# Course Structure

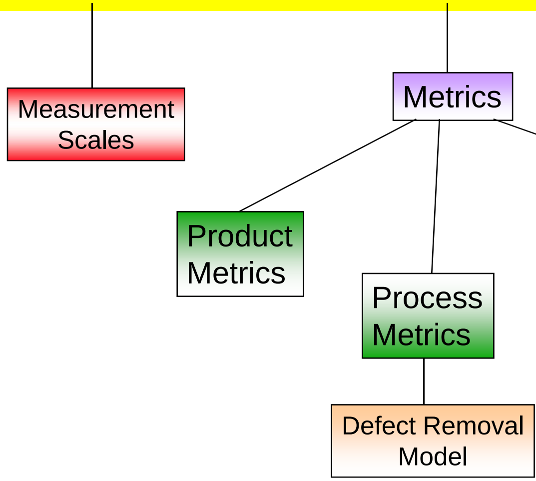
1. Software Quality Assurance
2. Testing Fundamentals
3. Code-based Techniques
4. Specification-based Techniques
5. Inspection Technique
6. Test Tools

**7.1 Measuring Software Quality**

8. TDD



# Measuring Software Quality






# Learning Objectives

- understand the importance of **measurement scales**
- learn 3 classes of metrics: **product, process,** and **resource metrics**
- identify a set of key metrics that can be used for monitoring, controlling and improving the testing process and quality of software products



# Is This Good Results?

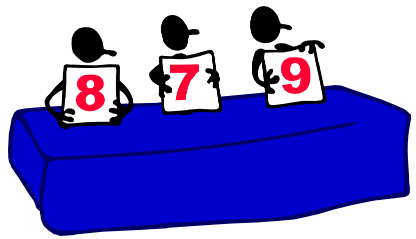
**Introduction**

Scale

Metrics

- ⌘ A team testing 50KLOC of code
- ⌘ Created 1200 test cases
- ⌘ Found 99 defects
- ⌘ Defects were fixed and no failures appeared in further testing

***No more defects in the code?***



# Why Measure? Automobile Example



Most people measure the miles/gallon (kilometer/liter) that their automobile achieves.

Reason?

- ⌘ to evaluate choice, compare alternatives, and monitor improvement
- ⌘ to have early warning of problems, and to make prediction
- ⌘ to benchmark against a standard



# Test Management Need Metrics



Effective management of test process is central to the success of any testing project.

To be effective, managers must have access to the right information for making the crucial decisions.

Manager need to answer:

- Is the system ready to go live?
- If I go live now what risk is associated with that?
- What coverage have we achieved in our testing to date?
- How much more testing is there to do?
- Can I prove the system is really tested?
- What is the impact of this change and what must be retested?

***"One accurate measurement is worth 1000 expert opinions"*** *Grace Murray Hopper, Rear Admiral, US Navy*

# Measurement Basics



- ⌘ **Measurement:** process of *objective* assignment of numbers to **entities**, to characterize an **attribute**.

**Entity** (object, event)
- attribute1 (feature, property)
- attributes 2, …

*example*

**Source code**
- size
- quality

- ⌘ Each entity is given a number, which tells us about its attribute.

  Example: a program has a line count, which tells us about its size.
- ⌘ ***Objective*** means measurement must be based on a well-defined rule whose results are repeatable; e.g., counting the LOC in a program



# Example Metrics

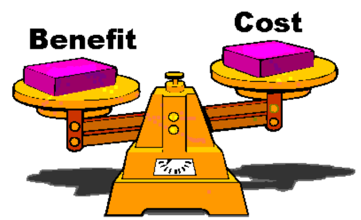


*Measure* → Source code
- *Size*
- *quality*

→ **Metrics:**
*KLOC*
*Defect/KLOC*

| Entity | Attribute | Metric |
|---|---|---|
| Source code | Size | KLOC (line count) |
| Source code | Quality | fault/KLOC |
| Testing process | Duration | time in hours from start to finish |
| Testing process | Efficiency | test/ defect |

# Measurement Scale






## Consider the following grading systems



- ⌘ **Letter Grade**: A+, A, B+, B, C+, C, D, F
- ⌘ **Word Grade**: outstanding, excellent, very good, good, wholly satisfactory, satisfactory, barely adequate, weak, inadequate
- ⌘ **Grade Point**: 4.5, 4, 3.5, 3, 2.5, 2, 1.5, 1, 0

They measure the same attribute:
performance of student



## Measurement Scales



Hierarchy: Ratio, Interval, Ordinal, Nominal

High → Low

Higher scale has the characteristics of a lower scale.

- ⌘ Different scales allow **different manipulations with the data**
- ⌘ Based on the permissible transformation on the value
- ⌘ Parametric statistical techniques may be used only with interval and ratio scales.
- ⌘ Only nonparametric statistical techniques may be used with nominal and ordinal scales.



## Nominal Scale



*Example*:
- student number (e.g., 11055555d)
- categorization of defects into (logical defects, computational defects, and data defects).

- ⌘ Assign "numeric name" to the entity
- ⌘ Divide a set of entities into categories, with *no particular ordering* among them
- ⌘ purely qualitative
- ⌘ We can count how many in each category.
- ⌘ We can identify the ***mode*** (the most frequent value).

# Ordinal Scale



*Example*:
- school grades (e.g., A, B, C),
- categorizing defects with respect to severity level (e.g., 1, 2, 3, 4 and 5)

⌘ Divides the set of entities into categories that are ordered.

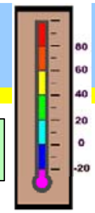
⌘ The distance between scale points is not equal!

⌘ Permitted transformation: any monotonically increasing function (e.g. '>', '<', '=') that preserve the ordering among the values of the measure.

⌘ Can compute the ***median***, rank, and rank correlation coefficients (compare the order, denote preference)



# Interval Scale






*Example*: temperature (Celsius, Fahrenheit)

⌘ Equal distance between scale points.

⌘ A 10-degree difference between 20º and 30º means the same as a 10-degree difference between 40º and 50º.

⌘ The zero point does not indicate an absence of temperature; it is an arbitrary point on the scale.

⌘ Permitted transformation: any positive linear transformation of the form f'=u.f+b, where b is any number (i.e., we can change the origin of the measure) and u>0 (i.e., we can change the unit of the measure).

⌘ Can compute the ***arithmetic means*** and standard deviations



# Ratio Scale (has a zero point)



*Example*:
- length, mass, volume, price, absolute temp (Kelvin),
- LOC as a measure of program size.
- Defect rate

0

⌘ The reference origin is fixed at 0. It includes the total absence of the property (e.g., no length, no defect).

⌘ The **absolute zero** allows us to know how many times greater one case is than another.

⌘ $100 is "twice as much" as $50. No money, no ability to buy anything.

⌘ Can compute the percentage, mean values, standard deviation, quotients, ***geometric mean***.



# A Zen Story



**A men dropped a book in a lake, but only looked for it in the bottom of his boat.**

**When asked why he was looking there for the book, he replied:**

**"Because I can't swim."**

***Lesson: We need to learn about metrics (like learning to swim), and not to avoid measurement.***

# What to Measure?



metrics → Input → metrics → **Test Process** → metrics → **Software product**

Output

We can measure the **test process** and the **output** of the test process and the **input** of the test process.

**3 Types of Metrics**

**Product metrics:** metrics related to test results or the quality (internal characteristics) of the product being tested [related to output of the test process]

**Process metrics:** metrics used to assess the effectiveness of the testing process.

**Resource metrics** (also called **project metrics**): metrics used to assess the cost and productivity of testing [related to input of the test process]

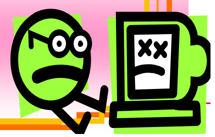
# Product Metrics

***Output***



1 Defect density
2 Defect age
3 Defect response time
4 Defect cost

*Just as people are known by the company they keep, a company will be known by its software and the problems it causes.*



# Process Metrics



Process metrics may be further divided into

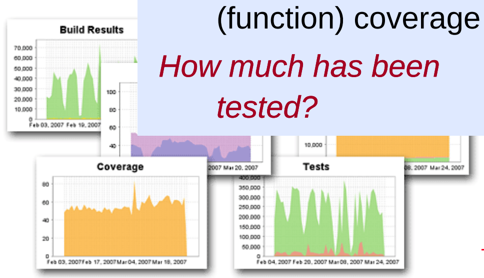
## Coverage Metrics

5 Instruction coverage
6 Path coverage
7 Requirement (function) coverage

*How much has been tested?*

## Effectiveness Metrics

8 %Defects uncovered in testing
9 Test efficiency
10 %Test automation
11 Defect removal effectiveness (DRE)
12 % test cases successfully executed
13 % fixed defects
14 % open defects
15 Maturity of the testing



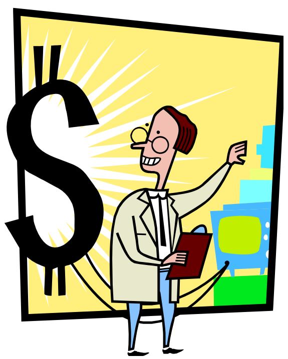



# Resource Metrics



**Resources consumed in testing process and the productivity of testing**

16 Relative test cost
17 Cost to locate defect
18 %Achieving budget
19 % test cases prepared
20 Productivity



# Attributes of Defect/Error/Fault

Introduction

Scale

**Metrics**

- Two key attributes related to defects are the levels of ***priority*** and ***severity***
- A ***priority*** level is a measure of how soon the defect needs to be fixed, i.e., **urgency**.
  - Critical (1), High (2), Medium (3), and Low(4)
- A ***severity*** level is a measure of the extent of the detrimental **effect** of the defect on the operation of the product
  - Critical (1), Major (2), Medium (3), and Low (4)



# Primitive Defect/Error/Fault Metrics

Introduction

Scale

**Metrics**

Primitive metrics are used to derive other metrics:

- Number of faults detected in each **module**
- Number of requirements, design, and coding faults found during unit and integration testing
- Number of errors by **type** (e.g., logic errors, computational, interface, documentation errors)
- Number of errors by **cause or origin** (e.g., communication cause, missing requirements)
- Number of errors by **severity** (e.g., critical, major, low errors)



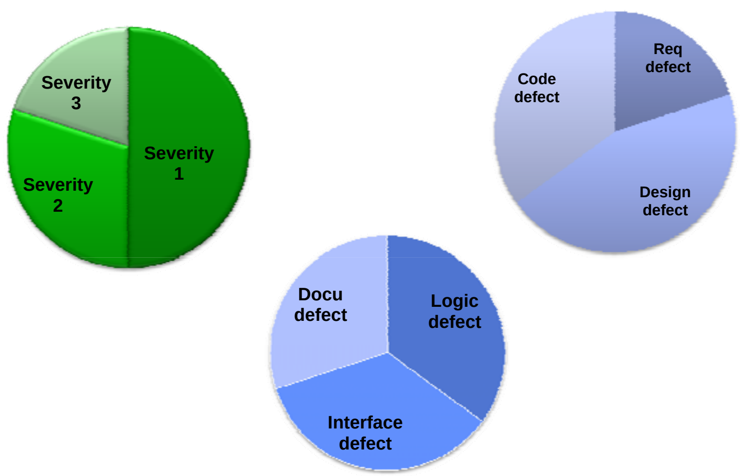
# Primitive Defect/Error/Fault Metrics

Introduction

Scale

**Metrics**

**Example:** 20 defects can be decomposed into:






## Defect Analysis Scorecard

© 2002 Six Sigma Advantage

Date:

Notes

Defect Source Distribution

| Code size | Size units |
|---|---|
| 1,250 | FP |

Total Defects> 946

| Origin Phase | Insertion % | Count | Found-in Phase: Reqts | Design | Code | Test | Post-Release | Total Containment Effectiveness (TCE) per phase | |
|---|---|---|---|---|---|---|---|---|---|
| Reqts | 22% | 208 | 139 | 34 | 18 | 12 | 5 | 97% | Reqts |
| Design | 35% | 331 | | 232 | 51 | 34 | 15 | 96% | Design |
| Code | 43% | 407 | | | 285 | 85 | 37 | 91% | Code |
| Test | | 0 | | | | 0 | 0 | | Test |
| Totals | 100% | 946 | 139 | 266 | 353 | 131 | 57 | 94.0% | <TCE |

| | Reqts | Design | Code | Test |
|---|---|---|---|---|
| PCE (errors) | 67% | 70% | 70% | |
| DCE (defects) | | 49% | 51% | 70% |

| | | Reqts | Design | Code | Test | Post-Release |
|---|---|---|---|---|---|---|
| Hours to Fix One Defect | Reqt | 1.01 | 2.44 | 6.27 | 14.00 | 57.00 |
| | Design | | 1.01 | 2.52 | 6.20 | 25.12 |
| | Code | | | 0.99 | 2.59 | 10.13 |
| | Test | | | | | |
| Total Fix Hours | Reqt | 141 | 82 | 112 | 168 | 298 |
| | Design | | 234 | 128 | 210 | 371 |
| | Code | | | 282 | 220 | 375 |
| | Test | | | | | |

| | | Reqts | Design | Code | Test | Post-Release | Totals | |
|---|---|---|---|---|---|---|---|---|
| Loaded Defect Fix Costs | Reqt | $ 14,055 | $ 8,245 | $ 11,247 | $ 16,806 | $ 29,809 | $ 80,161 | Reqt |
| | Design | | $ 23,409 | $ 12,766 | $ 21,019 | $ 37,101 | $ 94,295 | Design |
| | Code | | | $ 28,190 | $ 22,016 | $ 37,512 | $ 87,718 | Code |
| | Test | | | | | | $ - | Test |
| | Totals | $ 14,055 | $ 31,654 | $ 52,202 | $ 59,841 | $ 104,422 | $ 262,174 | |

| Project Cost Rollup | Hourly labor rate (loaded) | Person-months | Cost of Effort | | |
|---|---|---|---|---|---|
| | $ 100.00 | 53.0 | $ 828,125 | Cost per FP : | $ 209.74 |
| | | | | Cost per Released Defect | $ 1,831.00 |
| | | | | Fix Cost % | 32% |

TCE: Total containment effectiveness (bugs detected internally)

PCE (errors): Phase containment effectiveness (bugs detected at the same dev. phase)

DCE (defects): Defect containment effectiveness (bugs detected after the current phase)

*Definition:*

*Errors are found in the same phase*

*Defects are found in subsequent phases.*

# Defect Analysis Scorecard

Introduction

Scale

**Metrics**

- The top part of the scorecard predicts defect insertion and removal rates for key development activities.
- Bottom of the scorecard computes repair costs for each defect segment.
- The translation of defect data to dollars helps software engineers and managers talking directly to the business people.

Defect costs = 32% of the overall project cost (262174/ 828125).



# 1. Defect Density

Introduction

Scale

**Metrics**

**Product M**

Process M

Resource M

Defect density = number of defects detected / system size (e.g. defect/KLOC, defect/FP)

⌘ The higher the number of defects, the poorer the product quality.

- Defect density can be used to perform the following:
  - predict remaining defects by comparison with expected defect density;
  - determine if sufficient testing has been completed based on predetermined goals;
  - establish standard defect densities for comparison and prediction.

**Strength:** good correlation to the ability of the test process to eliminate defects.



# 1. Defect Density

Introduction

Scale

**Metrics**

**Product M**

Process M

Resource M

According to a study of 110 projects:

- DD range from 0.05 to 50 defects/KLOC
- Languages: C, Java, C++, Other

According to Steve McConnell,

- Industry data is 1-25 defects/KLOC.

Ref:
S M A Shah, M Morisio, M Torchiano, An overview of software defect density: a scoping study
Steve McConnell, 2004, Code Complete, 2nd ed, Microsoft Press,



# Weighted Defect Density

Introduction

Scale

**Metrics**

**Product M**

Process M

Resource M

Defect density may be weighted by severity :

$$\text{Weighted defect density} = \frac{W_1 * S + W_2 * A + W_3 * M}{\text{Size}}$$

where S = number of severe defects
A = number of major defects
M = number of minor defects
$W_i$ = weighting factors (defaults are 10, 3, & 1)

| | Released Defects Per Function Point | | |
|---|---|---|---|
| CMM Level | Minimum | Average | Maximum |
| 1 | 0.15 | 0.75 | 4.5 |
| 2 | 0.12 | 0.624 | 3.6 |
| 3 | 0.075 | 0.473 | 2.25 |
| 4 | 0.023 | 0.228 | 1.2 |
| 5 | 0.002 | 0.105 | 0.5 |

Source: Software Assessments, Benchmarks and Best Practices, Caper Jones, 2000.

# 2. Defect Age



- Defects are injected and removed at different phases of a software development cycle
- The cost of each defect injected in phase X and removed in phase Y increases with the increase in the distance between X and Y
- An effective testing method would find defects earlier than a less effective testing method would.

| Phase Injected | Phase Discovered | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Requirements | High-Level Design | Detailed Design | Coding | Unit Testing | Integration Testing | System Testing | Acceptance Testing |
| Requirements | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| High-Level Design | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Detailed Design | | | 0 | 1 | 2 | 3 | 4 | 5 |
| Coding | | | | 0 | 1 | 2 | 3 | 4 |



# 2. Defect Age



Defect age = time between when a defect is **introduced** to when it is detected or fixed

The defect age is the difference of the numbers corresponding to the phase introduced and phase detected.

Average defect age=phase detected-phase introduced / number of defects
(Summed over all the defects)

High number means V&V should be done earlier.



# 3. Defect Response Time



Defect response time = time between when a defect is **detected** to when it is fixed or closed

The defect response time should not be too long; otherwise, the users will be dissatisfied with the services.

***"It's fine to celebrate success but it is more important to heed the lessons of failure."***
*Bill Gates, 1995*

=> We need to learn from the defects

# 4. Defect Cost



Defect cost = cost to analyze the defect + cost to fix it + cost of failures incurred due to the defect

This metric provides data to calculate cost-benefit of any testing improvement project.

Example:
*Will introducing new test tool bring benefits more than its cost?*

# Suppose



"We have executed 90% of the test cases and 80% passed."

This means nothing unless it is known 'what' was tested, and how good are the test cases.

=> **We need coverage metrics (process metrics) to tell us what was tested.**



# 5. Instruction Coverage

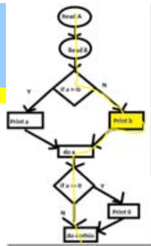


$$\text{Instruction coverage} = \frac{\text{number of instructions exercised}}{\text{total number of instructions}}$$

- ⌘ attempt to measure how much the program is tested
- ☺ **Strength:** the metric indicates what portions of the program have never been executed during test. These represent potential problems as some program areas have not been proven sound.
- ☹ **Weakness:** the evaluation does not show the importance of the areas not tested.



# 6. Path Coverage





$$\text{Path coverage} = \frac{\text{number of paths tested}}{\text{total number of paths}}$$

- ⌘ attempt to measure the extent of the program paths that have been tested
- ☺ **Strength:** it provides one of the best assurances that the program has been adequately tested. If all paths have been tested, there is a high degree of assurance that the program will function properly.
- ☹ **Weakness:** require many test cases. It is generally impractical and inefficient to test all the paths in a program. Therefore, try to achieve 100% coverage of critical (safety or security related) paths, not all paths.



# 7. Requirement Coverage

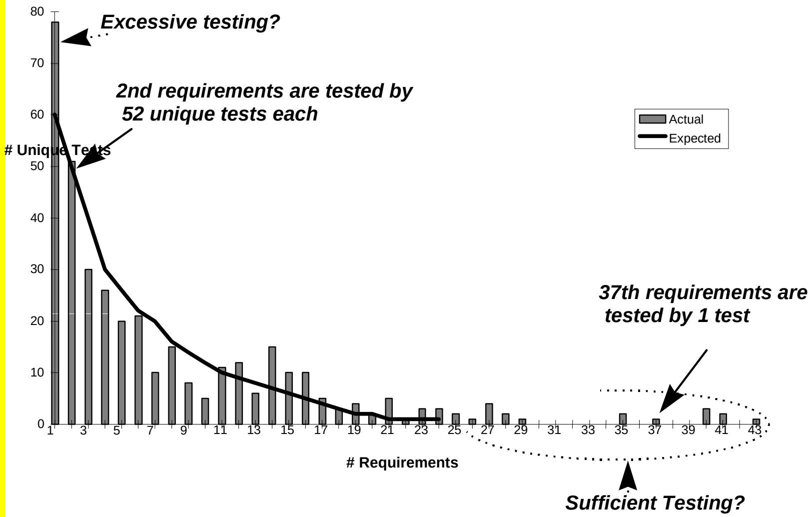


$$\text{Requirement coverage} = \frac{\text{passed requirements}}{\text{total requirements}}$$

- ⌘ Attempt to measure how much the requirements is tested.
- ☺ **Strength:** the metric provides one of the best assurances that the functions have been adequately tested.
- ☹ **Weakness:** some effort is required to trace test case to requirements.

# Test Case to Requirement Traceability








# Requirements traceability matrix



**These identifiers map the requirements between definition & specification documents.**

**These identifiers map all testing back to the requirements.**

| Requirement Definition ID | Requirement Spec. ID | Design Component ID | Code Component ID | Unit Test Case ID | Integration Test Case ID | System Test Case ID | Acceptance Test Case ID |
|---|---|---|---|---|---|---|---|
| RD.2.24 | RS2.2.4.1 | D2.2.4.1 | CC2.2.4.1 | UT2.2.4.1 | IT2.2..4 | ST.2.2.4 | AT2.2.4 |
| RD.2.24 | RS2.2.4.2 | D2.2.4.2 | CC2.2.4.2 | UT2.2.4.2 | IT2.2..4 | ST.2.2.4 | AT2.2.4 |
| RD.2.24 | RS2.2.4.3 | D2.2.4.3 | CC2.2.4.3 | UT2.2.4.3 | IT2.2..4 | ST.2.2.4 | AT2.2.4 |
| RD.2.24 | RS2.2.4.4 | D2.2.4.4 | CC2.2.4.4 | UT2.2.4.4 | IT2.2..4 | ST.2.2.4 | AT2.2.4 |

**These identifiers trace the design and code components back to the requirements.**

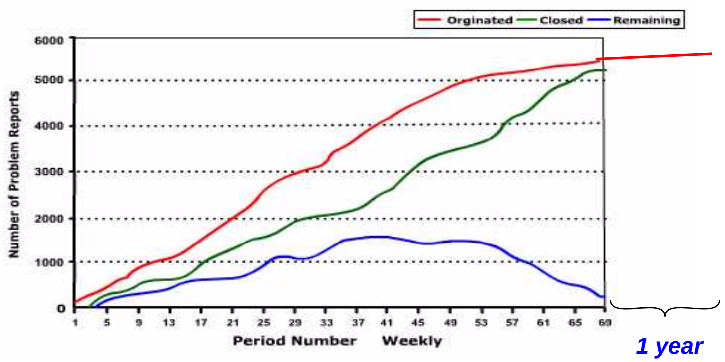


# 8. %Defects Uncovered in Testing



%Defects uncovered in testing = $\frac{\text{defects detected by testing}}{\text{total defects}}$

⌘ The first year is a normal time span in which to quantify the total defects.






# 8. %Defects Uncovered in Testing



- **Strength:** measure the effectiveness of the test process in detecting defects.

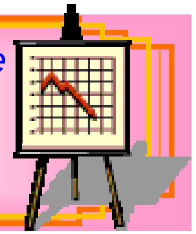
☹ **Weakness:** does not show the importance of the defects; it is possible that unimportant defects were detected and important ones were not, or vice versa

**How to improve this?**

***Classify defects by severity.***

*"Approximately 80 percent of defects come from 20 percent of modules."*

*Pareto Law*



**Thus, focus testing on a few key modules.**

# Question



Do you know how many test cases used by Microsoft for Office 2007?



# 9. Test Efficiency



Test efficiency = $\frac{\text{number of tests required}}{\text{number of defects}}$

e.g. 10 tests/defect

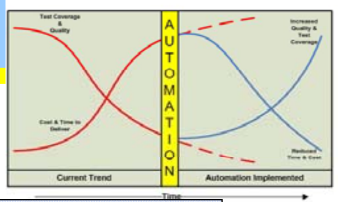
⌘ show proficiency on the part of the testing staff. The lower the metric, the more proficient the staff in testing.

☺ **Strength:** shows the number of test cases executed per detected defect.

Note that several defects may be detected by a single test.



# 10. %Test Automation






%Test automation = $\frac{\text{cost of automated test effort}}{\text{total test cost}}$

⌘ the use of tools to assist the test process is an indication of a more economical approach to testing.

☺ **Strength:** shows the automation of the test process and assumes that testing can be performed more effectively through automation

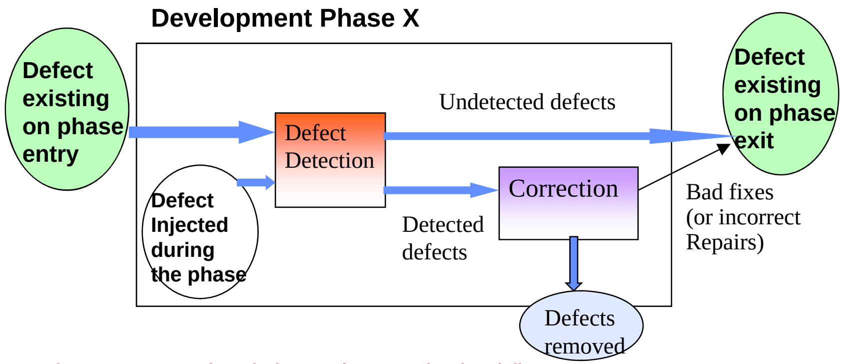
☹ **Weakness:** testing may not be more economical, just shows that the software tool is used.



# Defect Removal Model



Effective defect removal leads to reductions in development cycle time and better product quality.




Defects removed = defects detected – bad fixes

Defects at the **exit** of a development phase
= Defects escaped from previous phase + Defects injected in current phase - *Defects removed* in current phase

# Bid Fixes



When we try to fix a defect, we may introduce other defects or not really fix this defect!

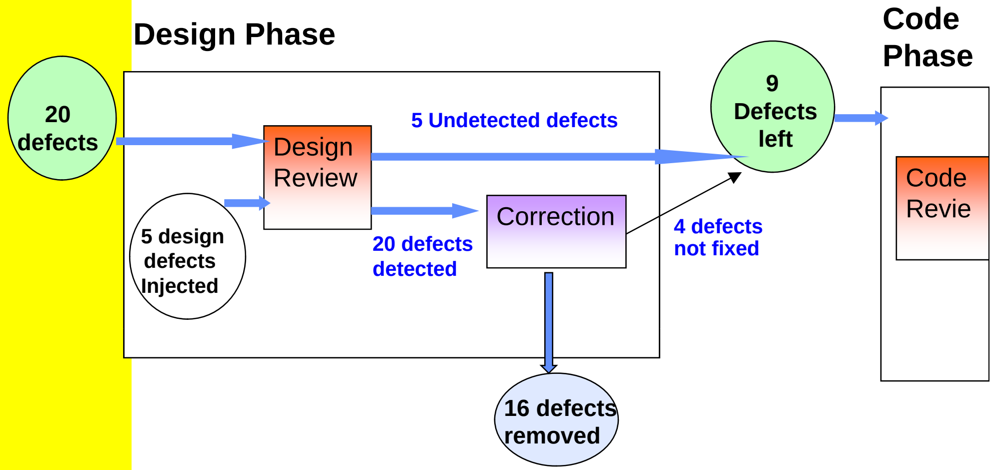
*Bad fixes* refers to secondary defects accidentally injected by means of a patch or defect repair that is itself flawed.

- Industry average: about 7%
- Unsuccessful projects: ~20%; i.e. one out of every 5 defect repairs introduced fresh defects.
- Successful projects: only 2% or less.

Reference: Jones, Capers. *Software Quality – Analysis and Guidelines for Success* . Boston, MA: International Thomson Computer Press, 1997.



# Example: Design Phase





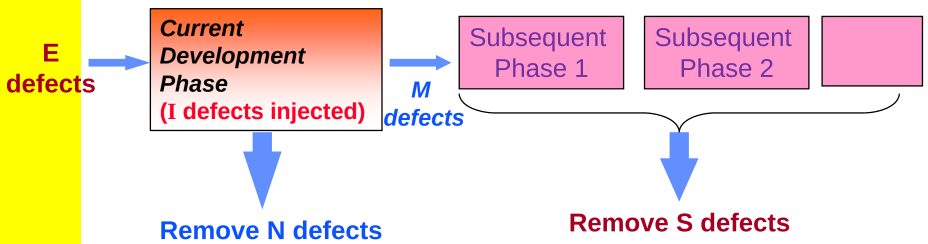

# 11. Defect Removal Effectiveness (DRE)

Two ways to compute DRE:

$$(1)\ DRE = \frac{N}{(N+M)} \times 100\%$$

where

- N is number of defect removed by this development phase.
- But, since we don't know M, we use S to approximate M.
- S is number of defect removed by subsequent phases, which are **present** at the exit of the current development phase.






# 11. Defect Removal Effectiveness (DRE)



$$(2)\ DRE = \frac{N}{(E+I)} \times 100\%$$

where

E is the number of defects existing on phase entry,
I is the number of defects injected in the phase

$$N + M = E + I$$

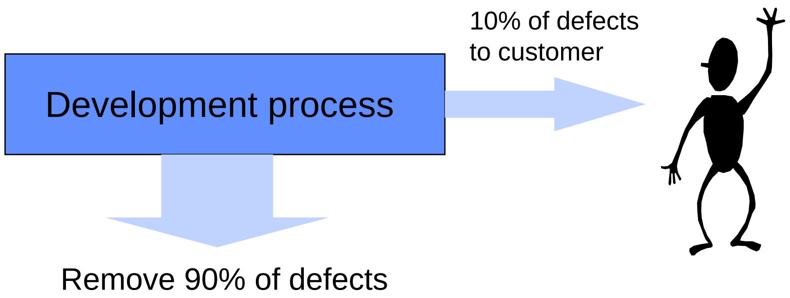
# DRE of the complete development process



A study by HP in 1999 indicated that:

DRE for good software company is

***90% only!***






# DRE

Introduction
Scale
Metrics
Product M
Process M
Coverage
Effectiveness
Resource M

The DRE can be computed for each lifecycle phase and plotted on a bar graph to show the relative DRE for each phase.

DRE may also be computed for a specific process (e.g., design inspection, unit test, six- month operation, etc.)

☺ **Strength:** shows the value of testing/analysis during a particular phase, e.g. design phase.

☹ **Weakness:** may not show the importance of the defects detected, merely the frequency of defects detected.

*How to improve this?*



# Industry Data on DRE

Introduction
Scale
Metrics
Product M
Process M
Coverage
Effectiveness
Resource M

| | DRE | |
|---|---|---|
| **Activity** | **Low (%)** | **High (%)** |
| Informal design review | 25 | 40 |
| Design inspection | 45 | 65 |
| Informal code review | 20 | 35 |
| Code inspection | 45 | 70 |
| Unit test | 15 | 50 |
| Regression test | 15 | 30 |
| Integration test | 25 | 40 |
| System test | 25 | 55 |
| Test of new function | 20 | 35 |

Reference: Jones, Capers, “Software defect-removal efficiency”, *IEEE Computer.* 5/96, pp. 94-96.

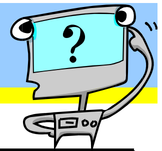


# Example: Phase-Based Defect Removal

Introduction
Scale
Metrics
Product M
Process M
Coverage
Effectiveness
Resource M

| **Phase** | (A) Defect Escaped From Previous Phase (per KLOC) | (B) Defect Injection (per KLOC) | Subtotal (A+B) | Removal Effectiveness | Defect Removal (per KLOC) | Defects at Exit of Phase (per KLOC) |
|---|---|---|---|---|---|---|
| Requirements | - | 1.0 | 1.0 | - | - | 1.0 |
| Design | 1.0 | 9.0 | 10 X | 60% | = 6.0 | 4.0 |
| Code | 4.0 | 12.5 | 16.5 X | 50% | = 8.3 | 8.2 |
| Unit test | 8.2 | - | 8.2 X | 30% | = 2.5 | 5.7 |
| Component test | 5.7 | - | 5.7 X | 35% | = 2.0 | 3.7 |
| System test | 3.7 | - | 3.7 X | 35% | = 1.3 | 2.4 |
| Field | 2.4 | | | | | |

# Given this data



Introduction

Scale

**Metrics**

Product M

**Process M**

Coverage
**Effectiveness**

Resource M

## Defect Origin

WHERE FOUND

| | REQUIR-EMENTS | DESIGN | CODE | UNIT TEST | COMPONE-NT TEST | SYSTEM TEST | FIELD | TOTAL |
|---|---|---|---|---|---|---|---|---|
| RQ | - | | | | | | | |
| DESIGN | 6 | 650 | | | | | | 656 |
| CODE | 10 | 105 | 920 | | | | | 1035 |
| UT | 15 | 40 | 210 | 2 | | | | 267 |
| CT | 25 | 55 | 250 | - | 2 | | | 332 |
| ST | 8 | 21 | 65 | - | - | 2 | | 96 |
| FIELD | 6 | 16 | 20 | - | - | - | 2 | 44 |
| TOTAL | 70 | 887 | 1465 | 2 | 2 | 2 | 2 | 2430 |

Compute

1. DRE(Design) 2. DRE(Code) 3. DRE(Unit test)



# 12. % Test Case Successfully Executed

Introduction

Scale

**Metrics**

Product M

**Process M**

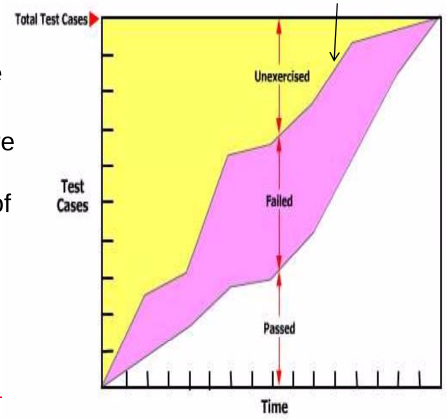
Coverage
**Effectiveness**

Resource M

% tc successfully executed = planned tc run to completion / total test cases

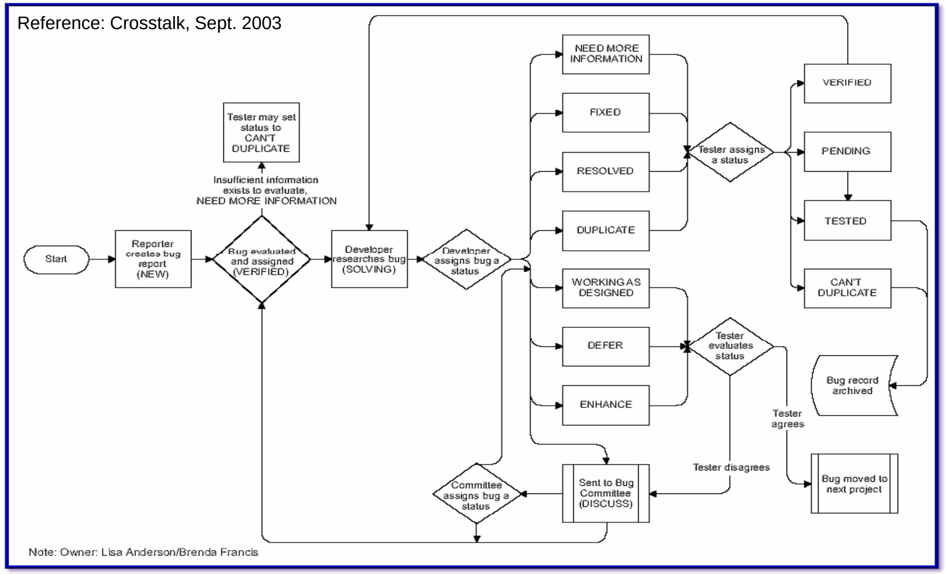
The data can be collected from the test log.

- **Strength:** Used to estimate remaining test time
- Typically, 90% of the tests are executed at least once (but not successfully) after 60% of the test execution effort.
- Few defects should be detected in the last part of test execution






# A Defect Life Cycle






# Status of a Defect

1. **Open**
   - Reported
   - Solving (being dealt with)
   - Need more information
   - Verified
   - Can't duplicate (not a defect)
   - Pending (not being dealt with)
   - Tested (being dealt with)

2. **Fixed (or resolved)**

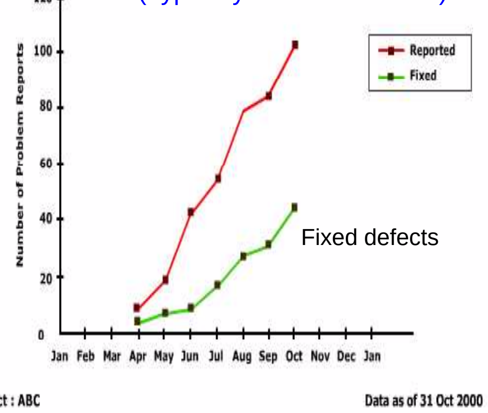
# 13. % Fixed Defects



$$\%\text{Fixed defects} = \frac{\text{Fixed defects}}{\text{Total defects}}$$

- ⌘ From defect log.
- ⌘ Look at the trend.
- ⌘ Falling % should be a signal for alarm
- ☹ **Weakness:** The focus should be on the defect trend, rather than the %.

Example: Trend of Fixed Defects
(Typically a ʃ shade curve)






# What Defects to Fix First?



## Prioritize Defect

- Assign each defect a severity rating and a likelihood rating.

Priority = severity x likelihood

- Severity rating of 1 is the most severe.
- The defect with the 1 rating should be fixed first. Lower severity defects allow longer time to fix.

| Severity Rating | Value |
|---|---|
| Hang | 1 |
| Loss, no workaround | 2 |
| Loss with Workaround | 3 |
| Inconvenient | 4 |
| Enhancement | 5 |

| Likelihood Rating | Value |
|---|---|
| Always | 1 |
| Usually | 2 |
| Sometimes | 3 |
| Rarely | 4 |
| Never | 5 |



# 14. % Open Defects



$$\%\text{Open defects} = \frac{\text{Open defects}}{\text{Total defects}}$$

- ⌘ Rising % should be a signal for alarm

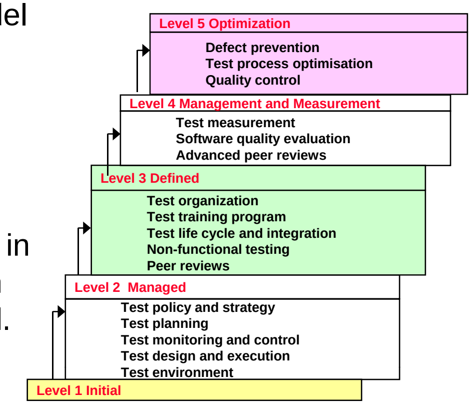
Example: Problem Reports **by Week**


# 15. Test Maturity



**Maturity level** (1-5) based on an assessment of the test process

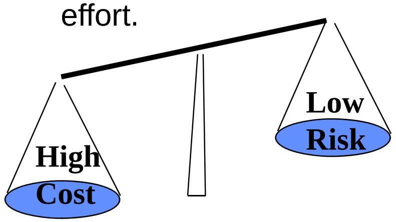
- ⌘ Use Test Maturity Model Integration (TMMI) or similar models
- ⌘ Assessor will evaluate the testing process, based on practices/ processes being used in the company and then assign a maturity level.
- ⌘ *Top level is 5; lowest level is 1*



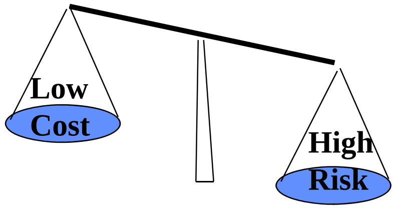
# Cost and Benefits

Introduction
Scale
Metrics
Product M
Process M
Resource M

Too much test: wasted testing effort.




Too little test: defects escape; wasted effort in rework.






# 16. Relative Test Cost

Introduction
Scale
Metrics
Product M
Process M
Resource M

Relative test cost = $\frac{\text{test cost}}{\text{total system cost}}$

⌘ test cost should be between 15-40% of total development effort

For critical systems (financial systems), testing can consume > 70% of the budget

☺ **Strength:** shows the amount of the development effort that is allocated to testing. It provides an indication as to the extent of testing

☹ **Weakness:** there may not be a direct relationship between the effectiveness of testing and the amount of time/effort allocated to testing.



# How many testers for each developer?

Introduction
Scale
Metrics
Product M
Process M
Resource M

| | Tester | Developer |
|---|---|---|
| Web development | 1 | 5-10 |
| Microsoft | 1 | 1 |
| Microsoft Windows 2000 | 2 | 1 |
| Safety critical software | 5 | 1 |
| NASA space shuttle flight control software | 10 | 1 |



# 17. Cost to detect Defect

Introduction
Scale
Metrics
Product M
Process M
Resource M

Cost to detect defect = $\frac{\text{cost of testing}}{\text{defects detected by the test process}}$

e.g. $500/defect

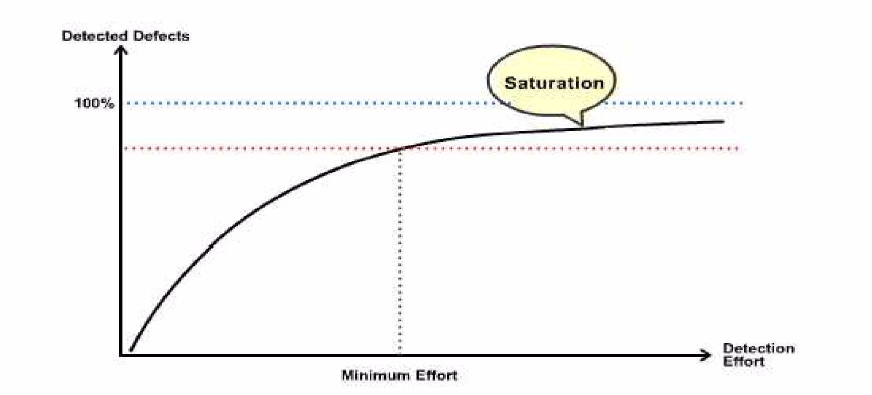
⌘ can be used to determine when it is more effective not to continue testing than to continue testing.

⌘ It may be cheaper to let the defect be placed into production than to detect it.

☺ **Strength:** shows the cost of eliminating defects.

☹ **Weakness:** does not show the potential loss associated with a defect, and thus it cannot put the cost relative to the benefit received from that cost.

# Defect Detection Curve




- Effort required to detect a defect increases as more and more defects have been removed, because fewer remaining defects.
- It may require a large effort to detect the last few remaining defects.



# 18. %Achieving Budget

Introduction
Scale
**Metrics**
Product M
Process M
**Resource M**

%Achieving budget = anticipated cost of testing / actual cost of testing

⌘ show the effectiveness of the test team in conducting the test within the budget. The lower the metric, the poorer the testing performance.

☺ **Strength:** shows the performance of the test team to accomplish the test within the project budget.

☹ **Weakness:** poor estimate of the amount of time may cause poor effectiveness



# 19. % Test Case Prepared

Introduction
Scale
**Metrics**
Product M
Process M
**Resource M**

%Test case prepared = actual test cases prepared / total test cases planned

⌘ Total number of test cases can be estimated from testing components.

⌘ Number of test cases prepared should increase steadily with resource use.

⌘ Alarming trend if preparation % keeps on the same level even if the resources are used up.



# 20. Tester Productivity

Introduction
Scale
**Metrics**
Product M
Process M
**Resource M**

**2 metrics:**

(1) Productivity = number of defects found / hour
(inverse of cost to locate defect)

(2) Productivity = LOC tested / hour or FP tested/hour

**Example Data from HP (2000-01):**

Quality (customer-reported-defects per month per million LOC averaged over the first 12 months after release)
Mean defect rate = 18.8 defects/month/Million LOC
= 0.23/KLOC

Productivity (new code developed per person-day)
= 26 LOC per person-day

# Example: Data analysis

Suppose we collect data: Defect counts and their severities and Testing effort (person-days)

*Defect detected by users*

We have the following data:

| Phase | Defect severity | | | Total defect | Testing effort (days) |
|---|---|---|---|---|---|
| | High | Medium | Low | | |
| Dev. | 23 (21%) | 35 (24%) | 87 (55%) | 145 (35%) | 35/260 |
| Test | 51 (47%) | 67 (47%) | 45 (28%) | 163 (40%) | 20 |
| Production | 34 (32%) | 42 (29%) | 27 (17%) | 103 (25%) | |
| Total | 108 | 144 | 159 | 411 (100%) | |

1. High and medium severity problems discovered by users is unacceptable. Testing is ineffective for the early discovery of these problems.
   Recommendation
   Test planning, scheduling and test resources should be assessed then improved to increase the number of errors found earlier.

2. More effort were expended by dev testing (35) than by those in the test dept (20) yet the number and severity of problems found by developers was ineffective. (Testers + users discovered 65% defects)
   Recommendation: Development teams should discover more problems in requirements, design, and coding

# Observations



3. As a general rule, testing costs are expected to be about 40% of total development costs. In this case, 55 person days were expended on testing (developers and testers) – 19% of total development and test costs.
   Recommendation
   More effort should be spent on test planning, scheduling, and resources to decrease the number of problems.

4. Developers and testers discovered 83% of low severity problems. Test planning and execution were apparently successful for this severity class but not so for high and medium severity problems (only 57% discovered).
   Recommendation
   Test planning should be substantially improved with test cases, tools, resources, and time to discover the majority of high and medium severity problems before production.



# Another example analysis of Defect

- The defect data provide insights to identify which processes cause the most defects and which processes allow defects to escape.
- From Table X, 48% of defects originating in design were detected (contained) in the design review process;
- 55 percent of defects originating in code were detected in the code review/unit test process;
- The overall defect containment which equal the total number of defects caught in-stage/total defects was:

$$(1{,}515+1{,}555+2{,}421+37+1+10+0)/11{,}292 = 49\ \%$$



# Table X

| Stage Detected | Stage Originated | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Requirements | Design | Code and Unit Test | SW Integration | SW Quality Test | System Integration and test | SW Maintenance | Total |
| Requirements | 1,515 | | | | | | | 1,515 |
| Design | 1,181 | 1,555 | | | | | | 2,736 |
| Code and Unit Test | 402 | 912 | 2,421 | | | | | 3,735 |
| SW* Integration | 200 | 420 | 1,525 | 37 | | | | 2,182 |
| SW Quality Test | 191 | 223 | 370 | 7 | 1 | | | 792 |
| System Integration and Test | 89 | 114 | 114 | 5 | 0 | 10 | | 332 |
| SW Maintenance | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Total | 3,578 | 3,224 | 4,430 | 49 | 1 | 10 | 0 | 11,292 |

* SW = Software

| Stage Detected | Stage Originated | | | | | | |
|---|---|---|---|---|---|---|---|
| | Requirements | Design | Code and Unit Test | SW Integration | SW Quality Test | System Integration and test | SW Maintenance |
| Requirements | 42% | | | | | | |
| Design | 33% | 48% | | | | | |
| Code and Unit Test | 11% | 28% | 55% | | | | |
| SW Integration | 6% | 13% | 34% | 76% | | | |
| SW Quality Test | 5% | 7% | 8% | 14% | 100% | | |
| System Integration and Test | 2% | 4% | 3% | 10% | 0% | 100% | |
| SW Maintenance | 0% | 0% | 0% | 0% | 0% | 0% | 0% |

# Quantitative Analysis

**Use *past data* to analyze current performance and predict future performance.**

- We create and maintain control limits based on performance capability.

To use defect containment in this predictive manner, we need:

1) establish a baseline by using defect containment data from previously completed projects,
2) normalize the defect data found in each stage by size (e.g. SLOC), and
3) apply statistical techniques to set limits of expected defect detection performance.



# 3 Defect Metrics

3 metrics derived from the defect containment matrix:

1) Cumulative Defects Originated in Design Detected by Stage;
2) Cumulative Defects Originated in Code and Unit Test Detected by Stage;
3) Defect Detection Distribution by Stage.

To create these metrics, the following **base and derived metrics** are required:

- Number of defects by stage of origin
- Number of defects by stage of discovery
- A size count, such as SLOC or function points.
- Normalize the defect count by the size count (e.g. x defects per KSLOC).



# 3 Defect Metrics

- Use comparison techniques on historical data to establish the range of expected defect detection, i.e., control limits.
- These data must come from independent observations of the same process (e.g. separate design reviews.)
- The data from each life-cycle stage is compared to data from its own stage.
- In cases where a defect in an earlier stage causes a defect in a later stage, the defect counts as a single defect in the stage it was originally introduced.
- Control limits is derived by calculating 3σ limits based on existing data. Then, defect data will fall between these (3σ) limits 99.7 percent of the time.
- Using 3-sigma limits avoids the need to make assumptions about the distribution of the underlying natural variation.



# Control Limits

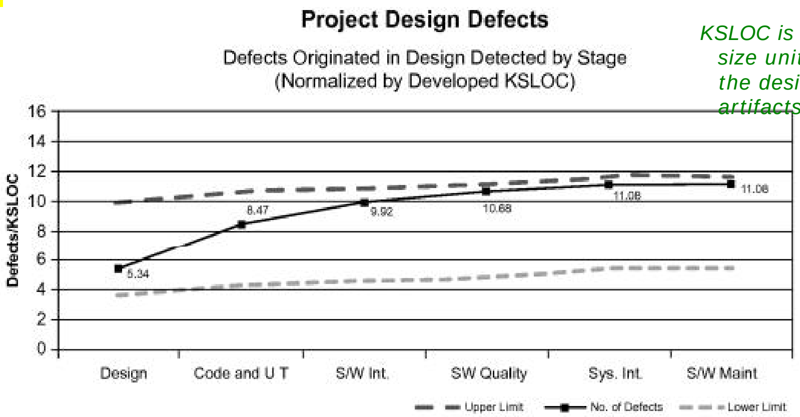
For Cumulative Defects Originated in Design Detected by Stage and for Cumulative Defects Originated in Code and Unit Test Detected by Stage, 3-sigma control limits are established using the following **u-chart formulas**:

$$UCL = \bar{u} + 3\sqrt{\frac{\bar{u}}{n_i}},$$

$$LCL = MAX\left[0, \bar{u} - 3\sqrt{\frac{\bar{u}}{n_i}}\right],$$

where ubar is the mean for each subgroup and $n_j$ is the sample size.

# u-chart of Design Defects

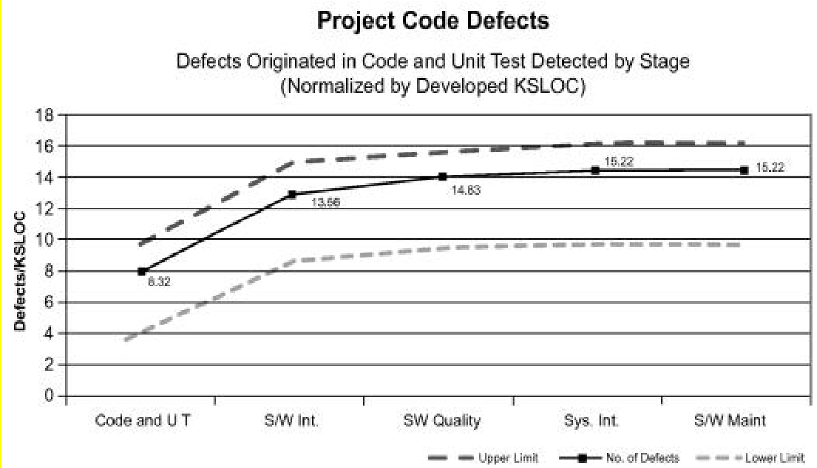



*KSLOC is the size units of the design artifacts.*

- x-axis is the life-cycle stage
- y-axis is the detected **design-originated defects** normalized by size.
- Plot the total normalized number of design defects found in-stage, followed by the total cumulative numbers of design defects detected in each subsequent life-cycle stage.



# u-chart of Code & Unit Test Defects






# Use of the control chart

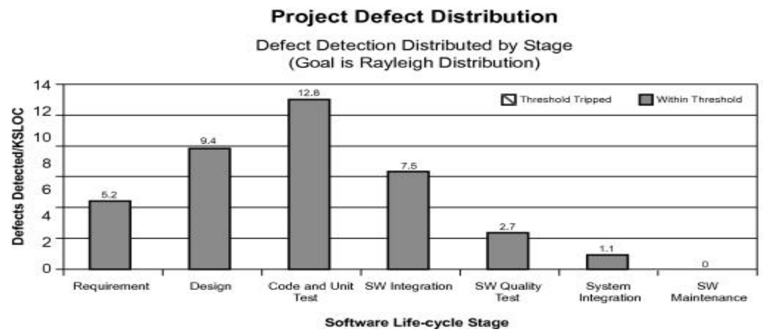
- If a current project falls above the upper control limit, perform causal analysis to understand the cause:
  - investigate means to reduce defects injected, adjusting control limits, and
  - identify best practices for defect detection to be implemented.
- If a current project falls below the lower control limit, try to get the current project to be as effective (e.g. during peer review) as the past projects

  Example action: improve design and code peer reviews.
- If the project data falls within the control limits.
  - deploy defect prevention measures that drive the data toward the lower control limit of the charts.
  - gather a large enough sample to tighten the existing control limits and decrease projected variability.



# Defect Distribution

- The number of defects detected in the software requirements stage should be less than the number found in the design stage.
- The number of defects detected should continue to increase through the code and unit test stage.
- After the code and unit test stage, the defects detected in each stage should decrease through the remaining stages with software maintenance stage detecting the least defects



# Use of metrics

- If the project has more defects detected in the design stage than the code stage, the project may not be ready to begin the software integration effort.
- Establishing control limits on defect detection can predict the number of defects that will be inserted into project work products, based on work product size and the standard organizational software development process.

| Lifecycle Stage Where Design Defects Detected | Design Defects/Actual KSLOC | |
|---|---|---|
| | Minimum | Maximum |
| Design | 3.7 | 9.9 |
| Code and Unit Test | 4.4 | 10.6 |
| SW Integration | 4.7 | 10.9 |
| SW Quality Test | 4.9 | 11.1 |
| System Integration and Test | 5.4 | 11.6 |
| SW Maintenance | 5.4 | 11.6 |

- Predicting defects inserted within a statistically derived range may be used to determine readiness to move from one development stage to the next, and to predict future rework costs.
- Utilizing organization data or industry standards on *hours to correct defects by stage*, return on investment can be determined.



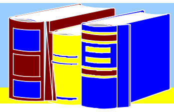
# Summary

- There are 4 measurement scales: normal, ordinal, interval, and ratio; they determine the kind of manipulations that can be applied to the data;
- Test metrics may be classified into product, process and resource metrics;

- Software firms which have used all or even some of the presented metrics achieved significant improvements.
- Why? Because management has **visibility** to the development and testing process and decisions are made based on data



# References



1. Kan, Stephen H., **Metrics and Models in Software Quality Engineering**, Addison Wesley, 1995
2. Jones, Capers, **Software Assessments, Benchmarks, and Best Practices**, Addison Wesley Longman, Boston, Mass., 2000.

## Web Resources

1. Home of the Practical software & system measurement support center, http://www.psmsc.com