UT Dallas

Software Quality and Software Testing

Part 1 – The Big Picture (How Quality Relates to Testing)
Part 2 – Measuring Software Quality
Part 3 – Software Reliability
Part 4 - Defect Containment
Part 5 – Measuring Software Complexity

UT Dallas

Software Quality and Software Testing

Part 1 – The Big Picture (How Quality Relates to Testing)

Part 2 – Measuring Software Quality

Part 3 – Software Reliability

Part 4 - Defect Containment

Part 5 – Measuring Software Complexity

Dennis J. Frailey

Retired Principal Fellow - Raytheon Company

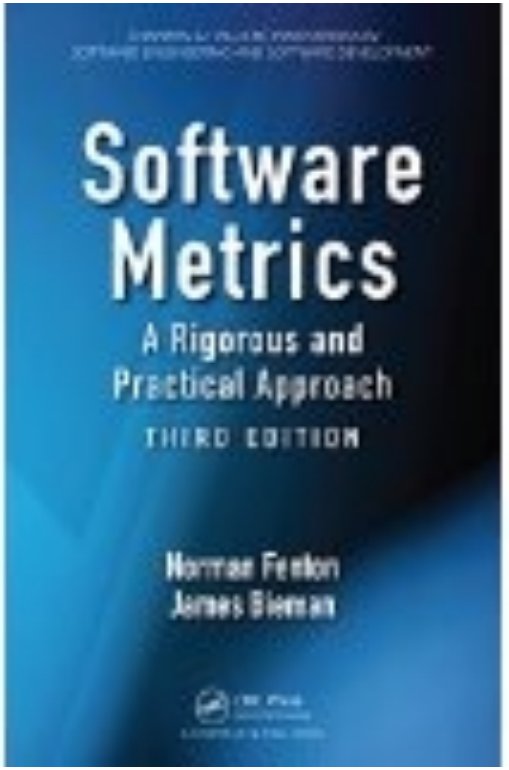
PhD Purdue, 1971, Computer Science
Assistant Professor, SMU, 1970-75
Associate Professor, SMU, 1975-77
(various titles), Texas Instruments, 1974-1997;
(various titles), Raytheon Co. 1997-2010
Adjunct Associate Professor, UT Austin, 1981-86
Adjunct Professor, SMU, 1987-2017
Adjunct Professor, UT Arlington, 2014-present

Areas of specialty: software development process, software project management, software quality engineering, software metrics, compiler design, operating system design, real-time system design, computer architecture

A Recommended Book on Measurement

Some of the material covered today is taken *from this book*.

Although not a book on testing, it is a very good book on measurement and addresses several aspects of testing.



Software Metrics – A Rigorous and Practical Approach
By Norman Fenton and James Bieman

More Recommended References

***SWX – The Software Extension to the Project Management Body of Knowledge*, available from PMI (www.pmi.com) and the IEEE Computer Society (www.computer.org).**

- This is a general reference that may be important if you want to apply some of today's techniques in project management.

***SWEBOK – The Guide to the Software Engineering Body of Knowledge*, available from the IEEE Computer Society and also at www.swebok.org**

- This is another general reference that gives an overall picture of software engineering knowledge and summarizes topics that any software engineer should know about.

Part 1

The Big Picture – How Quality Relates to Testing and Other Aspects of Software Engineering

Test and Evaluation

***Evaluation*: Appraising a product through one of the following:**

- Examination, analysis, demonstration
- Testing
- or other means

***Testing*: Exercising a system to improve confidence that it satisfies requirements or to identify variations between desired and actual behavior.**

"Evaluation" is the broader term.

But What Are We Appraising? What is "Desired Behavior"?

- Satisfies requirements
- Works correctly
- Does what I want it to do
- Does no harm
- Reliable – I can depend on it
- Easy to use
- Portable
- Easy to update and maintain
- Easy to test
- Runs efficiently / fast
- Consistent
- …

Can we test for these characteristics?

Can we measure them?

Measurement is Often Involved in How We Test Software

Requirement	How we might Test it
▪ Software must handle up to 10 transactions per minute	➢ Measure how many transactions it processes per minute
▪ Software must produce the correct output	➢ Run the software on 1000 different test cases and count how many produce correct output
▪ Software must be easy to use	➢ Have 25 people use the software and rate how easy it is to use
▪ Software must be easy to test	➢ Run standard test procedures and measure how long it takes

But What Are We Testing?
What is "Desired Behavior"?

- Satisfies requirements
- Works correctly
- Does what I want it to do
- Does no harm
- Reliable – I can depend on it
- Easy to use
- Portable
- Easy to update and maintain
- Easy to test
- Runs efficiently / fast
- Consistent
- …

These are all characteristics of Software Quality

I.e., testing is one way to assess software quality.

And measurement is often part of testing.

Guide to the Software Engineering Body of Knowledge

Editors

Pierre Bourque
Richard E. (Dick) Fairley

Downloadable at:
www.swebok.org

IEEE computer society

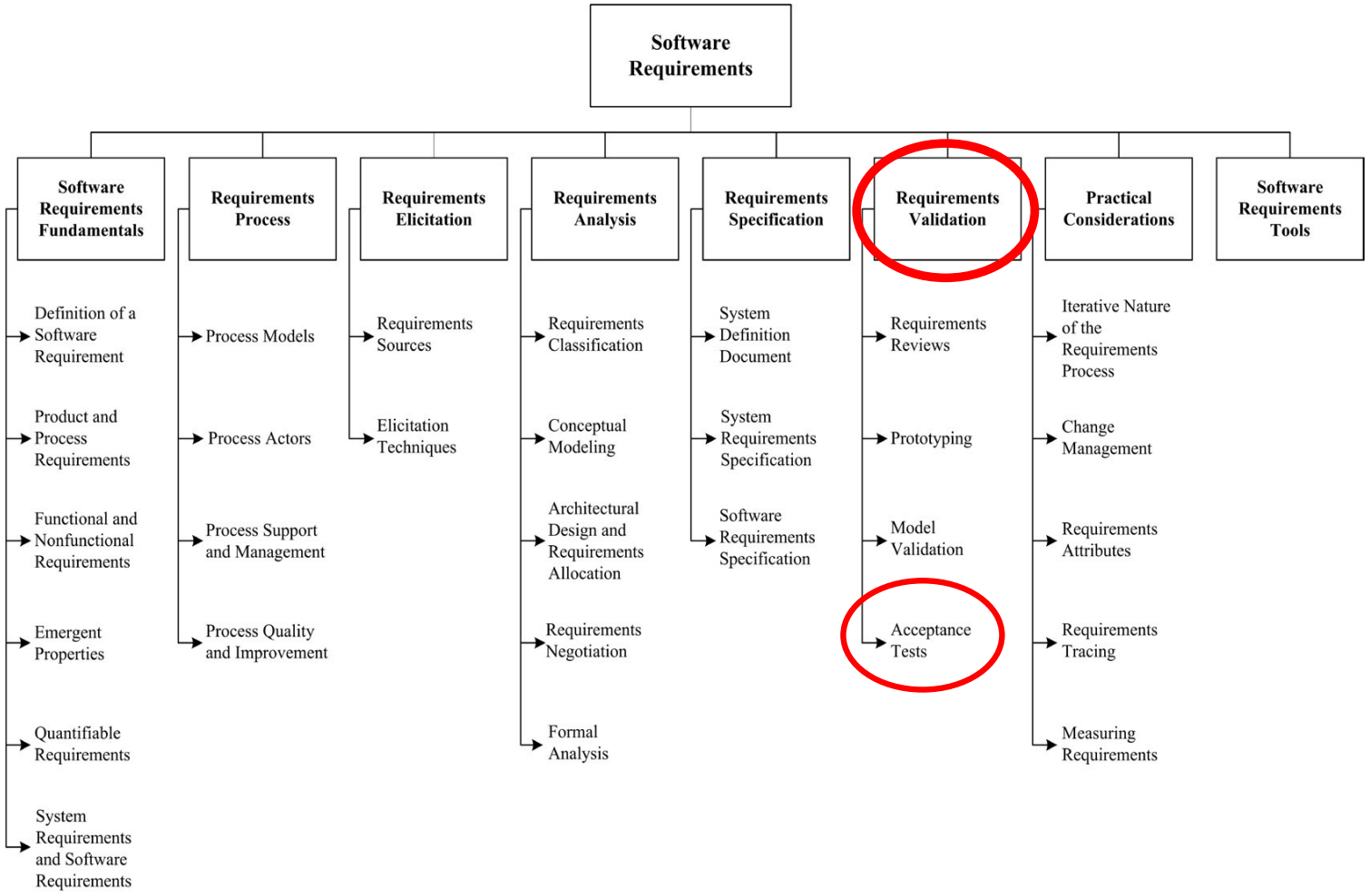
SWEBOK Facts

- **3 Editions have been produced since 1998**
- **2 Editors: Pierre Bourque and Richard Fairley**
- **8 Contributing and Co-Editors**
- **15 Knowledge Areas, each with its own Editors**
  - Each aligned with related ISO and IEEE standards
- **9-person Change Control Board**
- **Over 300 reviewers (chosen due to their expertise in various aspects of software engineering)**
  - Over 1500 comments received and adjudicated on various drafts (3rd edition)
- **36 Items in Consolidated Reference List**

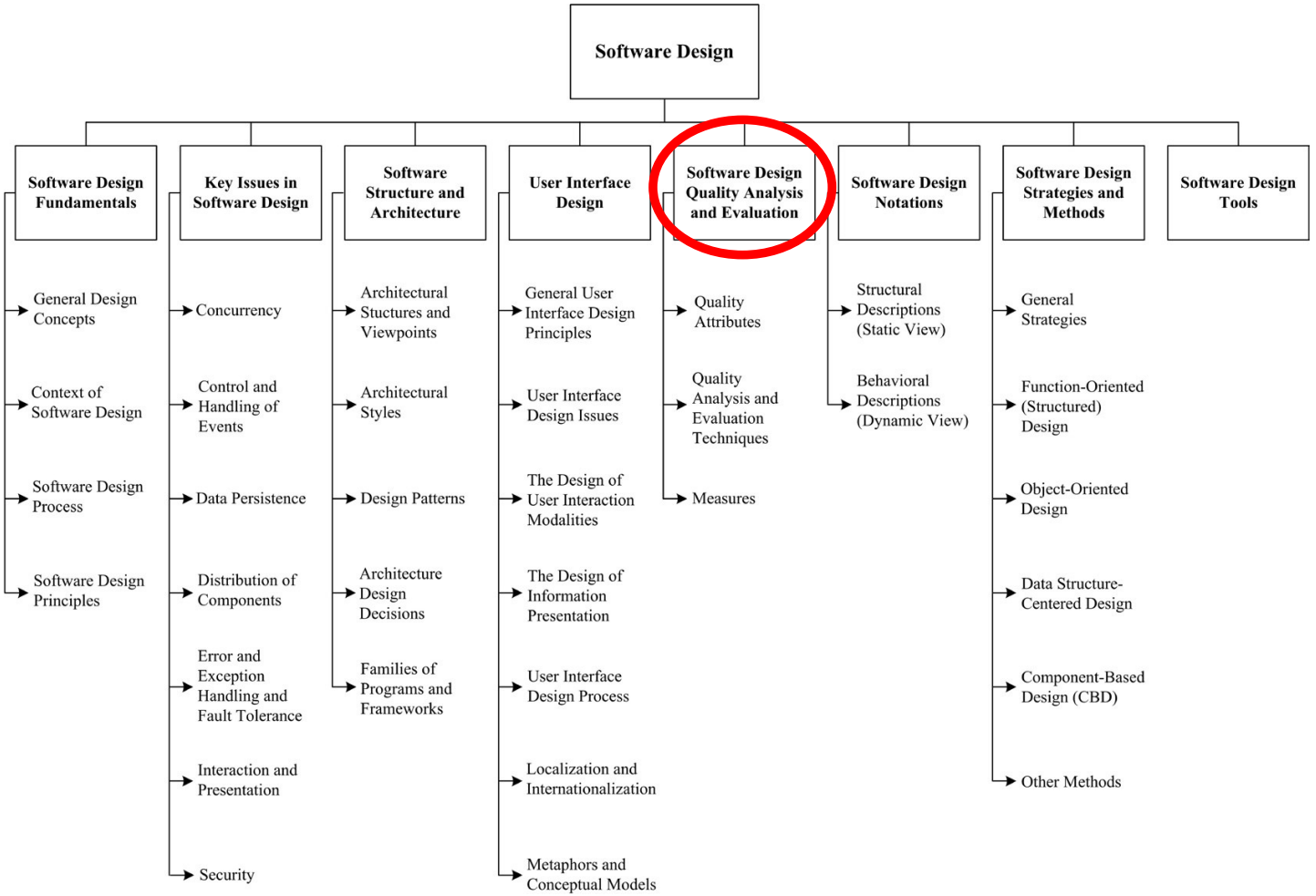
The 15 SWEBOK Knowledge Areas

- Software Requirements
- Software Design
- Software Construction
- Software Testing
- Software Maintenance
- Software Configuration Management
- Software Engineering Management
- Software Engineering Process
- Software Engineering Models and Methods
- Software Quality
- Software Engineering Professional Practice
- Software Engineering Economics
- Computing Foundations
- Mathematical Foundations
- Engineering Foundations

Software Requirements

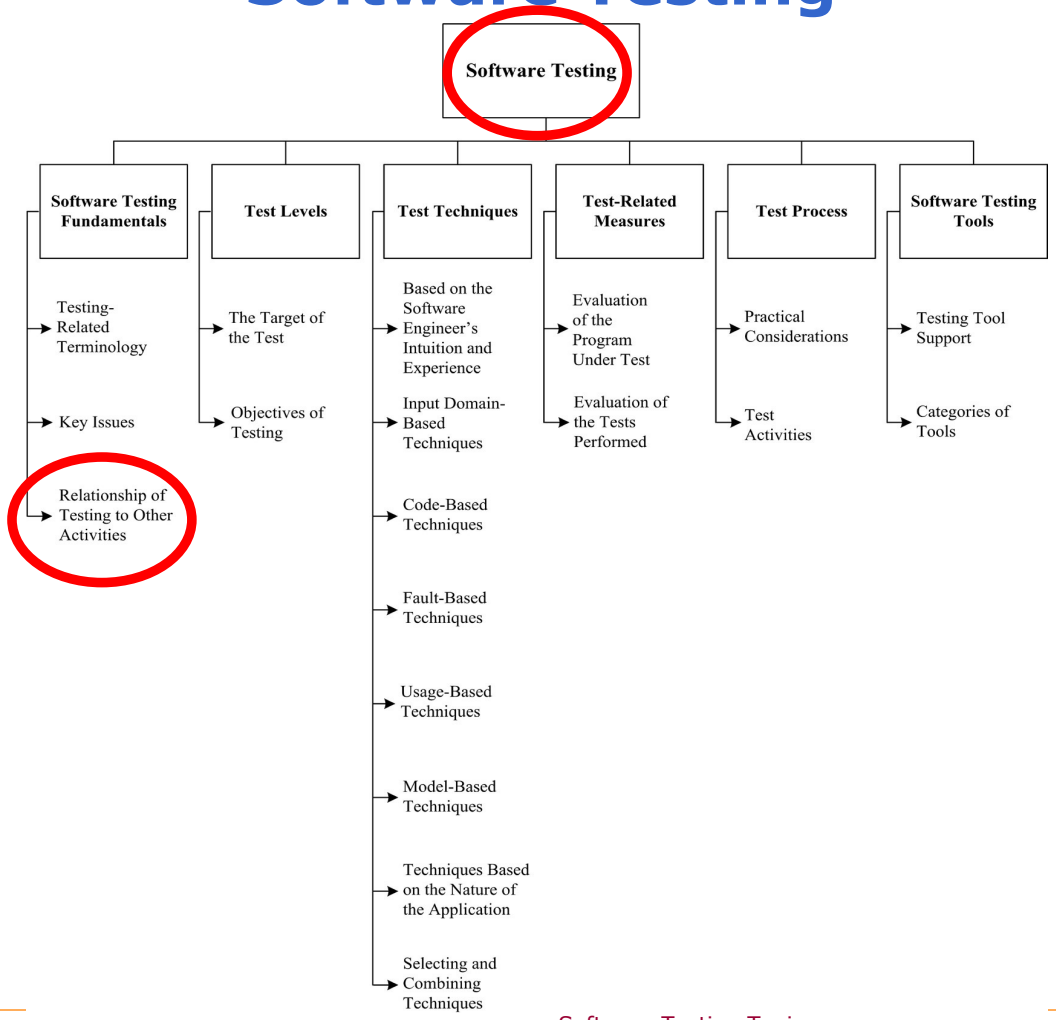


Software Design

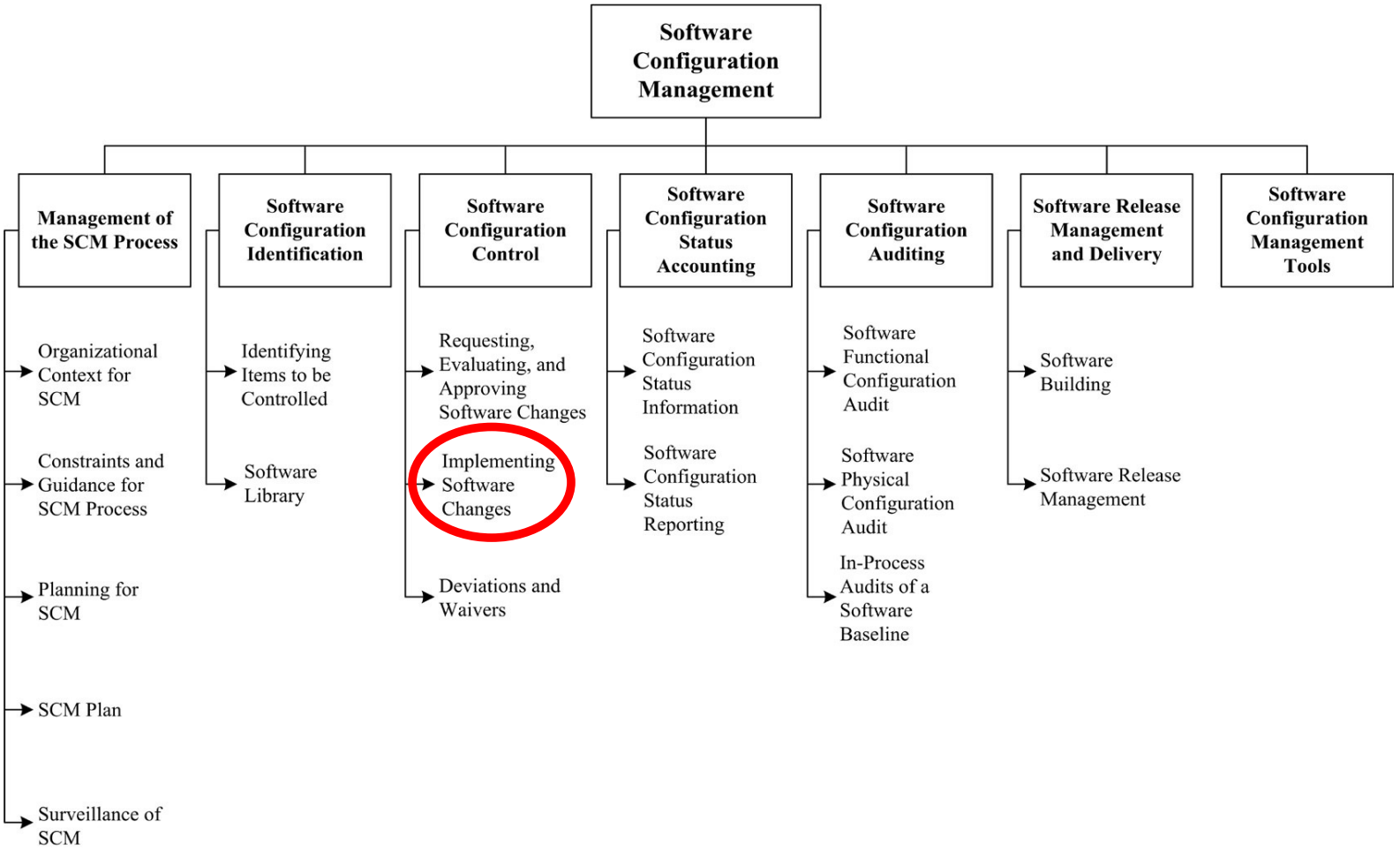


Software Construction

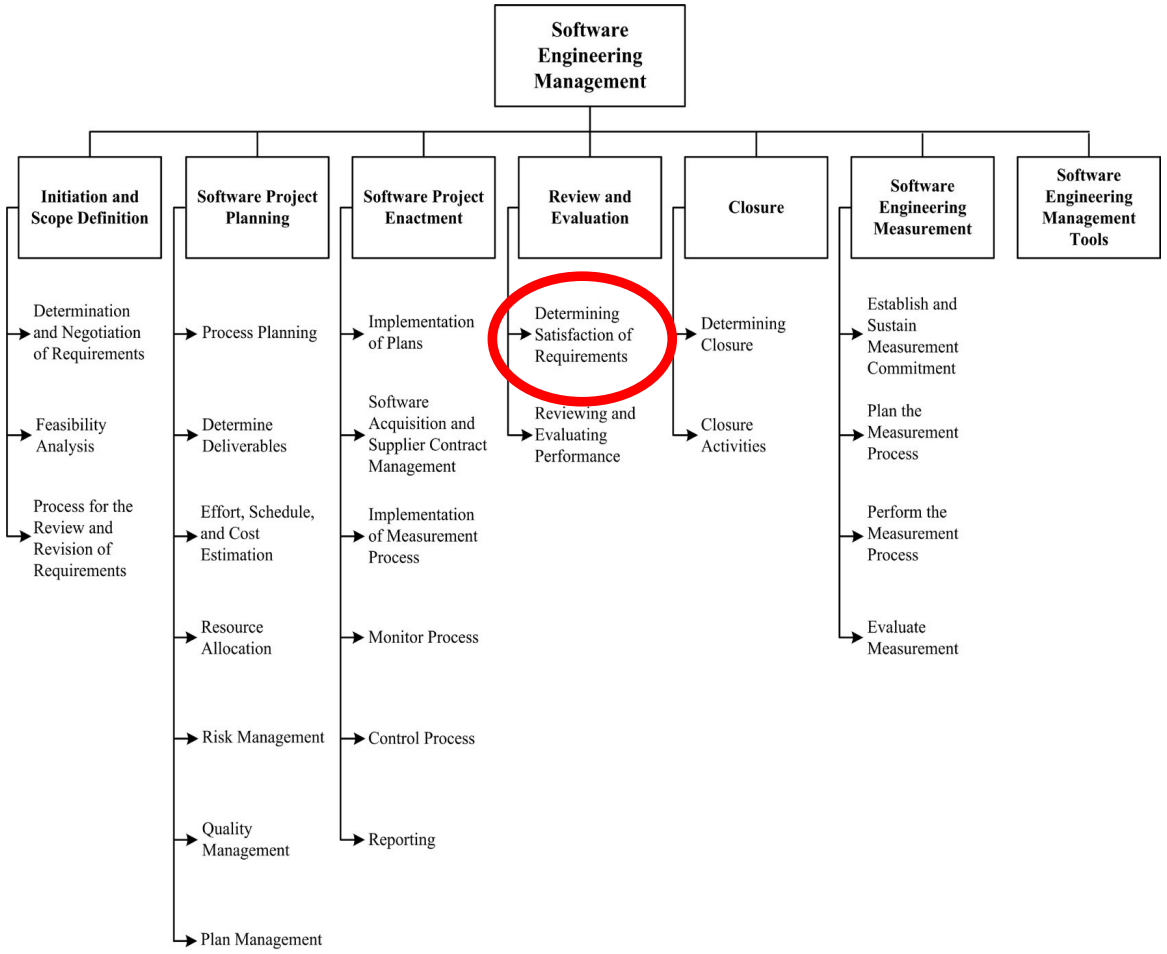
Software Testing



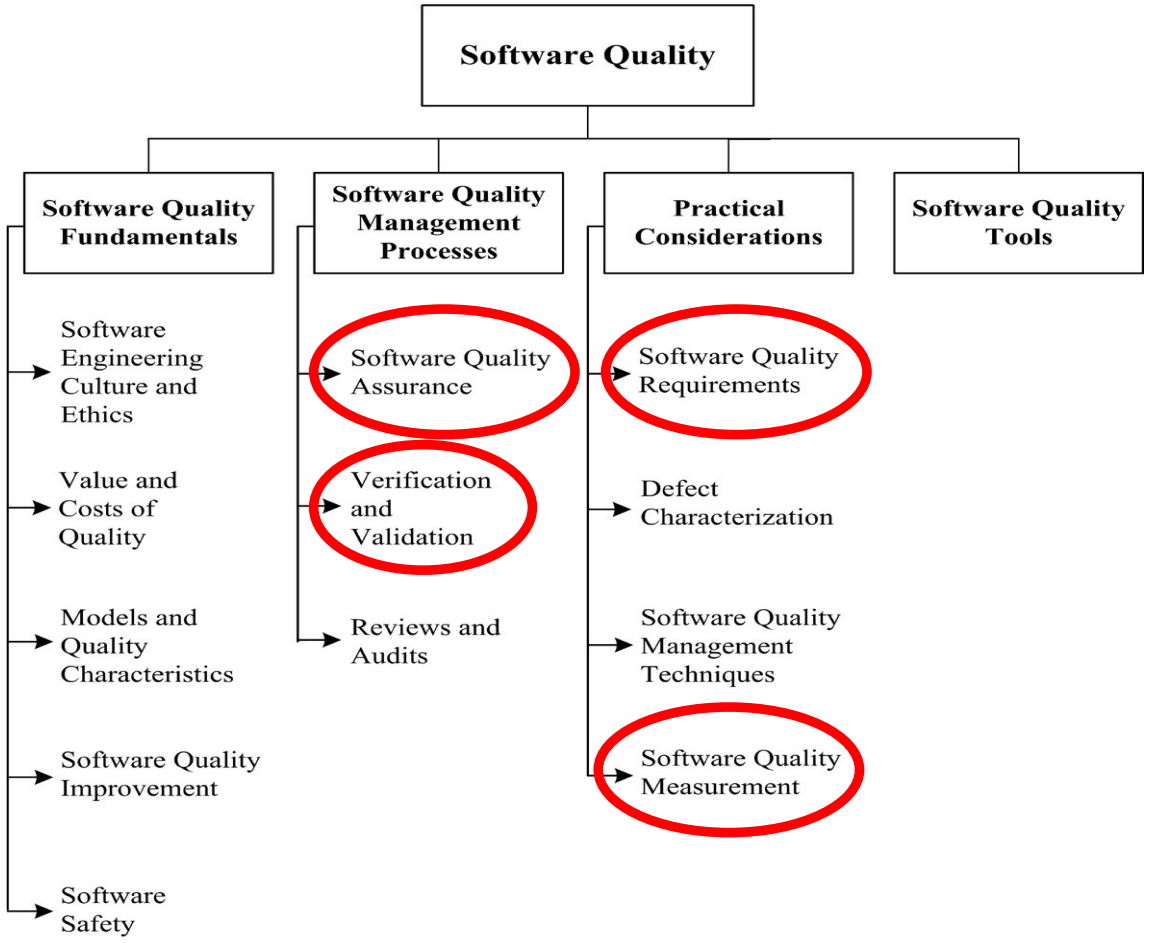
Software Configuration Management



Software Engineering Management



Software Quality



What Do We Mean by Quality?

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 decide *whether the product satisfies the requirements* – by testing it.**
- **If *measurable quality characteristics* can be established, then it is possible to decide on *the extent to which the product satisfies the requirements* – by measuring it.**
- **Thus you can avoid disputes and have workable contractual relationships**

HOWEVER …

Issues with "Conformance to Requirements" (1 of 4)

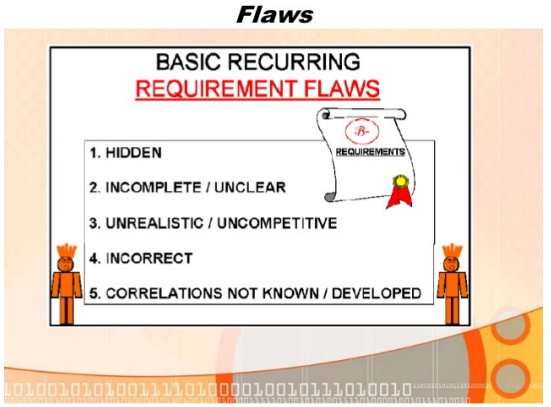
Who establishes the requirements?

- ***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)

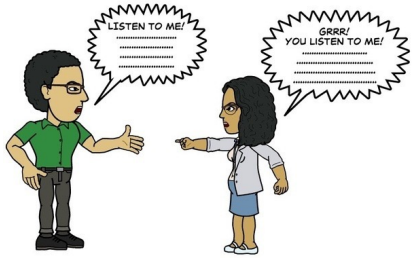
Are the requirements right?

- consistent
- complete
- visible
- correct



Slideshare.net

- **Who determines whether the requirements are right?**



Quora.com

- **What if you discover a problem later on?**

Issues with "Conformance to Requirements" (3 of 4)

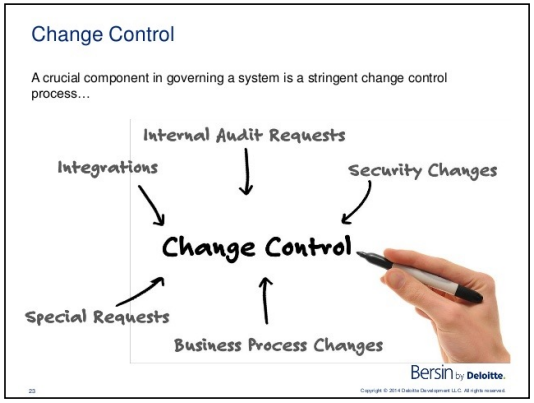
What about implicit vs. explicit requirements?

- ***Explicit requirement***: pizza should be hot and flavorful
- ***Implicit requirements***:
  - **comes sliced in reasonably sized pieces**
  - **not harmful**
  - **fits in the pizza box**
  - **…**

Issues with "Conformance to Requirements" (4 of 4)

What about when requirements change during the development process?

- ***Who makes the changes***?
- ***Who controls and authorizes 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 4)

Who defines *fitness*?

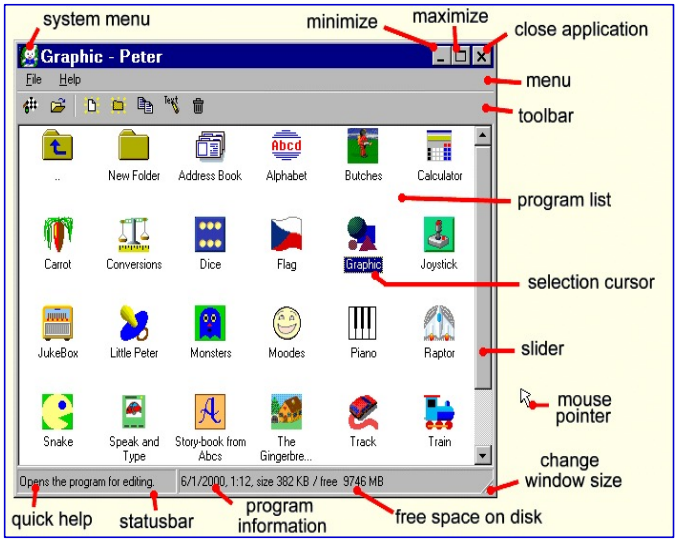
- Consider a TV set
  - which fitness characteristics are not understood by
    - Typical User
    - Engineer
    - Sales Personnel

Konga.com

Issues with "Fitness for Intended Use" (2 of 4)

Who defines *fitness*?

- Consider a software program
  - which fitness characteristics are not understood by
    - The typical software developer?
    - The inexperienced end user?
    - The experienced end user?



Gemtree.com

Issues with "Fitness for Intended Use" (3 of 4)

Different users have different definitions of fitness

- Ease of use for novices
- Control of fine details for experts
- Ease of maintenance for support staff
- Able to survive power failures
- Compatibility with previous system

Theodysseyonline.com

- **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" (4 of 4)

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 often tension 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 *quality is relative***
- **And it places increased emphasis on understanding *what quality means to the intended user* of the software**

HOWEVER …

Issues with "Value to Someone" (1 of 4)

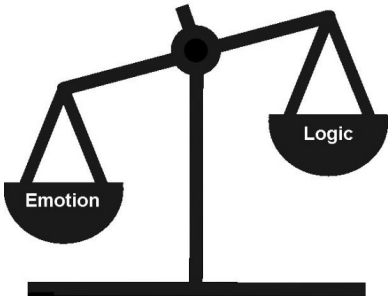
Whose opinion counts?

- You may need to weigh different opinions

Issues with "Value to Someone" (2 of 4)

Logic vs Emotion

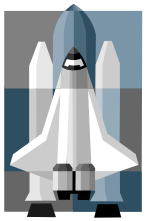
- "Glitz" v. "Substance"



Issues with "Value to Someone" (3 of 4)

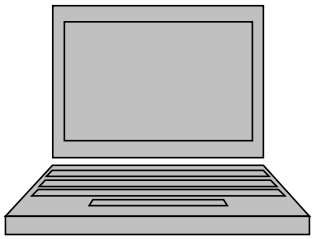
Value depends on What Features are Most Important

- Space Shuttle
  - 0 defects
  - Reliability



- Video Game
  - Good user interface
  - High performance

- School Laptop
  - Rugged
  - Fast
  - Good Battery Life
  - Good Software

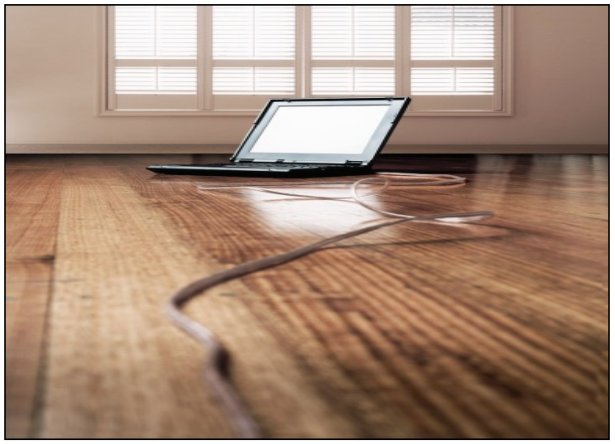


Issues with "Value to Someone" (4 of 4)

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***
  - Before you can **engineer it** into your product
  - … and before you can **measure it**
  - … or **test** whether the product has the desired quality attributes
- **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***

So How do We Test or Measure Quality?

- **We will cover this in Part 2.**
- **But first, a few more thoughts about testing.**

Observations on The Overall Testing Process

Test and Evaluation

***Evaluation*: Appraising a product through one of the following:**

- Examination, analysis, demonstration
- Testing
- or other means

***Testing*: Exercising a system to improve confidence that it satisfies requirements or to identify variations between desired and actual behavior.**

"Evaluation" is the broader term.

Testability

A product is testable if:

- It can be tested in a reasonable way (readily testable)
- The tests are well defined, comprehensive, and not overly redundant
- Each test can be directly traced to and from:
  - product requirements,
  - derived requirements resulting from design decisions, or
  - design or coding elements calling for specific testing
- Each test failure can be directly traced to:
  - a requirement that is not being met, or
  - A design element that was not properly implemented, or
  - A portion of the code that has a programming error

Good testing starts with testable requirements and designs.

Testing is unsuitable when ...

- It would destroy the product
- It is too dangerous
- It is too costly
- It cannot reasonably be expected to provide confidence that requirements are satisfied
- It cannot be done

Evaluation Techniques
(other than testing)

- ***Examination***
  - For example, reading designs or code or other documents to check for errors
- ***Demonstration***
  - e.g. flying an airplane to show that it can fly
  - e.g. running a program to show that it works
- **Other techniques (examples)**
  - providing a ***formal proof*** that a program is correct
  - showing through ***statistical analysis*** that the probability of a defect is below a threshold

The Steps Involved in a Good Testing Process

- ***Preparation***
- ***Test Execution***
- ***Repair*** **of defects (debugging)**

Test Preparation Activities

- **Making sure that requirements are testable**
- **Making sure that designs are testable**
- **Developing test plans**
- **Developing test cases**
- **Writing testable code**
- **Writing test code (or programming test machines)**
- **Devising procedures for testing, inspecting and reviewing of results**

These activities begin as requirements are being defined, and continue throughout the development process

Reasons why Requirements/Designs May be Hard to Test

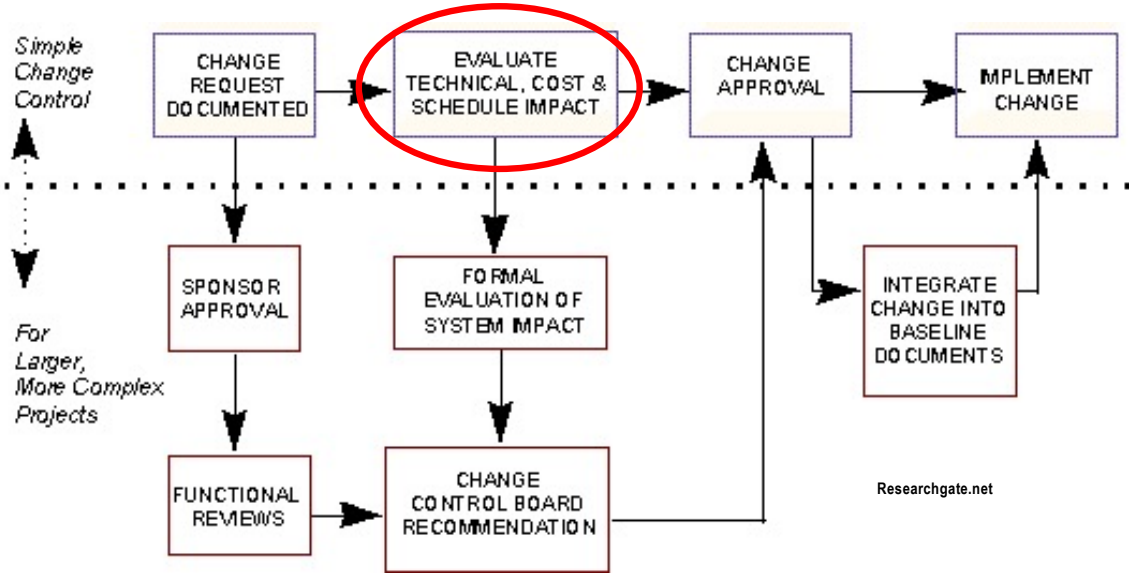
- **Requirements may not be well understood**
- **Requirements may not be well documented**
- **What seems obvious to the customer or the system designer may not seem clear or obvious to the software developer or tester**
  - Different kinds of knowledge
  - Unstated assumptions
- **The customer and the software developer may not agree on what constitutes an acceptable test**
- **Changes made during software development may not be communicated to the software team**

Suggestions (slide 1 of 3)

- **A requirement or design feature is not complete until you have reached agreement on how it is to be tested**
  - For each requirement, reach agreement between the software team and the customer or system engineer on how the requirement is to be tested
  - For each design feature, reach agreement between the software designer and the software test team on how the design feature is to be tested

Suggestions (slide 2 of 3)

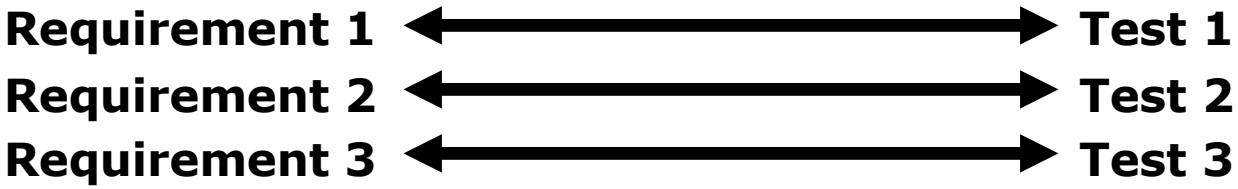
- **Control changes to requirements and design**
  - Don't allow a requirements or design change without a clear understanding of the effect of the change on the software cost, schedule and technical development
  - For each change to requirements or design, indicate how the corresponding tests must be changed.



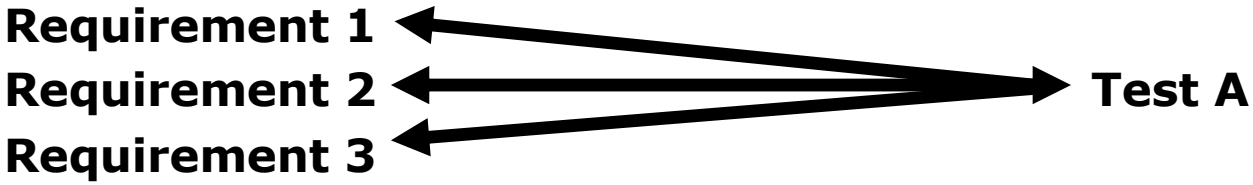
Suggestions (slide 3 of 3)

- **Keep track of which tests correspond to which requirements or design elements *(traceability)***

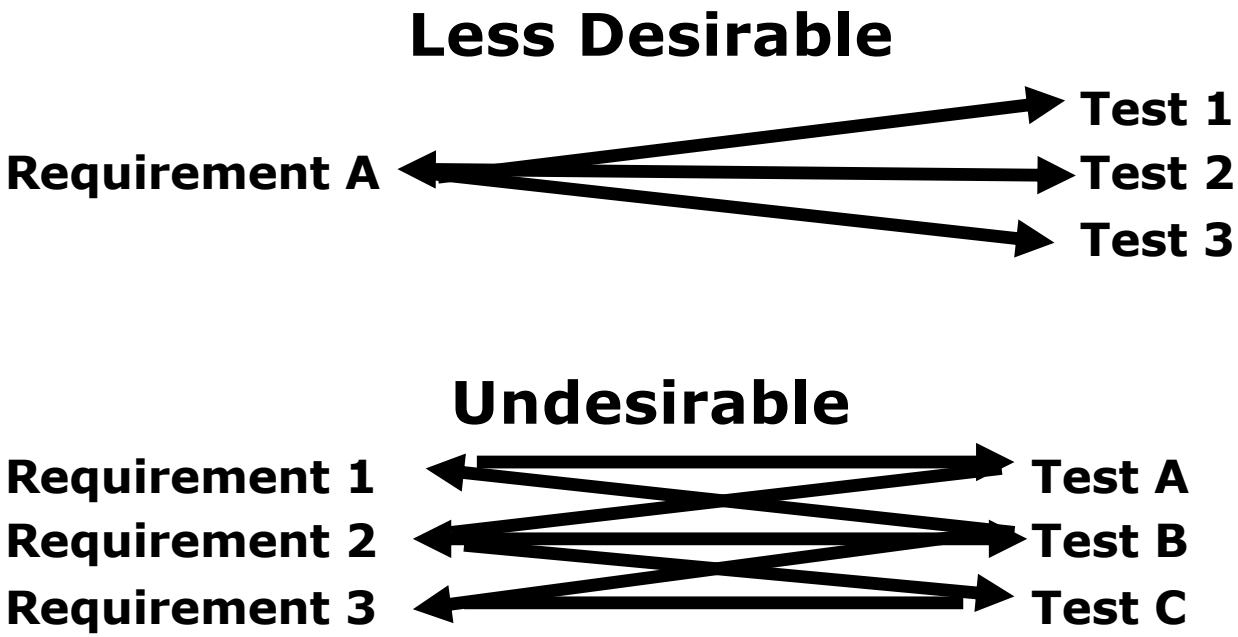
Ideal



Acceptable

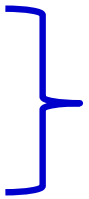


Other Traceability Options



Reasons Why Code May Be Difficult to Test

- **Code is not well structured**
  - Needlessly complex
  - Poorly organized



We will address this in parts 3 and 4

- **Code elements do not trace directly to requirements or design elements**
  - So when the code causes a failure, it is hard to determine whether the problem is with the code or the design or the requirement

- **Code is not well documented or does not follow coding conventions**
  - Hard to understand
  - Error prone

Sample Outline of a Test Plan

- **Summary of Major Testing and/or Integration Steps**
- **For each test and/or integration step:**
  - Purpose / goal of the step
  - What equipment is needed and what configurations must be set up
  - What hardware elements will be integrated/tested at this step
  - What software components will be integrated/tested at this step
  - Test cases to be performed (in order, if order is important)
    - For each test case:
      - what requirements will be tested and/or purpose of the test
      - what procedures should be followed
      - what results are expected

Ideally, this is started at the beginning of a project, with details filled in and revisions made as the project progresses

Sample List of Test Cases

Test Case ID	Test Case Name	Summary	Expected Results
S2R1	Get GPS Data	Pull the GPS data from the processing unit	**The data should match the values given by the GPS receiver. OR, if a GPS receiver is not available, then the data should match the canned data provided for testing purposes.**
S2R2	Get Radar Data – Raw A/D Samples (reduced range swath)	Pull the radar data from the processor. Format expected is the raw A/D samples	**The data should match the values given by the processor. (Details TBD.)**
S2R3	Get Radar Data – Decimated A/D Samples (full range swath)	Pull the radar data from the processor. Format expected is the decimated A/D samples.	**The data should match the values given by the processor. (Details TBD.)**
S2R4	Get Radar Data – Pulse Compressed Data	Pull the radar data from the processor. Format expected is the pulse compressed data.	**The data should match the values given by the processor. (Details TBD.)**
S2R5	Get Radar Data – CPI Range-Doppler Maps	Pull the radar data from the processor. Format expected is the CPI Range-Doppler maps.	**The data should match the values given by the processor. (Details TBD.)**
S2R6	Get Radar Data – Post NCI Range-Doppler Maps	Pull the radar data from the processor. Format expected is the Post NCI Range-Doppler Maps.	The data should match the values given by the processor (Details TBD).
S2R7	Get Radar Data – Exceedence Regions	Pull the radar data from the processor. Format expected is the exceedence regions.	The data should match the values given by the processor (Details TBD).
S2R8	Get System Health Information	Pull the radar data from the processor. This can be a dummy dwell, but we need to check the header information to ensure the system health status is working.	**System Health information**

Test Execution Activities

- **Conducting tests**
- **Conducting reviews of test results**
- **Conducting inspections of procedures or code**

These are the steps where actual testing is performed.

Loginworks.com

Repair Activities

- **Debugging (finding the cause of each test failure)**
- **Correcting errors**
- **Rerunning tests, inspections, etc.**

These can be very expensive activities if testing is not planned and performed well.

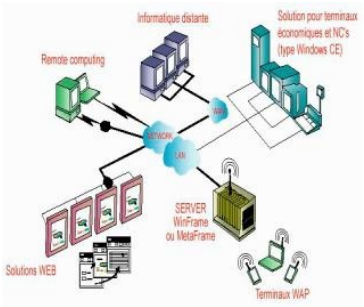
Re-running of tests can add significant cost and time to a project

Measuring the Progress of a Testing Activity

Testing Requires Resources

Resources are entities required in order to perform software processes and produce software products

- People
- Computers
- Software
- Networks
- Time
- …

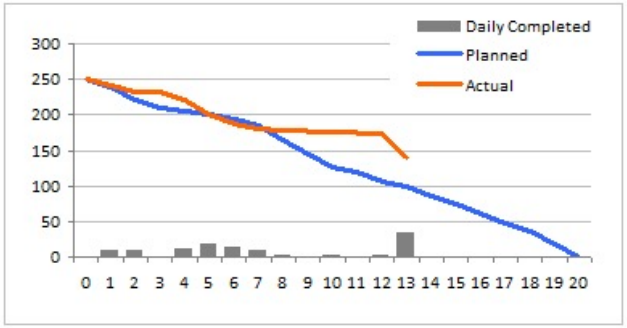


Resources usually cost money

- We want to ***use them efficiently*** – not waste them.
- And we want them to be **available**!

Some of the Things We Wish to Know About Testing Resources

- **Are they available as required?**
  - Staffing levels / employee turnover rates
  - Training (frequency, suitability)
  - Equipment and software availability
  - Network bandwidth
- **Are they performing as desired?**
  - Are testing facilities and tools working well?
  - Is the training effective?
- **Are the resources being used efficiently?**
  - Are we on schedule? Will the project be on time?
  - Are we over or under our budget?
  - What is our productivity?



Resource Measures are Important for Managing a Project

- **They tend to be focused on costs and schedules relative to plans or deadlines**
- **For example many projects use a work breakdown structure to measure project progress**
- **Other examples of resource measures that tell us about project status**
  - Earned value / Burndown Charts
  - PERT and GANTT charts (project status and plans)
  - Employee or network workload measures
  - Employee or equipment availability measures

Task Name	Duration	Start	Finish
Product Development	310 days	Mon 1/10/11	Thu 3/15/12
Develop Project Team	30 days	Mon 1/10/11	Fri 2/18/11
Planning	14 days	Mon 2/21/11	Thu 3/10/11
Define Development Environment	10 days	Mon 2/21/11	Fri 3/4/11
Define Functional Level Requirements	14 days	Mon 2/21/11	Thu 3/10/11
Development Environment Setup	3 days	Fri 3/11/11	Tue 3/15/11
Setup workstations at chosen location	3 days	Fri 3/11/11	Tue 3/15/11
Design	48 days	Fri 3/11/11	Tue 5/17/11
Database Design	12 days	Fri 3/11/11	Mon 3/28/11
Identify Database Requirements	1 day	Fri 3/11/11	Fri 3/11/11
Identify Database Entities	3 days	Mon 3/14/11	Wed 3/16/11
Document Database Design	3 days	Thu 3/17/11	Mon 3/21/11
Design Database Entities	5 days	Tue 3/22/11	Mon 3/28/11
Algorithm Design	5 days	Fri 3/11/11	Thu 3/17/11
Donor Notification Algorithm	5 days	Fri 3/11/11	Thu 3/17/11
Fundraiser Suggestion Algorithm	5 days	Fri 3/11/11	Thu 3/17/11
Interface Design	48 days	Fri 3/11/11	Tue 5/17/11
Development	110 days	Wed 5/18/11	Tue 10/18/11
Database Architecture	10 days	Wed 5/18/11	Tue 5/31/11
Mobile Donation Interface	10 days	Wed 5/18/11	Tue 5/31/11
Facebook App	5 days	Wed 5/18/11	Tue 5/24/11
uRaiSe Web Site	110 days	Wed 5/18/11	Tue 10/18/11
Web Services	20 days	Wed 5/18/11	Tue 6/14/11
Notification Services	2 days	Wed 5/18/11	Thu 5/19/11
Email	1 day	Wed 5/18/11	Wed 5/18/11
SMS	2 days	Wed 5/18/11	Thu 5/19/11
Twitter	2 days	Wed 5/18/11	Thu 5/19/11
Facebook	2 days	Wed 5/18/11	Thu 5/19/11
Testing	108 days	Wed 10/19/11	Thu 3/15/12
Integration Testing	30 days	Wed 10/19/11	Mon 11/28/11
System Testing	30 days	Sat 10/22/11	Thu 12/1/11
Acceptance Testing	25 days	Fri 12/2/11	Thu 1/5/12
Alpha Testing	25 days	Fri 1/6/12	Thu 2/9/12
Beta Testing	25 days	Fri 2/10/12	Thu 3/15/12

Tutorialspoint.com

Resource Measures Often Measure People

- **This can lead to *problems if people are not measured fairly***
  - People are very sensitive to fairness of measurements
- ***Productivity* of people is an *especially problematic* thing to measure**
  - **The person doing the hardest job tends to look like they are making the least progress**
- **Even measuring things like *defects* can be *misleading when applied to people***
  - The person developing the most complex part of the software tends to have more errors
  - The person testing the most difficult part of the software tends to discover the most defects and to take the most time

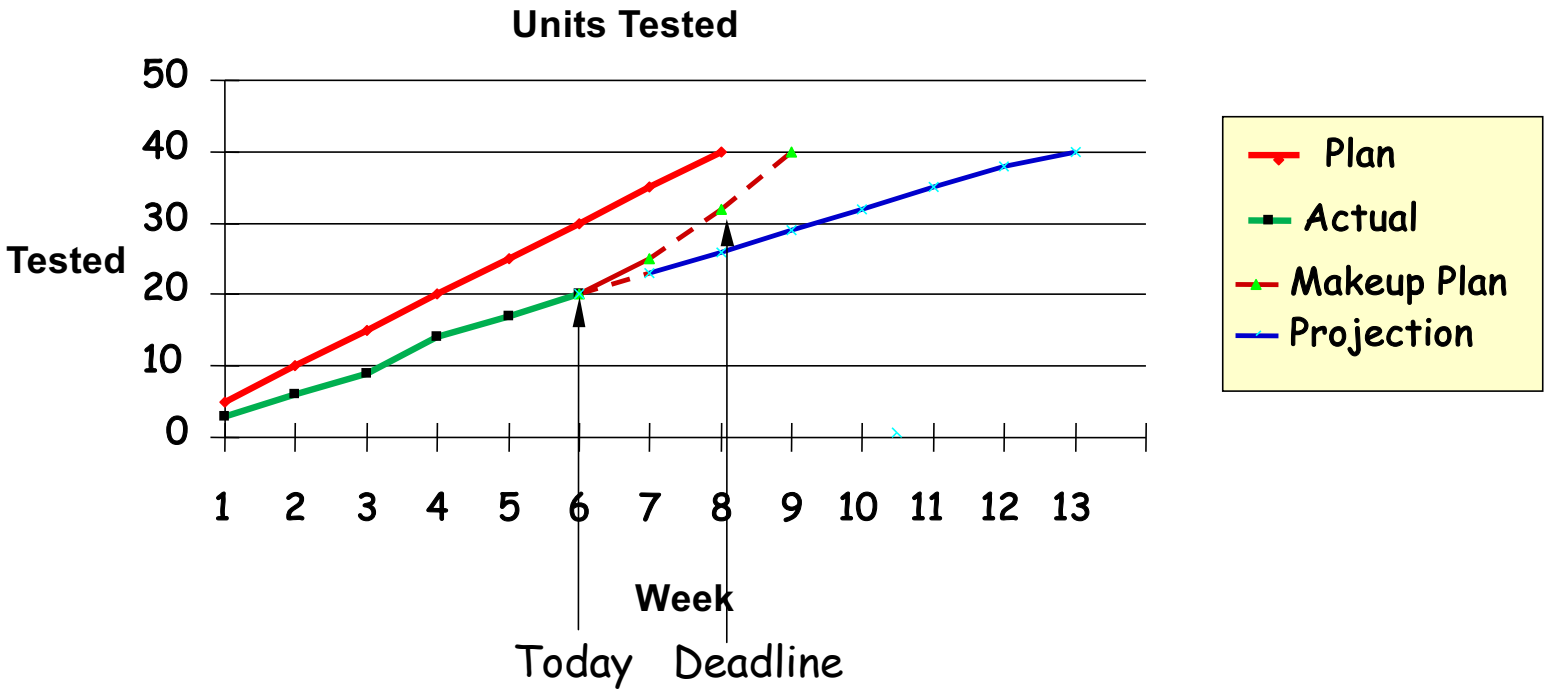
Measure Processes, Not People

- **It is important to measure *things that affect productivity of people*, such as:**
  - Training – is it accomplishing what we want it to accomplish?
  - Turnover (planned and unplanned)
  - Resource utilization
  - Resource availability
  - Staffing level
  - Effectiveness and usability of processes and procedures
- ***People will* usually *cooperate* if you try to *make their jobs more efficient***
  - But they will resist if you find ways to blame them

Resource Measures

Testing Progress

Measuring testing progress helps us predict schedule.



The Metric Should Not Be the Goal!

Suppose your goals are

- **Good** (effective) testing
- **Efficient** testing

Good uses **for a** ***testing progress*** **metric:**

- Identify problems in testing and use the information to ***find and fix the underlying problems***
  - Perhaps the test code isn't very good
  - Or perhaps there are equipment problems
  - Or perhaps you incorrectly estimated the difficulty of testing your software product

Potentially ***bad uses*** **for a testing progress metric:**

- Criticizing people for not meeting the deadline
- Rewards for the most tests done per week

Using Testing Progress Metrics Improperly

Wrong Performance Goals

- **Real goal: *good, efficient testing***
- **Performance goal for testing team:**
  - **more tests complete per week**
- **Potential consequences:**
  - Team **makes tests simpler (and less effective)** so they can get more tests done per week
  - Team focuses on **testing quickly instead of testing thoroughly** and effectively
  - Team creates **smaller test cases** rather than what makes sense

Time is wasted improving the numbers instead of improving the testing

Using Testing Progress Metrics Improperly
Measuring Individual Performance

If you measure *testing progress for individuals* you might encourage people to …

- **Run the easiest and least effective tests** in order to get more tests complete per week
- **Cut corners** (skip parts of the testing process) when doing testing in order to get more tests done each week
- Use tools in ways that **mask inefficiency**
  - Making it appear they have done more than they actually have
- Test **only the least complex parts** of the software

And you might reward the wrong people – the ones who run the most tests, not the ones who do the most effective testing.

Using Testing Progress Metrics Properly

- **Use the Test Progress metric as an *indicator of your true situation***
  - If there's a problem, fix the problem
  - ***don't***
    - ***pretend it isn't there***
    - ***encourage people to cover it up***
    - ***blame people***

- **Focus on the *test processes and procedures***
  - Are your tests being developed properly?
  - Are your tests being run properly?
  - Are you properly estimating the time required for testing?
- ***Enlist the aid of the software team* to analyze the problems and make improvements**

Seeding and Tagging
A simple and effective way to assess Testing Progress

Seeding and Tagging

Purpose: **To help you estimate how many undetected errors (defects) are in your code**

When to do this: **During test planning and during the testing process**

Suppose**: You have been testing your code and have discovered D_1 errors (defects).**

Question**: How many errors are left?**

Technique**: Seeding and Tagging**

Concept**: Introduce extra errors and see how many of them your test process has found.**

Overview

1. **Inject extra errors**

   before testing starts

2. **See how many of those errors you find during the normal testing process**

Seeding and Tagging Details

- **Introduce a given number of extra errors into the software -- say *E* of them**
- **Run standard tests, detecting D_2 of them**
- **Compute D_2/E = % of errors detected**
- **Suppose D_1 = number of genuine errors already detected**
- **Then you assume the total number of errors in the software is**

$$D_1 * E / D_2$$

Example of Seeding and Tagging

- **200 defects found so far**
- **You have injected 20 extra defects**
- **You have found 12 of these extra defects**
- **Therefore, assume total defects =**

200 * 20 / 12 = 4000 / 12 = 333 total defects

=> 333 - 200 = 133 defects remaining

By performing this analysis from time to time, you can estimate your defect density and your testing progress over time.

UT Dallas

Software Quality and Software Testing

Part 1 – The Big Picture (How Quality Relates to Testing)

Part 2 – Measuring Software Quality

Part 3 – Software Reliability

Part 4 - Defect Containment

Part 5 – Measuring Software Complexity

Quality Attributes are Seldom Directly Measurable

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

How can these be measured?

We need to find suitable *ways to measure* these attributes.

Some Attributes Are Measurable

Examples

- Water boils at 100° Centigrade
- My new application will complete at least 10 searches per minute
- Code written in C takes less memory space than code written in Python

The above statements may or may not be true, but they can all be tested because they are all measurable.

Some Attributes are Not Measurable

Examples:

- Joe's code is better than Jan's code
- Lisp is a superior programming language to C#
- Object oriented design produces code that is more maintainable

The above cannot be measured unless we define what we mean by:

- Better than
- Superior
- Maintainable

} **In a measurable way!**

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

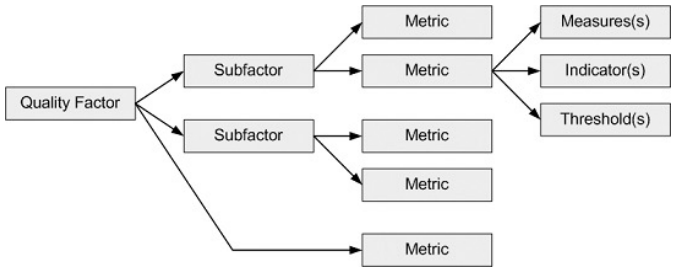
A surrogate may or may not accurately reflect the desired attribute

Examples:

- **Defects** are a common surrogate for quality
- But lack of defects may or may not reflect **quality**.
  - Lack of 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

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



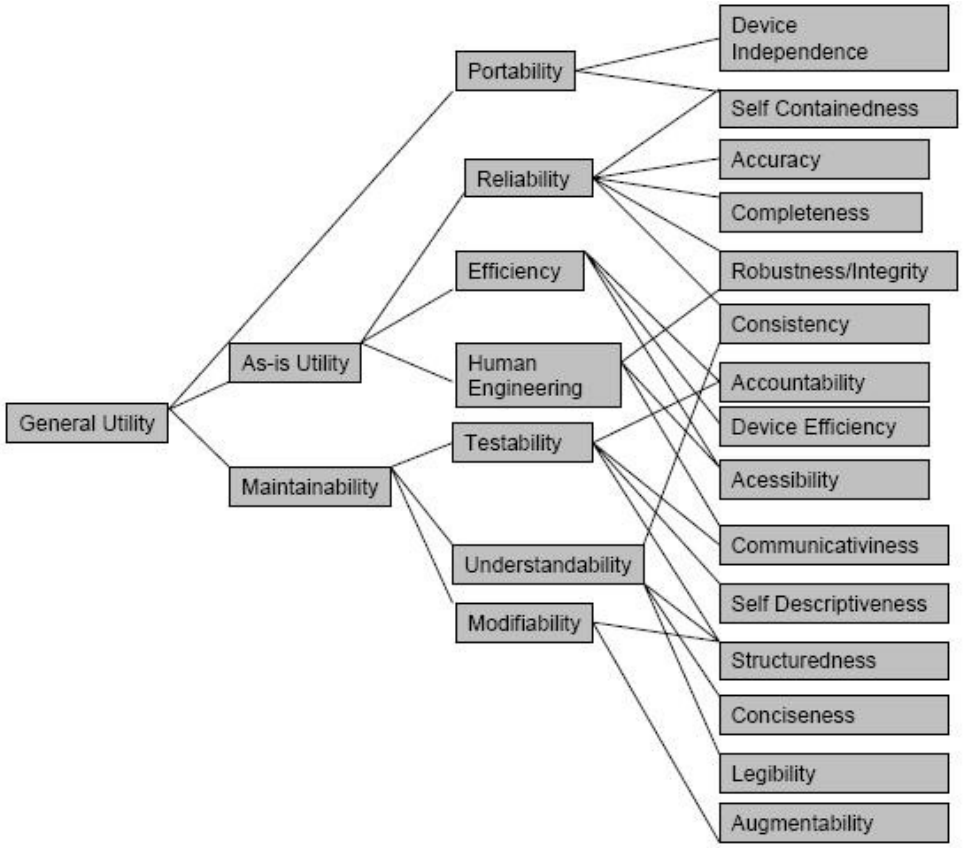
- **Standardized Approaches**
  - These enable comparisons of software from different organizations
  - But may not fit the desired quality characteristics of some software

Quality Characteristic	Boehm	McCall	FURPS	ISO 9126	Dromey
Testability	X	X		X	
Correctness		X			
Efficiency	X	X	X	X	X
Understandability	X		X	X	X
Reliability	X	X	X	X	X
Flexibility		X	X		
Functionality			X	X	X
Human Engineering	X				
Integrity		X		X	
Interoperability		X		X	
Process Maturity					X
Maintainability	X	X	X	X	X
Changeability	X				
Portability	X	X		X	X
Reusability		X			X

Bvicam.ac.in

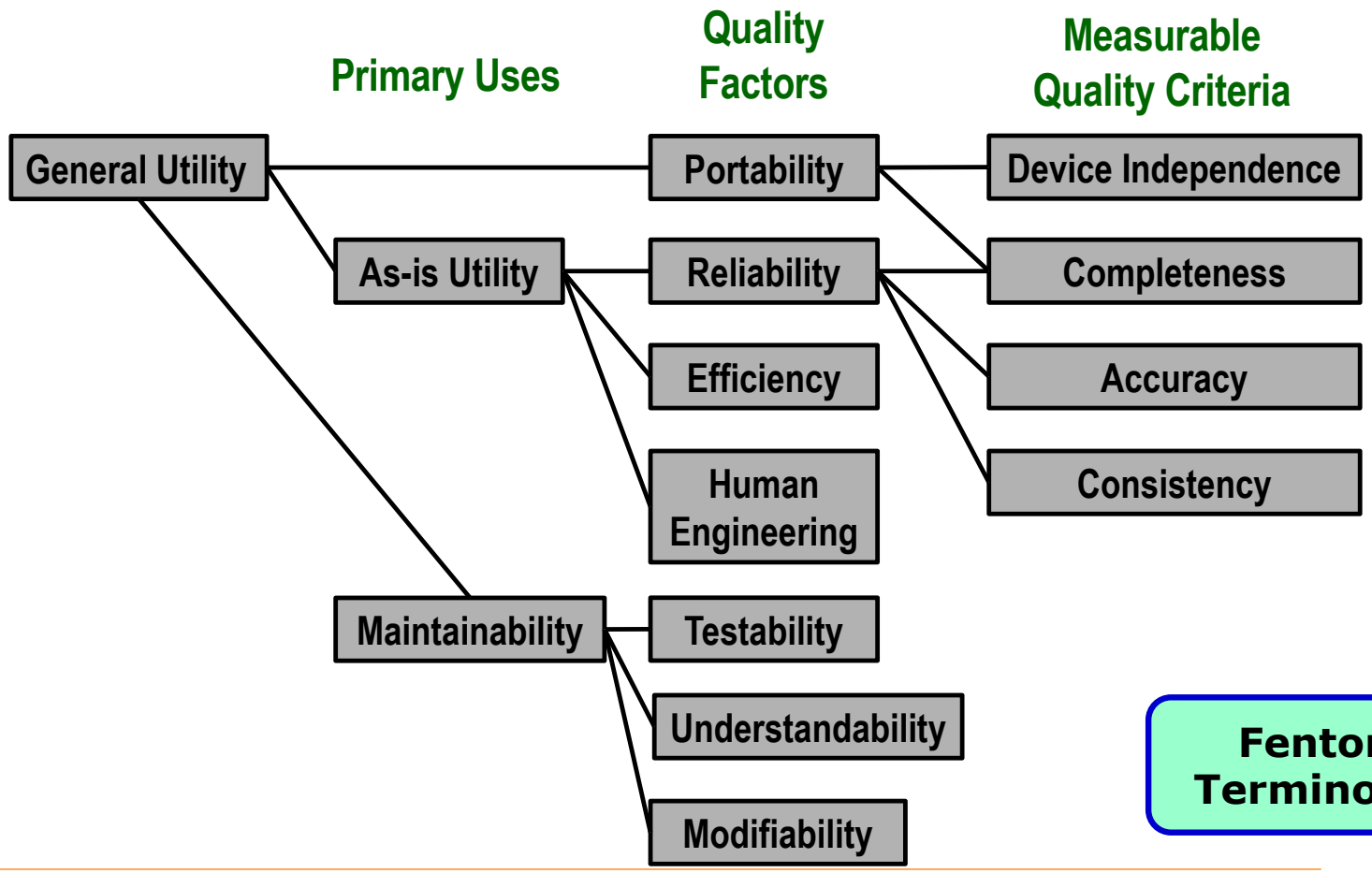
There is little consensus on how to measure quality attributes, so most organizations define them in ways that fit their specific customer needs.

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.

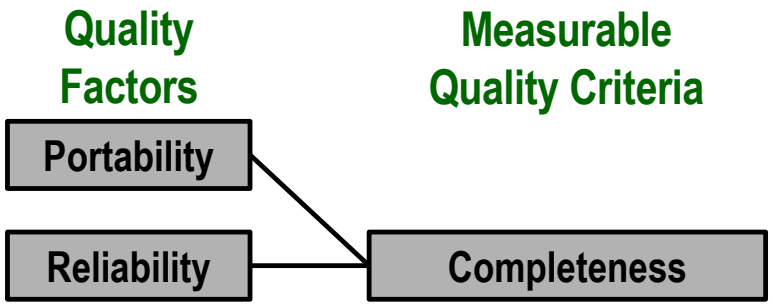
A Closer Look at the Boehm Model



Fenton's Terminology

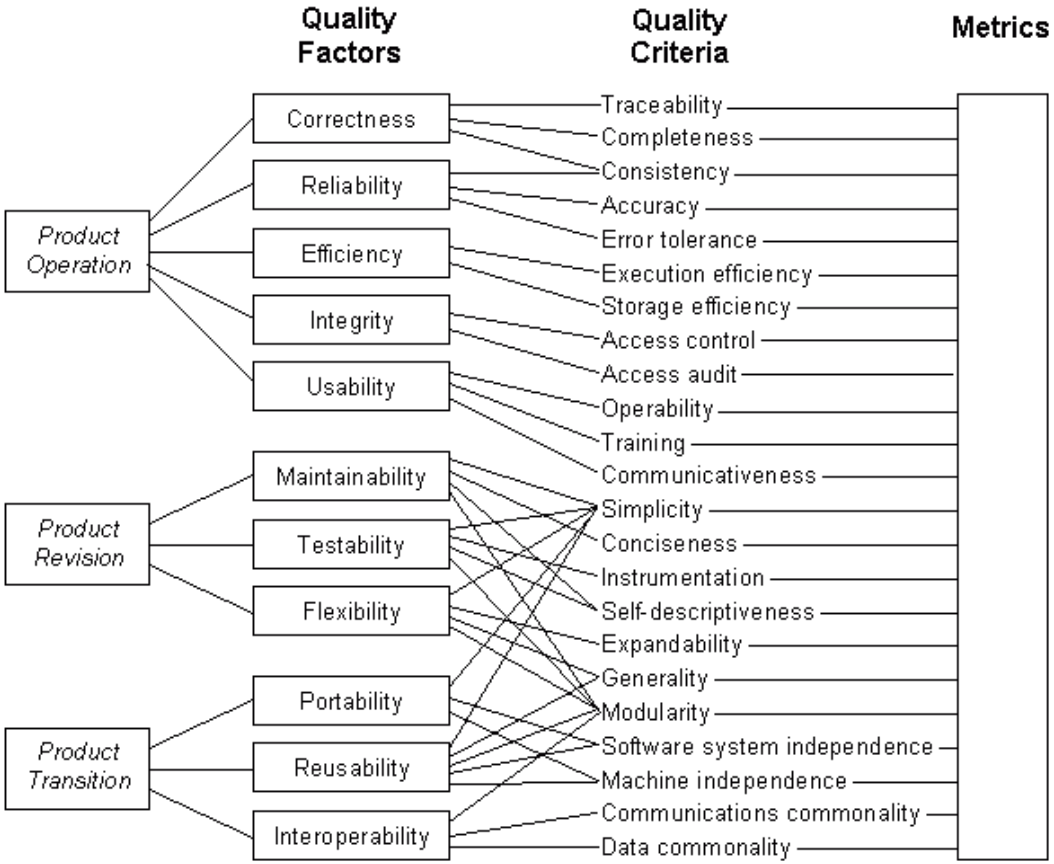
Comments on Boehm's Model

- **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



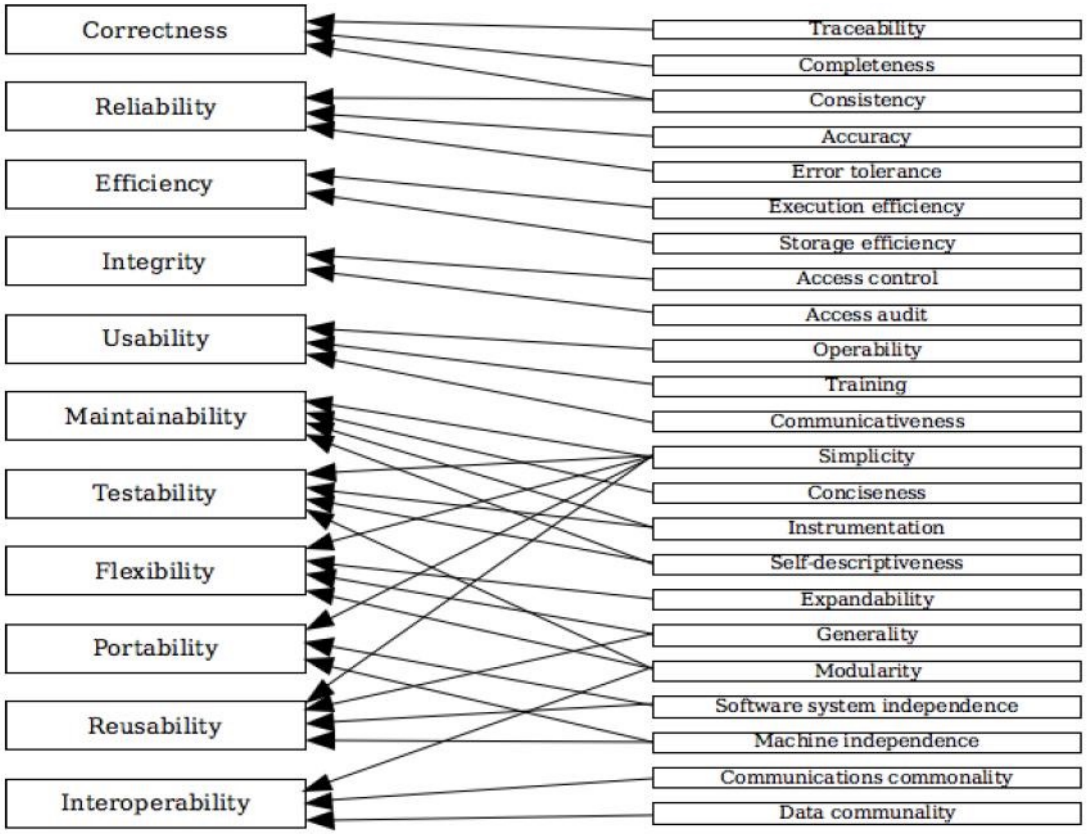
Decomposition Approaches

McCall Software Quality Model



As you can see, it's possible to establish a lot of criteria related to quality

McCall Model – Quality Factors and (Measurable) Criteria



As with the Boehm model, some criteria relate to multiple quality factors

Do I Really Need to Measure So 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***

Measures of Software Quality

Based on

Defects or Faults or Failures

Quality = Lack of Defects (or Lack of Faults or Lack of Failures)[1]

The advantage of this approach is that it is often *easier to test for* defects or failures and *easier to measure* them than many other measures of quality

- However this approach ***may not capture what quality means to the end user***
  - Ease of use
  - Speed
  - …
- And it ***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[1]

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

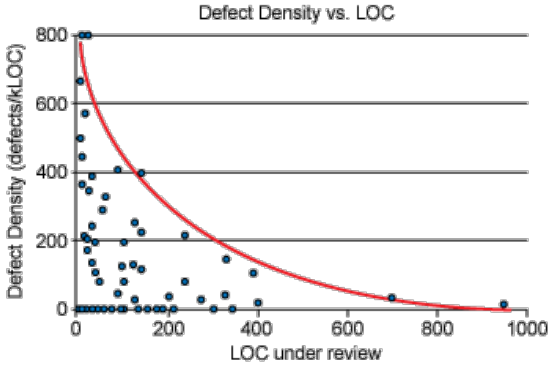
Variations:

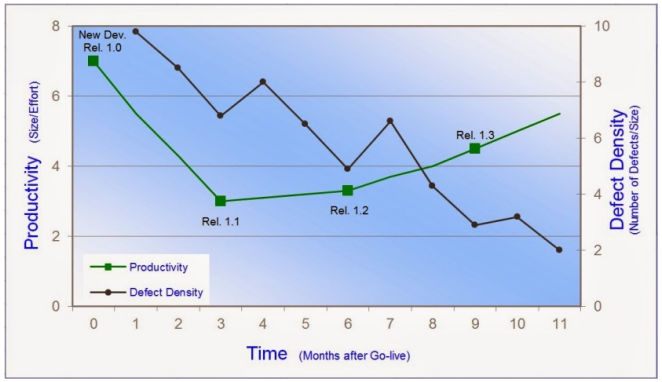
- *Failure Density* (instead of defects)
- *Number of Defects* (this can be defined in different ways)
  - *Known Defects*
  - *Total Defects* (*Known Defects* + *Latent Defects*[2])
- *Size of Software Product* (can be defined in different ways)
  - *It depends on the definition of size*

[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 3)

- **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[1]

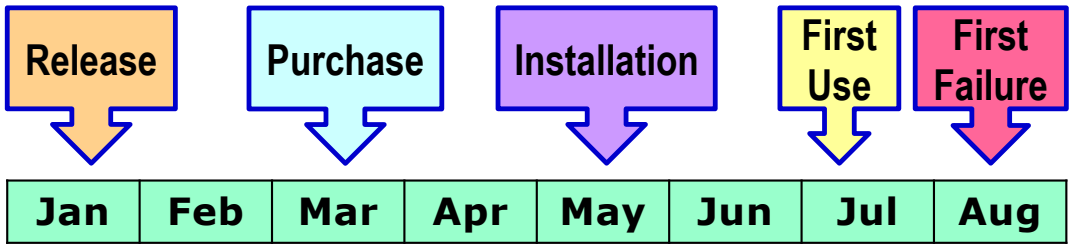
- **Approximately *one out of three defects* will only cause a user failure *once in 500 years*.**
- **A *very small portion* of defects (<2%) *cause the most important user failures***

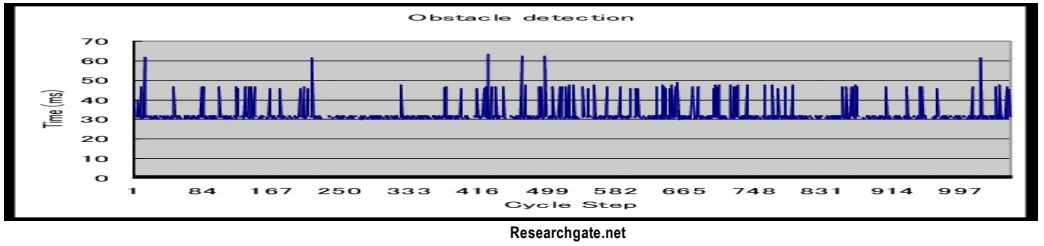
Number of defects <u>may</u> <u>not</u> be strongly correlated to the frequency or severity of end user failures.

[1] See Adams in reference list.

Defect Density Drawbacks
(2 of 3)

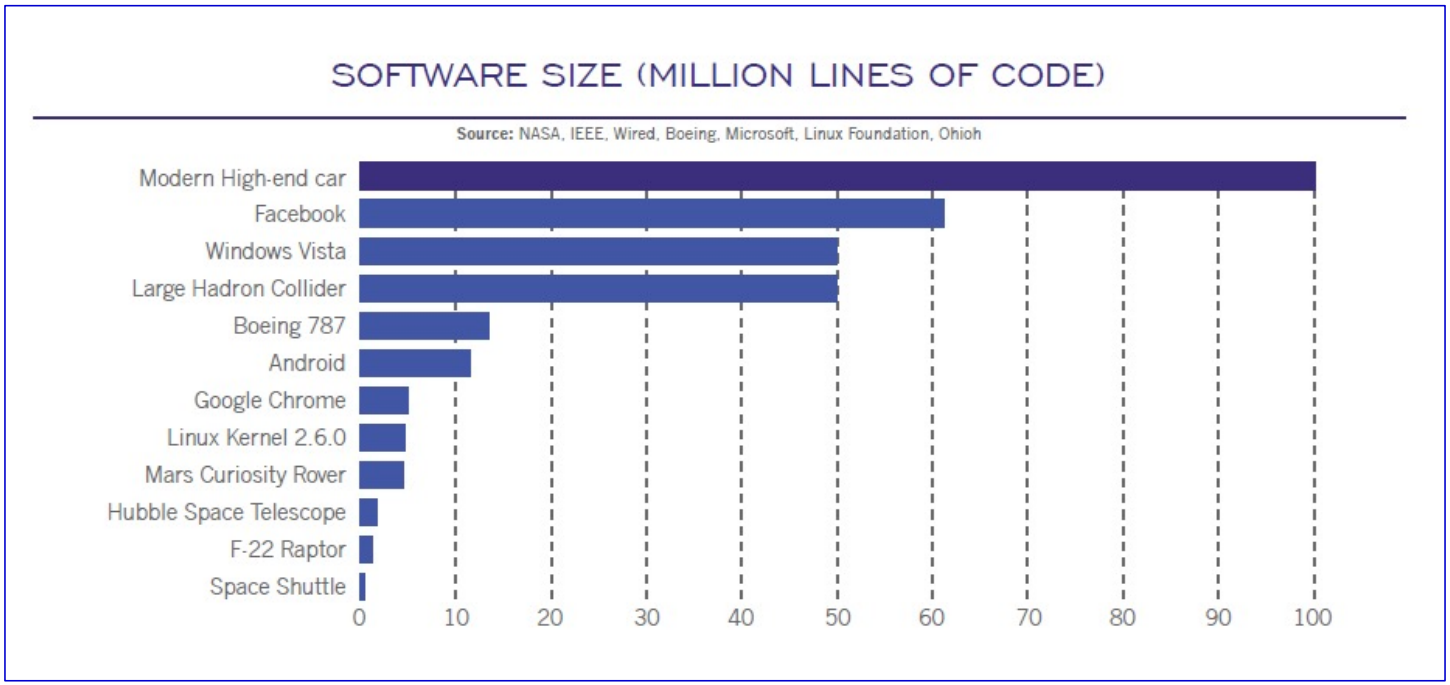
- ***Different measures of the time scale***
  - Amount of ***time since release of product***
  - Amount of ***time the product is actually used***
  - ***Processing time*** actually used by the product





Defect Density Drawbacks
(3 of 3)

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

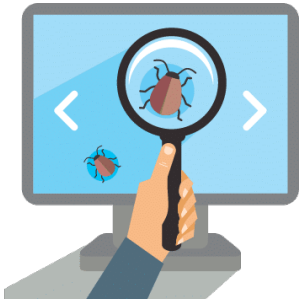


What is Defect Density Telling Us?

- **The quality of our software product?**

Or

- **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
Hard to Test For & Hard to Measure

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[1]

- **Effectiveness**
  - Can users complete the tasks correctly?
  - Example: $\text{Effectiveness} = \frac{Quantity * Quality}{100}$
- **Efficiency**
  - Time required to complete the tasks
  - Example: $\text{Efficiency} = \frac{Effectiveness}{Task\ Time}$
- **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 and help files

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

Use of these to predict usability is not recommended.

UT Dallas

Software Quality and Software Testing

Part 1 – The Big Picture (How Quality Relates to Testing)
Part 2 – Measuring Software Quality
Part 3 – Software Reliability
Part 4 - Defect Containment
Part 5 – Measuring Software Complexity

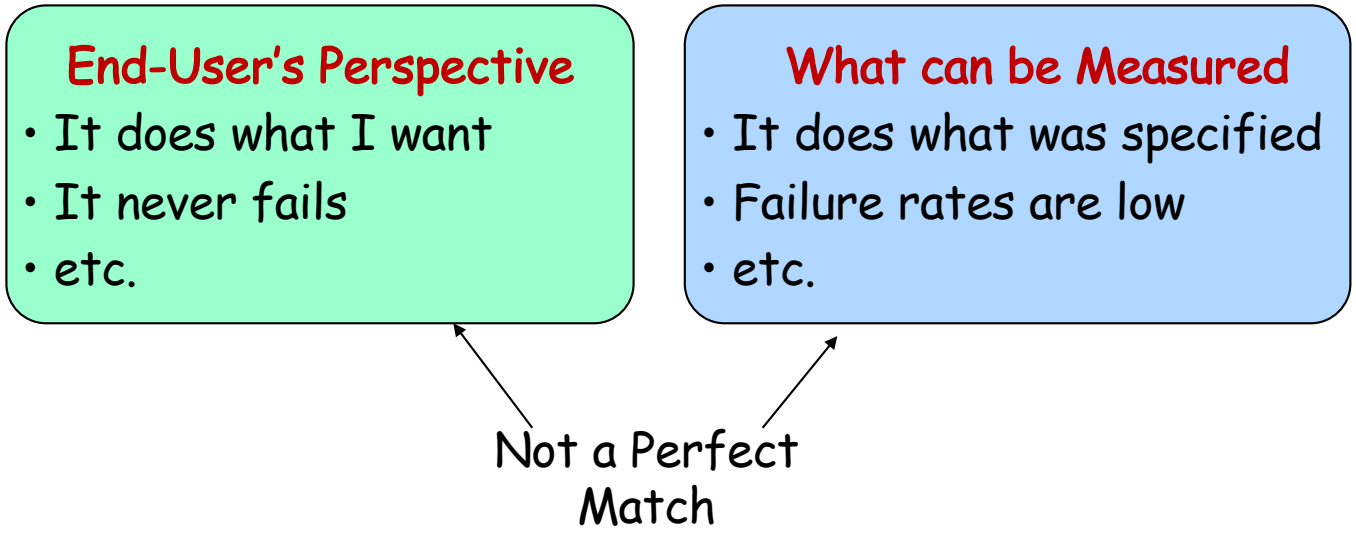
Contents

- **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?***



Reliability is Not Correctness

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

Whereas

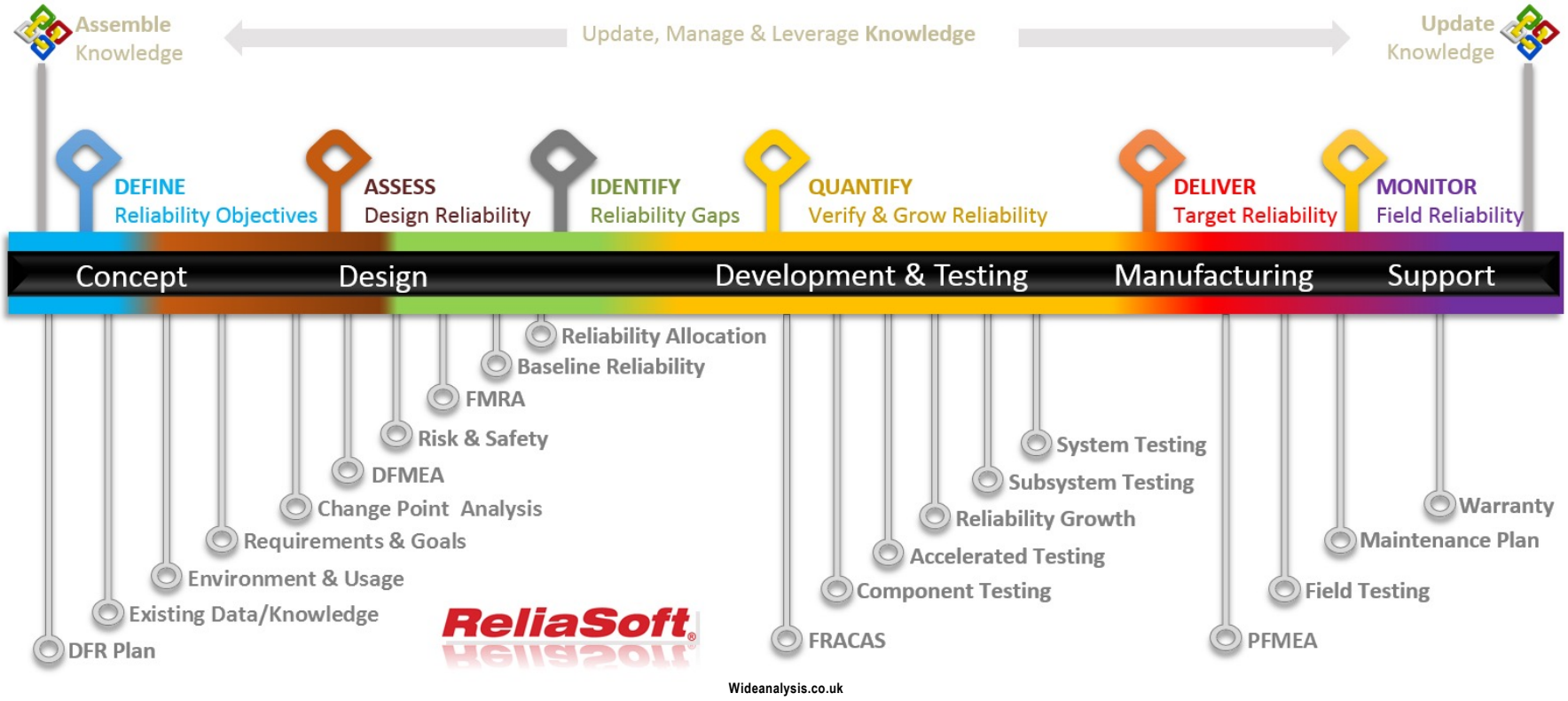
- **Correctness is a binary, "yes or no" condition**
- **Software is almost never perfectly correct**
  - But it can be highly reliable

Hardware Reliability Theory
Focuses on Materials and Production

- **Assumption: failure usually results from *physical effects***
  - breakage, wearout, fatigue, corrosion, overheating, shock, …
- **Or incorrect manufacturing processes**

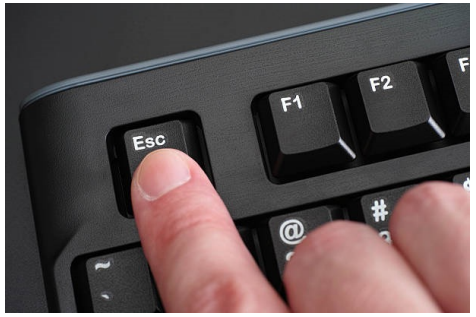
- **The theory of hardware reliability is founded on the assumption that these are *random* events**

But Product Design and Development Can Also be Factors in Quality & Reliability



Poor Design & Development Practices Can Lead to Hardware Failure

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



Freeimages.com

- **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.

Ways to Improve Software Reliability

www.softwarereliability.com

Improving Software Reliability
Option 1

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**

Improving Software Reliability
Option 2

Develop software to be *free of defects*

- Prevention activities
- Detection activities

} Cost of Quality Analysis

- **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
- …

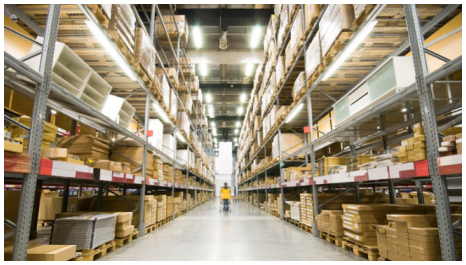
Improving Software Reliability
Option 3

Study *how software fails* and focus on <u>understanding failures</u>

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**
  - This is known as ***Reliability growth***

Sometimes Failures Are Due to Complex Causes



The Dissapearing 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

Reliability-safety-software.com

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 (λ) - the rate at which failures occur

- In some cases, λ 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:

  λ(t) = <some equation involving t>

In both cases, λ 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) (□) - <u>the time when the first failure is expected to occur</u> (on average)

- In some cases, □ is a constant, such as "520 hours"
- In other cases, it is not a constant
  - It is often expressed as a function of time:

    □(t) = <some equation involving t>

In both cases, □ represents the *most probable value*, based on what is known about the system and its operation.

The Relationship Between Failure Rate and MTTF

$\theta = 1 / \lambda$ } If it is a constant

Or

$\theta(t) = 1 / \lambda(t)$ } If it is a function

What is Reliability?

Reliability is the *probability* that software will continue to function correctly (without failure)

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) = probability of *operation without failures* 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 ***functioning correctly*** at time 0
2. Failures occur ***randomly***
3. Failures occur at a ***constant rate***, that depends on the specific hardware. This rate is usually represented by the symbol λ

Discussion of These Assumptions As They Apply to Software

1. **The system is functioning correctly at time t = 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, λ**
   - This assumption ***may not make sense for hardware*** because, as hardware ages, failures are more common
   - This assumption makes ***little sense for software*** because of reliability growth (see later slides)
   - So for both hardware and software, we often represent failure rate by a continuous function **f(t)** rather than as a constant λ.

Reliability is Exponential if All Three Assumptions are True

R(t) = **probability of *operation without failures* in time interval 0-t**

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

λ 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.

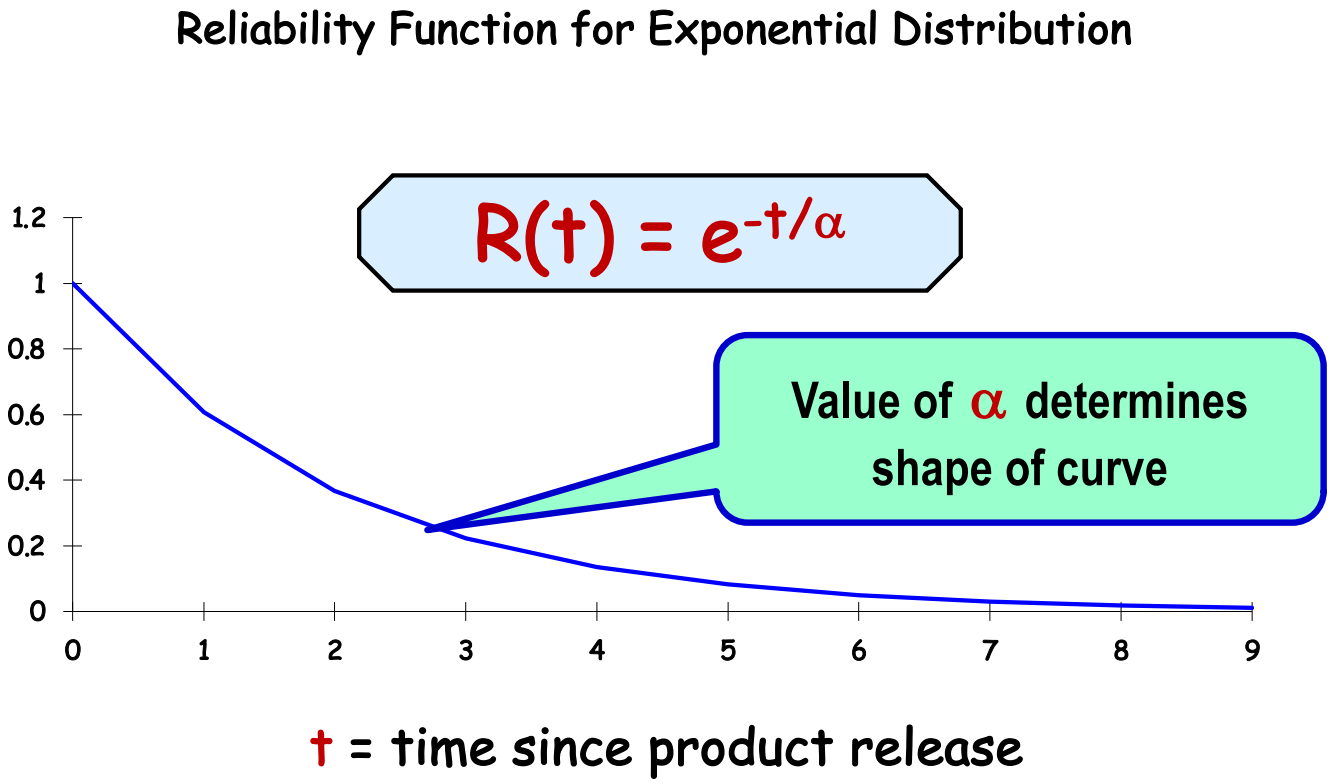
Why Is Exponential Failure Rate Useful for Understanding Software Reliability?

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 .**

Graph of Exponential Reliability Function



α Measures Reliability as a Constant

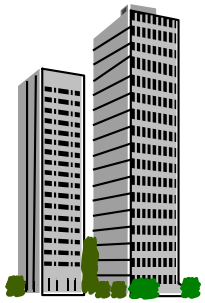
- **α is the *mean time to failure* (MTTF).**
  - **Actually, the mean time to the first failure.**
- **For large values of α, the probability of operation without failure remains high for a longer period of time**
- **For small values of α, the probability of operation without failure deteriorates quickly**

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

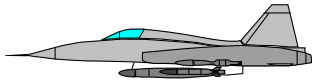
Additional Notes about Reliability

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:

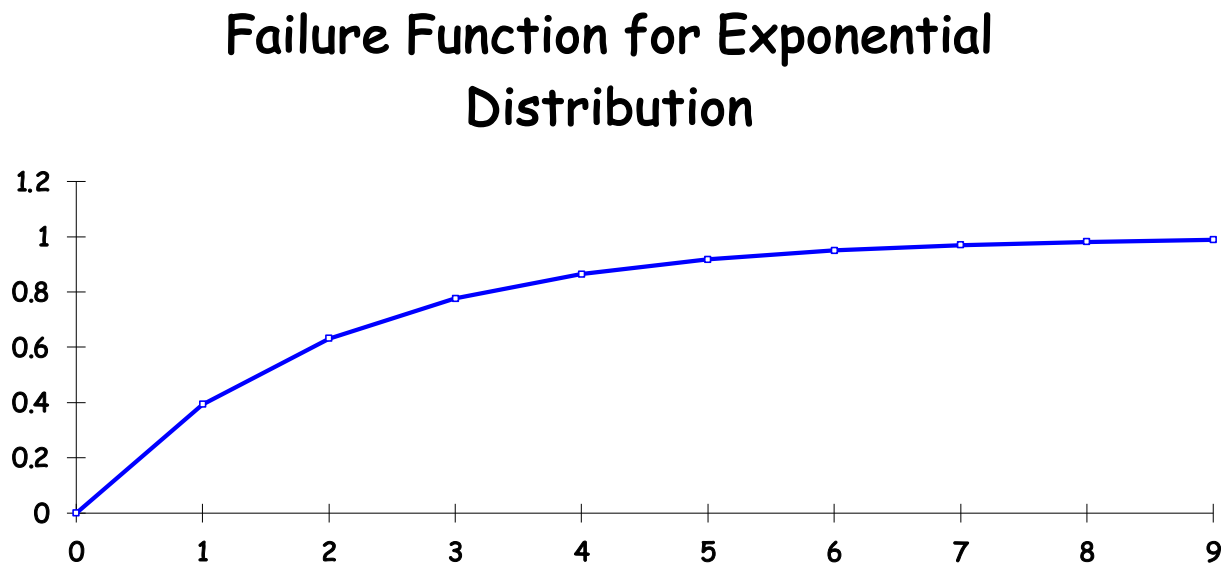
F(t) = 1 - R(t) = probability of failure in time interval **0-t**

- **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



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

Probability Density Function or Probability Distribution Function

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

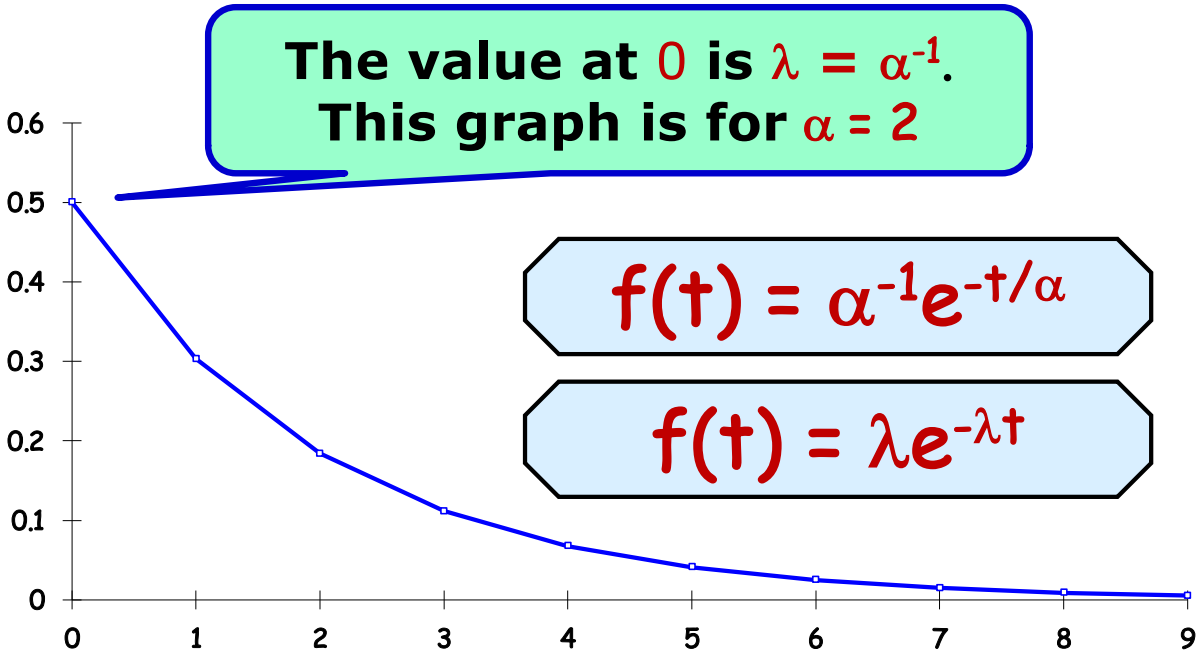
*"(approximately) what is the likelihood that a failure will occur at time **t**"*

$$f(t) = dF(t)/dt$$

For the exponential distribution, the formula is:

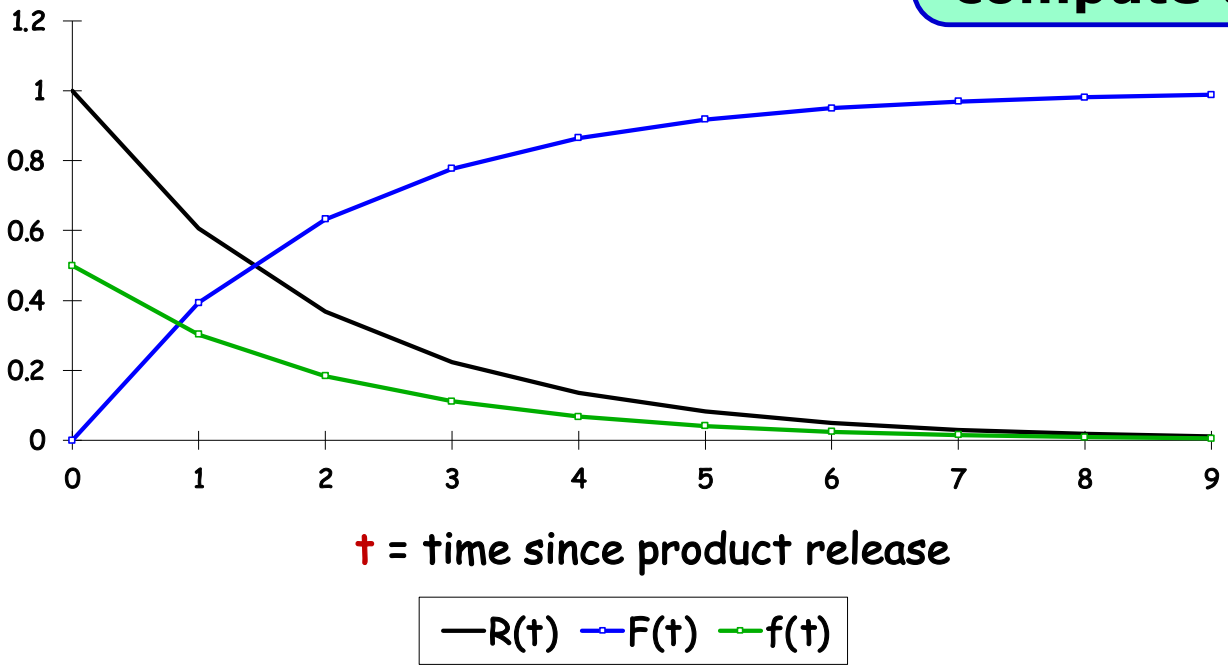
$$f(t) = \alpha^{-1} e^{-t/\alpha} \quad \textbf{or} \quad f(t) = \lambda e^{-\lambda t}$$

Graph of Exponential Density Function



All three functions ($\alpha = 2$)

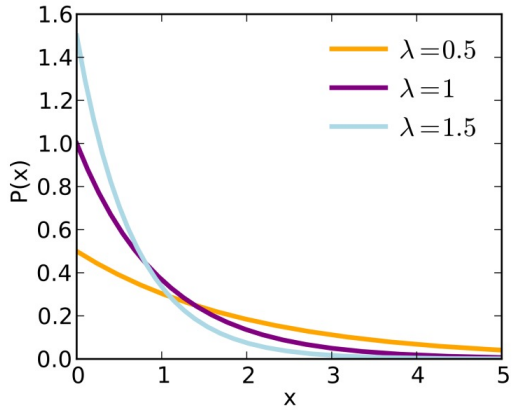
Note that given any one of these we can compute the others.



Conditional Failure Rate (λ)
(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.

$$\lambda(t) = f(t)/R(t) = -dR(t)/dt$$



How to Determine $\lambda(t)$

$$\lambda(t) = 1/\alpha(t)$$

- **I.e., the higher the reliability, the lower the failure rate**
- **If λ is a constant, then α is a constant and**

$$\lambda = 1/\alpha$$

Failure Rate vs Number of Defects

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.

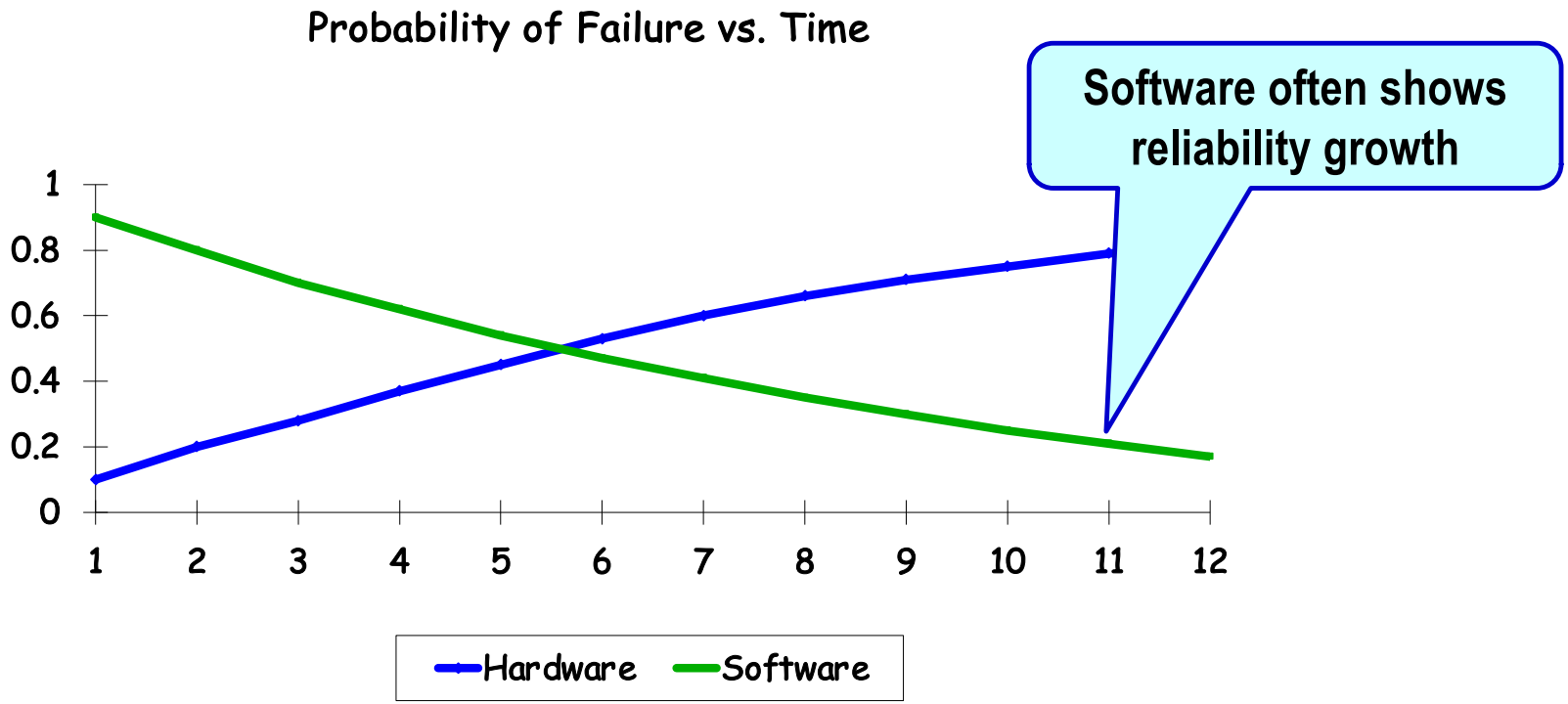
Problems with the Assumptions for Classic Hardware Definitions (when applied to software)

- **The classic assumption for hardware devices is that *defects are random* with respect to the structure of the product**
  - **But this is usually not true for software**
    - **Some parts of software are harder to write than others and thus more likely to have defects**
- **The classic hardware assumption assumes *testing is uniform* with respect to the product**
  - **But with software, some parts are likely to be more effectively tested than others**

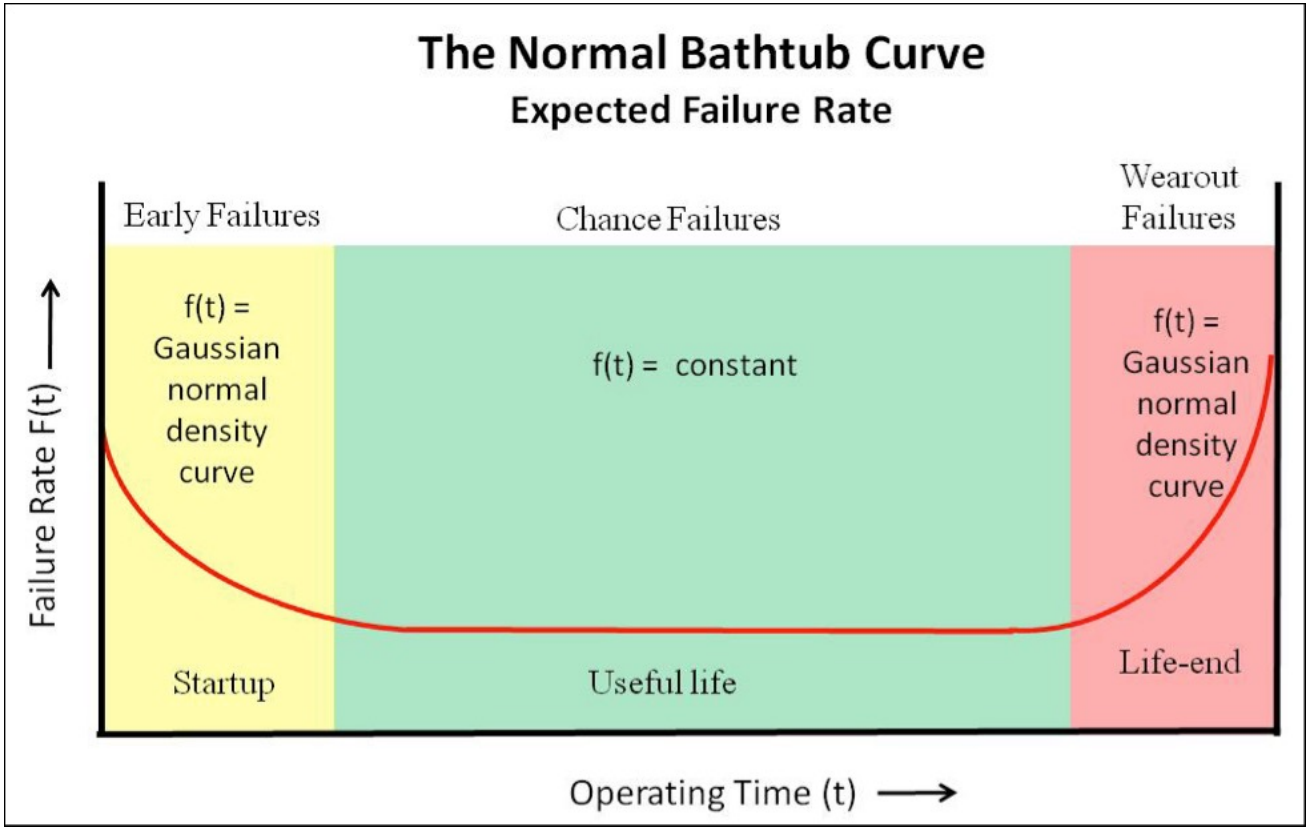
More Problems with Classic Hardware Assumptions

- **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

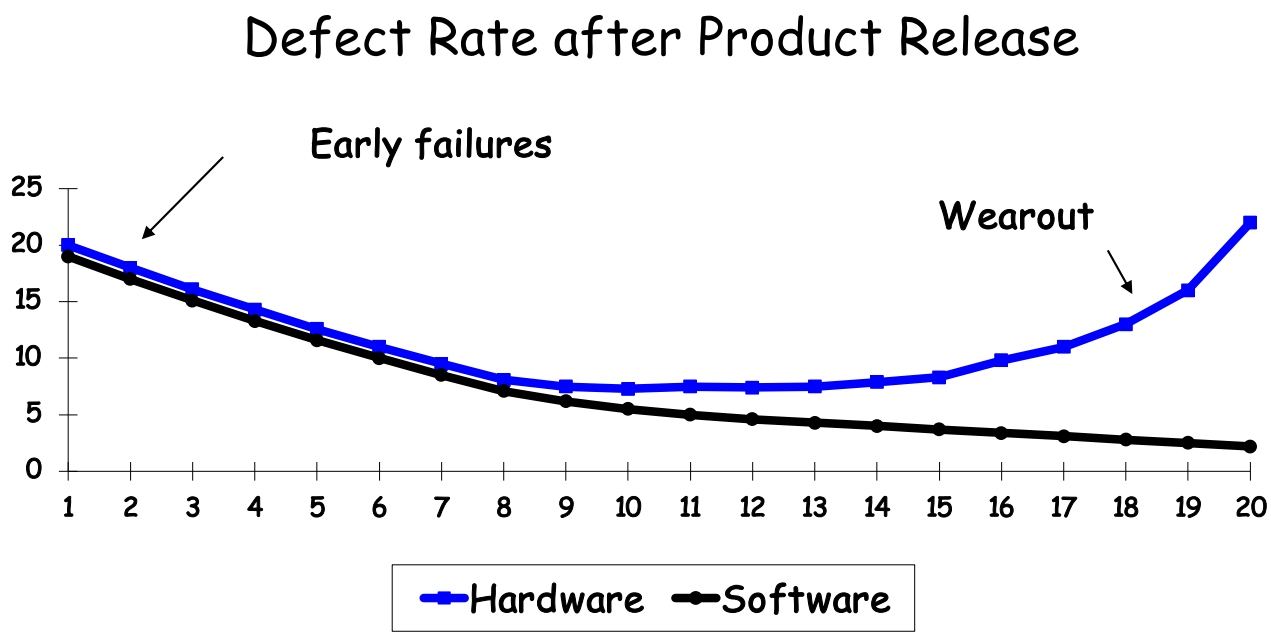


The Bathtub Curve is Often A Better Description for Hardware



Software Tends to Get More Reliable Over Time Because Parts Don't Wear Out

[In the absence of major modifications]



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

Natural Units

Each of these produces different results and may fit different models

The Nature of The Application

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

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.**

UT Dallas

Software Quality and Software Testing

Part 1 – The Big Picture (How Quality Relates to Testing)
Part 2 – Measuring Software Quality
Part 3 – Software Reliability
Part 4 - Defect Containment
Part 5 – Measuring Software Complexity

Defect Containment (Phase Containment)

This requires that you collect additional information about each defect you discover during an inspection or as a result of a test:

- In what ***phase of development*** was the defect ***created***?
- In what phase was it ***detected***?

Defects Phase Containment / Leakage
(High Severity Defects - Priority 1, 2 &3)

Life Cycle Phase Originated			Life Cycle Phase Discovered							
			L	R	D	C	I	T	A	In the red
	Legacy	6.7%	75%	0%	0%	0%	25%	0%	0%	25%
	Requirements	15.0%		100.0%	0%	0%	0%	0%	0%	0.0%
	Design	15.0%			89.9%	11%	0%	0%	0%	0.0%
	Code and Unit Test	63.3%				94.7%	5.3%	0%	0%	0.0%
	Integration Test	0.0%					0%	0%	0%	0.0%
	Test									
	After Test									

Insights.sei.cmu.edu

Note on Defect Containment

- There are several variations on this method
- All use the same basic data (base measures) but they use the data in different ways

In this lecture we will illustrate one of the variations on this method.
You may find others at www.sei.cmu.edu

Example of Defect Containment

- **Suppose you *detect* a lot of defects during *system test***
- **And suppose you discover that most of them *occurred* due to *bad design procedures***
- **Then you know that the best way to fix the problem is to *improve your design procedures***

In-Phase Defects

In-phase defects **are those that are** ***corrected in the same development phase*** **where they were introduced**

- Example: a coding error that is caught and corrected while you are writing the code, before going to system test

- **Measuring in-phase defects tells you which parts of your process generate large numbers of defects**

In-phase defects are generally the least costly to correct.

Out-of-Phase (Leaking) Defects

Out-of-phase defects **are those that are detected (and corrected)** ***after*** ***they leave*** **the phase where they were introduced**

- Example: a design error caught during unit test

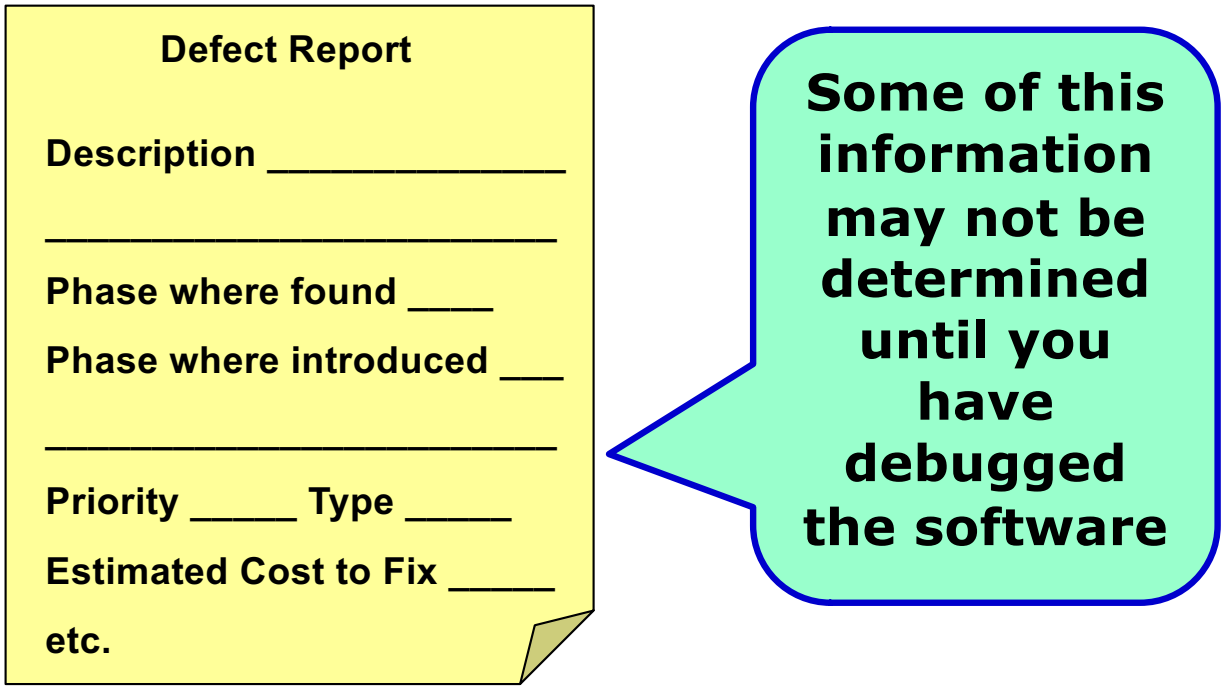
- **Measuring out-of-phase defects indicates how often you allow defects to "leak" from the phase where they originate**
  - **this is a predictor of post-release failures**
  - and also a good help in ***root cause analysis***

Finding the Ultimate Cause of a Defect

Out-of-phase defects are generally the most costly to correct.

Defect Containment Analysis
Step 1 – Collect the Data

Track Each Defect and Record Phase of Origin



Defect Containment Analysis
Step 2 – Record and Display the Data

Defect Containment Matrix – Sequential Process

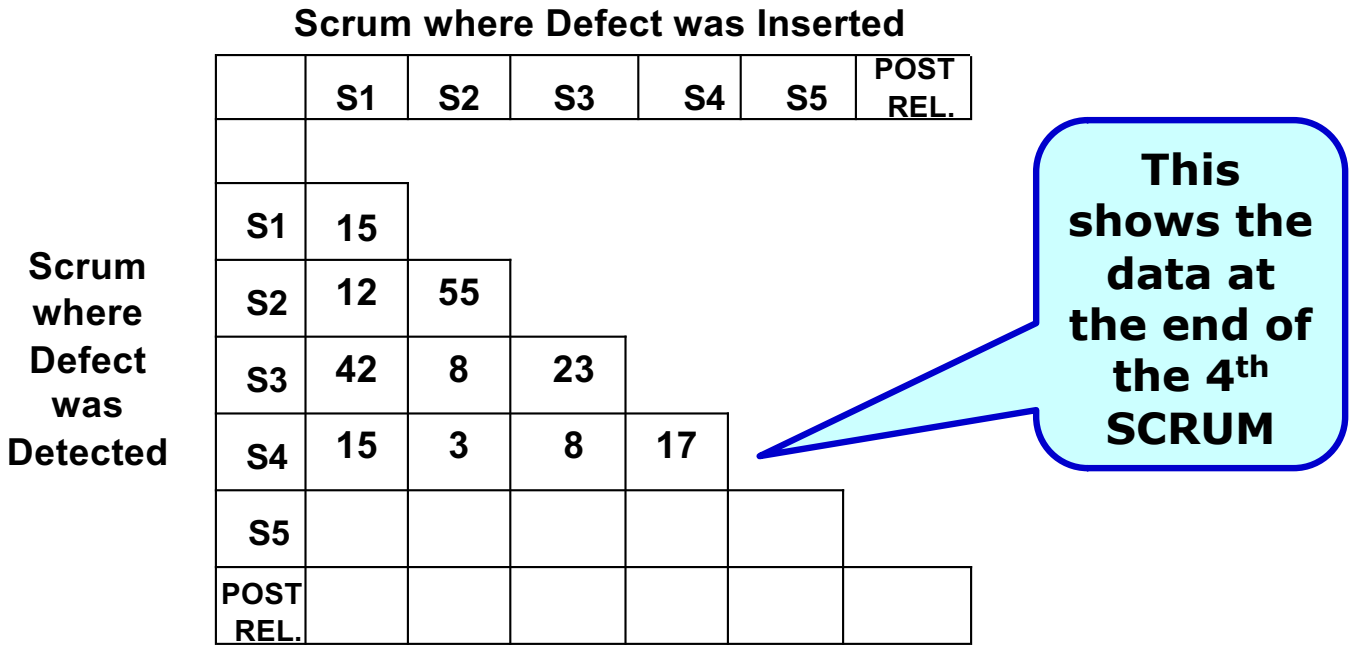
Phase where Defect was Inserted

Phase where Defect was Detected	RA	PD	DD	C&T	I&T	POST REL.
RA	15					
PD	12	55				
DD	42	8	23			
C&T	15	3	8	17		
I&T						
POST REL.						

This shows the data at the end of the C&T phase

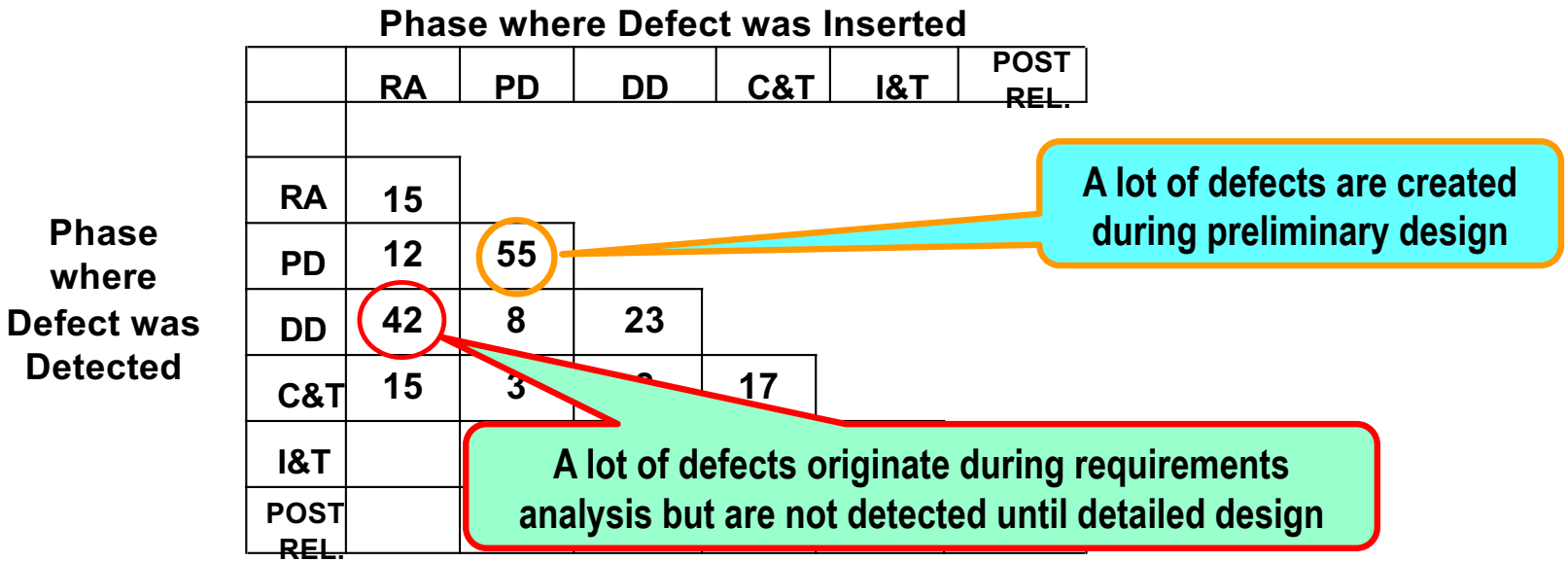
Defect Containment Analysis
Step 2 – Record and Display the Data

Defect Containment Matrix – SCRUM Process



Defect Containment Analysis Step 3 - Using the Data

If you see many out-of-phase defects in a specific cell, you can narrow down the source of defects



Defect Containment Analysis Step 4 - Using the Data to Provide Additional Insight

Over time, you can correlate

- **the number of defects in the matrix**
- **to the number of failures found by the customer**

➢ ***You can use this to predict and ultimately to manage the number of failures***

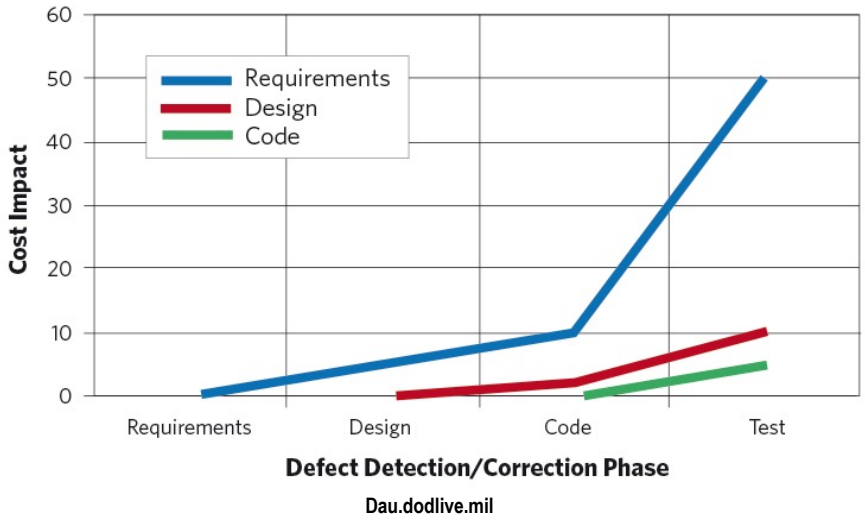
A method for doing this will be shown briefly in today's lecture

Observations on This Method

1. **Definition of a defect must be adhered to *in a consistent way* across the project and, preferably, across all projects in an organization**
   - Some projects may resist defining defects the same way as other projects.
2. **As shown, there is no distinction by type or severity of defect**
   - But this distinction can also be made if the original data are good enough)

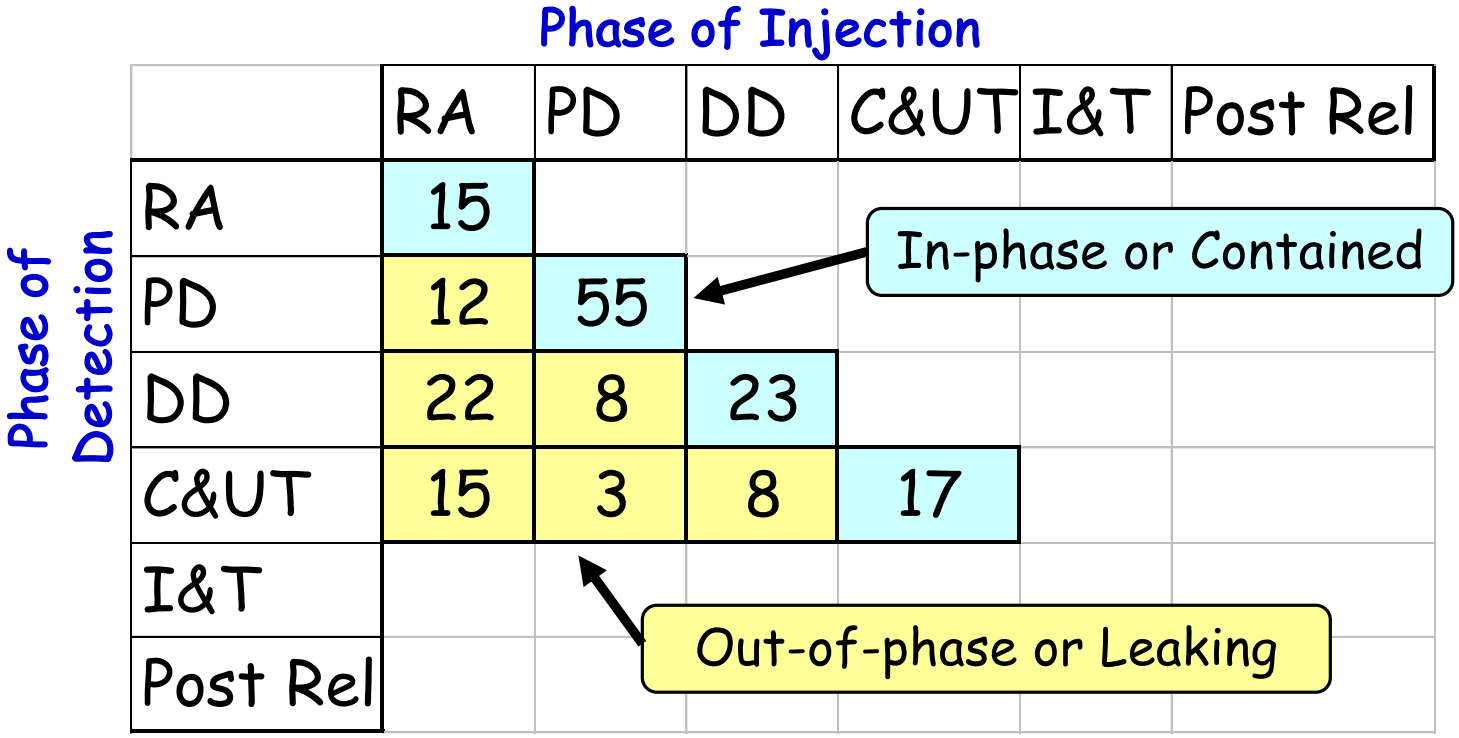
A Key Lesson Learned from Measuring Defect Containment

If you detect and correct defects early, it *greatly reduces cost and reduces post-release failures* (i.e., those seen by the customer)



- **But this requires very good understanding of requirements and of customer "care-abouts"**

Contained and Leaking Defects



Large Numbers Indicate Software Development Process Problems

- **Large numbers in any column indicate that your development process is generating many defects in that process phase**
- **A large number in a "leaking" cell means you are also paying a lot of money for rework**

This tells you where to focus process improvement efforts

A Typical Defect Containment Chart

	Phase Originated							
Phase Detected	RA	PD	DD	CUT	I&T	SYS INT	POST REL	**tot**
RA	730							73
PD	158	481						63
DD	19	2	501					52
CUT	15	0	12	63				9(
I&T	25	4	35	321	9			39
SYS INT	4	0	7	19	4	2		3(
POST REL	48	2	0	36	0	0	67	15
total	999	489	555	439	13	2	67	25(

Least Costly Defects are on the Diagonal

These defects are "Contained" within the step where they were caused

Escaping Defects are Those Not Detected until After Release

	Phase Originated							
Phase Detected	RA	PD	DD	CUT	I&T	SYS INT	POST REL	**total**
RA	730							730
PD	158	481						639
DD	19	2	501					522
CUT	15	0	12	63				90
I&T	25	4	35	321	9			394
SYS INT	4	0	7	19	4	2		36
POST REL	**48**	**2**	**0**	**36**	**0**	**0**	**67**	153
total	999	489	555	439	13	2	67	2564

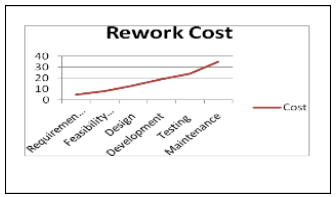
Escaping Defects Cost the Most of All

Other Uses of Defect Containment Data

There are many uses of defect containment

- **Calculating *total repair cost***
  - By recording labor cost to repair defects
- **Calculating *rework cost***
  - Reduction in rework can be compared with cost of prevention activities
- ***Organizational-level* analysis**
- ***Prediction* of *defects* and *warranty costs***
- ***Prediction* of *reliability***

Aspennw.com



ljser.org

Sciencedirect.com

Defect Repair Cost

Labor Cost to Repair Defects

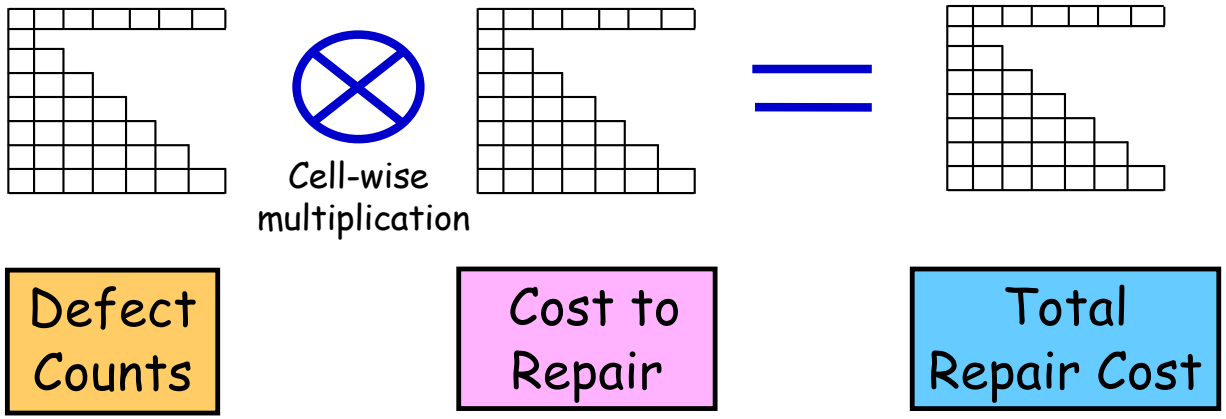
Phase of Injection

Phase of Detection	RA	PD	DD	C&UT	I&T	Post Rel
RA	$1					
PD	$12	$2				
DD	$22	$8	$2			
C&UT	$45	$18	$8	$2		
I&T						
Post Rel						

Cell i,j indicates the *average* labor cost to repair a defect *created* in phase i and *detected* in phase j

Total Repair Cost

If you multiply the defect containment chart by the "labor cost to repair" chart, you get total repair cost



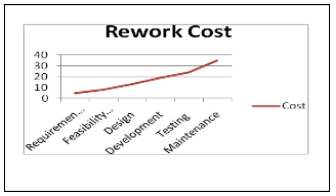
Total Repair Cost Example

Phase of Injection

Phase of Detection	RA	PD	DD	C&UT	I&T	Post Rel
RA	$15					
PD	$144	$110				
DD	$484	$64	$46			
C&UT	$675	$54	$64	$34		
I&T						
Post Rel						

Cell i,j indicates the *total labor cost* to repair all defects *created* in phase i and *detected* in phase j

Rework Costs Are The Portion Of the Prior Chart That Are Not On The Diagonal



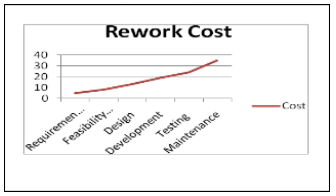
ljser.org

Phase of Injection

Phase of Detection	RA	PD	DD	C&UT	I&T	Post Rel
RA	$15					
PD	**$144**	$110				
DD	**$484**	**$64**	$46			
C&UT	**$675**	**$54**	**$64**	$34		
I&T						
Post Rel						

Costs off-diagonal are rework costs

This Concept Applies Throughout the Product Lifetime



ljser.org

You can *track repair cost* and *rework cost* during development

and

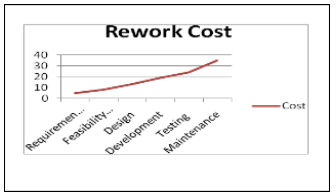
after delivery to the customer


Aspennw.com

- **You can further *break defects down by characteristics*:**
  - Phase of Development where Defect Occurred
  - Severity
  - Importance to Customer
  - Cost to Repair
  - Time to Repair
  - Which Part of the Software was Responsible
  - Etc.


Imgkid.com

This Can Help You Justify Process Improvements



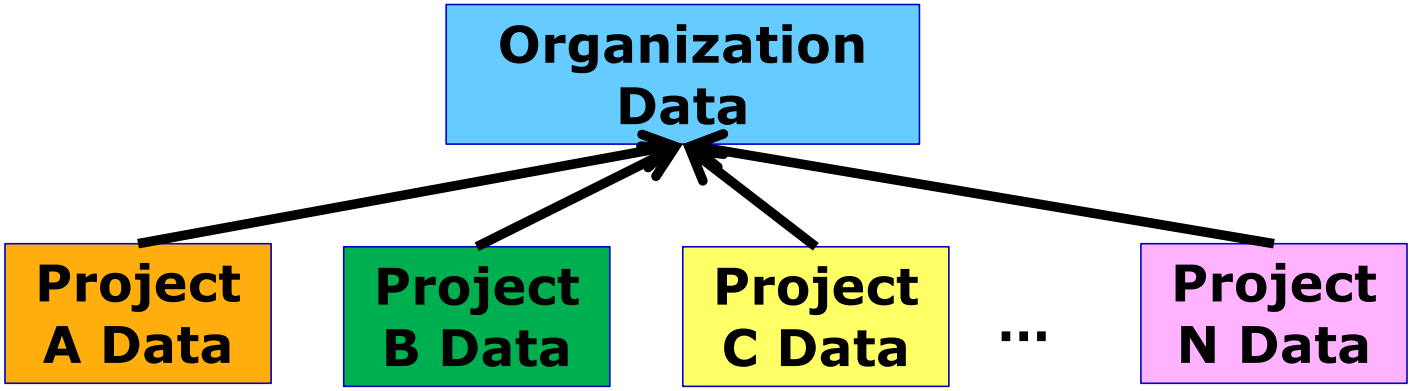
ljser.org

Rework costs are the equivalent of "software scrap"

- **If you can *reduce scrap* by *investing in defect prevention* activities, you can save a lot of money (see earlier slides)**
- **If you make an improvement in your development process, you can *use the defect containment chart to show the savings* in reduced repair cost**
- **And you can use the chart to *determine which parts of the process are most important to improve***

Analyzing Defect Data at the Organizational Level

- **By collecting data from many projects, we can show historical costs for rework**
- **And we can also show *patterns* of defect containment**



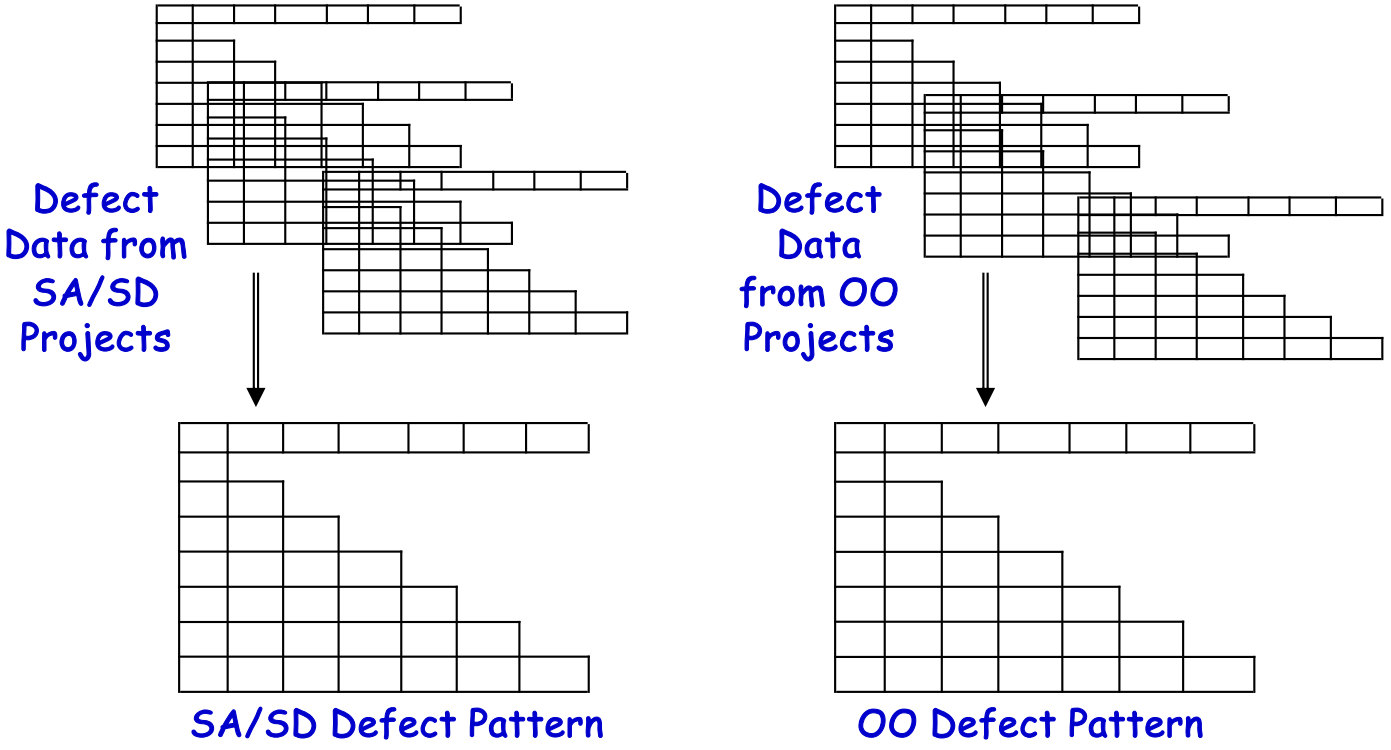
Organizational Analysis of Defect Containment Data

Analysis of defect containment data for many projects over a period of time may show such organizational information as:

- ***Most frequent*** types of defects
- ***Most costly*** defects
- ***Time required to fix*** defects
- ***Process steps generating the most defects***
- ***Which design standards help or hurt defects***

Typically we collect the data needed for statistical process control: averages, ranges, distributions, maximum, minimum, etc.

Example: Determining an Organizational Process Metric



UT Dallas

Software Quality and Software Testing

Part 1 – The Big Picture (How Quality Relates to Testing)
Part 2 – Measuring Software Quality
Part 3 – Software Reliability
Part 4 - Defect Containment

Part 5 – Measuring Software Complexity

Contents

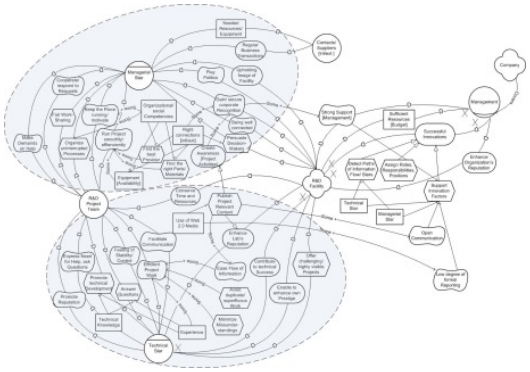
- **Complexity: what and how to measure**
- **Structured Programs and Flowgraph Analysis**
- **Measures of Complexity**
- **Closing Remarks**

Contents

Complexity

We tend to think that complex software is more difficult to develop, test and maintain and has greater quality problems.

But what do we mean by complexity?



Dictionary definitions of complex:

1. Composed of ***many interconnected parts***
2. Characterized by a very ***complicated arrangement of parts***
3. So ***complicated or intricate*** as to be ***hard to understand***

Complex vs Complicated

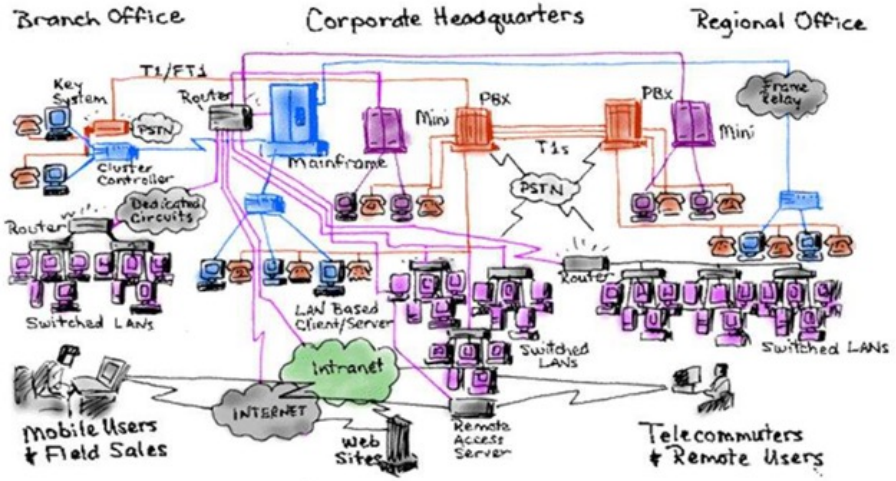
***Complicated*: being <u>difficult to understand</u> but with time and effort, ultimately knowable**

***Complex*: having <u>many interactions</u> between a large number of component entities.**

- As the number of entities increases, ***the number of interactions*** between them will ***increase exponentially***
- It can get to a point where it would be **impossible to know and understand** all of them.

Changing Complex Software

- **Higher levels of complexity in software increase the risk of unintentionally interfering with interactions and so *increase the chance of introducing defects* when making changes.**



Labs.Sogeti.com

- **In more extreme cases, complexity can make *modifying the software virtually impossible*. Changes introduce more problems than they fix. This is called *inherent instability*.**

Can We Measure Complexity?

Measures of complexity would need to address:

- the ***parts*** of the software,
- the ***interconnections*** between the parts,
- and the ***interactions*** between the parts.

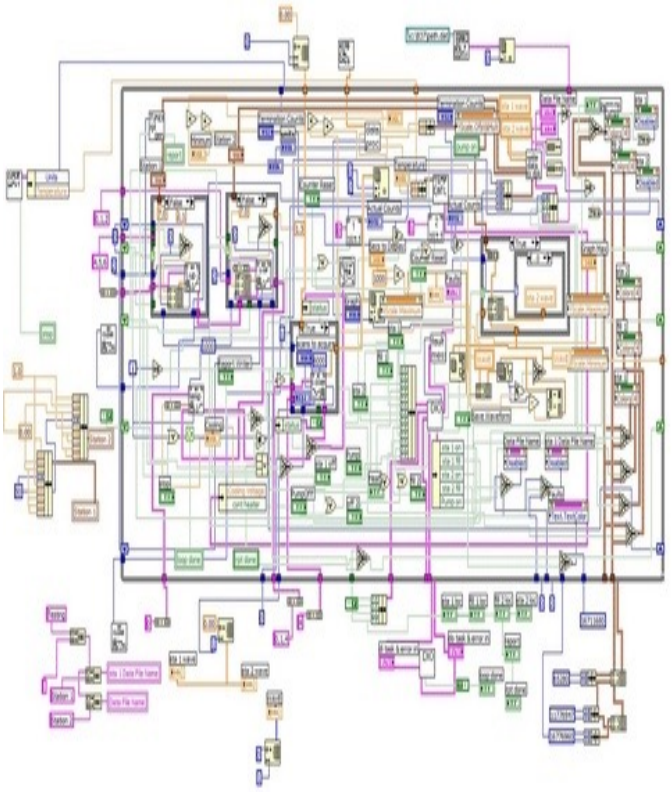
Information Need

- Something that will help us ***estimate***
  - difficulty of programming,
  - difficulty of testing and maintaining,
  - expected level of quality
- Something that will ***help us evaluate and improve our software*** with regard to the above characteristics

How Can We Measure Complexity?

The *base measures* would quantify the attributes of:

- The ***parts*** or ***components*** of the software
- ***How many*** parts or components there are
- The ***arrangement*** of the parts
- The ***interactions*** of the parts



Compound Measures

Combining the base measures into calculations that help us address our information needs, answering questions such as:

- What aspects of software structure can help ***forecast*** development effort and quality?
- ***Is my software structure good***?
- ***How should I test my software***?
- ***How can I improve*** my software structure?
- ***How much has it improved***?

What Can We Measure?

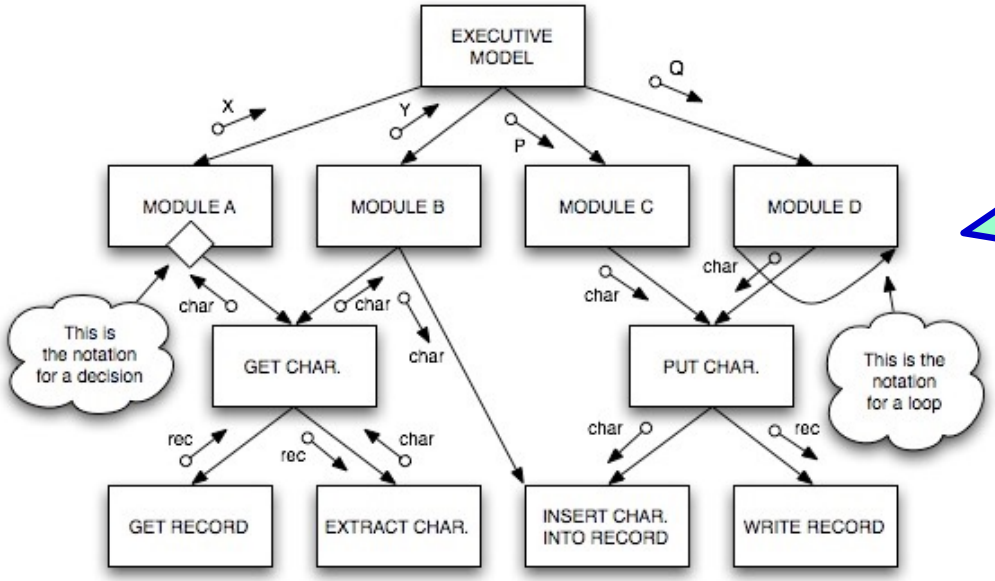
We might learn something about the structure and complexity of software by measuring:

- ***Requirements***
  - Models, use cases, test cases
- ***Architecture and Design***
  - Models, design patterns, structure, control flow, data flow
- The **code** itself
  - Statements, variables, nesting, control flow, data flow
- The ***way the code is assembled*** to produce the final product
  - Load files, use of libraries

One Problem Is That There are Many Systems for Describing Software Structure

Generally Speaking We Measure Complexity of Systems and of Components that Make up Systems

We usually start with the *architecture* of the system



This is the architecture of a system defined using *structured analysis*. There are complexity measures for the system and for the individual components.

With Object Oriented Systems, the Nature of the Components Varies with the Methodology

This means we must sometimes devise *methodology-specific measures*

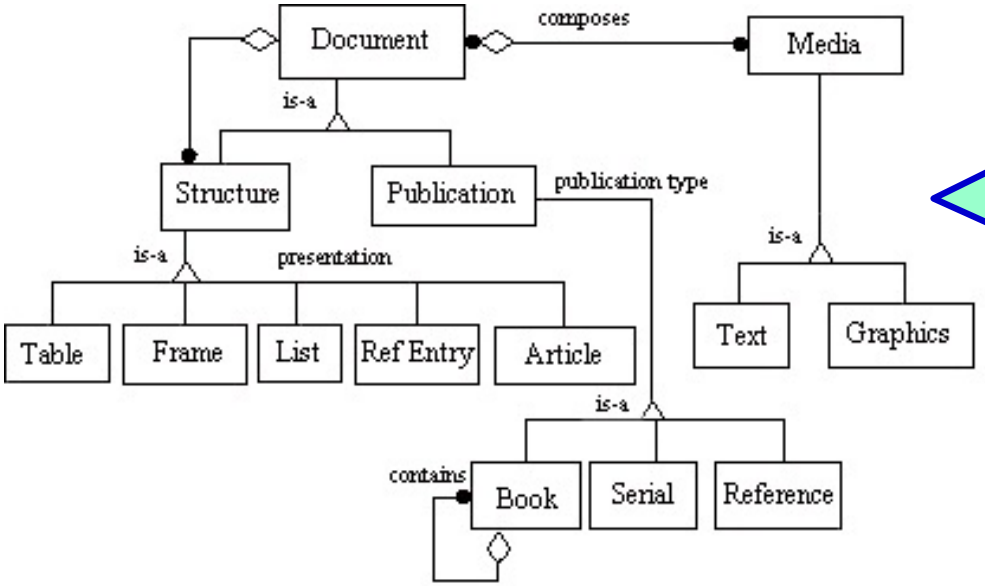


Figure 1. Multimedia Document Model - Object diagram

This is the architecture of a system defined using *object oriented methodology*. There are complexity measures for the system and for the individual components.

Order of Presentation

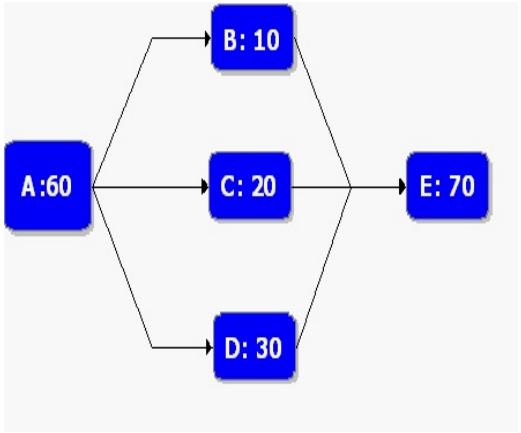
We will focus on complexity of *structured, procedural software*

- Because this is where most of the research has been focused
- Because the results apply to software in many different languages
- Because ***most of the results also apply to object oriented software***

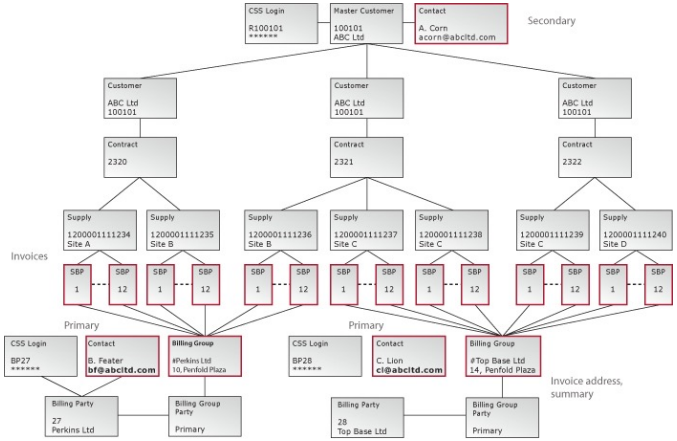
From time to time we will mention how the concepts are applied to *object oriented software*

System Level Complexity

Fundamentally, the *complexity* of a system depends on the *number of components* and the *number of links* between the components of the system



VS



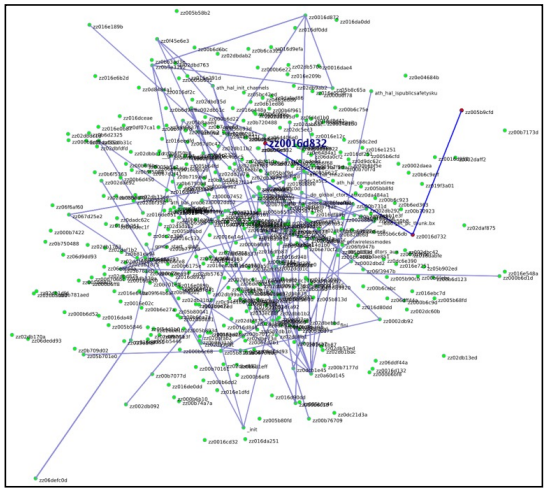
It can be *further complicated* by the degree to which the *components share common elements* (coupling)

Contents

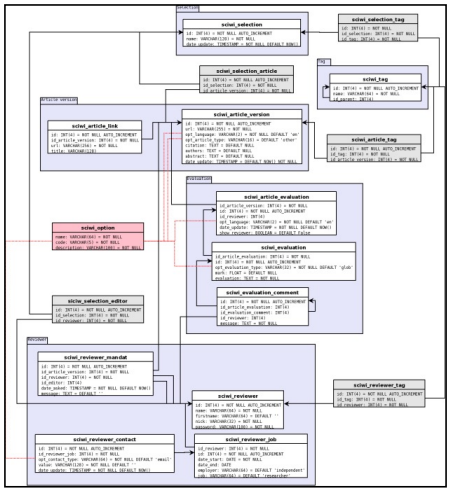
- Complexity: what and how to measure
- ***Structured Programs and Flowgraph Analysis***
- Measures of Complexity
- Closing Remarks

Control Flow Captures Major Complexity-related Attributes

Our intuitive notions of complexity would say that when there are *more parts* and more *complex ways they interact*, we have more complex software.

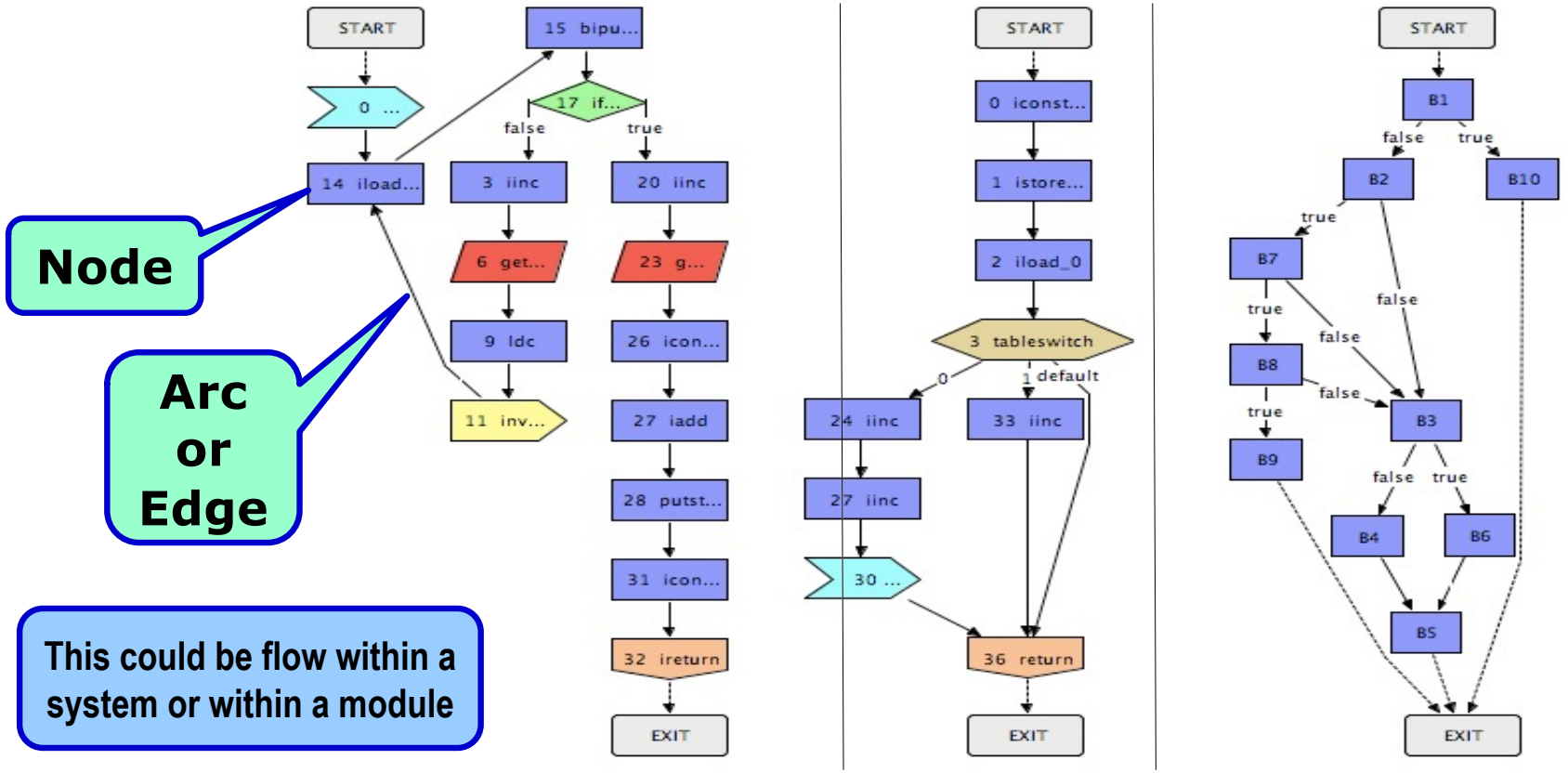


vs



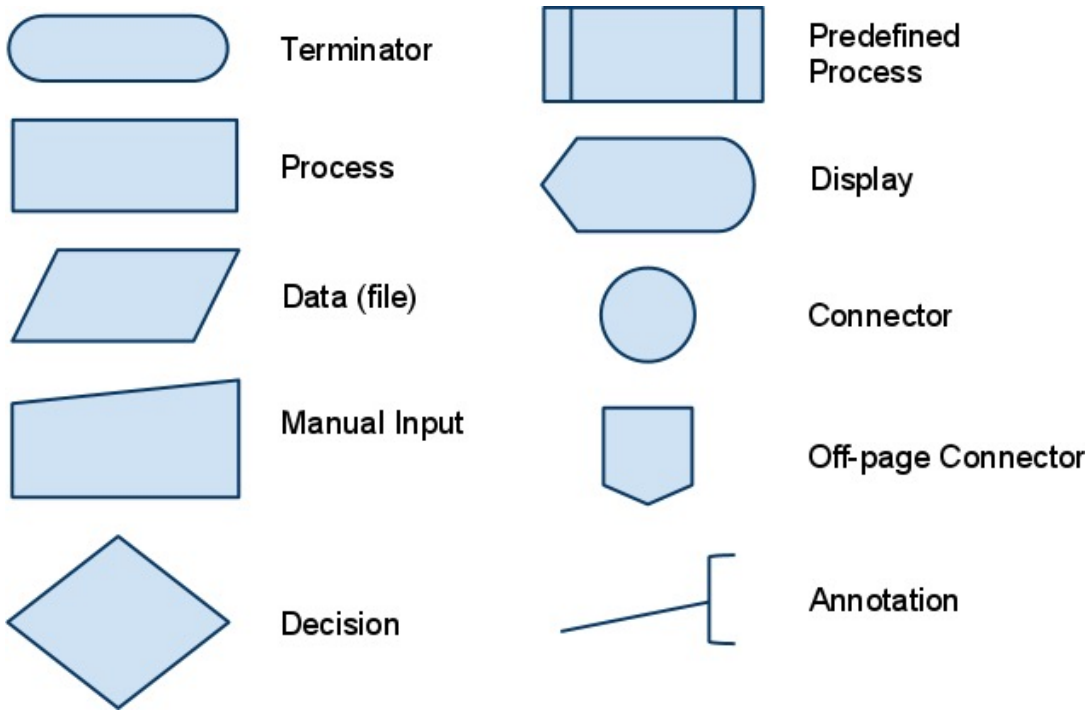
Many measures of complexity make use of control flow analysis.

Control Flow is Often Modeled with Directed Graphs



In Many Notations, the Shape of the Node Conveys the Nature of What it Represents

For example, flowcharts:

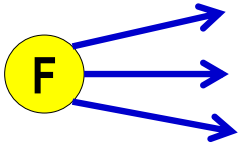


Notation To Be Used Here
(in these slides)

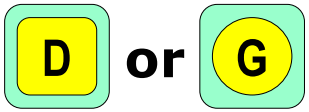
- **Arc or Edge**  A path between nodes
- **Procedure Node**
  - A block of code. Any decisions are internal to the block. One exit.

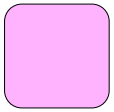
  Squarish shape, Exactly one arc leaving
- **Predicate Node**
  - One that makes a decision.

  Round shape, Two or more arcs leaving
- **Start Node**

- **Stop Node**

Colors of procedure and predicate nodes are not part of the notation.
Colors are used only to clarify points being made on a slide.

A FlowGraph

A flowgraph is a directed graph with

- ***One start node***, and
- ***One end node***,

➢ **that has the following property:**

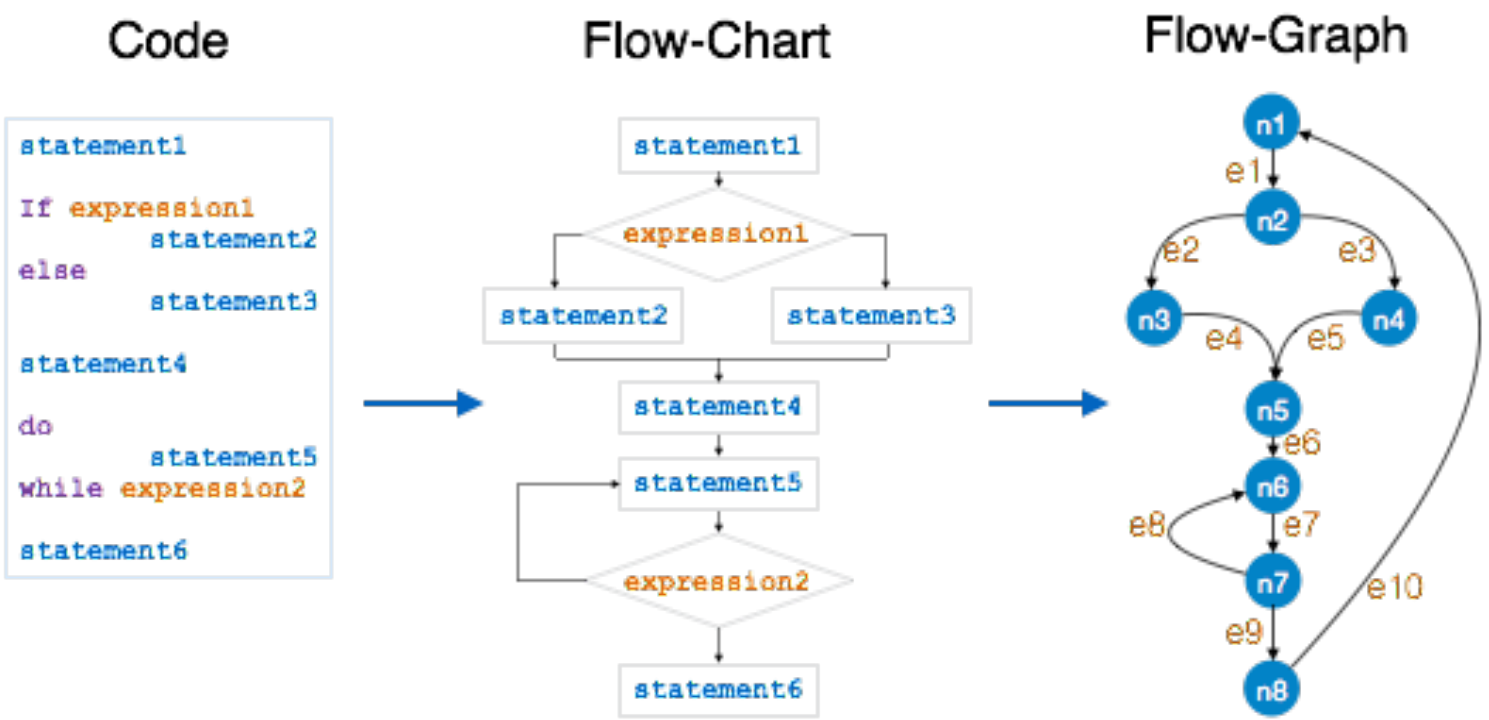
- ***Every other node lies on a path between the start node and the end node***

Notes:

- This notation works for any procedural programming language
- But not all languages can represent all possible flowgraphs
- Certain common language constructs have readily recognized flowgraph forms

See later slides or Fenton, page 379 for some examples.

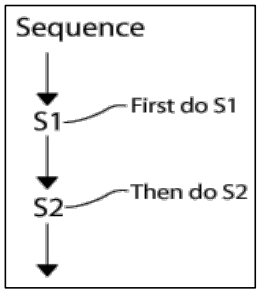
Example: Code, Flowchart, and Flowgraph

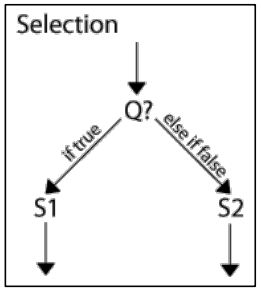


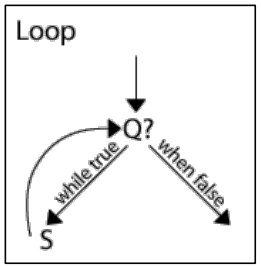
What is a Structured Program?

A structured program is one *constructed out of three fundamental control structures*:

- ***Sequence*** (ordered statements and/or subroutines)
  - Examples: A = B+C; D = FUNC(E,F)
- ***Selection*** (one or more statements is executed, depending on the state of the system)
  - Example: If C1 Then <true option> Else <false option>
- ***Iteration*** [loop] (a statement or block is executed until the program has reached a certain state)
  - Examples: While; Repeat; For; Do… Until

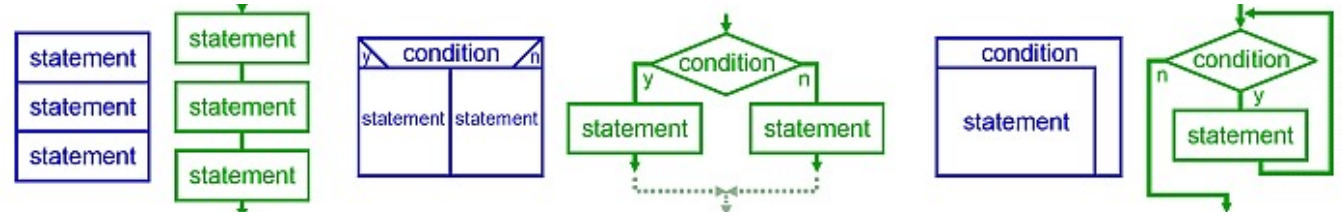




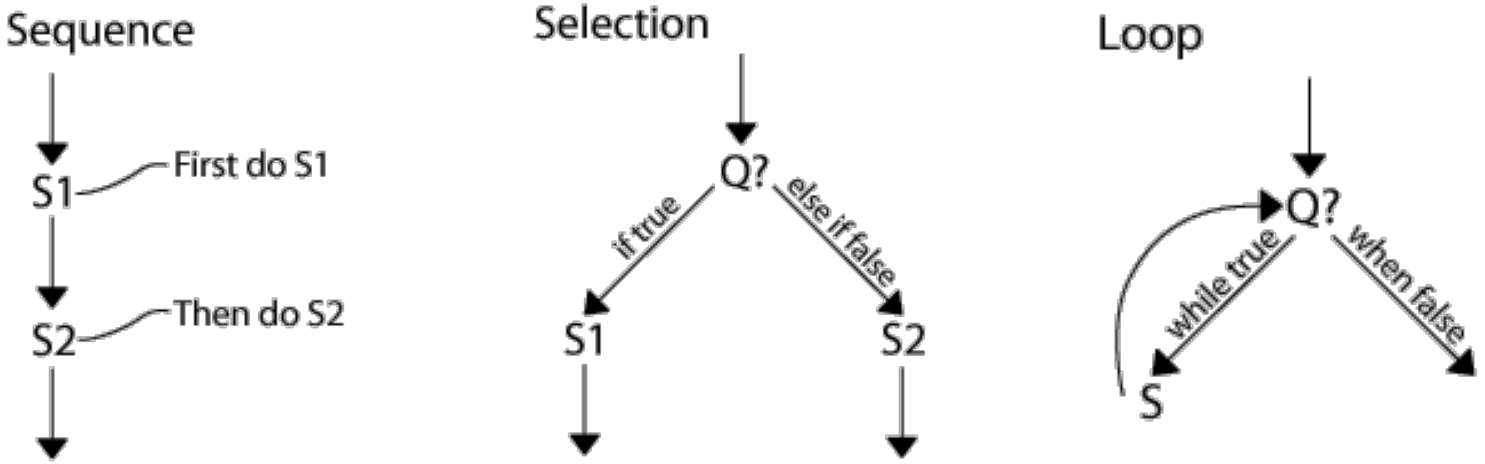


Structured Program Notation

Sequence **Selection** **Iteration (Loop)**



Blue: NS Diagram notation; Green: Flowchart notation



These Three are Sufficient to Represent Any Program

The *structured program theorem*, also known as the Böhm-Jacopini theorem, says that the class of flowgraphs representing *the three control structures above can compute any computable function*

- **Note: This does not necessarily mean it is the only way or the best way.**
- **The theorem simply states that it is possible to represent any function with only the three control structures.**

Why Are Structured Programs Important?

Studies have shown that limiting the software to a small number of well defined control structures has these benefits:

- Easier to understand
- Less error prone
- Easier to analyze and test
- Easier to measure

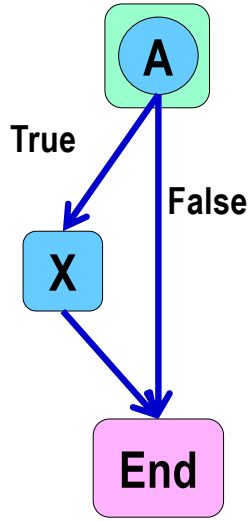
This started out as a theoretical concept, developed by Edsger Dijkstra and others.

It became more widely known when Dijkstra wrote his famous "*Go To Considered Harmful*"[1] letter to the editor of *Communications of the ACM* (in 1968).

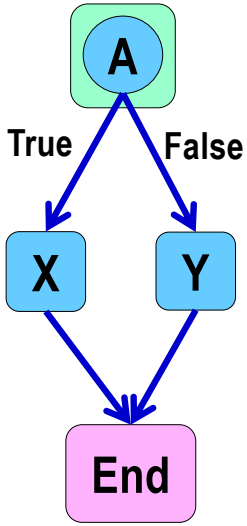
[1] See References

There May Be More Than One Flowgraph Representing A Particular Kind of Control Structure

Example: Two flowgraphs for selection



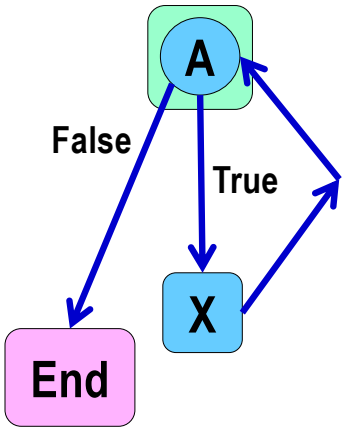
If A then X
(D_0)



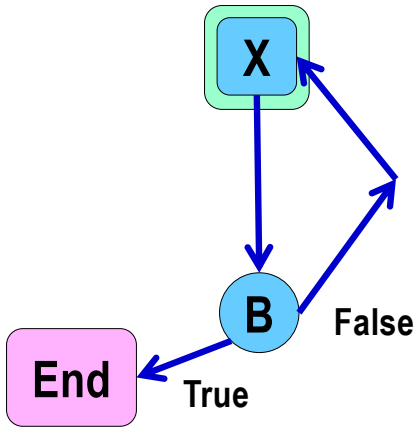
If A then X else Y
(D_1)

Each of these is also a "*prime*" *flowgraph*, meaning it cannot be reduced to a simpler form. We'll discuss this further in later slides.

Two Prime Flowgraphs for Iteration



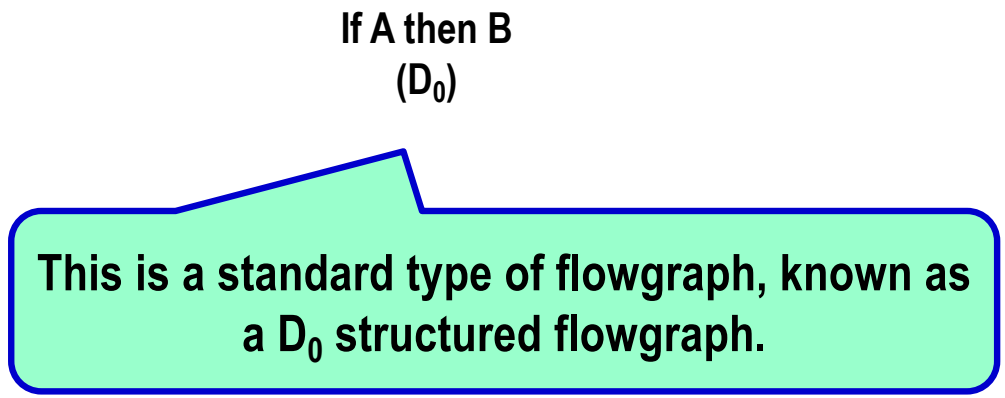
While A Do
X
(D_2)



Repeat X
Until B
(D_3)

Prime Flowgraphs and D Notation

- **A *prime flowgraph* is one that *cannot be reduced* (to a simpler flowgraph).**
  - D_0, D_1, D_2 and D_3 are all prime.
  - See discussion of "reduction" in later slides.
- **The D notation is a widely recognized way of denoting certain standard, prime flowgraphs.**



The Flowgraphs D_0-D_3 (and sequencing) Can Be Used To Represent Any Program

As a result, *some define a program to be "structured" only if it is represented by a combination of these flowgraphs*.

However, there are several additional prime flowgraphs that represent *commonly used language constructs* and that can greatly simplify some programs.

So different organizations and researchers have defined *additional prime flowgraphs* that may be permitted in "structured" programs.

In other words, every organization defines *structured* in its own way.

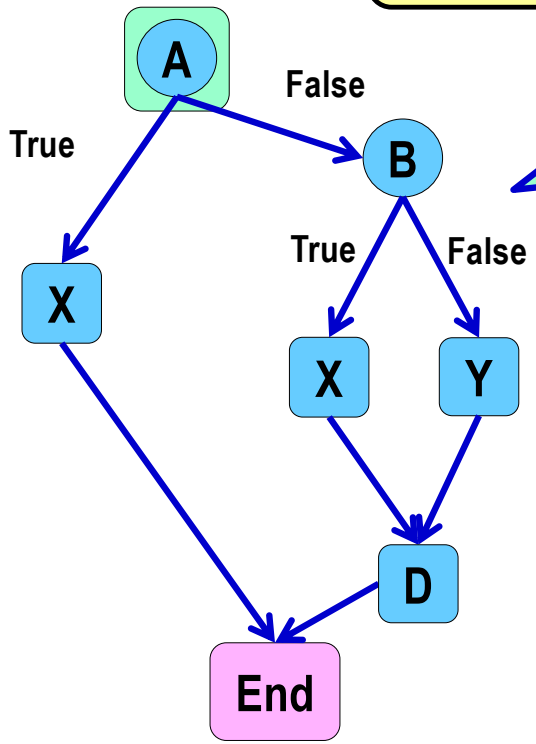
Structured Program Flowgraphs: What Is Common and What Is Not

- **What all structured programs have in common**
  - Definitions of edges, nodes, etc.
  - Built out of the ***three fundamental constructs***: sequence, selection, and iteration
  - It must be ***possible to reduce*** a program to a combination of a selected set, **S**, of prime flowgraphs
- **What is Different**
  - Which prime flowgraphs are included in the set **S**.

See Fenton, section 9.2 for a discussion of flowgraphs and structure and, in particular, section 9.2.1.2 for a *generalized notion of structuredness*.

An Example of Why Additional Prime Flowgraphs are Useful

IF A or B THEN X
ELSE Y

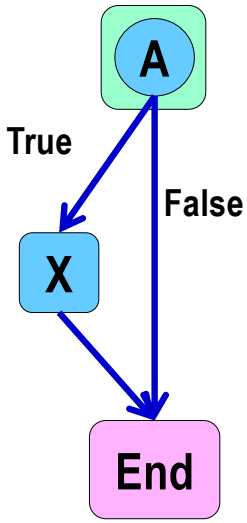


If only D0 and D1 can be used to represent this code, then we must use a D1 within another D1 and *must show X twice.*
This is the equivalent of rewriting the source code as shown below.

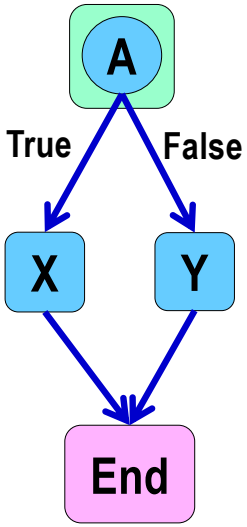
IF A THEN X
ELSE
IF B THEN X
ELSE Y

X must be duplicated. If X is a lot of code this is inconvenient.

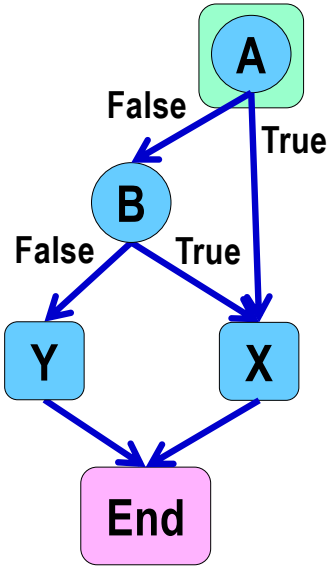
D_5 Was Introduced To Allow Common Boolean Selection Decisions



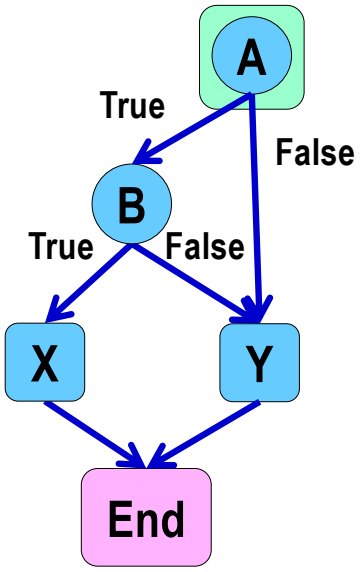
If A then B
(D_0)



If A then B else C
(D_1)

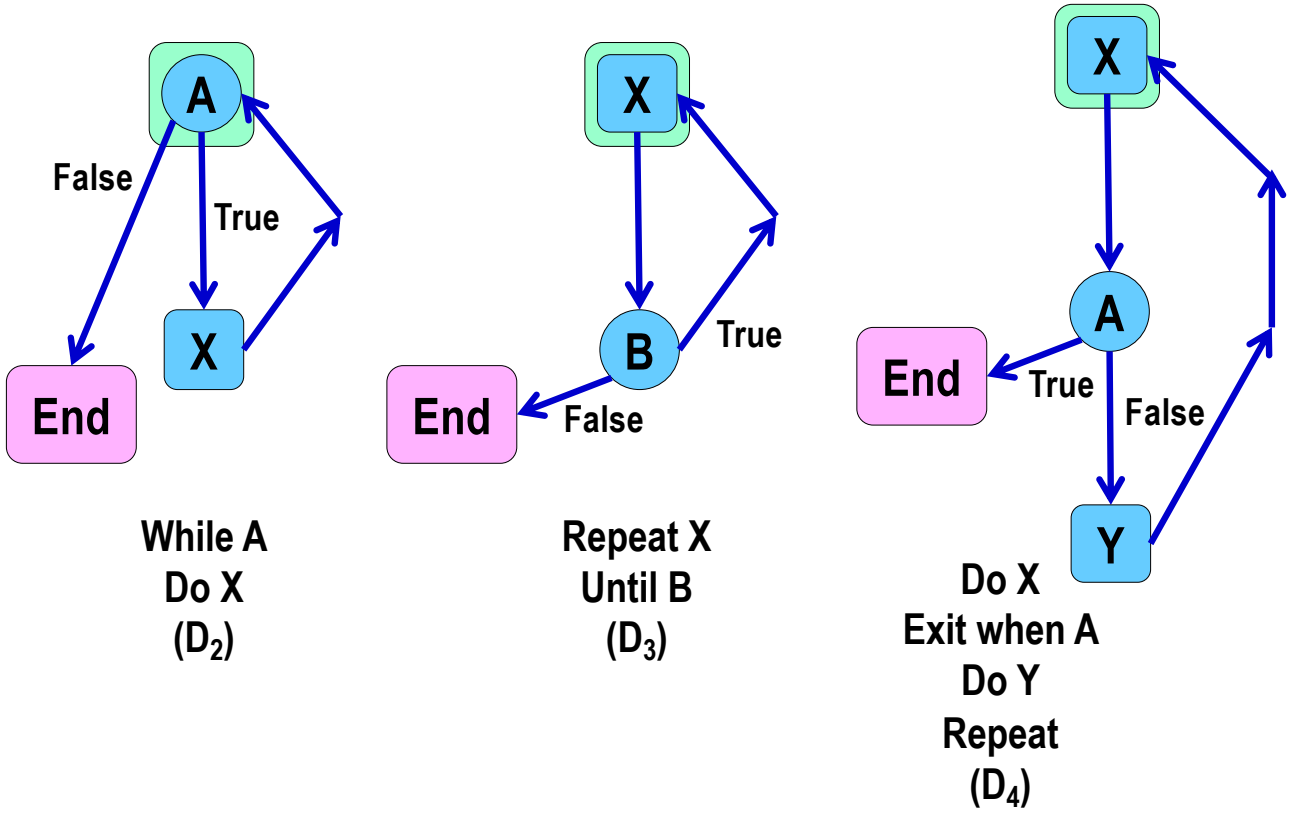


If A or B then X
else Y
(D_5)

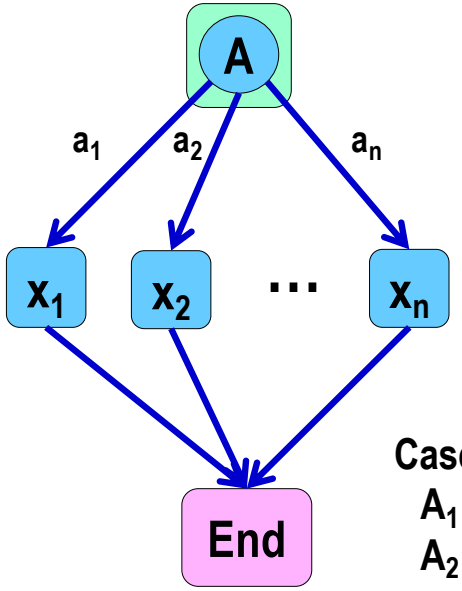


If A and B then X
else Y
(D_5)

D_4 Was Introduced to Allow Middle-Exit Loops



C Flowgraphs are Prime Flowgraphs for CASE Statements

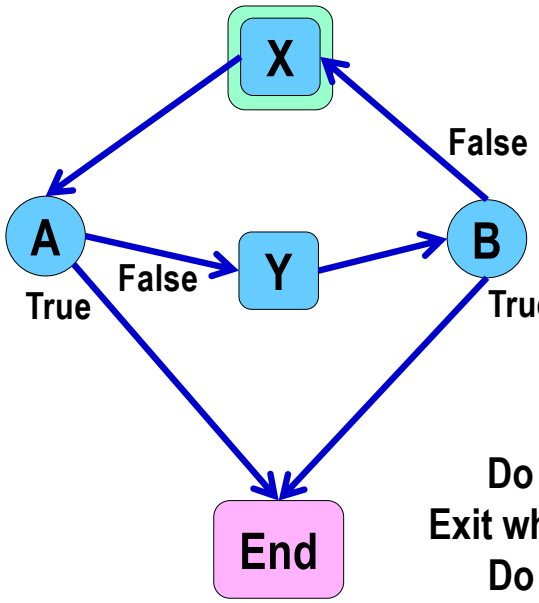


Case A of
$A_1 : X_1$
$A_2 : X_2$
...
$A_n : X_n$
$(C_{1...n})$

Note that there are an arbitrary number of these, depending on n – the number of possible selections.

Note also that these are classified as "C" structured flowgraphs, not "D" structured flowgraphs, because, technically, the CASE statement is not one of the three fundamental control structures.

L Structured Flowgraphs Represent Multi-Exit Loops



Do X
Exit when A
Do Y
Exit when B
Repeat
(L_2)

A two-exit loop is shown (L_2). This is commonly used. However higher numbers of exits could be represented as well.

This also has its own classification (L) rather than being considered a D flowgraph because it is not one of the three fundamental control structures.

Why Use Flowgraphs to Measure Complexity?

- **Directed Graphs clarify the flow of control between software elements**
- **Many measures of software complexity can be determined from directed graphs**
- **It is fairly easy to represent any program with a directed graph**
  - Note that there might be several ways to graph a program, but they should all have the same measure of complexity if they are done correctly

Combining Flowgraphs

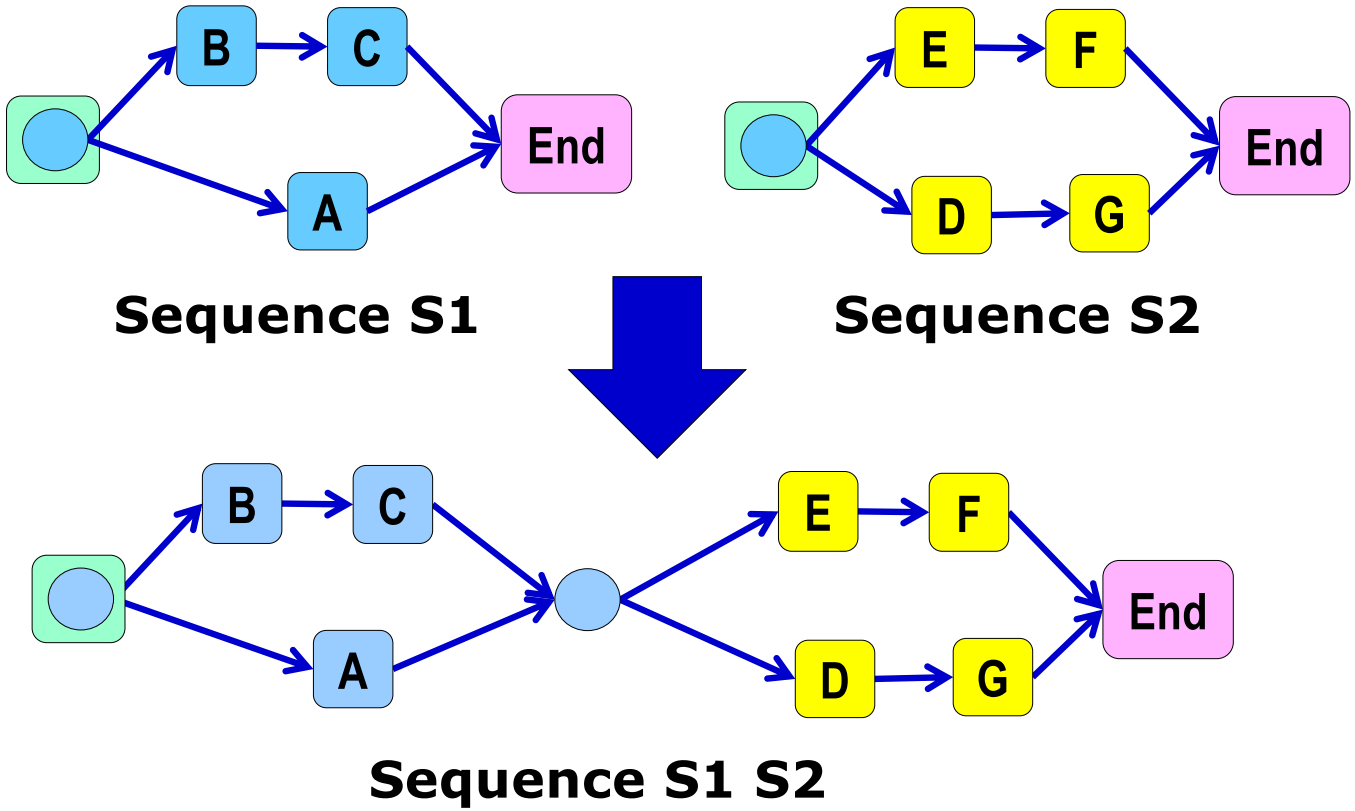
Flowgraphs with a single entry and single exit can be *combined* in the following ways:

- ***Sequencing*: *Merging the end node* of one flowgraph *with the start node* of the other**
- ***Nesting*: *Replacing an arc* in one flowgraph *with the other flowgraph***

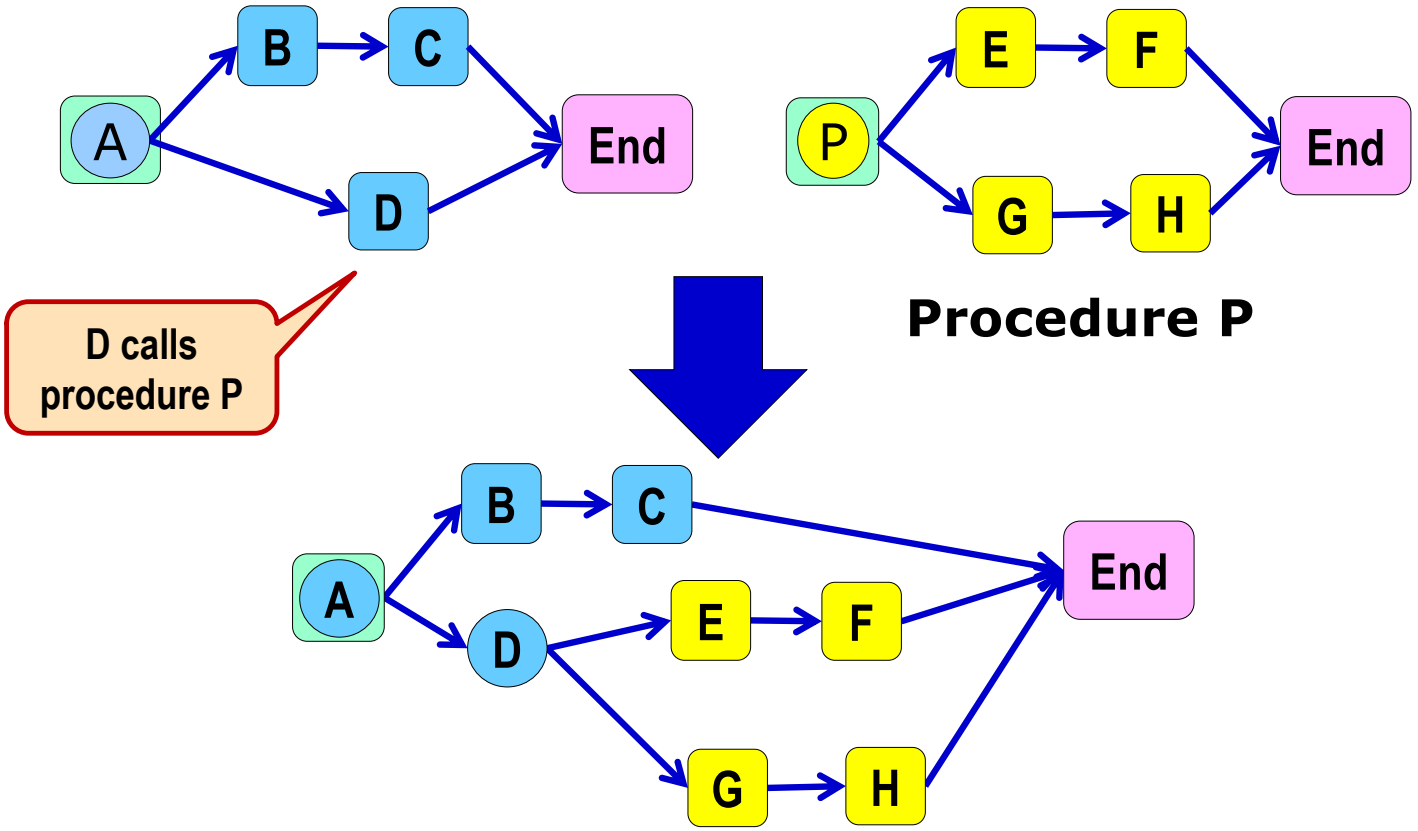
Flowgraphs can also be *reduced* or *condensed* or *decomposed* by reversing the above

- **For example, collapsing a nested flowgraph into a single node and arc**
  - This is, conceptually, the equivalent of replacing the nested flowgraph with a procedure call

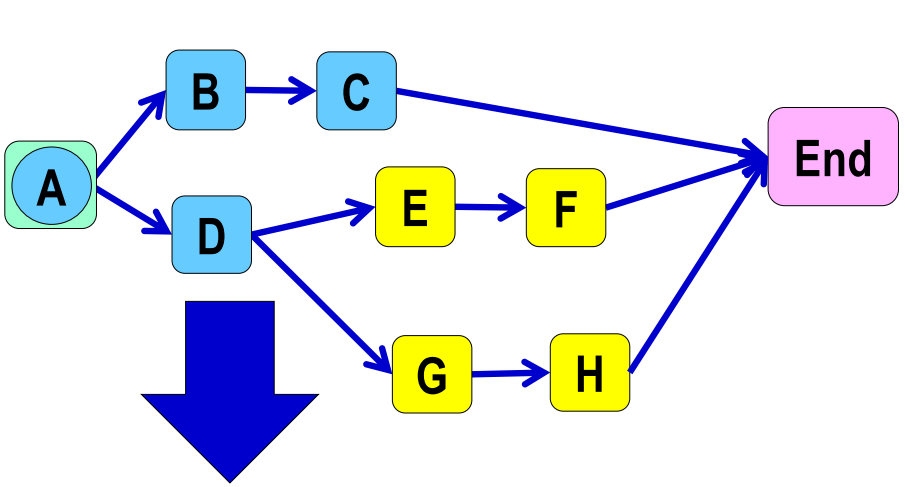
Sequencing Example



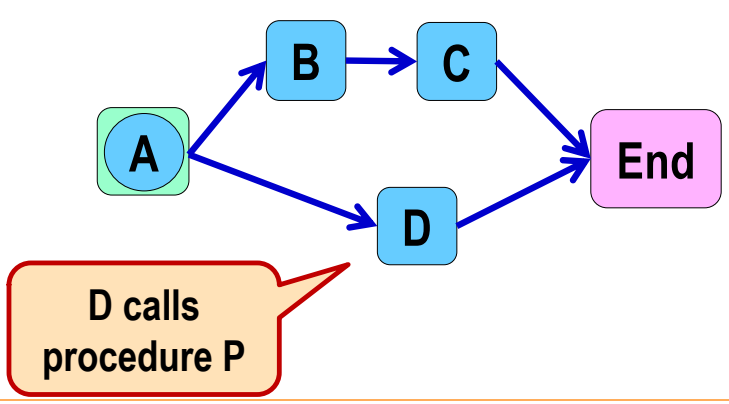
Nesting Example



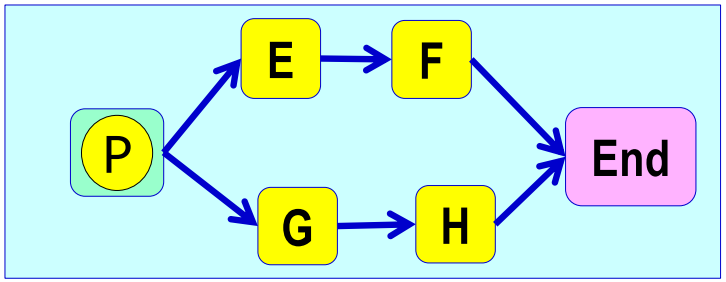
Reduction Example 1



Any single-entry, single-exit sub-graph can be replaced by a procedure call

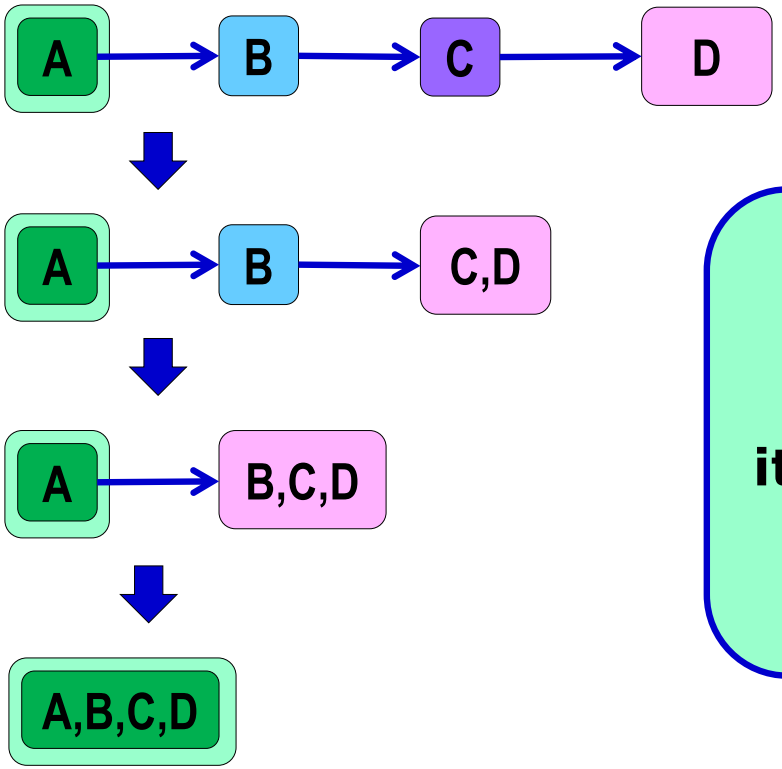


D calls procedure P



Procedure P

Reduction Example 2



Any sequence containing no decisions or iterations can be reduced to a single node

McCabe Cyclomatic Complexity

The *Cyclomatic Complexity* (v) of a Module or a System is:

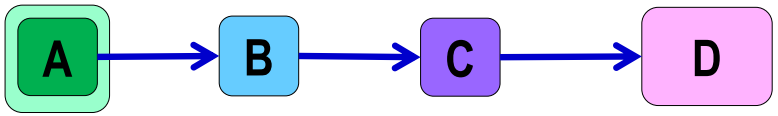
- The number of **linearly independent[1] paths** (basis paths) through the module or system
- If F is a flowgraph[2], then **v(F) = e – n + 2**
  - Where **e** is the number of edges (arcs)
  - And **n** is the number of nodes
- If a system consists of multiple flowgraphs that are not connected together, the formula becomes:

  v(F) = e – n + 2c
  - Where **c** is the number of separate flowgraphs[3]

[1] To be discussed a little later [2] With one entry and one exit
[3] In graph theory these are called ***connected components***

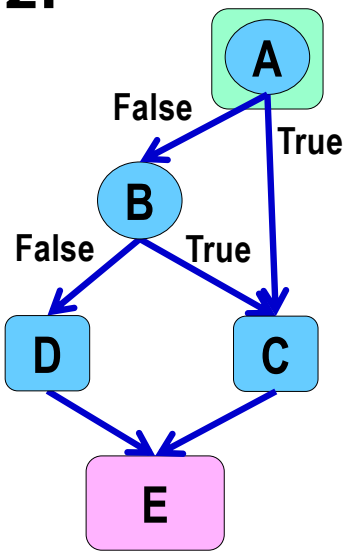
Examples of Cyclomatic Complexity

- **Example 1:**



  - **v(F) = e – n + 2 = 3 – 4 + 2 = 1**
  - There is only 1 path through the code

- **Example 2:**



  - **v(F) = e – n + 2 =**
    6 – 5 + 2 = 3
  - There are 3 paths through the code:
    - A B D E
    - A B C E
    - A C E

Why Is Cyclomatic Complexity Useful?

- **Number of paths indicates *maximum number of separate tests* needed to test all paths**
  - This should relate to the ***difficulty of testing*** the program
- **It also indicates the *number of decision points* in the program (plus 1)**
  - This should relate to the ***difficulty of understanding and testing*** the program

Cyclomatic complexity is not a perfect measure of these things (see Fenton, chapter 9) but it is a fairly reliable guide.

The Higher the Cyclomatic Complexity, the Harder the Code Is to Maintain

Cyclomatic Complexity

CC Value	Interpretation	Bad Fix Probability*	Maintenance Risk
1-10	Simple procedure	5%	Minimal
11-20	More complex	10%	Moderate
21-50	Complex	20% - 40%	High
50-100	"Untestable"	40%	Very High
>100	Holy Crap!	60%	Extremely High

*Bad Fix Probability represents the odds of introducing an error while maintaining code.

What Do We Mean by Linearly Independent Paths?

The *number of linearly independent paths* is the minimum number of end-to-end paths required to *touch every path segment* at least once.

- Sometimes the actual number of paths needed to cover the system is less than this because it may be possible to combine several path segments in one traversal.

***There may be more than one set of linearly independent paths* for a given flowgraph**

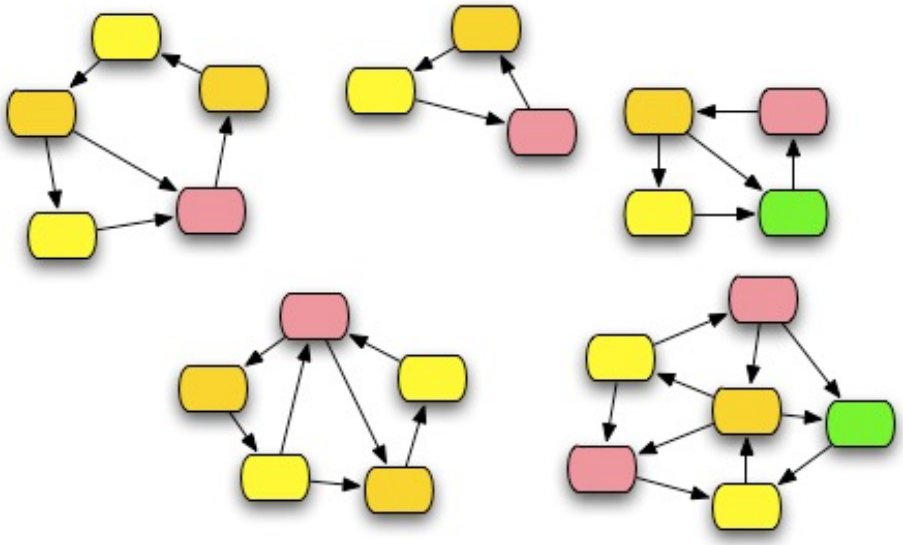
- This becomes more likely as you get more complex flowgraphs

Determining a set of linearly independent paths is something you might study in a course on testing or in a course on graph theory

- It gets harder as the cyclomatic complexity goes up

A Graph with Five Connected Components

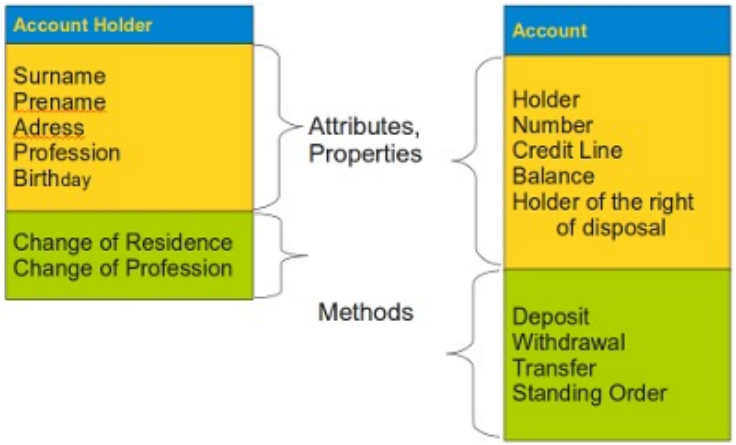
This graph has *five separate regions*, which are connected within themselves, but not to each other. Each region is called a *connected component*.



The graph above is not a flowgraph by our strict definition, because it has more than one start and stop node and not all nodes are connected to any given start or stop node. But it illustrates the concept of *connected components*.

Why Would We Care About Graphs with Many Connected Components?

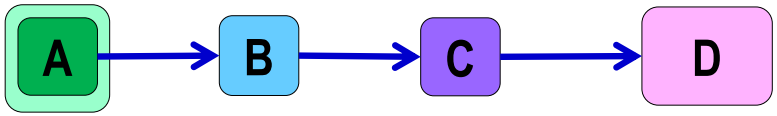
- **We could measure the cyclomatic complexity of a *system consisting of several separate modules***
- **In object oriented systems we could measure the cyclomatic complexity of a *class containing multiple methods***



McCabe Essential Complexity

The *Essential Complexity* (ev) of a Module or a System is:

- The cyclomatic complexity of the **fully reduced flowgraph**
- **Example:**



  - **ev(F) = 1** because this can be reduced to one node

➢ **If the flowgraph is constructed completely of prime flowgraphs (i.e., it is structured) then the essential complexity will be 1.**

Some Issues with Essential Complexity
(slide 1 of 2)

Essential complexity is intended to tell us *how well structured a program is.*

However

- **As originally defined, the only valid primes were the four D structured primes: D_0, D_1, D_2, D_3**
  - So if you allow additional primes, do you revise the definition of essential complexity to include the new primes?
  - Do you allow D_4 and D_5 but nothing else?
  - What about the C structured primes and the L structured primes?

Some Issues with Essential Complexity
(slide 2 of 2)

If your program is not "structured" it isn't clear whether the essential complexity tells us much beyond that

- Does a larger essential complexity actually mean anything?
- If two programs have the same essential complexity, are they equally complex?
  - See fig. 9.13 in Fenton for an example
  - He shows two flowgraphs that have the same essential complexity, but intuitively one of them is a lot more complex and harder to understand than the other.

Contents

There is No Single Measure of Complexity

- **As we have seen, there are different ways to measure complexity**
- **Research shows that sometimes the attributes of complexity may conflict**
  - For example
    - low coupling doesn't always mean high cohesion
    - low cyclomatic complexity doesn't always mean easy to understand
    - structured software may be awkward to produce in languages without certain constructs

Use complexity measures as guidelines, not as "magic numbers" that result in rigid requirements for code.

END OF
Part 5

Any Questions?

End of Lecture

References
Part 1

Bourque, P. and R.E. Fairley, eds., *Guide to the Software Engineering Body of Knowledge, Version 3.0*, IEEE Computer Society Press, 2014. ISBN 978-0769551661. Available in PDF format (free) at www.swebok.org.

Crosby, Philip, *Quality is Free*. New York: McGraw-Hill, 1979. ISBN 0-07-014512-1.

Fenton, Norman and James Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, Chapman and Hall, 2014. ISBN 978-1439838228.

Juran, Joseph M., *Juran on Quality by Design: The New Steps for Planning Quality into Goods and Services*. Free Press, 1992. **ISBN-13:** 978-0029166833.

Project Management Institute, *SWX – The Software Extension to the PMBOK Guide Fifth Edition, Project Management Institute, 2013. ISBN 978-1628250138.*

Weinberg, Gerald M., *Quality Software Management, Volume 1, Systems Thinking*, Dorset House, New York, 1992. ISBN: 0-932633-22-6.

References
Part 2

Devore, Jay, N. Farnum, and J. Doi, *Applied Statistics for Engineers and Scientists, 3nd Edition*, Thompson, 2013. ISBN 978-1133111368.

Fenton, Norman and James Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, Chapman and Hall, 2014. ISBN 978-1439838228.

Stevens, S. S., *"On the Theory of Scales of Measurement". Science* (7 June 1946). 103 (2684): 677–680.

References – Part 3

- Lyu, Michael R., *Handbook of Software Reliability Engineering*, IEEE, 1996, Catalog # RS00030. ISBN 0-07-039400-8.
- Musa, John, *Software Reliability Engineering: More Reliable Software, Faster Development and Testing,* McGraw Hill. ISBN: 0-07-913271-5.
- Xie, M. *Software Reliability Modeling*, World Scientific, London, 1991. ISBN 981-02-0640-2.

References
Part 4 (1 of 2)

Chatfield, C., *Statistics for Technology, A Course in Applied Statistics, Third Edition*, Chapman and Hall, London (1983), ISBN 978-0412253409.

Fenton, Norman and James Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, Chapman and Hall, 2014. ISBN 978-1439838228, Chapter 6.

Hedstrom, John and Dan Watson, "Developing Software Defect Prediction," *Proceedings, Sixth International Conference on Applications of Software Measurement*, 1995.

Jones, Capers, *Applied Software Measurement*, McGraw Hill, 1991. ISBN: 0-07-032813-7.

Knuth, Donald, *Seminumerical Algorithms: The Art of Computer Programming, Vol II,* Addison-Wesley, 1969. ASIN: B00157WFAU

References
Part 4 (2 of 2)

Ott, R.L. and M. T. Longnecker, *An Introduction to Statistical Methods and Data Analysis, 6th Edition*, Duxbury Press (2008), ISBN 978-0495017585.

Snyder, Terry and Ken Shumate, Defect Prevention in Practice (Draft white paper), Hughes Aircraft Company, October 22, 1993.

Ross, Sheldon M.. *Introduction to Probability Models*, Academic Press, 1993. Musa, John, *Software Reliability Engineering: More Reliable Software, Faster Development and Testing,* McGraw Hill. ISBN: 0-07-913271-5.

References
Part 4 (1 of 2)

Abran, A., et. al., "Functional Complexity Measurement", *Proceedings, IWSM 2001 - International Workshop on Software Measurement*.

Chidamber, S. and Chris Kemerer, *A Metrics suite for Object Oriented Design*, MIT Sloan School of Management E53-315 (1993).

Fenton, Norman and James Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, Chapman and Hall, 2014. ISBN 978-1439838228**, Chapter 9**

Fenton, N. and A. Melton, "Deriving Structurally Based Software Measures," *Journal of System Software*, vol 12 (1990), pp 177-187.

Henry, S. and D. Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering*, Volume SE-7, No. 5 (Sept, 1981), pp 510-518.

References
Part 4 (2 of 2)

IEEE 9982.2 (1988). *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, A25. Data of Information Flow Complexity.* P112.

Stevens, W., G. Myers and L. Constantine, "Structured Design", *IBM Systems Journal*, vol 13, no 2 (1974), pp 115-139.

Kitchenham, B. A., "Measuring to Manage", in Mitchell, Richard J. (editor), *Managing Complexity in Software Engineering*, London, Peter Peregrinus, Ltd. (1990). ISBN 0 86341 171 1

Lavazza, L. and G. Robiolo, "Functional Complexity Measurement: Proposals and Evaluations", *Proceedings of ICSEA 2011: the Sixth International Conference on Software Engineering Advances.*