variables x and y;
3. point out a potential issue with this code that data flow analysis would be able to spot;
4. using symbolic execution, identify the path condition that allows the program to follow the path 1 2 3 4 5 8 9 2 3 4 6 7 8 9 2 10 11.

References

- C. Ghezzi, M. Jazayeri, D. Mandrioli *Fundamentals of Software Engineering*, 2nd Edition, Prentice Hall, 2003 (Italian translation Pearson Ed. 2004)
- M. Pezzè, M. Young *Software Testing and Analysis: Process, Principles, and Techniques*. To be published. (A draft version will be available on the course site)
- Michael Fagan, Advances in Software Inspections, *IEEE Transactions on Software Engineering*, July 1986 www.mfagan.com/aisi1986.pdf
- Nasa, "Software Formal Inspections Guidebook," Office of Safety and Mission Assurance, NASA-GB-A302 approved August 1993 (http://satc.gsfc.nasa.gov/fi/gdb/fitext.txt)
- Check list for inspections of Java code by Christopher Fox (available on the course web site)
- A tool for automatic inspection of Java code http://www.hammurapi.org