Software reliability

Software reliability

- Software reliability:
  probability of failure-free software operation for a specified period of time in a specified environment

- Transaction reliability measure = probability that a single transaction will not fail
  - assumes typical operations
  - the input obeys the operational profile

- Software reliability prediction and estimation:
  failure data are properly measured by various means during software development and operational phases

[1] Yashwant K. Malaiya, Software Reliability: A Quantitative Approach.
Chapter 13, https://www.cs.colostate.edu/~malaiya/p/SoftwareReliability.Chapter13.18.pdf
Book: System Reliability Management, 2018

Software Reliability

A fault (bug or defect) causes a failure when the code containing it is executed, and the resulting error propagates to the output.

Bugs are detected and removed during testing, and the reliability of the system increases.

Bug fixes are released as patches updating the sw, resulting in a next version of the sw. After the release of the sw, the reliability does not change, as long as the same operational environment is used and no patches are applied.

The past failure history is an indicator of the reliablity

Results from similar sw systems can also be used

Software life-cycle

Requirement phase:
significant number of bugs detected by code review and inspection

Design phase:
design is revised to find errors

Coding phase:
code analysed in teams to identify errors

Testing phase: can take 30%-60% of the overall development effort

- *Unit testing*
- *Integration testing*
- *System testing*

  the inputs should be drawn uniformly from the input partitions other than using the frequencies that would be encountered during actual operations

- *Acceptance testing*

  assess reliability encountered in the actual operational environment
  estimating the frequencies describing how the actual users use the system

Software life-cycle

Operational use:

sw released when the reliability requirement is satisfied
the bugs detected by the users are recorded. Fixed in a patch

Maintenance phase:

regression testing after major modification to the sw

The fraction of total faults introduced and found during a phase depends on the specific development process used

Software life-cycle

Phase	Defects (%)		
	Introduced	Found	Still Remaining
Requirements	10	5	5
Design	35	15	25
Coding	45	30	40
Unit testing	5	25	20
Integration testing	2	12	10
System testing	1	10	1

small number of faults, but these are the remaining defetcts and much effort is required to be found

From: Yashwant K. Malaiya, Software Reliability: A Quantitative Approach.

Failure rate and Fault density

We distinguis between

failire rate during testing λ_t

failure rate during normal operation λ_{op}

$$\lambda_t > \lambda_{op}$$

Effectiveness of testing strategy: $\frac{\lambda_t}{\lambda_{op}}$

(10 means that testing is 10 times more efficient to find bugs than normal use)

Fault density: number of faults for 1.000 source line of code (KSLOC)

Failure rate can be assumed to be proportional to fault density

Target fault density for critical systems: 0.1 defects for KLOC

for other applications: 0.5 defects for KLOC

Fault density and failure rate

Test coverage metrics (how a code has been thoroughly exercited by a test suite)

- statement coverage (fraction of statements executed during testing)
  100% relatively easy to achieve

- branch coverage (fraction of branches executed during testing)
  85%-90% is used as an acceptable criterion
  exponentially more difficult to reach higher level in terms of number of test cases needed

Fault detectability

Fault detectability

probability that the fault will be tested by a random chosen test

Some faults can be easily detected, other faults are hard to find
As testing progresses the remaining faults are harder to find

Faults found during **limited** testing are not representative of all the remaining faults
(they are the easier-to-find-faults)

Detectability of a fault depends on

- how frequently the site of the fault is traversed and
- how easy the error propagates to the output

Example: faults in a code, which is executed only when an error is encountered and a recovery is attempted, have low probability of discovery with normal usage or with random testing.

Software reliability models

There are basically two types of software reliability models

1) prediction models (static measures)
"fault density" models: number of faults for KLOC
complexity metrics
attempt to predict software reliability from **design parameters**
use code characteristics such as line of codes,
nesting loops, input/output, …

2) estimation models (dynamic measures)
during the testing phase
characterizing occurrence of failures and corrections
usage profile & environment
Software Reliability Growth Models (SRGM) provide usefull information
for developers and testers during the testing/debugging phase

Fault density

According to [1], fault density depends from different factors

- F_{ph} : Test Phase factors
- F_{pt} : Programming team factor (capability of the sw development team)
- F_m : Maturity factor of the development process
- F_s : Structure factor of the sw
- F_{cc} : Code churn factor (the impact of changes in the code caused by changes in the requirements)
- F_{ru} : Code reuse factor , which considers the influence of code reuse (fault density lower if most of the code is reused)

$$D = C \cdot F_{ph} \cdot F_{pt} \cdot F_m \cdot F_s \cdot F_{cc} \cdot F_{ru}$$

C is a constant of proportionality representing the default fault density per KSLOC
C estimated using past data from the same company on similar projects

The beginning of test phase is taken as default in the overall model.
Works suggest C in the range 6-20 defects for KSLOC (consider a lower and upper estimate for predictions)

Fault density

F_{ph} : Test Phase factors
at the beginning of the phase :
Unit testing: 4 , Subsystem testing: 2.5, System testing: 1, Operation: 0.35

F_{pt} : Programming team factor
Number of defects related to programmer experience
Strong: 0.4, Average 1, Weak: 2.5
Personal discipline
Understanding of the problem domain

F_m : Maturity factor
Maturity of the software process at an organisation
Initial: 1.5, Repeatable: 1, Defined: 0.4, Managed: 0.1, Optimizing: 0.05

Fault density

F_{cc} : Code churn factor considers
f_c : fraction of changed code
t_c : time at which the change occurs

β is a constant
The impact is higher when the change occurs later

$$F_{cc}(t_c) = (1 - f_c) + f_c\, e^{\beta\, tc}$$

F_s : Structure factor of the sw considers various aspects of the sw structure
- program complexity
- language type (high level languages and the fraction of code in assembly – higher fault density for code written in assembly – 40% more faults),
- module size (a module can be a function, a class, a file, …). Very small size, higher fraction of faults related to the interaction among modules

Code reuse factor F_{ru}

fault density lower if most of the code is reused (has alredy passed testing phase)

What about testing approaches

Better result with combinations of
- black box + white box
- random testing + partition testing (input space divided into partitions defined using the ranges of input variables/structures, each partition analyzed thoroughly, partitions exercised randomly, and deterministically for boundery cases)

Test inputs
- chosen in accordance to the operational profile

Operational profile
- set of disjoint operations that a sw performs
- their probability of occurrence (frequences that occur in actual operation)
  [(A1, 0.74), (A2, 0.15), …..]

operation A1 occurs 74% of the time (acceptance test A1 occurs 74% of the time)

Sw used in different settings: operational profile for each setting may be very different ->
Input space needs to be divided into sufficient small partitions

Input domain Software Reliability Models

Let I be the set of possible inputs, and I_e the set of inputs that lead to a failure
Reliability is given by:

$$R = 1 - I_e / I$$

Let the input space divided into j=1, …, k partitions, with pj the probability that the input belongs to partition j as given by the operational profile

$$R = 1 - \Sigma_{j=1, \ldots, k} \; pj \; * \; I_{ej} / I_j$$

I_j be the set of possible inputs in partition j, and I_{ej} the set of inputs that lead to a failure

(this is an approximation because faults in different partitions are not statistically independent)

[Nelson] T. A. Thayer, M. Lipow and E. C. Nelson, *Software Reliability*, North-Holland Publishing, TRW Series of Software Technology, Amsterdam, 1978.

Software Reliability Growth Models

When the software is tested and debugged, the reliability grows with testing time *t* and faults are removed.

Perfect debugging is assumed: a detected fault is always removed

Testing time measured in terms of
(i) CPU execution time;
(ii) number of man-hours or days;
(iii) number of transactions encountered

More than 250 models developed. There is no one model that is best in all situations.

We consider the *exponential model*, which represents several models.

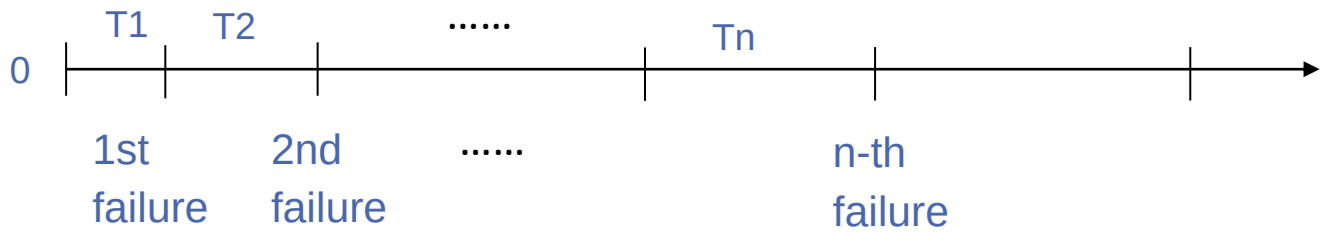
Reliability Growth characterization: Time Between Failures

continuous time reliability growth

Assume times between successive failures are modeled by random variables T1, ..., Tn

T1, time to the first failure

Ti, i>1, time between failure i-1 and failure i



Reliability growth: $T_i <=_{st} T_k$ for all $i < k$

Prob {Ti < x} >= Prob {Tk <= x} forall i < k and for all x

Jelinski and Moranda Model

(the earliest and the most commonly used model)

Assumptions

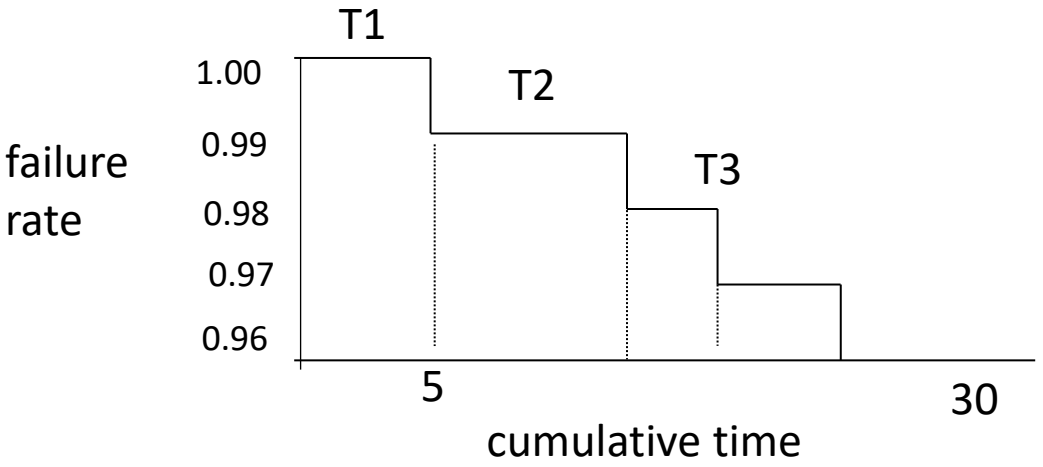
- Fixed number N(0) of faults at the beginning
- time intervals between sw failures are indipendent and exponentially distributed
- detected faults are removed in a negligible time and no new faults are introduced
- the failure rate is proportional to the current number of faults in the sw

Let Ti be the i-th failure interval: the sw has the following constant failure rate, with
ϕ the constant failure intensity contributed by each failure

$$\lambda(ti) = \phi(N(0) - (i-1))$$

Jelinski and Moranda Model

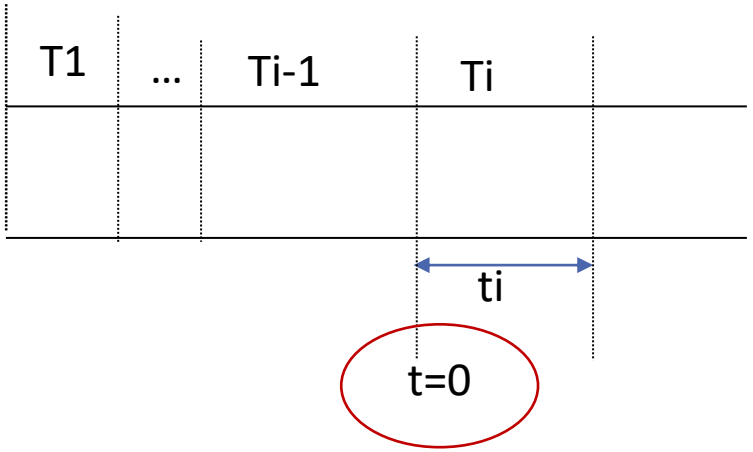
$\lambda(ti) = \phi(N(0) - (i-1))$ with $N(0)=100$ $\phi = 0.01$



Failure rate in the interval T1: 1.0
Failure rate in the interval T2: 0.99
Failure rate in the interval T3: 0.98
.....
λ decreases at step of ϕ

Schick and Wolverton Model

A variation of the previous model
Software failure rate is proportional to the current number of remaining faults and the time elapsed since the last failure



Let t be the time between (i-1)th and i-th failure

$$\lambda(ti) = \phi(N(0)-(i-1))\ ti$$

Failure rate is linear with time within each failure interval.

Failure rate is equal to 0 at the occurrence of a failure and increases linearly again

Reliability growth models

Let **N(t)** be the number of faults in the system at time t

exponential model

At any time, **in the exponential model,** the rate of finding (and removing) faults is proportional to the number of faults still present in the software.
It can be stated as follows:

$$\frac{-\,d\,N(t)}{dt} = \beta_1\, N(t) \quad (**)$$

$N(t+\Delta t) - N(t) <= 0$

β_1 constant of proportionality

$$\beta_1 = \frac{K}{(S * Q * 1/r)}$$

S software size (number source instructions)
Q number of object instructions for source instruction (result from compilation from high level to machine language)
r instruction execution rate for the machine instructions of the computer

k fault exposure ratio (we assume K constant)

if t is measured in sec of CPU execution time, k has been found in the range $1x10^{-7}$ and $10x\ 10^{-7}$

Reliability growth models

The differential equation (**) can be solved as

$$N(t) = N(0)\, e^{-\beta 1 t}$$

with N(0) the number of faults before the testing.

Total number of expected faults detected at time t

$$\mu(t) = N(0) - N(t) = N(0)\,(1 - e^{-\beta 1 t})$$

Generally written as:

$$\mu(t) = \beta_0\,(1 - e^{-\beta 1 t})$$

with β_0 equals to N(0)

Reliability growth models

Failure rate in terms of β_0 and β_1

$$\lambda(t) = \frac{d}{dt}\mu(t) \qquad \mu(t) = \beta_0 (1 - e^{-\beta 1 t})$$

$$\lambda(t) = \beta_0 \beta_1 e^{-\beta 1 t}$$

In general, different sections of the sw can have different parameters

Using the data from different projects, it has been shown that the exponential model has good capability of prediction

Another model that has been shown to have better capability of prediction is the Musa and Okumoto logarithmic model

Reliability growth models

How software reliability growth models can be used?

Case 1.
SRGM can be used before testing begins to estimate the effort needed to achieve the desired reliability
Assumption: an estimation of the parameters is known before testing activity (using static attributes like software size)

Case 2.
SRGM can be used after testing and debugging.
Actual test data can be collected and used to estimate the parameters values and predict the additional testing time needed to reach the required level of reliability.

Reliability growth models

Example (case 1)

Initial fault density: 25 faults / KSLOC
Software size: 10.000 line of C
Code expansion ratio for target CPU: Q= 2.5 (compiled code: 25.000 lines)
CPU rate during testing: 70 MIPS

$K = 4 \times 10^{-7}$

What is the testing time to achieve fault density of 2.5 faults/KSLOC?
CPU testing time, person-hours, ….

Use (i) equation $\beta_1 = \frac{K}{(S * Q * 1/r)}$ (iii) $N(t) = N(0)\, e^{-\beta 1 t}$

(ii) fault density $\beta_0 = N(0)$

Reliability growth models

Exponential model.

$\beta_0 = N(0) = 25 \times 10 = 250$ faults

$$\beta_1 = \frac{K}{(S * Q * 1/r)} = \frac{4.0 \times 10^{-7}}{10.000 \times 2.5 \times (70 \times 10^6)} = 11.2 \times 10^{-4} \text{ per second}$$

Let t1 the testing time required to achieve fault density equal to 2.5 /KLOC

$$N(t) = N(0)\, e^{-\beta 1 t} \qquad N(t1) = N(0) \exp(-11.2 \times 10^{-4}\, t1) \qquad \frac{N(t1)}{N(t0)} = \exp(-11.2 \times 10^{-4}\, t1)$$

$$\frac{N(t1)}{N(t0)} = \frac{2.5 \times 10}{25 \times 10} = 10^{-1} \qquad \exp(-11.2 \times 10^{-4}\, t1) = 10^{-1}$$

$$t1 = \frac{-\ln(0.1)}{11.2 \times 10^{-4}} = 2056 \text{ seconds CPU time}$$

$$\lambda(t) = \beta_0\, \beta_1\, e^{-\beta 1 t}$$

$$\lambda(t1) = 250 \times 11.2 \times 10^{-4} \exp(-11.2 \times 10^{-4}\, t1)$$

Reliability growth models

Consider the same example, with the assumptions that:

- K is not known
- from previous project, it has been estimated that:
  with 10 KLOC source code, the final value for $\beta_1 = 2x10^{-3}$ per second

What is the value of for β_1 for 20 KLOC?

Since K does not depend on the source code size (sec of CPU execution time), for 20/KLOC, it is:

$\beta_1(S \; Q \; (1/r)) = \beta' (S' \; Q \; (1/r))$ $\beta_1(S) = \beta' (S')$

$2x10^{-3}$ 10 KLOC = β' 20 KLOC

$1x10^{-3} = \beta'$

For 20/KLOC, the value of β_1 is equal to $1x10^{-3}$ per second

Reliability growth models

SRGM assume that the testing strategy is uniform throughout testing.
Let us consider the case in which different approaches are used.

Each approach is initially very efficient for detecting a specific class of faults.
We have a spike in failure intensity (**N(t)**). This must be considered to minimize the errors in the model.

What about reliability of systems that consists of multiple components developed and tested separately by different teams? Or components reused by a previous version?

Multiple-version programming

3 versions + voter

Case 1: failures in the 3 versions are independent
Let p be the probability that one version fails

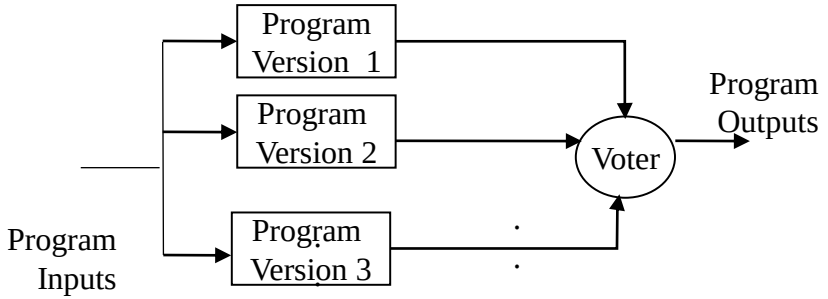
$$P_{sys} = p^3 + 3(1-p)\,p^2$$

Case2:
Let q3 be the probability that all three versions fail for the same input

Let q2 be the probability that any two versions fail for the same input

Probability that the system fails for a transation is:

$$P_{sys} = q3 + 3q2$$

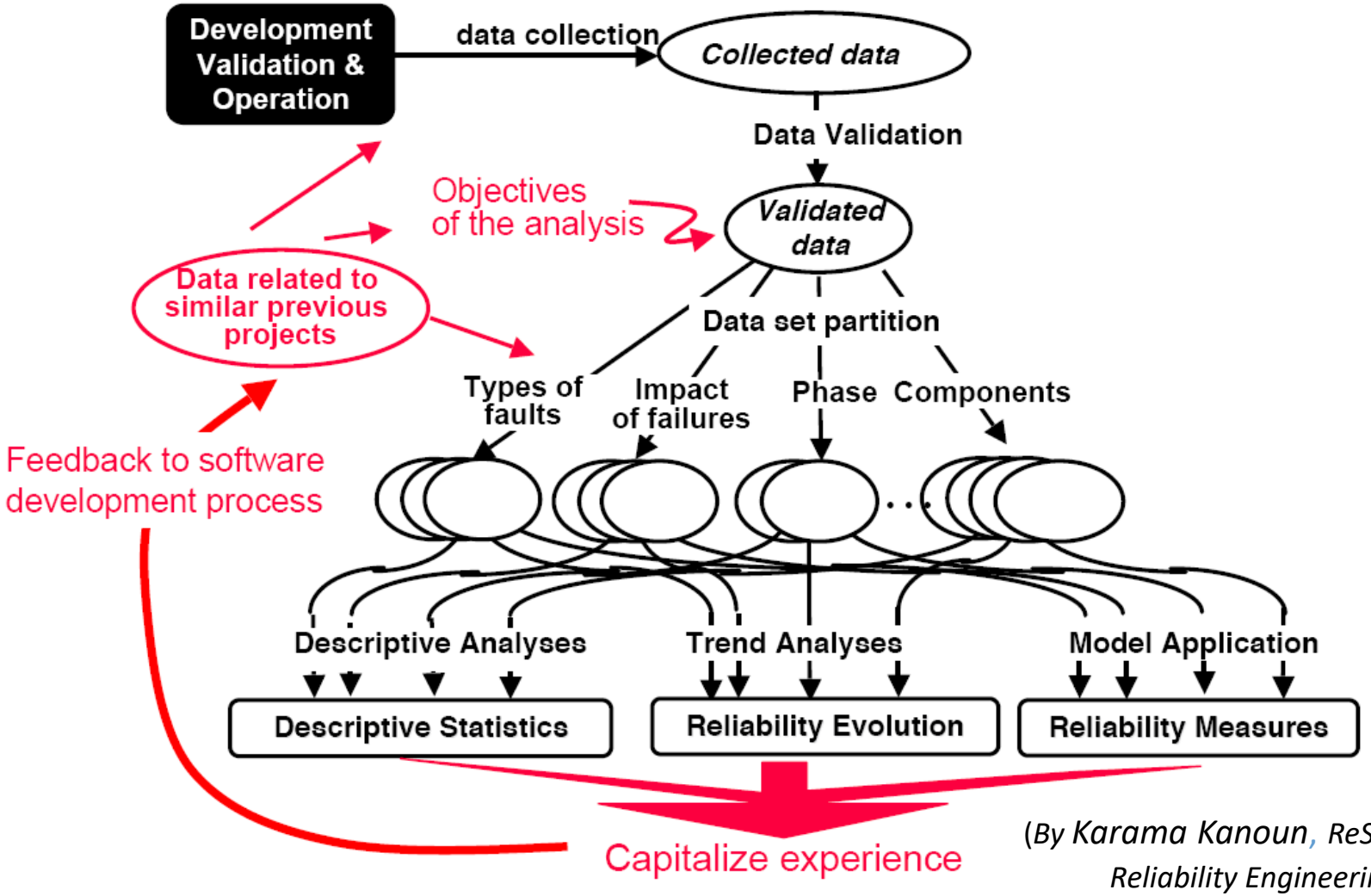


Experimental data
p= 0.0004 q3= 2.5×10^{-7} q2= 2.5×10^{-6}

Case1:
$P_{sys} = 4.8 \times 10^{-7}$
Improvement: $0.0004/4.8 \times 10^{-7} = 833.3$

Case2:
$P_{sys} = 7.75 \times 10^{-6}$
Improvement: $0.0004/4.8 \times 10^{-7} = 51.6$

Software Reliability Engineering (SRE)



Software Reliability Engineering (SRE) is the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability

(*By Karama Kanoun, ReSIST network of Excellence Courseware "Software Reliability Engineering", 2008 http://www.resist-noe.org/*)

Software Reliability Engineering (SRE): Data

Data collection

includes data relative to product itself (software size, language, workload, ...), usage environment, verification & validation methods and data on failures

Generation of *Failure reports* (FR) and *Correction reports* (CR)

Data validation process

data elaborated to eliminate FR reporting of the same failure, FR proposing a correction related to an already existing FR, FR signalling a false or non identified problem, incomplete FRs or FRs containing inconsistent data (Unusable) …

Data extracted from FRs and CRs

Time to failures (or between failures)

Number of failures per unit of time

Cumulative number of failures

……..

Databases with software failure rates are available but numbers should be used with caution

Software reliability

Descriptive statistics

Make syntheses of the observed phenomena

Analyses Fault typology, Fault density of components, Failure / Fault distribution among software components (new, modified, reused)

Analyses Relationships Fault density / size / complexity;
Nature of faults / components;
Number of components affected by changes made to resolve an FR .
…….

Reliability evolution

Control the efficiency of test activities (Trend tests)

Reliability decrease at the beginning of a new activity: OK

Reliability grow after reliability decrese: OK

Sudden reliability grow caution!

…….

Software Reliability models

Limits of the models:

- do not consider that correct a bug may introduce new bugs

- given a design flaw, only some type of inputs will exercise that fault to cause failures.
  Number of failures depend on how often these inputs exercise the sw flaw
  Different errors may contribute differently to the total failure rate

- Failure rate is related to the workload of the system: double workload with the same input distribution leads to double failures

- Models consider only Bohrbugs
  Bohrbug: the system always fails for the same input
  HeisenBug: the system may not fail for the same input
  (e.g., concurrent software, scheduling tasks … or processes may be different on different executions)