Unit 3. Quality Models and Measurements

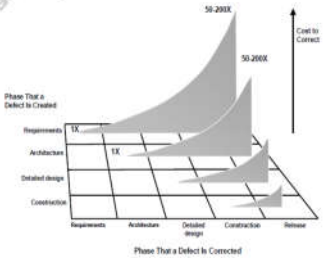
1. Introduction
2. Models for Quality Assessment
3. Product Quality Metrics
4. In-Process Metrics for Software Testing
5. In-Process Quality Management
6. Effort/Outcome Model

Reading: TB-Chapter 13 – : Sections 13.4, 13.11
Chapter 12 – : Section 12.81.1 (pages 380-384)

1. Introduction

-It is important to assess the quality of a software product, project the number of defects, or estimate the mean time to next failure when development **work is complete**.

-It is even more important to monitor and manage the quality of the software when it is **under development**.



-Objective quality management requires established quality metrics and models. The purpose of this chapter is to introduce some of these metrics and models.

Software Measurement and Metrics

-***Software measurement*** is concerned with deriving a numeric value for some attribute of a software product or a software process.

•Comparing these values to each other and to relevant standards allows drawing conclusions about the ***quality*** of software ***product*** or the ***effectiveness*** of software ***processes***.

Metrics will target three kinds of entities:
-Process
-Def. Remov. Effic. (DRE)
-Inspection Rate
-Product
-Size
-Defect Count
-Project
-Head count
-Development time
-Cost
-Budget

Software Quality Metrics

-Subset of software metrics that focus on the quality aspects of the product, process, and project.

•Are, in general, more closely associated with process and product metrics than with project metrics.

•Nonetheless, project parameters such as the number of developers and their skill levels, the schedule, the size, and the organization certainly affect product quality.

-Can be further divided into ***end-product quality*** metrics and ***in-process quality*** metrics.

•The ultimate goal of software quality engineering is to investigate the relationships among in-process metrics, project characteristics, and end-product quality, and based on these findings to engineer improvements in both process and product quality.

2. Models for Quality Assessment

-Analytical models that provide quantitative assessment of selected quality characteristics based on measurements data from software projects.

•can help us obtain an ***objective assessment of the product quality***

•when applied over time, these models can provide us with an accurate ***prediction of the future quality***, which can help make project scheduling, resource allocation, and other management decisions.

•some models can help ***identify problematic areas*** so that appropriate remedial actions can be applied for quality or process improvement.

Models for Quality Assessment (ctd.)

-Existing quality assessment models can be classified into two broad categories: ***generalized*** models and ***product-specific*** models.

•Generalized models:
-Provide rough estimates of product quality in the form of industrial averages or general trends for a wide variety of application environments with little or no project-specific data required.

→ For instance, using a single estimate of product quality such as ***defect density***.

→ For instance, using different quality estimates for different industrial segments, such as ***different defect density estimates according to market segments*** or products groups.

-Safety Critical
-Air traffic control, nuclear, etc
-Failure Probability: < 10^{-7}
-Commercial
-Business applications, etc.
-Failure Probability: 10^{-7} to 10^{-3}
-Auxiliary
-Games, etc.
-Failure Probability: > 10^{-3}

•**Product-Specific Models:**
-Provide more precise quality assessments using product-specific data

-Examples:

→***Defect Dynamics Model:*** extrapolate product history to predict quality for the current project by tracking defect injection and removal by development phase.

→***Reliability Growth Models:*** estimate quality based on observations from the current project. Observed failures and associated time intervals are fitted to statistical models to evaluate product reliability.

→***Machine Learning/Regression Models:*** establish predictive relations between various early measurements and product quality, and use them for quality assurance and improvement.

3. Product Quality Metrics

-De facto definition of software quality consists of two factors: ***intrinsic product quality*** and ***customer satisfaction***.

Customer Satisfaction Metrics

-Often measured by customer survey data via the five-point scale:
• Very satisfied, Satisfied, Neutral, Dissatisfied, Very dissatisfied

-Can be monitored using, e.g., the CUPRIMDSO categories:

capability, functionality, usability, performance, reliability, installability, maintainability, documentation/information, service, and overall.

-Based on the five-point scale, various metrics may be computed:

(1) Percent of completely satisfied customers
(2) Percent of satisfied customers : satisfied and completely satisfied
(3) Percent of dissatisfied customers: dissatisfied and completely dissatisfied
(4) Percent of non-satisfied customers: neutral, dissatisfied, and completely dissatisfied

3. Product Quality Metrics (ctd.)

Intrinsic Product Quality

-Usually measured by the number of "bugs" in the software or by how long the software can run before encountering a crash.

•In operational definitions, the two metrics used for intrinsic product quality are:
- ***Defect density (rate)***
- ***Mean Time To Failure (MTTF)***

-The two metrics are correlated but are different both in purpose and usage.

•***MTTF:***
-Most often used with special-purpose systems such as safety-critical systems
-Measure the time between failures

•***Defect density***:
-Mostly used in general-purpose systems or commercial-use software.
-Measure the defects relative to the software size (e.g., lines of code, function points)

Size-Oriented Metrics

•**Size** may be estimated:
-Directly using size oriented metrics: lines of code (LOC)
-Indirectly using function oriented metrics: function points (FP)

Lines of Code (LOC)

-Lines of code metric is anything but simple:

•Major difficulty comes from the ambiguity of the operational definition, the actual counting.

•Early days of Assembler programming: LOC definition was clear because one physical line was the same as one instruction.

•With high-level languages the one-to-one correspondence doesn't work.

•Differences between physical lines and instruction statements (or logical lines of code) and differences among languages contribute to the huge variations in counting LOCs.

Lines of Code (ctd.)

•***Example :*** How many (physical and logical) lines of code are the following C code snippets:

Code snippet 1:

```
for (i = 0; i < 100; i += 1) printf("hello"); /* code snippet 1 */
```

Code snippet 2:

```
/* code snippet 2 */
for (i = 0; i < 100; i += 1)
{
  printf("hello");
}
```


Lines of Code (Ctd.)

•Possible approaches used include:

-Count only executable lines
-Count executable lines plus data definitions
-Count executable lines, data definitions, and comments
-Count executable lines, data definitions, comments, and job control language
-Count lines as physical line on an input screen
-Count lines as terminated by logical delimiters.

-Method for LOC counting should always be described, when any data on size of program products and their quality are presented, whether it is based on physical or logical LOC.

•When straight LOC count is used, size and defect rate comparisons across (different) languages are often invalid.

•Example: the approach used at major software companies consists of counting source instructions including executable lines and data definitions but excluding comments and program prologue.

Defect Density Metrics

-In some organizations the following LOC count metrics, based on logical LOC, are used:

- ***Shipped Source Instructions (SSI):*** LOC count for the total product
- ***Changed Source Instructions (CSI):*** LOC count for the new and changed code of the new release.

-The relationship between SSI and CSI is given by:

SSI (current release) = SSI (previous release)
+ CSI (new and changed code instructions for current release)
- deleted code (usually very small)
- changed code (to avoid double count in both SSI & CSI)

-Defects after the release of the product are tracked.

- Defects can be field defects (found by customers), or internal defects (found internally).

-Postrelease defect rate metrics can be computed by thousand SSI (KSSI) or per thousand CSI (KCSI):

1. **Total defects per KSSI:** a measure of code quality of the total product
2. **Field defects per KSSI:** a measure of defect rate in the field
3. **Release-origin defects (field and internal) per KCSI:** a measure of development quality
4. **Release-origin field defects per KCSI:** a measure of development quality per defects found by customers.

- Metrics (1) and (3) are process measures; their field counterparts, metrics (2) and (4) represent the **customer's perspective**.

-Customer's point of view:

- The defect rate is not as relevant as the total number of defects that might affect their business
- A good defect rate target should lead to a release-to-release reduction in the total number of defects, regardless of size.

Exercise 3.1: We are about to launch the third release of an inventory management software system.

-The size of the initial release was 50 KLOC, with a defect density of 2.0 def/KSSI.

-In the second release, 20KLOC of code was newly added or modified (20% of which are changed lines of code, with no deleted code).

-In the third release, 30KLOC of code was newly added or modified (20% of which are changed lines of code, with no deleted code).

-Assume that the LOC counts are based on source instructions (SSI/CSI).

a. Considering that the quality goal of the company is to achieve 10% improvement in overall defect rates from release-to-release, calculate the total number of additional defects for the third release.

b. What should be the maximum (overall) defect rate target for the third release to ensure that the number of total defects does not exceed that of the second release.

4. In-process Metrics for Software Testing

Defect Arrival Pattern During Machine Testing

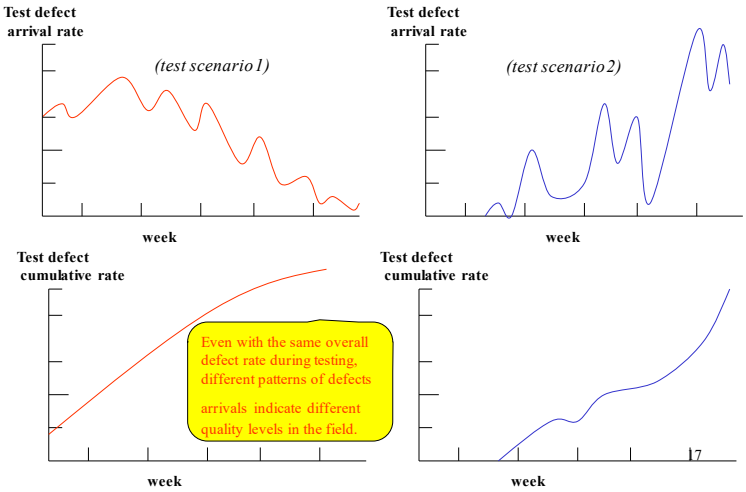
-Defect rate during formal machine testing is actually positively correlated with the defect rate in the field.

- Higher defect rates found during testing is often an indicator that the software has experienced higher error injection during its development process.
- Rationale: if a piece of code has higher testing defects, it is a result of more effective testing or it is because of latent defects in the code.

-Hence, the simple metric of ***defects per KLOC or function point*** is a good indicator of quality while the software is still being tested.

-However, defect density during testing is a summary indicator; more information is actually given by **the pattern of defect arrivals** (which is based on the times between failures).

Examples: Defect arrival patterns for two different projects



Defect Arrival Pattern During Machine Testing (ctd.)

-***Defect tracking and management during the testing phase:*** highly recommended as a standard practice for all software testing.

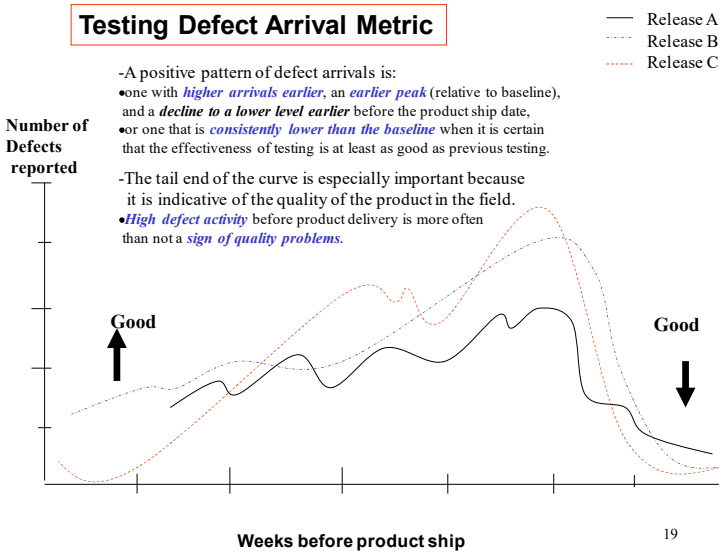
- Recommended to track the defect arrival pattern over time, in addition to tracking by test phase.

- Objective: always look for defect arrivals that stabilize at a very low level, or times between failures that are far apart, before stopping the testing effort and releasing the software to the field.

-For the metric to be useful, it should contain the following:

- Data for a comparable baseline (a prior release, a similar project, or a model curve) if such data is available.
- The unit for the *x*-axis is time (e.g., hour, day, week) before product ship. For models that require **execution time** data, the time intervals is in **units of CPU time.**
- The unit for the y-axis is the number of defect arrivals for the week, or its variants.

Testing Defect Arrival Metric



Test Progress S Curve

-*Tracking the progress of testing* is considered one of the most important tracking tasks in software testing.

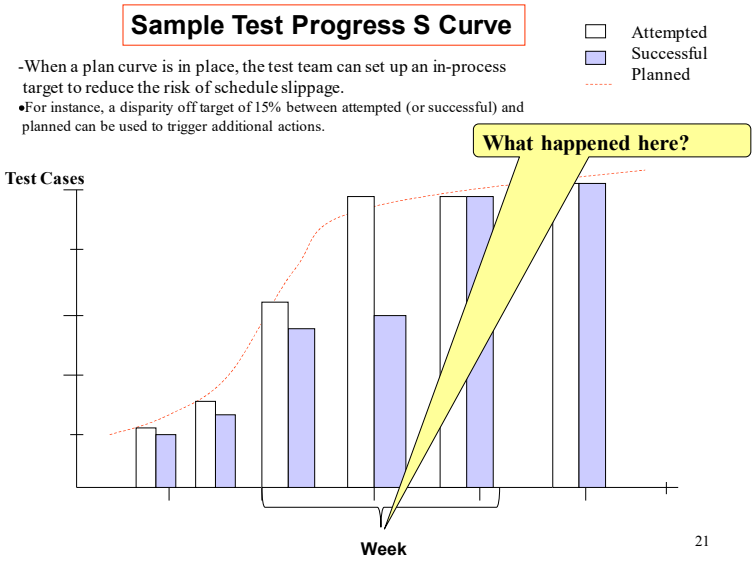
-A metric used for that purpose is the test progress S curve over time.
•The *x-axis* of the S curve represents the time units and the *y*-axis represents the number of test cases or test points.

-The purpose of this metric is to track *test progress and compare it to the plan*, and therefore be able to *take action upon early indications* that testing activity is falling behind.

-For the metric to be useful, it should contain the following information on one graph:
•*Planned progress over time* in terms of number of test cases or number of test points to be completed successfully by time unit (e.g., hour, day, week).
•Number of test cases *attempted* by time unit.
•Number of test cases *completed successfully* by time unit.

Sample Test Progress S Curve

-When a plan curve is in place, the test team can set up an in-process target to reduce the risk of schedule slippage.
•For instance, a disparity off target of 15% between attempted (or successful) and planned can be used to trigger additional actions.



5. In-Process Quality Management

Weibull Distribution

-Used for decades in various engineering fields for reliability modeling

•Key characteristic: tail of the probability density function approaches zero asymptotically, but never reaches it.

•Cumulative distribution function (CDF) and probability density function (PDF):

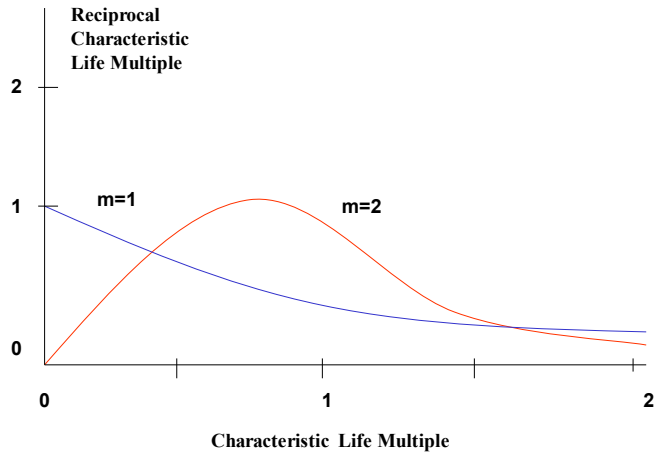
$$CDF: \quad F(t) = 1 - e^{\left(\frac{-t}{c}\right)^m}$$

$$PDF: \quad f(t) = \frac{m}{t}\left(\frac{t}{c}\right)^m e^{\left(\frac{-t}{c}\right)^m}$$

Where *m* is the shape parameter, *c* is the scale parameter, and *t* is the time.

Note -When applied to software:
- the PDF often means the defect density over time or the defect arrival pattern;
- the CDF means the cumulative defect arrival pattern.
- a constant *K*, corresponding to the total number of defects or the total cumulative defect rate, is multiplied to the formula.

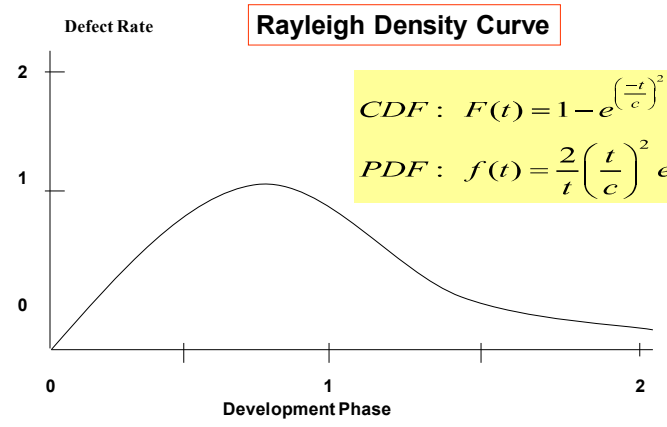
Weibull Probability Density



The Rayleigh Model

-Special case of the Weibull distribution **when *m=2***.
• The Rayleigh PDF first increases to a peak and then decreases at a decelerating rate.

Rayleigh Density Curve



$$CDF: \quad F(t) = 1 - e^{\left(\frac{-t}{c}\right)^2}$$

$$PDF: \quad f(t) = \frac{2}{t}\left(\frac{t}{c}\right)^2 e^{\left(\frac{-t}{c}\right)^2}$$

Rayleigh Quality Model

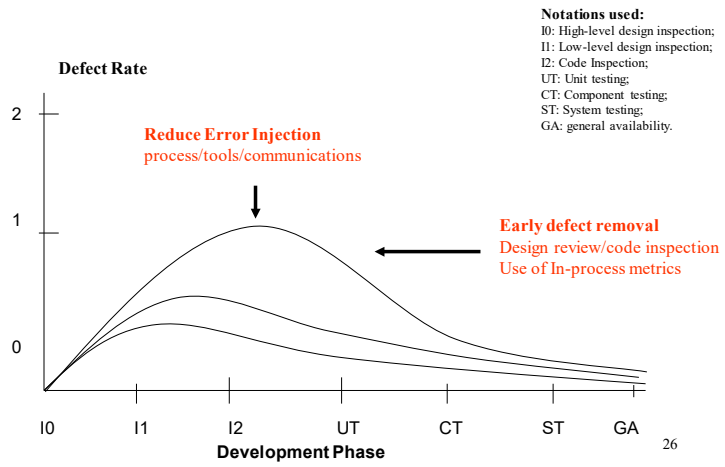
-Good ***generalized model*** for quality management.

- Quality management, in this case, is based on defects prevention and early defect removal principles.
- Assume that more defect removal at the front end of the development process will lead to a lower defect rate at later testing phases and during maintenance.

-A Rayleigh model derived from a previous release or from historical data can be used to track the pattern of defect removal of the project under development.

- If the current pattern is more front loaded than the model would predict, it is a positive sign, and vice versa.
- The overall goal of the quality management process, in this case, is to shift the peak of the Rayleigh curve to the left while lowering it as much as possible.

Rayleigh Model: Directions for Development Quality Improvement



Defect Removal Effectiveness

Defect Injection and Removal

-Defects are injected into the software product or intermediate deliverables of the product at various phases.

Development phase	Defect Injection	Defect Removal
Requirements	Requirements gathering functional specifications	Requirements analysis and review
High-level design	Design work	High-level design inspections
Low-level design	Design work	Low-level design inspections
Code implementation	Coding	Code inspections
Integration/build	Integration and build process	Build verification Testing
Unit test	Bad fixes	Testing itself
Component test	Bad fixes	Testing itself
System test	Bad fixes	Testing itself

-The total number of defects at the end of a phase is given by:

Defects at exit of a development step = defects escaped from previous step
+ defects injected in current step
- defects removed in current step

Defect Removal Effectiveness (DRE) Metrics

-Measure the overall defect removal ability of the development process.

- Can be calculated for the entire development process, for the front end (before code integration) or for each phase:
  - -When used for the front end, it is referred to as *early defect removal*
  - -When used for specific phase, it is referred to as *phase effectiveness*
- The higher the value of the metric, the more effective the development process and the fewer defects escape to the next phase or to the field.

Defect Removal Effectiveness Metrics (ctd.)

-Defined as follows:

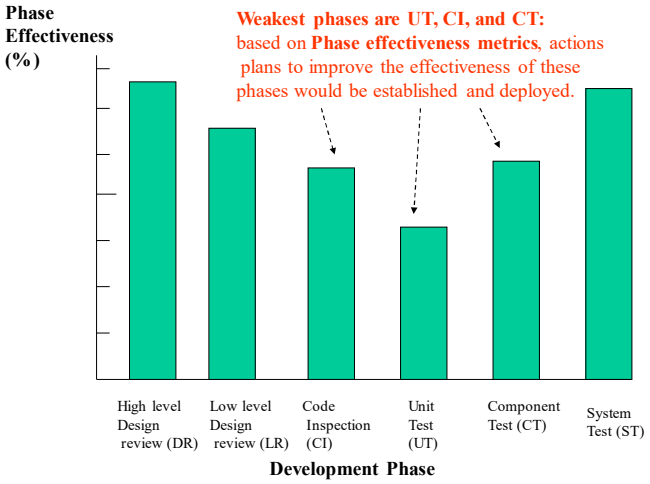
$$DRE = \frac{Defects\ \text{Removed during a Development Phase}}{Defects\ latent\ \text{in the product}} \times 100\%$$

-The number of defects latent in the product at any given phase is usually obtained through estimation as:

Defect latent = Defects removed during the phase + defects found later

Defects Latent(step i) = defects escaped (step i -1) + defects injected (step i)

Example: Phase Effectiveness of a Software Project



Defect Dynamic Model

-Defect data cross-tabulated by where found and defect origin

Defect Origin (where injected)

Where Found	RQ	HLD	LLD	Code	UT	CT	ST	GA	Total
RQ	0								0
I0	7	0							7
I1	3	8	0						11
I2	1	4	13	0					18
UT	0	1	3	63	0				67
CT	0	2	4	24	1	0			31
ST	2	6	5	37	0	1	0		51
GA	4	1	0	12	0	0	2	-	19
Total	17	22	25	136	1	1	2	-	**204**

Notations used:
I0: High-level design inspection;
I1: Low-level design inspection;
I2: Code Inspection;
UT: Unit testing;
CT: Component testing;
ST: System testing;
GA: General Availability (Field)

Exercise 3.3: Calculate the defect removal effectiveness for the different phases, and derive the overall defect removal effectiveness.

Estimating The Number of Escaped Defects using Fault Seeding

-Estimate the number of actual defects using an extrapolation technique.

-Consist of injecting a small number of representative defects into the system and measuring the percentage of defects that are found.

-Suppose that the product contains ***N*** defects and ***K*** defects are seeded, and at the end of the experiments, ***n*** unseeded and ***k*** seeded defects were found, the fault seeding theory asserts the following:

$$\frac{k}{K} = \frac{n}{N} \Rightarrow N = n\left(\frac{K}{k}\right)$$

Exercise 3.4: In a test project, the number of defects found in different phases of testing are: unit testing (UT), 163 defects; integration testing (IT), 186 defects; system testing (ST), 271 defects.

Calculate the overall defect removal effectiveness (DRE), assuming that by conducting fault seeding experimentation, 20 out of 25 seeded defects were detected by the sustaining testers in addition to the 55 (unseeded) reported during the product general availability (GA).

Calculate the value of the system test (ST) DRE.

6. Effort/Outcome Model

-Major problem with the defect removal model: the interpretation of the rates could be flawed:

- When tracking the defect removal rates against the model, ***lower*** actual defect removal could be the result of ***lower error injection*** or ***poor reviews*** and ***inspections***.
- In contrast, ***higher*** actual defect removal could be the result of ***higher error injection*** or ***better reviews*** and ***inspections***.

-To solve this problem, ***additional indicators must be incorporated*** into the context of the model for better interpretation of the data.

-One such additional indicator is the quality of process execution.

- For instance, the metric of ***inspection effort*** is used as an indicator for how rigorous the inspection process is executed.
- This metric combined with the ***defect rate*** can provide useful interpretation of the defect model.

-Specifically a 2x2 matrix, named ***effort/outcome matrix***, can be used.

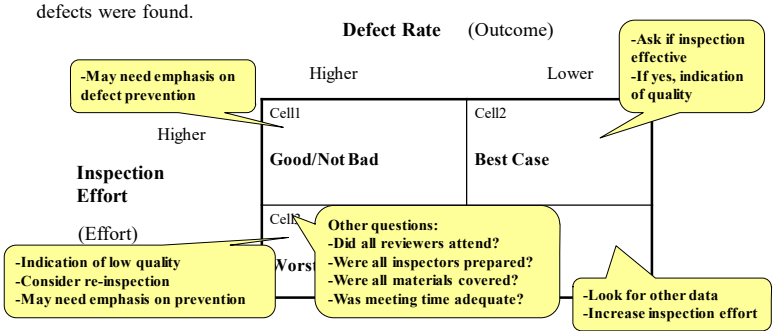
- Formed by combining the scenarios of an effort indicator and an outcome indicator.
- Can be applied to any phase of the development process with any pairs of meaningful indicators.
- The high-low comparisons are between actual data and the model, or between the current and previous releases of the product.

Effort/Outcome Model for Inspection

Defect Rate (Outcome)

Inspection Effort (Effort)	Higher	Lower
Higher	Cell1 **Good/Not Bad**	Cell2 **Best Case**
Lower	Cell3 **Worst-Case**	Cell4 **Unsure**

- Best-case scenario (Cell2): the design/code was cleaner before inspections, and yet the team spent enough effort in design review/code inspection that good quality was ensured.
- Good/not bad scenario (Cell1): error injection may be higher, but higher effort spent is a positive sign and that may be why more defects were removed.
- Worst-case scenario (Cell3): High error injection but inspections were not rigorous enough. Chances are more defects remained in the design or code at the end of the inspection process.
- Unsure scenario (Cell4): one cannot ascertain whether the design and code were better, therefore less time was needed for inspection or inspections were hastily done, so fewer defects were found.



Effort/Outcome Model for Testing

-The metrics used for test management can broadly be classified into two categories:

- ***Effort indicators***: measure the testing effectiveness or testing effort (e.g., test effectiveness assessment, test progress S curve, CPU utilization during testing etc).
- ***Outcome indicators***: indicate the outcome of the test in terms of quality, or the lack thereof (e.g., defect arrivals – total number and arrivals pattern, number of system crashes and hangs etc.).

-The effort/outcome matrix is used to achieve good test management, and effective in-process quality management.

- It is a 2x2 matrix equivalent to the one used for inspection-related metrics.

		Outcome (Defects found)	
		Higher	Lower
Effort (Testing effectiveness)	Better	Cell1 **Good/Not Bad**	Cell2 **Best Case**
	Worse	Cell3 **Worst-Case**	Cell4 **Unsure**

- Best-case scenario (Cell2): indication of good intrinsic quality of the design and code of the software – low error injection during the development – and verified by testing.
- Good/not bad scenario (Cell1): represents the situation that latent defects were found via effective testing.
- Worst-case scenario (Cell3): indicates buggy code and probably problematic designs – high error injection during the development process.
- Unsure scenario (Cell4): one cannot ascertain whether the lower defect rate is a result of good code quality or ineffective testing.