Software Testing

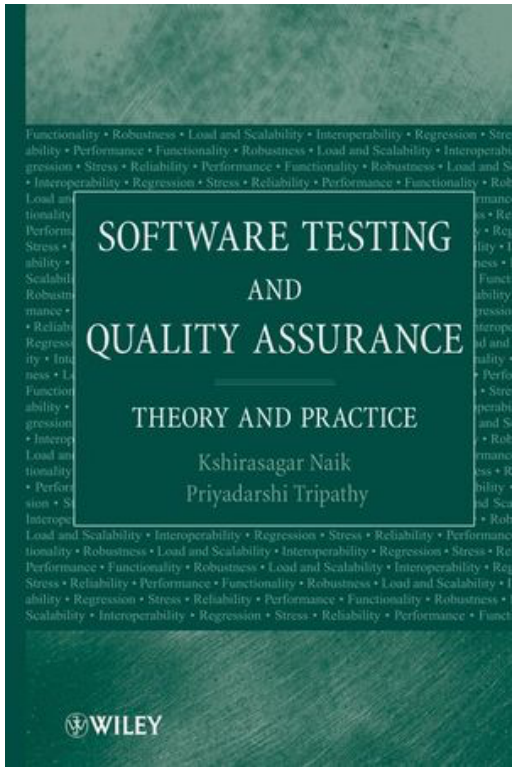
ETS 200

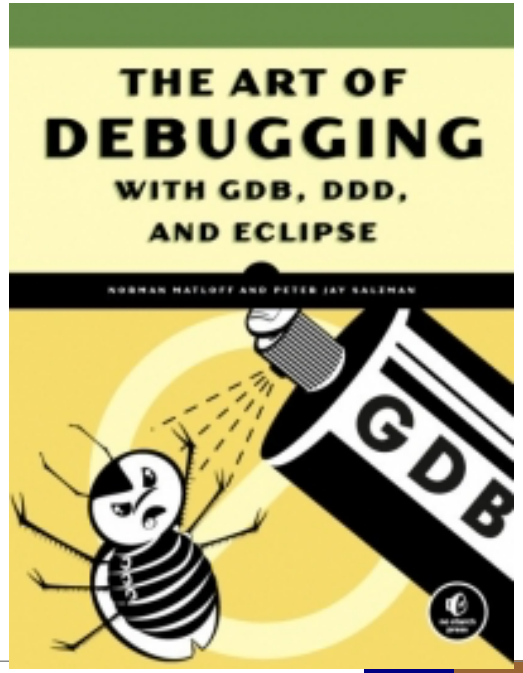
http://cs.lth.se/ets200

Matloff & Salzmann,

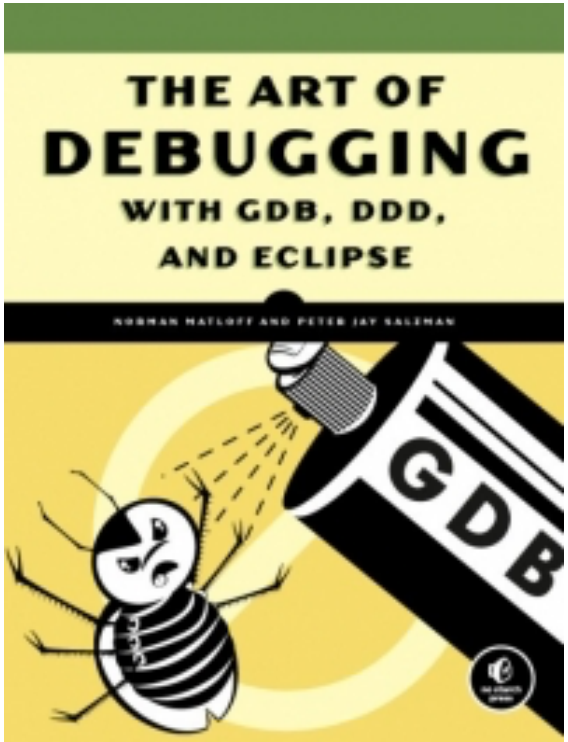
Naik & Tripathy Ch 3.2, 10.1-10.4, 15

Prof. Per Runeson





Debugging



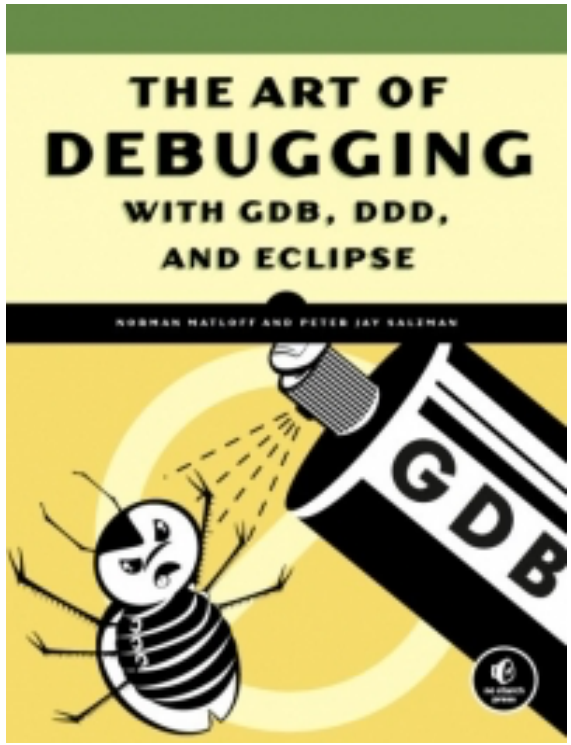
Preparations for Lab 3…

… and for life

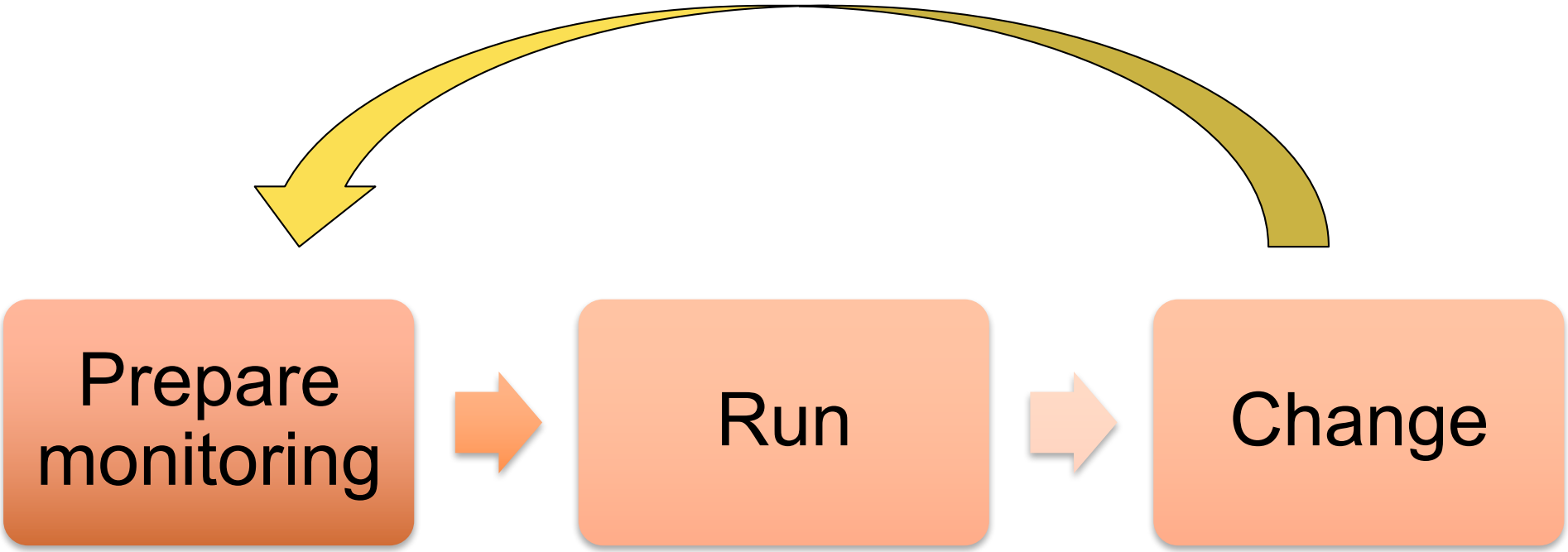
http://it-ebooks.directory/book-1593271743.html

Debugging

- is a separate activity, distinct from testing
- needs appropriate tool support
- must be done systematically



Debugging process



Prepare monitoring

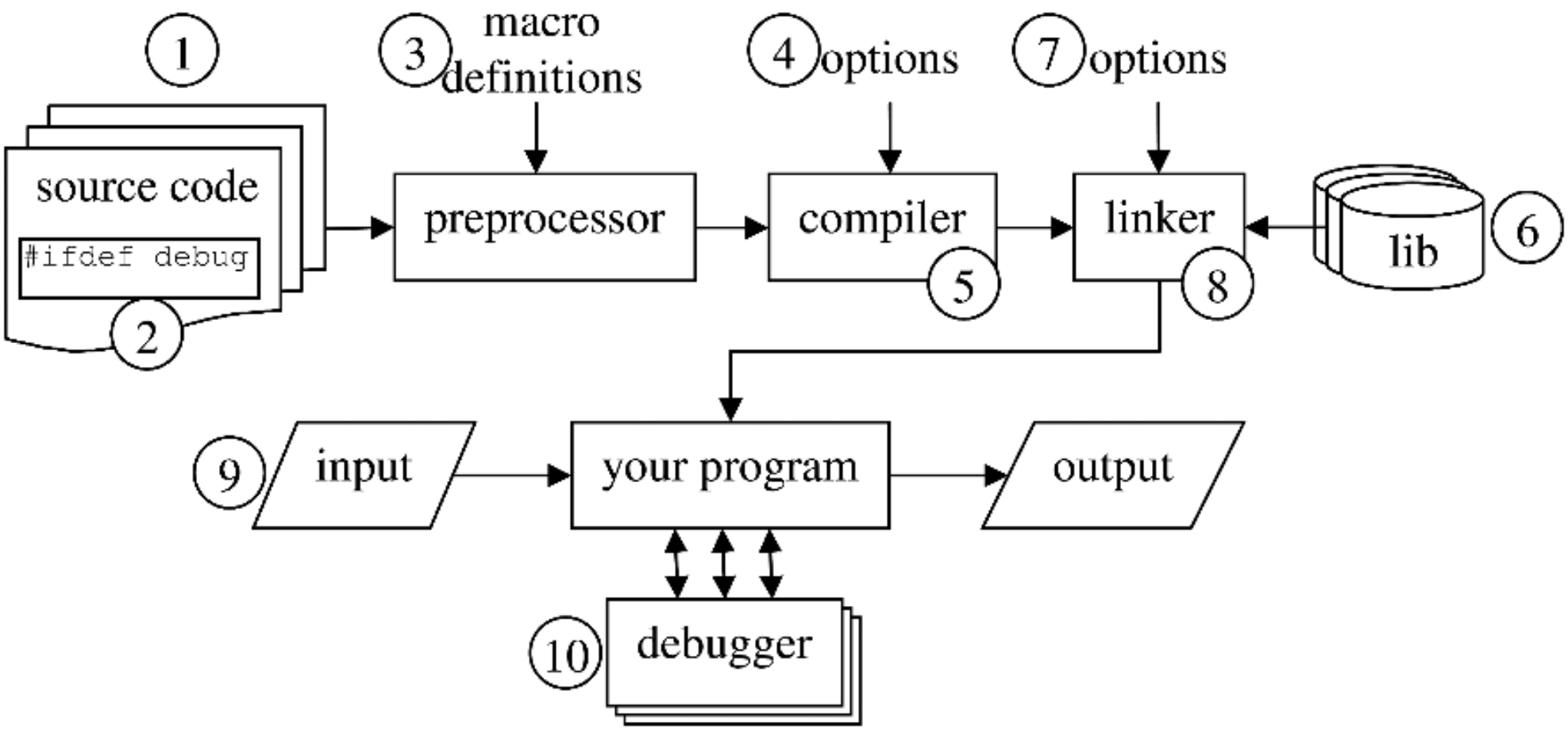
- Basic: println

+ quick start
– requires editing
– impacts performance

- Advanced: debugging tools

+ no editing
+ breakpoints, set, get
– may impact performance

Example debugging environment



The Developer's Guide to Debugging, Grötker et al

Run – Use binary search principles

- Assemble a list of potential problems and how to debug them
- Separate changes of the environment and source code changes
    - Track down changes of the environment
    - Isolate source code changes via back-out builds
- Zoom in and conquer
    - Memory debugger
    - Conventional source code debugging
    - Side-by-side debugging

Change

- ONLY CHANGE ONE THING AT A TIME!
- Keep an audit trail
- Regression test!

Reviews

Reviews (Ch 3.2)

Terminology

- **Static testing** – testing without software execution
- **Review** – meeting to evaluate software artifact
- **Inspection** – formally defined review
- **Walkthrough** – author guided review

Why Review?

- Main objective
  - Detect faults
- Other objectives
  - Inform
  - Educate
  - Learn from (others') mistakes → Improve!
- (Undetected) faults may affect software quality negatively - even during the development process!

Empirical Results

- R*equirements defects* – reviews since finding defects early is cheaper
- D*esign defects* – inspections are both more efficient and more effective than testing.
- C*ode defects* – functional or structural testing is more effective or efficient than inspection.
  - May be complementary
- G*enerally, reported effectiveness* → low
  - Inspections find 25-50% of an artifact's defects
  - Testing finds 30-60% of defects in the code

Empirical Results

Table 3

Average values of effectiveness and efficiency for defect detection

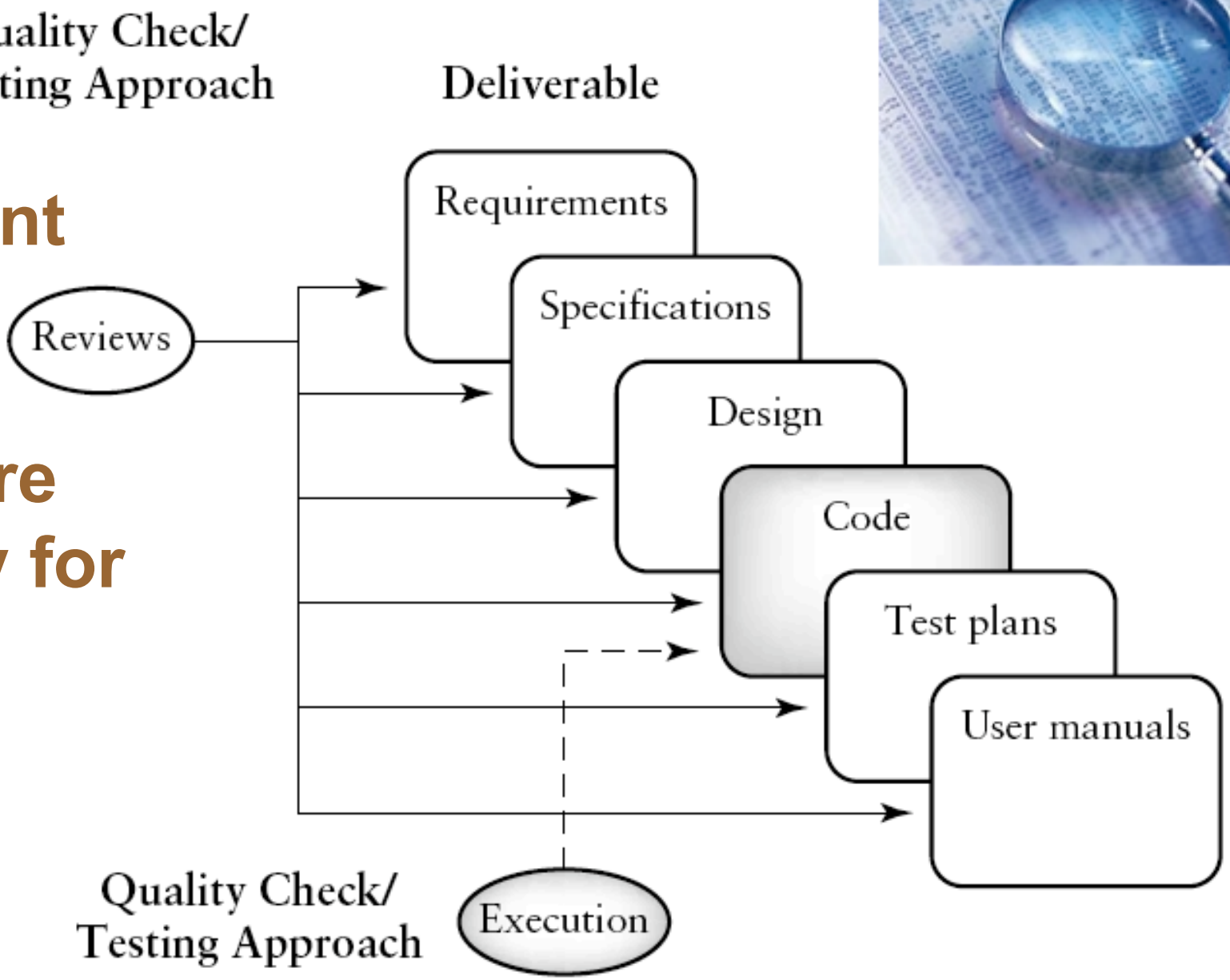
		Study	Inspection effectiveness (%)*,†	Inspection efficiency†,‡	Testing effectiveness*,§	Testing efficiency‡,§	Different faults found
Experiments	Code	Hetzel[17]	37.3	–	47.7; 46.7	–	–
		Myers[18]	38.0	0.8	30.0; 36.0	1.62; 2.07	Yes
		Basili and Selby[2]	54.1	Dependent on software type	54.6; 41.2	–	Yes
		Kamsties and Lott[19]	43.5 50.3	2.11 1.52	47.5; 47.4 60.7; 52.8	4.69; 2.92 3.07; 1.92	Partly (for some types)
		Roper et al.[20]	32.1	1.06	55.2; 57.5	2.47; 2.20	Yes
		Laitenberger[11]	38	–	9#	–	No
		So et al.[12]	17.9, 34.6	0.16; 0.26	43.0	0.034	Yes
		Runeson and Andrews[13]	27.5	1.49	37.5	1.8	Yes
		Juristo and Vegas[14]	20.0 –	– –	37.7; 35.5 75.8; 71.4	– –	Partly (for some types)
	Design	Andersson et al.[15]	53.5	5.05	41.8	2.78	Yes for one version, no for the other
Case studies		Conradi et al.[16]	–	0.82	–	0.013	–
		Berling and Thelin[3]	86.5 (estimated)	0.68 (0.13)	80	0.10	Yes

* *Percent of the artifact's defects that are detected.*
† *Single entries involve code reading; multiple entries in one cell are reported in this order: code reading, Fagan inspection.*
‡ *Detected defects per hour.*
§ *Single entries involve functional testing; multiple entries in one cell are reported in this order: functional test, structural test.*
\# *Testing is conducted in sequence after the inspection.*

Source:
Runeson, P.; Andersson, C.; Thelin, T.; Andrews, A.; Berling, T.; , "What do we know about defect detection methods?", *IEEE Software*, vol.23, no.3, pp. 82-90, May-June 2006

Reviews complement testing

Reviews are mandatory for e.g. safety critical software



Review method: Walkthrough

- Author guides through artifact
- Attendants scrutinize and question
- Possibly ”simulation”

Review method: Walkthrough

- Objective
  - Detect faults
  - Become familiar with the product
- Elements
  - Planned meeting
  - Team (2 to 7 people)
  - Brainstorm
- Roles
  - Presenter (author)
  - Reviewers (Inspectors)
- Disadvantage
  - Find fewer faults than (formal) inspections

Review method: Inspections

- Objective:
  - Detect faults
  - Collect data
  - Communicate information
- Roles
  - Moderator
  - Reviewers (Inspectors)
  - Presenter
  - Author
- Elements
  - Planned, structural meeting
  - Preparation important
  - Team (3 to 6 people)
- Disadvantages
  - Short-term cost

Code review Process

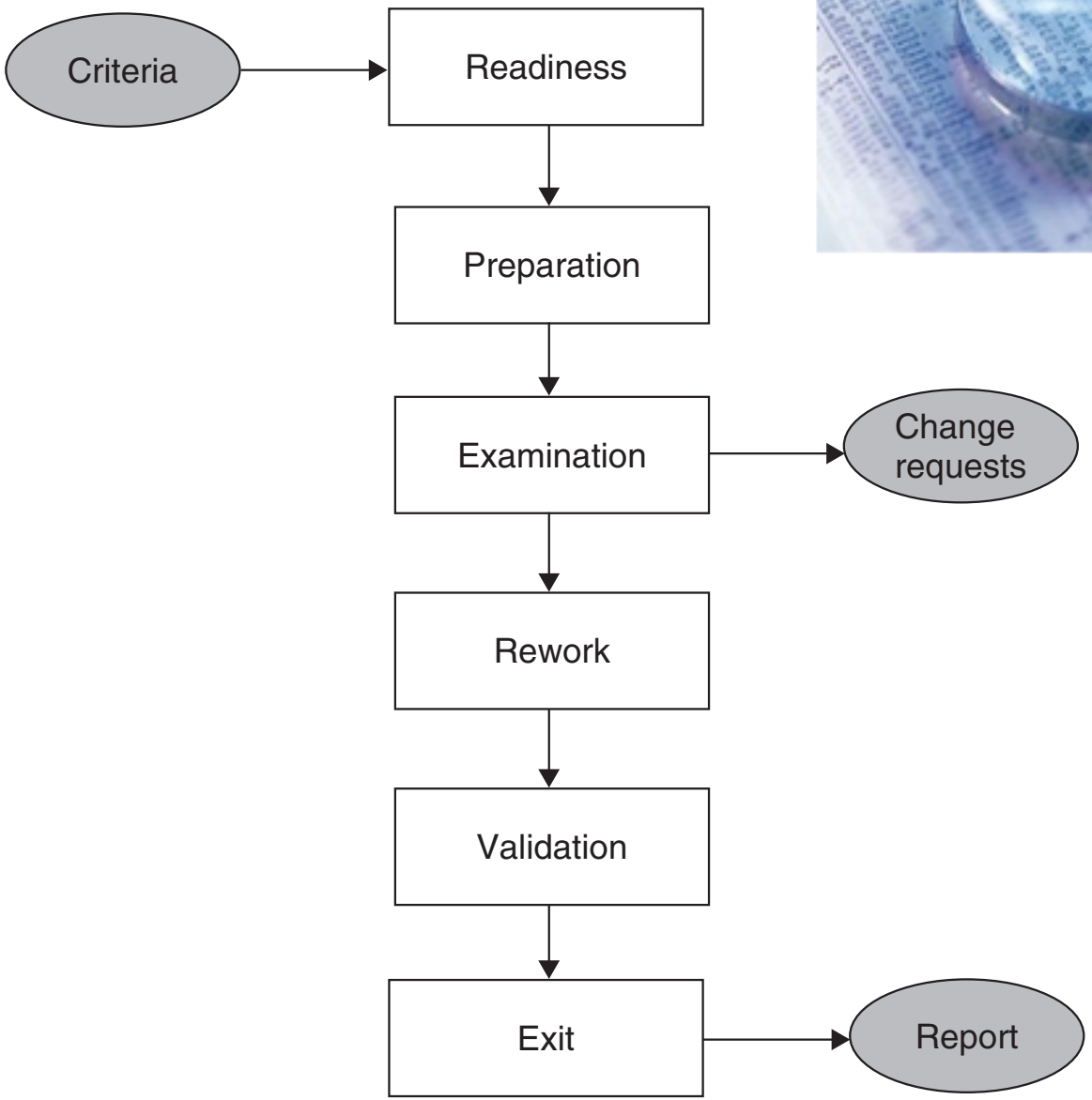
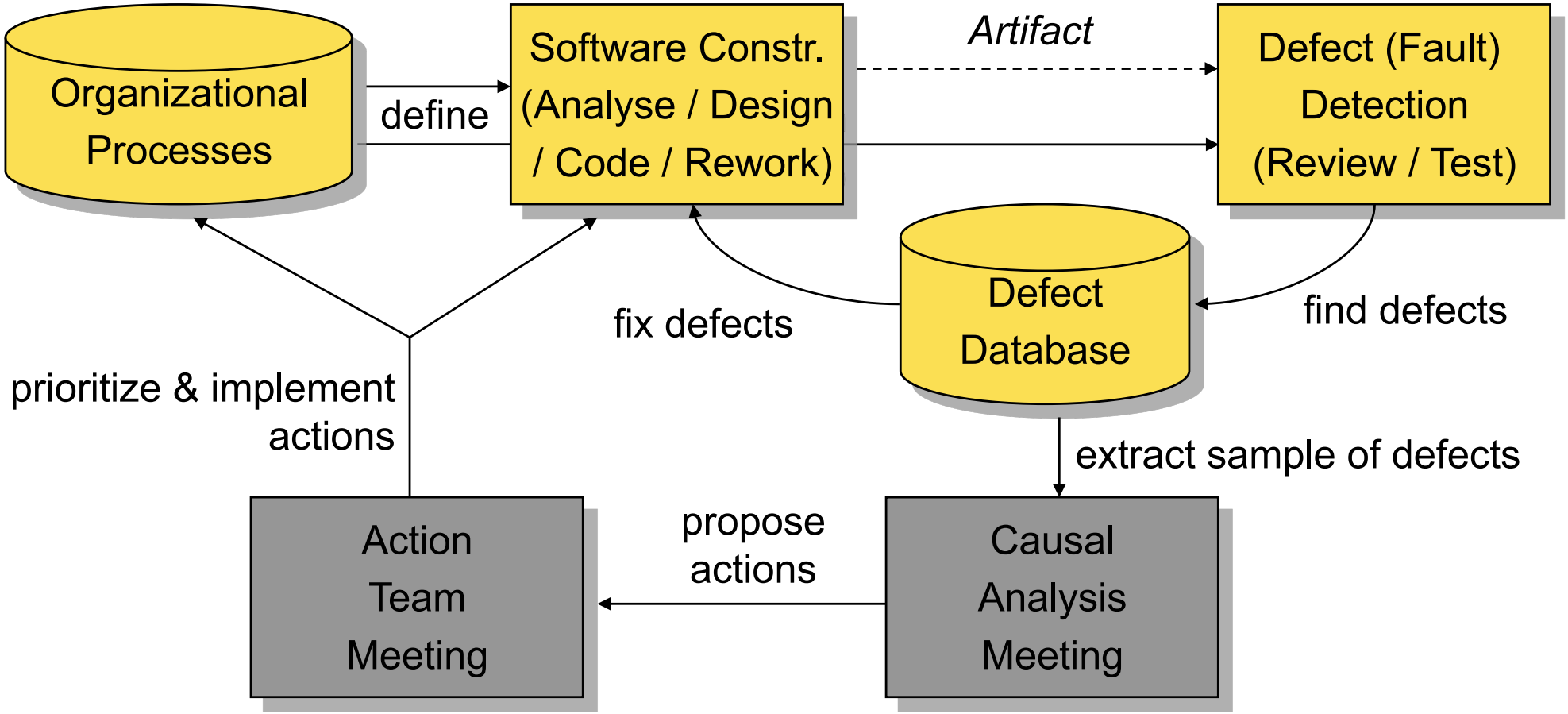


Fig 3.1

Defect Causal Analysis (DCA)



Reviews in Open Source communities

- Gate to getting code accepted
- Performed by experts
- Issues of triaging and priority

Table 1
Overview of peer review processes in Mozilla and Python.

	Mozilla	Python
Bugs reported	7322	1850
Distinct bug reporters	5419	829
Distinct bug contributors (reporters and commenters)	6704	1188
Core developers	176	127
(Ratio among all contributors)	(2.63%)	(10.69%)
Bug reported by core developers	779	487
(Ratio among bugs reported)	(10.64%)	(26.32%)
False positives (non-real bugs)	3418	469
(Ratio among bugs reported)	(46.68%)	(25.35%)
Fixed bugs	498	832

P. C. Rigby, D. M. German, and M. D. Storey. Open source software peer review practices: a case study of the apache server. ICSE, pages 541–550. ACM, 2008.
J. Wang, P. C. Shih, Y. Wu, and J. M. Carroll. Comparative case studies of open source software peer review practices. Information and Software Technology, 67:1 – 12, 2015.

Getting the best from reviews

- The author
  - ”… is in the hot seat”
  - How do you react?
- The development team
  - Better prepared
  - Feedback +/–
  - Communication
- The review team
  - Critical thinking
  - Ability to detect omissions
  - Who should participate in the review?
- Cost-effective verification
  - Minimising cost of correction
  - Is it cost-effective?

Review Metrics

Basic

- Size of review items
- Review time & effort
- Number of defects found
- Number of slipping defects found later

Derived

- Defects found per review time or effort
- Defects found per artifact size
- Size per time or effort

Reading Techniques

- Ad hoc
- Checklist-based
- Perspective-based
- Usage-based
- …

Checklist

Problem/Defect Type: General Checklist

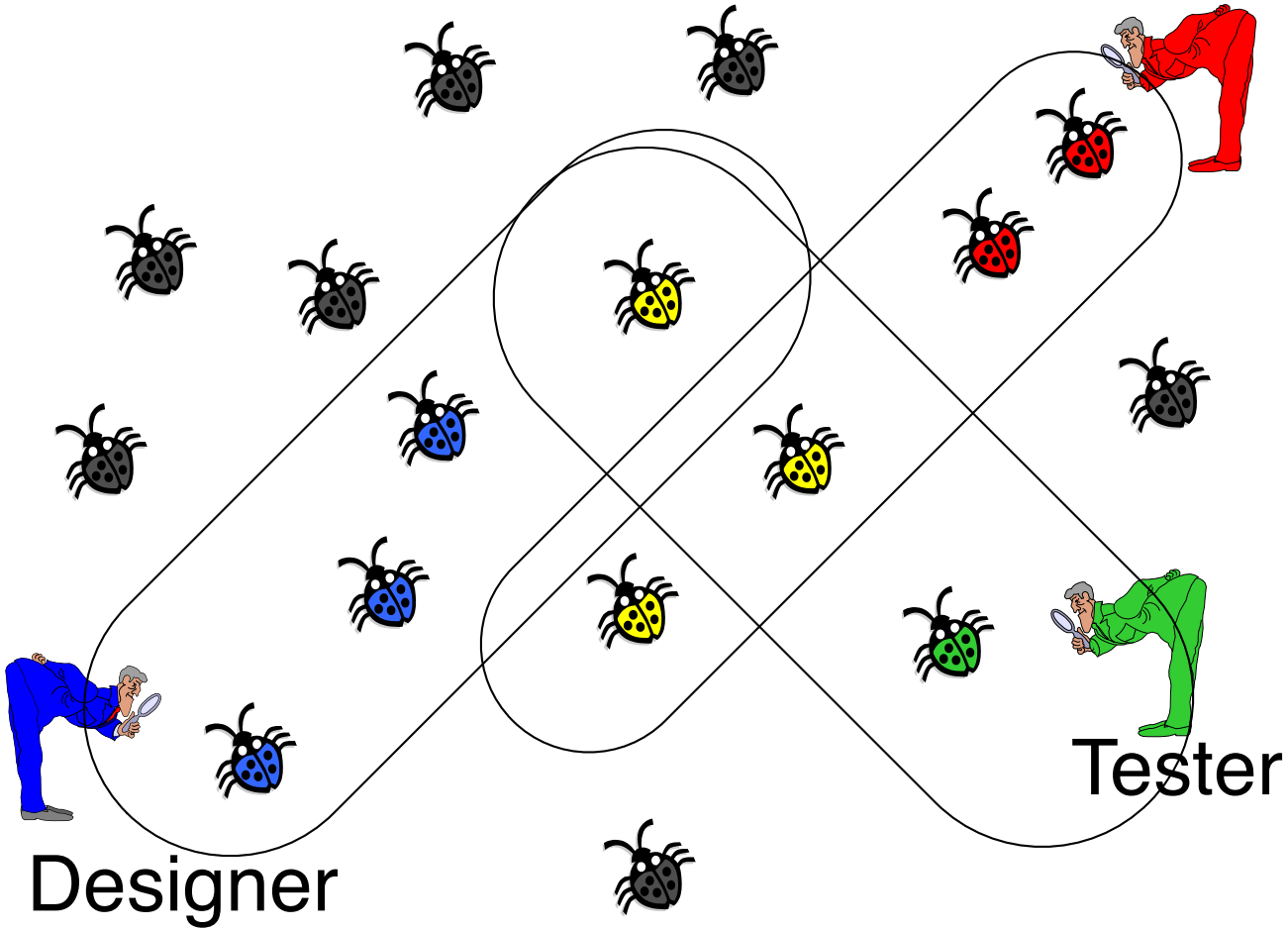
Coverage and completeness
Are all essential items completed?
Have all irrelevant items been omitted?
Is the technical level of each topic addressed properly for this document?
Is there a clear statement of goals for this document? Are the goals consistent with policy?

Correctness
Are there any incorrect items?
Are there any contradictions?
Are there any ambiguities?

Clarity and Consistency
Are the material and statements in the document clear?
Are the examples clear, useful, relevant, correct?
Are the diagrams, graphs, illustrations clear, correct, use the proper notation, effective, in the proper place?
Is the terminology clear, and correct?
Is there a glossary of technical terms that is complete and correct?
Is the writing style clear (nonambiguous)?

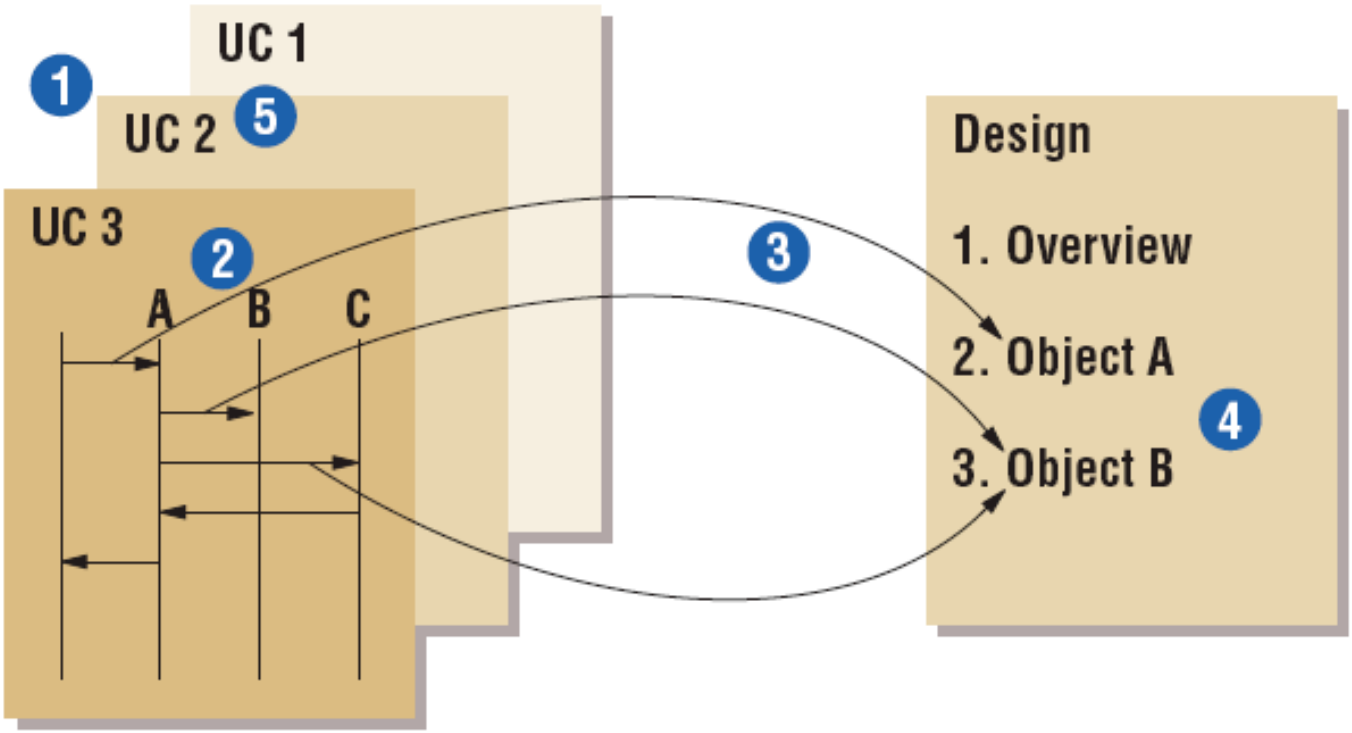
References and Aids to Document Comprehension
Is there an abstract or introduction?
Is there a well-placed table of contents?
Are the topics or items broken down in a manner that is easy to follow and is understandable?
Is there a bibliography that is clear, complete and correct?
Is there an index that is clear, complete and correct?
Is the page and figure numbering correct and consistent?

Perspective-based Reading



- Scenarios
- Purpose
  - Decrease overlap (redundancy)
  - Improve effectiveness

Usage-Based Reading



software inspection

Prioritized Use Cases as a Vehicle for Software Inspections

Thomas Thelin and Per Runeson, *Lund University*
Claes Wohlin, *Blekinge Institute of Technology*

Software inspection has garnered much attention since Michael Fagan introduced it in 1976.[1] Inspection, a static verification and validation technique, is applicable to all kinds of software development artifacts, because inspections don't require dynamic execution. It is a common industry practice in domains where the delivered software product quality is important. Although researchers have studied inspection a lot, the traditional checklist-based reading (CBR) approach still dominates industry.[2,3]

Another software engineering trend over the last two decades has been object-oriented design methods and the application of use cases in this context.[4] *Scenarios* (sequences of steps describing an interaction between a user and a program) and *use cases* (sets of scenarios tied together by a common user goal) help developers focus on users' needs instead of on technology alone.[5] For example, developers can document use cases and scenarios with UML use case diagrams and sequence diagrams or by using structured text. Use cases are typically used in specifying a system but can also be used for inspection purposes later in the development process.

Usage-based testing, or operational-profile testing, helps focus test activities on a system's users and usage.[6] The testers select the test cases according to the frequency or probability of operational use. So, testers check the most important user functions first and devote the most time to them. The typical inspection process[7] involves

1. A group of reviewers overviewing the artifact to be inspected
2. Each reviewer reading the artifact (individual preparation)
3. Compiling the findings into one list of issues
4. Conducting an inspection meeting
5. Reworking and following up

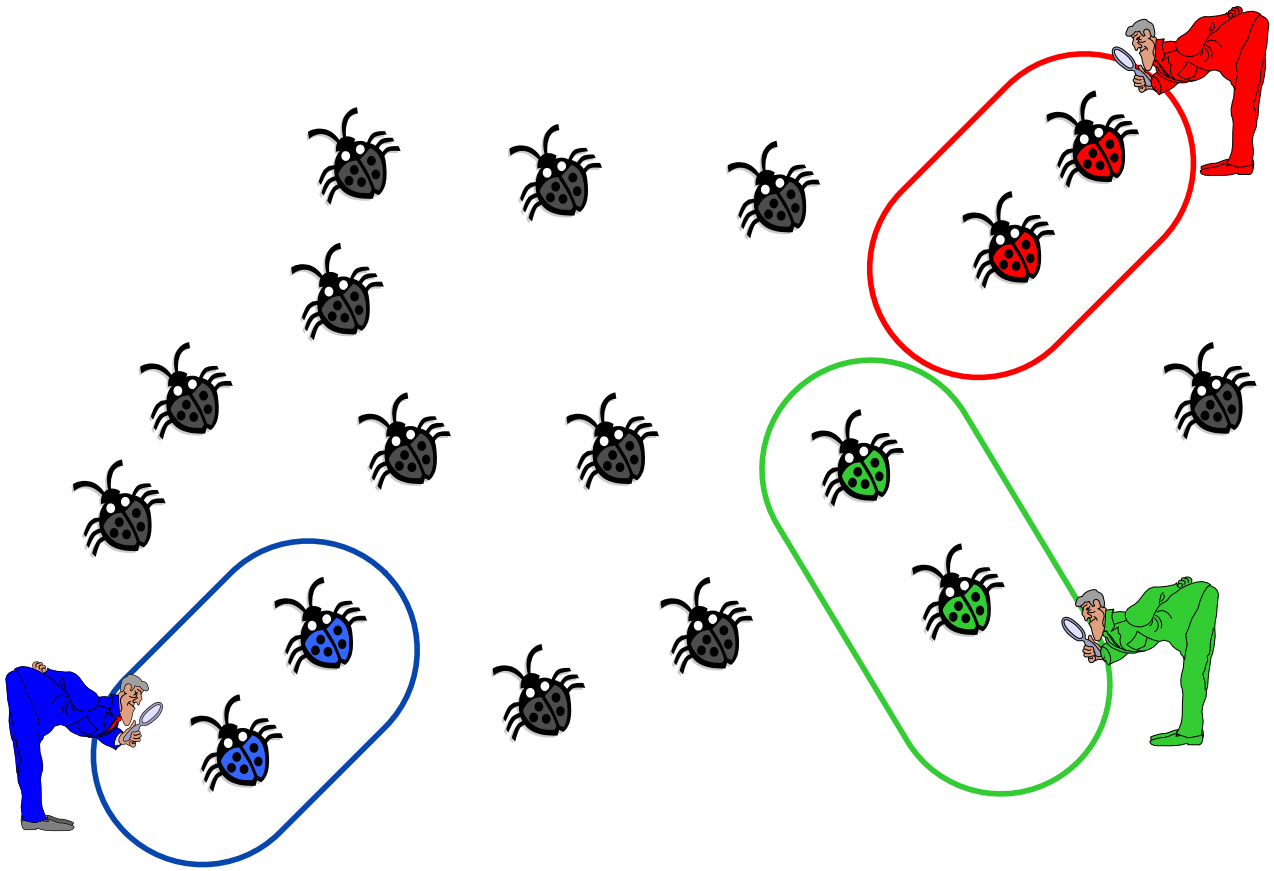
The usage-based reading technique combines traditional inspection principles, use cases, and operational profile testing to create an efficient, user-oriented software inspection reading technique. UBR can find faults more effectively and efficiently than the traditional checklist-based method.

30 IEEE SOFTWARE *Published by the IEEE Computer Society*

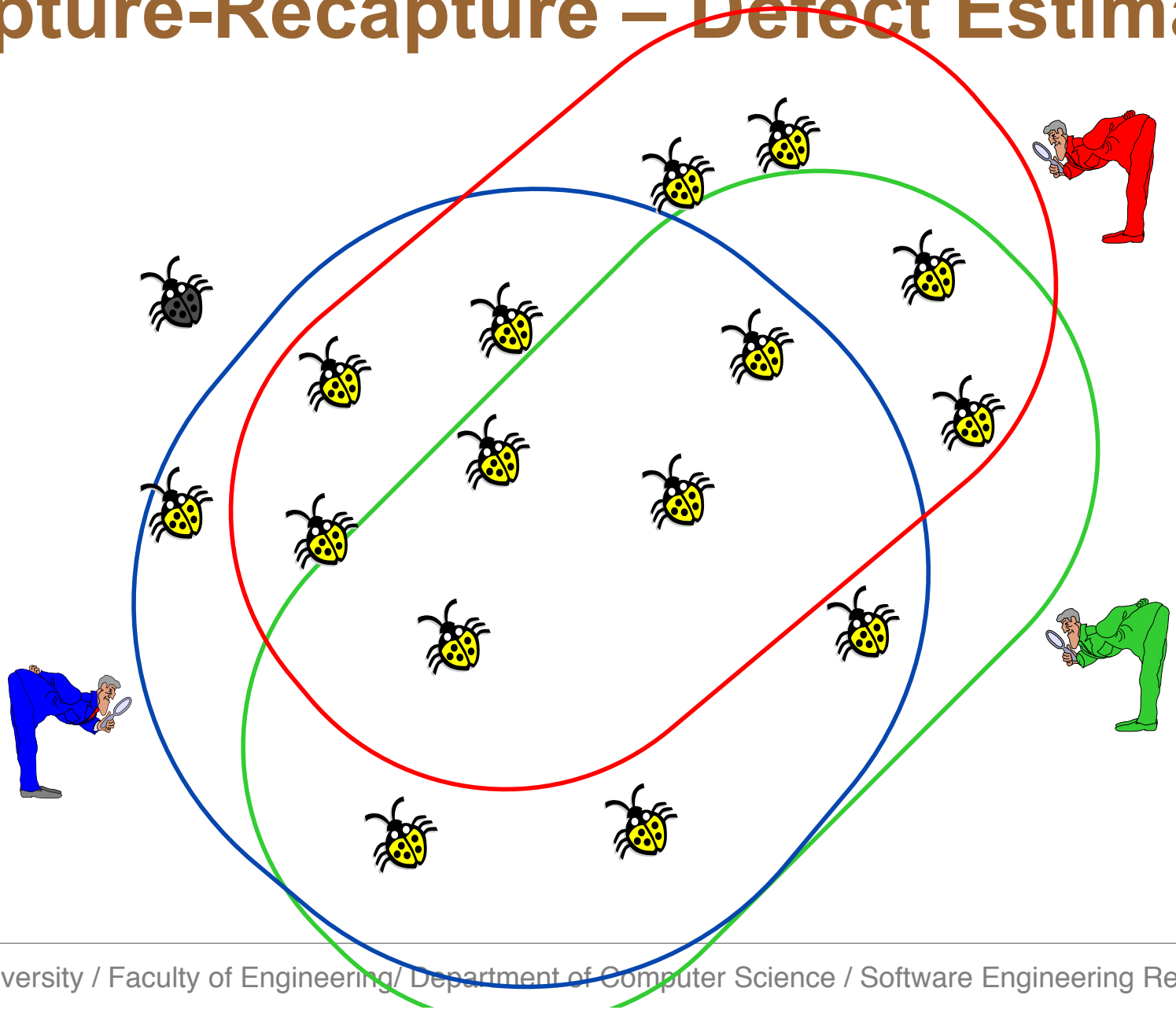
0740-7459/03/$17.00 © 2003 IEEE

Source:
Thelin, T.; Runeson, P..; Wohlin, C.; ”Prioritized Use Cases as a Vehicle for Software Inspections”, *IEEE Software* , vol. 20, no.4, pp. 30-33, July-August 2003

Capture-Recapture – Defect Estimation



Capture-Recapture – Defect Estimation



Capture-Recapture – Defect Estimation

- Situation: Two inspectors are assigned to inspect the same product (Lincoln-Petersen Model)
  - d_1: defects detected by Inspector 1
  - d_2: defects detected by Inspector 2
  - d_{12}: defects detected by Inspector 1 and Inspector 2
  - N_t: total defects (detected and undetected)
  - N_r: remaining defects (undetected)

$$N_t = \frac{d_1 d_2}{d_{12}} \qquad N_r = N_t - (d_1 + d_2 - d_{12})$$

Capture-Recapture – Example

- Situation: Two inspectors are assigned to inspect the same product
  - d_1: 50 defects detected by Inspector 1
  - d_2: 40 defects detected by Inspector 2
  - d_{12}: 20 defects by both inspectors
  - N_t: total defects (detected and undetected)
  - N_r: remaining defects (undetected)

$$N_t = \frac{d_1 d_2}{d_{12}} = \frac{50 \cdot 40}{20} = 100 \qquad N_r = 100 - (50 + 40 - 20) = 30$$

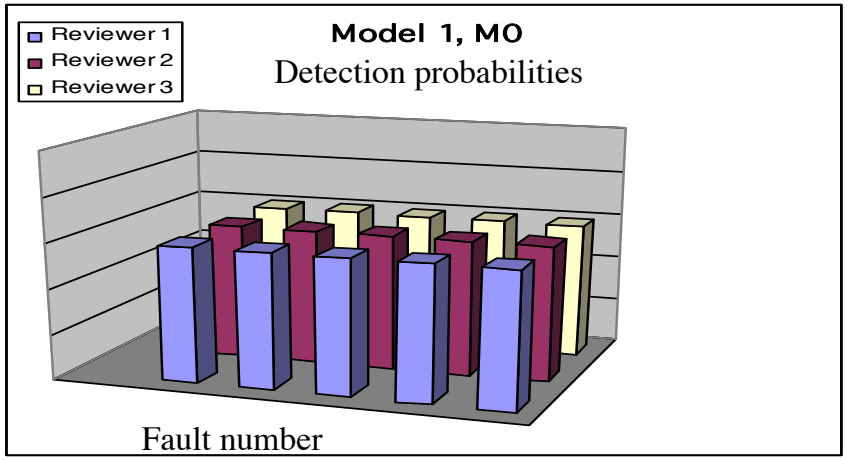
Advanced Capture-Recapture Models

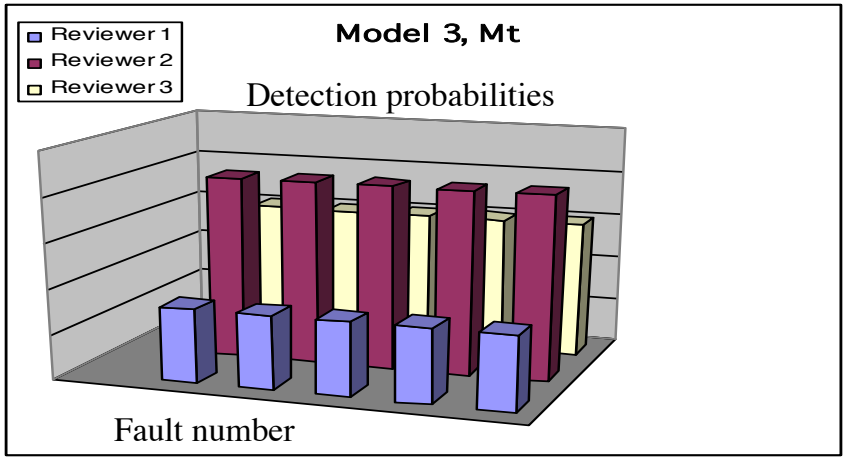
- Four basic models used for inspections
  - Degree of freedom
- Prerequisites for all models
  - All reviewers work independently of each other
  - It is not allowed to inject or remove faults during inspection

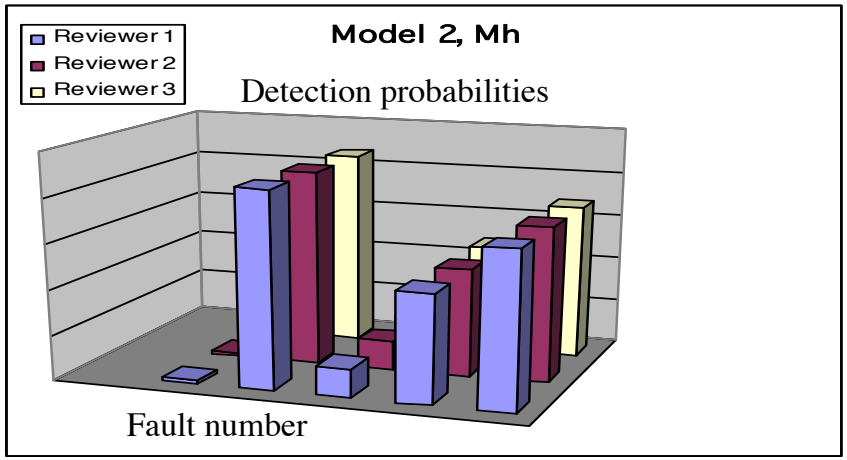
Advanced Capture-Recapture Models

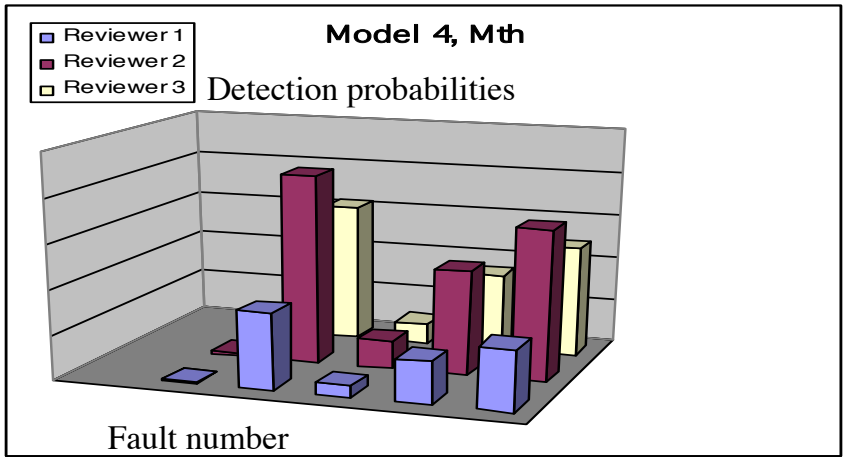
Model	Probability of defect being found is equal across ...		Estimator
	Defect	Reviewer	
M0	Yes	Yes	Maximum-likelihood
Mt	Yes	No	Maximum-likelihood
			Chao's estimator
Mh	No	Yes	Jackknife
			Chao's estimator
Mth	No	No	Chao's estimator

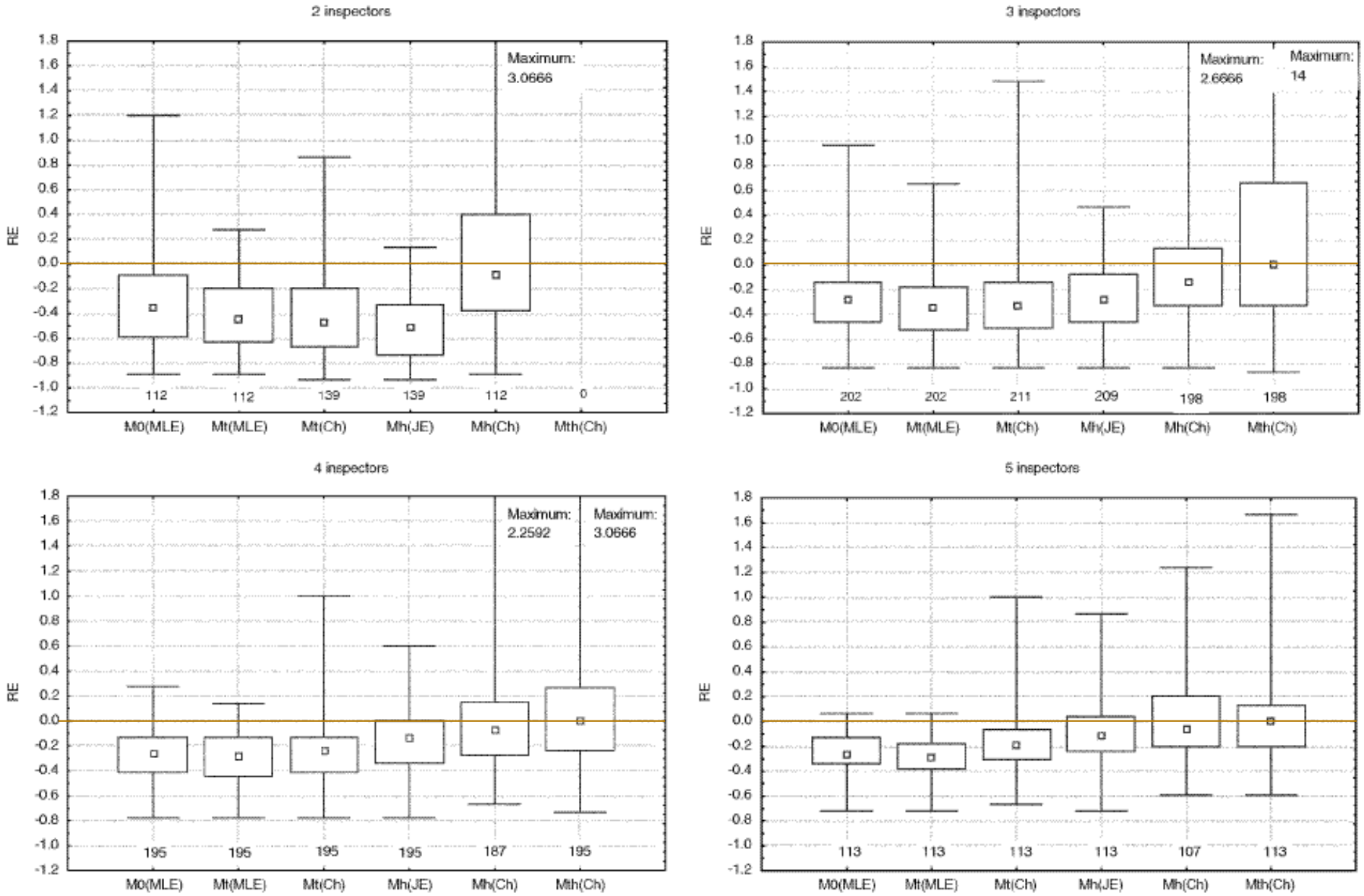
Model prerequisites





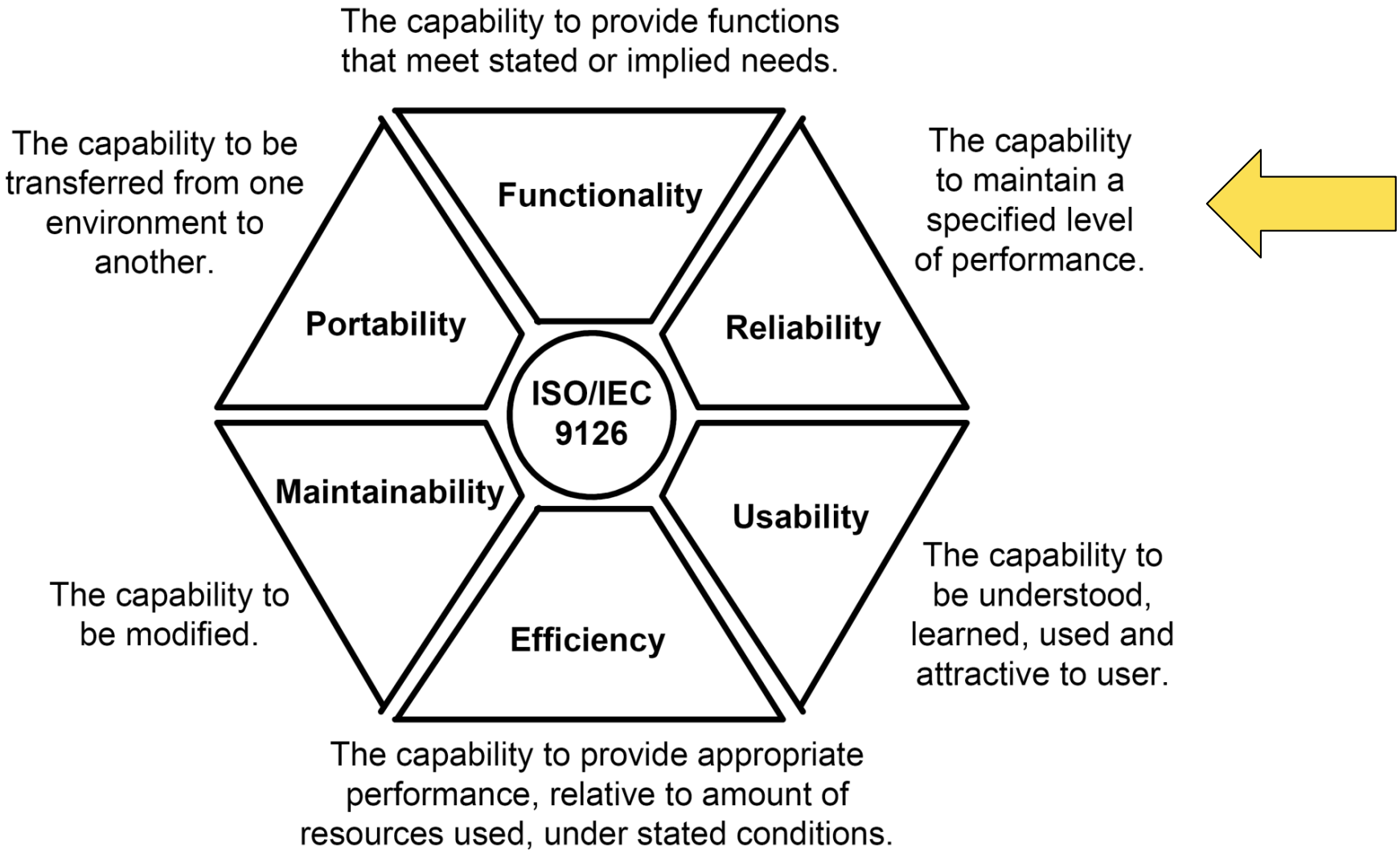





2 inspectors
Maximum: 3.0666
RE
112
112
139
139
112
0
M0(MLE)
Mt(MLE)
Mt(Ch)
Mh(JE)
Mh(Ch)
Mth(Ch)
3 inspectors
Maximum: 2.6666
Maximum: 14
202
202
211
209
198
198
4 inspectors
Maximum: 2.2592
Maximum: 3.0666
195
195
195
195
187
195
5 inspectors
113
113
113
113
107
113

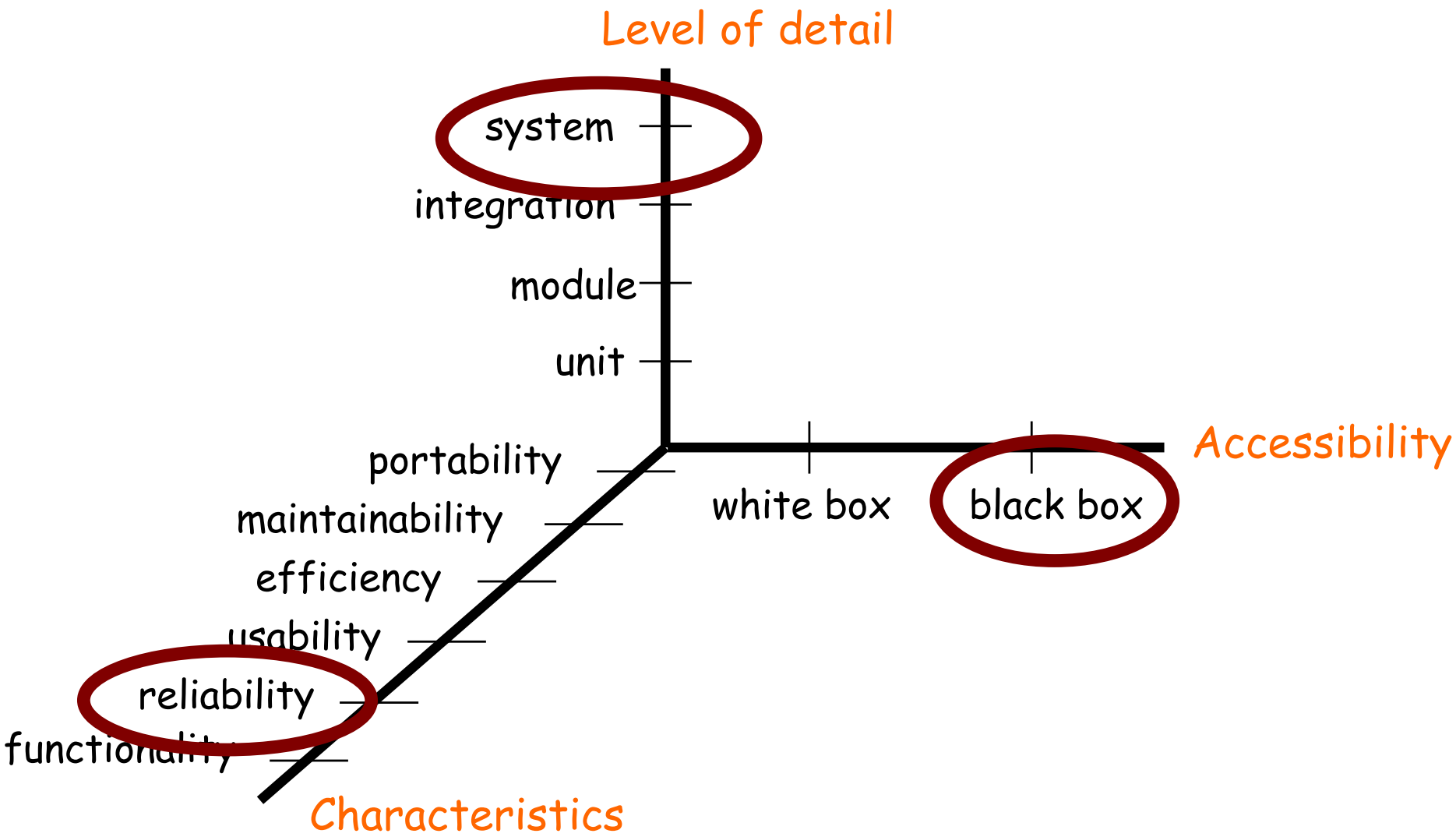
[Briand00]

Quality Attributes – ISO 9126

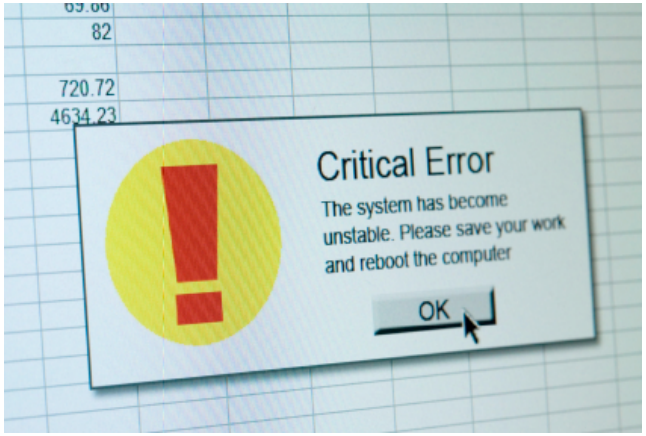


Reliability testing

$$\lambda(\tau) = \lambda_0 \times e^{-\frac{\lambda_0}{\nu_0}\tau}$$

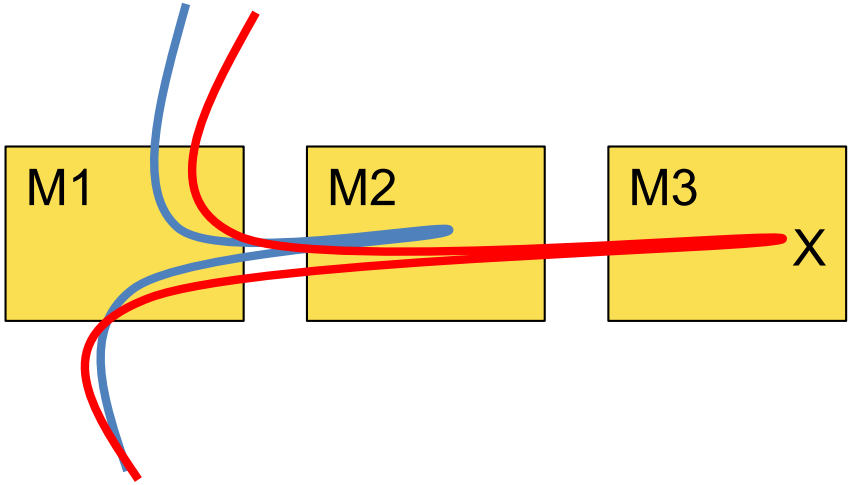


Reliability – Terminology



- **Reliability (1):** The probability of failure-free operation of a software system for a specified time in a specified environment.
- **Reliability (2):** Failure intensity is a measure of the reliability of a software system operating in a given environment.
- **Availability:** Availability is the degree to which a system or component is operational and accessible when required for use.

Reliability depends on usage



—> Sampling from usage to estimate reliability

Time intervals

- Execution time
- Calendar time
- Which is most relevant?

Mean time to failure (MTTF)

Mean time to repair (MTTR)

Mean time between failure (MTBF)

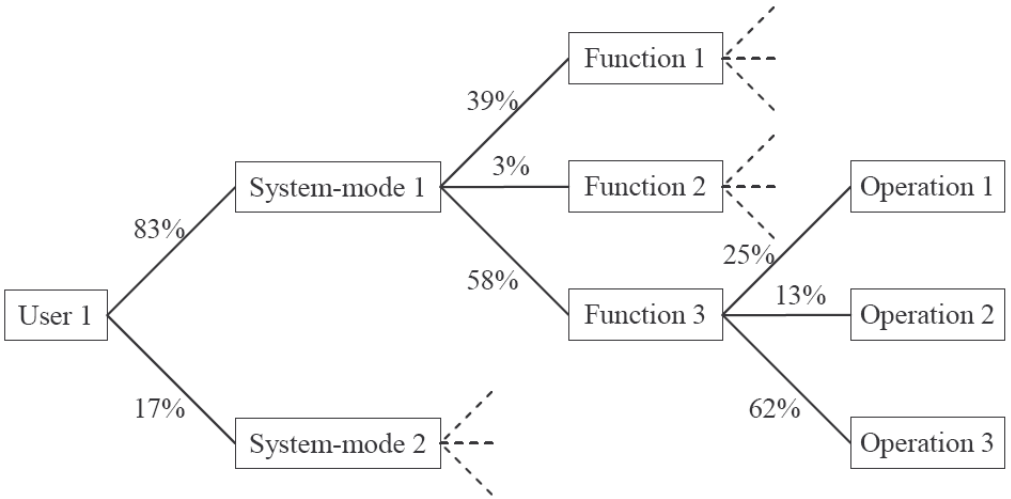
MTBF = MTTF + MTTR

Applications of software reliability

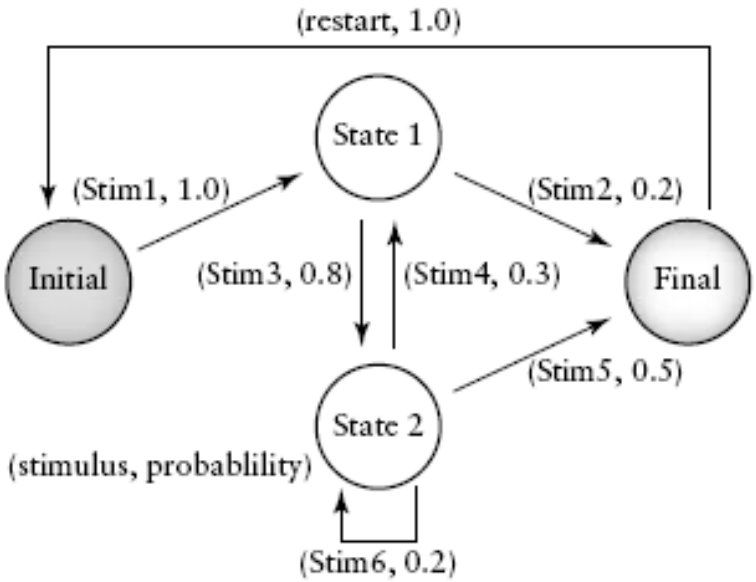
- Measuring the Progress of System Testing
- Controlling the System in Operation
- Better Insight into Software Development Process

Usage Specification Models

Operational Profile



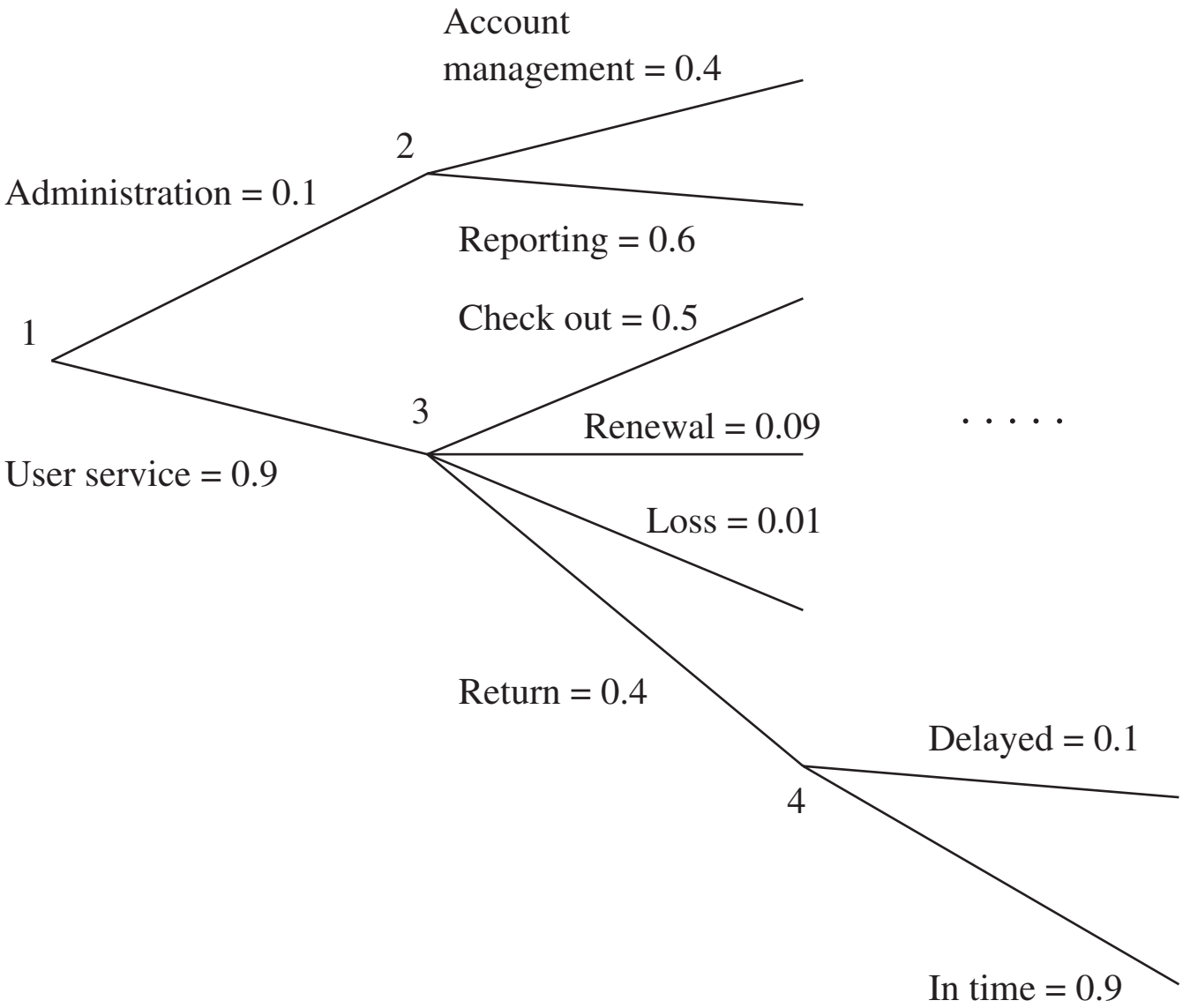
State-Transition Diagram



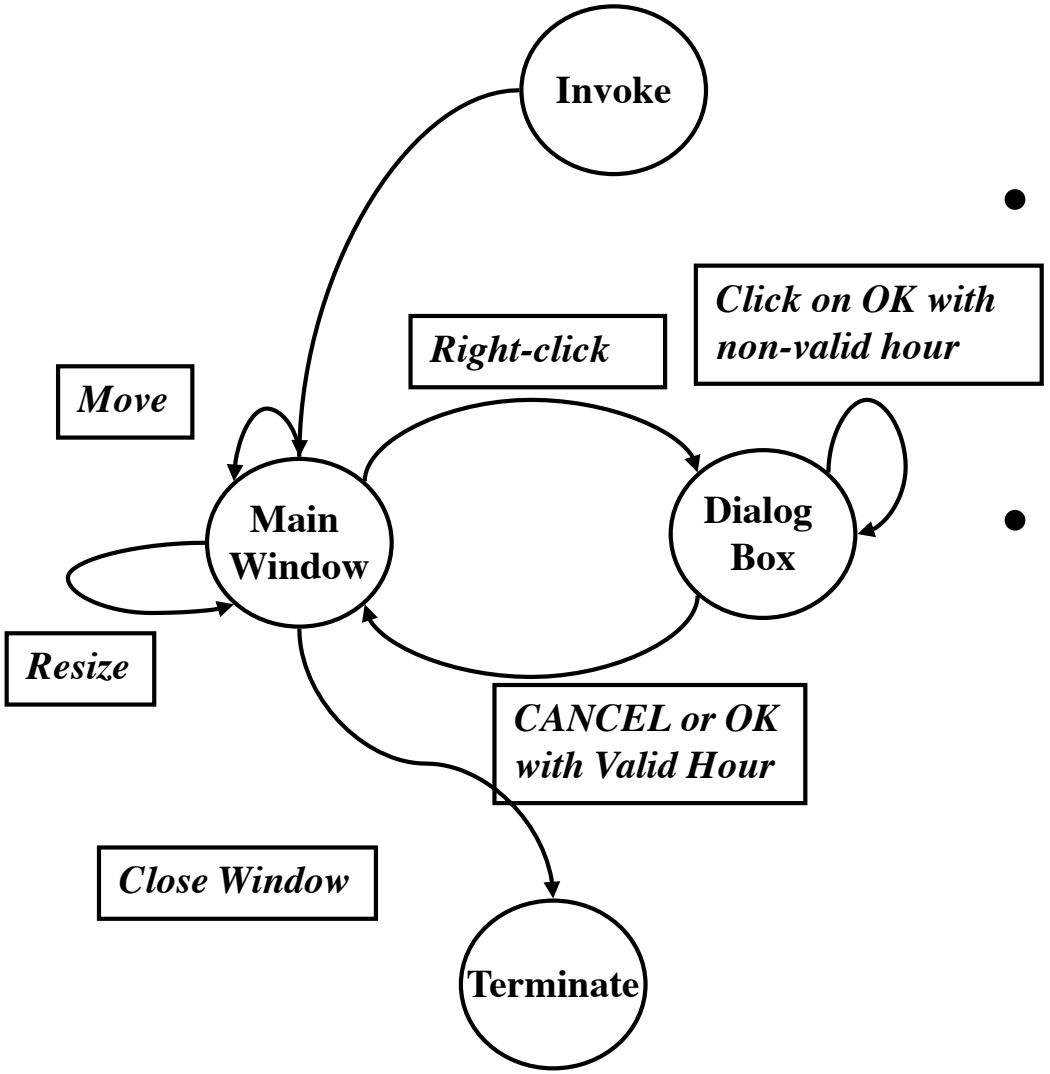
Tabular operational profile

Operation	Operations per Hour	Probability
Book checked out	450	0.45
Book returned in time	324	0.324
Book renewed	81	0.081
Book returned late	36	0.036
Book reported lost	9	0.009
⋮	⋮	⋮
Total	1000	1.0

Graphical operational profile



Usage Modelling

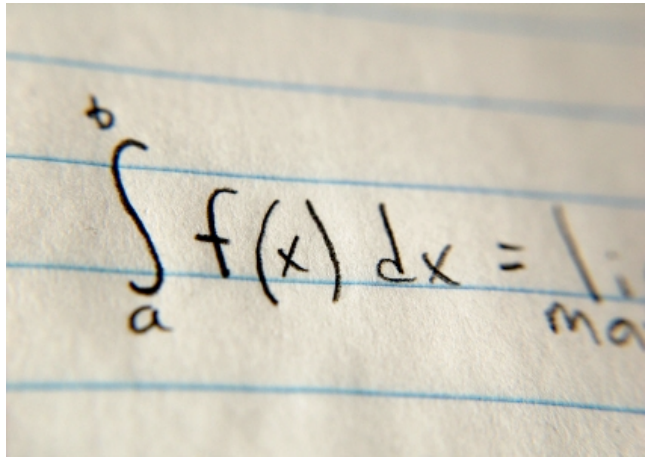


- Each transition corresponds to an external event
- Probabilities are set according to the future use of the system

Other Uses of Operational Profiles

- Guide UI design. More frequently used operations should be easy to learn and easy to use.
- Developing a version of a system for early release. The early-release version can contain the more frequently used operations.
- To determine where to put more resources for system development. Spend more resources on frequenctly used operations.
- Guide organizing user manuals. Frequently used operations are described earlier than the rest.

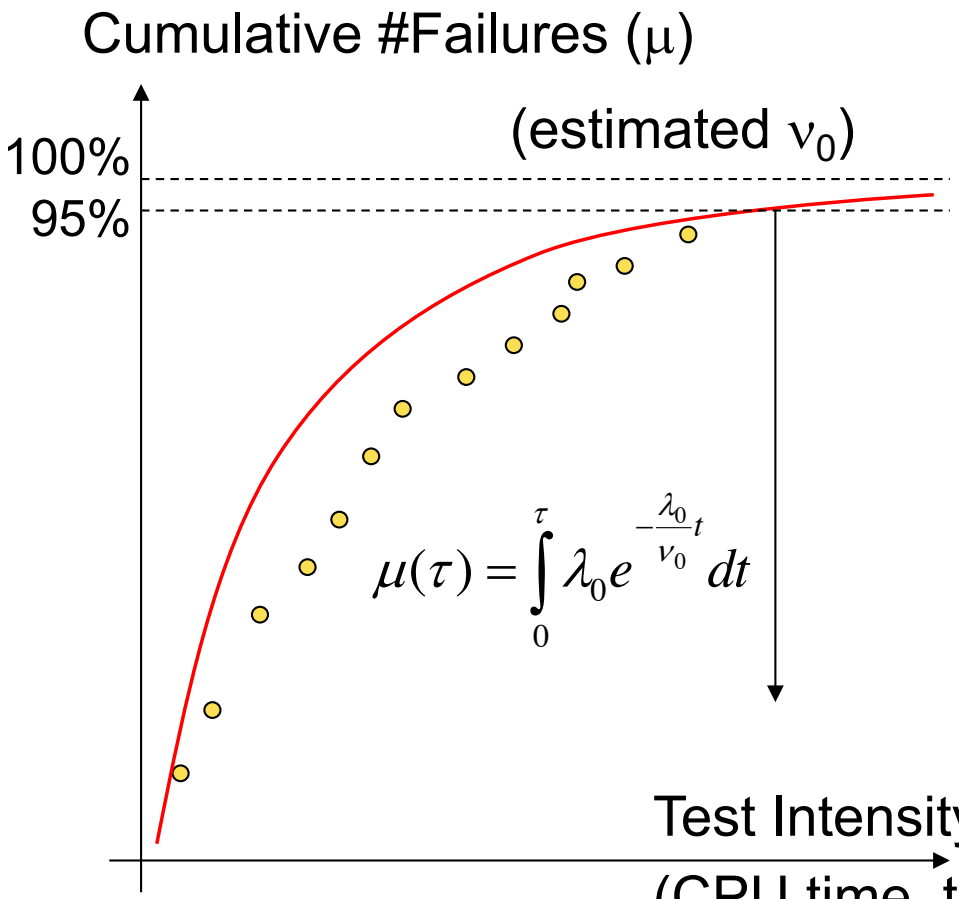
Reliability Growth Models



Purpose:

- To predict the probability of future failure occurrence based on past (observed) failure occurrence
- Can be used to estimate
  - the number of residual (remaining) faults or
  - the remaining test time until a reliability objective is achieved or
  - the time until the next failure occurs
- Application typically during system test

Reliability Growth Models (RGMs)



Test Intensity (τ)

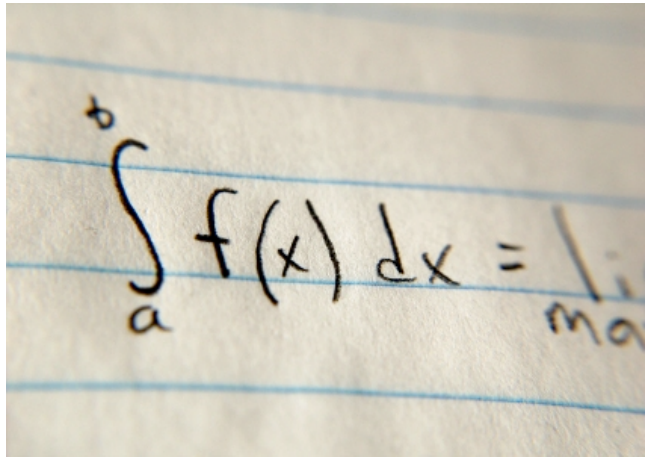
(CPU time, test effort, test days, calendar days, …)

<u>Purpose:</u>

Stop testing when

- a certain percentage (90%, 95%, 99%, 99.9%, …) of estimated total number of failures has been reached
- a certain failure rate has been reached

Many Different RGMs



- Basic Exponential model (Musa)
- Logarithmic Poisson model (Musa-Okumoto)
- Jelinski-Moranda model (considers perfect correction)
- Littlewood-Verrall model
- Goel-Okumoto model
- Delayed S-Shaped model (considers learning during test process)
- Inflection S-Shaped model (considers dependencies between defects)
- …

Textbook Examples

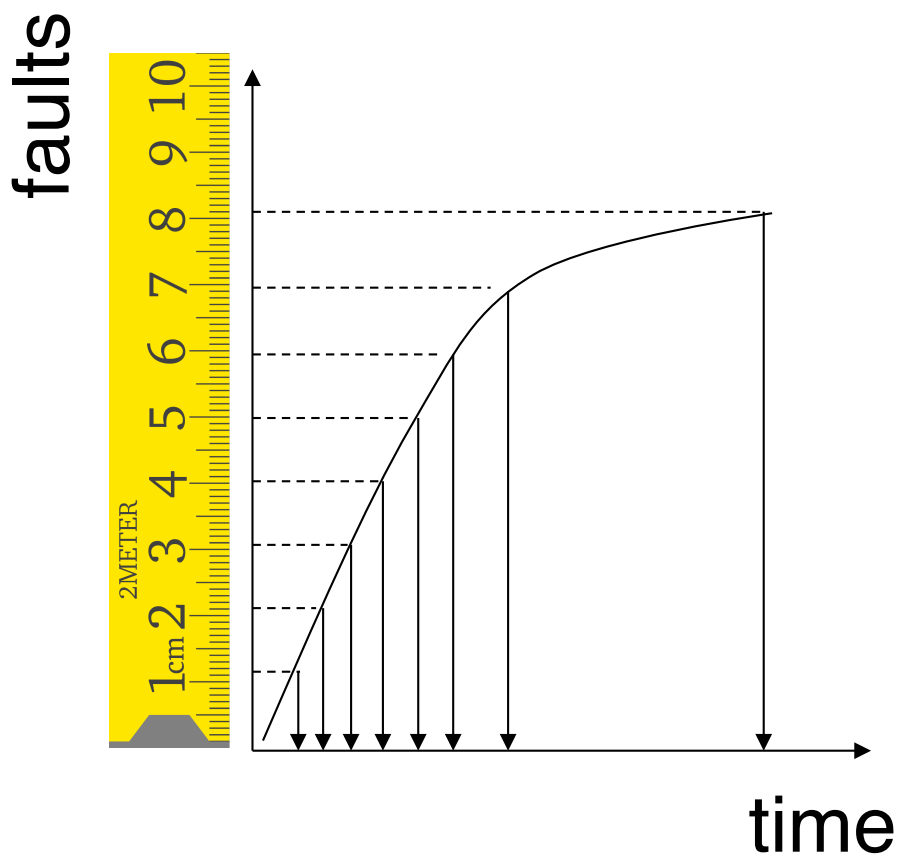
More than 40 models exist!

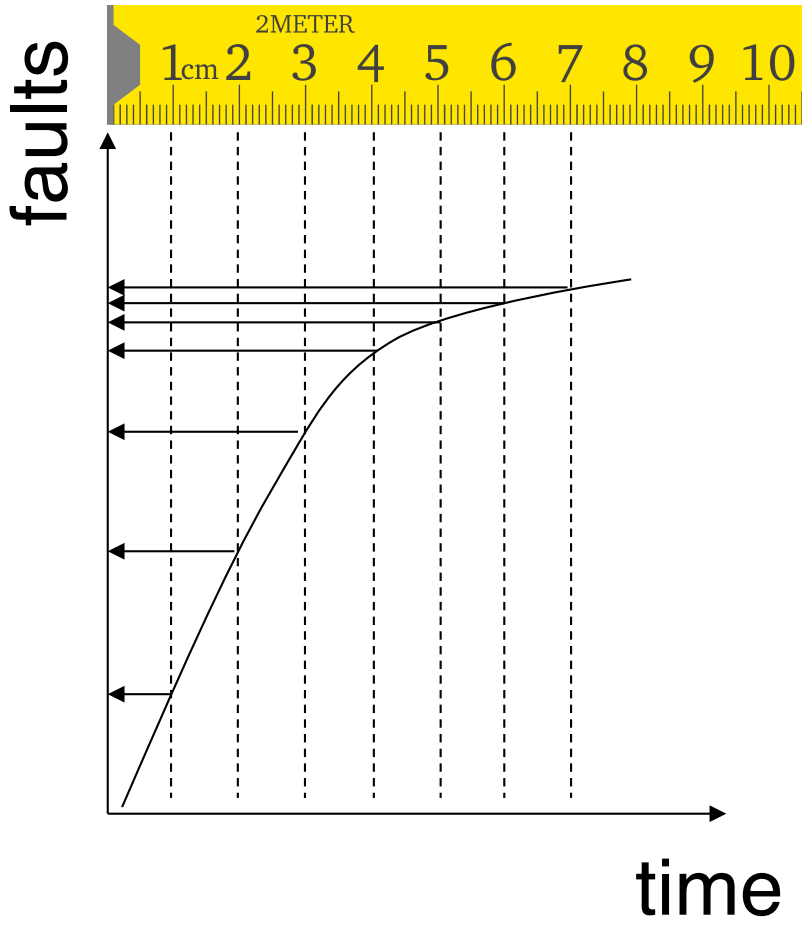
Different models have different underlying assumptions and thus different properties

2 main categories:

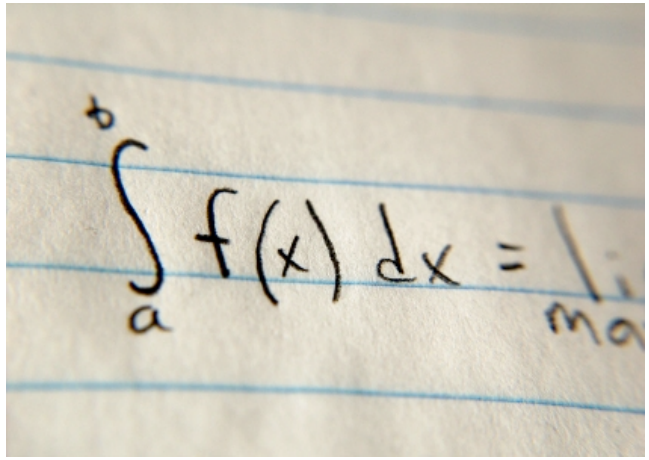
- Time between failure models
- Failure count models

Failure Data Format /3





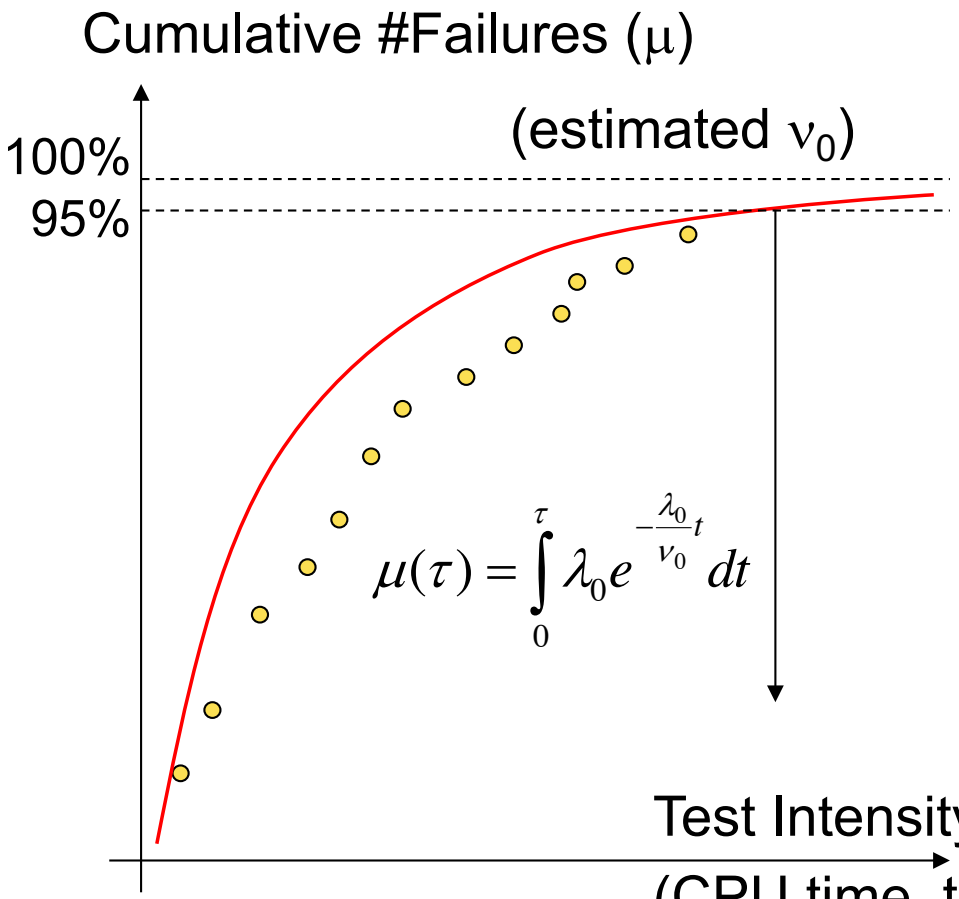
RGMs – Parameters



- Typical parameters involved in RGMs:
  - Mean number of failures (μ): expected number of failures experienced in a test period (or time interval [0, t])
  - Failure intensity (λ): number of failures per test intensity unit (or time unit); $\lambda(t) = \mu'(t) = \mu(t)*d/dt$
  - Test intensity (τ): test time or effort spent since start of testing (e.g., execution time spent since the program is running under test)

Note: Execution time (CPU time) is different from calendar time or test time (or effort)

Reliability Growth Models (RGMs)



Test Intensity (τ)

(CPU time, test effort, test days, calendar days, …)

<u>Purpose:</u>

Stop testing when

- a certain percentage (90%, 95%, 99%, 99.9%, …) of estimated total number of failures has been reached
- a certain failure rate has been reached

Problem: unstable predictions

JM & GO Example (Kim et al. 2007)

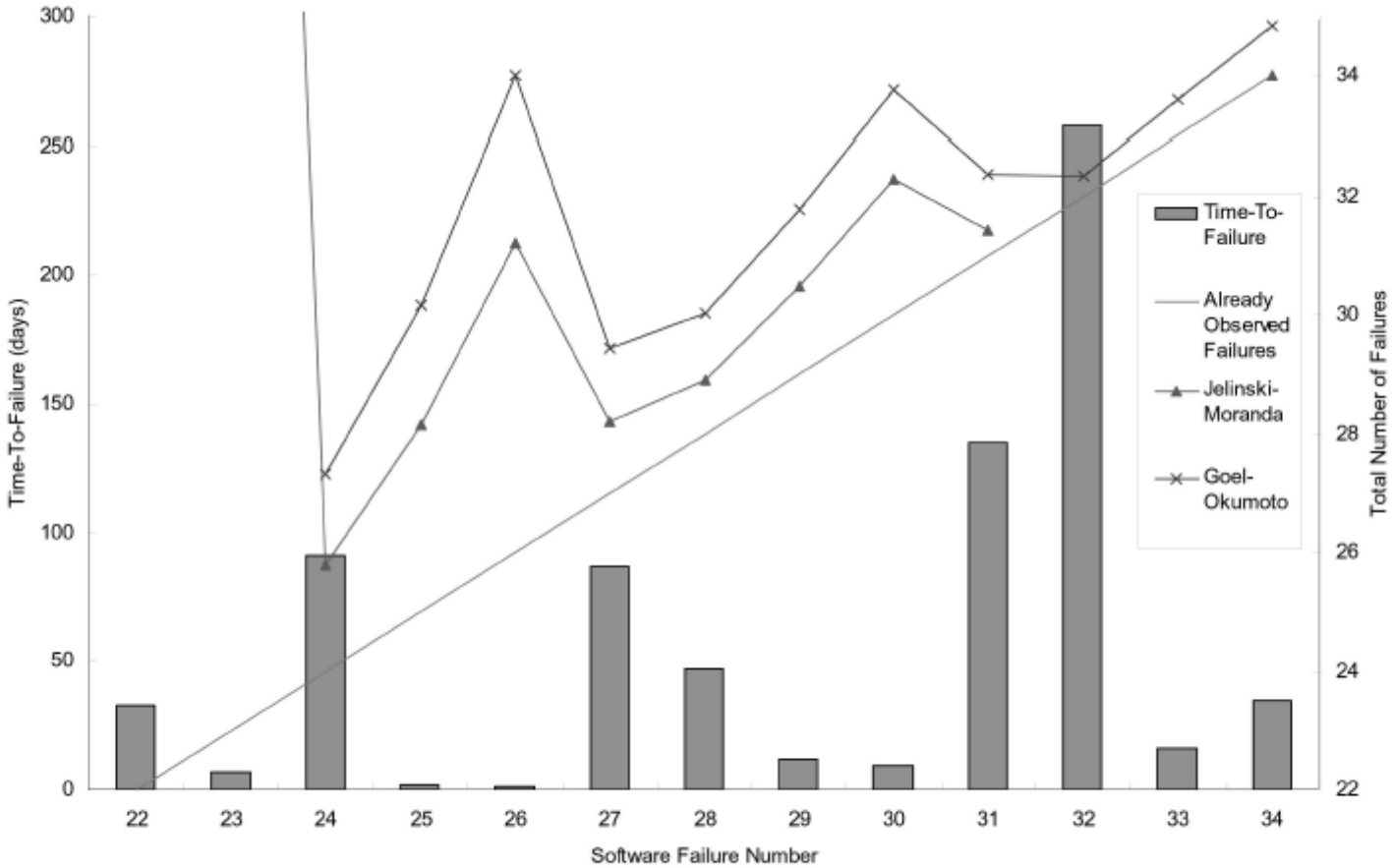


Fig. 1. Change of Estimated Total Number of Inherent Software Faults Calculated by the Jelinski-Moranda Model and the Goel-Okumoto NHPP Model

Recommended exercises

- Chapter 3
  - 1, 2, 3, 5, 7
- Chapter 15
  - 2, 3, 5, 8, 9