Lecture Slides for Managing and Leading Software Projects

Chapter 7: Measuring and Controlling Work Products

developed by

Richard E. (Dick) Fairley, Ph.D.

to accompany the text

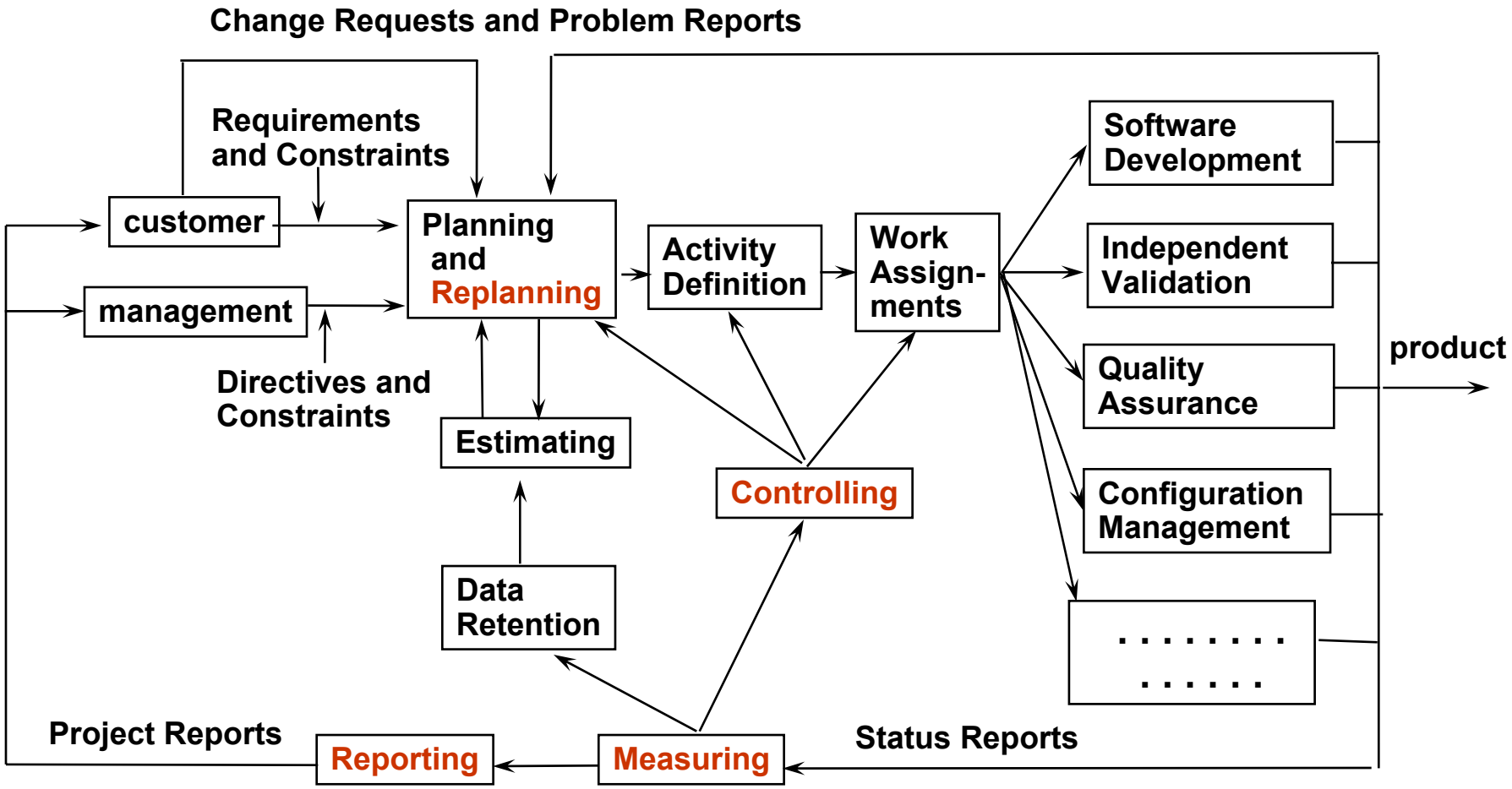
Managing and Leading Software Projects

published by Wiley, 2009

Four Kinds of Project Management Activities

1. Plan and Estimate
   - tasks, schedule, budget, resources
2. Measure and Control
   - work products (quantity & quality)
   - schedule, budget, resources, progress
3. Communicate and Coordinate
   - help people do their work activities
   - represent the project to others
4. Manage Risk
   - Identify and confront risk factors

A Workflow Model for Measuring and Controlling Software Projects



Chapter 7 Topics

- Measures and measurement
- Measuring product attributes
- Defects, defect measurement, and rework
- Reliability, availability, maintainability

Additional Sources of Information

- The four sets of standards and guidelines for managing project presented in this text; namely the CMMI-DEV-v1.2 process framework, the ISO/IEEE standard 12207, IEEE standard 1058, and the PMI Body of Knowledge address measurement and control of work products to varying degrees.
- Aspects of measurement and control in these documents are presented in Appendix 7A to Chapter 7.
- In addition, the Practical Software and Systems Measurement (PSM) approach is presented in the appendix
  - an overview of PSM is provided in section 8 of Chapter 7.

Objectives for Chapter 7

- After reading this chapter and completing the exercises you should understand:
  - measures and measurement scales
  - product measures for different kinds of work products
  - the role of configuration management in measurement and control of work products
  - the roles of inspections, walkthroughs, and developer testing
  - complexity measures for software
  - reliability and availability measures
  - the defect detection and repair process
  - ways to document and analyze defects and defect repairs
  - guidelines for choosing product measures
  - sources of standards and guidelines for measurement and control

Why Measure?

- There are several reasons you should measure various attributes of your software projects:
  - to provide frequent indicators of progress
    - or lack thereof,
  - to provide early warning of problems,
  - to permit analysis of trends for your project,
  - to allow estimates of the final cost and completion date of your project, and
  - to build a data repository of project histories for your organization.

Measuring and Controlling

- Measuring is concerned with
  1) collecting,
  2) validating, and
  3) analyzing
  project status information
- Controlling is concerned with applying corrective action when actual status does not conform to planned status
  - o status of the work products
    - quantity and quality of work products
  - o status of the development process
    - schedule, budget, resources, risk factors

What Should be Measured and Controlled?

- It is difficult to imagine a software project for which some level of measurement and control for each of the following attributes is not important to assure a successful outcome:
- Chapter 7:
  - product features: requirements implemented and demonstrated to work
  - quality attributes of the product: defects, reliability, availability, response time, throughput, and others as specified
- Chapter 8:
  - effort: amount of work expended for various work activities
  - schedule: achievement of objectively measured milestones
  - cost: expenditures for various kinds of resources, including effort
  - progress: work products completed, accepted, and baselined
- Chapter 9:
  - risk: status of risk factors and mitigation activities

A Bit of Measurement Theory (1)

- A *measurement* is a mapping from some real-world phenomenon to a scale of symbols having well-defined operations
  - for example, measurement of a computer program to determine program size or program complexity
- A *measure* is a number or symbol assigned to some attribute of a real-world phenomenon
  - for example, measuring program size as an integer number of lines-of-code
  - or measuring program size as Small, Medium, or Large

Some Measurement Systems

Reference	°F	°C	°Ce	°K
Freezing point of water	32	0	100	273.15
Human body temperature	98.6	37	63	310.15
Boiling Point of Water (at sea level)	212	100	0	373.15

Some Questions

- Why are there different systems for measuring the same phenomenon?
  - which one is correct?
- Are the following measurement systems equivalent?
  - 1, 3, 5
  - Low, Medium, High

A Bit of Measurement Theory (2)

- A *direct measure* is a number or symbol determined by direct examination of the phenomenon
  - for example, program size in lines-of-code
  - or hours worked by a programmer
- An *indirect measure* is a measure derived from a direct measure
  - for example, measuring productivity as
    - lines-of-code per programmer-day
  - or measuring defect density as
    - defects per thousand lines-of-code

Some Measurements and DIRECT Measures

MEASUREMENT	DIRECT MEASURES
Size	lines-of-code
Number of People	number of programmers
	number of testers
Progress	number of requirements baselined
	number of requirements changed
Resource Usage	CPU cycles used
	memory bytes used
Time	weeks-to-milestone
	computer response time
Quality	Number of defects
	computer response time

Some Measurements and INDIRECT Measures

MEASUREMENT	INDIRECT MEASURES
Size	function points
Productivity	lines-of-code per programmer-month
Production Rate	lines-of-code per month
Testing Rate	tests-conducted per staff-day
Defect Density	defects per thousand lines-of-code
Detection Effectiveness	number-of-defects-detected / total number of defects
Detection Efficiency	number-of-defects-detected per staff-hour
Requirements Stability	current number / initial number
Cost Performance Index	actual cost / budgeted amount

A Hierarchy of Measurement Scales

Scale	Characteristics
Nominal	Frequency distributions among measurement categories
Ordinal	Ordering within categories; arbitrary intervals among measures
Interval	Equal intervals among measures; arbitrarily determined zero element
Ratio	Equal intervals among measures; objectively determined zero element
Absolute	Similar to ratio but with uniqueness of measures

Some Measurement Scales

Scale Type	Permissible Operations
• Nominal (categorizing)	comparing* categories
examples: number of staff per functional area	
number of installations by country	
• Ordinal (ranking)	comparing individual items
examples: abilities of staff-members within a functional area	
complexity of programs (L, M, H)	
• Ratio (equal intervals)	above plus add, subtract, multiply, divide
examples: salaries of staff-members ($50K, $60K)	
lines of code in a module (50, 75, 100)	

* relational operators

Metrics Evaluation Criteria (1)

Each metrics used should be:

- Clearly defined
  - unambiguous units of measure (e.g., loc, cplx)
  - meaning understood by all involved parties
- Purposeful
  - reasons for and ways to be used clearly understood
- Accessible
  - easily obtained or computed
- Objective
  - not easily manipulated by subjective opinion
- Orthogonal
  - redundant factors avoided in different metrics

Metrics Evaluation Criteria (2)

- Parsimonious
  - metrics that have little impact on desired outcomes avoided
- Sensitivity
  - small changes in the metrics do not have large impacts on desired outcomes
- Predictive
  - cause and effect relations are evident

Cause and Effect Relationships

- A cause and effect relationship between two events, A and B, exist if and only if:
  - A precedes B in time
  - A and B are correlated (positively or negatively)
  - the relationship is not spurious

- A spurious relationship exists if A and B are both caused by another event C
  - for example: young children who watch Baby Einstein have fewer language skills.
  - Is Baby Einstein the cause?

Measuring Product Attributes

- Product attributes include:
  - requirements quantity & quality
  - software size
  - design complexity
  - software complexity
  - phase-dependent measures

Counting the Number of Requirements

- count the number of “shalls” in the requirements document
- count the number of *weighted* “shalls” in the requirements document, scale 0 – 10
- count the number of function points
- count the number of unique tasks the system must perform
- count the number of use cases
- count the number of scenarios
- count domain-specific factors
  - interrupts, control signals
  - windows, menus
  - etc

Desirable Attributes of Software Requirements (1)

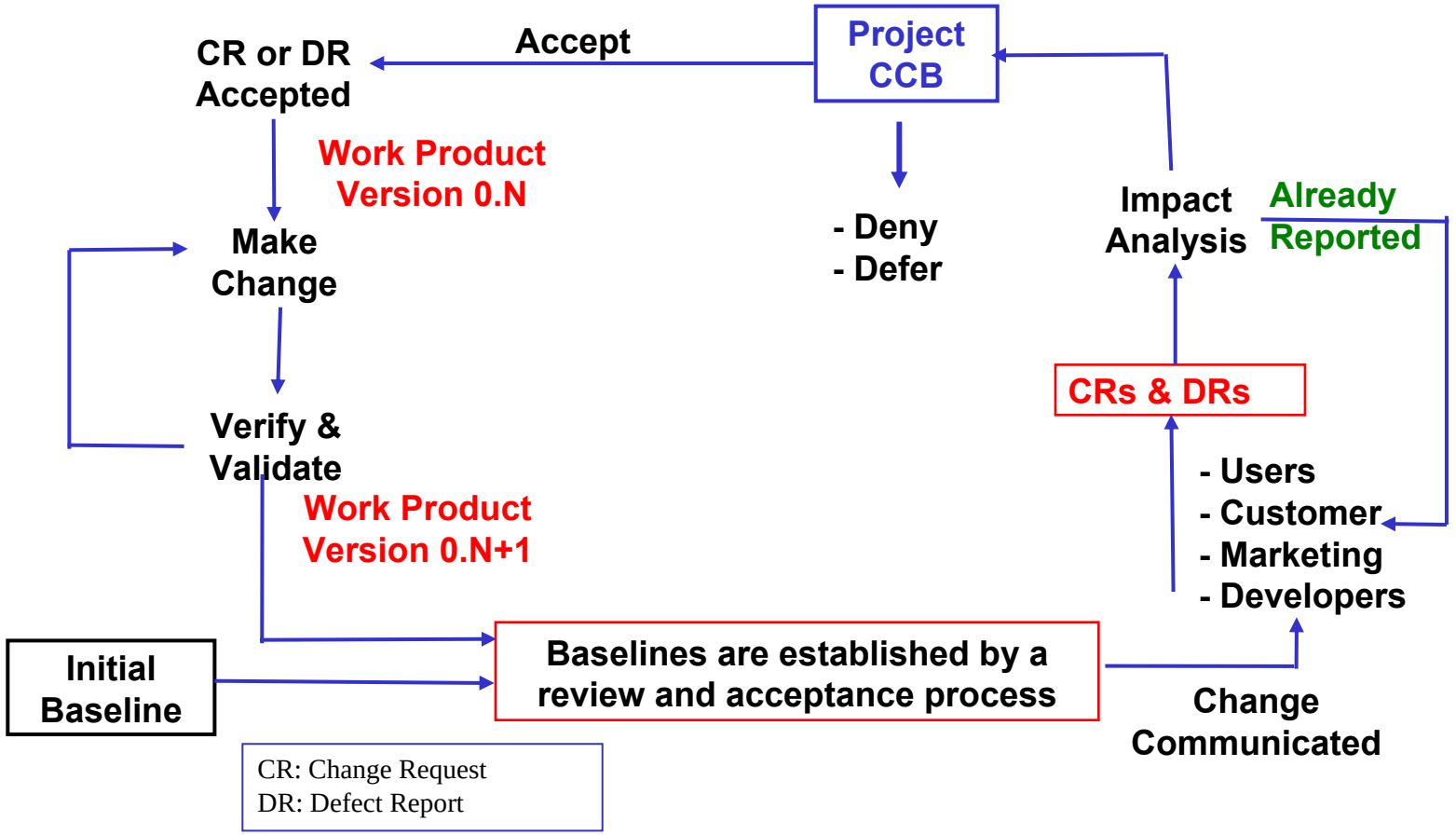
Software requirements should be:

- Comprehensive:
  - functions, performance, capacities, constraints, interfaces, and quality attributes are specified
- Detailed:
  - hidden complexities and risk factors are exposed; experts can make estimates
- Prioritized:
  - essential, desirable, and optional categories specified
  - prioritized within categories
- Stable:
  - stable does not mean static, but within statistical control limits

Desirable Attributes of Software Requirements (2)

- Testable:
  - valid requirements can be validated by objective procedures
  - design goals are separately identified
- Feasible:
  - the requirements are within the state of the art – for those who will implement the product
- Traced:
  - each requirement is traced to a user need, customer expectation, or system-level requirement

Change Control



Measuring and controlling work products requires a change control process

Duties of a CCB

- Duties of a CCB include:
  - review CRs, DRs, and impact analysis reports
  - accept, deny, or defer CRs and DRs
  - track progress on accepted CRs and DRs to closure
- Members of a CCB must have the *authority, resources, and will* to make decisions

CCBs and change control mechanisms are an important element of all development activities, not just requirements engineering

Requirements Verification

- Requirements verification is the process of determining the degree to which requirements satisfy the conditions imposed by other work products and work processes
  - i.e., are the operational requirements complete, correct, and consistent wrt to users' needs, customer's expectations, and acquirer's conditions
  - are the technical specifications correct, complete, and consistent wrt the operational requirements

Requirements Verification Techniques

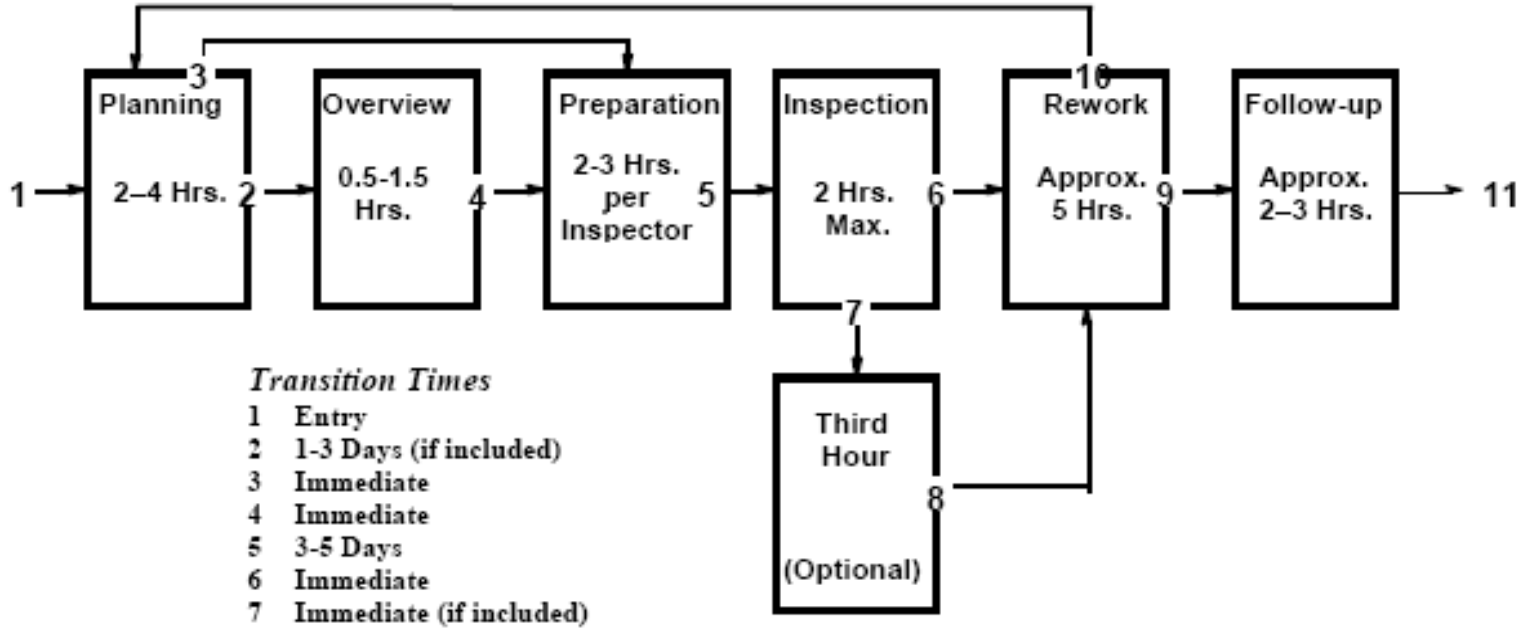
- Techniques for verifying requirements include:
  - inspections
  - traceability
  - prototyping
  - analysis
  - preparation of validation tests

Kinds and Purposes of Reviews

Kind of review	Purpose
Inspection	To find defects and document discovery and the repair processes
Walkthrough	To review work products and communicate issues
Team	To review progress and plan work activities
Project	To review progress, process and product constraints, and confront risk factors
Customer	To review progress, constraints, and risk factors
Department	To review portfolios of projects, identify and confront common risk factors, assess status of budgets and customers, and confirm/revise mission and vision statements

The Inspection Process

Flow Diagram for the Inspection Process



Transition Times

1 Entry
2 1-3 Days (if included)
3 Immediate
4 Immediate
5 3-5 Days
6 Immediate
7 Immediate (if included)
8 Immediate
9/10 1 Wk Min. after Inspection
11 Exit
Two Wk Max. process

Managing the Inspection Process

- With respect to inspections, your job as project manager is to:
  - provide training in the inspection process, if needed;
  - allow adequate time in the schedule to prepare for and conduct the inspections;
  - review the results
  - provide feedback to inspection participants as to the efficiency and effectiveness of inspections; and
  - make improvements in your inspection and development processes as indicated by the reported trends.

In particular, the inspection process requires that you, as project manager, schedule sufficient time for the preparation, meeting, rework, and follow-up activities of inspections.

Distinctions Between Inspections and Walkthroughs

Inspections	**Walkthroughs**
Purpose: to find defects	Purpose: to communicate
Training of participants	No training of participants
Assigned roles: moderator, reader, recorder, developer	No assigned roles
Names of participants recorded	Names of participants (usually) not recorded
Developer does not present	Developer typically presents
Record keeping	No record keeping
Analysis of inspection results	No analysis of the walkthrough results

inspections are used to find defects
walkthroughs are used to communicate

A Note

- Inspections can be applied to all work products of software engineering:
  - requirements
  - design
  - code
  - test plan
  - project plan
  - user documentation
  - etc
- Inspection procedures are described in Appendix 7B to Chapter 7 of the textbook

A Requirements-Source Traceability Matrix

- A requirements-source traceability matrix is used to identify the source(s) of operational requirements:

Source:		Customer	User Gp1	User Gp2	Marketing
Rqmt:	[1]	X			
	[2]		X	X	X
	[3]		X		
	[4]		X		
	[5]?				
	[6]?				
	[7]	X			
	[8]				X
	[9]	X		X	
	[10]			X	
	[11]	X			
	[12]	X		X	
	[13]				X
	[14]		X		

Activity-Dependent Measures (1)

- Suppose our processes for software development and modification include the following types of work activities:
    - analysis
    - architectural design
    - detailed design
    - coding
    - unit testing
    - integration testing
    - system testing
    - acceptance testing

Each type of work activity provides opportunities to measure the process and the work products

Activity-Dependent Measures (2)

- The following tables in Chapter 7 list measures that can be used to measure and control various kinds of work products:
  - Table 7.6 Requirements
  - Table 7.9 Architecture
  - Table 7.10 Implementation
  - Table 7.15 Integration and Verification
  - Table 7.16 Verification and Validation

Some Measures for Requirements

- Number of User Scenarios Developed
- Status of Prototyping Efforts
- Number of Requirements Baselined
- Number of Baselined Rqmts Changed
- Number of Requirements-Based Test Scenarios Generated
- Number of Design Goals
- Status of Traceability Matrices
  - Operational Requirements to Primary Specifications
  - Requirements to Test Cases

Some Measures for Architecture

- Updates to Requirements Status Indicators
- Number of Requirements traced to design components
- Number of quality scenarios completed
- Number of Modules Specified
- Number of Interfaces Baselined / Changed
- Number of Design-Based Test Cases Generated
- Traceability Matrix
  - Requirements to Design
  - Design to Test Cases

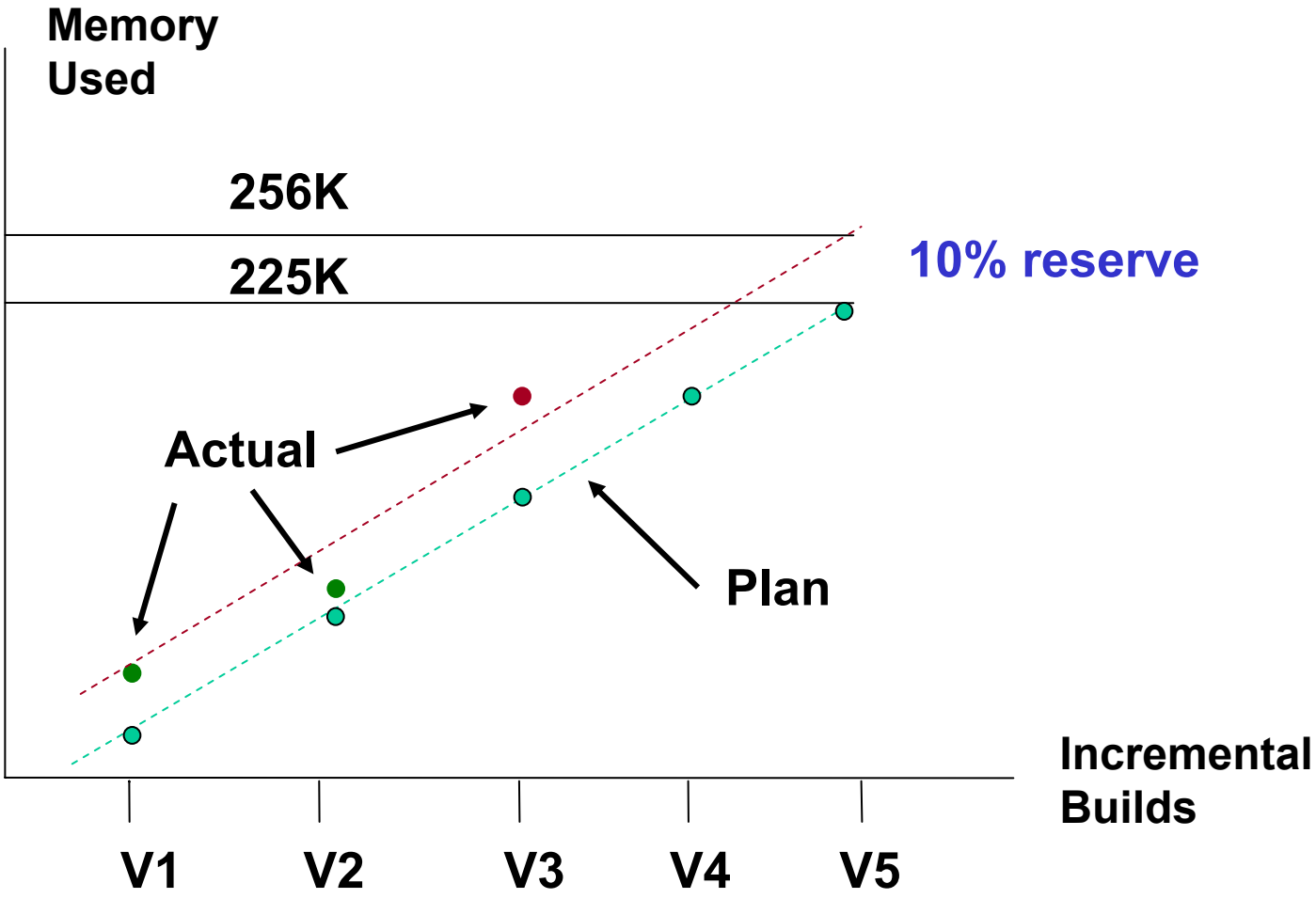
Some Measures for Detailed Design

- Updates to Requirements Status Indicators
- Updates to Arch. Design Status Indicators
- Number of Modules Designed
- Number of Modules Baselined / Changed
- Number of Unit Test Cases Generated
- Status of Traceability Matrices
  - Modules to Architecture
  - Unit Test Cases planned for each Modules
  - Unit Test Cases Completed for each module

Some Measures for Implementation

- Updates to Rqmts. and Design Status
- Number of Coded Modules Baselined & Number Changed
- Currently in Revision
- Number of Modules in Revision from Last Reporting Period
- Traceability to Detailed Design & Test Cases Completed
- Forecast for Completion of Code and Unit Test
- Discovered Defects
- Code Complexity

An Example: Tracking Memory Usage using the TPM* Approach



* TPM (Technical Performance Measurement) is presented in Appendix 7A of Chapter 7

Some Measures for Integration and Test

- Updates to Earlier Status Indicators
- Number of Modules Successfully Integrated
- Number of SPRs opened, closed, trend
- Discovered Defects per Thousand Lines of Code
- Defect Density by Total Size, Module, and Subsystem
- Traceability to Design-Based Test Cases Completed
- Forecast for Completion of Integration (incremental vs waterfall)

SPR: Software Problem Report

Some Measures for System Test

- Number of System Tests Passed / Number Failed
- Traceability from System Tests to Rqmts
- Number of New SPRs in this Period
- Number of Open SPRs by Severity Level
- Number of Open SPRs > 30 Days
- SPR Density per KLOC
- Forecast for Completion of System Testing

*KLOC: Thousand Lines of Code

Some Measures for System-level Verification and Validation

- Number of scenario-based tests and demos executed successfully
- Number of scenario-based tests and demos failed
- Number of quantitative system tests passed
- Number of quantitative system tests failed
- Number of new SPRs* in this reporting period
- Number of open SPRs by severity level
- Number of SPRs open more than X days
- SPR density per size unit
- Forecast for completion of system testing
- *SPR: Software Problem Report

Measuring Product Attributes

- Product attributes include:
  - requirements quantity & quality
  - software size
  - complexity

Ways to Measure Size

- Lines of code
- Function points
- Objects and relationships
- Windows, menus, buttons
- Values, sensors, alarms
- Interrupts, priority levels, responses

Software size measurement is covered in Chapter 6 of the text

Measuring Software Complexity (1)

- Software complexity is somewhat subjective
  - it depends on our familiarity with the application domain
- Complexity measures include:
  - Boehm's COCOMO complexity adjustment factor
  - McCabe's cyclomatic complexity
  - Constantine's coupling and cohesion

Measuring Software Complexity (2)

- Boehm* provides a five-element measure of product complexity (CPLX):
  - complexity of the control operations
  - complexity of the computations
  - complexity of data management operations
  - complexity of the device-dependent operations
  - Complexity of user interface management operations
- Boehm uses a complexity table to select a complexity adjustment factor to increase or decrease the estimated effort and schedule for a project

$$0.7 <= CAF <= 1.65$$

* B. Boehm et al, "Software Cost Estimation with COCOMO II," Prentice-Hall, 2000

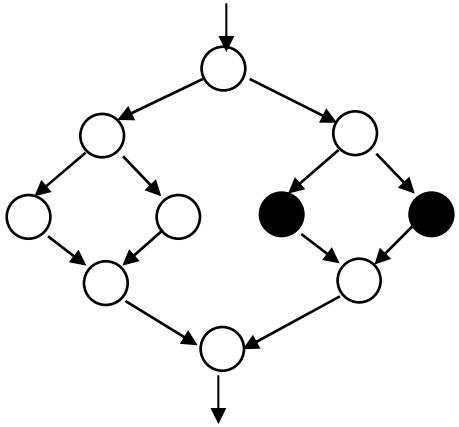
Cyclomatic Complexity

- The structural complexity of a software module is determined by computing the cyclomatic complexity of the module's control-flow graph
- The cyclomatic complexity, C, of a module, M, is computed as:

$$C(M) = E - N + 2$$

where N is the number of nodes in the control-flow graph and E is the number of connections between nodes

- For example:



C(M) = 12 – 10 + 2 = 4

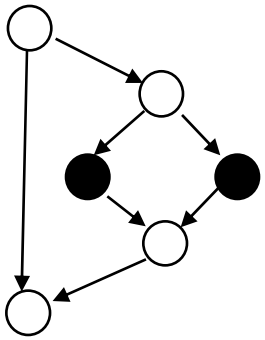
McCabe's Rule of Thumb

- C(M) > 10 is too complex for a single module
- Too complex means:
  - too hard to document
  - too hard to understand
  - too hard to test
  - too hard to modify

Software Design Complexity (1)

- The structural complexity of a collection of software modules is called the Design Complexity
- Design Complexity is computed in two steps:

  (1) Compute the Design Complexity of each module, by considering only the nodes that have interfaces to other modules:



DC(M) = 7 – 6 + 2 = 3

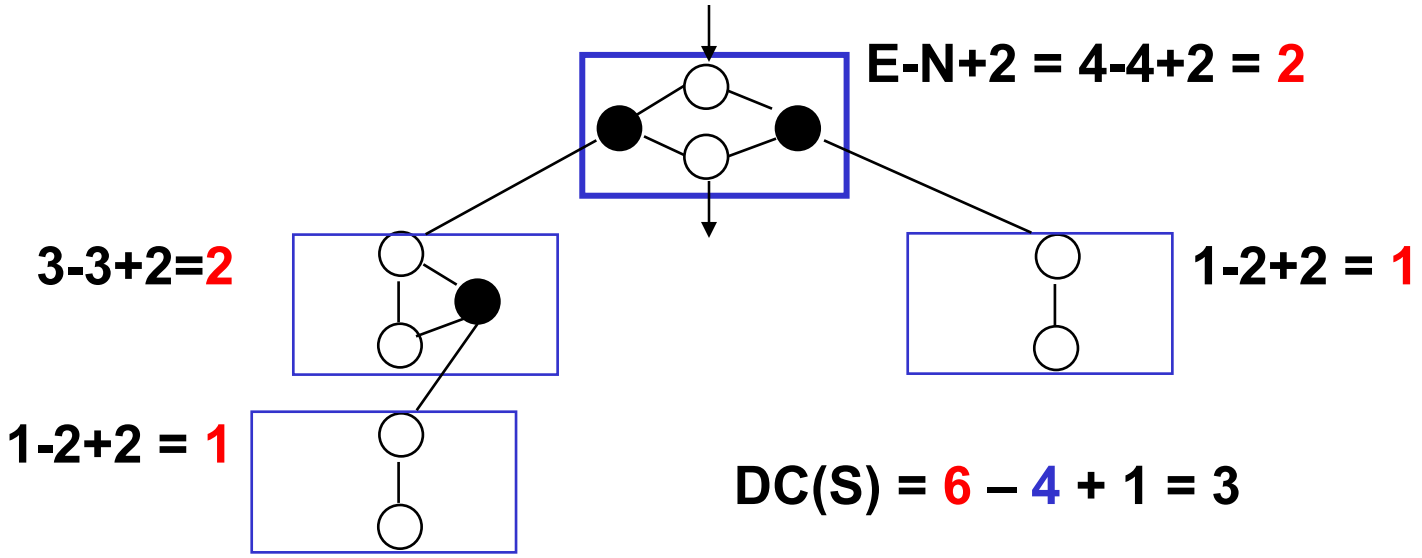
Software Design Complexity (2)

- Design Complexity is computed in two steps:

  (2) Compute the Design Complexity of the collection of modules, S, as:

$$DC(S) = \Sigma\, DC(M_j) - N + 1$$

  where $DC(M_j)$ is the design complexity of module j and N is the number of modules



Software Entropy

- Lehman's 2nd Law of Software Evolution
  - as a software system evolves, its structure deteriorates unless work is done to maintain the structure
  - also know as the

    Law of Software Entropy
- Measuring *cyclomatic complexity* when the system is modified can tell us when to do some work to improve the system structure
  - improving the system structure is one reason to do

    Preventative Maintenance

Coupling and Cohesion Complexity Measures

- Coupling is concerned with the relationships *among* the software components
- Cohesion is concerned with the relationships *within* software components

software should have *low coupling* and *high cohesion* to control software complexity

MEASURING SOFTWARE COUPLING

- Coupling is the measure of the strength of association between software components
- Coupling is a way of evaluating the design partitioning of a system
- Measure of coupling (ordered best to worse)
  1. Message – request for service
  2. Data coupling – communicate by parameters
  3. Stamp coupling – refer to the same data structure
  4. Control coupling – control information is passed
  5. Common coupling – refer to the same global data area
  6. Content coupling – refer to the insides of another software component
- Strive for loose coupling.

The best coupling methods are message and data coupling

SOFTWARE COHESION

- Cohesion is a way of measuring the binding of the elements within a module
- Measures of cohesion (strongest to weakest)
  1. Object binding – all elements support a single concept
  2. Functional binding – all elements related to a single function
  3. Sequential binding – output of one element is input to another
  4. Communication binding – use same set of input or output data
  5. Information binding – related by data
  6. Temporal binding – related by time
  7. Coincidental binding – no meaningful relationship
- Strive for strong cohesion

- Object, and functional provide the best binding; they are describable by specifying inputs and outputs
- and, *they have no side effects*

Coupling and Cohesion

- Coupling is a measure of the strength of the interconnections among code modules
- Cohesion is a measure of the strength of the interactions within a code module
- Code that is low in complexity has weak coupling among the modules and strong cohesion within the modules
- This makes the code:
  - easy to understand
  - easy to document
  - easy to test
  - easy to modify

  because strong cohesion encapsulates functionality and weak coupling limits the side effects of each module

Project Attributes to be Measured and Controlled

- Attributes to be measured and controlled include:
  - product attributes
  - quality factors
  - rework
  - effort, cost, schedule, and progress
  - risk factors

Quality Requirements

- Quality requirements for software include:
  - safety: will do no harm
  - security: will protect information
  - reliability: will not fail more than specified
  - availability: quickly repaired after failure
  - usability: can be used efficiently and effectively
  - maintainability: easy to modify
- Different systems have different kinds and different levels of quality requirements

Defects are, by definition, the root cause of poor software quality

Failures, Defects, and Errors

- A failure occurs when software does not satisfy its requirements, needs, or expectations when operated as intended in its intended environment
- A failure results from one or more defects
- Defects are the result of human errors
  - errors of omission
  - errors of commission

Severity Levels of Failure for Safety-Critical Software

- Some severity levels of failures:
- Catastrophic: failure results in loss of several lives and / or significant loss of property or information
- Critical: failure results in loss of one life and / or serious loss of property of information
- Severe: failure results in serious injury to one or more persons and / or some loss of property or information
- Marginal: failure results in minor injury to one or more persons and / or performance degradation and / or temporary loss of system availability
- Minor: failure results in unscheduled maintenance

What is a Defect?

- Broad definition: a defect is anything that makes the customer or user unhappy (six sigma definition)
  - in the software code, the installation procedures, the supporting materials, the help line, and so forth
- Intermediate definition: defects, when encountered in operation, result in failures that cause a system to perform incorrect operations or to behave in an unexpected manner
- Narrow definition: defects, when encountered in operation, cause failures that result in departures from specified requirements
- Each organization must decide what constitutes a defect
  - is “hard to learn” a software defect?

Why Do Humans Make Mistakes? (1)

- Breakdowns in communication and coordination are the major reason for human mistakes in software engineering
  - "I didn't receive the necessary information"
  - "The information I received was out of date"
  - "The information changed and I wasn't told"
  - "I misinterpreted the correct information"
  - "I didn't know I was supposed to do that part"
  - "I thought I was supposed to do that part"

human mistakes are the cause of software failures

Why Do Humans Make Mistakes? (2)

- Other reasons humans make mistakes in software engineering include lack of training, tools, or experience
  - o "I didn't know how to do that job"
  - o "I didn't have the correct tool for the job"
  - o "I have never done that job before"
- Human fallibility is another reason for human mistakes:
    - "I was sick, tired, troubled, . . ."
- Other reasons?

Defects and Rework

- Defects during software development result in avoidable rework
- Three kinds of rework:
  1. Evolutionary
  2. Retrospective
  3. Corrective
- Evolutionary rework adds value to the evolving product
  - and must be accompanied by adjustments to schedule and resources, as necessary
- Retrospective rework and corrective rework are avoidable
  - Retrospective: should have been done earlier
  - Corrective: fixing defects
    - Omission defects & commission defects

All 3 kinds of rework are tracked using a version control system

Iterative Rework*

Table 1. An iterative rework taxonomy.

Type of rework	Characteristics	Good, bad, or ugly?
Evolutionary	Work performed on a previous version of an evolving software product or system to enhance and add value to it	Good—if it adds value without violating a cost or schedule constraint Bad—if it violates a cost or schedule constraint Ugly—if it smacks of "gold plating"
Avoidable Retrospective	Work performed on a previous version of an evolving software product or system that devlopers should have performed previously	Good—small amounts are inevitable; better now than later Bad—if it occurs routinely Ugly—if excessive, it indicates a need to revise work processes
Avoidable Corrective	Work performed to fix defects in the current and and previous versions of an evolving software product or system	Good—if total rework is within control limits Bad—if it results in patterns of special-cause effects Ugly—if it results in an out-of-control development process

* from the paper "Iterative Rework: The Good, Bad, and the Ugly" by R. Fairley and M. Willshire, *IEEE Computer*, September, 2005

Tracking Rework with Control Charts*

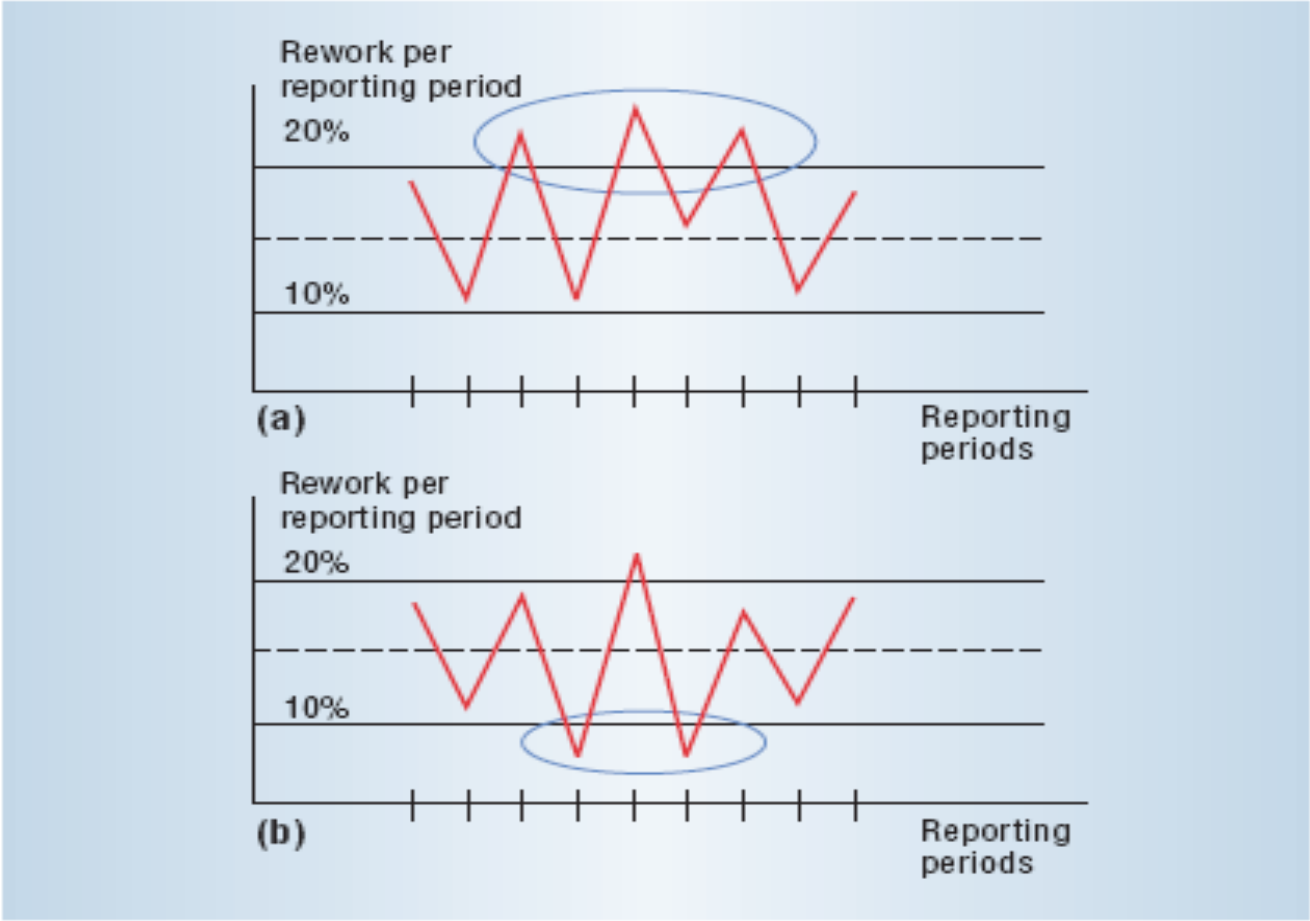


Figure 4. Rework control charts illustrating (a) excessive rework and (b) insufficient rework.

* from the paper “Iterative Rework: The Good, Bad, and the Ugly” by R. Fairley and M. Willshire, *IEEE Computer*, September, 2005

When Do We Start Counting Defects?

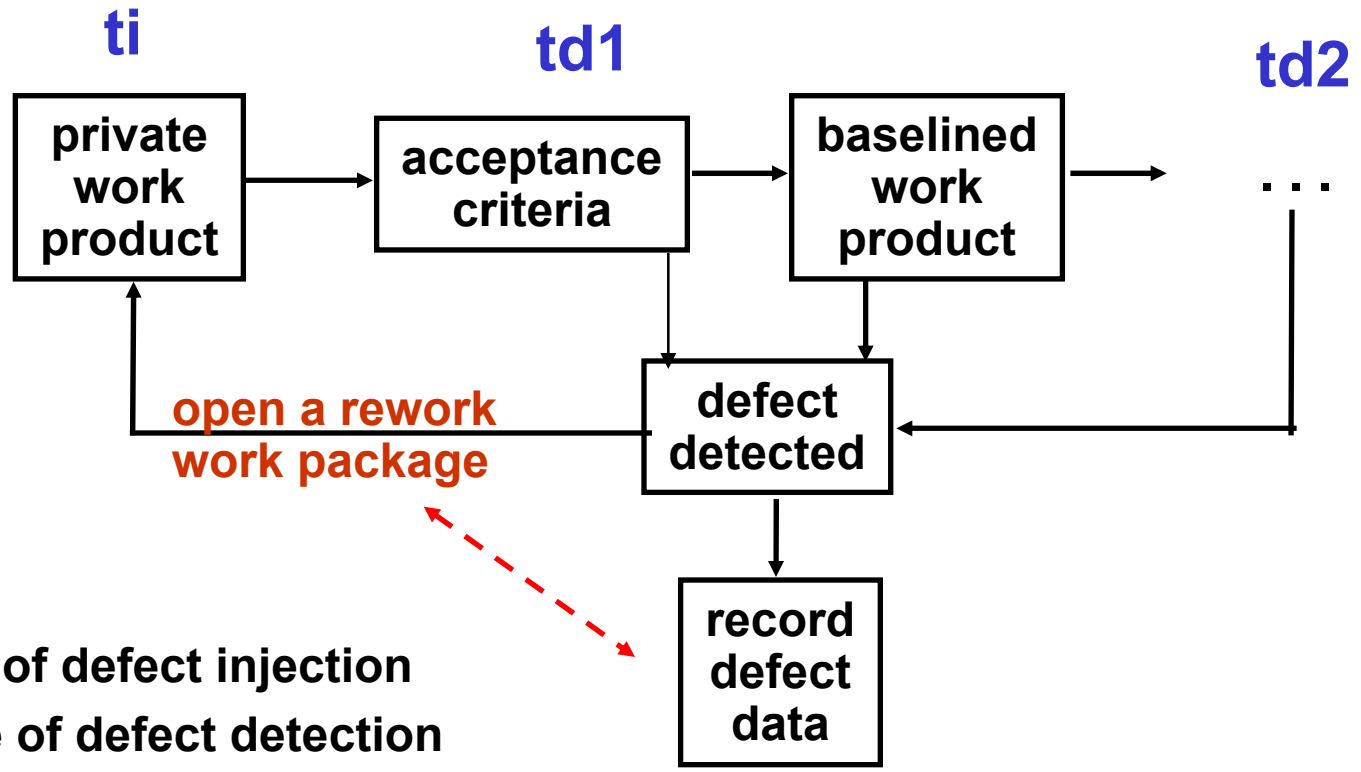
- We do we start counting defects:
  - In requirements?
  - In design?
  - In coding?
  - In testing?
  - In maintenance?

Rework of ***any baselined work product*** that must be changed because of a mistake should be counted as a defect

Defects and Quality

- A defect that produces a failure results in loss of product quality, which may be manifest as a:
    - safety failure
    - security failure
    - reliability failure
    - availability failure
    - usability failure
    - maintainability failure
- Detection and prevention of defects are thus the underlying mechanisms of software quality control
    - o Verification and Validation (V&V) are the mechanisms of **defect detection**
    - o Process Improvement is the mechanism of **defect prevention**

A Defect Tracking Model Based on Work Packages



ti: time of defect injection

td_j: time of defect detection

defect lifetime = ti - tdj

Rework Work Packages

A rework work package specifies:

- Original work package identifier (modified by type of rework)
  - Type of rework

    EV: evolutionary, RE: retrospective, or CO: corrective
- Rework description
- Date started
- Date finished
- Effort expended
- Other work products modified
- Re-acceptance criteria applied to the work product(s)
- Corrective rework:
  - type of defect (requirements, design, code, . .)
  - phase when discovered
  - brief description of cause

A Re-Work Package Example

Task : 3.2.2.3CO Rework TRANSACTION_PROCESSOR code

Rework description: Corrected the input parameters in Trans. Processor

Date started: 6/15/05

Date finished: 6/29/05

Effort expended: 7 work hours

Other work products modified: interface design & integration test library

Acceptance criteria: re-ran and passed all interface tests

CORRECTIVE REWORK ONLY:

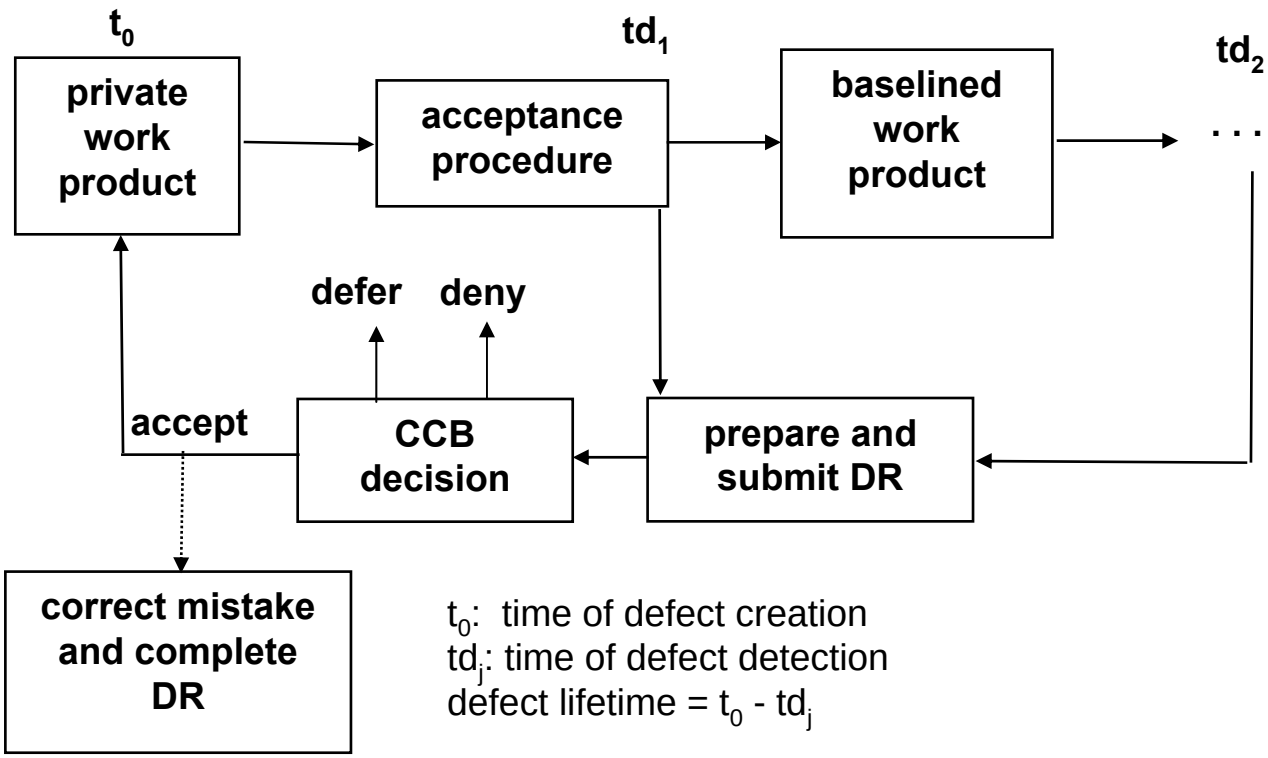
Type of defect: rqmts___design_X_code___test___

When discovered: rqmts___design___code___test_X_ops___

Brief description of cause:

interface specification for the Transaction Processor was incorrectly stated in the interface design spec

A Defect Tracking Model Based on Defect Reports



t_0: time of defect creation
td_j: time of defect detection
defect lifetime = t_0 - td_j

Tracking Defects by Work Phase

defect kind: \ phase in which defects found:	Rqmts	Design	Code	Test	Ops	Totals
Rqmts	RDr	RDd	RDc	RDt	RDo	RDt
Design		DDd	DDc	DDt	DDo	DDt
Code			CDc	CDt	CDo	CDt
Test				TDt	TDo	TDt
Totals	RSt	DSt	CSt	TSt	OSt	TOTAL

legend: RDd: requirements defects detected in the design stage
DDd: design defects detected in the design stage
DSt: total defects detected in the design stage
DDt: total detected design defects

An Example of Defect Tracking

defect kind: \ phase in which defects found:	Rqmts	Design	Imple.	Verif.	Valid.	Ops	Totals
Rqmts	50	25	13	6	3	3	100
Design		60	30	15	8	7	120
Imple.			80	40	20	10	150
Verif.				6	3	0	9
Valid.					7	0	7
Totals:	50	85	123	67	41	20	386

Some Defect Calculations

- percent of requirements defects detected during the requirements phase:

$$(RDr \times 100) / RDt$$

- percent of design defects that escape the design phase:

$$[\ 1 - (DDd / DDt)\ ] \times 100$$

- percent of total defects detected by users:

$$(OSt \times 100) / TOTAL$$

Corrective Rework for the Example

Work hours to correct defects:

defect kind:	Rqmts	Design	Imple.	Verif.	Valid.	Ops	Totals
Rqmts	50	100	130	200	250	300	1030
Design		60	90	150	225	200	725
Imple.			80	220	200	150	650
Verif.				6	10	0	16
Valid.					7	0	7
Totals:	50	160	300	576	692	650	2428

Defect Reporting in Each Reporting Period

Project: Date:	Total number	# newly reported	# carried forward
# of Major defects			
# of Minor defects			
# of Inconvenient defects			

Reporting of Defect Aging by Severity Level

Project: Date:	open < 1 day	open < 3 days	open < 5 days	open > 5 days
# of Major defects				
# of Minor defects				
# of inconven-ience defects				

Defect Containment

- Defect containment is calculated by determining the percent of defects that are detected during the work activity that generates the work product
  - also called “defect detection effectiveness”
- The goal of defect containment should be

  > 75% defect detection at each stage of our work
- With the goal of 75% defect detection at each stage, defect containment for the Space Shuttle on-board software, prior to product release, is ~ 99%

cumulative defect containment for many organizations is < 90%

A NASA Space Shuttle Example

For one system, 123 total defects were detected*

- 92 detected during requirements and design inspections
    - 74.8 percent effective
- 31 defects remained after inspections
    - 22 detected during development testing
        - 71 percent effective
- combined inspection + development testing
    - 92.7 percent effective

9 defects remained after inspections and development testing

- 7 detected during independent testing
    - 77.8 percent effective

2 of 123 defects released to, and found by, IV&V

- 98.4 percent combined effectiveness of inspection and testing

* cumulative for system in use more than one year

Effort versus Defect Reduction*

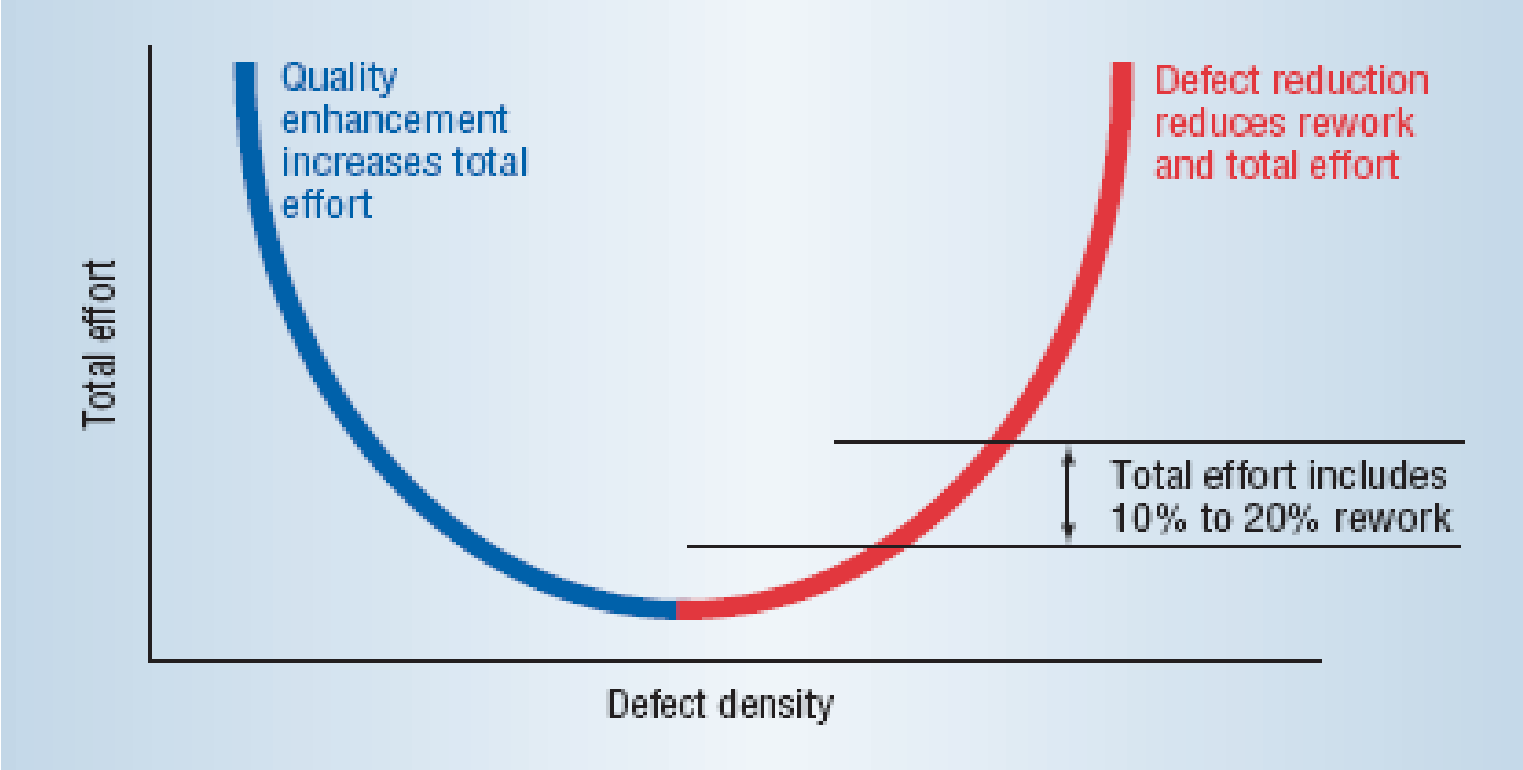


Figure 5. Effort saved by reducing defects, thus reducing avoidable rework.

* from the paper “Iterative Rework: The Good, Bad, and the Ugly” by R. Fairley and M. Willshire, *IEEE Computer*, September, 2005

Measuring Reliability, Availability, and Maintainability

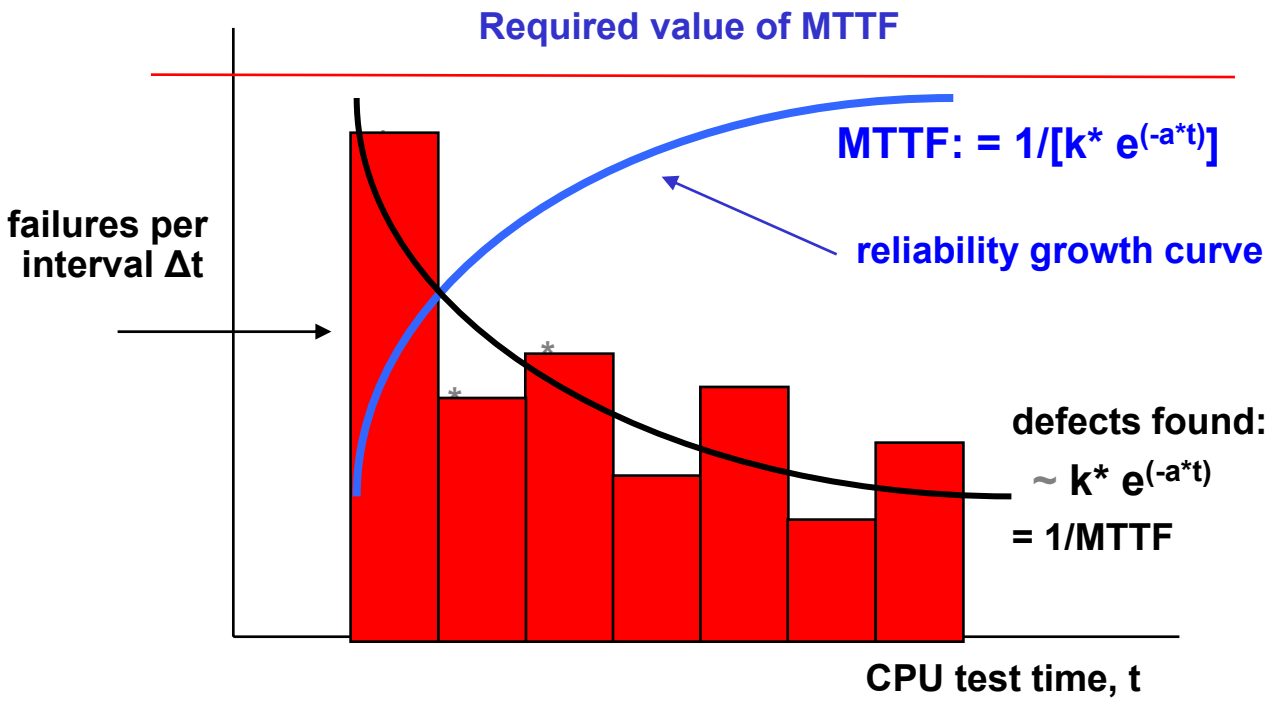
- *Reliability* is measured as the probability a system will operate without failure under stated conditions for a given time period
- *Availability* is the probability a system will be operational at a given point in time
- *Maintainability* is the probability a maintenance activity can be accomplished within a given period of time using specified procedures, tools, and techniques

Reliability

- Reliability has three attributes
  1. the system will perform specified functions
  2. under stated conditions
  3. for a specified period of time
- Reliability is measured as Mean Time To Failure (MTTF)
  - failure means a system does not perform specified function under stated conditions for a specified period of time
- The reciprocal of MTTF is failures per unit time interval
  - if MTTF = 4 CPU hours of system testing time, we should detect an average of 1 failure each 4 CPU hours of testing

    i.e., 0.25 failures per CPU test-hour

Reliability Growth Models



- During system testing, the defects found per time interval typically display an exponential decay of the form: $k^* e^{(-a^*t)}$
- MTTF is the reciprocal: $1/ [k * e^{(-a^*t)}]$

Availability

- Availability is the probability a system will be operational at a given point in time
- Availability is calculated in terms of Mean Time To Failure (MTTF) plus Mean Time To Repair (MTTR)

$$MTBF = MTTF + MTTR$$

- Availability, A:

$$A = MTTF / (MTTF + MTTR)$$

- For example, if MTTF = 8 hrs and MTTR = 2 hrs

$$A = 8 / (8 + 2) = 8 / 10 = 0.8$$

it is 80% probable that the system will be available at any particular time

Maintainability

- Maintainability is the probability a maintenance activity can be accomplished within a given period of time, T, using specified procedures, tools, and techniques
- Maintainability is based on Mean Time To Repair

$$M = T / MTTR$$

- If MTTR = 2 hrs, the probability a maintenance activity can be accomplished in 1 hour is:

$$M = 1 / 2 = 0.5 \ (50\%)$$

Using GQM* to Choose Product Measures (1)

A GQM Example:

- Goal: reduce defects during software development
- Questions:
  - How many defects are introduced during software development?
  - What percent of total defects found during software development are introduced in each phase of software development?
  - What percent of total defects found during software development are found in each phase of software development?
  - What kinds of defects are found in each phase of software development?
  - By what percent should defects found during software development be reduced in the next 12 months?

* GQM: Goals, Questions, Metrics

Using GQM* to Choose Product Measures (2)

- Metrics:
  - Total number of defects found during software development
  - Percent of total defects introduced in each phase of software development
  - Percent of total defects found in each phase of software development
  - Kinds of defects found in each phase of software development

The Main Points of Chapter 7 (1)

- periodic measurement of product attributes permits comparison of actual status to planned status
- control (corrective action) is exerted when actual status differs from planned status by more than an acceptable amount
- product and process measures are, or should be, a byproduct of the procedures, tools, and techniques used to develop software; if not the development process must be modified
- a measure is a mapping from a phenomenon of interest to a symbol
- different measurement scales permit different kinds of operations on the measures
- each work product should be verified and validated; in addition, specific attributes of each kind of work product can be measured
- version control of work product baselines is necessary for measurement and control of work products

The Main Points of Chapter 7 (2)

- inspections are the most cost-effective technique known to find defects in work products, especially in requirements and design when they are easily corrected
- inspections are used to find defects; walkthroughs are used to communicate technical issues
- software that is hard to understand, hard to document, hard to verify and validate, and hard to modify is too complex
- cyclomatic complexity, the COCOMO CPLX cost driver, and coupling and cohesion are three measures for software complexity
- a reliability rating is the probability that a system will not fail to perform its intended functions within its intended environment for a stated period of time

The Main Points of Chapter 7 (3)

- an availability rating is the probability a system will be available for use when needed; availability is measured as the ratio of MTTF / (MTTR+MTTR)
- defects are the result of human mistakes; defects in an operational system cause departures from specified or expected behavior or results
- software failures result when defects are detected during the operation of a system by its intended users within its intended environment
- systematic record keeping of the defect detection and repair process permits analysis of defect containment and escape during the various phases of software development
- the time, effort, and cost of measuring and controlling work products, like the time, effort, and cost of risk management, is an investment you make to provide early warning of problems and increase the probability of success

The Main Points of Chapter 7 (4)

- like risk management, the amount you invest in measurement and control of work products depends on the criticality of delivering an acceptable product within the project constraints, and the cost of failing to do so
- SEI, ISO, IEEE, PMI, and PSM-INCOSE provide frameworks, standards, and guidelines for measuring and controlling product attributes (see Appendix 7A to this chapter)
- procedures and forms for conducting software inspections are contained in Appendix 7B to this chapter