Measuring Software Quality

Part 1

Background, Introduction

Defining Quality

Quality Engineering
Contents

- Introduction
- Defining Quality
- Quality Engineering
- Summary
Contents

- Introduction
  - Defining Quality
  - Quality Engineering
  - Summary
Why is There So Much Poor Quality in Software Development?

- Organizations *focus on near-term cost or schedule* ...

... instead of looking at the *long term issues*

- Organizations *don’t know how* to produce high quality software products

- Organizations *fail to see the benefits* of using quality engineering techniques
The “Zero Sum Game” Trap

Cycle Time

Productivity

Pick Any Two

Quality
The Secret to Effective Product Development

Make the Process Efficient!

- Eliminate waste
- Eliminate mistakes
- This makes things faster, less costly, and higher in quality
Avoid the “Zero Sum Game” Trap

Don’t make these mistakes!

“to cut cost or save time you must reduce quality”;

“to improve quality you must make the product more expensive.”
Effective Quality Engineering is Fundamental to Productivity and Cycle Time Improvement
Contents

- Introduction
- Defining Quality
- Quality Engineering
- Summary
Concepts of Quality

The dictionary defines quality as:

1) “that which makes something what it is"
2) “the degree of excellence”

But is this what we mean for software?

1) “our software is what it is - that makes it a quality product"
2) “the more perfect the software the higher the quality”
e.g. Purity of tone is a quality of music

- But perhaps not in certain musical styles
- What defines the quality of “hard rock” music?

Is musical quality in the ear of the listener?
Is there a universally accepted characteristic of musical quality?
“Degree of Excellence”

Which has higher quality: a Ferrari or a Toyota Corolla??

- Which has more prestige?
- Which can go faster?
- Which uses less fuel?
- Which costs less and leaves money for other expenses?
- Which is more reliable?
- Which weighs more?
Concepts of Quality for Products

“Quality is *conformance to requirements*”
Crosby

“Quality is *fitness for intended use*”
Juran

“Quality is *value to someone*”
Weinberg
“Quality is Conformance to Requirements”

- If *testable requirements* can be established, then it is possible to run tests and evaluate whether the product satisfies the requirements.

- Thus you can avoid disputes and have workable contractual relationships.

*However* ...
Issues with “Conformance to Requirements” (1 of 4)

Who establishes the requirements?

- **Sponsor** - The one who pays for the product
- **End User** - The one who will use the product
- **Sales or Marketing** - The one who will sell the product
- **Engineering** - The ones who will design and build it
Issues with “Conformance to Requirements” (2 of 4)

Are the requirements right?
- consistent
- complete
- correct

➢ Who determines whether the requirements are right?

➢ What if you discover a problem later on?
Issues with “Conformance to Requirements” (3 of 4)

What about **implicit vs. explicit requirements**?

- **Explicit requirement**: pizza should be hot and flavorful

- **Implicit requirement**: not harmful
Issues with “Conformance to Requirements” (4 of 4)

What about when requirements change during the development process?

- Who makes the changes?
- Who controls the changes?
- Who pays for the consequences of changes?
“Quality is Fitness for Intended Use”

- This definition is based on a fundamental concept of law - that a product should be suitable for the use that it is intended for.

- This definition accommodates the fact that we may not be able to fully define the requirements.

**However ...**
Issues with “Fitness for Intended Use” (1 of 3)

Who defines *fitness*?

- Consider a TV set -- which fitness characteristics are not understood by
  - Typical User
  - Engineer
  - Sales Personnel

- Consider a software program -- which fitness characteristics are not understood by the typical software developer?
Different users have **different definitions of fitness**

- Ease of use for novices
- Control of fine details for experts
- Ease of maintenance for support staff

➤ **Uses change as users grow in experience**

- Too many “ease of use” and “automatic” features may frustrate an expert
Issues with “Fitness for Intended Use” (3 of 3)

The “pleasant surprise” concept

- User gets more than he or she expected

They really knew what they were doing when they designed this software!

There is always a balance between the engineer knowing better than the customer and the customer knowing better than the engineer.
“Quality is Value to Someone”

- This definition incorporates the idea that \textit{quality is relative}

- And it places increased emphasis on understanding \textit{what quality means to the intended user} of the software

\textbf{However} ...
Issues with “Value to Someone” (1 of 4)

Whose opinion counts?

- What features do you want?
- Can it survive spilled drinks?
- Does it have Facebook and Twitter?
- How is the financial software?
- I want hot games

➢ You may need to weigh different opinions
Issues with “Value to Someone” (2 of 4)

Logic vs Emotion
- “Glitz” v. “Substance”

Which Car is Best for Our Family?
Issues with “Value to Someone” (3 of 4)

Value depends on **What Features are Most Important?**

- Space Shuttle
  - 0 defects
  - Reliability
- Video Game
  - Good user interface
  - High performance
- School Laptop
  - Rugged
  - Fast
  - Good Software
Issues with “Value to Someone” (4 of 4)

Some Needs are Implicit (unstated)

Explicit
- I need an office
- It must have a computer
- And lots of space

Implicit
- I need a desk
- And a chair
- And convenient electrical outlets
Definitions of Software Quality

**IEEE**: The degree to which the software possesses a desired combination of attributes

**Crosby**: The degree to which a customer perceives that software meets composite expectations

Note that both definitions imply multiple expectations.
Summary of Quality Definition

Issues

- **You Must Define Quality**
  - You must define it before you can measure it
  - ... and before you can engineer it into your product

- **Quality has Multiple Elements**
  - It reflects a multitude of expectations

- **Quality is Relative**
  - Quality is in the eye of the customer

- **Quality encompasses fitness, value, and other attributes**
Question For Those Who Have Worked for a Software Development Organization

How does your organization define quality?

(Assuming it defines quality)
Another Question

How can we measure quality attributes such as fitness, value, and satisfaction of requirements?
Contents

- Introduction
- Defining Quality
  - Quality Engineering
- Summary
The Evolution of Quality

W. Edwards Deming, Joseph M. Juran

Quality Assurance

Walter Shewhart

Quality Control

See Berger, Chapter 1, for further background

1916  1950's  today  future
Walter A. Shewhart
The Father of Statistical Quality Control

Shewhart was the first to apply statistical techniques to the control of quality in manufacturing processes.

He worked for the US Bell Telephone company, originally focusing on improving the reliability of long distance transmission services.

He created many techniques still used today, notably control charts.

Walter A. Shewhart
1891-1967
Quality Control

Preventing unacceptable products from being released to the customer

- Emphasis is on finding defects and fixing them after the product has been developed.
  - This may be accompanied by suggestions for improvement

"A regulatory process through which we measure actual quality performance, compare with standards, and act on differences."

Juran
Quality Control

**Goal:** Keep Quality at an Acceptable Level by *Rejecting Unacceptable Products*

![Diagram showing the process flow of Quality Control with stages for Requirements, Development, QC Inspection (testing), and Standards of Quality. The diagram illustrates the flow from Requirements to Development, then to QC Inspection (testing), with paths for either 'Pass' or 'Fail' leading back to Requirements based on the inspection results.](image)
Headrest Story - Part I: Independence

Why go to college?
I’ll get a job at an automobile assembly plant!
I found a quality control job on the assembly line ... finding defective headrests.
Headrest Story - Part III: Excitement

The highlight of my day!!!
They switched from pink to blue!
Production rate is too low! You're too picky!

These are substandard! Pay more attention to the criteria!
Headrest Story - Part V: “Discussion”

Discussion (as used in automobile assembly lines):
Verbal communication characterized by extensive use of profanity and threats of bodily harm.
Headrest Story - Part VI: The Following Fall

My Brother
Problems with Quality Control

Keeps many defective products from being shipped to the customer, but:

- Does not always reduce the number of defects in the development process
- Does not always improve the process
- Does not always result in better products
- Does not always motivate improvement
- Often results in adversarial relationships
Quality Assurance

Assuring Product Quality: “Building Quality In”

“Providing evidence that the quality function is being performed adequately”

Juran

“Providing assurance that the product meets the customer’s quality expectations”

Worksters.com/services/qa/
W. Edwards Deming
Father of “Total Quality”

Deming championed and furthered Shewhart’s methods, introducing techniques based on *system level analysis* and psychology, such as *continuous improvement* and *reduction of rework*. His primary influence occurred after World War II.

He is credited with being a major force in the Japanese “economic miracle” after WWII, and is highly revered there, where the annual *Deming Prize for quality is considered the most important award in the field of quality.*

Deming himself has won many awards in the field of quality.
Quality Assurance

“A planned and systematic pattern of all actions necessary to provide adequate confidence that the product conforms to established technical requirements”

IEEE (George Tice)
Software Quality Assurance

A system of methods and procedures used to assure that the software product meets its requirements.

Don Riefer

- These methods and procedures include:
  - Planning, measuring and monitoring of all work performed by software engineers, software testers, etc.
Quality Assurance Looks at the Entire Process

Requirements → Development → QC Inspection

- Process and Design Standards
- Standards of Quality

Fail → Requirements
Pass
Quality Assurance is More Effective than Quality Control ...

... because the *emphasis moves to the development process*

- You attempt to fix problems *before and during* the development process

- You *continuously improve the process* and therefore reduce the number of defects in a lasting manner
However ...

- Quality improvement is still separate from other process improvement and software development activities
  
- Adversarial relationships may still be there
  - quality assurance staff vs. software developers
  - validating and testing vs. design and coding

- Motivation to improve is inconsistent

- It costs more to have people monitoring people
Quality Engineering

Similar to quality assurance in goals and methods, but **the responsibility shifts to everyone on the software development team.**

- Quality is *built into the software development process.*
  - Requirements, Design, Coding, Testing, etc.

This is a very professional and responsible approach to software development.
The Philosophical Change in Quality Engineering

- **Problems result in process changes**
  - not punishment of people

- **Finding errors is good**
  - it keeps them from leaking through to the customer

- **Everyone appreciates having a competitive development process**
  - It’s the way to remain a viable competitor

- **Measurements are used**
  - so that decisions are based on data (in addition to intuition and experience)
Quality Engineering Requires a Cultural Change

- **Pride in quality**
  - in addition to pride in product features or performance

- **Professionalism**
  - rather than fear of criticism

- **Overcoming the fear of metrics**

- Seeing software development as much more than programming and design

"We" rather than "They"
Quality Engineering Concepts

- **Build quality into the product** as part of the development process
  - Measure quality
  - Understand quality
  - Improve quality

- Engineer the *whole process* for improvements in quality, productivity and cycle time (“Process Engineering”)

A *defined process* is an essential part of the quality engineering approach
Quality Engineering Practices

- **Understand the process** and its role
- **Define and measure value and quality**
  - and focus on adding both of these to the product
- **Manage process performance**
  - through programs such as *six-sigma* or *zero defects* or *statistical process control*
- **Analyze the cost of quality**
- **Define and manage software reliability**

We will discuss some of these in later slides.
Benefits of the Quality Engineering Approach

- **Less adversarial**
  - Because everyone is responsible for quality

- **Motivation and information** to improve

- **Flexibility** to change the process in response to a problem
  - you understand the problem and its cause
  - you understand the consequences of a change in the process

*Knowledge* is the foundation of successful quality engineering
Contents

- Introduction
- Defining Quality
- Quality Engineering
- Summary
Summary

- Product development is not a “zero sum game”

- Quality must be defined in terms of things that matter to customers

- Quality assurance focuses on the process rather than simply inspecting the product

- Quality engineering focuses on the process and shifts responsibility to the whole project team
Study Questions (1 of 2)

- Your work colleague says “it’s not possible to have quality and low cost and short cycle time – you have to give up one to get the others”. This is the classic “zero sum game” argument. Explain to your colleague why this is wrong, using an example of a product that has become faster, less expensive and of higher quality over time.

- Give an example from your own experience (not the class lectures) of a situation where two different people would have entirely different definitions of what quality means for a product or service.
Study Questions (2 of 2)

- Explain to a colleague the difference between quality engineering and quality control.
References

UT Dallas

Measuring Software Quality

Part 2

Establishing Measurable Quality Attributes
Agenda

- Introduction
- Decomposition Approaches
  - Fixed Models
  - Tailored Models
- Standardized Approaches
- Pragmatic Issues
- Examples of Popular Metrics
- Summary
Agenda

Introduction

- Decomposition Approaches
  - Fixed Models
  - Tailored Models

- Standardized Approaches

- Pragmatic Issues

- Examples of Popular Metrics

- Summary
Quality Attributes are Seldom Directly Measurable

- Fitness for intended use
- Value to someone
- Satisfaction of requirements
  - Including implicit, unstated requirements
- Maintainability
- Reliability
- Supportability
- Testability
- ...

We need to find suitable ways to measure these “unmeasurable” attributes.
Surrogates

In order to measure an un-measurable attribute
  – such as “quality” or “maintainability”

We may need to measure indirectly
  – we measure something else that is associated with that attribute
  – such as “defects” or “repair cost”

This alternative, measurable attribute is called a *surrogate*. 
Surrogates Are Not the Real Thing

A surrogate may or may not accurately reflect the desired attribute

Examples:

- **Defects** may or may not reflect **quality**.
  - Defects may reflect failure to do effective testing
  - Or failure of the customer to use the product

- **Repair cost** may or may not reflect **maintainability** of the software
  - Perhaps “repair” included many changes to the software to add new features
  - Or perhaps the maintenance staff are not competent
There Are Systematic Ways to Identify Surrogates for Quality

- **Decomposition Approaches**
  - Fixed models
  - Individualized models

- **Standardized Approaches**
  - These enable comparisons of software from different organizations
  - But may not fit the desired quality characteristics of some software

There is little consensus on how to measure quality attributes, so most organizations define them in ways that fit their specific customer needs.
Agenda

- Introduction
  - Decomposition Approaches
    - Fixed Models
    - Tailored Models
- Standardized Approaches
- Pragmatic Issues
- Examples of Popular Metrics
- Summary
Decomposition Approaches
Boehm Software Quality Model

The concept here is to decompose quality attributes or factors into subfactors until you find factors that are measurable.
A Closer Look at the Boehm Model

Primary Uses
- General Utility
  - As-is Utility
    - Maintainability
    - Testability
    - Understandability
- Intermediate Constructs
  - Portability
    - Reliability
    - Efficiency
    - Human Engineering
    - Modifiability
- Primitive Constructs
  - Device Independence
    - Completeness
    - Accuracy
    - Consistency

Boehm’s Terminology
A Closer Look at the Boehm Model

Primary Uses

- General Utility
  - As-is Utility
    - Maintainability
      - Testability
      - Understandability
      - Modifiability
    - Human Engineering
      - Efficiency
      - Reliability
      - Portability
      - Device Independence
      - Completeness
      - Accuracy
      - Consistency

Quality Factors

Measurable Quality Criteria

Fenton’s Terminology
Comments on Boehm’s Model

- This is a way to decompose what we mean by “quality” until we have measurable attributes (quality criteria)

- These quality criteria are surrogates for quality
  - There are many of them
  - Some of them relate to multiple quality factors

Diagram:

- Quality Factors: Portability, Reliability
- Measurable Quality Criteria: Completeness
As you can see, it’s possible to establish a lot of criteria related to quality.
McCall Model
Top Level View – Quality Factors

Maintainability
Flexibility
Testability

Portability
Reusability
Interoperability

Correctness
Reliability
Usability
Integrity
Efficiency

PRODUCT REVISION
PRODUCT OPERATION
PRODUCT TRANSITION
As with the Boehm model, some criteria relate to multiple quality factors.

<table>
<thead>
<tr>
<th>Quality Factor</th>
<th>Measurable Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness</td>
<td>Traceability, Completeness</td>
</tr>
<tr>
<td></td>
<td>Consistency, Accuracy</td>
</tr>
<tr>
<td></td>
<td>Error tolerance, Execution efficiency</td>
</tr>
<tr>
<td>Reliability</td>
<td>Storage efficiency, Access control</td>
</tr>
<tr>
<td></td>
<td>Access audit, Operability</td>
</tr>
<tr>
<td></td>
<td>Training, Communicativeness</td>
</tr>
<tr>
<td></td>
<td>Simplicity, Conciseness</td>
</tr>
<tr>
<td></td>
<td>Instrumentation, Self-descriptiveness</td>
</tr>
<tr>
<td></td>
<td>Expandability, Generality</td>
</tr>
<tr>
<td></td>
<td>Modularity, Software system independence</td>
</tr>
<tr>
<td></td>
<td>Machine independence, Communications commonality</td>
</tr>
<tr>
<td></td>
<td>Data communality</td>
</tr>
<tr>
<td>Efficiency</td>
<td></td>
</tr>
<tr>
<td>Integrity</td>
<td></td>
</tr>
<tr>
<td>Usability</td>
<td></td>
</tr>
<tr>
<td>Maintainability</td>
<td></td>
</tr>
<tr>
<td>Testability</td>
<td></td>
</tr>
<tr>
<td>Flexibility</td>
<td></td>
</tr>
<tr>
<td>Portability</td>
<td></td>
</tr>
<tr>
<td>Reusability</td>
<td></td>
</tr>
<tr>
<td>Interoperability</td>
<td></td>
</tr>
</tbody>
</table>
Decomposition Approaches
FURPS Software Quality Model

Hewlett Packard: F.U.R.P.S.

Quality

Functionality
- Feature set
- Capabilities
- Generality
- Security
- Aesthetics
- Consistency
- Documentation

Usability
- Human factors
- Recoverability
- Predictability
- Accuracy
- Mean time to failure

Reliability
- Frequent/severity of fail.

Performance
- Speed
- Efficiency
- Resource consumption
- Throughput
- Response time

Supportability
- Testability
- Extensibility
- Adaptability
- Maintainability
- Compatibility
- Configurability
- Serviceability
- Installability
- Localizability
There are Many Other Quality Models

Dromey Model

ISO/IEC 9126

Figure 3. ISO product quality model
In Some Situations, Further Decomposition is Needed

It all depends on what quality means for the specific application.
The Next Step is Going from Measurable Surrogates to actual Metrics

ISO 9126 Quality Attribute Model
Approaches to Selecting Metrics from a Decomposition Model

- Fixed Model Approach
- Tailored Model Approach
- Standardized Approach
Fixed Model Approach

- We *accept a model* (such as Boehm or McCall or FURPS) *as suitable for our software*
  - The quality criteria
  - The quality factors
  - The suggested metrics, if any
  - The relationships among the criteria, factors and metrics
    - I.e., which factors are associated with which criteria
    - And which metrics make sense for measuring which factors

- *We collect data based on the model* to evaluate the quality of the software
Tailored Model Approach

- We **accept** a model’s *overall philosophy*
  - Quality has many attributes, attributes can be decomposed into factors, etc.

- We interview **users** (or prospective users) to determine *which quality attributes are most important* to them
  - We may use the model to prompt them or remind them of things, but we listen to what they say is important to them

- We revise **the model so that it represents our best understanding of what quality means for our specific end user population**
  - Usually guided by the original model
Advantages of the Tailored Model Approach

- You usually end up with fewer things to measure
  - Eliminating factors that are less important to the user

- It enables you to fit the end user’s needs
  - “Value to Someone” – end user defines value
  - “Suitability” – end user defines what is suitable

- It can be made to fit just about any software development methodology
  - Agile
  - Iterative
  - ...

- It can fit projects of all sizes
Drawbacks of the Tailored Approach

- You may have to define some of the metrics, as they may be unique to your situation.
- You may also have to define how to graph and interpret the metrics.
- It is hard to compare with other projects that don’t use the same model.
- You may have to explain the metrics to people unfamiliar with your specific project.

Make sure the benefits are worth the extra effort.
Agenda

- Introduction
- Decomposition Approaches
  - Fixed Models
  - Tailored Models
- **Standardized Approaches**
- Pragmatic Issues
- Examples of Popular Metrics
- Summary
Standardized Models – ISO/IEC 25010


This model is the culmination of several years of effort to standardize quality metrics.

“Software Quality is the degree to which a software product satisfies stated and implied needs when used under specified conditions.”
Other Features of the ISO/IEC 25010 Standard

- Defines the *roles of various stakeholders* in quality evaluations

- Describes a *recommended evaluation process*
  1. Establish evaluation requirements: objectives, scope, quality requirements
  2. Select measures and criteria for measurement and evaluation
  3. Design evaluation activities
  4. Conduct evaluation
  5. Conclude the evaluation: analyze results, produce reports, provide feedback, store results [for future use]
Ways to Use the Standard Model

- **Use the fixed model approach**
  - Accept the model and measure everything it specifies
  - But this may be too many measures for many situations

- **Use the tailored model approach**
  - Start with the model and tailor as appropriate
Other Uses of the Standard Model

- It can act as a checklist of things you need to think about
  - To remind you of things you might want to measure

- Many of the techniques, measures, etc. may be readily applicable or easily adapted to your specific needs

- You can use it for training

- If there are objections to your proposed measurement program, you can refer to the standard to justify what you propose to do
Advantages of the Standard Model

- The eight characteristics are claimed to be **comprehensive**
  - They cover most of the quality attributes most organizations would want to measure

- There are **standard definitions** of each of the sub-characteristics and how to measure them

- Some companies and researchers have **published results** using these metrics
  - So it is possible to **compare performance of different companies** as well as different organizations, projects, development methodologies, etc. within a company
Agenda

- Introduction
- Decomposition Approaches
  - Fixed Models
  - Tailored Models
- Standardized Approaches
  - Pragmatic Issues
- Examples of Popular Metrics
- Summary
Do I Really Need to Measure That Many Attributes?

- The various models tend to be comprehensive
  - But you may need to *use only a portion* of a model for your specific situation
  - Ultimately you need to *measure only what will actually be used* and be *useful*

- If *you don’t want to show that many metrics* to management or customers, there are ways to *combine them*
  - See next slide for an example
Compound Metrics Based on Multiple Other Metrics

If a model has many measures that apply to a specific attribute, we might devise compound metrics based on all of those.

Example:
- Suppose correctness depends on three sub-factors:
  - Completeness
  - Traceability
  - Consistency
  - Suppose each of these can be measured
- We might define correctness as follows:
  - Correctness = \( \frac{\text{Completeness} + \text{Traceability} + \text{Consistency}}{3} \)
Warning About Such Compound Metrics

- Since the individual attributes may not actually have any relationship to each other, it may not make sense to “add them together” to produce the compound metric
  - Violation of basic measurement theory

- However, for practical use, the compound metric may still be a good overall guide to the characteristic we are attempting to evaluate
  - It may simplify the way you present the results and convey the right message
  - But be aware that the numerical value may not have a precise meaning
Agenda

■ Introduction
■ Decomposition Approaches
  – Fixed Models
  – Tailored Models
■ Standardized Approaches
■ Pragmatic Issues
  ➢ Examples of Popular Metrics
■ Summary
Subjectivity vs Objectivity

An **objective measure** would be something that **everyone agrees on**

- Example: the temperature is 95° Fahrenheit

A **subjective measure** would be something that **depends on the individual user** and his or her perspective as to what is important

- Example:

  I like the noodles best of all!

  My favorite is the curry.
Software Quality is a Highly Subjective Attribute

To me, quality means accuracy – I want highly accurate calculations

I care more about ease of use. To me, quality means I can understand how to use it.
Portability – An Example of a Subjective Software Measure

\[ \text{Portability} = 1 - \frac{ET}{ER} \]

\( ET \): Effort (hours of work) needed to move the software to a new environment.

\( ER \): Effort (hours of work) needed to implement the software from scratch in the new environment.

Portability ranges from 0 to 1.
- If \( ET = 0 \), Portability = 1
- If \( ET = \frac{1}{2} \) of \( ER \), Portability = 0.5
- If \( ET = ER \), Portability = 0
This makes sense intuitively.

But someone else might want to measure portability in terms of the time required to port it.
Measures of Software, Especially Measures of Quality, are often Highly Subjective

The views and needs of the end user are important, because the end user is the ultimate judge of quality
Measures of Software Quality

Based on

Defects or Faults or Failures
Quality = Lack of Defects (or Faults or Failures)\(^1\)

The advantage of this approach is that it is often \textit{easier to measure} defects or failures than many other measures of quality

- However this approach \textit{may not capture what quality means to the end user}
  - Ease of use
  - Speed
  - ...
- And it \textit{may not reflect all that the developer considers important}
  - Maintainability
  - Supportability
  - ...

\(^1\) Defects and faults usually mean the same thing – causes of failures.
Defect Density

Defect Density = \frac{\text{Number of Defects}}{\text{Size of Software Product}}

Variations:

- **Failure Density** (instead of defects)
- **Number of Defects** can be defined in different ways:
  - **Known Defects**
  - **Total Defects** (Known Defects + Latent Defects)
- **Size of Software Product Varies**
  - There are many different definitions of size

---

1 Sometimes called “defect rate”, although this is inaccurate
2 Latent defects are defects we have not yet discovered
Defect Density Advantages

- Easily measured, compared with other options
- Gives a good, *general idea of the overall quality* of the software
- This measure has been used for over 50 years to measure software, and overall the defect density has correlated well with perceived quality of products
Defect Density Drawbacks (1 of 2)

- **People can’t always agree on what constitutes a defect**
  - Failure in operation vs mistake in the code
  - Post-release defects vs defects found during development
  - Discovered vs latent defects

- **Severity of problems caused by defects may be hard to assess**
  - Some software defects have no significant impact on customer’s perception of quality
  - Different customers use the software in different ways
Example from IBM

- Approximately *one out of three defects* will only cause a user failure *once in 500 years*.
- A *very small portion* of defects (<2%) *cause the most important user failures*.

Number of defects may not be strongly correlated to the frequency or severity of end user failures.

1 See Adams in reference list.
Defect Density Drawbacks (2 of 2)

- Different measures of the time scale
  - Entire life of product vs amount of time actually used vs processing time vs ...

- Different measures of size
  - This can make it hard to compare different projects or processes or development methods or organizations

- What is defect density telling us?
  - The quality of our product?
  - The effectiveness of our defect detection and correction process?
Despite These Drawbacks, Defect Density is Very Widely Used

Some metrics that incorporate defect density

- Cumulative defect density
  - During development or after delivery
- Total serious defects found
- Mean time to fix serious defects
- Defects found during design reviews per KLOC
- Code inspection or peer review defects found per KLOC
- System test errors found per KLOC
- Customer-discovered problems per KLOC or per product
Usability

**Formal Definition:**
Usability is the degree to which a system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.

ISO/IEC 25010 (2011)

Commonly used concepts of usability:
- User Friendliness
- Ease of use

This is a very complex concept that is hard to measure, but important to most end users.
Three Categories of Usability

- **Effectiveness**
  - Can users complete the tasks correctly?
  - Example: \( \text{Effectiveness} = \frac{\text{Quantity} \times \text{Quality}}{100} \)

- **Efficiency**
  - Time required to complete the tasks
  - Example: \( \text{Efficiency} = \frac{\text{Effectiveness}}{\text{Task Time}} \)

- **Satisfaction**
  - Degree to which the end user likes the software
  - This is a very subjective measure

\(^1\) See Fenton, section 10.3 for further details
Internal Attributes Generally Viewed as Related to Usability

These are more readily measured and can be measured before the software is released

- Good use of menus
- Good use of graphics
- Good help functions
- Consistent interfaces
- Well-organized reference manuals

Researchers have been unsuccessful in relating these to effectiveness, efficiency or customer satisfaction.

Use of these to predict usability is not recommended.
Maintainability

Formal Definition:
Maintainability is the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers.
ISO/IEC 25010 (2011)

Categories of maintenance:
- Corrective – repairing faults or defects
- Adaptive – changing to fit new hardware, networks, standards, OS versions, etc.
- Preventive – fixing problems before they result in failures
- Perfective – adding new features or improving the software in some manner
Maintainability Applies to More than Code

- Internal Documentation
- Specifications
- Designs
- End User Documentation
- Debugging Features
- ...

There are not many standards for maintainability. Measures of maintainability tend to be very specific to individual organizations.

See Fenton, section 10.4 for further details and examples.
Summary (1 of 2)

- Decomposition and standard models are two ways to break quality into a collection of measurable attributes
  - These serve as measurable surrogates for quality
  - These attributes represent different aspects of what we mean by quality
- Fixed models provide consistency and comparability between projects
- Tailored models enable a better fit to customer quality needs
- In practice, you usually don’t need to use all the metrics defined in a model
Summary (2 of 2)

- Most quality concepts involve multiple attributes

- Compound metrics can sometimes be useful to aggregate multiple quality attributes /metrics,
  - But be careful about metrics with different scales

- Measures of quality tend to be highly subjective
  - Pay attention to what matters to the customer

- Defect density, fault density and other defect-based metrics are widely used
  - They are relatively easy to measure
  - They are often helpful in understanding quality
Study Questions (1 of 3)

- Use an example to explain how decomposition can be used to select measures for quality.

- Explain the importance of the end user or customer when deciding what metrics to use for quality. Use an example.

- Explain the difference between the fixed model approach and the tailored model approach when using a decomposition model to select quality metrics.

- Give two advantages and two drawbacks of the tailored model approach.
Study Questions (2 of 3)

- Give three advantages of using a standardized quality decomposition model, using an example for each advantage.
- Explain the difference between objective and subjective measures, using an example.
- Explain the difference between a failure and a fault, using examples.
- A product is believed to have very few defects (low defect density). Give an example to explain why the product might still have several significant failures.
A product is found to have many defects. Explain why the product might not have many failures, using examples.
References (page 1 of 2)


References (page 2 of 2)


UT Dallas

Measuring Software Quality

Part 3

Software Reliability Overview
Contents

- Introduction
- Measuring Reliability
- Software Reliability Issues
  - Measuring Time
  - Application Characteristics
  - Reliability Growth
- Summary
Introduction
Reliability is the “Bottom Line” of Software Quality

- Reliability is the most conspicuous attribute of quality
- But what do we mean by reliability?

End-User's Perspective
- It does what I want
- It never fails
- etc.

What can be Measured
- It does what was specified
- Failure rates are low
- etc.

Not a Perfect Match
Reliability is Not Correctness

- Reliability means that it does what you want it to do *often enough* to be satisfactory

  Whereas

- Correctness is a binary, "yes or no" condition

- Software is almost never perfectly correct
  - But it can be highly reliable
Hardware Reliability Theory
Focuses on Materials and Production

- Assumption: failure results usually results from **physical effects**
  - breakage, wearout, fatigue, corrosion, overheating, shock, ...

- Or incorrect manufacturing processes

- The theory of hardware reliability is founded on the assumption that these are **random** events
But Product Design and Development Can Also be Factors in Quality & Reliability

Bulletproof Vest
- 2” thick, heavy duty vs.
- lightweight synthetic material

How do these options compare?
- Comfort?
- Suitability for Purpose?
- Quality?
- Reliability?
Poor Development Practices Can Lead to Hardware Failure

- The design may put undue strain on a part
  - Example: frequently used key on smart phone
    - Wears out sooner than the rest of the phone
    - Was the failure due to a faulty key?
      - or to the design of the phone, causing excessive wear on that key?

- What if the product wasn't properly tested?
  - Car overheats in the desert (never tested that severely)
Poor Software Development Practices can lead to Failure

Software failures are often attributable to software development practices

- Requirements
- Design
- Testing
- Coding
- Configuration Management
Software Reliability

“The extent to which software correctly performs the functions assigned to it”

- **Failure**: The software does not do what it is supposed to do
- **Defect** or **Fault**: The reason for the failure
  - Bad code/data/design/requirements
  - Bad configuration control
  - etc.

But as we will see, not all failures are due to defects in the software.
Design software to be *fault tolerant*

- Redundancy
- Multiple algorithms

- **This approach has been shown to have very little effect on overall software reliability**
  - The redundant code introduces more chance of error

- **It is a better fit to the hardware paradigm that involves fatigue of parts**
Develop software to be *free of defects*

- Prevention activities
- Detection activities

- This is where software experts usually concentrate their efforts

- Being free of all defects is not usually possible to achieve

- But with modern techniques of testing, quality assurance and quality engineering, it is possible to make defects relatively uncommon
But Many Failures Are Not Due to Defects in the Software

Possible Causes of Software Failures (other than defects):

- Incorrect or changing requirements
- Lack of user involvement in defining the requirements
- Unrealistic expectations
- Operator error
- Poor communication between users and developers
- Confusing or inadequate documentation
- Unexpected hardware failures
- Unexpected interaction with other software or systems
- ...

Copyright 2017, Dennis J. Frailey
Improving Software Reliability
Option 3

Study *how software fails* and focus on understanding failures

Examples of issues:

- Some failures are caused by *unexpected interactions with other systems*
- Some failures occur because *the problem is not well understood*
- Over time, software tends to become more reliable
  - *Reliability growth*
Sometimes Failures Are Due to Complex Causes

The Disappearing Warehouse

1. A major retail company had been shutting down some of its warehouses to save money.

2. A defective software program somehow erased a warehouse from the system, even though it was still active.

3. Goods destined for the warehouse were automatically rerouted elsewhere
   - Goods in the warehouse just stayed there
The Disappearing Warehouse (continued)

4. For three years, nothing arrived at or left the warehouse
   - The employees at the warehouse said nothing because they feared their warehouse would be shut down and they would lose their jobs

5. The employees kept getting paid because the payroll was handled by a different computer system

6. When upper management finally figured out what happened, they fixed the problem and continued operation as usual
   - They were embarrassed to let anyone know of this mistake
Measuring Reliability
The Goal of Measuring Software Reliability

To Predict **When or How Often** the Software Will Fail

This is the information need.

Note that *we assume it will fail* but want to know *how often* or *when*.
The Problem of Measuring Reliability

- **We cannot know** when the software will fail
  - Unless the failure was designed into the software
- **So the best we can do is to predict failure** in terms of *probabilities*
  - In a given time period:
    - *How likely is it to fail?*
    - *How likely is it to function without failure?*
  - On average:
    - *How soon will it fail?*

---

The theory of reliability is based on analysis of probabilities.
Note about Terminology

- **Terminology** for the various functions and other concepts discussed here *tends to vary* among statisticians and reliability experts

- The terminology we use in these slides matches that used in Fenton’s book

- But from time to time we will mention other terminology that is often used
Definitions

Failure
- When the product does not do what it is expected to do for a given set of input or operating conditions.

Fault (depends on author)
- A condition that causes failures.

Defect (depends on author):
- A fault found before / after product release
- Any cause of failure
- Any error, regardless of whether it is caught before release
- Other terms: bug, mistake, malfunction, etc.
More Definitions – Failure Rate

Failure Rate ($\lambda$) - the rate at which failures occur

- In some cases, $\lambda$ is a constant, such as “3 failures per thousand hours of operation”.

- In other cases, it is not a constant
  ➢ It is often expressed as a function of time:
    $$\lambda(t) = \text{<some equation involving } t\text{> }$$

In both cases, $\lambda$ represents the *most probable value*, based on what is known about the system and its operation.
More Definitions – Mean Time to Failure

Mean Time to Failure (MTTF) (\( \mu \)) - the time when the first failure is expected to occur (on average)

- In some cases, \( \mu \) is a constant, such as “520 hours”

- In other cases, it is not a constant
  - It is often expressed as a function of time:
    \[ \mu(t) = <\text{some equation involving } t> \]

In both cases, \( \mu \) represents the most probable value, based on what is known about the system and its operation.
What is Reliability?

Reliability is the *probability* that software will continue to function correctly for a given *time period* under given *conditions*

- The time period can be measured in *natural units* or *time units*
  - **Natural unit** – something that measures the amount of processing performed by the software, such as “runs”, pages of output, screens displayed, jobs completed, etc.
  - **Time units** - hours, minutes, days, weeks, etc.

*Reliability can also be measured as failure intensity* – the number of failures expected per natural unit or time unit
Measuring Reliability via Probabilities

If reliability is measured in terms of a time interval, denoted $t$, then

- $t$ is a random, failure free time interval.
- We would like to know: *how long is $t$?*
  - In other words, how long will the software function without failure?
- But since we cannot know this, we can only estimate the *probability of failure* for a given value of $t$. 
Reliability is Usually Expressed as a Function:

\[ R(t) = \text{probability of } \textit{operation without failures} \text{ in time } t \]
(i.e., in the interval 0-t)

For hardware reliability theory, there are three important assumptions about failures:

1. The system is \textit{functioning correctly} at time 0
2. Failures occur \textit{randomly}
3. Failures occur at a \textit{constant rate}, that depends on the specific hardware. This rate is usually represented by the symbol \( \lambda \)
Discussion of the Assumptions

1. The system is functioning correctly at time 0
   - This assumption makes sense for hardware and software

2. Failures occur randomly
   - This assumption makes some sense for hardware but not necessarily for software

3. Failures occur at a constant rate, $\lambda$
   - This assumption may not make sense for hardware because, as hardware ages, failures are more common
   - This assumption makes no sense for software either because of reliability growth (see later slides)
Reliability is Exponential if All Three Assumptions are True

\[ R(t) = \text{probability of operation without failures in time interval} \ 0-t \]

\[ R(t) = e^{-\lambda t} \]

\( \lambda \) is the failure rate, which is constant according to assumption 3.

This can also be expressed as:

\[ R(t) = e^{-t/\alpha} \]

\( \alpha = 1/\lambda \) is the mean time to failure
The Exponential Failure Rate is Very Convenient for Analysis

As we shall see, there are many relatively simple ways to analyze exponential data.

But software failures may not occur randomly or at a constant rate.

Nevertheless, the exponential failure rate is useful in studying software reliability.
Why Is Exponential Failure Rate Useful for Understanding Software Reliability?

1. The well-established theory of hardware reliability, which assumes exponential failure rates, provides a *framework* and a set of *terminology* and *concepts* that can be applied to non-exponential situations.

2. The exponential case is relatively *easy to explain* so it’s good for training and education.

3. Exponential rates often help with analysis of software situations even if the failure rate isn’t exponential.
Reliability Function for Exponential Distribution

\[ R(t) = e^{-\frac{t}{\alpha}} \]

Value of \( \alpha \) determines shape of curve

\( t \) = time since product release
\( \alpha \) Measures Reliability as a Constant

- \( \alpha \) is the *mean time to failure* (MTTF).
  - Actually, the mean time to the *first* failure.

- For large values of \( \alpha \), the probability of operation without failure remains high for a longer period of time

- For small values of \( \alpha \), the probability of operation without failure deteriorates quickly

If the reliability function is not exponential, there may not be a simple constant to measure reliability of the total product.
Additional Notes about Reliability

The desired value of $t$ depends a lot on the application and the priorities

- **Commercial application**
  - $t$ is large
  - the goal is to have *few failures over the life of the application* in order to keep maintenance cost low

- **Real time application**
  - e.g. an aircraft application
  - $t$ is relatively short
  - but failures in operation are critical – the goal is *zero failures* during operation of the aircraft
Failure Function or Unreliability Function

Another popular approach is to look at the probability of a failure:

$$F(t) = 1 - R(t) = \text{probability of failure in time interval } 0-t$$

- The latter is called a *failure function*§.
- It is the *cumulative distribution function* of the time interval $0-t$.
- For the exponential distribution, the failure function is:

$$F(t) = 1 - e^{-t/\alpha}$$

§ This is also known as the *distribution function* or the *cumulative density function* or the *unreliability function*. 
Graph of Exponential Failure Function

Failure Function for Exponential Distribution

\[ F(t) = 1 - e^{-t/\alpha} \]
Probability Density Function or Probability Distribution Function

This function attempts to put it in another form that means something to a user:

“(approximately) what is the likelihood that a failure will occur at time $t$”

$$f(t) = \frac{dF(t)}{dt}$$

For the exponential distribution, the formula is:

$$f(t) = \alpha^{-1} e^{-t/\alpha}$$

or

$$f(t) = \lambda e^{-\lambda t}$$
The value at 0 is $\lambda = \alpha^{-1}$. This graph is for $\alpha = 2$.

\[ f(t) = \alpha^{-1}e^{-t/\alpha} \]

\[ f(t) = \lambda e^{-\lambda t} \]

$t = \text{time since product release}$
All three functions
($\alpha = 2$)

Note that given any one of these we can compute the others.

$\mathbf{t}$ = time since product release
Conditional Failure Rate ($\lambda$)
(Hazard Function; Failure Intensity)

This is an attempt to estimate the anticipated number of times the software will fail in a given time interval, assuming no prior failures.

$$\lambda(t) = \frac{f(t)}{R(t)} = -\frac{dR(t)}{dt}$$
How to Determine $\lambda$

- $\lambda(t) = \frac{1}{\alpha(t)}$

- I.e., the higher the reliability, the lower the failure rate

- If $\lambda$ is a constant, then $\alpha$ is a constant and

  $\lambda = \frac{1}{\alpha}$
Most of the hardware-based models assume the failure rate is directly related to the number of defects remaining in the product.

But, as we've discussed, software failures are not always due to defects in the software.

Furthermore, some defects cause no failures and others cause major failures.
Problems with the Assumptions for Classic Hardware Definitions

- The classic assumption for hardware devices is that *defects are random* with respect to the structure of the product
  - But some parts of software are harder to write than others and thus more likely to have defects

- The classic assumption assumes *testing is uniform* with respect to the product
  - But with software, some parts are likely to be more effectively tested than others
More Problems with Classic Hardware Assumptions

- **All defects are equally likely to occur**
  - But for software it depends on the paths taken most often

- **All defects produce equally serious failures**
  - Clearly not the case for software

- **Testing correctly simulates normal, stressful and unusual conditions**
  - Generally this is very hard to do for software
Error Probability
Hardware vs. Software

Probability of Failure vs. Time

Software often shows reliability growth

Hardware vs. Software
The Bathtub Curve is Often A Better Description for Hardware
Software Tends to Get More Reliable Over Time Because Parts Don’t Wear Out

[In the absence of major modifications]

Defect Rate after Product Release

Early failures

Wearout

Hardware

Software
Software Reliability Issues
How to Measure Time When Evaluating Software Reliability

- The measure of time is a matter of considerable dispute
- This may dramatically affect the measure of reliability
Three Ways to Measure Time for a Software Product

- **Real Time (Calendar Time)**
  - Number of weeks or months since some event

- **Use Time**
  - Number of hours the software is in actual use

- **Processor Time**
  - Number of hours using the processor

Each of these produces different results and may fit different models
Different applications can have very different notions of reliability.
Different Applications - Different Reliability Implications

Application: Financial Transactions
Problem: Floating Point Round off Errors
Not a Problem: Excessive Time for Calculations

Application: Space Craft Local Navigation
Problem: Excessive Time for Calculations
Not a Problem: Floating Point Round off Errors

Application: Space Craft Flight Path Calculations (ground based)
Same as Financial Transactions
Reliability Growth

Software reliability generally gets better over time

- Assuming you fix bugs and don’t make major changes

This is known as the “reliability growth” phenomenon

Predicting reliability growth is very difficult

- Depends on many factors such as type of application

See Fenton, Chapter 11, for discussion of several reliability growth models
Summary (1 of 2)

- Reliability is important, but hard to measure in a way that relates to customer expectations
- Software reliability is mainly determined by development practices rather than by manufacturing or materials
- Reliability theory and principles for hardware depend on three assumptions, two of which don’t necessarily apply to software
- Nevertheless, the theory provides a framework for discussing software reliability
Summary (2 of 2)

- Three ways of improving software reliability are adding redundant code, making it defect free and studying how software fails. Of these, the first is the least effective, the second is useful but hard to do and the last is still very much a research topic.
Study Questions (1 of 2)

- Explain why software failures are not random although they often are for hardware
- Discuss the differences in how hardware fails and how software fails. Use examples.
- Give two examples of ways that software can fail that are not the result of defects in the software
- A colleague assumes that reliability means correctness. Explain the difference using an example.
Study Questions (2 of 2)

- Explain reliability growth, using an example.
- Explain why software may have reliability growth whereas hardware usually doesn’t.
- Give two examples of software applications for which the reliability requirements are different. Use different examples from any that were used in the class lectures.
References

Any Questions?
End of Lecture