UT Dallas

Software Quality and Software Testing

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

UT Dallas

Software Quality and Software Testing

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

Part 2 –Achieving Software Quality

Part 3 - Defect Containment

Part 4 – Measuring Software Complexity

Dennis J. Frailey

Retired Principal Fellow - Raytheon Company

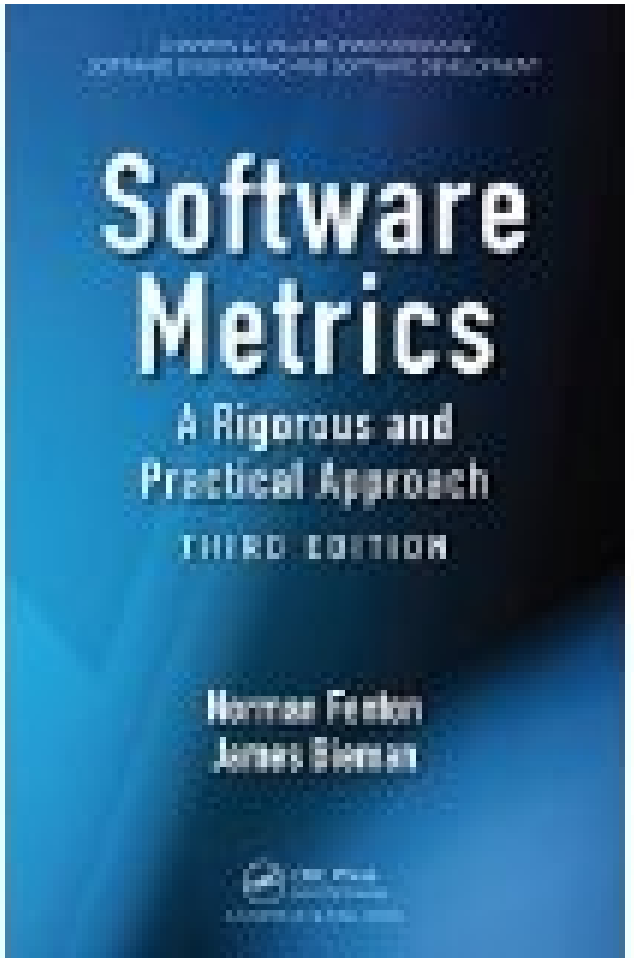
PhD Purdue, 1971, Computer Science
Assistant Professor, SMU, 1970-75
Associate Professor, SMU, 1975-77
(various titles), Texas Instruments, 1974-1997;
(various titles), Raytheon Co. 1997-2010
Adjunct Associate Professor, UT Austin, 1981-86
Adjunct Professor, SMU, 1987-2017
Adjunct Professor, UT Arlington, 2014-2020

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

Part 1 - Outline

➢The Scope of Software Quality

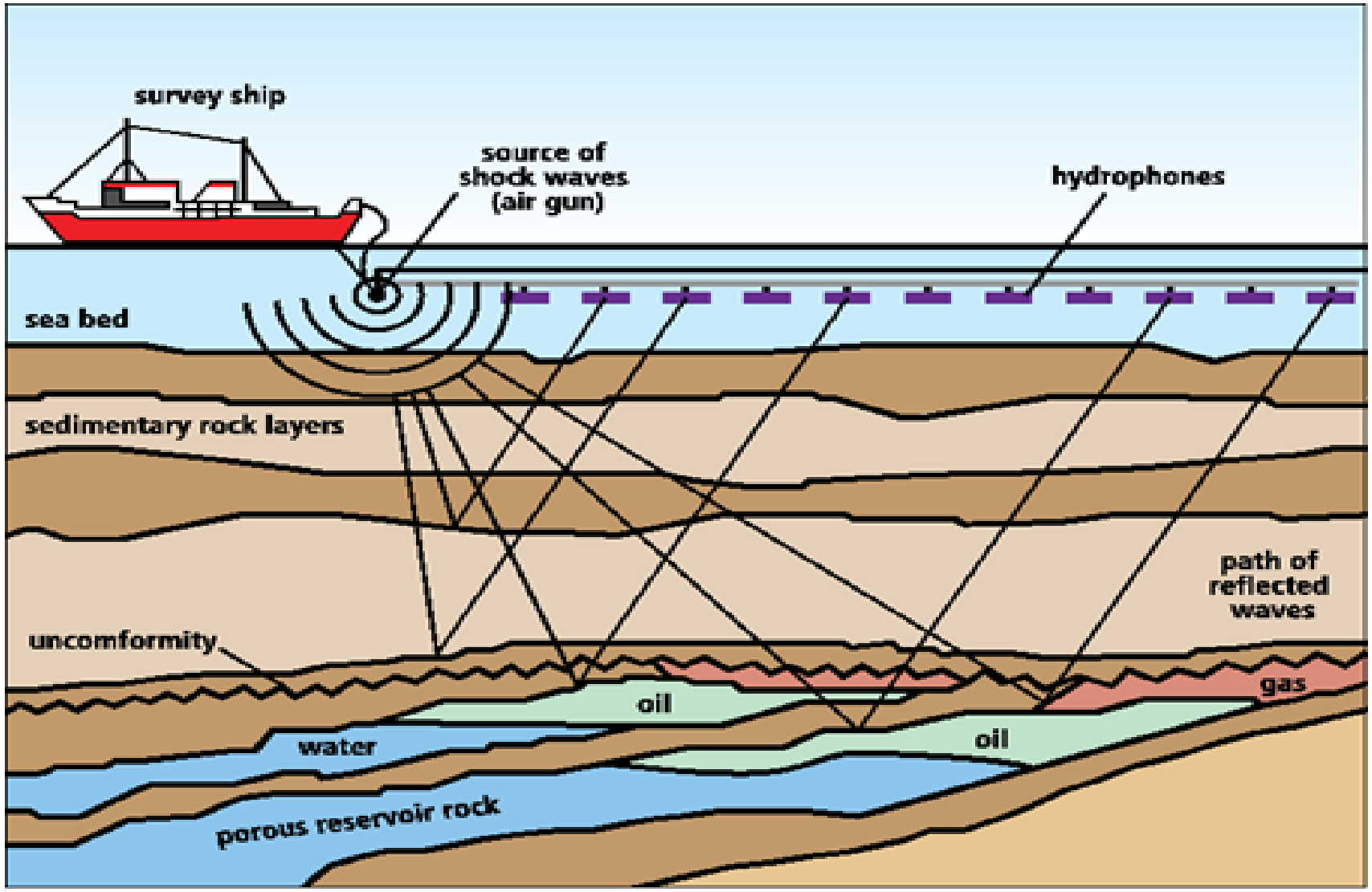
- **Defining Quality**
- **Observations on the Testing Process**

My Story

How I Learned the Importance of Software Quality and Reliability

My First Really Big and Difficult Computing Problem

Marine Seismic Exploration



Characteristics of the Situation

- **About a dozen "ships" around the world, searching for oil**
  - 26' ships – many sailors would call these "boats", not "ships"
- **1970's-vintage minicomputer**
  - Less memory and computing power than a modern smart phone
- **The computer must:**
  - ***Navigate the ship*** so you know where you are
    - There was no GPS - but did have satellite signals twice a day
  - ***Collect and record the seismic data***
    - Massive amounts of data
  - ***Do cursory analysis*** of the data
- **Each ship only comes to port once every 3 months or so**
  - Helicopters bring supplies about once a month
- **It costs several million dollars a day to operate each ship**

Consequences of a Software Failure Phase 1 – Getting to the Ship

- **Problem is described via satellite phone to central facility**

Content.time.com

- **If no quick fix is found, the responsible programmer is identified**

Pinterest.com

- **The responsible programmer is flown to the nearest port**

- **Then flown to the ship via helicopter**

Verticalmag.com

Consequences of a Software Failure
Phase 2 – On the Ship

- **The programmer is seasick**

  for a day or two

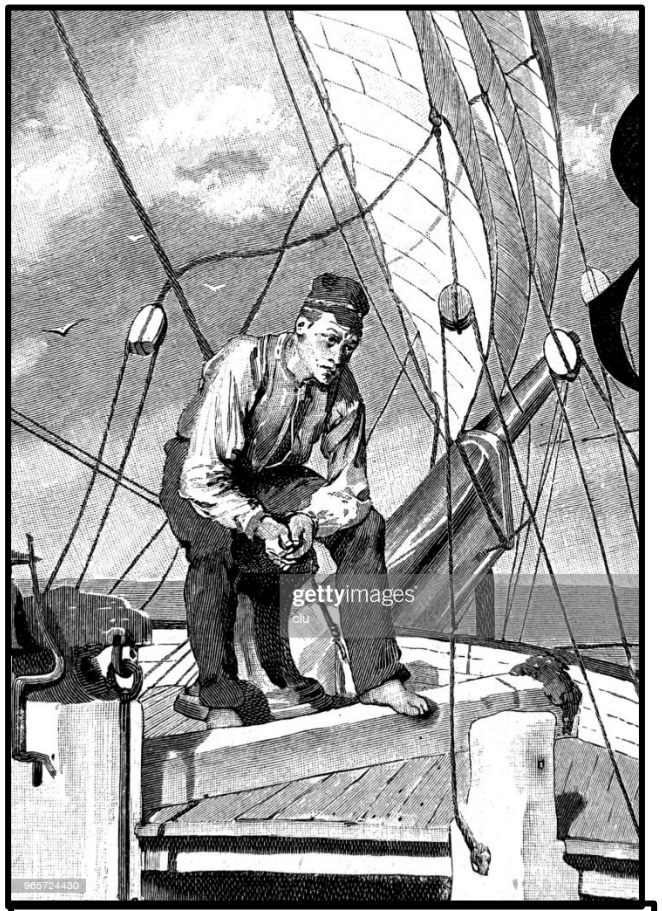
Sailingscuttlebut.com

- **The programmer fixes the problem**

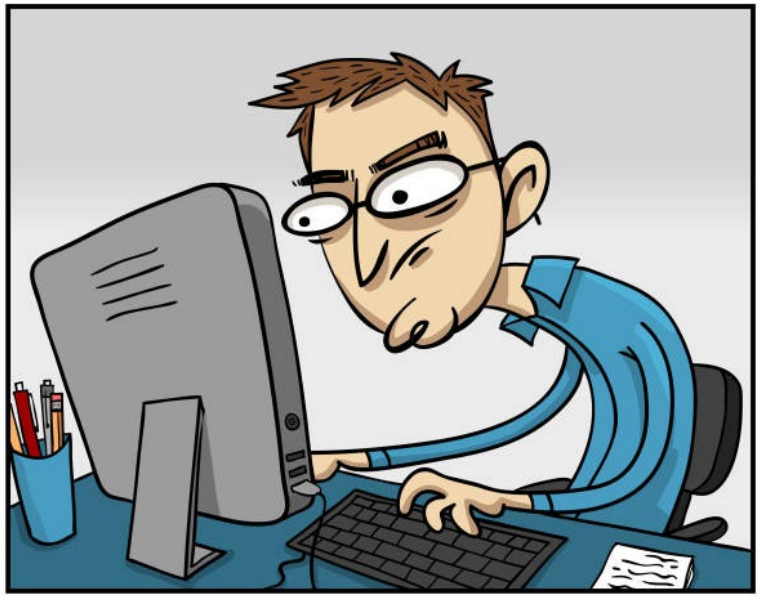
Science.museum.com

Consequences of a Software Failure
Phase 3 – Getting Off the Ship

- **The programmer stays on the ship until it next comes to port**
  - It would cost too much to send them back by helicopter
  - Occasionally a supply helicopter will have room for an extra passenger
  - Sometimes the captain will let you off on a nearby shore



In Other Words

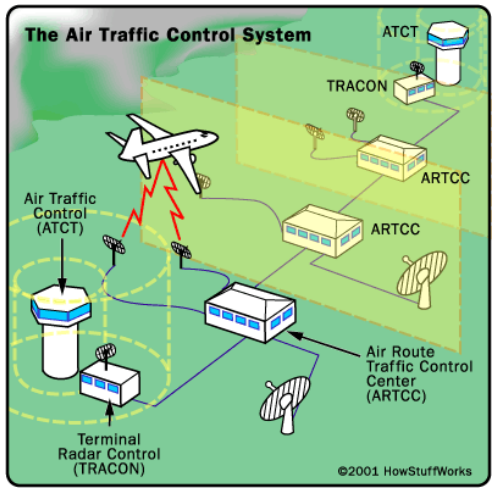


The Programmer is Very Strongly Motivated to have Highly Reliable Software

Real Projects for Real Customers

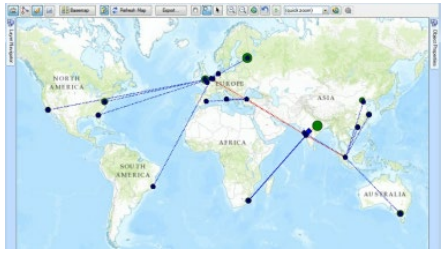
Most Interesting Projects are Big, Complex and Challenging

Projects are Often Big & Complex



Characteristics of Big Projects

- **Lots of People – hundreds or even thousands**
- **Millions of lines of code**
- **Many different companies may be involved**
- **Multiple locations**



- **Many different disciplines**
  - **Systems engineers**
  - **Quality engineers**
  - **Mechanical engineers**
  - **Software engineers**
  - **Electrical engineers**
  - **Safety engineers**
  - **Logistics engineers**
  - **Financial experts**
  - **Project managers**
  - **Subcontract managers**
  - **…**

Many Organizations Claim to Develop High Quality, Reliable Software

- **But how many of them have defined what they mean by "High Quality"?**
- **How many of them can measure the quality of their software?**
- **How many of them can evaluate whether their software has achieved "High Quality"?**
- **How many of them know how to engineer high quality into their software?**

What Do We Mean When We Talk About "High Quality Software"?

- 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
- …

Do we know how to achieve these characteristics?

Can we test for these characteristics?

Can we measure them?

Measurement is Often Involved in How We Test or Evaluate 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 be reliable	➢ Count the faults during operation? ➢ Count something during development?
▪ 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 or how well the defects are found

But What Are We Testing or Evaluating? 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

There are many known methods of achieving these. And much in common

Testing is one way to assess software quality.

And measurement is often part of testing.

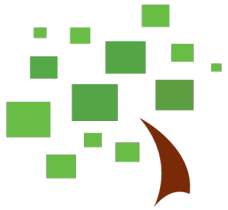
Test and Evaluation

***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.



SWEBOK® V3.0

Guide to the Software Engineering Body of Knowledge

Editors

Pierre Bourque
Richard E. (Dick) Fairley

IEEE computer society

Downloadable at:
www.swebok.org

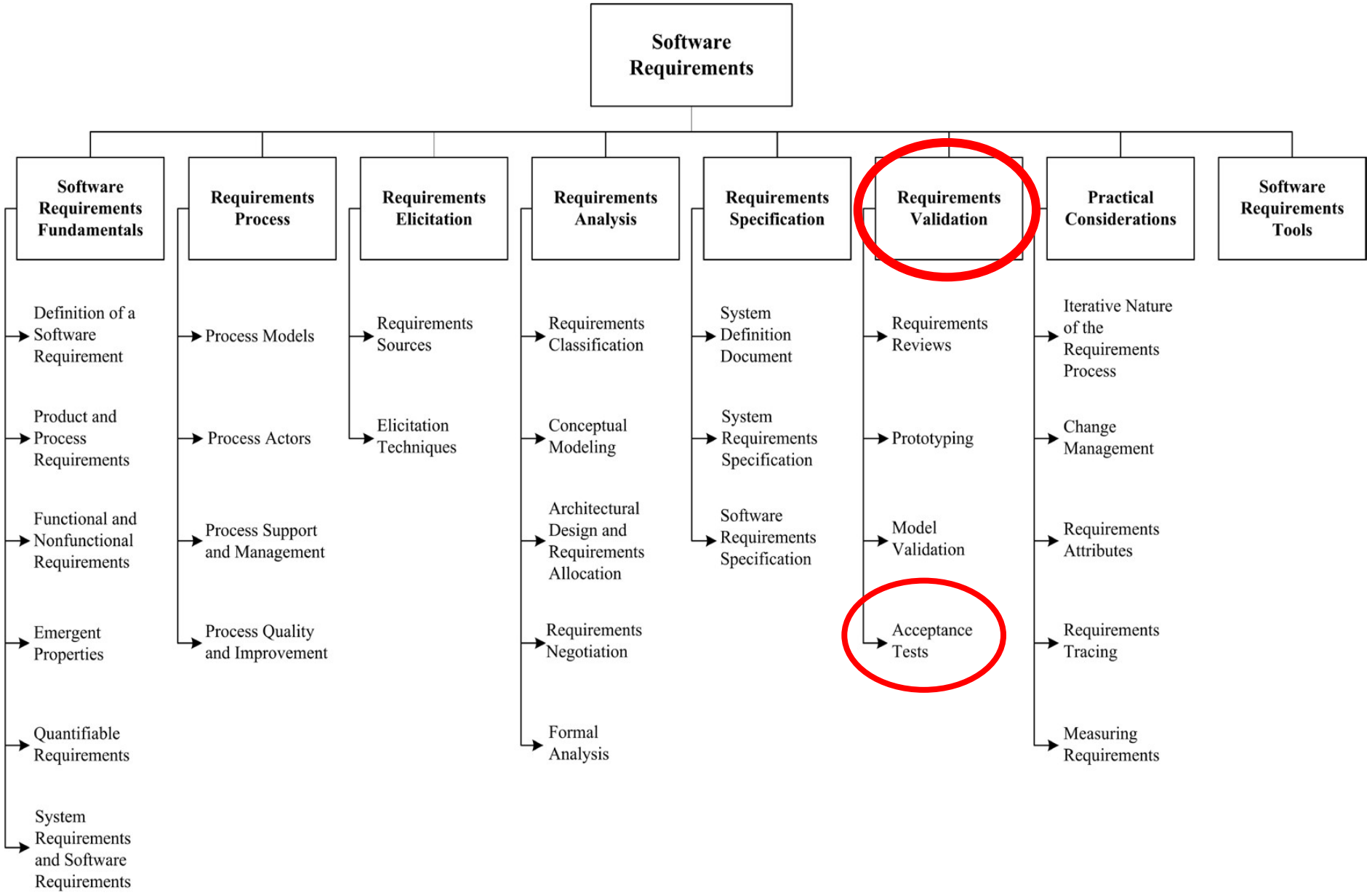
SWEBOK Facts

- **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**

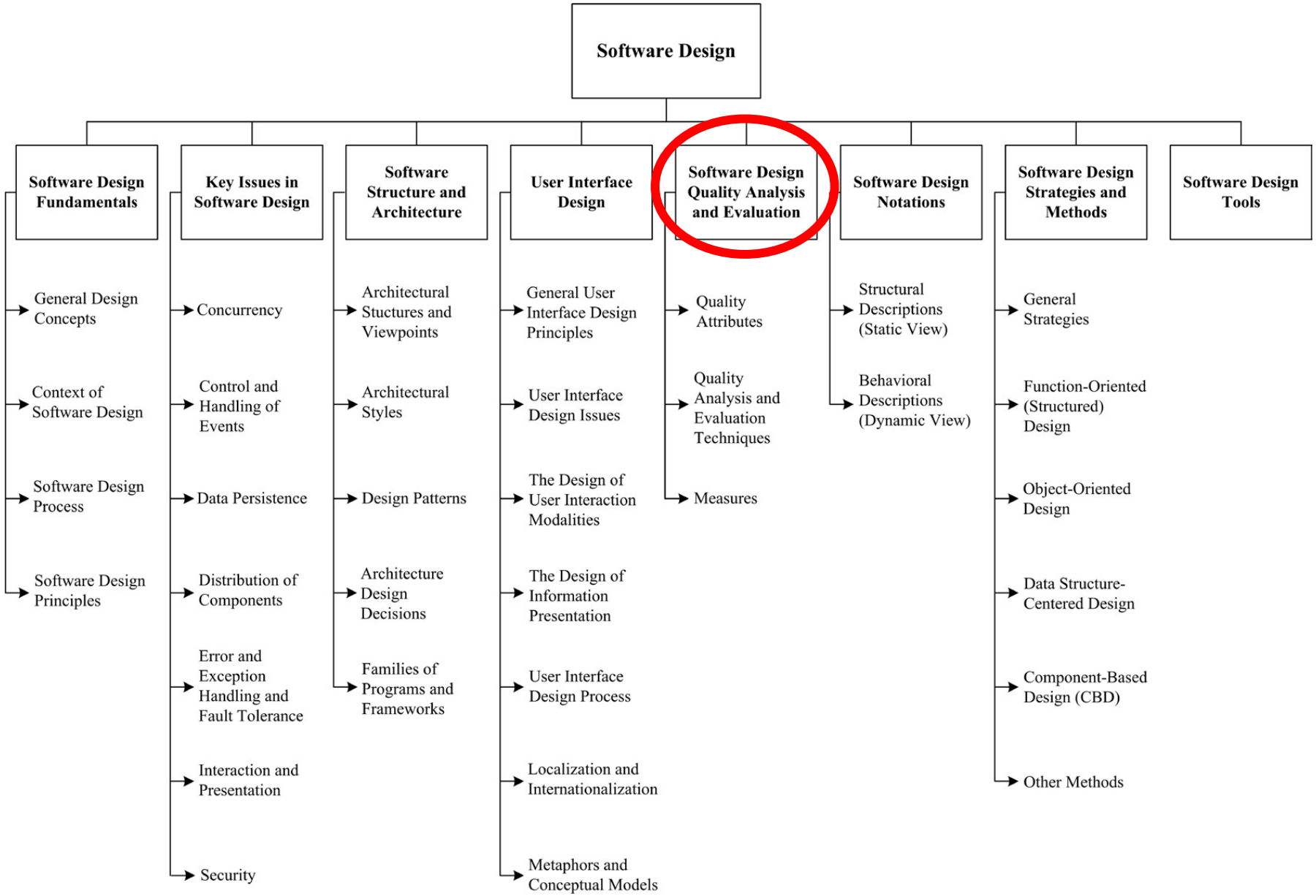
The 15 SWEBOK Knowledge Areas

- 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

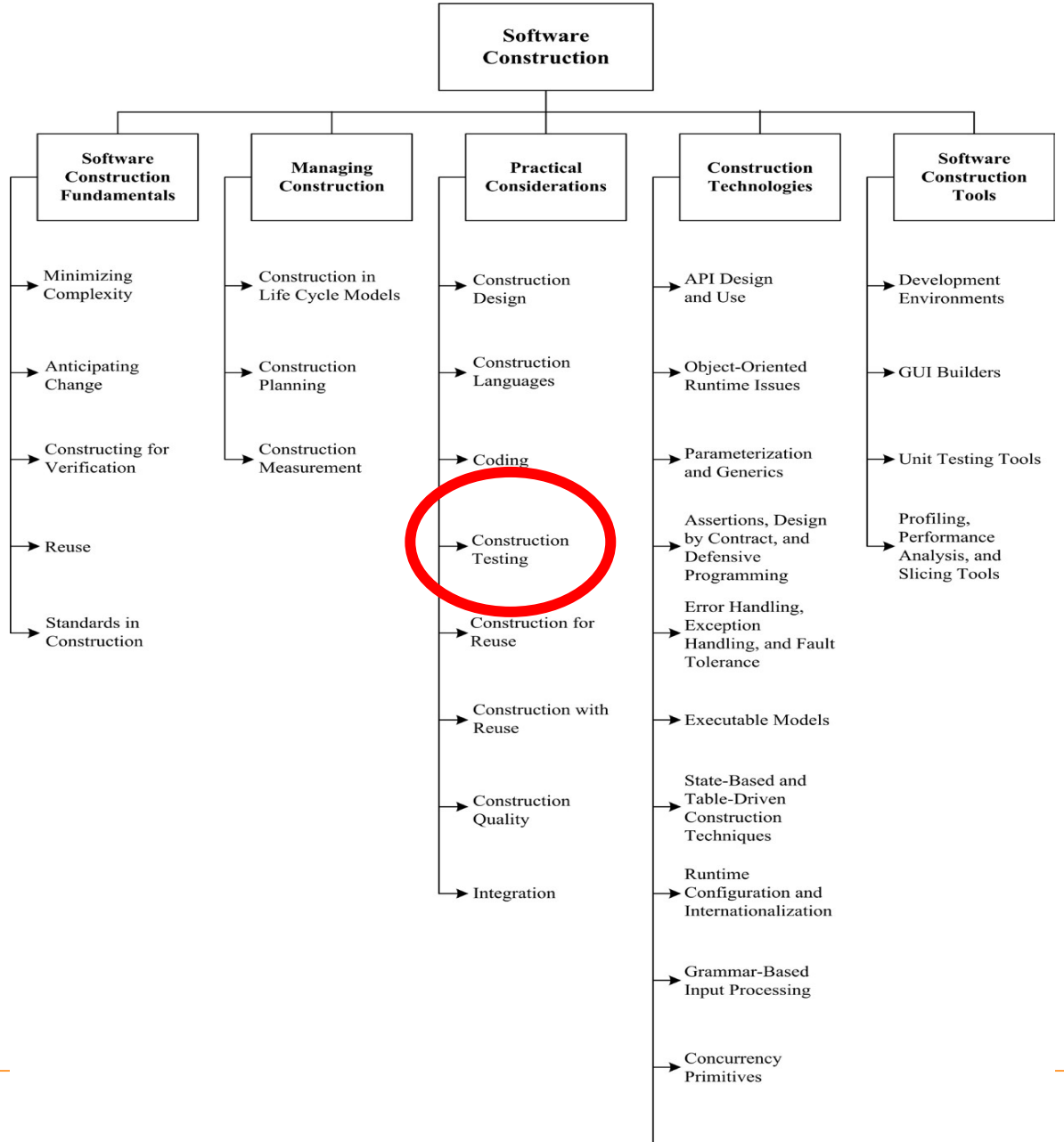
Software Requirements



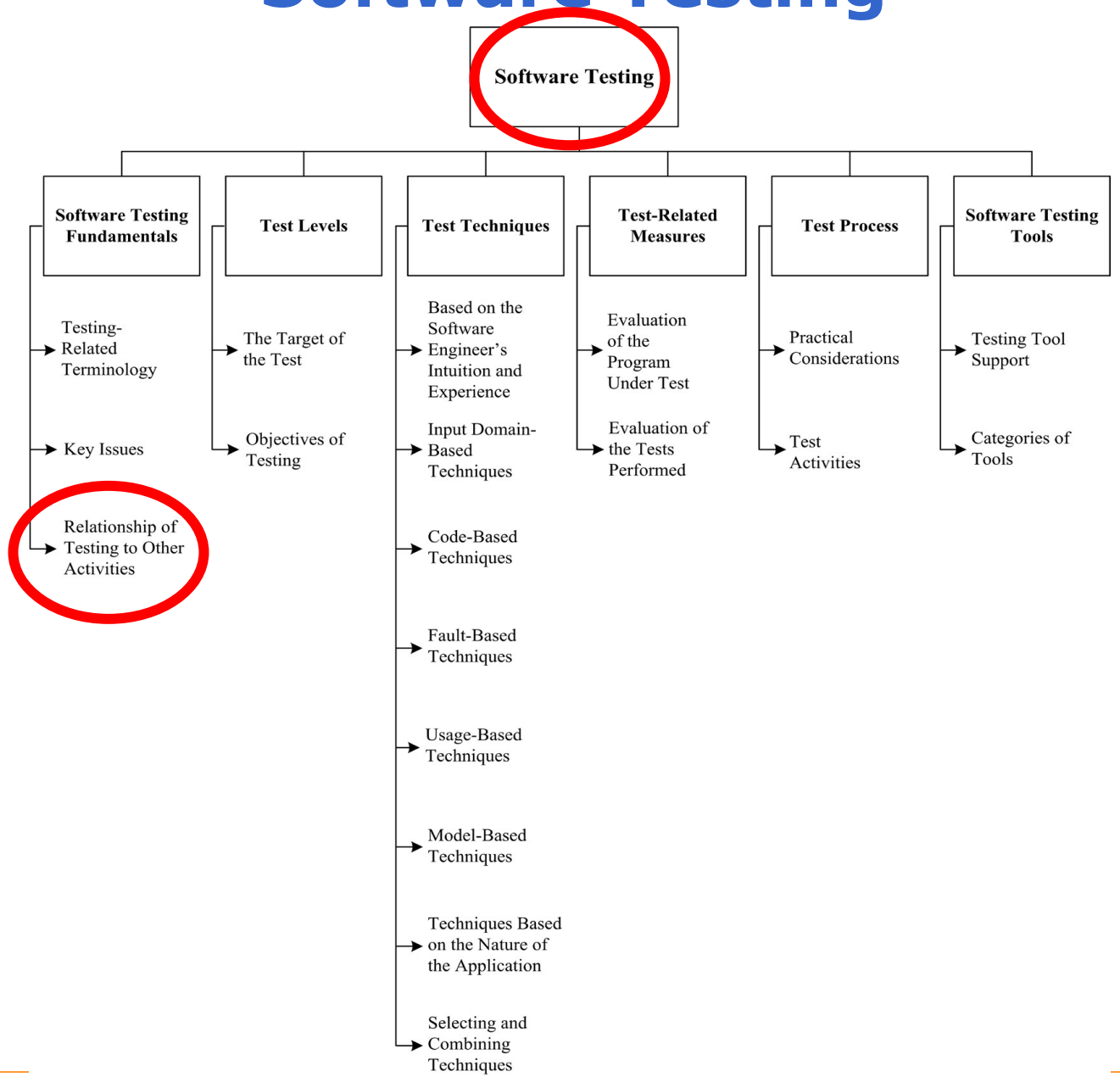
Software Design



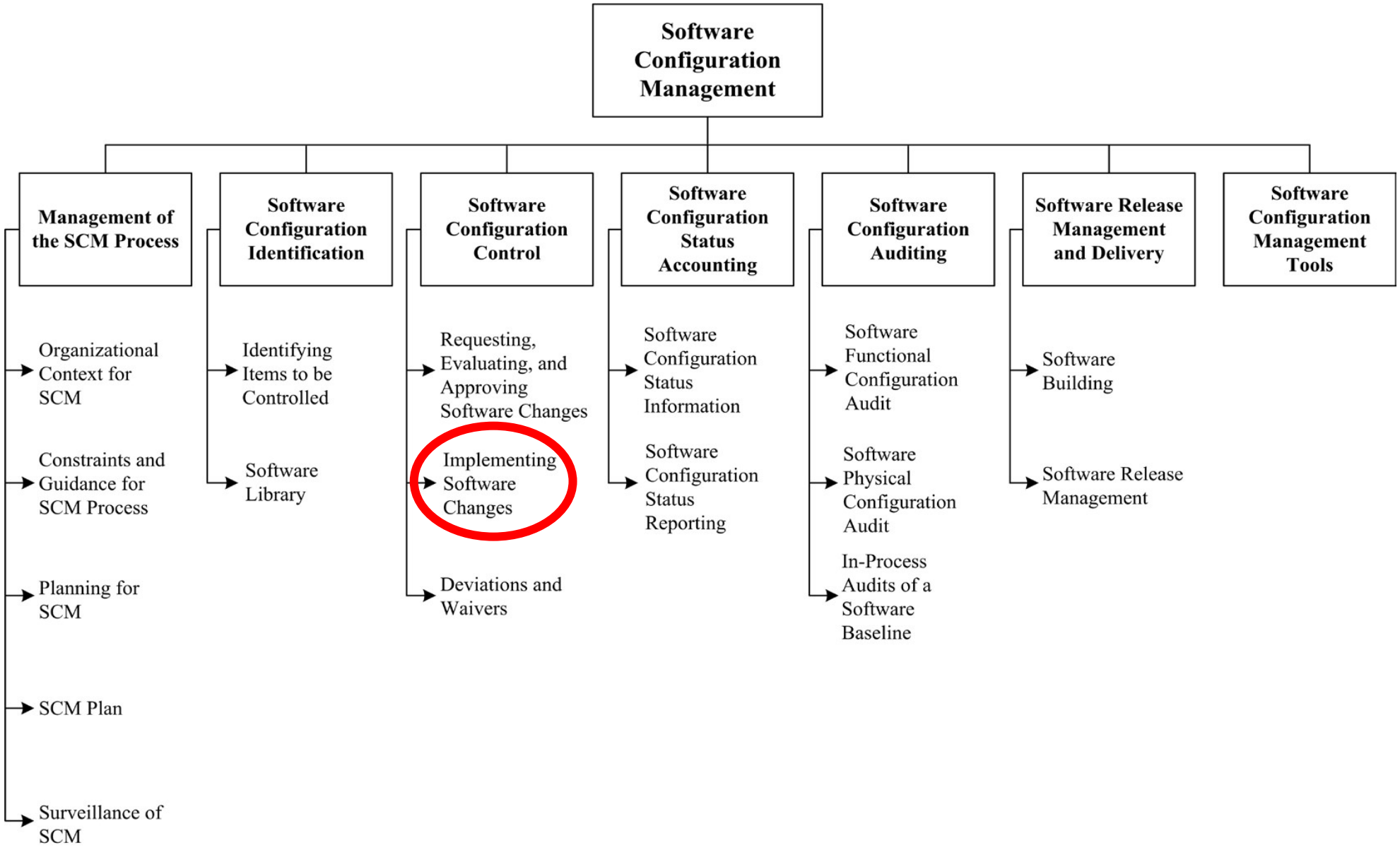
Software Construction



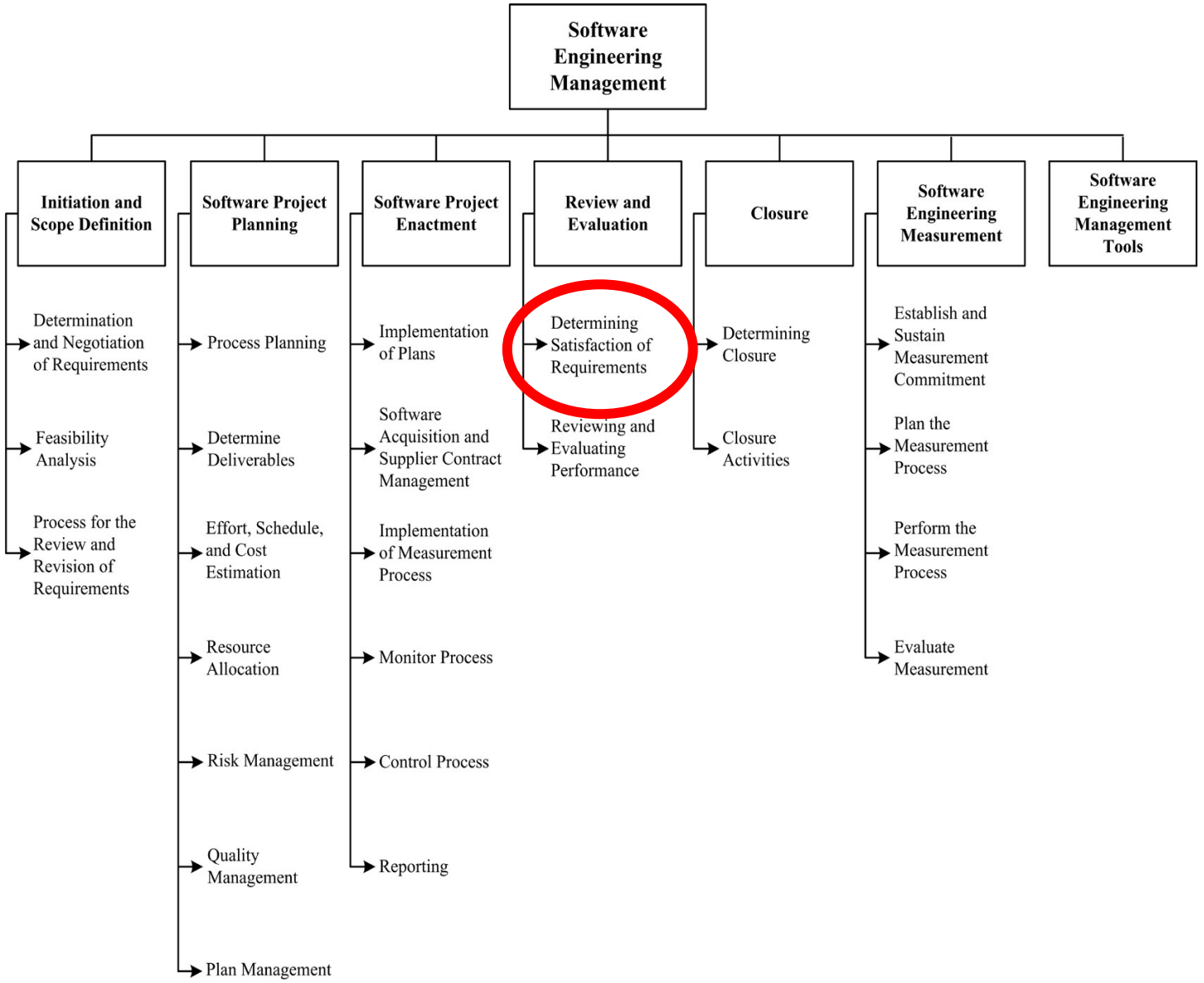
Software Testing



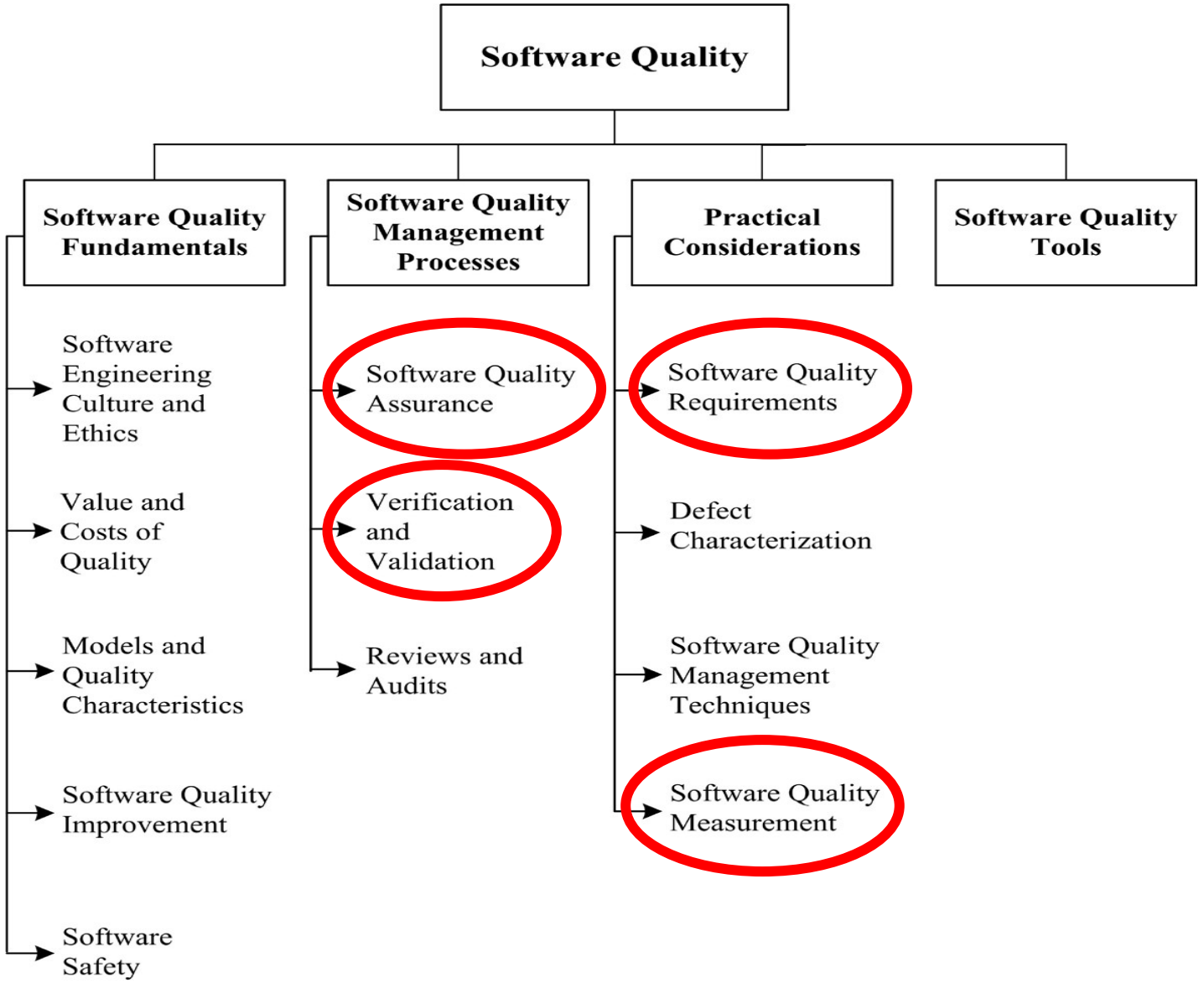
Software Configuration Management



Software Engineering Management



Software Quality



Part 1 - Outline

- The Scope of Software Quality
- **Defining Quality**
- Observations on the Testing Process

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 5)

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

What the end user wants

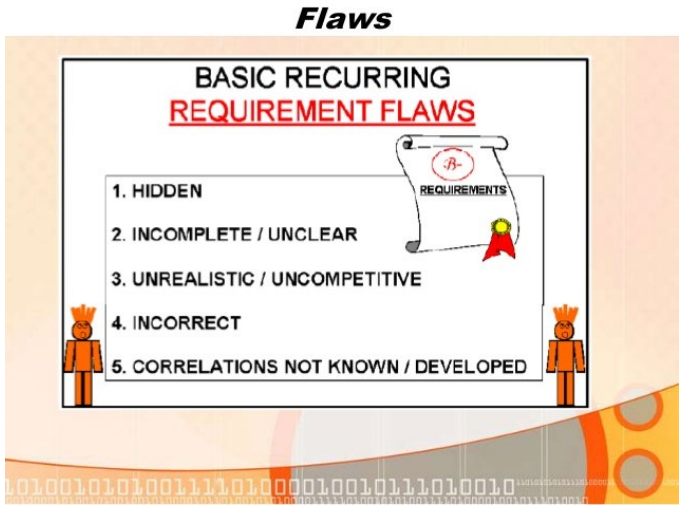
Planetgeek.ch

What the engineer builds

Issues with "Conformance to Requirements" (2 of 5)

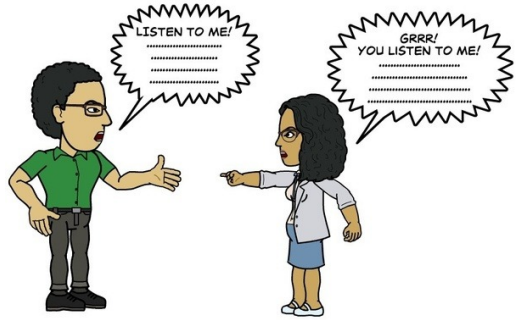
Are the requirements right?

- 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 5)

Is it even possible to define the requirements in a measurable and testable way?

- ***Requirement***: software must be reliable
- ***What does this actually mean?***
  - **Doesn't fail very often? --- *How often is too often?***
  - **Failures do not cause severe problems?**
  - **I know it when I see it?**
  - **…**

Issues with "Conformance to Requirements" (4 of 5)

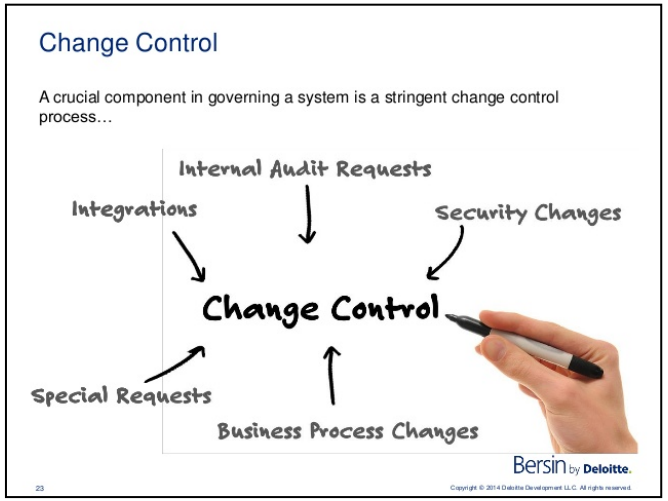
What about implicit vs. explicit requirements?

- ***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" (5 of 5)

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*?

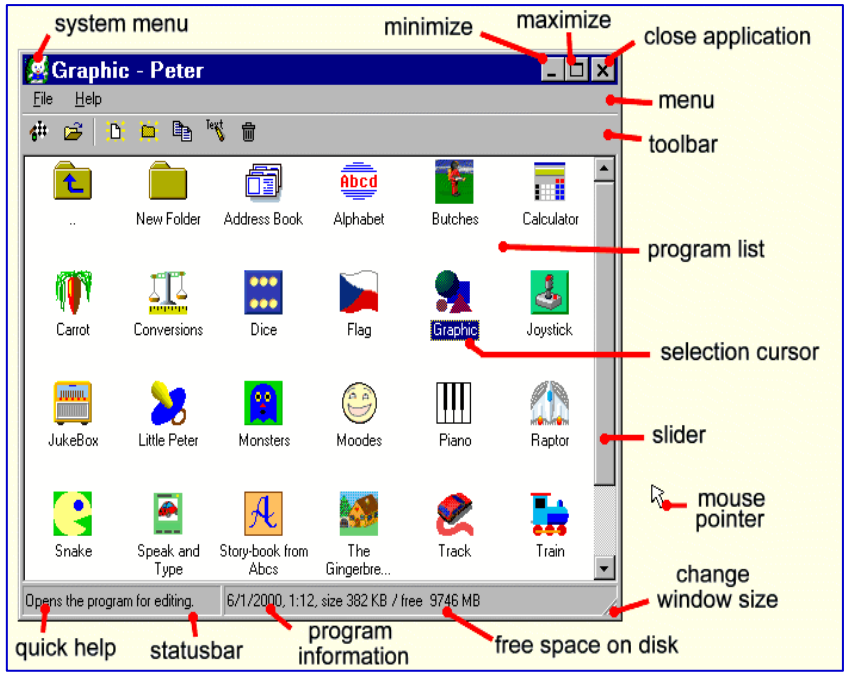
- Consider a TV set
  - which fitness characteristics are not understood by
    - Typical User
    - Engineer
    - Sales Personnel

Konga.com

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

Who defines ***software*** ***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)

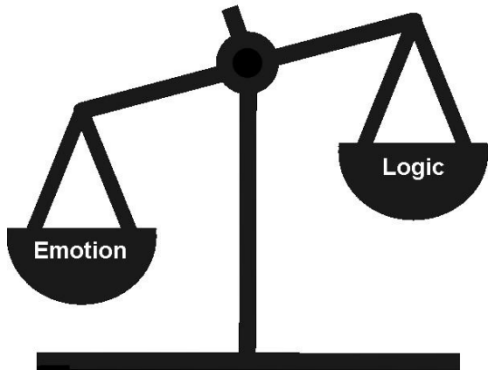
Whose opinion counts?

- You may need to weigh different opinions

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

Logic vs Emotion

- "Glitz" v. "Substance"

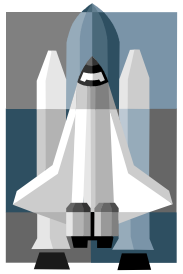


Which Car is Best for Our Family?

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

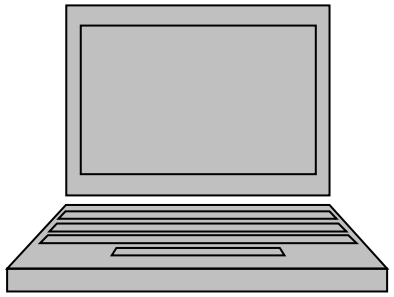
Value depends on What Features are Most Important

- Space Shuttle
  - 0 defects
  - Reliability



- Video Game
  - Good user interface
  - High performance

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

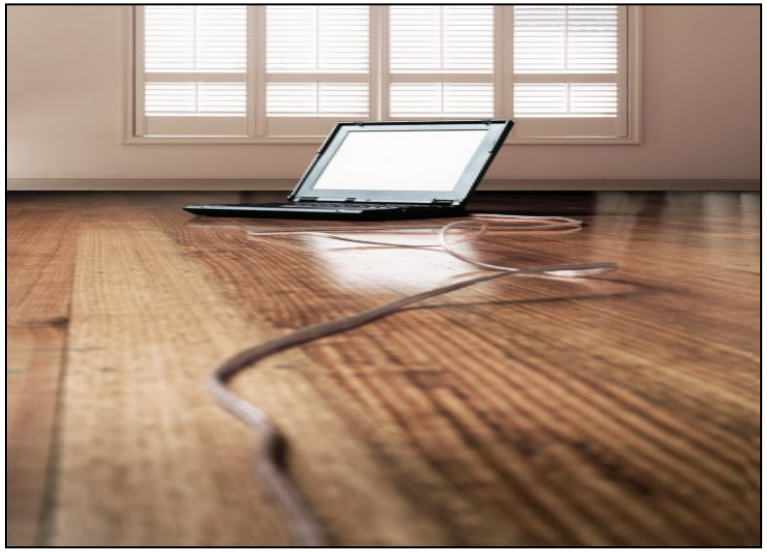


Issues with "Value to Someone" (4 of 4)

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***

Part 1 - Outline

- The Scope of Software Quality
- Defining Quality
- **Observations on the 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**

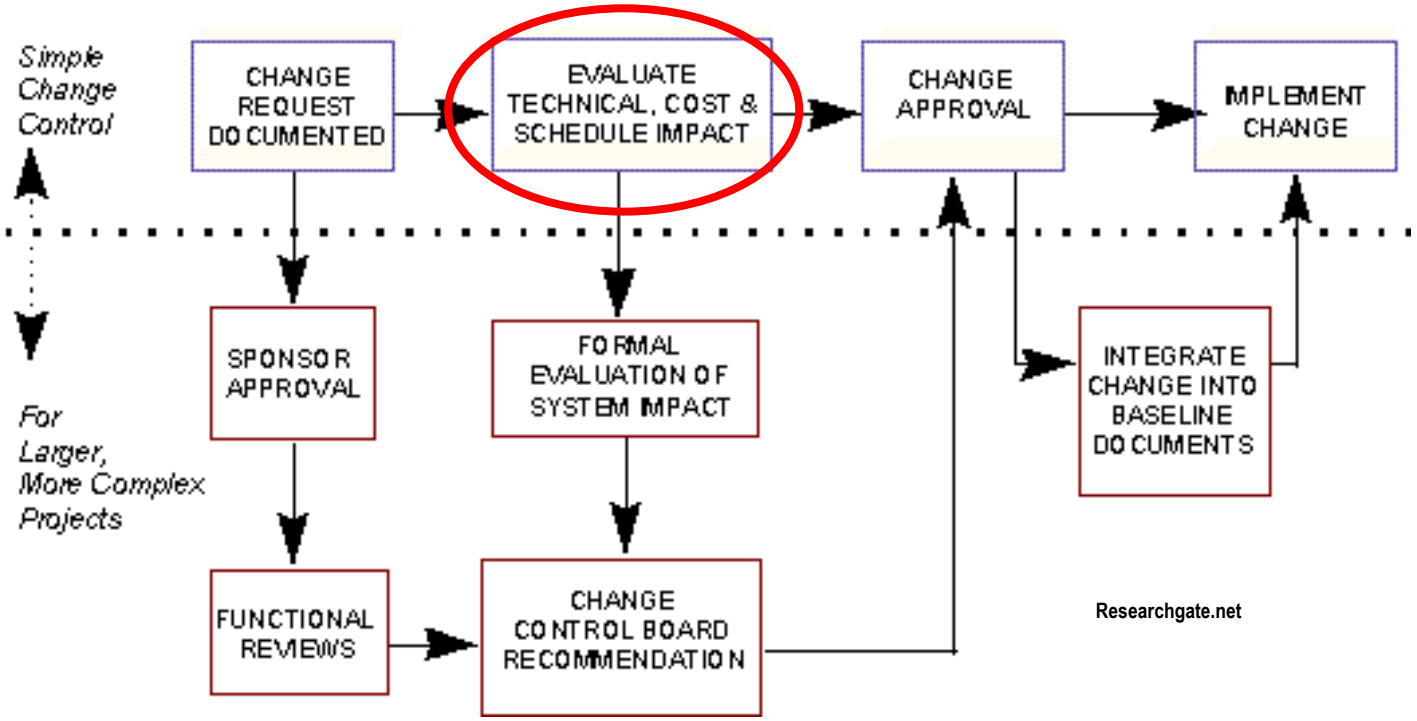
Suggestions (slide 1 of 3)

- **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

www.cigniti.com

Suggestions (slide 2 of 3)

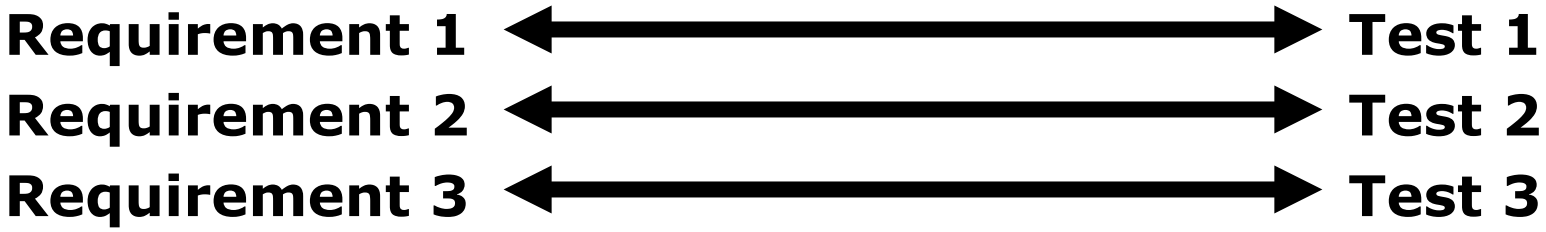
- **Control changes to requirements and design**
  - Don't allow a requirements or design change without a clear understanding of the effect of the change on the software cost, schedule and technical development
  - For each change to requirements or design, indicate how the corresponding tests must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



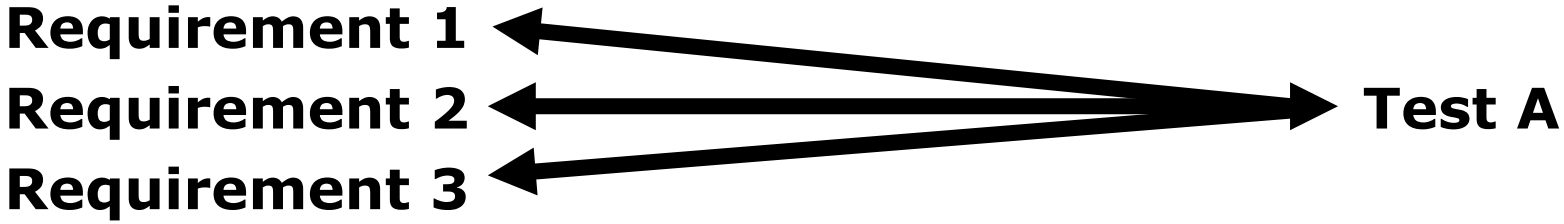
Suggestions (slide 3 of 3)

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

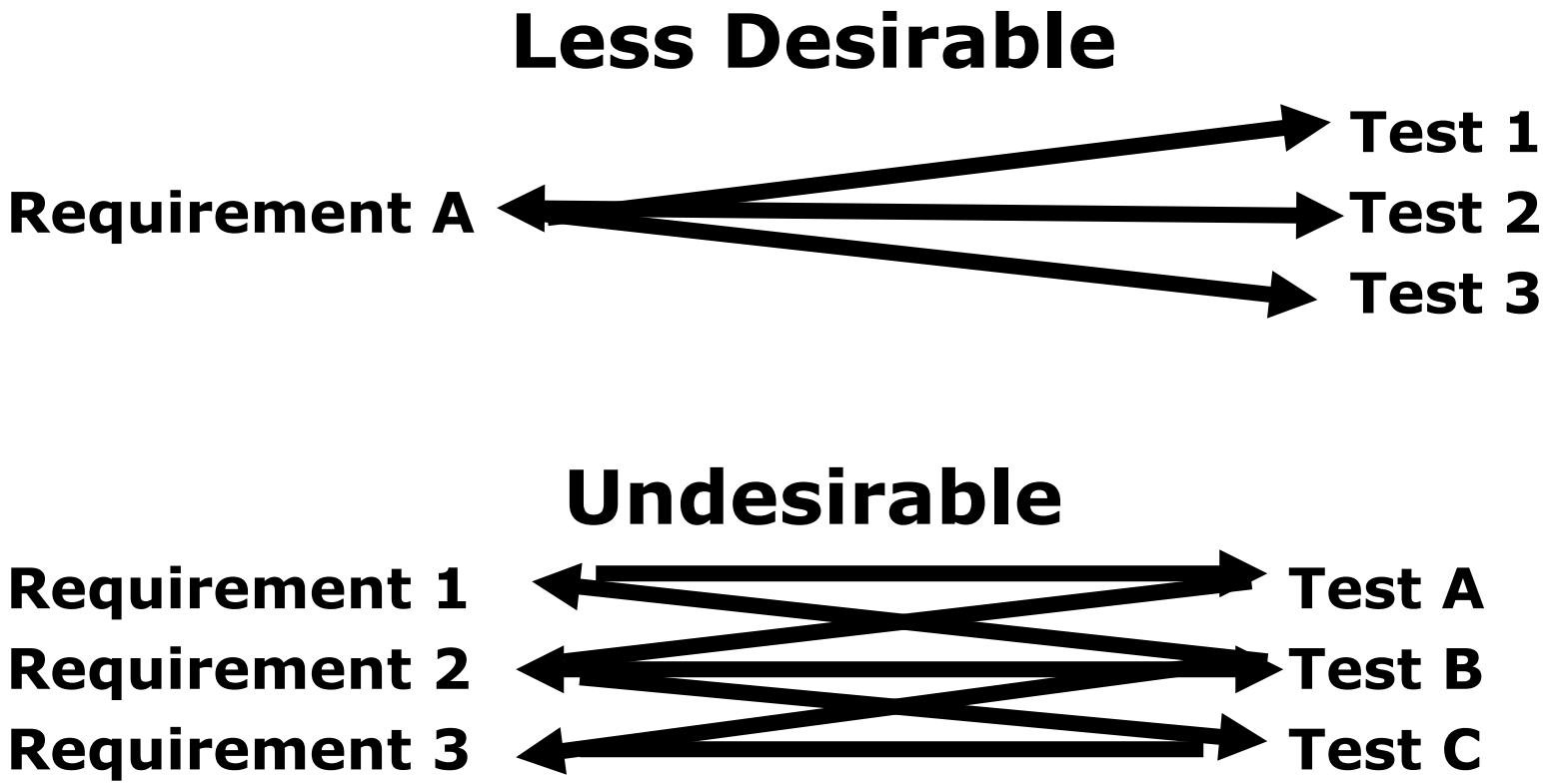
Ideal



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 part 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**
- **Re-running 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

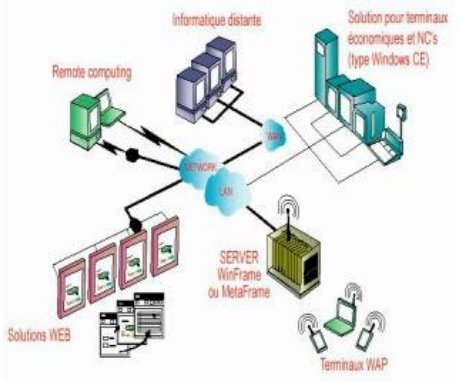
Failure to re-run tests is a major source of software problems

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
- …

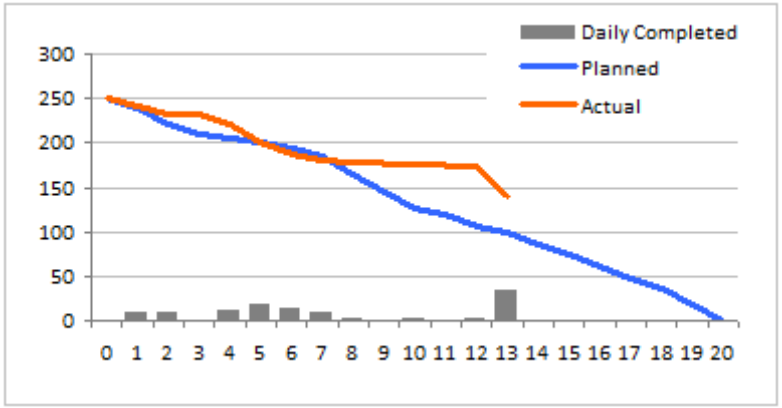


Resources usually cost money

- 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

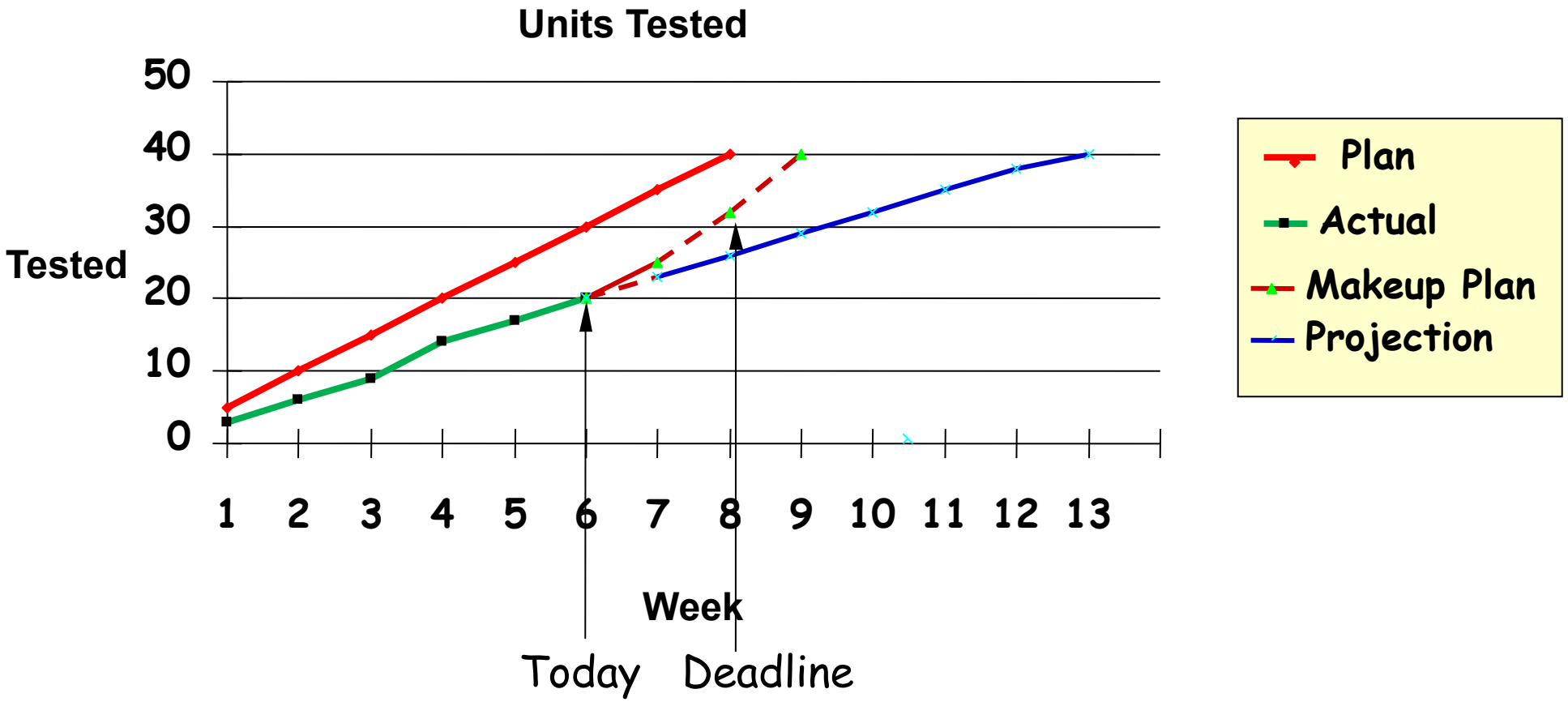
- **This can lead to *problems if people are not measured fairly***
  - People are very sensitive to fairness of measurements
- ***Productivity* of people is an *especially problematic* thing to measure**
  - **The person doing the hardest job or the most thorough job tends to look like they are making 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, especially if rushed to meet deadlines.
  - 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 – Achieving Software Quality

Part 3 - Defect Containment

Part 4 – Measuring Software Complexity

Part 2 - Outline

- **Introduction**
  - Six Sigma Overview
  - Value Added Analysis Overview
  - Cost of Quality Analysis Overview

Quality Emcompasses Many Characteristics

There are Many Well Established Techniques for Defining, Improving, Measuring, Testing, Predicting and Managing Quality

The Good News

Most techniques for improving quality apply to multiple quality characteristics

- **For example, *reducing rework* tends to**
  - Reduce defects,
  - Improve reliability,
  - Reduce development costs,
  - Reduce likelihood of doing harm,
  - Reduce maintenance costs, and
  - Improve customer satistaction.
- ***Defect containment also* enables you to**
  - Predict reliability, warranty cost, etc.
  - Identify the process steps that produce the worst defects
  - Identify development practices that help or hurt reliability and quality

A truly effective reliability program should be integrated with a truly comprehensive quality engineering program.

The Bad News

Failure to apply a comprehensive quality improvement approach may result in quality problems, even in "perfect" software.

- **A "perfect" software product may have these quality problems:**
  - The customer doesn't like it
  - Hard to use
  - Not compatible with other software or systems
  - Difficult to maintain
  - Satisfies requirements but doesn't actually solve the customer's problem very effectively
  - …

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

[1] See Adams in reference list.

Issues on Real Projects That Have Resulted in *More Comprehensive Quality Engineering Approaches*

- **What is needed for some products is often *not needed for others***
  - You need to know what the customer requires
- **Those doing the work must believe the goal is realistic**
  - Otherwise, they see it as pushing them to perform beyond their capacity for marginal benefit
- **Many of the problems have to do with management decisions or corporate culture**
  - You can't expect technical experts to solve the problems alone – they need support and commitments

How Good Do You Have to Be?

How Many Products or Services Must Be Defect Free?

- **99%**
  - This would mean 1 error per 100 course slides which is probably fairly typical
  - But --- 200,000 wrong drug prescriptions per year - very bad
- **99.9%**
  - 1 spelling error per page in a book or student paper – fairly good
  - But 500 surgical errors per week – not acceptable
- **99.99%**
  - 2000 mail delivery errors per hour

In other words, it depends on the product or service!

What About Variance?

Average product vs **worst case product**

Which product is better?

Product	Average	Worst Case
A	2 defects	45 defects
B	3 defects	7 defects

Many methods of improving quality and reliability focus on average rather than worst case scenarios.

Other Limitations of Many Quality Improvement Programs

- ***No insight* into the nature of the problems?**
  - We may measure or predict failure rates but learn little about the ***causes*** of the failures or ***how to cure them***
- ***Little consensus* on what to do**
  - Although different quality and reliability experts have recommended ways to improve
    - they may ***not be aligned or compatible***
    - there is usually ***no overall conceptual framework*** or comprehensive theoretical model
- **How do you *justify the costs*?**
  - How do you balance costs and benefits

Part 2 - Outline

- Introduction
- **Six Sigma Overview**
- Value Added Analysis Overview
- Cost of Quality Analysis Overview

Six Sigma Origins

Six Sigma is a *comprehensive and integrated* set of tools and techniques introduced by Motorola Corporation in the mid-1980's

1985 Bill Smith coins the term "Six sigma"

1987 Motorola trademarks the term "Six Sigma"

- **Goal:**
  - ***Improve quality***
- **Methods:**
  - ***Integrate many different methods***
  - ***Define a uniform way of measuring quality***
  - ***Remove the causes of defects*, and**
  - ***Minimize variability*** in product development and business processes

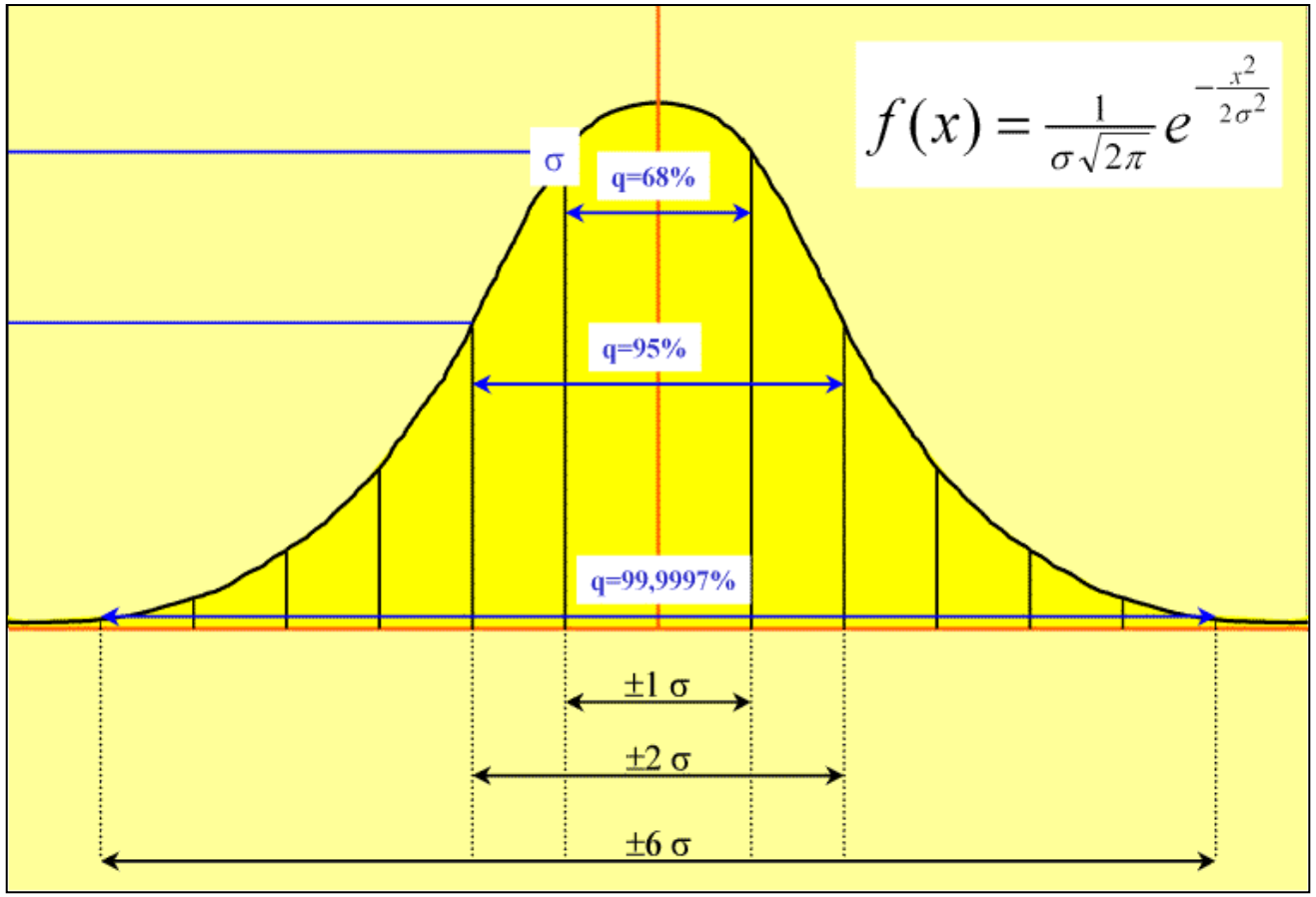
Six sigma incorporates the best ideas of Juran, Deming, Crosby and other quality experts into a comprehensive approach.

Six Sigma is a Comprehensive Program

- **Six Sigma requires *commitment from the entire organization*, especially top management**
  - Management must be **willing to change culture** and **processes** to achieve higher quality
  - Management must ***support the effort***, **even when it means missing deadlines or raising development cost**
- **Six Sigma focuses on *measurement and analysis* of *process characteristics* that impact quality**
  - **Reliance on verifiable data** rather than assumptions and guesswork
- **Six Sigma *reduces process variation* in order to achieve stable and predictable results**

The Origin of the Term "Six Sigma"

It refers to the normal distribution curve
and the standard deviation (a measure of variance)



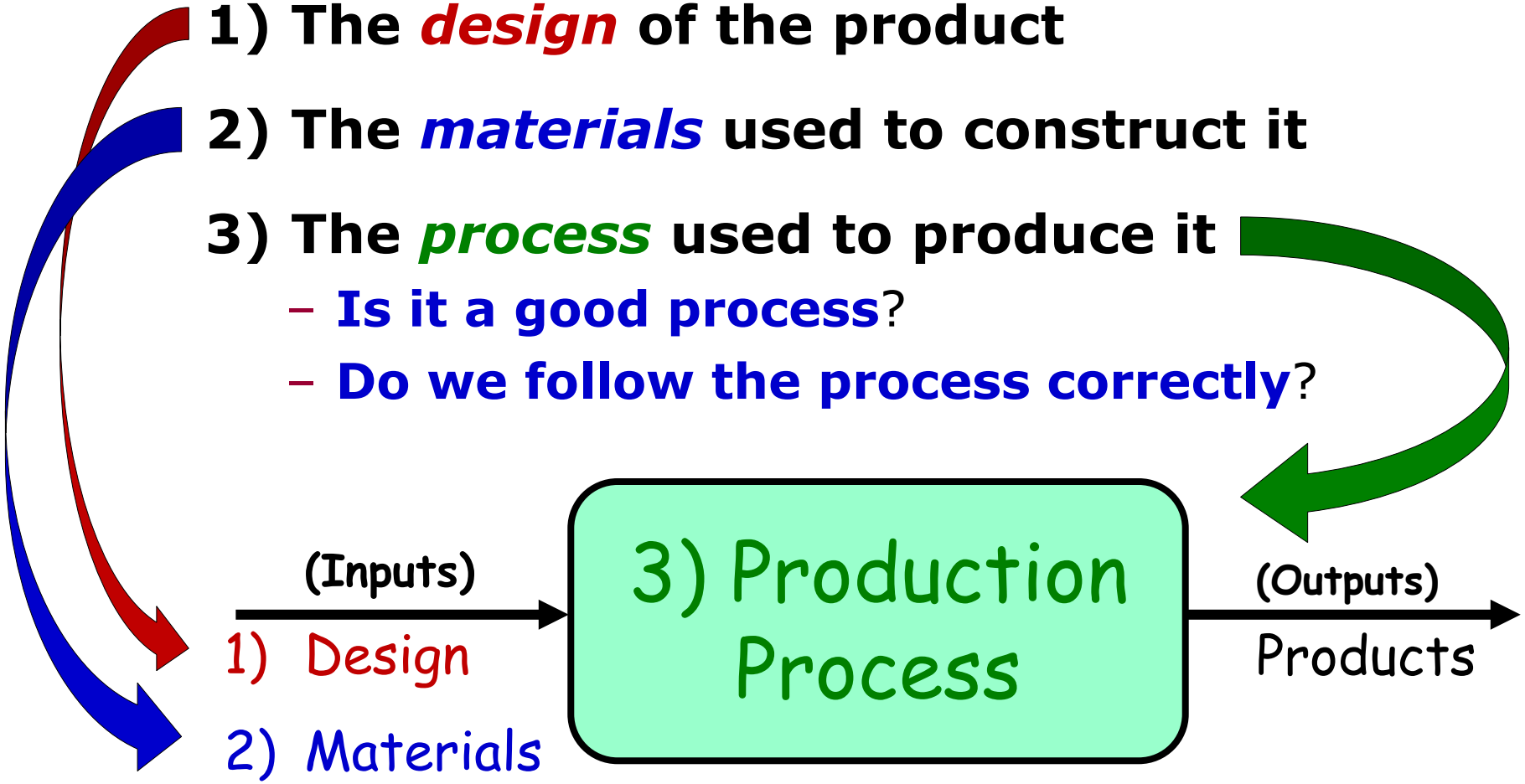
Six sigma attempts to provide a quality improvement approach that is ***achievable*** and where you can have ***a good sense of how much you have achieved***.

Product Quality Depends on Three Things

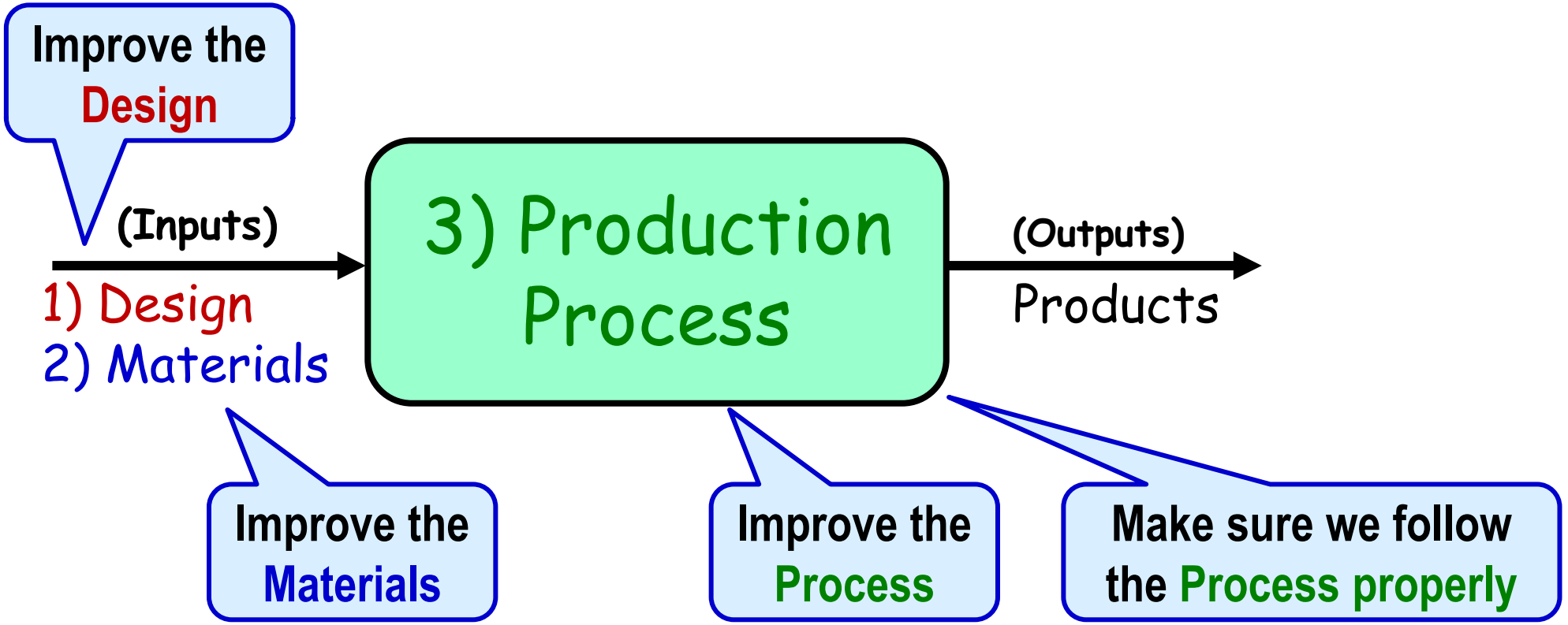
1) The *design* of the product

2) The *materials* used to construct it

3) The *process* used to produce it
   - Is it a good process?
   - Do we follow the process correctly?



To Improve Quality You Must Address All Three Factors



Some Key Elements of Six Sigma Programs

- **You use a *process* to produce something**
- **The *process can vary* as well as the product**
  - So **you must measure and control process variance**
- **Average number of defects is not an acceptable measure . . .**
  - You need to understand the ***worst case*** and why it happens
  - You need to **measure and control the worst case** (not just the average number of defects)
  - You also need to control **variations from day to day**, resulting from incidental factors that are often ignored

Applying to Software

- **Software development produces what is essentially a design!**
  - "***Manufacturing***" of software is a relatively small contributor to quality or reliability problems
- **So the three factors become:**
  - Inputs:
    1) The ***architecture*** of the system, of which software is a part
       - And also the architecture of the software
    2) The ***requirements*** of the software
  - Process:
    3) The ***software development process***

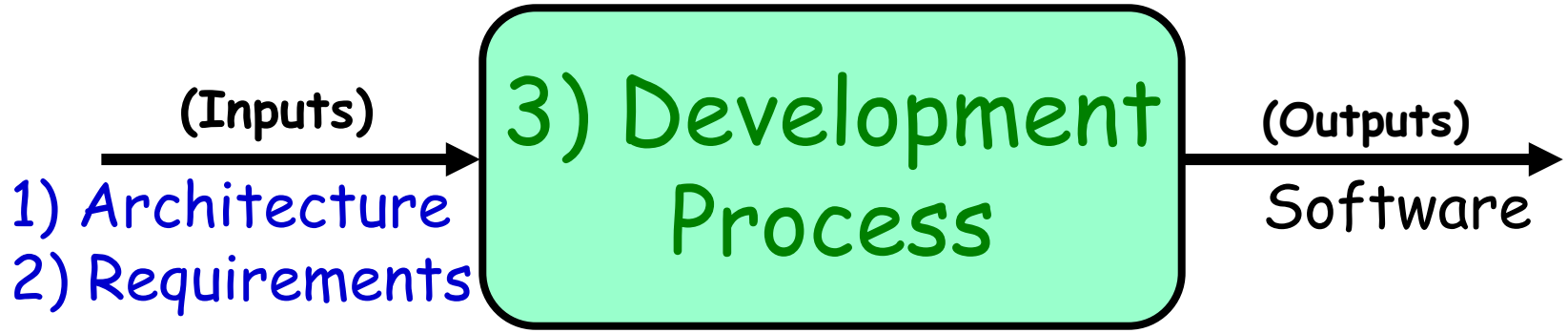
Software Quality Depends on:

1) The *architecture* of the system or product

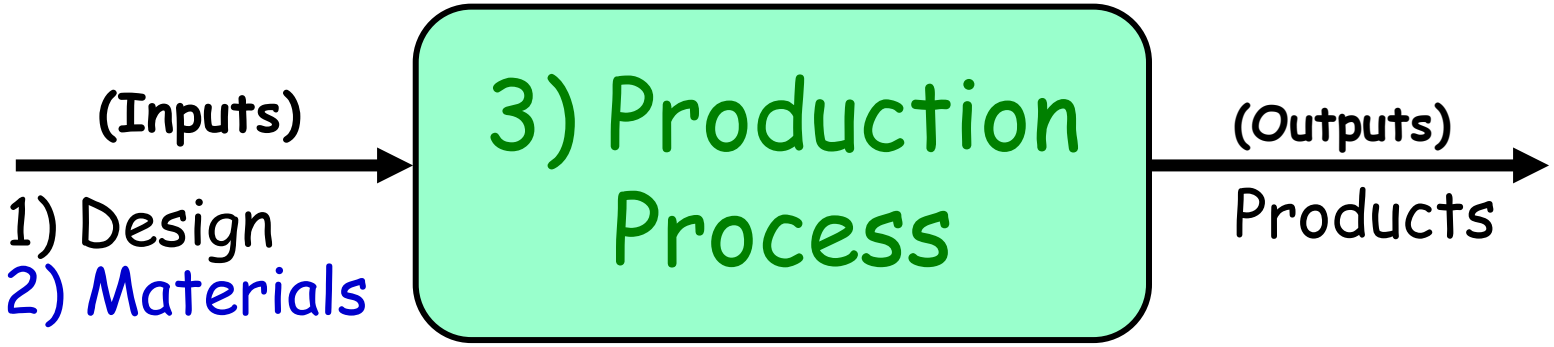
2) The *requirements* of the software

3) The *process* used to develop the software

   - *Is it a good process?*
   - *Do we follow it correctly?*



Achieving 6 Sigma Quality
(for a manufactured product)



1) Design **the product for** ***quality*** **and** ***producibility***

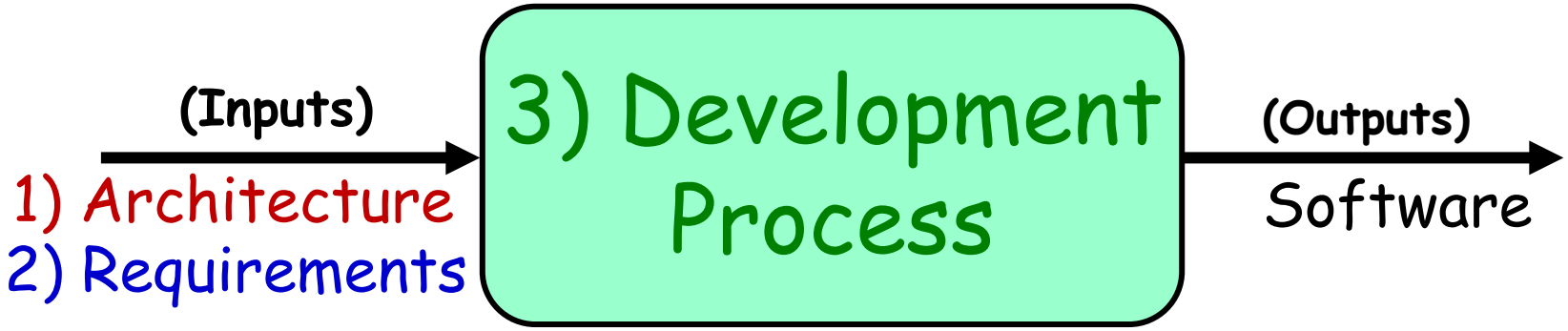
- *this includes improving the design process*

2) Improve the ***quality of the materials***

3) Design the ***production process*** **to produce** ***quality products***

- ***And follow that process correctly***

Achieving 6 Sigma Quality for Software



1) ***Architect*** **the system and the software correctly but with emphasis on ease of development and maintenance**

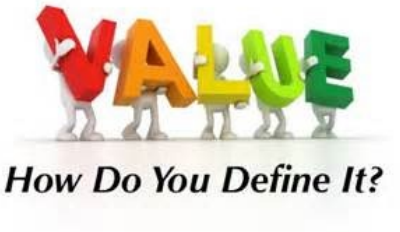
2) **Improve the quality of the *requirements***

3) **Design the *development process* to produce quality software**
   - And ***follow that process correctly*** when developing software

Part 2 - Outline

- Introduction
- Six Sigma Overview
- **Value Added Analysis Overview**
- Cost of Quality Analysis Overview

Defining Value



Correctly defining value is the first step of

customer satisfaction

What Really Matters to the Customer?

Tasks or features that do not directly or indirectly contribute to value are not desirable:

They add cost and risk but do not provide appropriate benefits

Dimensions of Customer Value (and how we achieve them)

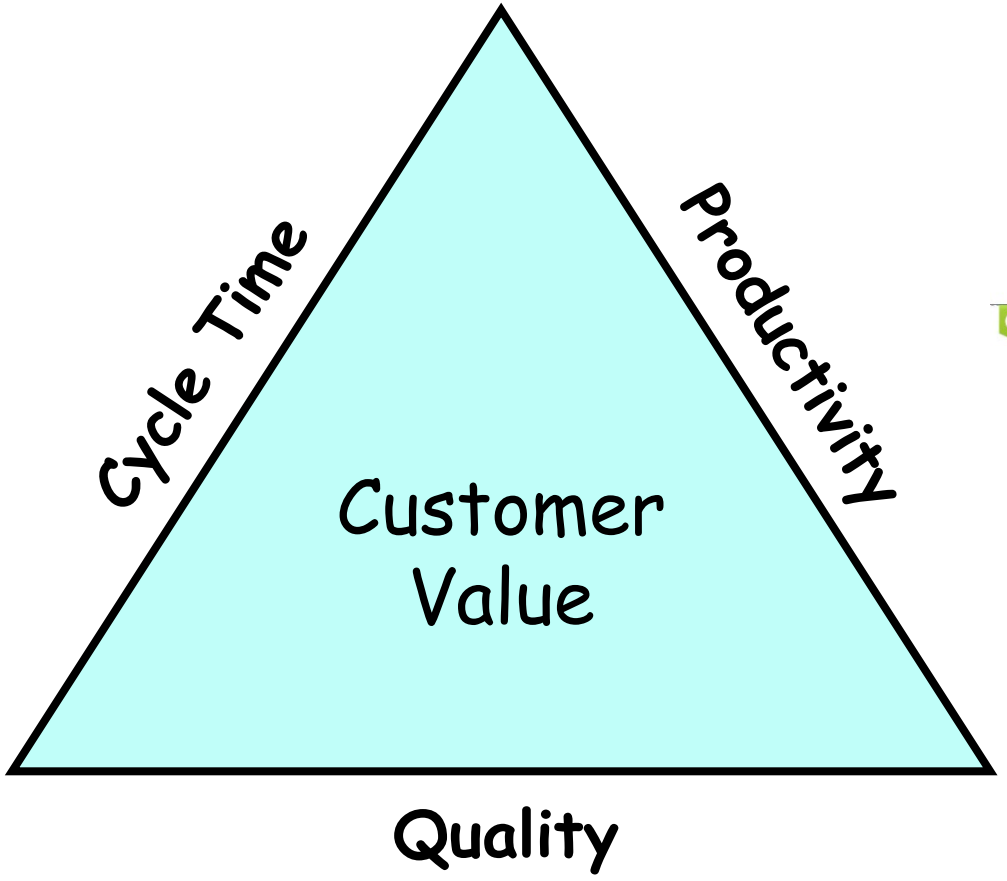
- **Low Costs / High Benefit**
  - Product development or manufacturing efficiency
  - Attractive price

- **High Quality**
  - Customer satisfaction
  - Reliability & few defects

- **Short Cycle Time**
  - Rapid product development
  - Rapid response to orders

The Goal

Improve all components of the [customer value triangle]

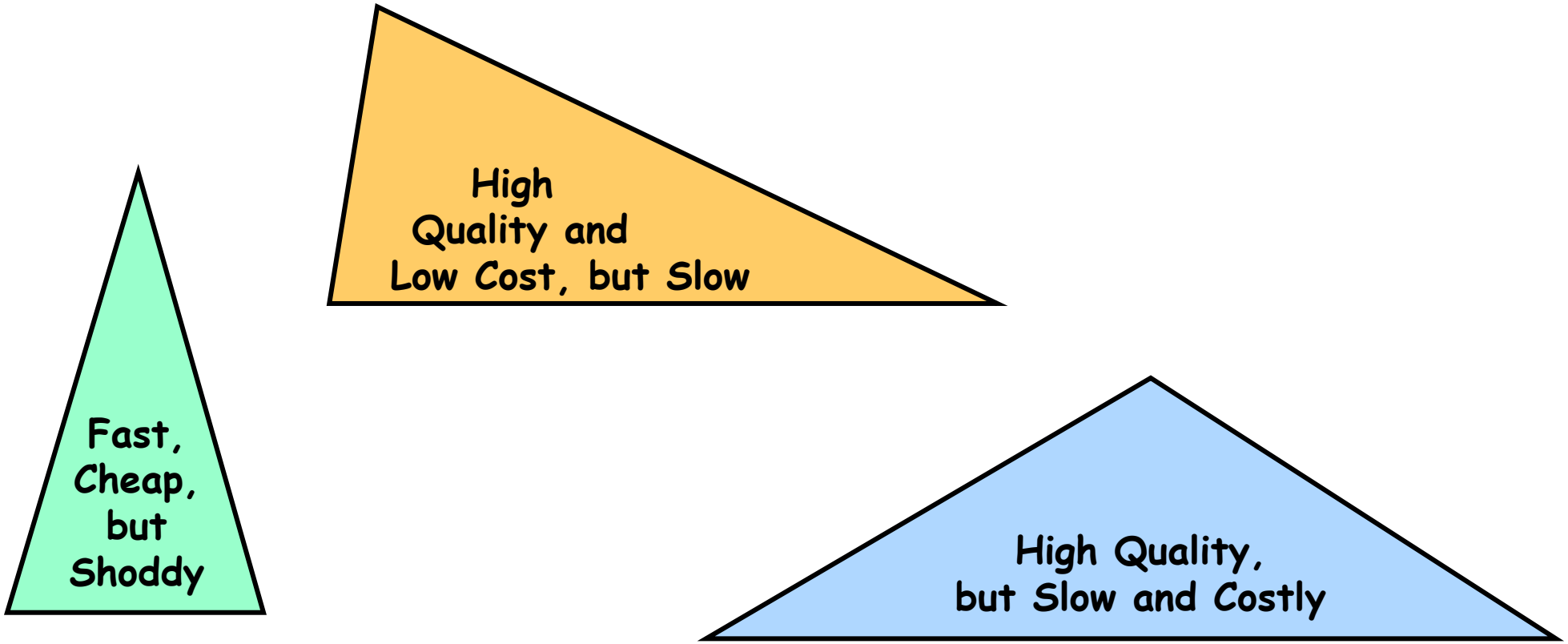


Conventional Thinking

You Can Have Any Two of These

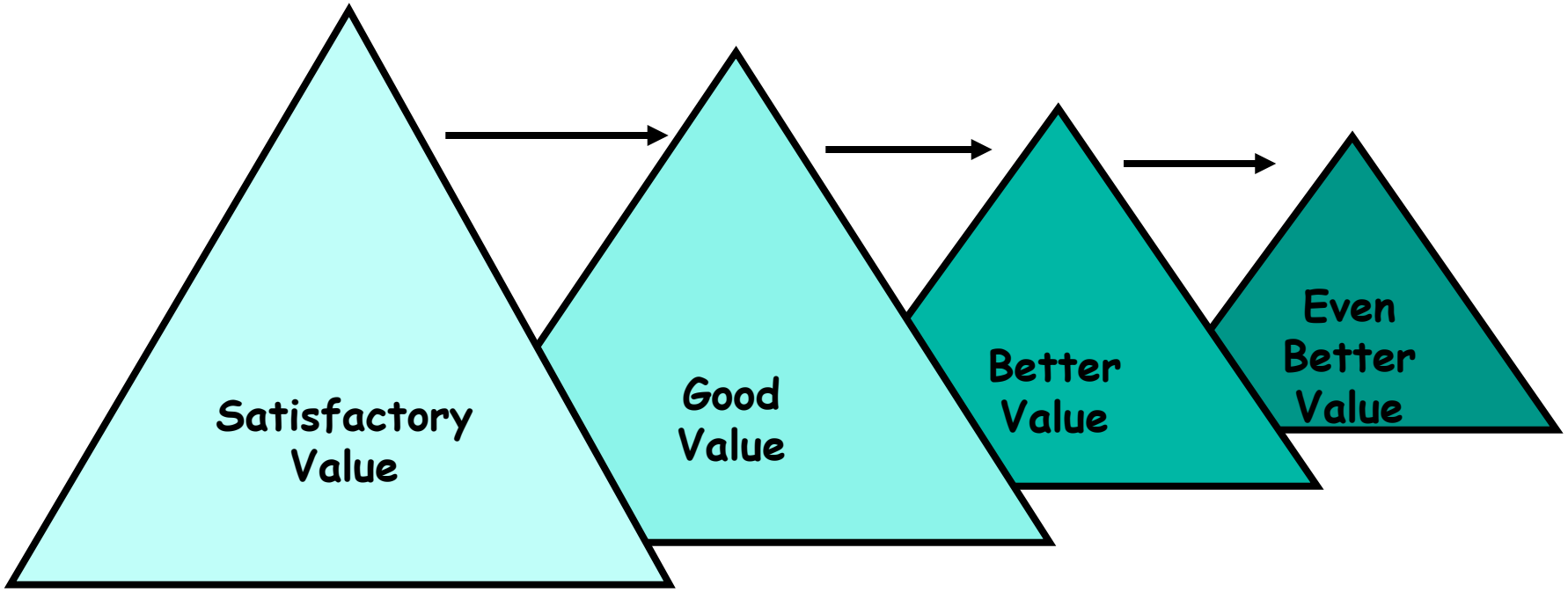
Conventional Thinking ...

You can improve any one at the expense of the others



Modern Thinking

… you can improve all together



How Can We Improve All Three?

By Changing Process & Culture

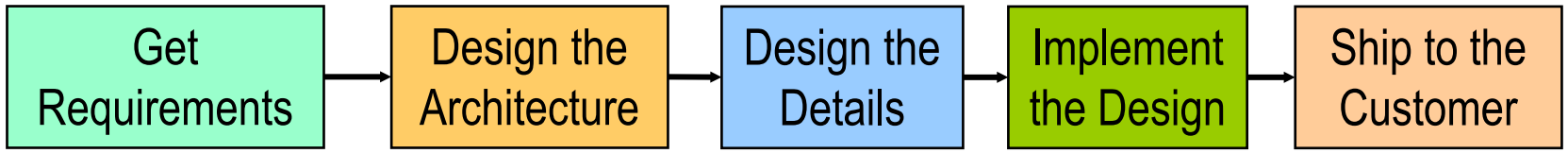
What we do **How we do it** **Why we do it**

Value-Added Analysis

By focusing on
***Where we Add Value*,**
we can
Reduce Cost,
Reduce Defects, and
Reduce Time

The Value Stream

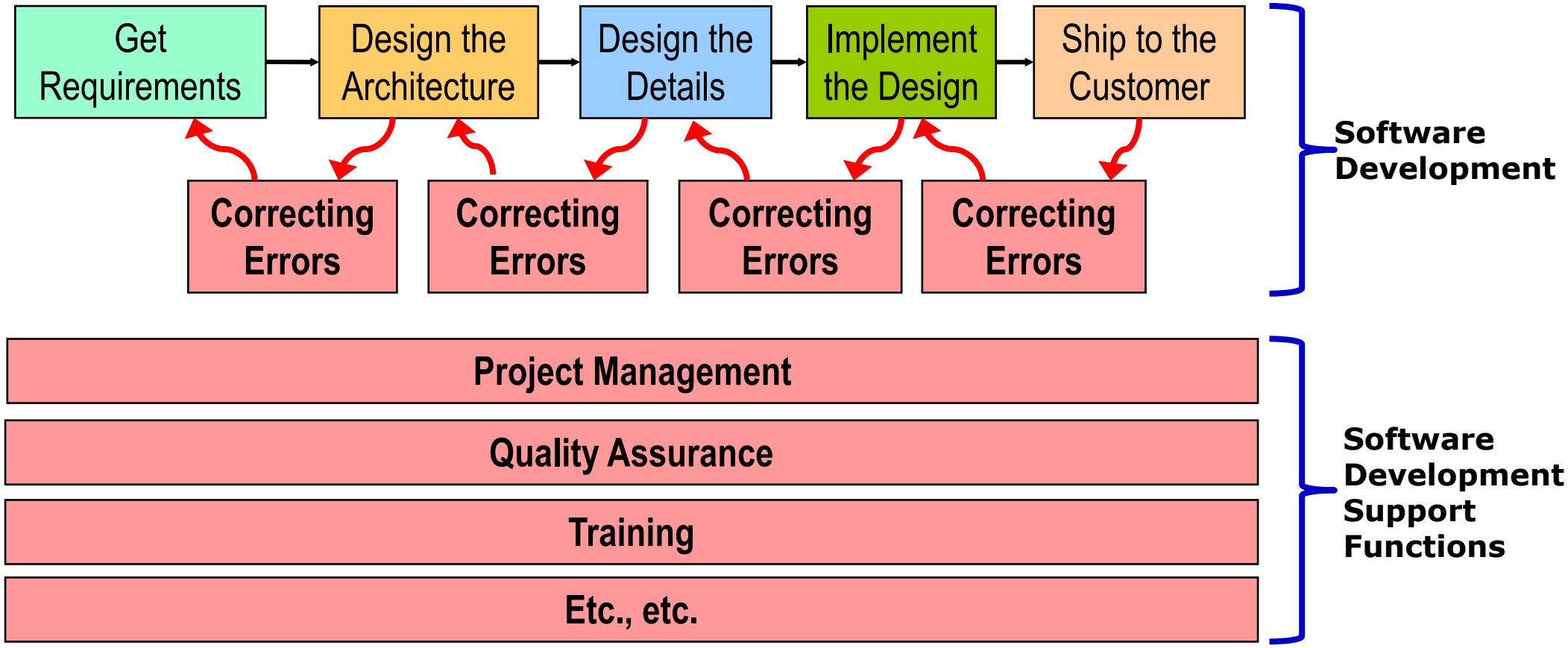
Consider the sequence of steps that add value for the customer



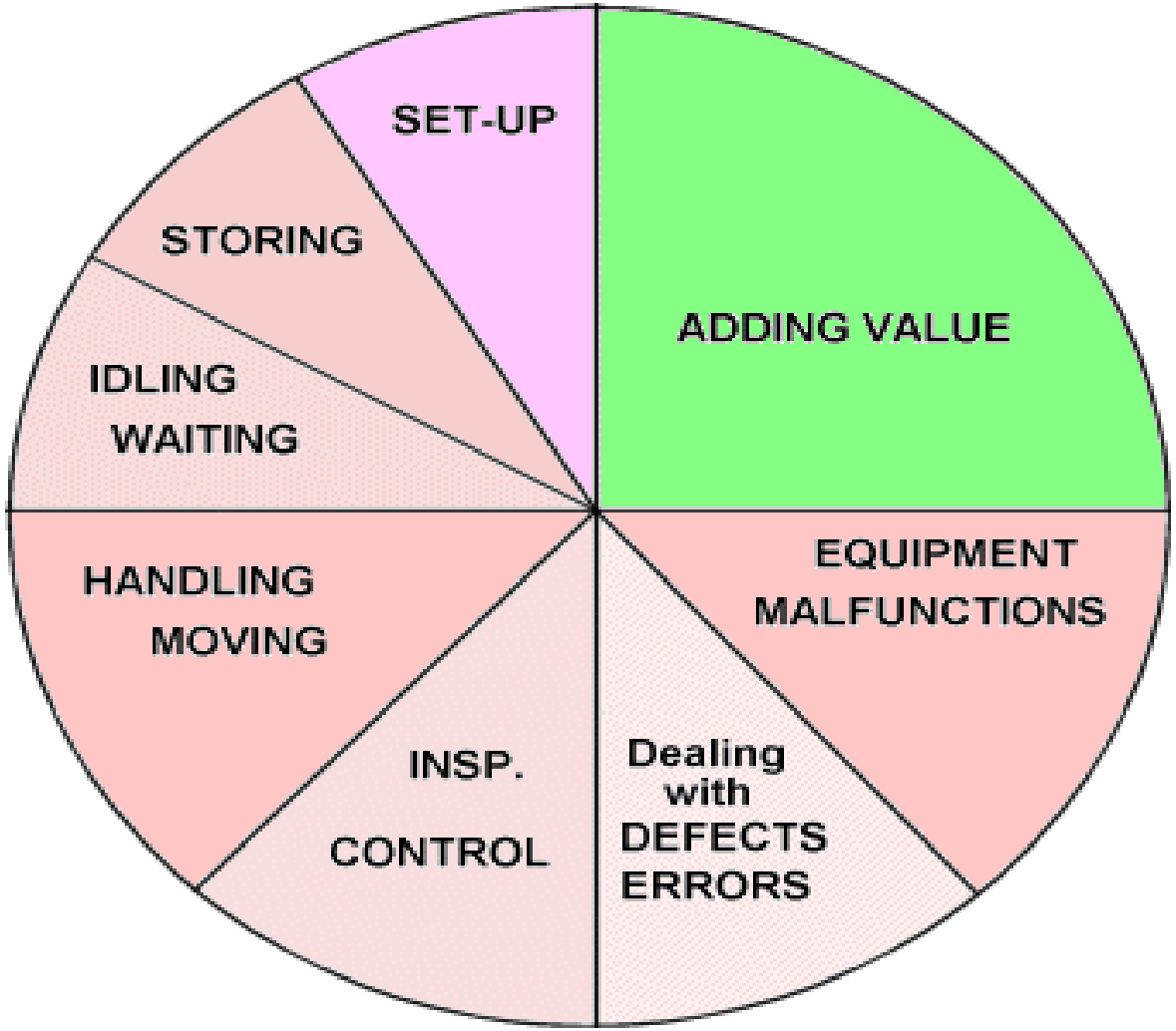
- **These are known as the value-added steps**
- **The complete sequence is called the value stream**

The Non–Value-Added Steps

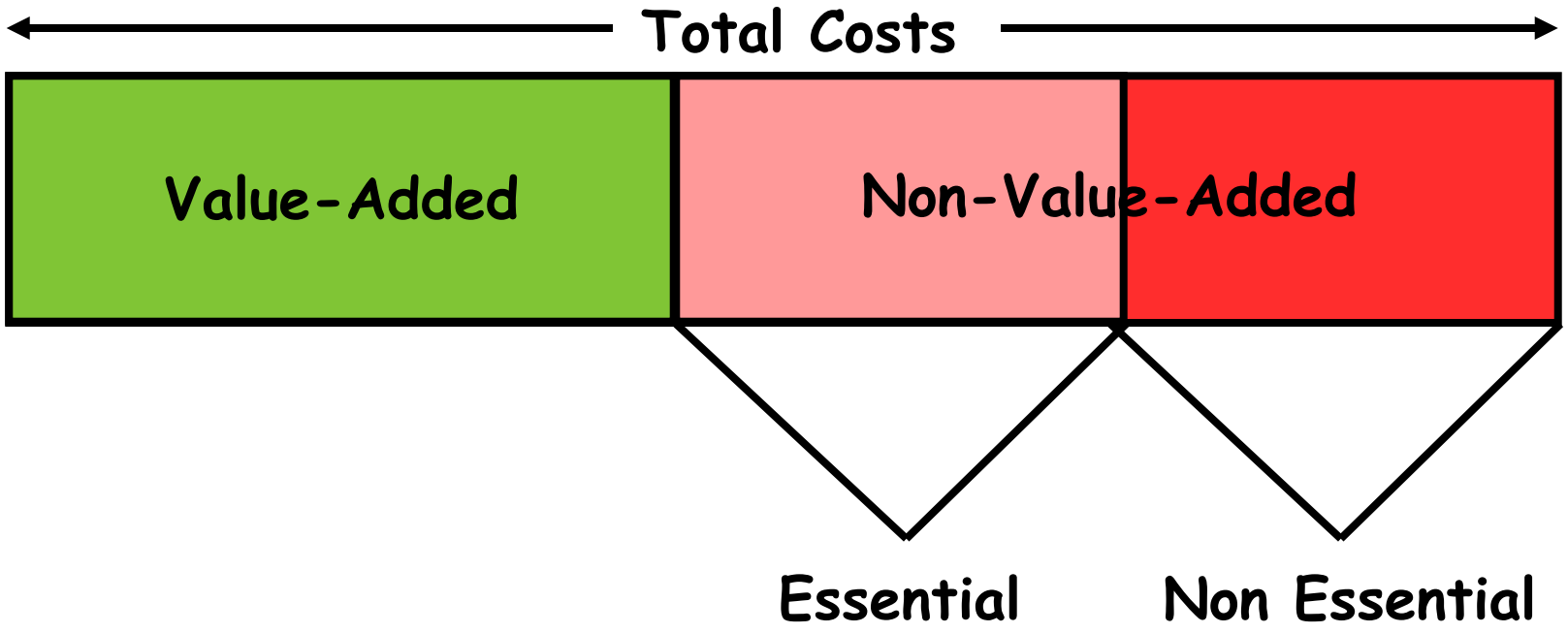
The other things we do to produce the product



And Another Perspective



What Are the Costs of Software Development?



Value-Added Costs

Costs for supplies and tasks performed …

- Materials (e.g., paper, software)
- Labor hours (salaries, benefits)
- Capital equipment (workstations, facilities)

… that produce value

- Products
- Customer satisfaction
- Future labor that will not be expended
  - For example, reduced maintenance and repair

The Strict Definition of "Value-Added"

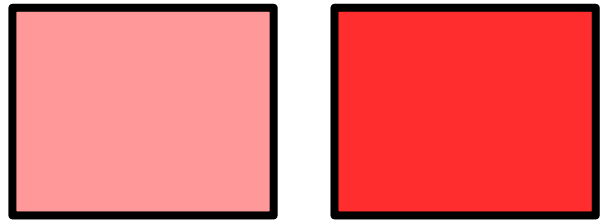
Any activity that is part of the development process is considered a value-added activity if it meets three criteria:

1) *Must change the product* in some way
2) *Must make the product more desirable* to the customer (i.e., the customer wants the change)
3) *Must be done right* the first time

The Strict Definition

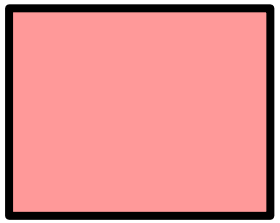
- **This very strict definition helps us open our minds**
- **So we identify the proper targets for process improvement.**
- **Anything that is not value-added is a suitable target for removal or improvement.**

Things Not Part of Value-Added



- **Features the software engineer thinks are nice but the *customer doesn't care about***
- **Moving a product around**
- **Translating between *incompatible tools***
- **Repairing *mistakes***
- **Tests and inspections**
- **Most management activities**
- **Activities unrelated to the development process**
- **Many other things we tend to think of as "necessary" or "desirable"**
  - ***And some of them are necessary!***

Some Non-Value-Added Activities Must be Done



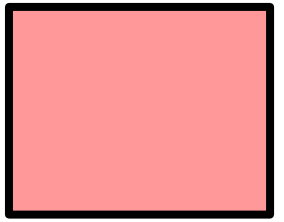
- **Management**
- **Quality Assurance**
- **Testing**
- **…**

May be worthwhile even though they do not add value

The term "value-added" is used to help us open our minds as we improve our processes.

It does not mean that the above tasks are not worthwhile or that the people who do them are not important.

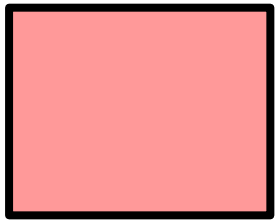
Such Tasks are Called "Non-Value-Added Essential" Tasks



Tasks performed because the process of developing software is not perfectly efficient

- Peer reviews
- Evaluations, inspections, verification and validation (testing)
- Data collection, storage and analysis
- Extra reviews and verifications required by customer or company policy (usually because of past problems)
- Certain overhead costs (employee benefits, support activities)

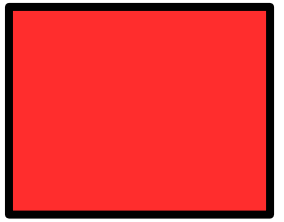
Why are These Tasks Essential?



- **They might not be necessary in a perfect world**
- **But they are necessary with our current methods of product development**
  - and our current level of product development knowledge

Every process has *some* essential, non-value-added elements

Non-Value-Added, Not Essential

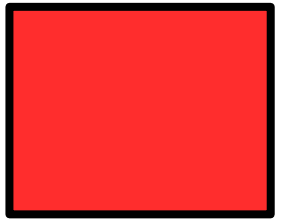


Tasks that are not value-added and that are not essential

Typically, these are tasks that we perform because our processes are inefficient or error prone

- **Examples:**
  - Things we do wrong because we are ***careless***
  - Things we do wrong because we ***don't know how to do them correctly***
  - Things we have always done but ***no longer need to do***
  - Things that ***once made sense*** but ***don't any more*** due to newer technologies or changes in the organization or environment

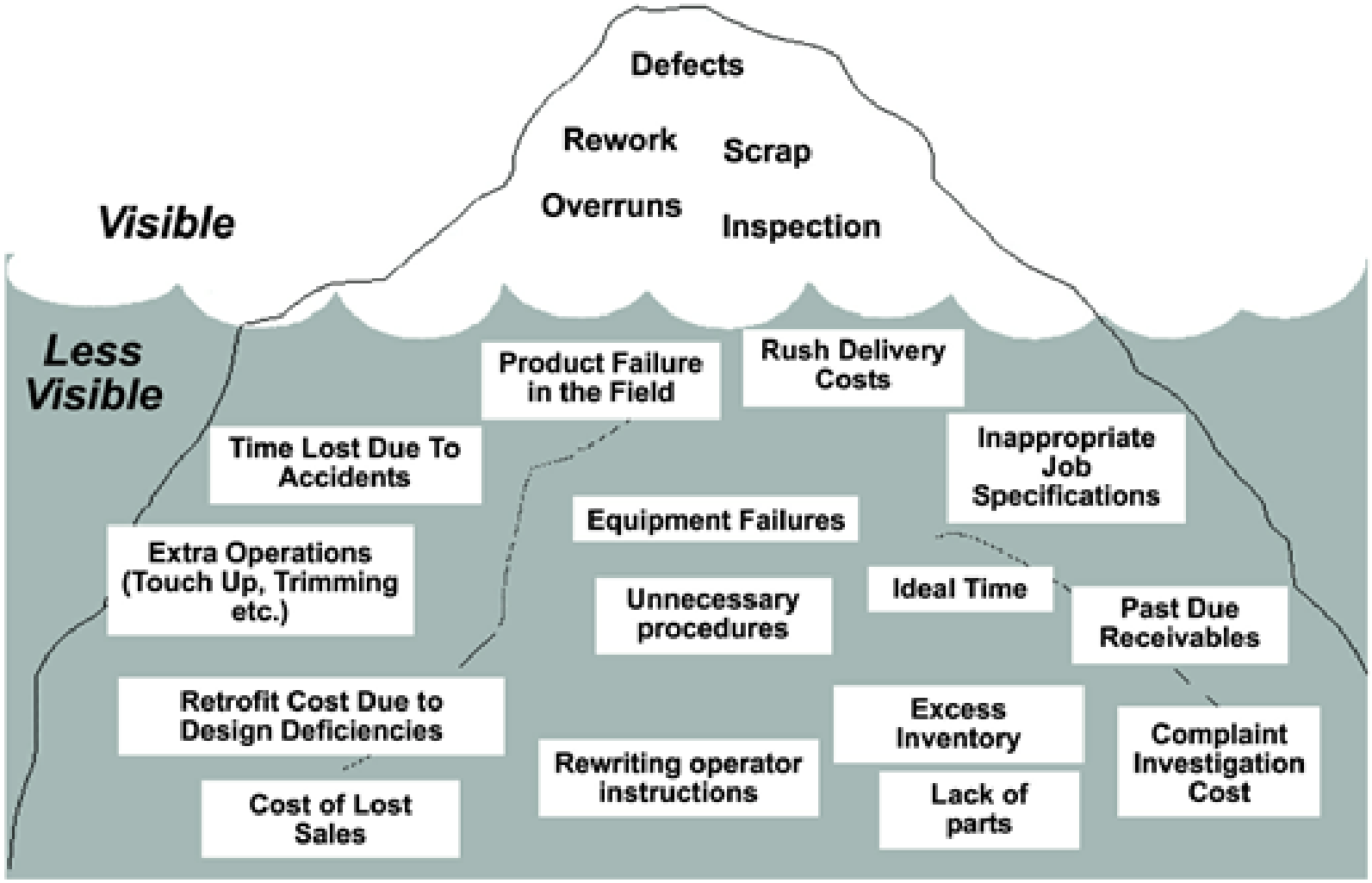
Examples of Non-Value-Added, Not Essential Tasks



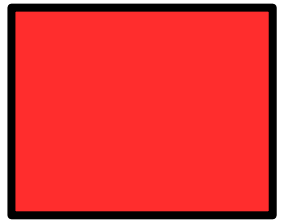
- ***Excessive paperwork*** or approvals
- ***Rework*** – doing it over again because we did it wrong the first time
- ***Waits*** for equipment repairs, networks, test equipment, approvals, etc.
- ***Debugging*** because we did a sloppy job of design or coding
- Costs resulting from ***bugs or other deficiencies in our software development tools***
- Costs for ***activities unrelated to the development process***

These tasks should be eliminated or streamlined first, as they add cost and risk for no useful purpose

Non-Value-Added Tasks are Often Hard to See



Some Costs are Especially Painful



Tasks not performed during software development (or not performed at the right time or in the right way) that cause *high costs later on*

- Failure related costs
- Debugging & Correcting defects
- Maintenance and repair
- Dealing with unhappy customers

High costs we could have avoided

- **These can subtract value:**
  - Loss of customer good will
  - Future labor that must be expended

Typical Value-Added Categories

Value-Added	Non-Value-Added (costs $, no value to customer)	
1) Customer Wants 2) Changes Product 3) Done Right the First Time	Essential	Non-Essential
• Design • Development • Fabrication • Documentation • Assembly • Process • Creation • Upgrade • Shipping	• Set-Up • Training • Planning • Customer-required test • Moving Data Between Steps • Many Quality Improvement activities	• Extra paperwork • Waits • Delays • Bottlenecks • Counting • Installing Software Tools • Extra Un-wanted Features • Rework • Service • Modification • Expediting • Recall • Correction • Retest • Error Analysis

Example: Result of Value Added Analysis

Value-Added	Non-Value-Added (costs $, no value to customer)	
1) Customer Wants 2) Changes Product 3) Done Right the First Time	Essential	Non-Essential
• Requirements analysis • Design • Coding • Documentation • Integration • Manufacturing • Packaging • Shipping	• Estimating • Training • Planning • Customer-required acceptance test • Configuration Control • Inspections	• Approval by 7 people! • Delays for test systems • Data conversion between design tool and coding tool • Wait for subcontracted hardware • Debugging • Service calls • Warranty costs • Shipping costs for patches • Loss of customer goodwill • etc.

Typical Result After Cost Analysis

- **Value-added -- 35% of total cost**
- **NVA Essential -- 20% of total cost**
- **NVA Non-essential -- 45% of total cost**

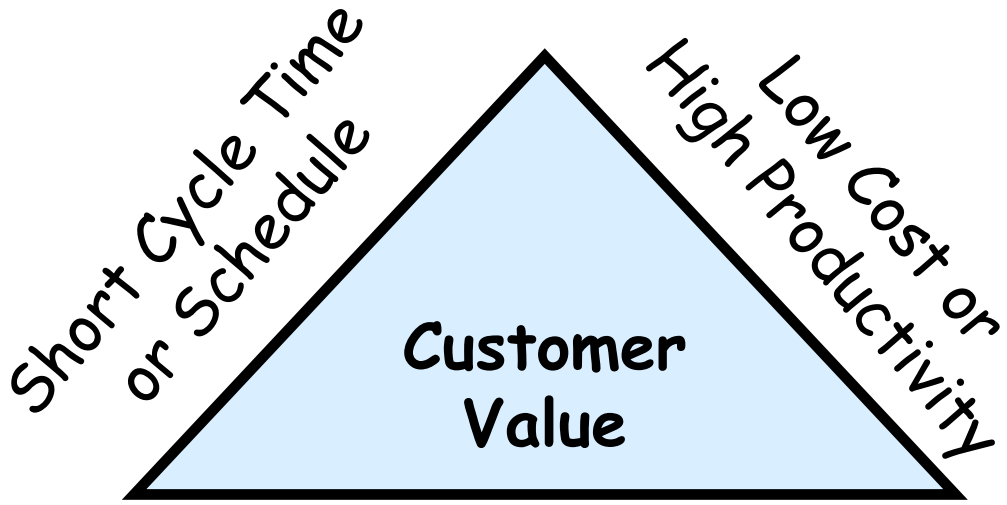
➢ **Top three non-value-added items for typical software projects:**

- ***Rework*** due to design and coding errors -- 14%
- ***Extra customer support*** -- 12%
- Labor costs for individuals ***waiting*** for equipment that is not available -- 11%

Part 2 - Outline

- Introduction
- Six Sigma Overview
- Value Added Analysis Overview
- **Cost ot Quality Analysis Overview**

The Cost of Quality



Quality
(Fewer Defects; Customer satisfaction)

Quality costs money

But improving quality can save money

The issue: how to save more than it costs

Categorizing Quality-Related Costs

1) Cost of Conformance

- The cost of activities that ***improve quality***
  - **Prevention Costs** – activities that ***prevent poor quality***
  - **Appraisal Costs** – activities that ***detect poor quality***
    - So you can fix the product right away

2) Cost of Non-Conformance

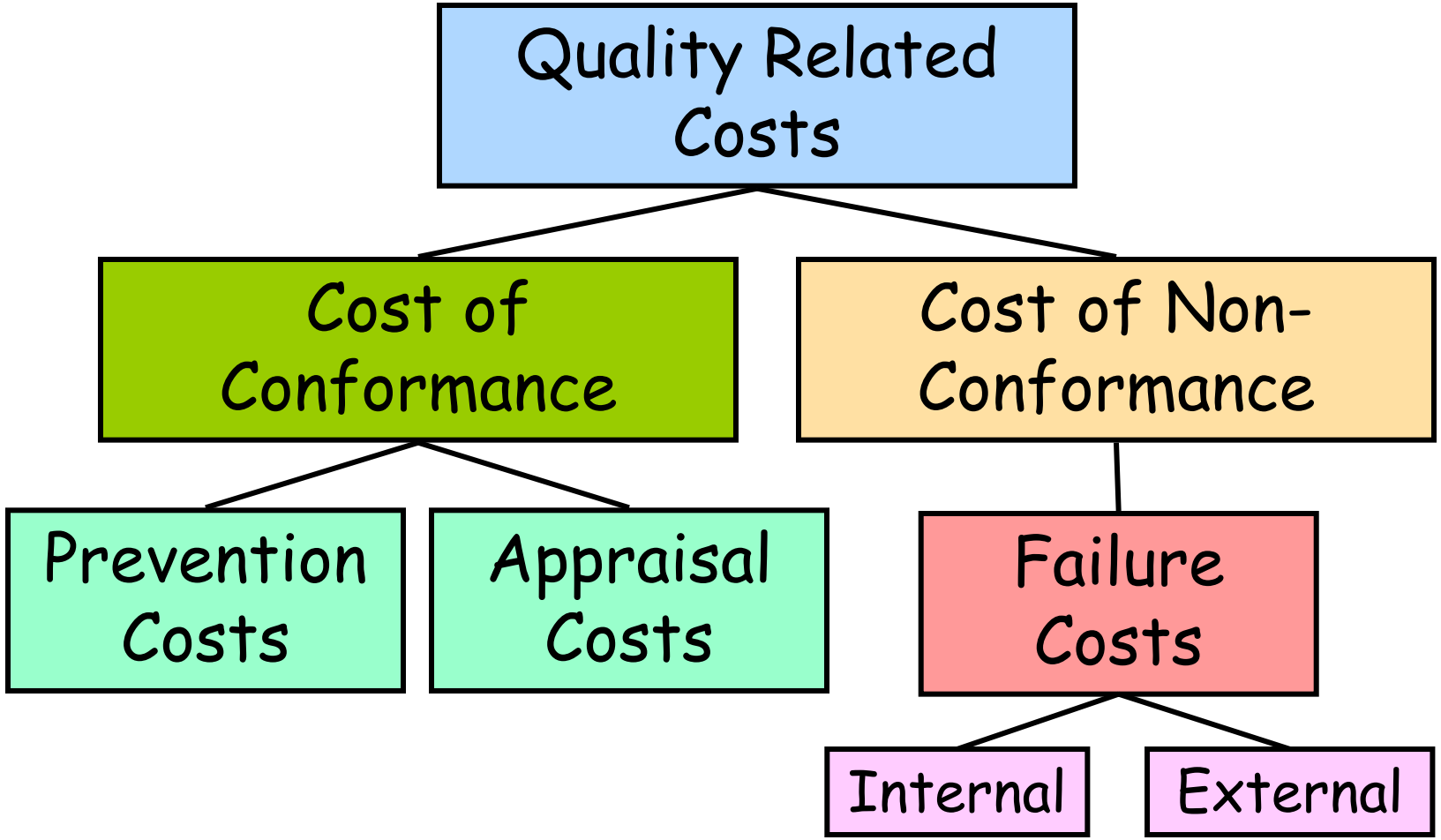
- The ***price of failure*** to achieve quality
  - **Internal** Failures - Costs ***before product shipment***
  - **External** Failures - Costs ***after product shipment***

Recommended Quality Strategy:
Invest in conformance to save in non-conformance

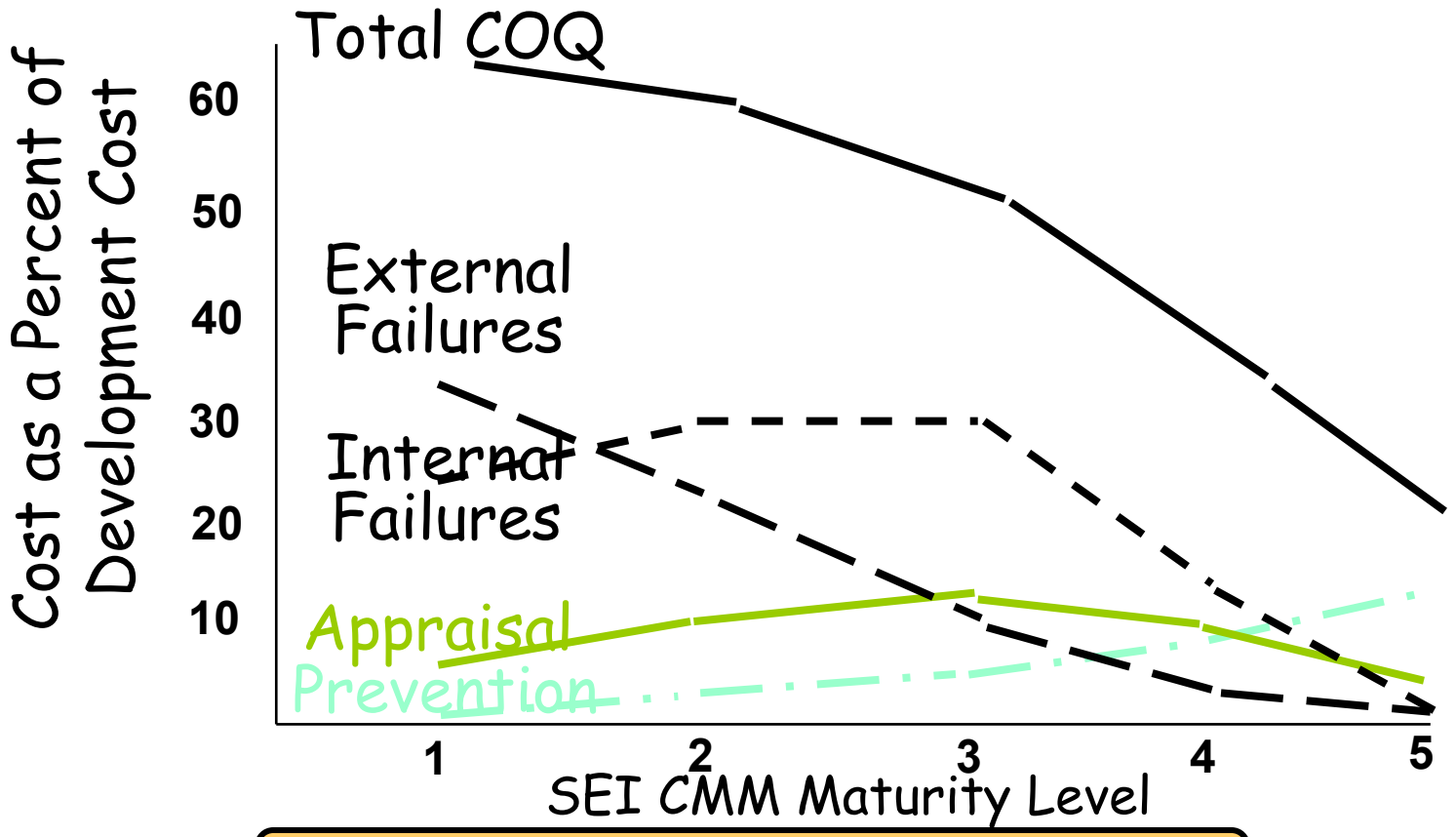
Example

- **If you test the software first, before shipping, and catch a bug before shipping, then the bug might cost you very little:**
  - ***Cost to test the software***
  - ***Cost to debug the software***
  - ***Cost to repair 1 copy of the software***
- **If you develop the software without much testing, and ship to 1000 customers, then a bug might cost you a lot:**
  - ***Cost to debug the software***
  - **Cost to repair 1000 copies of the software**
  - **Costs associated with product failure**
  - **Loss of good will and trust by many customers**
    - **They may buy the next product from someone else**

Categorizing Quality-Related Costs



Effects of Process Maturity on Costs



As reported by Knox (see references)

Net Cost of a Process
Categorizing Tasks & Subtasks

Non Cost of Quality		Cost of Quality (all non-value-added)		
Value Added	Non-Value-Added	Cost of Conformance		Cost of Non Conformance
		Prevention	Appraisal	Failure
• Design • Development • Fabrication • Documentation • Assembly • Process • Creation • Upgrade • Shipping	• Over-head • Errors • Ineff-icien-cies	• Training • Planning • Simulation • Modeling • Consulting • Qualifying • Certifying	• Inspection • Testing • Audits • Monitoring • Measure-ment • Verification • Analysis	• Rework • Service • Modification • Expediting • Recall • Correction • Retest • Error Analysis

End of Part 2

UT Dallas

Software Quality and Software Testing

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

Part 2 –Achieving Software Quality

Part 3 - Defect Containment

Part 4 – 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 Report

Description _____

_____

Phase where found ____

Phase where introduced ___

_____

Priority _____ Type _____

Estimated Cost to Fix _____

etc.

Some of this information may not be determined until you have debugged the software

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

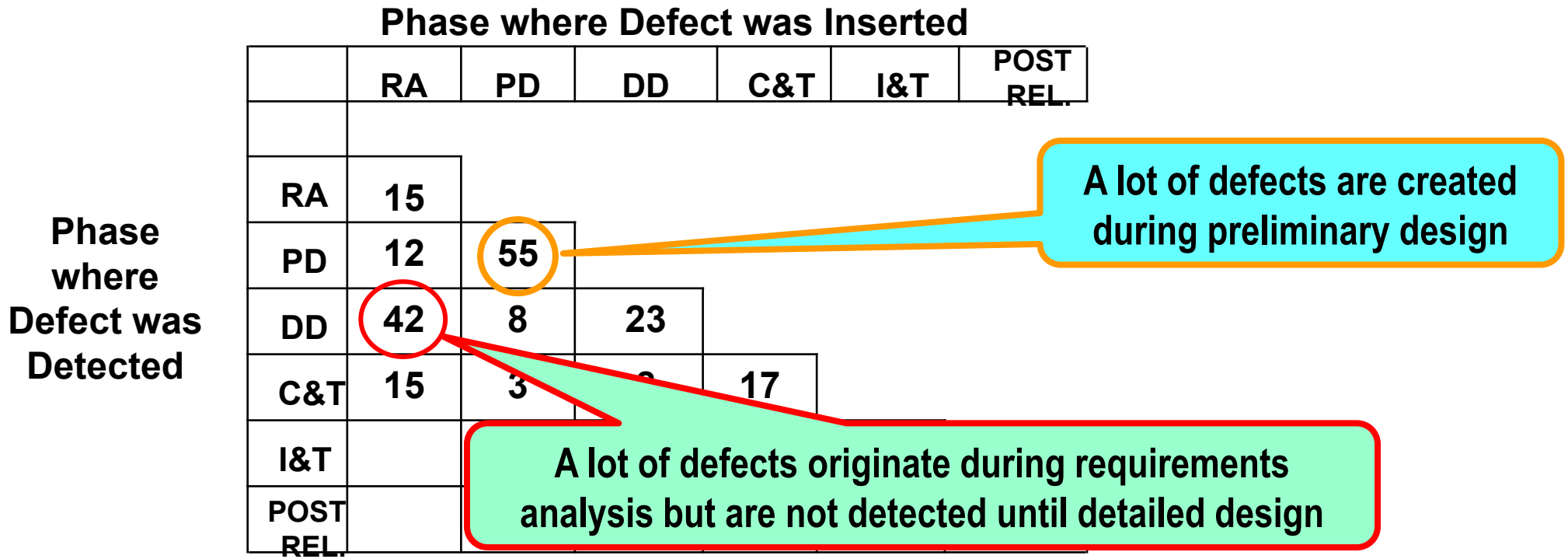
Scrum where Defect was Inserted

Scrum where Defect was Detected	S1	S2	S3	S4	S5	POST REL.
S1	15					
S2	12	55				
S3	42	8	23			
S4	15	3	8	17		
S5						
POST REL.						

This shows the data at the end of the 4th SCRUM

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***

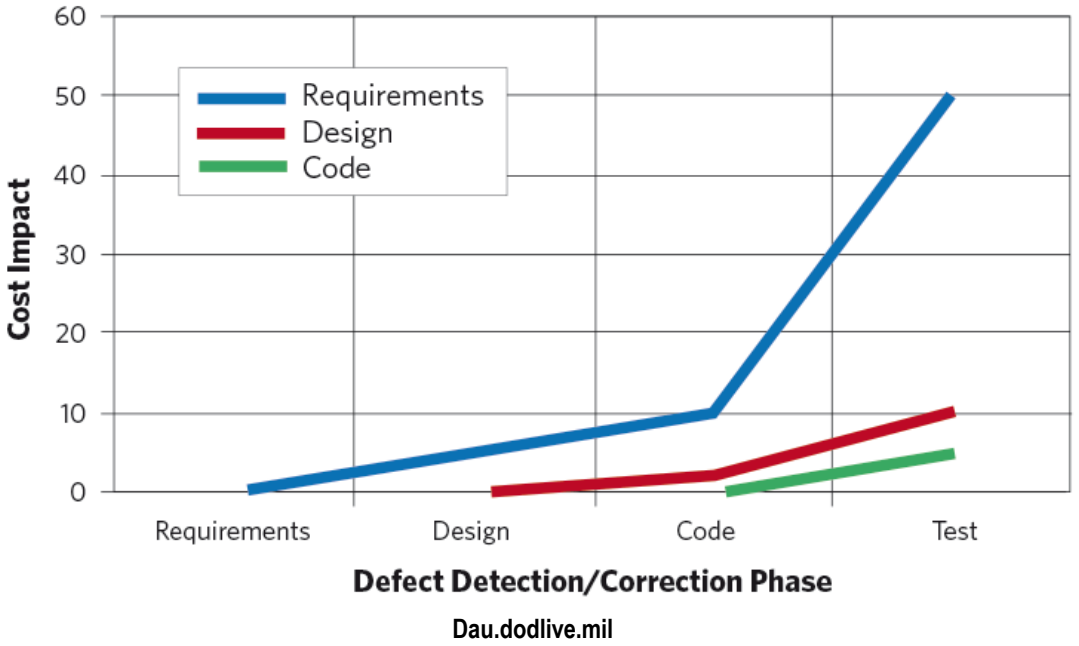
A method for doing this will be shown briefly in today's lecture

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)

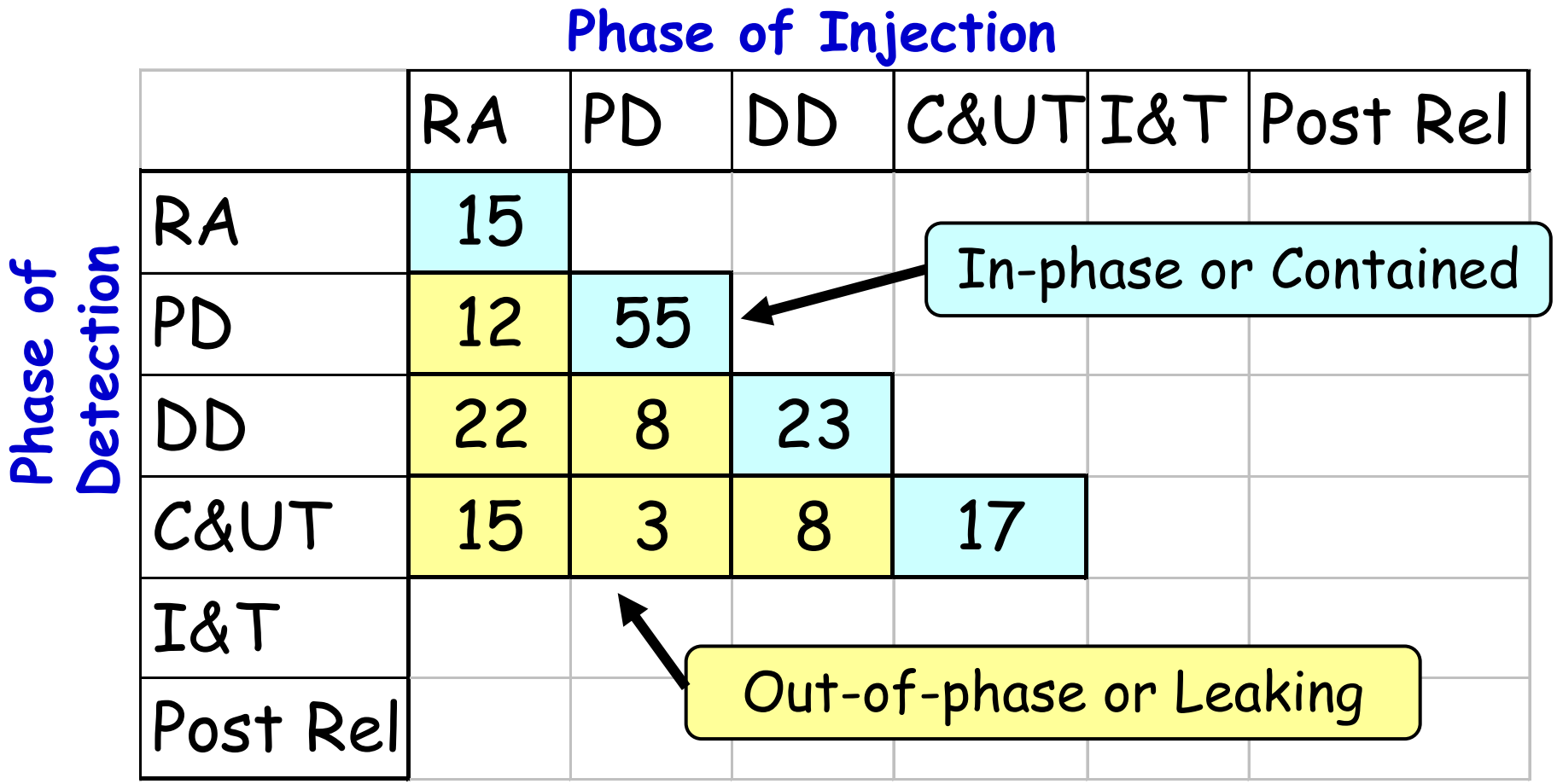
A Key Lesson Learned from Measuring Defect Containment

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

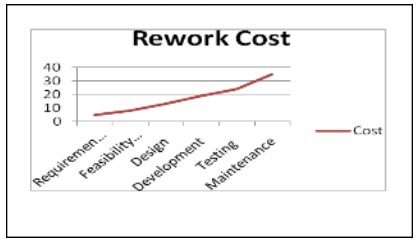
Escaping Defects Cost the Most of All

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

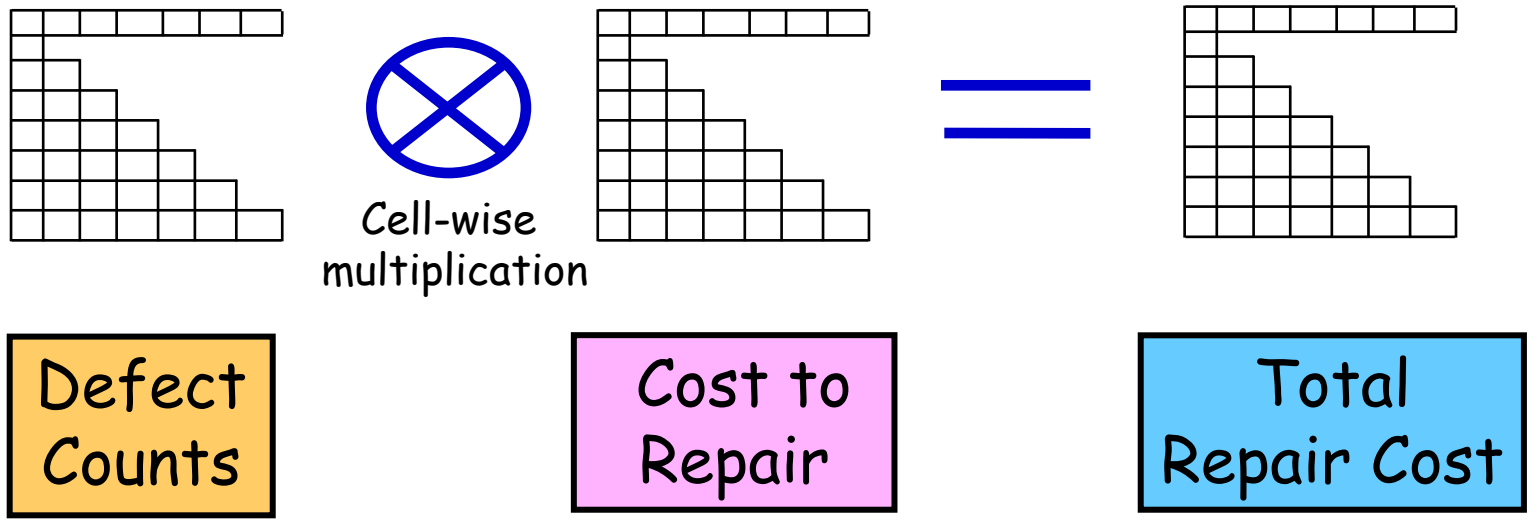
Phase of Injection

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



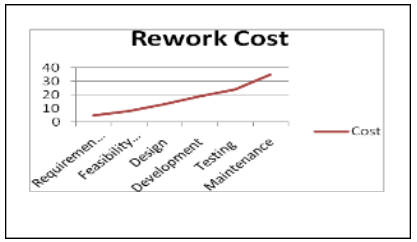
Total Repair Cost Example

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



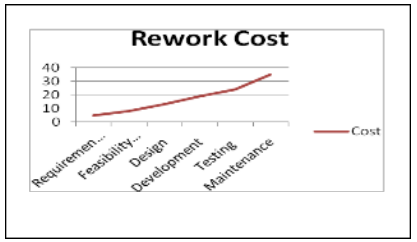
ljser.org

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

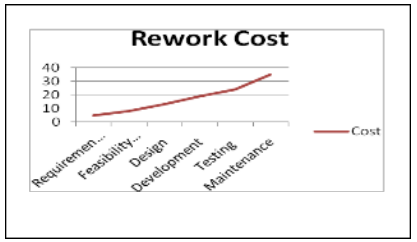
after delivery to the customer

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



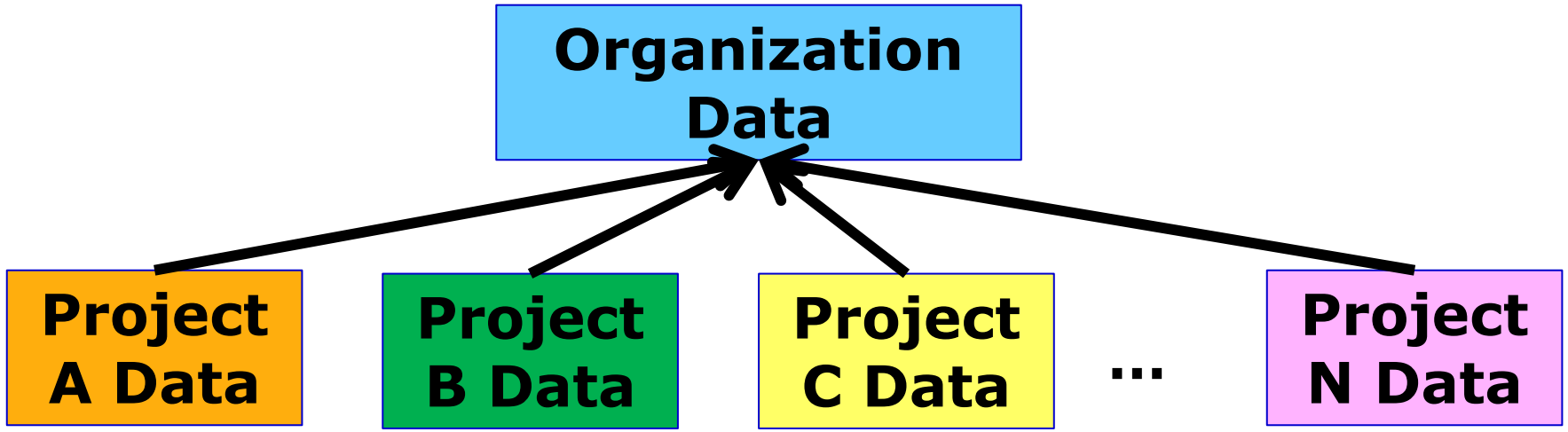
ljser.org

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**



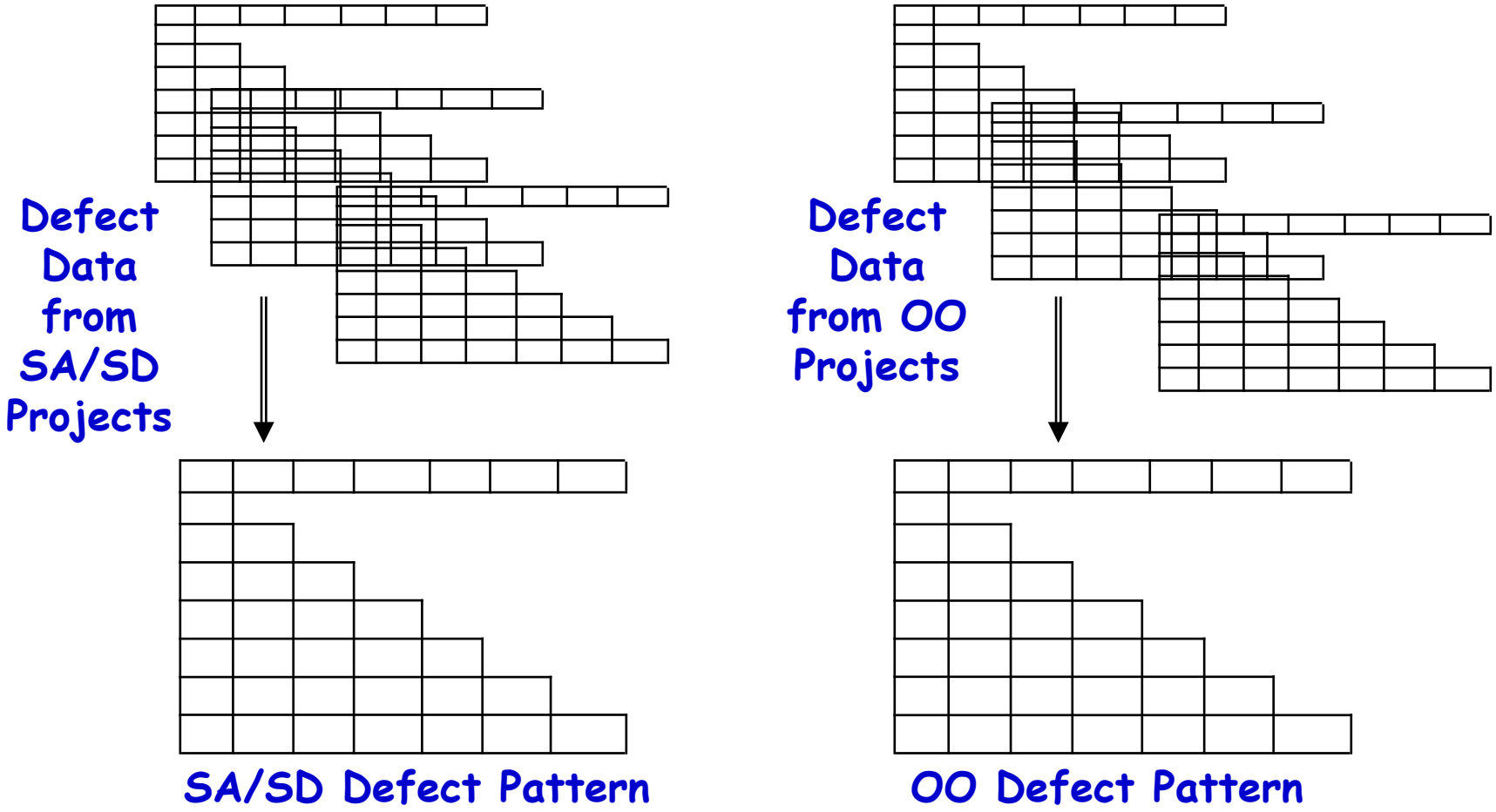
Organizational Analysis of Defect Containment Data

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 –Achieving Software Quality
Part 3 - Defect Containment

Part 4 – Measuring Software Complexity

Contents

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

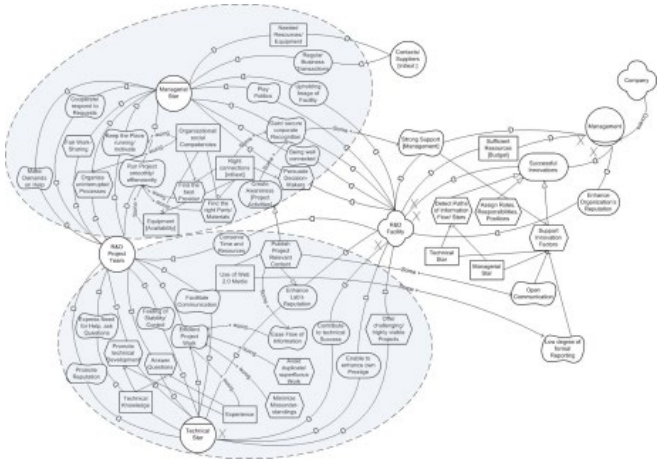
Contents

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

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

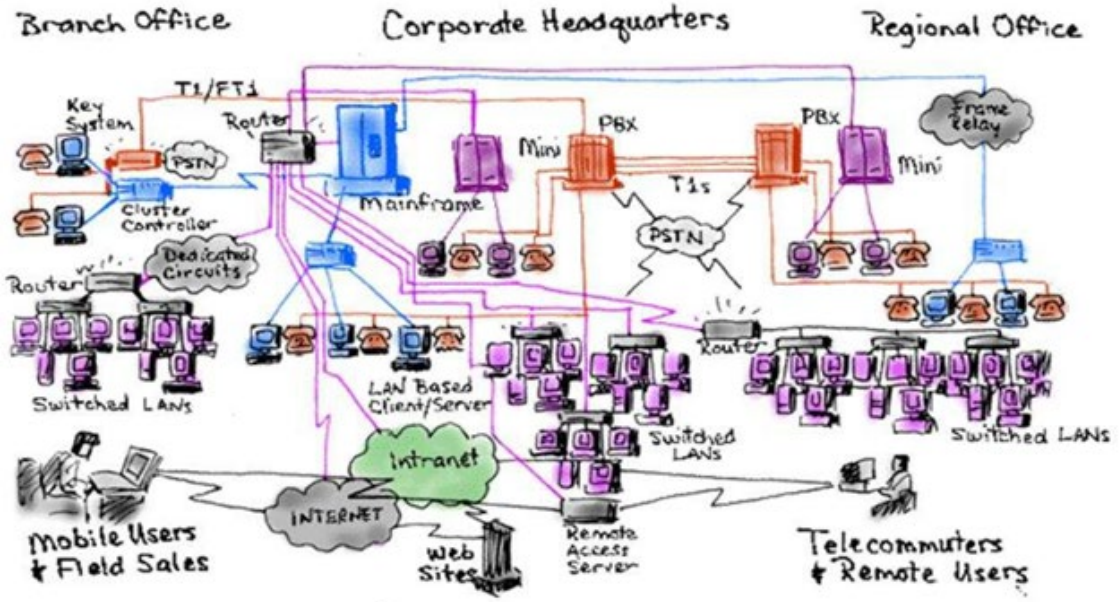
***Complicated*: being difficult to understand but with time and effort, ultimately knowable**

***Complex*: having many interactions 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.

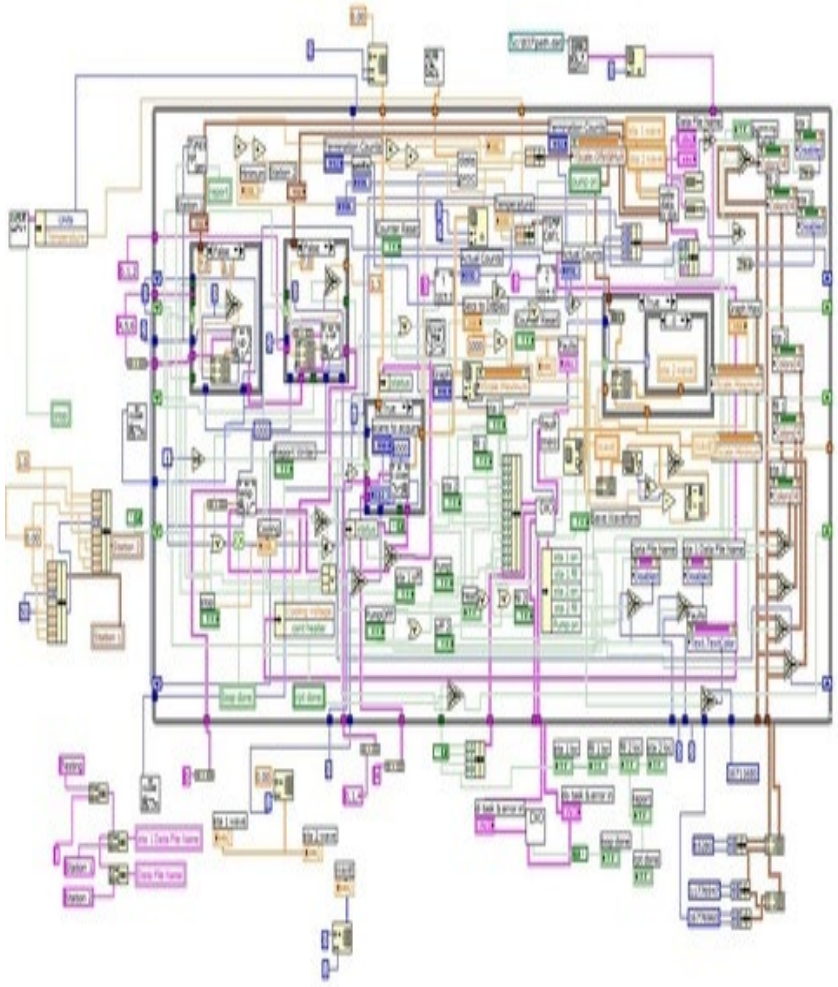
Information Need

- 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?

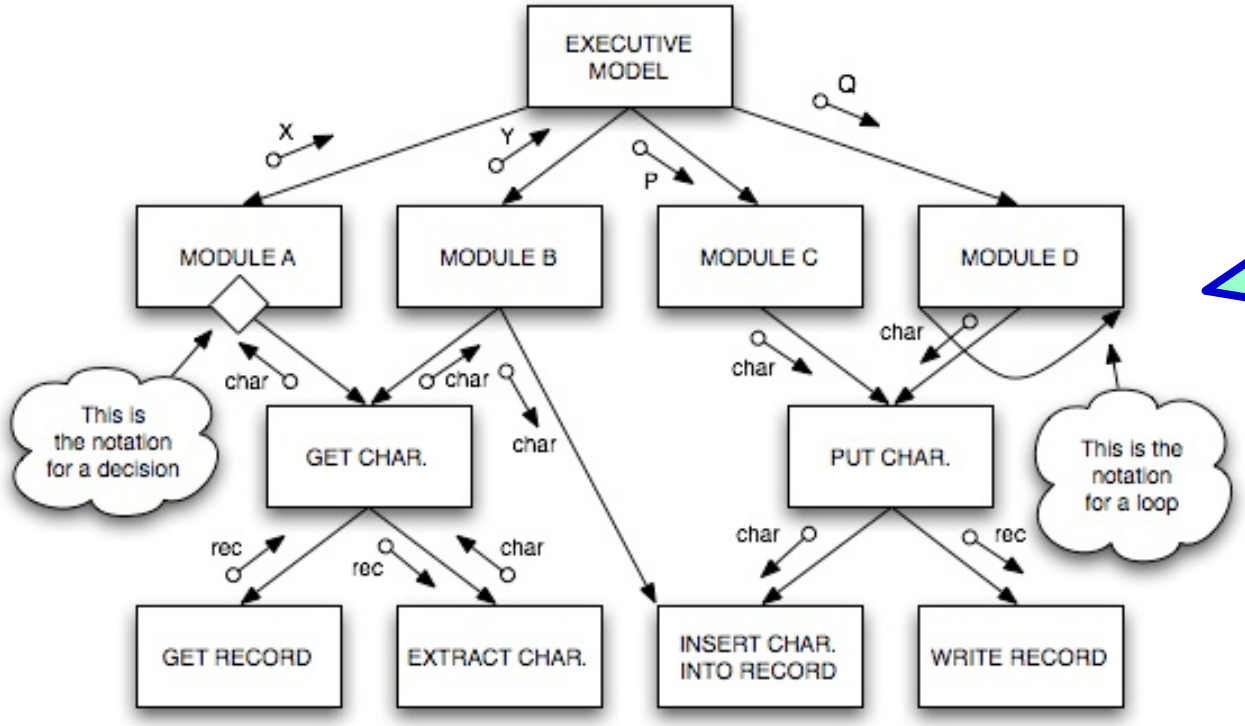
We might learn something about the structure and complexity of software by measuring:

- ***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*

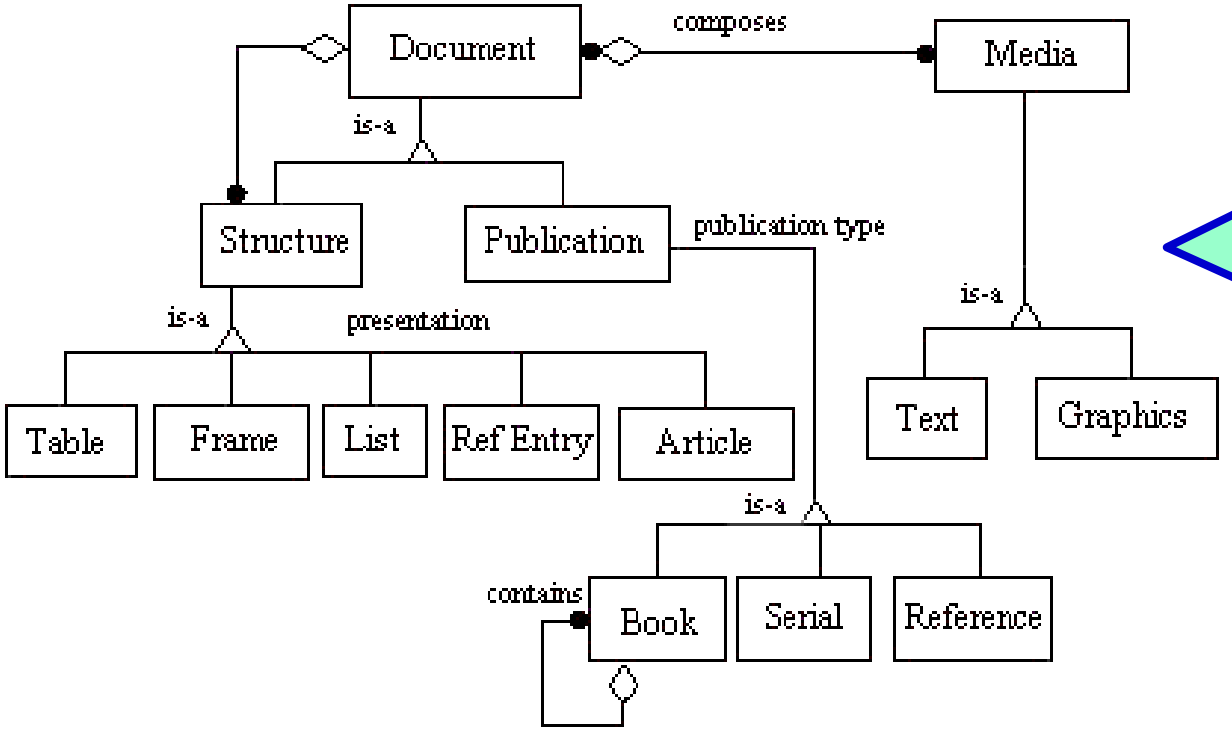


Figure 1. Multimedia Document Model - Object diagram

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

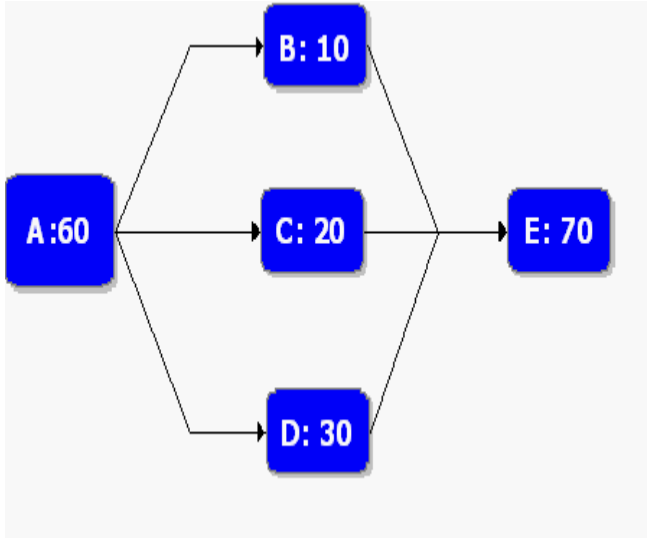
We will focus on complexity of *structured, procedural software*

- 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***

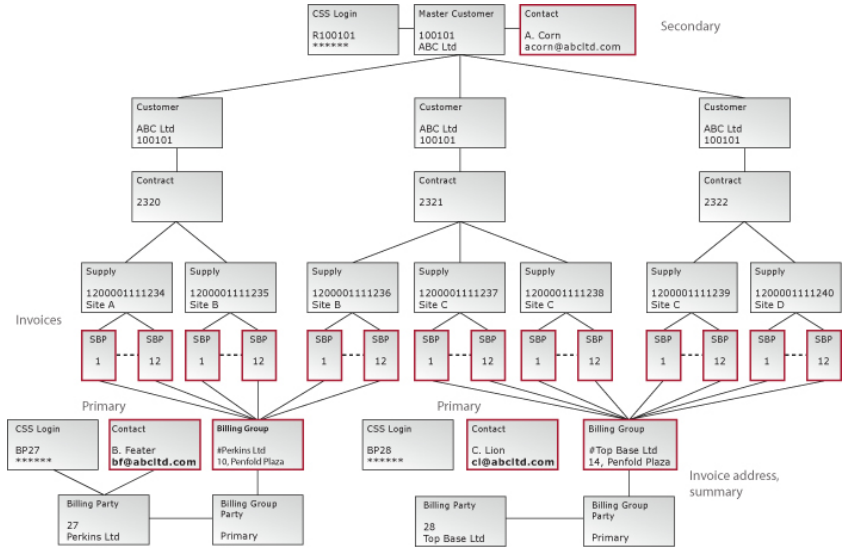
From time to time we will mention how the concepts are applied to *object oriented software*

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



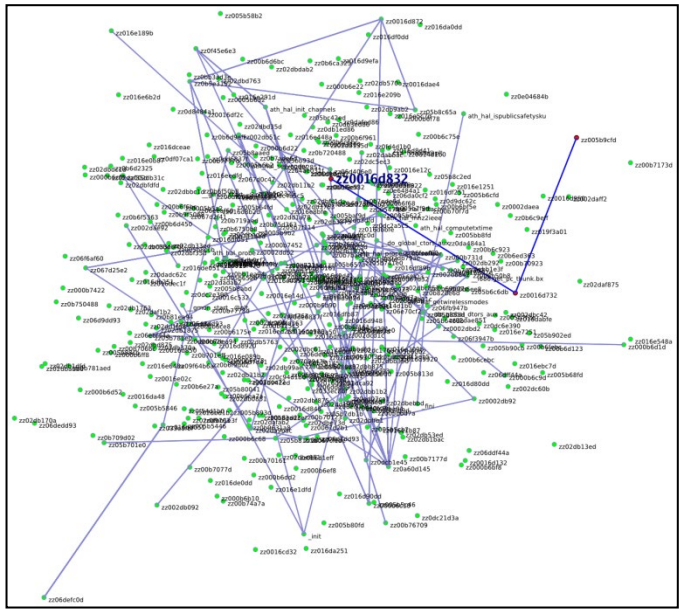
It can be *further complicated* by the degree to which the *components share common elements* (coupling)

Contents

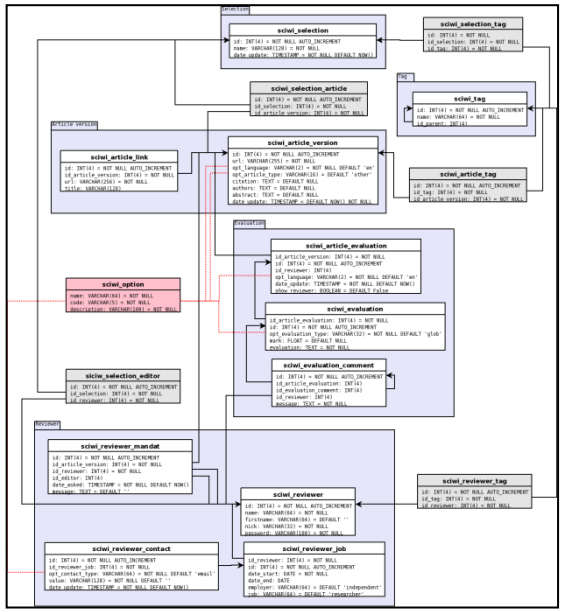
- Complexity: what and how to measure
- ***Structured Programs and Flowgraph Analysis***
- Measures of Complexity
- Closing Remarks

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.

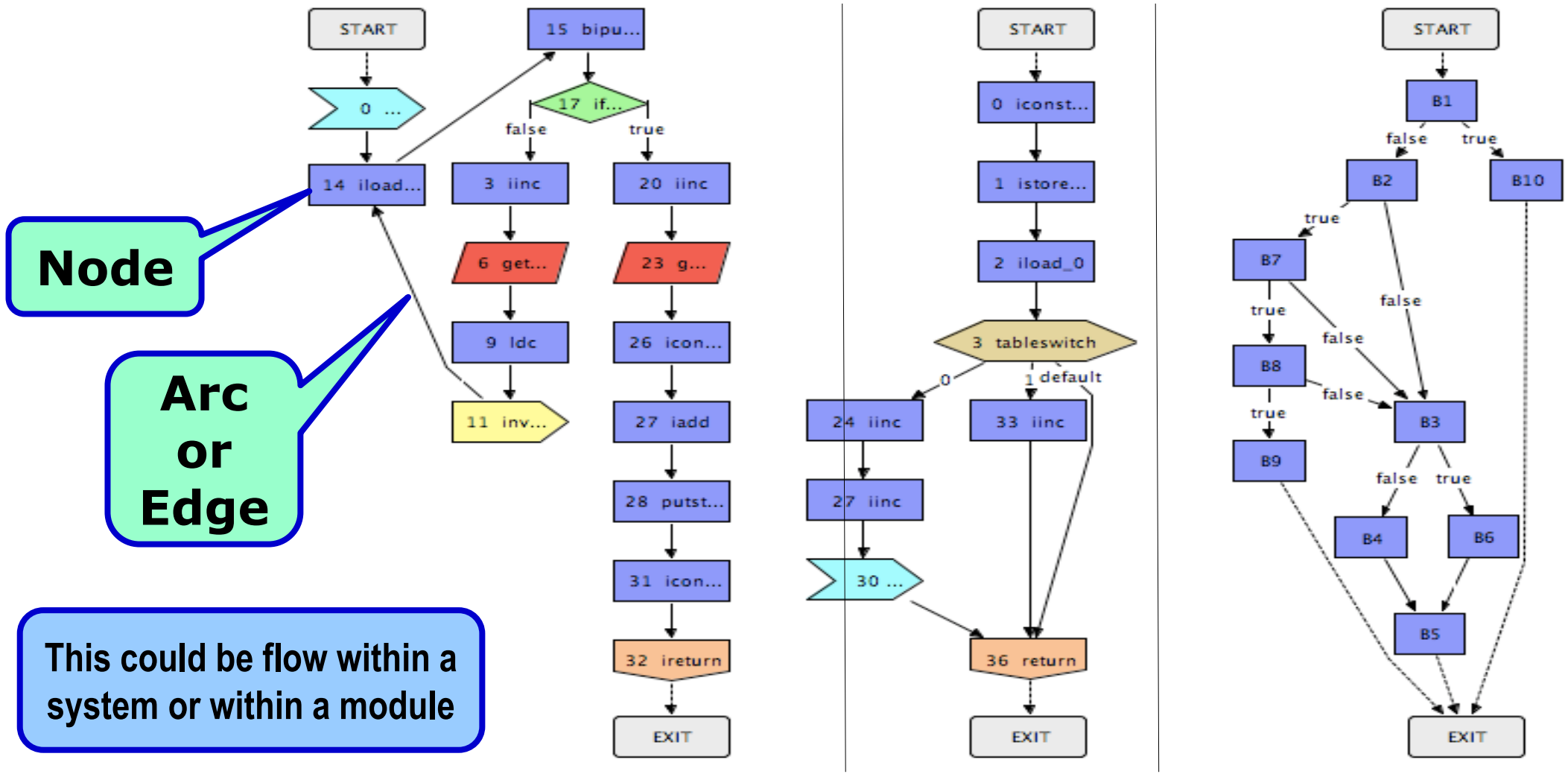


vs



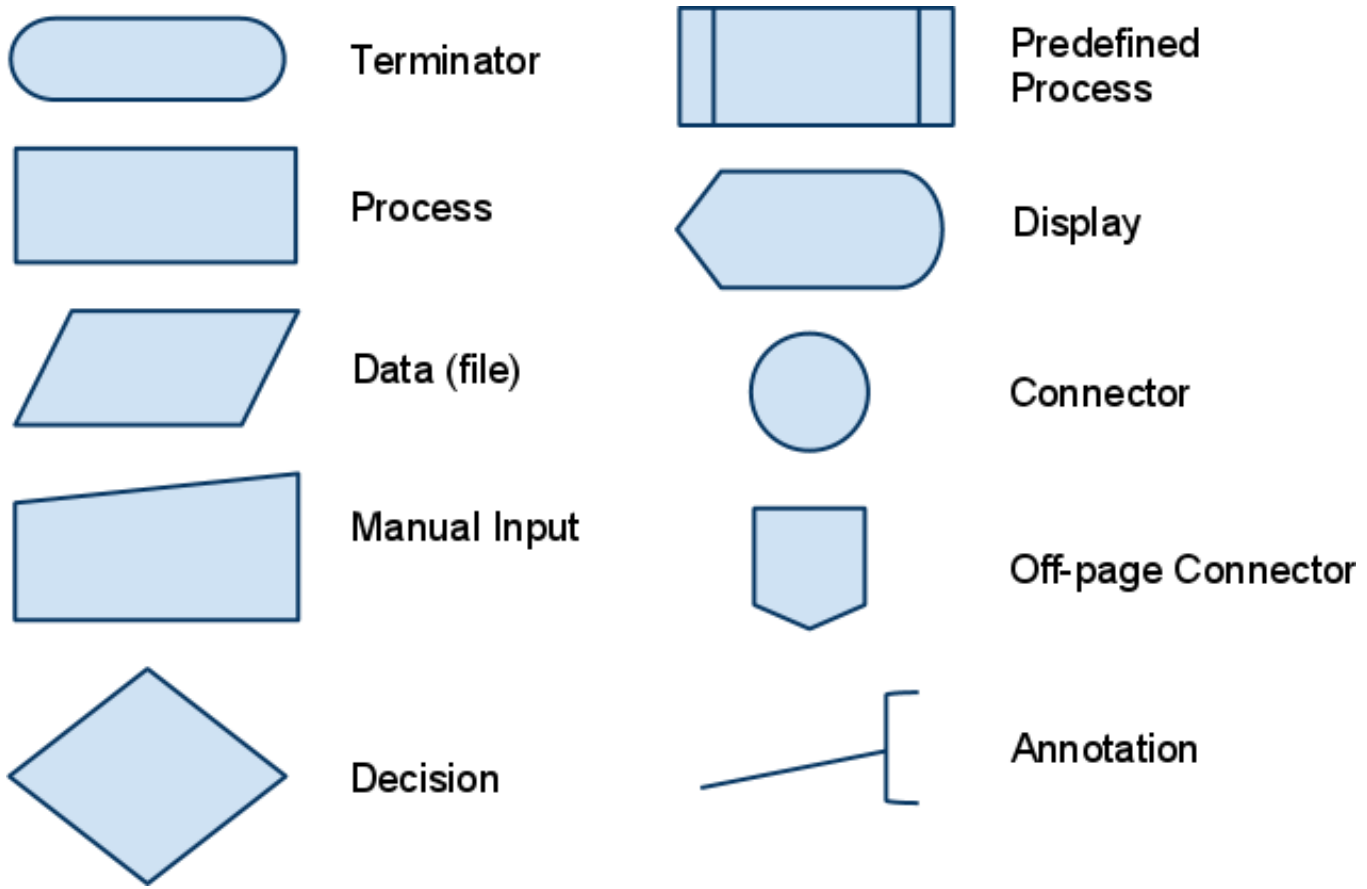
Many measures of complexity make use of control flow analysis.

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)

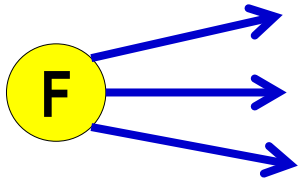
- **Arc or Edge**

A path between nodes

- **Procedure Node**
  - A block of code. Any decisions are internal to the block. One exit.

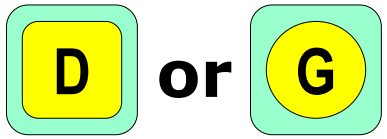
Squarish shape,
Exactly one arc leaving

- **Predicate Node**
  - One that makes a decision.

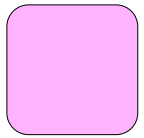


Round shape, Two or more arcs leaving

- **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:**

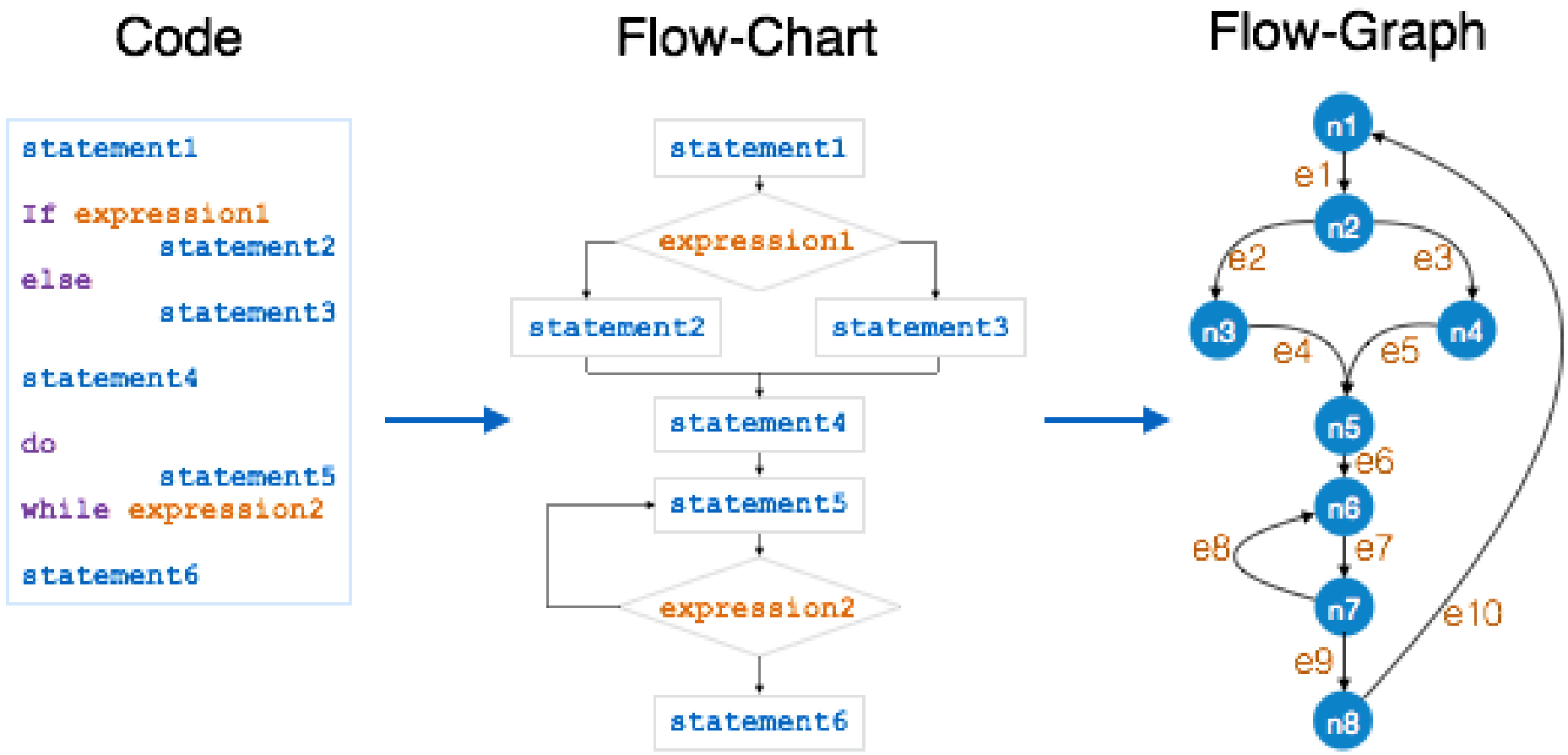
- ***Every other node lies on a path between the start node and the end node***

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.

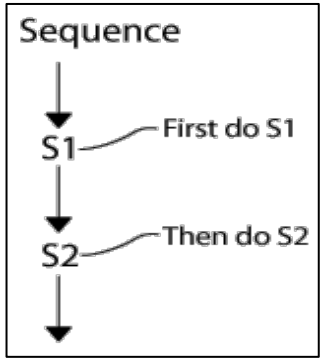
Example: Code, Flowchart, and Flowgraph



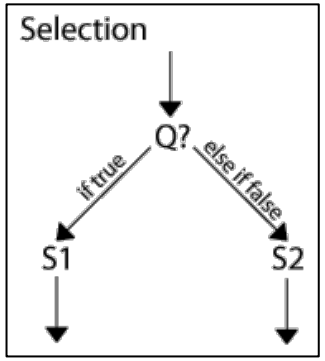
What is a Structured Program?

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

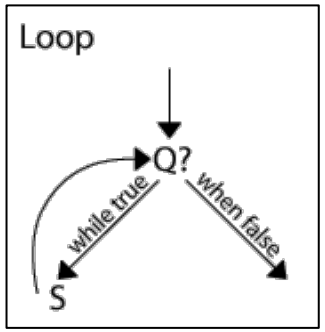
- ***Sequence*** (ordered statements and/or subroutines)
  - Examples: A = B+C; D = FUNC(E,F)



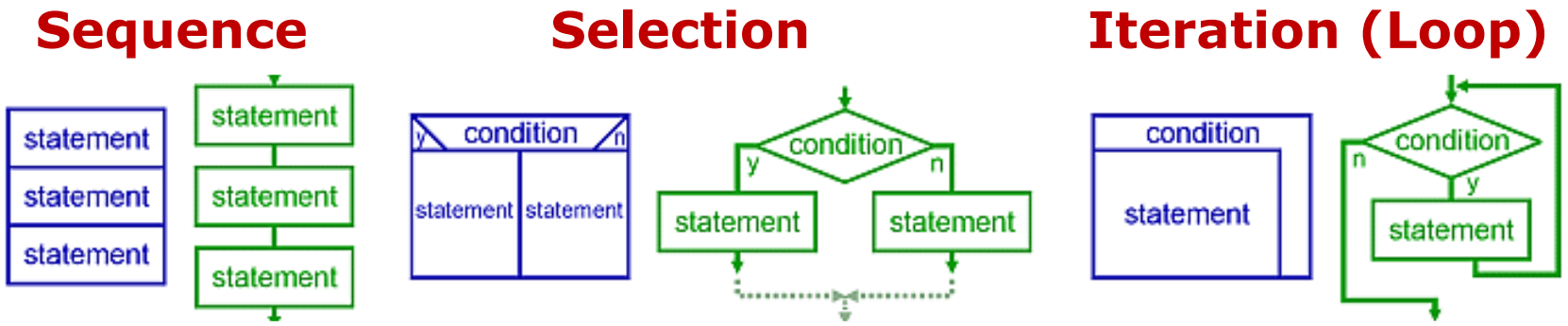
- ***Selection*** (one or more statements is executed, depending on the state of the system)
  - Example: If C1 Then <true option> Else <false option>



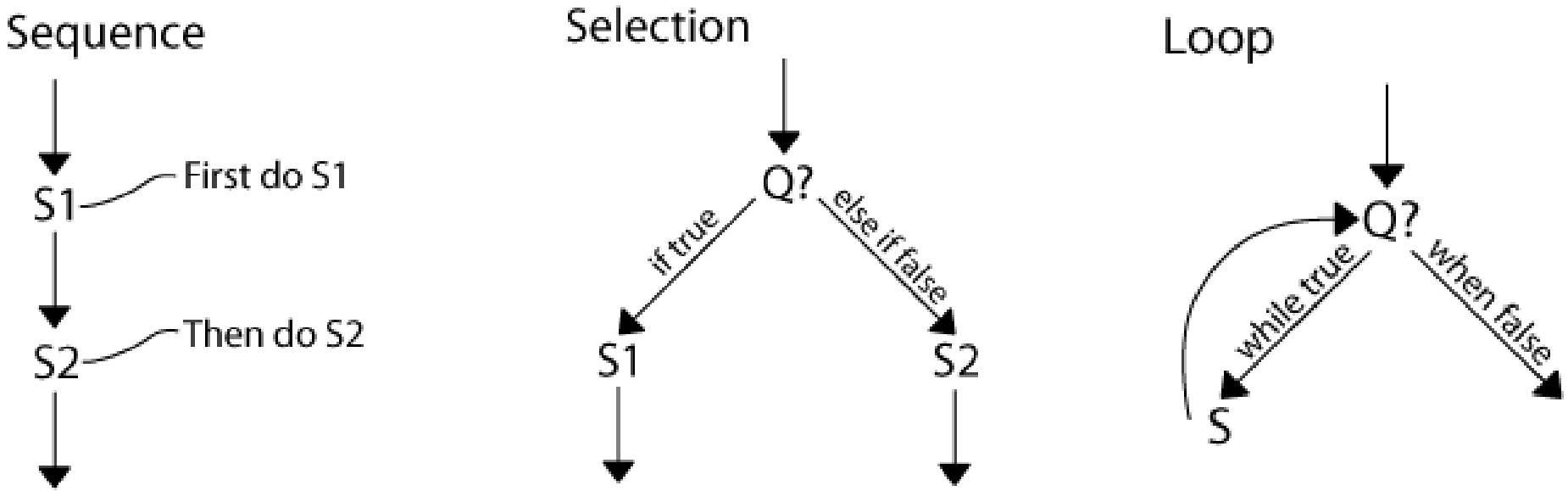
- ***Iteration*** [loop] (a statement or block is executed until the program has reached a certain state)
  - Examples: While; Repeat; For; Do… Until



Structured Program Notation



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

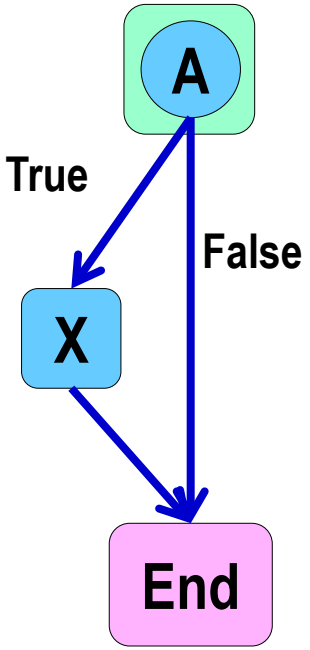
This started out as a theoretical concept, developed by Edsger Dijkstra and others.

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).

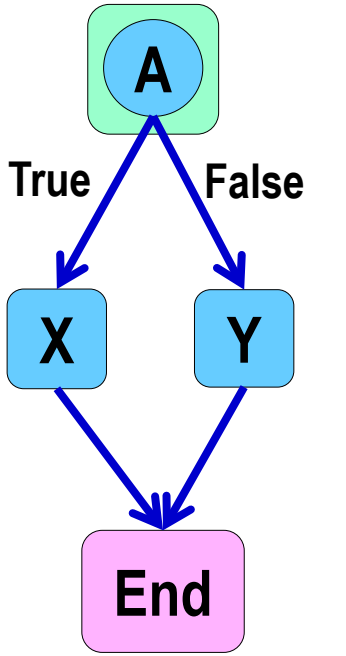
[1] See References

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

Example: Two flowgraphs for selection



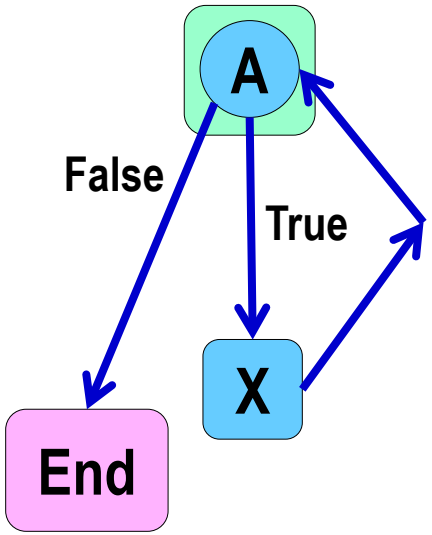
If A then X
(D_0)



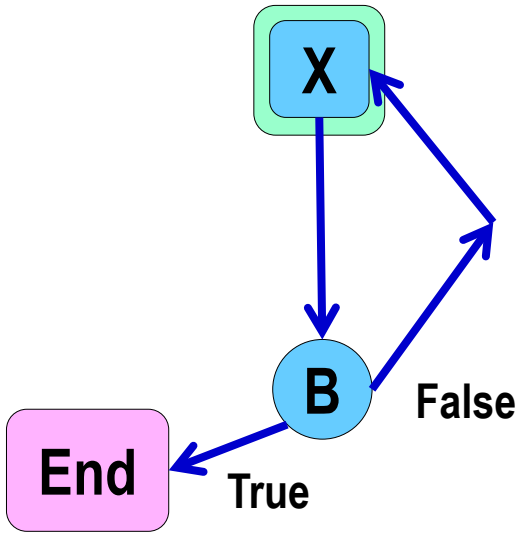
If A then X else Y
(D_1)

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



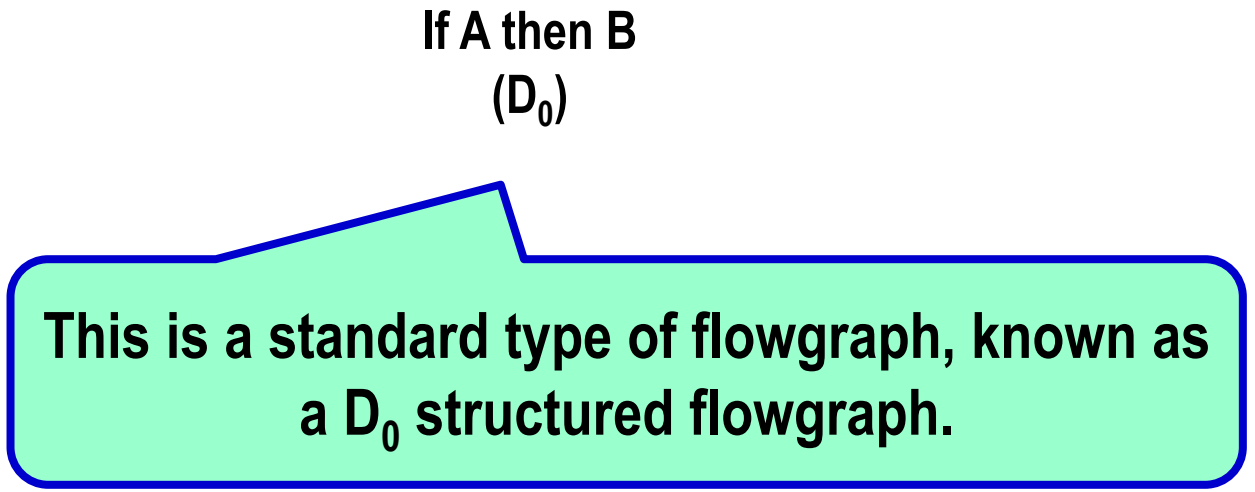
While A Do
X
(D_2)



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.

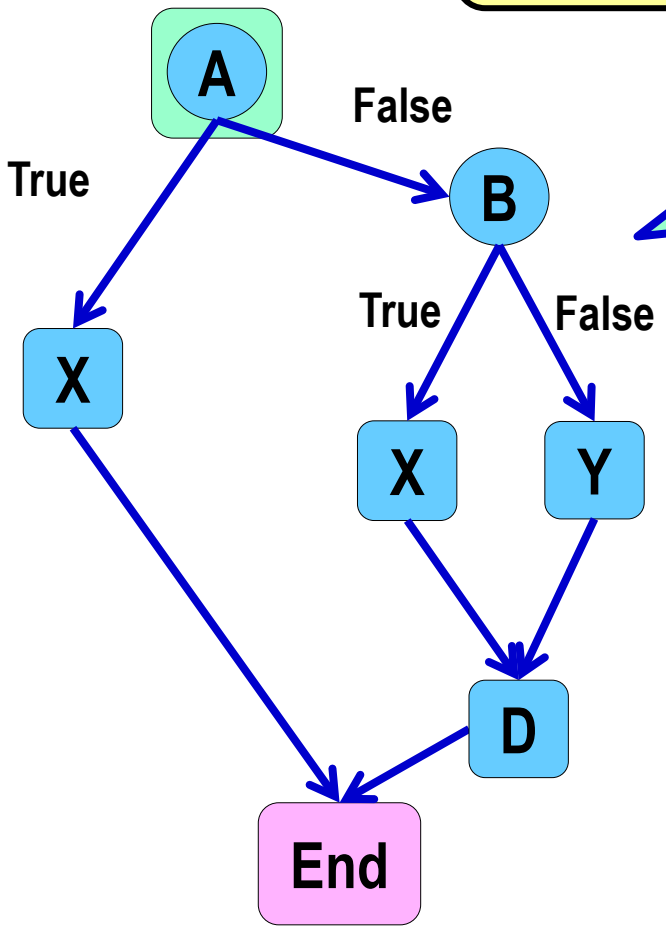
Structured Program Flowgraphs: What Is Common and What Is Not

- **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

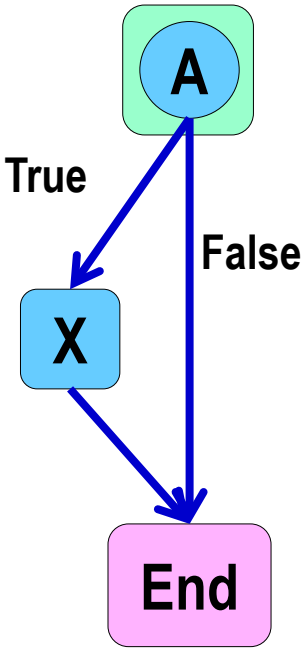


If only D0 and D1 can be used to represent this code, then we must use a D1 within another D1 and *must show X twice.*
This is the equivalent of rewriting the source code as shown below.

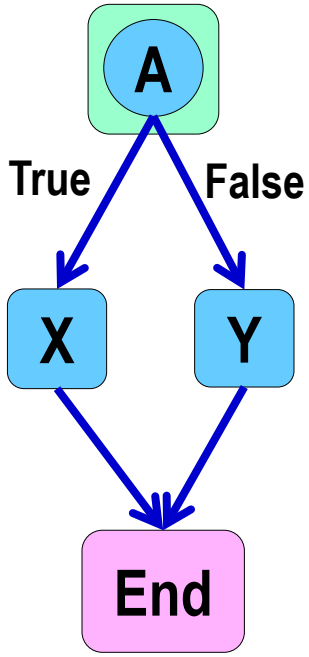
IF A THEN X
ELSE
** IF B THEN X**
** ELSE Y**

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

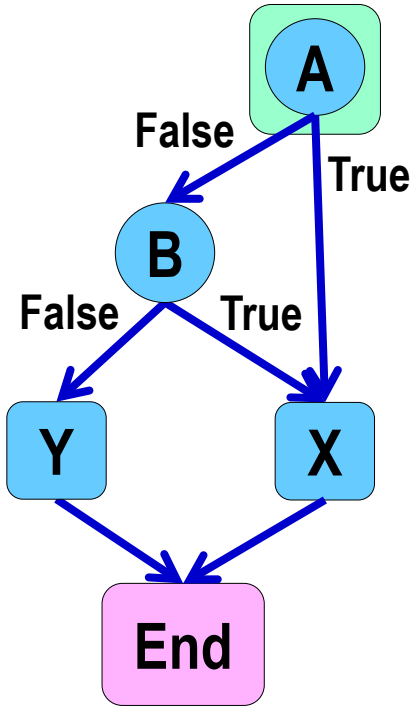
D_5 Was Introduced To Allow Common Boolean Selection Decisions



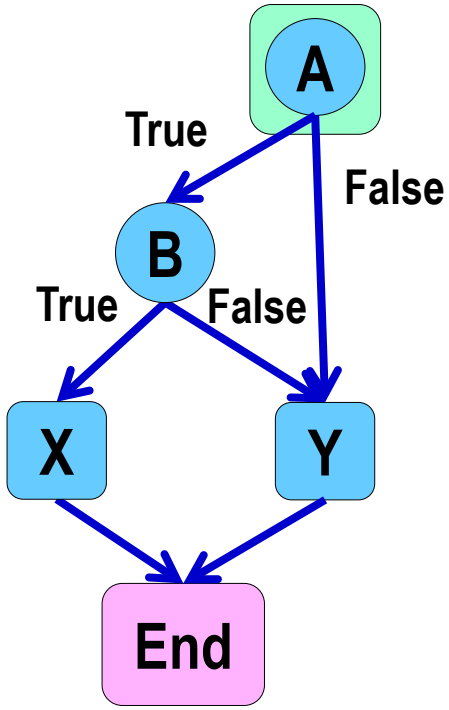
If A then B
(D_0)



If A then B else C
(D_1)

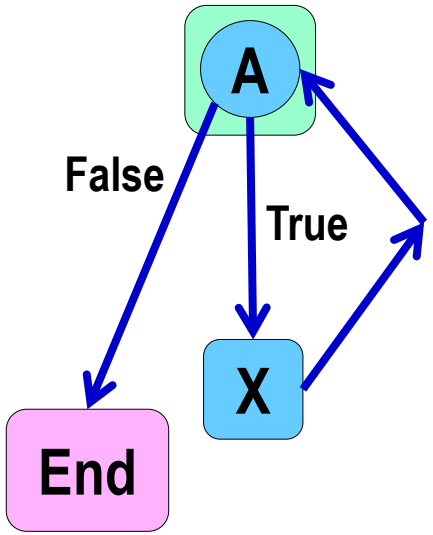


If A or B then X
else Y
(D_5)

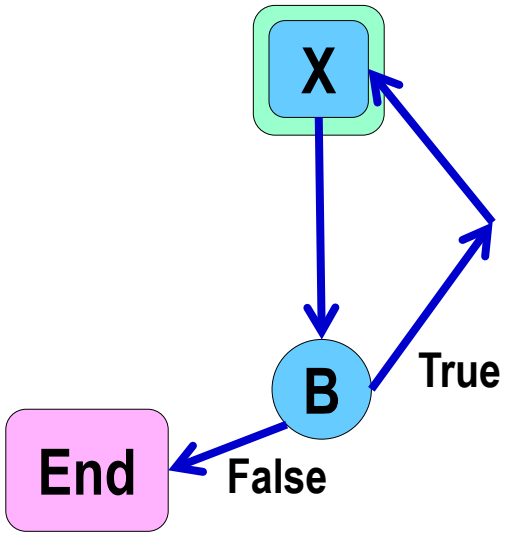


If A and B then X
else Y
(D_5)

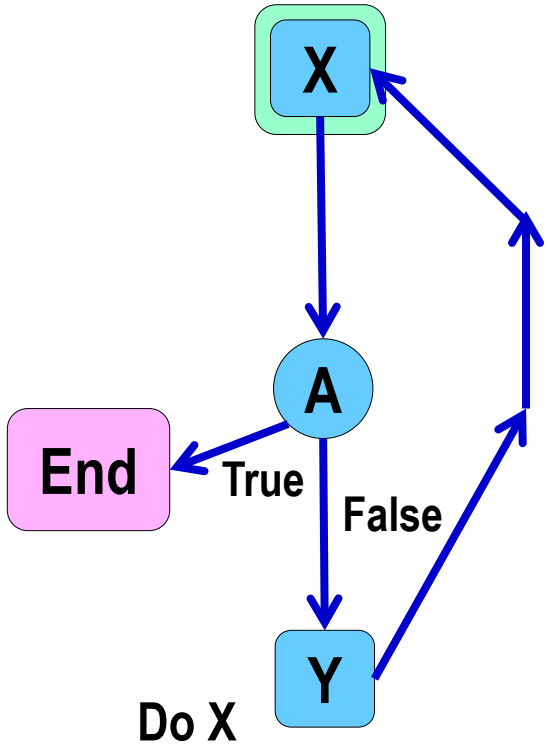
D_4 Was Introduced to Allow Middle-Exit Loops



While A
Do X
(D_2)

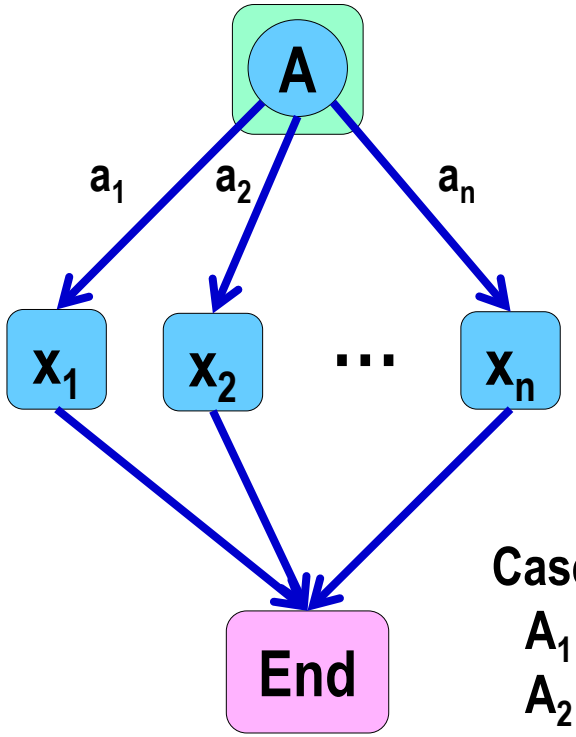


Repeat X
Until B
(D_3)



Do X
Exit when A
Do Y
Repeat
(D_4)

C Flowgraphs are Prime Flowgraphs for CASE Statements

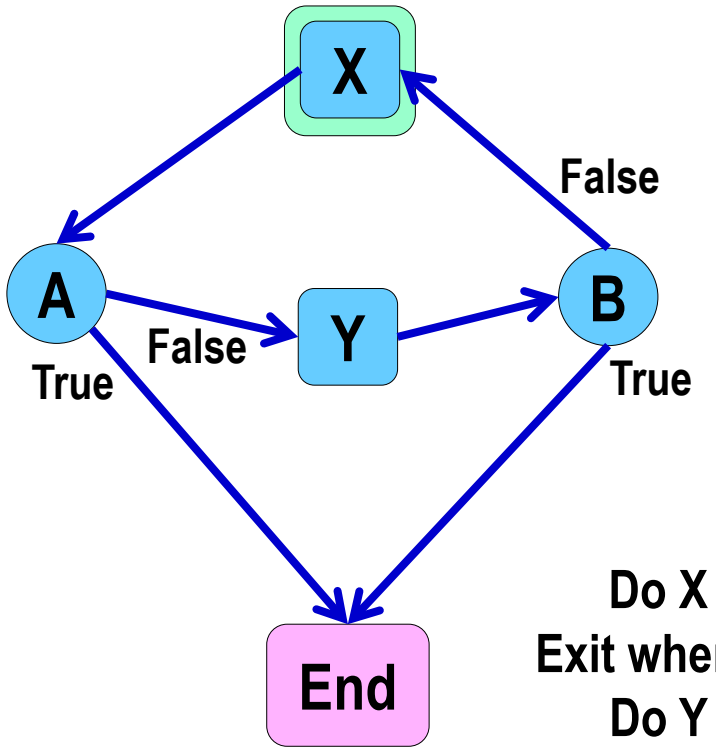


Case A of
$A_1 : X_1$
$A_2 : X_2$
...
$A_n : X_n$
$(C_{1...n})$

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

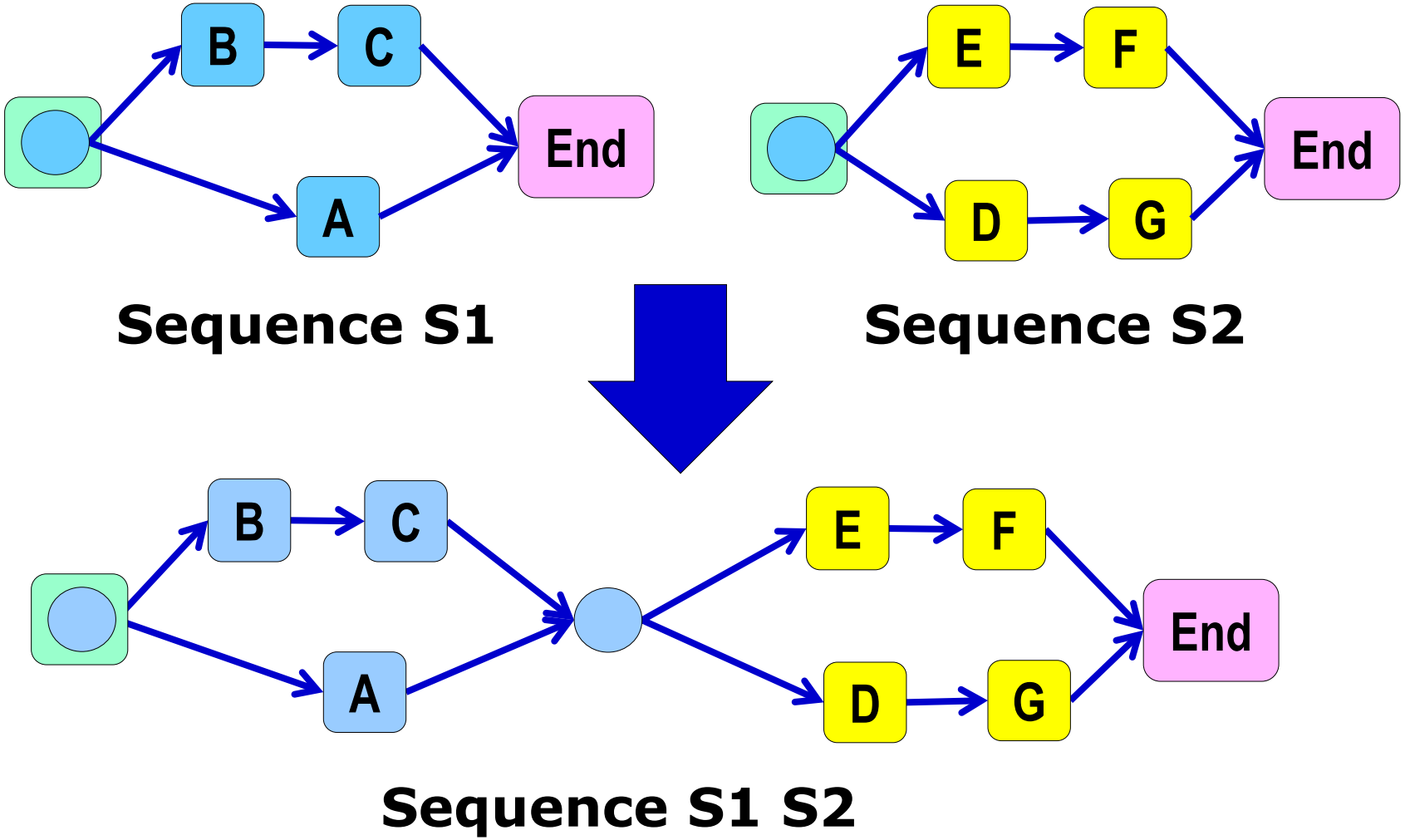
Flowgraphs with a single entry and single exit can be *combined* in the following ways:

- ***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***

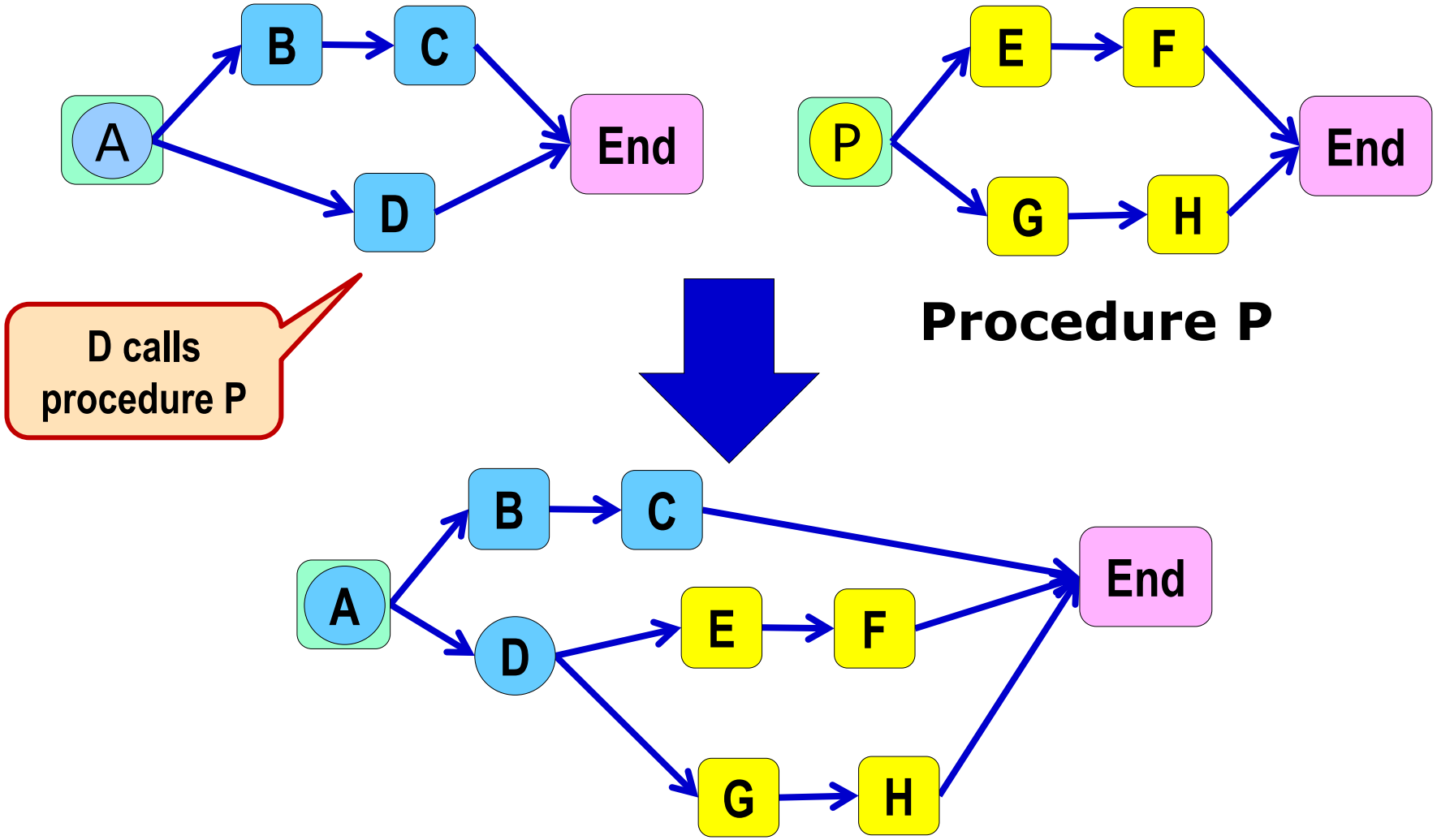
Flowgraphs can also be *reduced* or *condensed* or *decomposed* by reversing the above

- **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

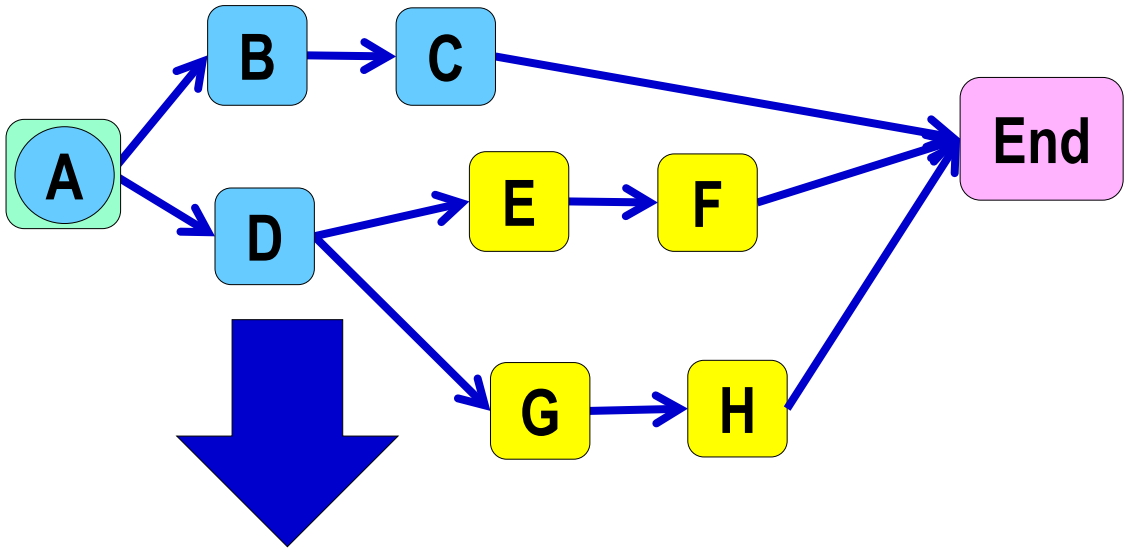
Sequencing Example



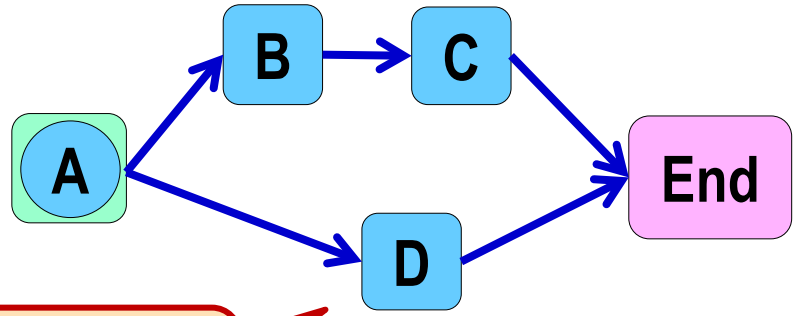
Nesting Example



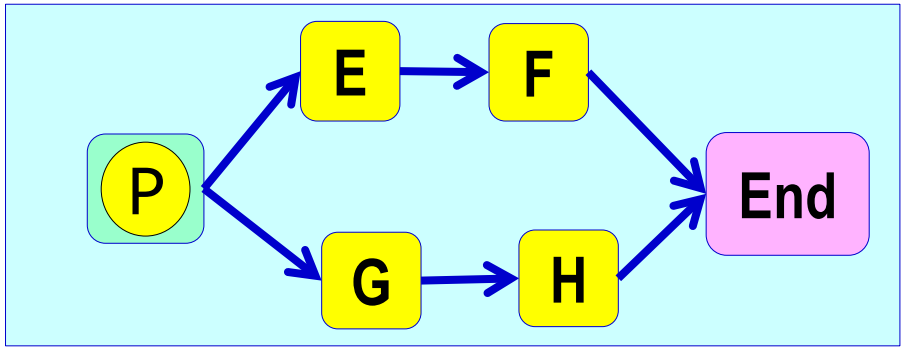
Reduction Example 1



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

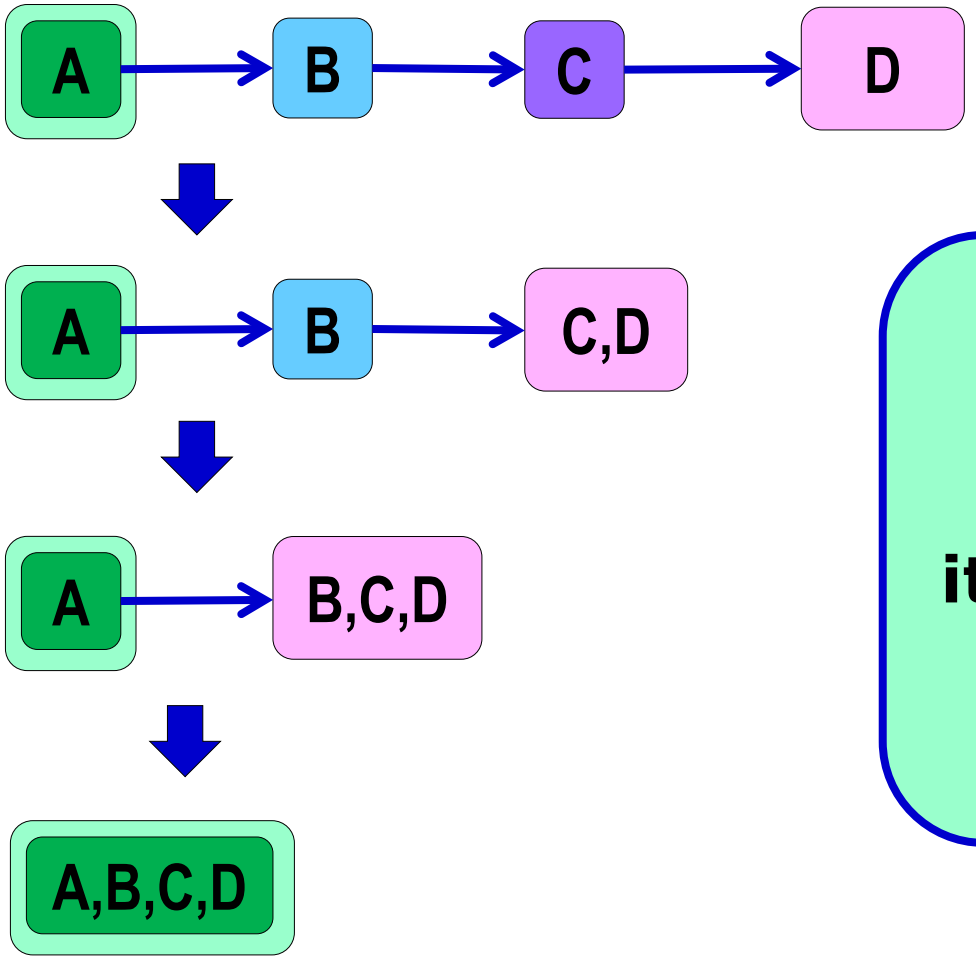


D calls procedure P



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:

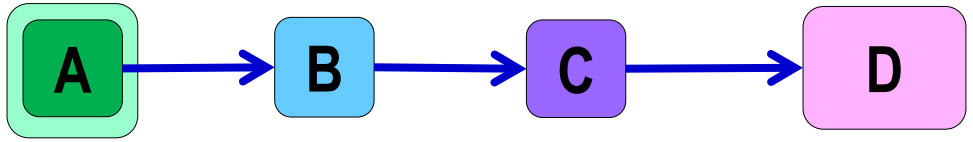
- The number of **linearly independent[1] paths** (basis paths) through the module or system
- If F is a flowgraph[2], then **v(F) = e – n + 2**
  - Where **e** is the number of edges (arcs)
  - And **n** is the number of nodes
- If a system consists of multiple flowgraphs that are not connected together,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

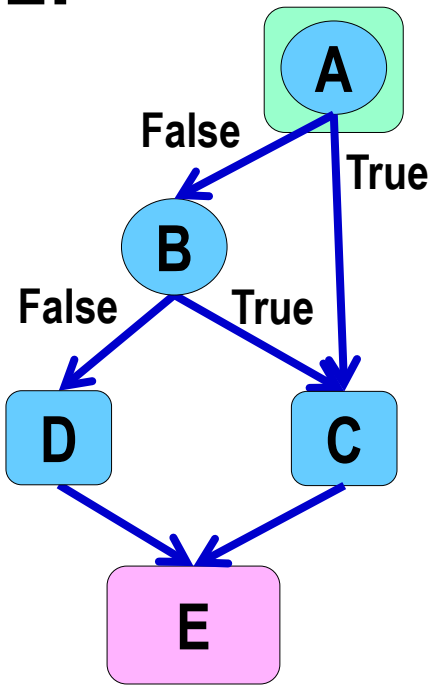
Examples of Cyclomatic Complexity

- **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 possible 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**

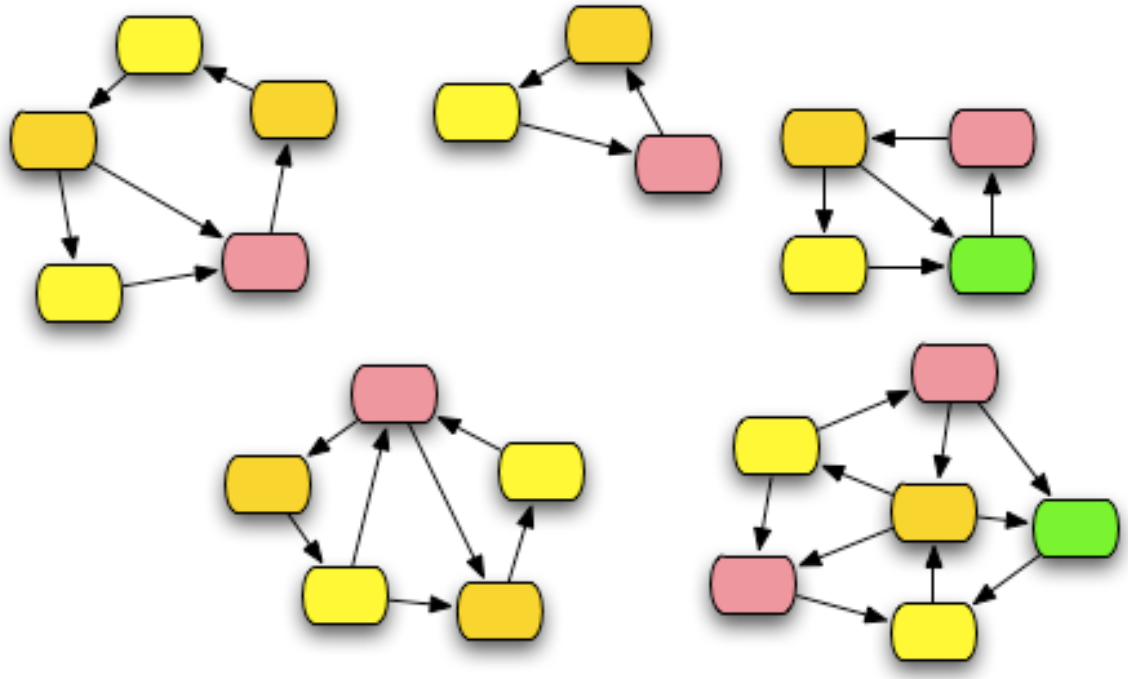
- This becomes more likely as you get more complex flowgraphs

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

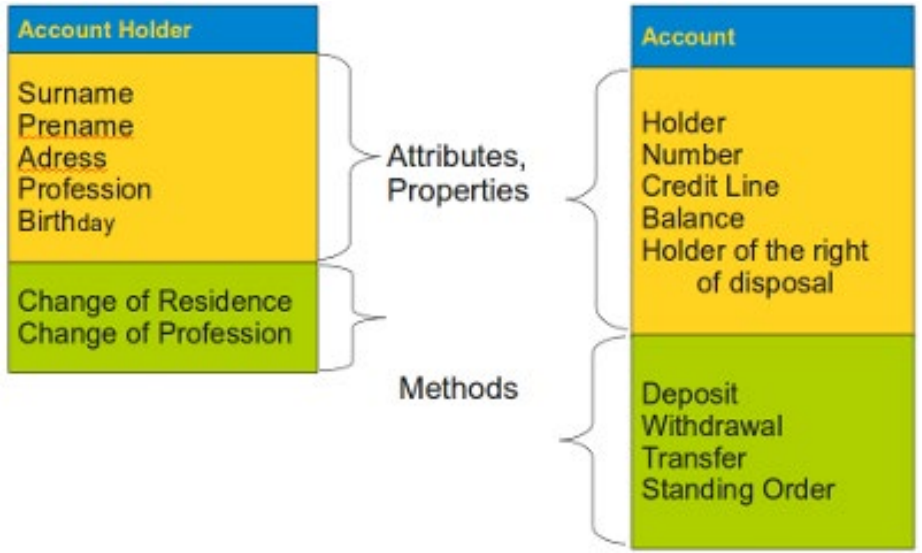
This graph has *five separate regions*, which are connected within themselves, but not to each other. Each region is called a *connected component*.



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?

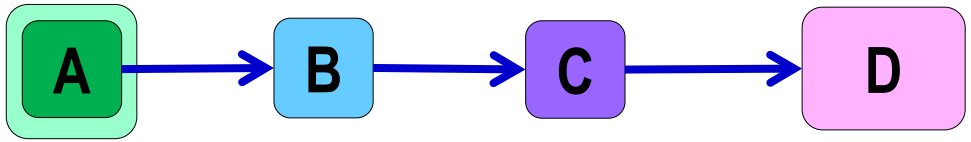
- **We could measure the cyclomatic complexity of a *system consisting of several separate modules***
- **In object oriented systems we could measure the cyclomatic complexity of a *class containing multiple methods***



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

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

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 4

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
(1 of 4)

- **Crosby, Philip B. *Quality is Free*, New York, McGraw-Hill, 1979.**
  - Practical guidance on how to reduce cost and improve quality by value-added analysis
- **Deming, W. Edwards, *Out of the Crisis*, MIT Press, 1986, ISBN: 0911379010**
  - Deming originated most of the ideas in value-added analysis
- **Eswaramurthi, K. and P. V. Mohanram, "**Value and Non-Value Added (VA/NVA) Activities – Analysis of a Inspection Process – A Case Study**", *International Journal of Engineering Research & Technoloogy*, V 2 #2 (February, 2013).**
  - An excellent case study applied to manufacturing.
- **Juran, J. M., *Juran on Leadership for Quality: An Executive Handbook*, The Free Press, 1989.**
  - One of the most frequently cited sources of info on this subject.

References
Part 2
(2 of 4)

- **Knox**, Presentation on Raytheon studies, as reported by Houston and Keats, ***Software Quality Matters***, ***vol 5, no 1*** (Spring, 1997), U. of Texas SW Quality Institute
  - Actual study done in 1993.
- **Ketkamon, Kanyakorn and Jirarat Teeravaraprug, "***Value and Non-value Added Analysis of Incoming Order Process*," ***Proceedings of the 2009 International Multiconference of Engineers and Computer Scientists (IMECS 2009), Vol II*, Hong Kong**
  - Applying Six Sigma principles to printed circuit board manufacturing.
- **http://www.brighthubpm.com/six-sigma/48826-the-importance-of-value-added-analysis-in-lean-six-sigma/**
  - This site discusses how processes are analyzed in "six sigma" programs to identify what is value-added

References
Part 2
(3 of 4)

- **Abran, A., et. al.,** *"Functional Complexity Measurement",* ***Proceedings, IWSM 2001 - International Workshop on Software Measurement.***
- **Adams, E.,** *"Optimizing preventive service of software products",* ***IBM Journal of Research and Development,*** vol 28, no. 1 *(1984), pp. 2-14*.
- **Chidamber, S. and Chris Kemerer,** ***A Metrics suite for Object Oriented Design*****,** MIT Sloan School of Management E53-315 (1993).
- **Dijkstra, Edsger,** *"GO TO Considered Harmful",* letter to the editor of ***Communications of the ACM*****,** March, 1968.
- **Fenton,** Chapter 9
- **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 2
(4 of 4)

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.

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***.

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

References
Part 3
(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 3
(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.*