Chapter 14

- **Quality Concepts**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

Software Quality

- In 2005, *ComputerWorld* [Hil05] lamented that
  - "bad software plagues nearly every organization that uses computers, causing lost work hours during computer downtime, lost or corrupted data, missed sales opportunities, high IT support and maintenance costs, and low customer satisfaction.
- A year later, *InfoWorld* [Fos06] wrote about the
  - "the sorry state of software quality" reporting that the quality problem had not gotten any better.
- Today, software quality remains an issue, but who is to blame?
  - Customers blame developers, arguing that sloppy practices lead to low-quality software.
  - Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated.

Quality

- The *American Heritage Dictionary* defines *quality* as
  - "a characteristic or attribute of something."
- For software, two kinds of quality may be encountered:
  - Quality of design encompasses requirements, specifications, and the design of the system.
  - Quality of conformance is an issue focused primarily on implementation.
  - User satisfaction = compliant product + good quality + delivery within budget and schedule

Quality—A Philosophical View

- Robert Persig [Per74] commented on the thing we call *quality*:
  - Quality . . . you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others, that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about. But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one knows what it is, then for all practical purposes it doesn't exist at all. But for all practical purposes it really does exist. What else are the grades based on? Why else would people pay fortunes for some things and throw others in the trash pile? Obviously some things are better than others . . . but what's the betterness? . . . So round and round you go, spinning mental wheels and nowhere finding anyplace to get traction. What the hell is Quality? What is it?

Quality—A Pragmatic View

- The *transcendental view* argues (like Persig) that quality is something that you immediately recognize, but cannot explicitly define.
- The *user view* sees quality in terms of an end-user's specific goals. If a product meets those goals, it exhibits quality.
- The *manufacturer's view* defines quality in terms of the original specification of the product. If the product conforms to the spec, it exhibits quality.
- The *product view* suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product.
- Finally, the *value-based view* measures quality based on how much a customer is willing to pay for a product. In reality, quality encompasses all of these views and more.

Software Quality

- Software quality can be defined as:
  - *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.*
- This definition has been adapted from [Bes04] and replaces a more manufacturing-oriented view presented in earlier editions of this book.

Effective Software Process

- An *effective software process* establishes the infrastructure that supports any effort at building a high quality software product.
- The management aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality.
- Software engineering practices allow the developer to analyze the problem and design a solid solution—both critical to building high quality software.
- Finally, umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.

Useful Product

- A *useful product* delivers the content, functions, and features that the end-user desires
- But as important, it delivers these assets in a reliable, error free way.
- A useful product always satisfies those requirements that have been explicitly stated by stakeholders.
- In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high quality software.

Adding Value

- By *adding value for both the producer and user* of a software product, high quality software provides benefits for the software organization and the end-user community.
- The software organization gains added value because high quality software requires less maintenance effort, fewer bug fixes, and reduced customer support.
- The user community gains added value because the application provides a useful capability in a way that expedites some business process.
- The end result is:
  - (1) greater software product revenue,
  - (2) better profitability when an application supports a business process, and/or
  - (3) improved availability of information that is crucial for the business.

Quality Dimensions

- David Garvin [Gar87]:
  - **Performance Quality.** Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end-user?
  - **Feature quality.** Does the software provide features that surprise and delight first-time end-users?
  - **Reliability.** Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error free?
  - **Conformance.** Does the software conform to local and external software standards that are relevant to the application? Does it conform to de facto design and coding conventions? For example, does the user interface conform to accepted design rules for menu selection or data input?

Quality Dimensions

- **Durability.** Can the software be maintained (changed) or corrected (debugged) without the inadvertent generation of unintended side effects? Will changes cause the error rate or reliability to degrade with time?
- **Serviceability.** Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period. Can support staff acquire all information they need to make changes or correct defects?
- **Aesthetics.** Most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious "presence" that are hard to quantify but evident nonetheless.
- **Perception.** In some situations, you have a set of prejudices that will influence your perception of quality.

Other Views

- **McCall's Quality Factors** (*SEPA,* Section 14.2.2)
- **ISO 9126 Quality Factors** (*SEPA,* Section 14.2.3)
- **Targeted Factors** (*SEPA,* Section 14.2.4)

The Software Quality Dilemma

- If you produce a software system that has terrible quality, you lose because no one will want to buy it.
- If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it's going to take so long to complete and it will be so expensive to produce that you'll be out of business anyway.
- Either you missed the market window, or you simply exhausted all your resources.
- So people in industry try to get to that magical middle ground where the product is good enough not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete. [Ven03]

"Good Enough" Software

- Good enough software delivers high quality functions and features that end-users desire, but at the same time it delivers other more obscure or specialized functions and features that contain known bugs.
- Arguments *against* "good enough."
  - It is true that "good enough" may work in some application domains and for a few major software companies. After all, if a company has a large marketing budget and can convince enough people to buy version 1.0, it has succeeded in locking them in.
  - If you work for a small company be wary of this philosophy. If you deliver a "good enough" (buggy) product, you risk permanent damage to your company's reputation.
  - You may never get a chance to deliver version 2.0 because bad buzz may cause your sales to plummet and your company to fold.
  - If you work in certain application domains (e.g., real time embedded software, application software that is integrated with hardware can be negligent and open your company to expensive litigation.

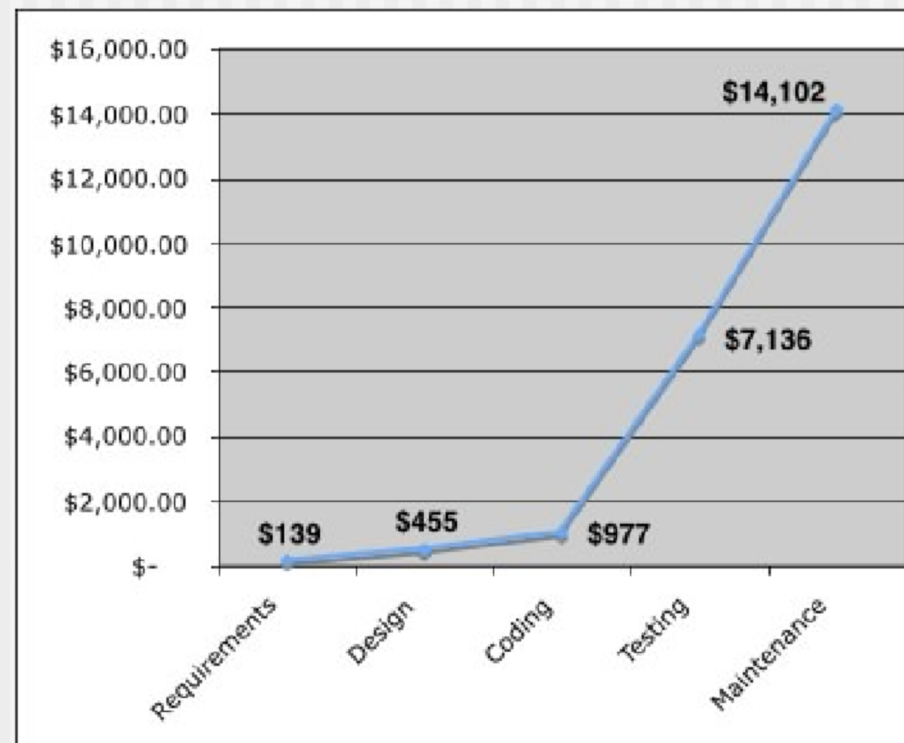
Cost of Quality

- *Prevention costs* include
  - quality planning
  - formal technical reviews
  - test equipment
  - Training
- *Internal failure costs* include
  - rework
  - repair
  - failure mode analysis
- *External failure costs* are
  - complaint resolution
  - product return and replacement
  - help line support
  - warranty work

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Cost

- The relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure costs.



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Quality and Risk

- *"People bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right."* SEPA, Chapter 1
- Example:
  - *Throughout the month of November, 2000 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers. In the months that followed, five of these patients died from radiation poisoning and 15 others developed serious complications. What caused this tragedy? A software package, developed by a U.S. company, was modified by hospital technicians to compute modified doses of radiation for each patient.*

Negligence and Liability

- The story is all too common. A governmental or corporate entity hires a major software developer or consulting company to analyze requirements and then design and construct a software-based “system” to support some major activity.
  - The system might support a major corporate function (e.g., pension management) or some governmental function (e.g., healthcare administration or homeland security).
- Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad.
- The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval.
- Litigation ensues.

Quality and Security

- Gary McGraw comments [Wil05]:
- "Software security relates entirely and completely to quality. You must think about security, reliability, availability, dependability—at the beginning, in the design, architecture, test, and coding phases, all through the software life cycle [process]. Even people aware of the software security problem have focused on late life-cycle stuff. The earlier you find the software problem, the better. And there are two kinds of software problems. One is bugs, which are implementation problems. The other is software flaws—architectural problems in the design. People pay too much attention to bugs and not enough on flaws."

Achieving Software Quality

- Critical success factors:
  - **Software Engineering Methods**
  - **Project Management Techniques**
  - **Quality Control**
  - **Quality Assurance**

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Chapter 15

- **Review Techniques**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

Reviews

... there is no particular reason why your friend and colleague cannot also be your sternest critic.

Jerry Weinberg

What Are Reviews?

- a meeting conducted by technical people for technical people
- a technical assessment of a work product created during the software engineering process
- a software quality assurance mechanism
- a training ground

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

What Reviews Are Not

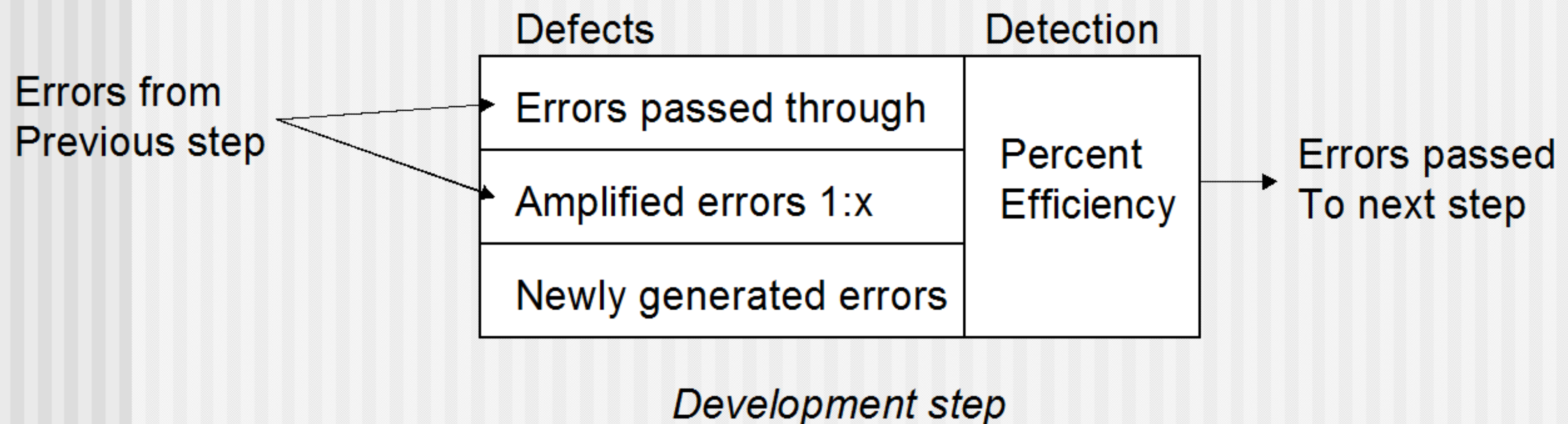
- A project summary or progress assessment
- A meeting intended solely to impart information
- A mechanism for political or personal reprisal!

What Do We Look For?

- Errors and defects
  - *Error*—a quality problem found *before* the software is released to end users
  - *Defect*—a quality problem found only *after* the software has been released to end-users
- We make this distinction because errors and defects have very different economic, business, psychological, and human impact
- However, the temporal distinction made between errors and defects in this book is *not* mainstream thinking

Defect Amplification

- A *defect amplification model* [IBM81] can be used to illustrate the generation and detection of errors during the design and code generation actions of a software process.



Defect Amplification

- In the example provided in SEPA, Section 15.2,
  - a software process that does NOT include reviews,
    - yields 94 errors at the beginning of testing and
    - Releases 12 latent defects to the field
  - a software process that does include reviews,
    - yields 24 errors at the beginning of testing and
    - releases 3 latent defects to the field
  - A cost analysis indicates that the process with NO reviews costs approximately 3 times more than the process with reviews, taking the cost of correcting the latent defects into account

Metrics

- The total review effort and the total number of errors discovered are defined as:
  - $E_{review} = E_p + E_a + E_r$
  - $Err_{tot} = Err_{minor} + Err_{major}$
- *Defect density* represents the errors found per unit of work product reviewed.
  - Defect density = Err_{tot} / WPS
- where …

Metrics

- *Preparation effort,* E_p—the effort (in person-hours) required to review a work product prior to the actual review meeting
- *Assessment effort,* E_a— the effort (in person-hours) that is expending during the actual review
- *Rework effort,* E_r— the effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review
- *Work product size, WPS*—a measure of the size of the work product that has been reviewed (e.g., the number of UML models, or the number of document pages, or the number of lines of code)
- *Minor errors found,* Err_{minor}—the number of errors found that can be categorized as minor (requiring less than some pre-specified effort to correct)
- *Major errors found,* Err_{major}— the number of errors found that can be categorized as major (requiring more than some pre-specified effort to correct)

An Example—I

- If past history indicates that
  - the average defect density for a requirements model is 0.6 errors per page, and a new requirement model is 32 pages long,
  - a rough estimate suggests that your software team will find about 19 or 20 errors during the review of the document.
  - If you find only 6 errors, you've done an extremely good job in developing the requirements model *or* your review approach was not thorough enough.

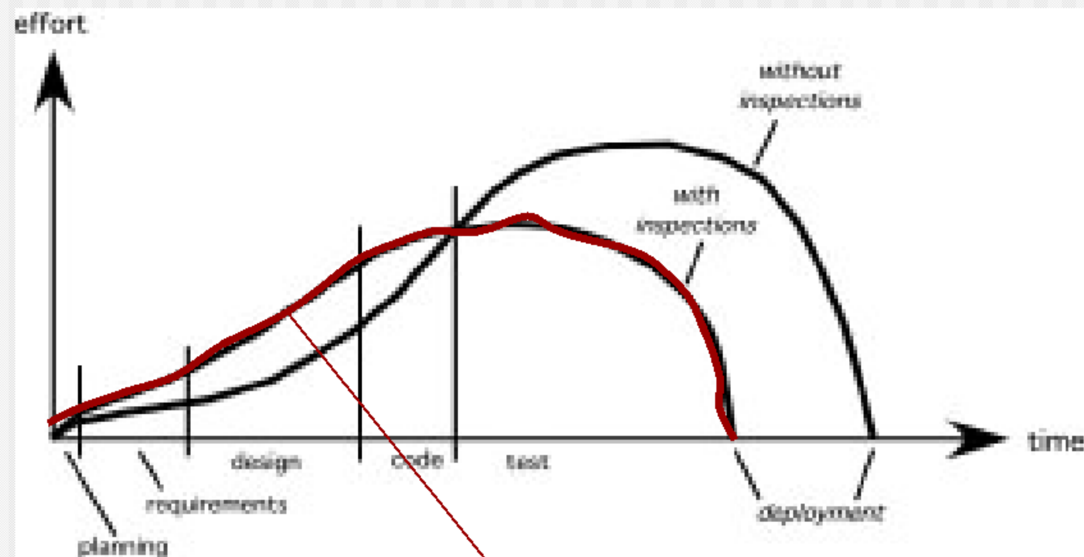
An Example—II

- The effort required to correct a minor model error (immediately after the review) was found to require 4 person-hours.
- The effort required for a major requirement error was found to be 18 person-hours.
- Examining the review data collected, you find that minor errors occur about 6 times more frequently than major errors. Therefore, you can estimate that the average effort to find and correct a requirements error during review is about 6 person-hours.
- Requirements related errors uncovered during testing require an average of 45 person-hours to find and correct. Using the averages noted, we get:
- Effort saved per error = $E_{testing} – E_{reviews}$
- 45 – 6 = 30 person-hours/error
- Since 22 errors were found during the review of the requirements model, a saving of about 660 person-hours of testing effort would be achieved. And that's just for requirements-related errors.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Overall

- Effort expended with and without reviews



with reviews

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Reference Model

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Informal Reviews

- Informal reviews include:
  - a simple desk check of a software engineering work product with a colleague
  - a casual meeting (involving more than 2 people) for the purpose of reviewing a work product, or
  - the review-oriented aspects of pair programming
- *pair programming* encourages continuous review as a work product (design or code) is created.
  - The benefit is immediate discovery of errors and better work product quality as a consequence.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Formal Technical Reviews

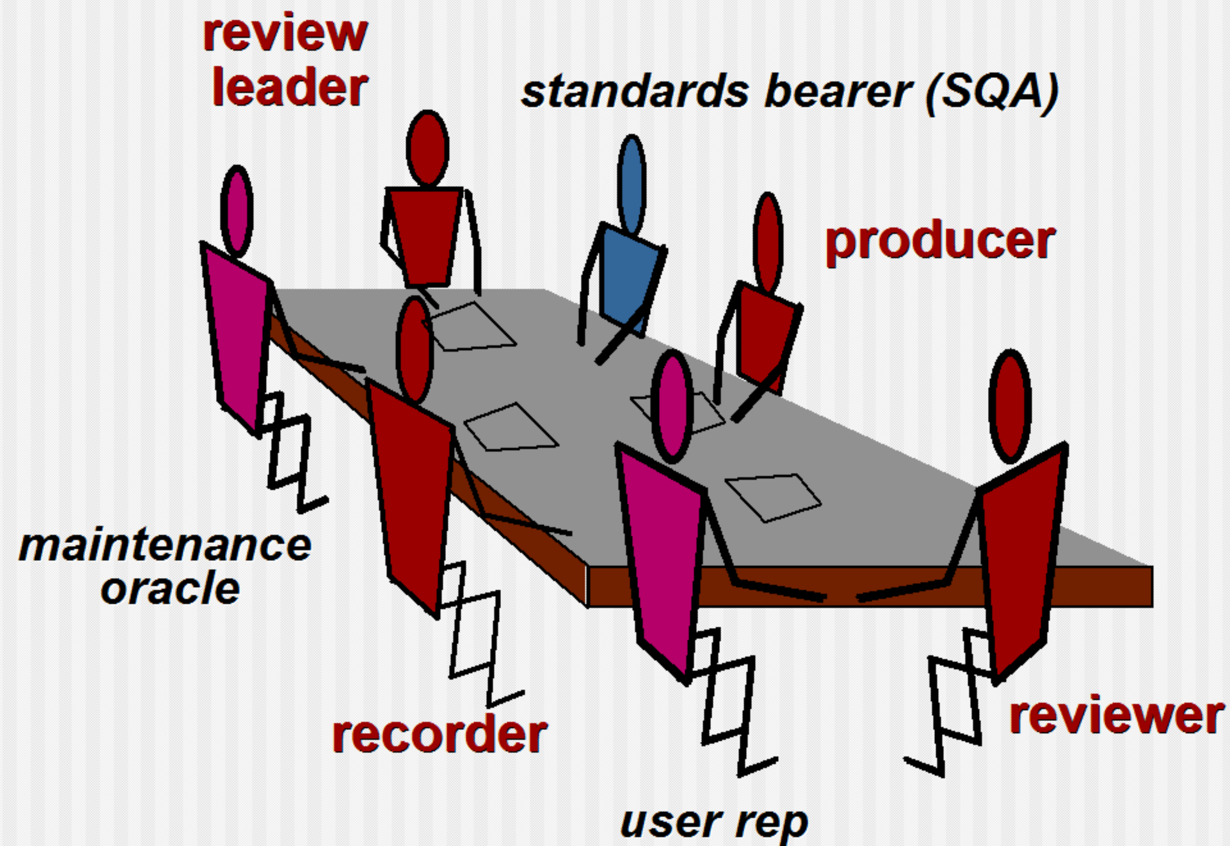
- The objectives of an FTR are:
  - to uncover errors in function, logic, or implementation for any representation of the software
  - to verify that the software under review meets its requirements
  - to ensure that the software has been represented according to predefined standards
  - to achieve software that is developed in a uniform manner
  - to make projects more manageable
- The FTR is actually a class of reviews that includes *walkthroughs* and *inspections*.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

The Review Meeting

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.
- Focus is on a work product (e.g., a portion of a requirements model, a detailed component design, source code for a component)

The Players



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

The Players

- *Producer*—the individual who has developed the work product
  - informs the project leader that the work product is complete and that a review is required
- *Review leader*—evaluates the product for readiness, generates copies of product materials, and distributes them to two or three *reviewers* for advance preparation.
- *Reviewer(s)*—expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.
- *Recorder*—reviewer who records (in writing) all important issues raised during the review.

Conducting the Review

- *Review the product, not the producer.*
- *Set an agenda and maintain it.*
- *Limit debate and rebuttal.*
- *Enunciate problem areas, but don't attempt to solve every problem noted.*
- *Take written notes.*
- *Limit the number of participants and insist upon advance preparation.*
- *Develop a checklist for each product that is likely to be reviewed.*
- *Allocate resources and schedule time for FTRs.*
- *Conduct meaningful training for all reviewers.*
- *Review your early reviews.*

Review Options Matrix

	IPR*	WT	IN	RRR
trained leader	no	yes	yes	yes
agenda established	maybe	yes	yes	yes
reviewers prepare in advance	maybe	yes	yes	yes
producer presents product	maybe	yes	no	no
“reader” presents product	no	no	yes	no
recorder takes notes	maybe	yes	yes	yes
checklists used to find errors	no	no	yes	no
errors categorized as found	no	no	yes	no
issues list created	no	yes	yes	yes
team must sign-off on result	no	yes	yes	maybe

***IPR—informal peer review   WT—Walkthrough**
IN—Inspection   RRR—round robin review

Sample-Driven Reviews (SDRs)

- SDRs attempt to quantify those work products that are primary targets for full FTRs.

To accomplish this …

- Inspect a fraction a_i of each software work product, *i.* Record the number of faults, f_i found within a_i.
- Develop a gross estimate of the number of faults within work product *i* by multiplying f_i by $1/a_i$.
- Sort the work products in descending order according to the gross estimate of the number of faults in each.
- Focus available review resources on those work products that have the highest estimated number of faults.

Metrics Derived from Reviews

- inspection time per page of documentation
- inspection time per KLOC or FP
- inspection effort per KLOC or FP
- errors uncovered per reviewer hour
- errors uncovered per preparation hour
- errors uncovered per SE task (e.g., design)
- number of minor errors (e.g., typos)
- number of major errors
  (e.g., nonconformance to req.)
- number of errors found during preparation

Chapter 16

- **Software Quality Assurance**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e

by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

Comment on Quality

- Phil Crosby once said:
  - The problem of quality management is not what people don't know about it. The problem is what they think they do know . . . In this regard, quality has much in common with sex.
  - *Everybody is for it.* (Under certain conditions, of course.)
  - *Everyone feels they understand it.* (Even though they wouldn't want to explain it.)
  - *Everyone thinks execution is only a matter of following natural inclinations.* (After all, we do get along somehow.)
  - *And, of course, most people feel that problems in these areas are caused by other people.* (If only they would take the time to do things right.)

Elements of SQA

- Standards
- Reviews and Audits
- Testing
- Error/defect collection and analysis
- Change management
- Education
- Vendor management
- Security management
- Safety
- Risk management

Role of the SQA Group-I

- **Prepares an SQA plan for a project.**
  - The plan identifies
    - evaluations to be performed
    - audits and reviews to be performed
    - standards that are applicable to the project
    - procedures for error reporting and tracking
    - documents to be produced by the SQA group
    - amount of feedback provided to the software project team
- **Participates in the development of the project's software process description.**
  - The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Role of the SQA Group-II

- **Reviews software engineering activities to verify compliance with the defined software process.**
  - identifies, documents, and tracks deviations from the process and verifies that corrections have been made.
- **Audits designated software work products to verify compliance with those defined as part of the software process.**
  - reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made
  - periodically reports the results of its work to the project manager.
- **Ensures that deviations in software work and work products are documented and handled according to a documented procedure.**
- **Records any noncompliance and reports to senior management.**
  - Noncompliance items are tracked until they are resolved.

SQA Goals (see Figure 16.1)

- **Requirements quality.** The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow.
- **Design quality.** Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements.
- **Code quality.** Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability.
- **Quality control effectiveness.** A software team should apply limited resources in a way that has the highest likelihood of achieving a high quality result.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Statistical SQA

Product & Process

Collect information on all defects
Find the causes of the defects
Move to provide fixes for the process

... an understanding of how to improve quality ...

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Statistical SQA

- Information about software errors and defects is collected and categorized.
- An attempt is made to trace each error and defect to its underlying cause (e.g., non-conformance to specifications, design error, violation of standards, poor communication with the customer).
- Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the *vital few*).
- Once the vital few causes have been identified, move to correct the problems that have caused the errors and defects.

Six-Sigma for Software Engineering

- The term "six sigma" is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard.
- The Six Sigma methodology defines three core steps:
  - *Define* customer requirements and deliverables and project goals via well-defined methods of customer communication
  - *Measure* the existing process and its output to determine current quality performance (collect defect metrics)
  - *Analyze* defect metrics and determine the vital few causes.
  - *Improve* the process by eliminating the root causes of defects.
  - *Control* the process to ensure that future work does not reintroduce the causes of defects.

Software Reliability

- A simple measure of reliability is *mean-time-between-failure* (MTBF), where

  MTBF = MTTF + MTTR

- The acronyms MTTF and MTTR are *mean-time-to-failure* and *mean-time-to-repair*, respectively.
- *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

  Availability = [MTTF/(MTTF + MTTR)] x 100%

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Software Safety

- *Software safety* is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail.
- If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.

ISO 9001:2000 Standard

- ISO 9001:2000 is the quality assurance standard that applies to software engineering.
- The standard contains 20 requirements that must be present for an effective quality assurance system.
- The requirements delineated by ISO 9001:2000 address topics such as
  - management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques.

Chapter 17

- **Software Testing Strategies**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e

by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

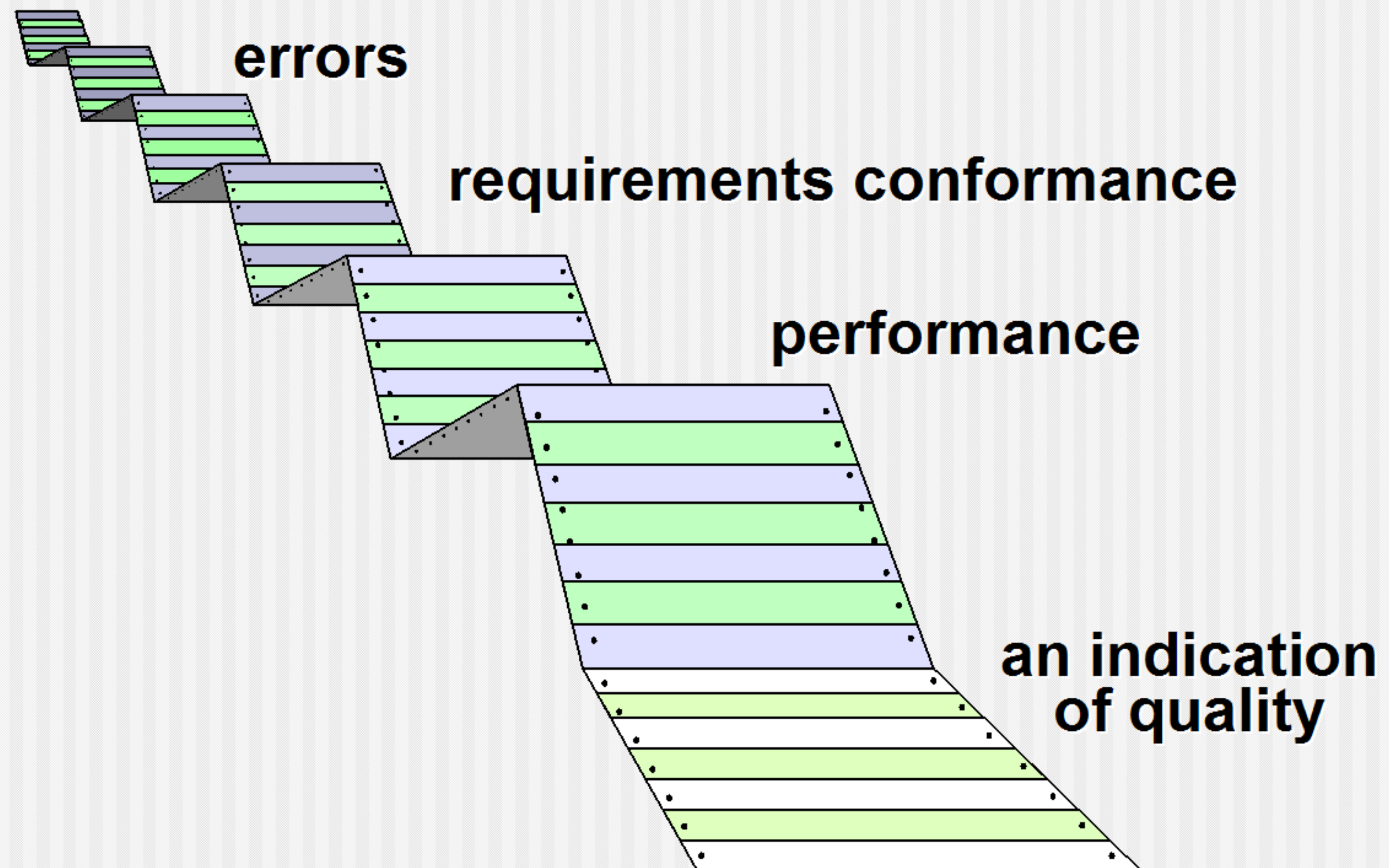
All copyright information MUST appear if these slides are posted on a website for student use.

Software Testing

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

What Testing Shows



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

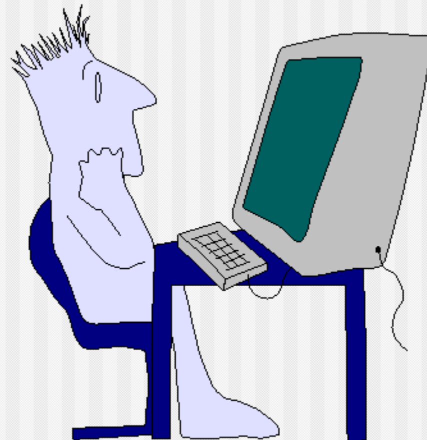
Strategic Approach

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

V & V

- *Verification* refers to the set of tasks that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:
  - *Verification:* "Are we building the product right?"
  - *Validation:* "Are we building the right product?"

Who Tests the Software?



developer

Understands the system

but, will test "gently"

and, is driven by "delivery"

independent tester

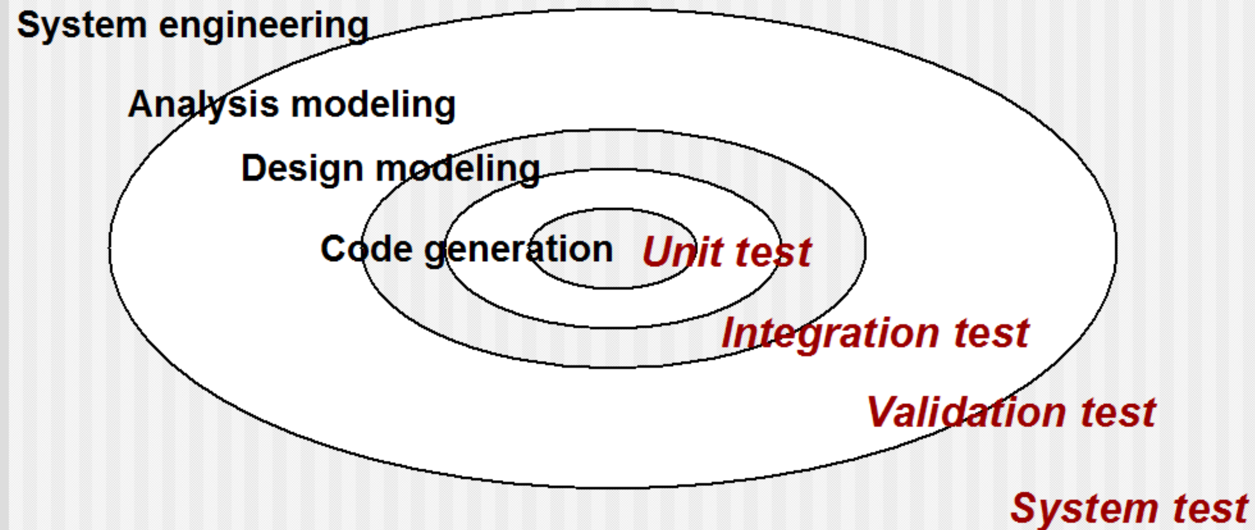
Must learn about the system,

but, will attempt to break it

and, is driven by quality

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Testing Strategy



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Testing Strategy

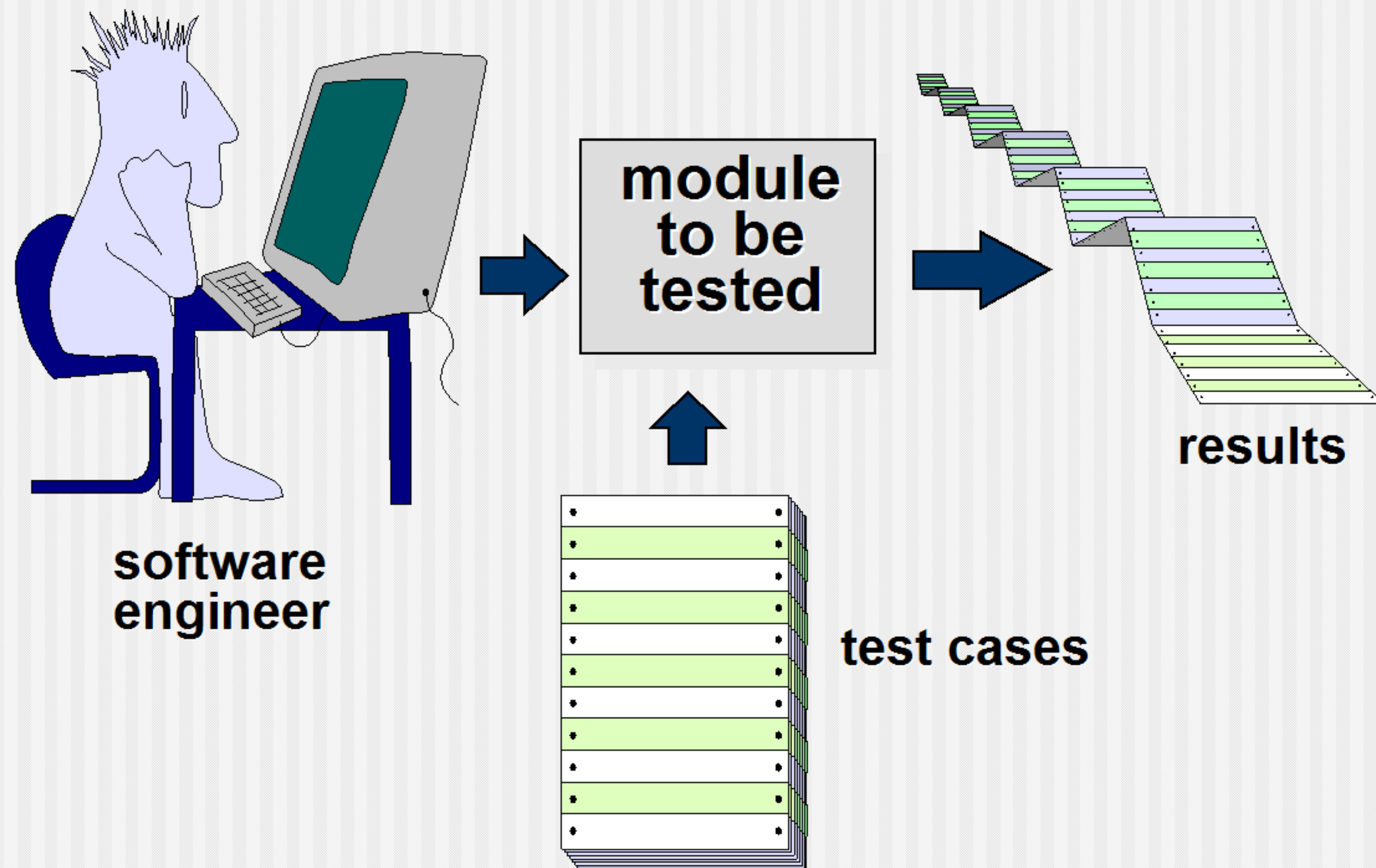
- We begin by 'testing-in-the-small' and move toward 'testing-in-the-large'
- For conventional software
  - The module (component) is our initial focus
  - Integration of modules follows
- For OO software
  - our focus when "testing in the small" changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

Strategic Issues

- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes “rapid cycle testing.”
- Build “robust” software that is designed to test itself
- Use effective technical reviews as a filter prior to testing
- Conduct technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

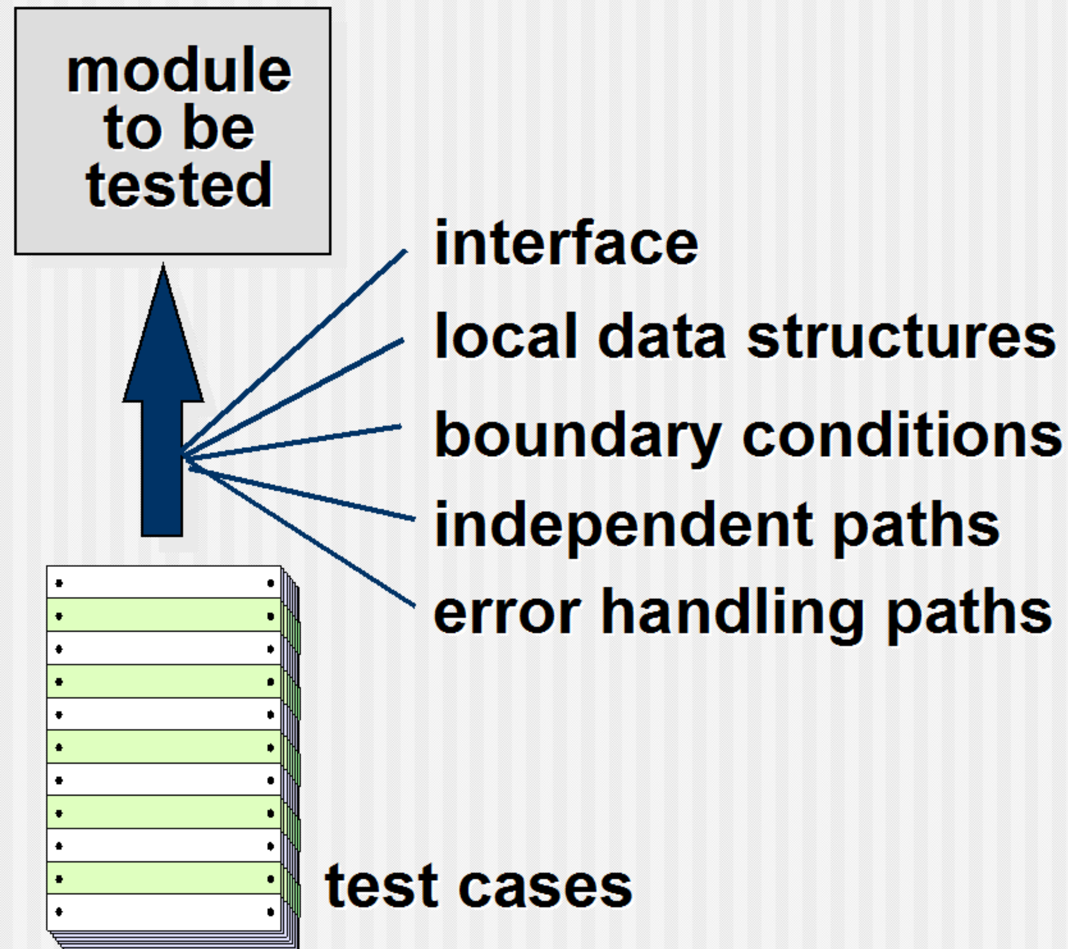
These slides are designed to accompany *Software Engineering: A Practitioner’s Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Unit Testing



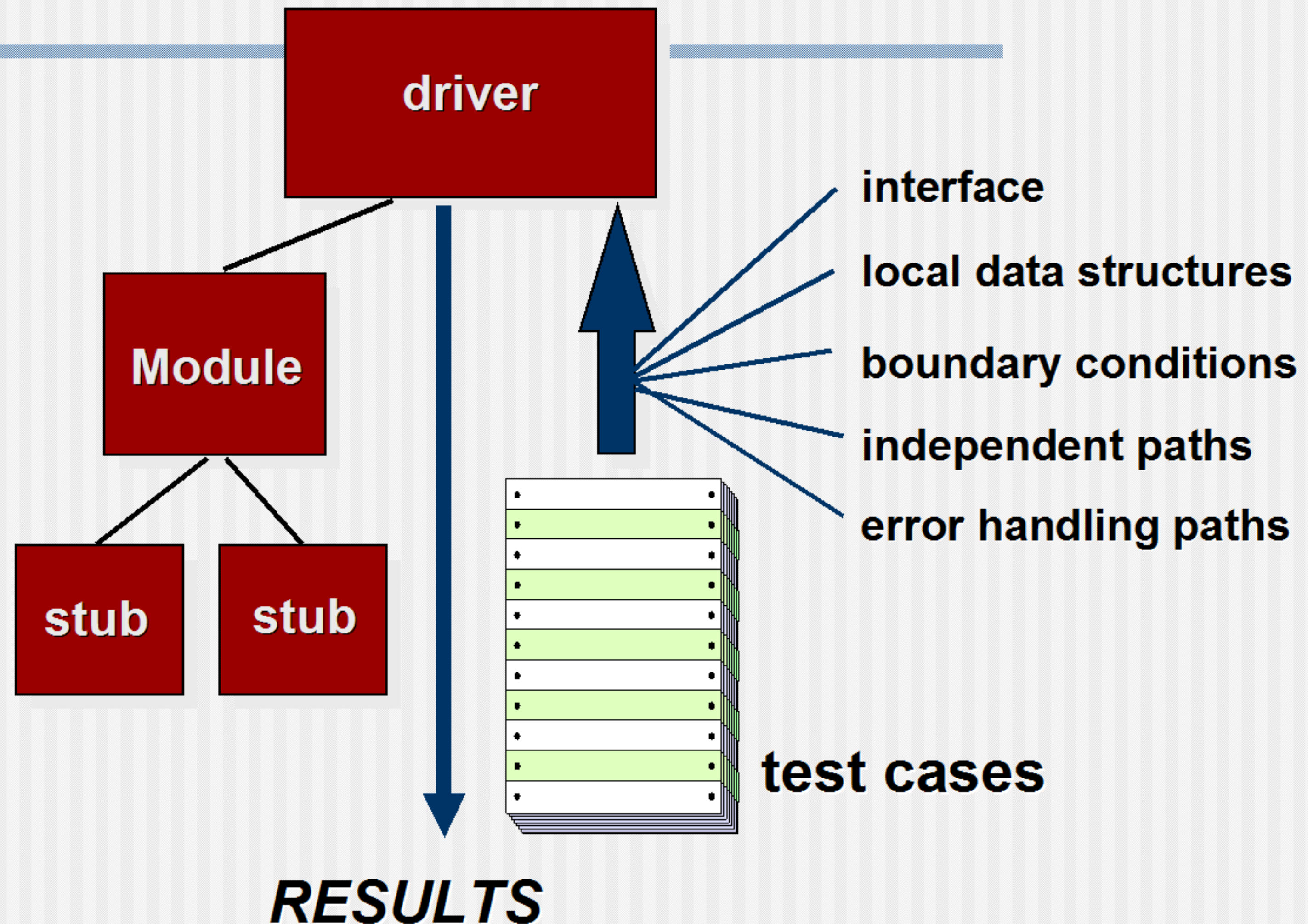
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Unit Testing



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Unit Test Environment

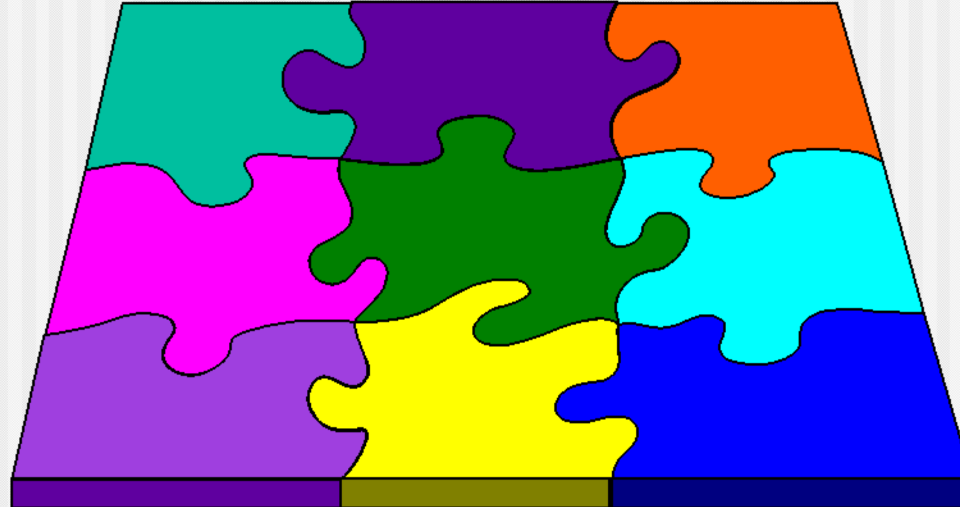


These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Integration Testing Strategies

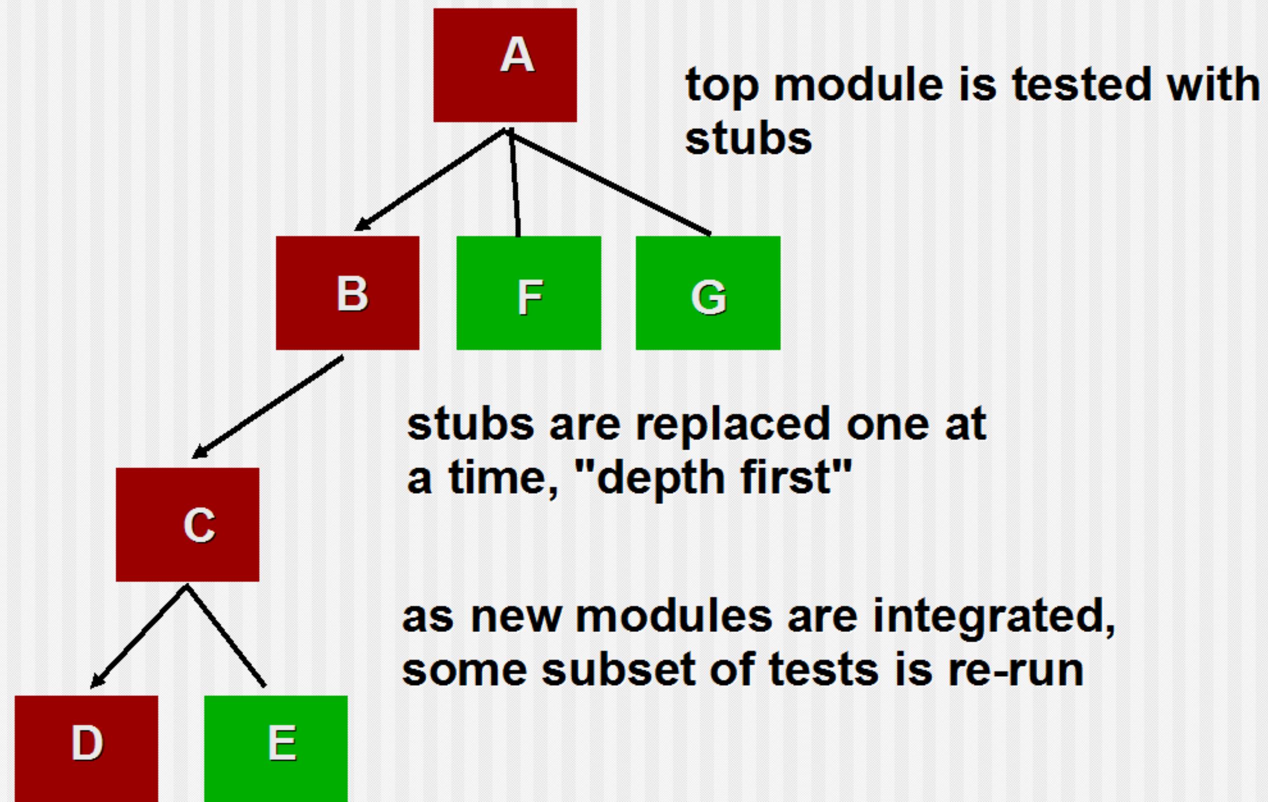
Options:

- **the “big bang” approach**
- **an incremental construction strategy**



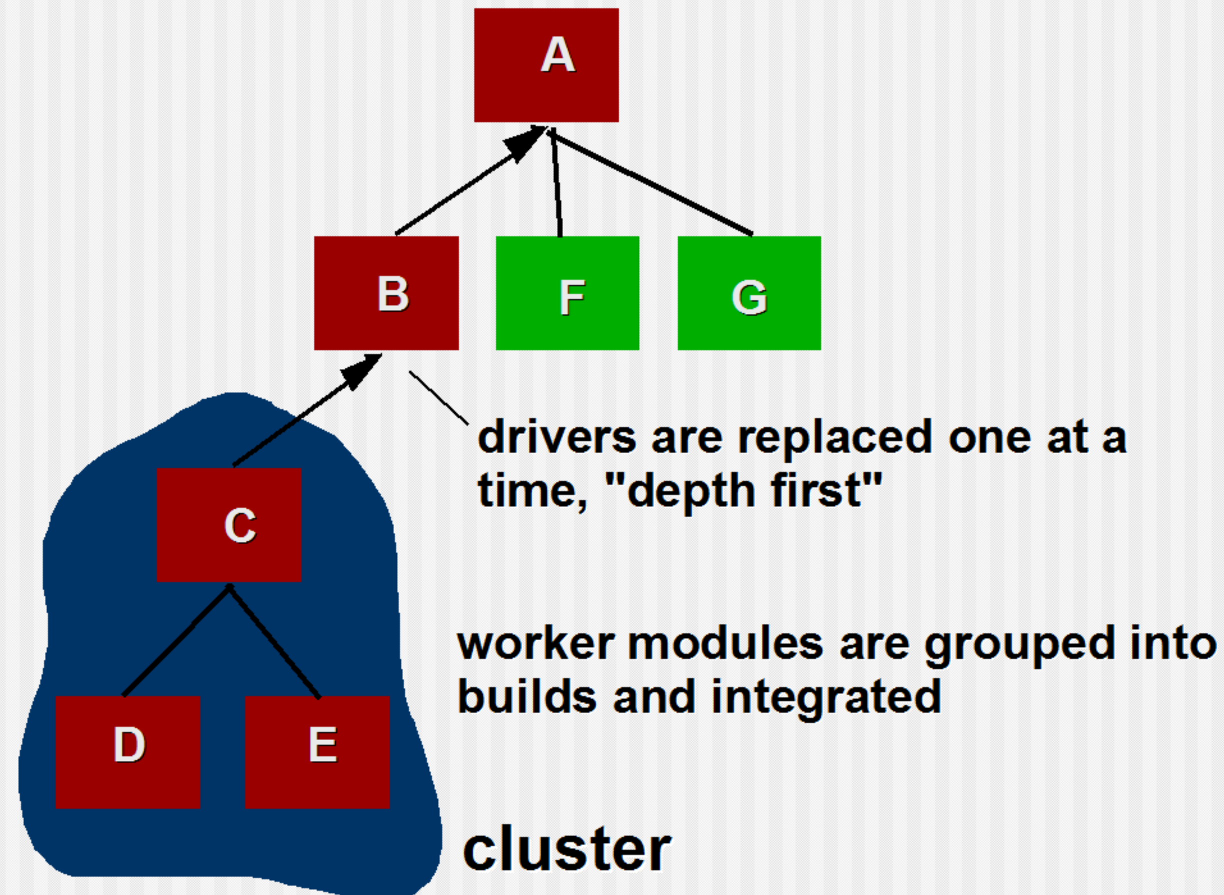
These slides are designed to accompany *Software Engineering: A Practitioner’s Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Top Down Integration



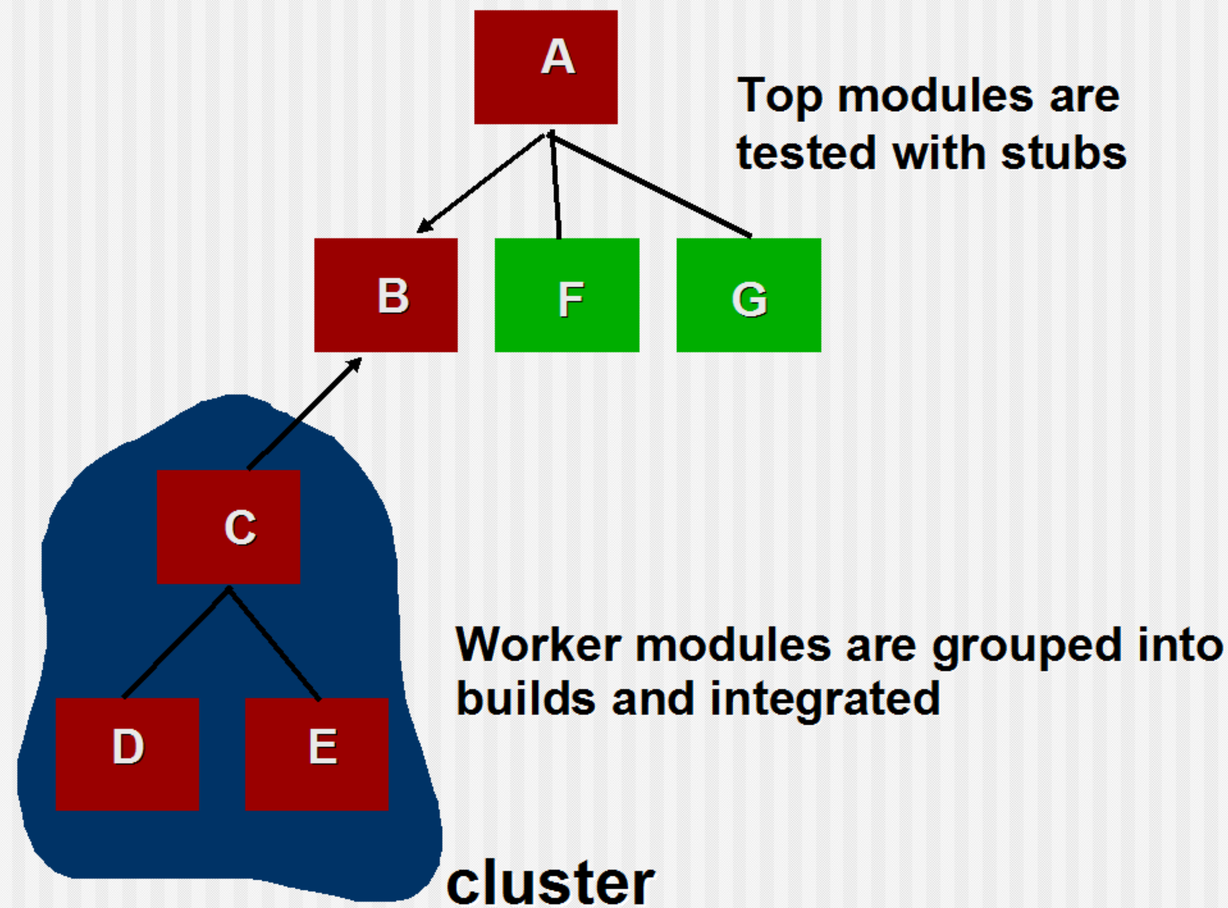
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Bottom-Up Integration



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Sandwich Testing



Regression Testing

- *Regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

Smoke Testing

- A common approach for creating "daily builds" for product software
- Smoke testing steps:
  - Software components that have been translated into code are integrated into a "build."
    - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  - A series of tests is designed to expose errors that will keep the build from properly performing its function.
    - The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
    - The integration approach may be top down or bottom up.

Object-Oriented Testing

- begins by evaluating the correctness and consistency of the analysis and design models
- testing strategy changes
  - the concept of the 'unit' broadens due to encapsulation
  - integration focuses on classes and their execution across a 'thread' or in the context of a usage scenario
  - validation uses conventional black box methods
- test case design draws on conventional methods, but also encompasses special features

Broadening the View of "Testing"

It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level. Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side effects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

Testing the CRC Model

1. Revisit the CRC model and the object-relationship model.
2. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
3. Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
4. Using the inverted connections examined in step 3, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
5. Determine whether widely requested responsibilities might be combined into a single responsibility.
6. Steps 1 to 5 are applied iteratively to each class and through each evolution of the analysis model.

OO Testing Strategy

- class testing is the equivalent of unit testing
  - operations within the class are tested
  - the state behavior of the class is examined
- integration applied three different strategies
  - thread-based testing—integrates the set of classes required to respond to one input or event
  - use-based testing—integrates the set of classes required to respond to one use case
  - cluster testing—integrates the set of classes required to demonstrate one collaboration

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

WebApp Testing - I

- The content model for the WebApp is reviewed to uncover errors.
- The interface model is reviewed to ensure that all use cases can be accommodated.
- The design model for the WebApp is reviewed to uncover navigation errors.
- The user interface is tested to uncover errors in presentation and/or navigation mechanics.
- Each functional component is unit tested.

WebApp Testing - II

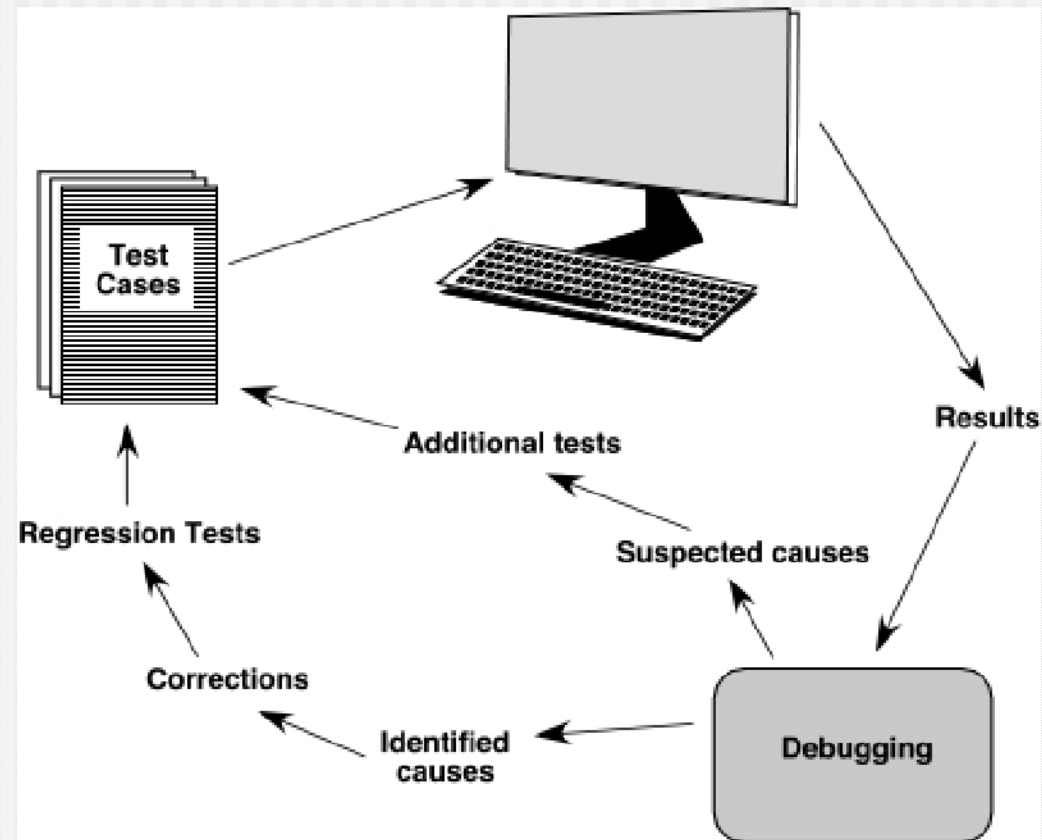
- Navigation throughout the architecture is tested.
- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
- Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
- Performance tests are conducted.
- The WebApp is tested by a controlled and monitored population of end-users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

High Order Testing

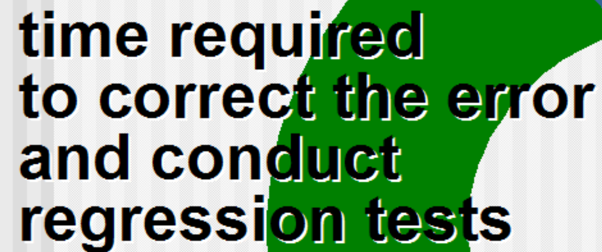
- Validation testing
  - Focus is on software requirements
- System testing
  - Focus is on system integration
- Alpha/Beta testing
  - Focus is on customer usage
- Recovery testing
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing
  - verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- Stress testing
  - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance Testing
  - test the run-time performance of software within the context of an integrated system

Debugging: A Diagnostic Process

The Debugging Process



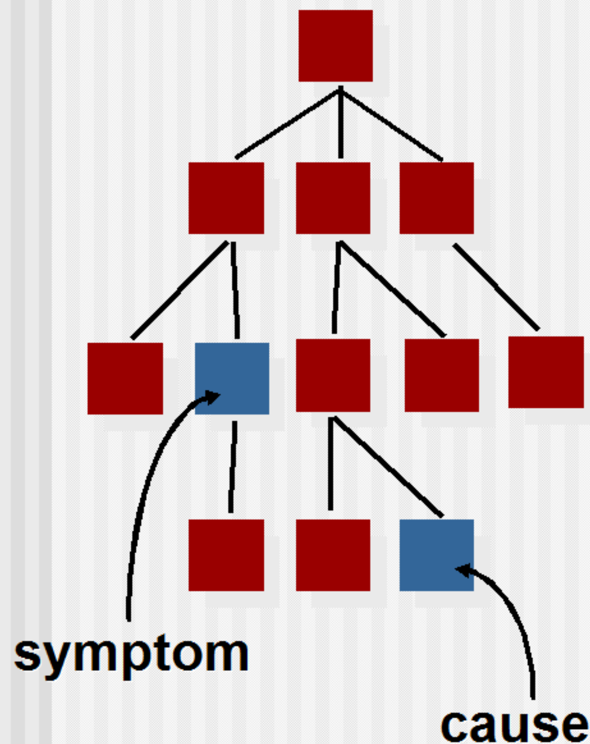
Debugging Effort



time required
to diagnose the
symptom and
determine the
cause

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

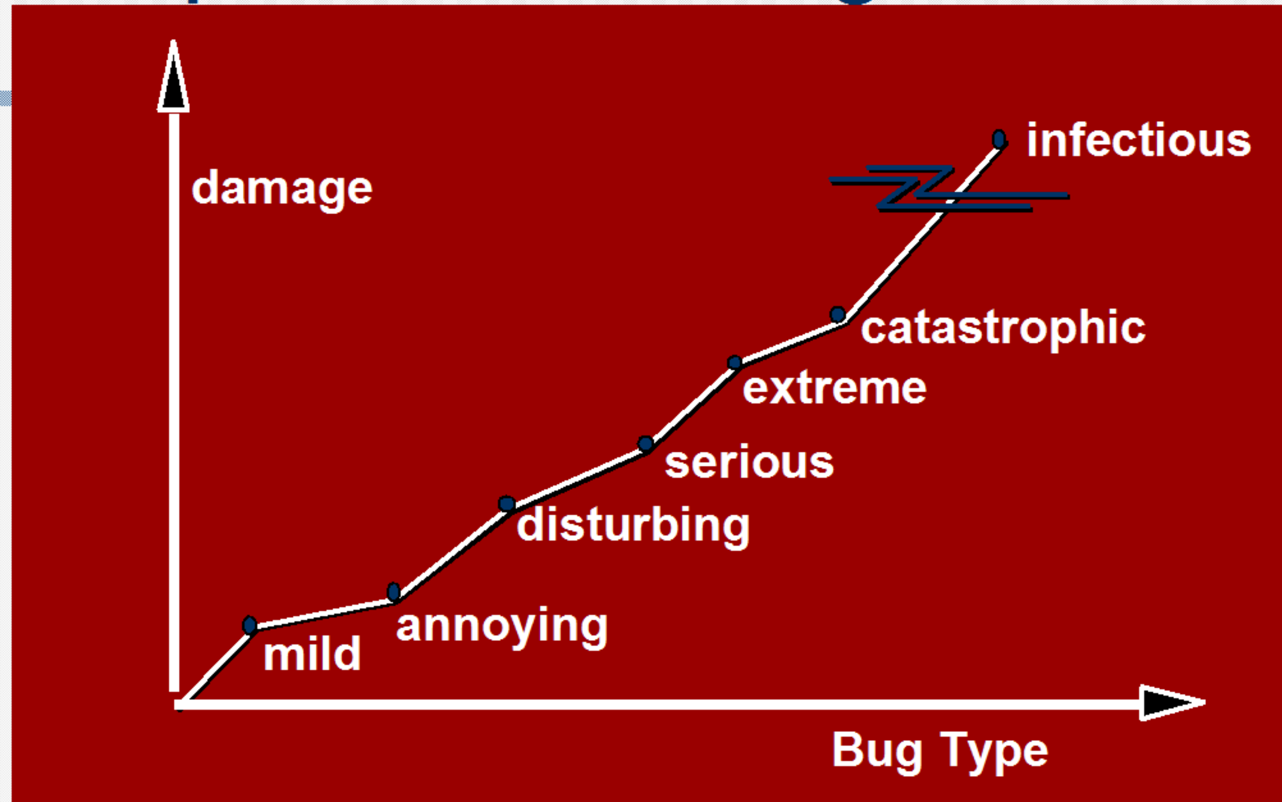
Symptoms & Causes



- symptom and cause may be geographically separated
- symptom may disappear when another problem is fixed
- cause may be due to a combination of non-errors
- cause may be due to a system or compiler error
- cause may be due to assumptions that everyone believes
- symptom may be intermittent

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Consequences of Bugs



Bug Categories: **function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.**

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Debugging Techniques

- brute force / testing
- backtracking
- induction
- deduction

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Correcting the Error

- *Is the cause of the bug reproduced in another part of the program?* In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere.
- *What "next bug" might be introduced by the fix I'm about to make?* Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures.
- *What could we have done to prevent this bug in the first place?* This question is the first step toward establishing a statistical software quality assurance approach. If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

Final Thoughts

- *Think* -- before you act to correct
- Use tools to gain additional insight
- If you're at an impasse, get help from someone else
- Once you correct the bug, use regression testing to uncover any side effects

Chapter 18

- **Testing Conventional Applications**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e

by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

Testability

- Operability—it operates cleanly
- Observability—the results of each test case are readily observed
- Controllability—the degree to which testing can be automated and optimized
- Decomposability—testing can be targeted
- Simplicity—reduce complex architecture and logic to simplify tests
- Stability—few changes are requested during testing
- Understandability—of the design

What is a “Good” Test?

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test should be “best of breed”
- A good test should be neither too simple nor too complex

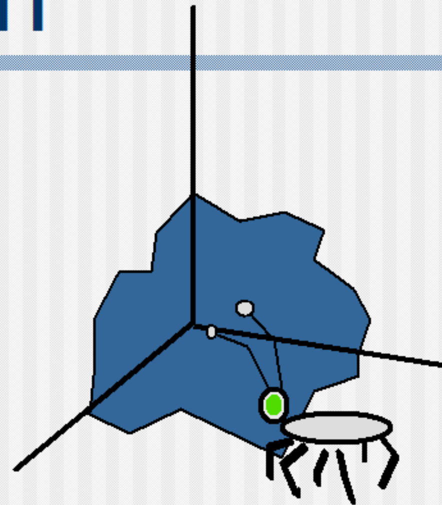
Internal and External Views

- Any engineered product (and most other things) can be tested in one of two ways:
  - Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
  - Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

Test Case Design

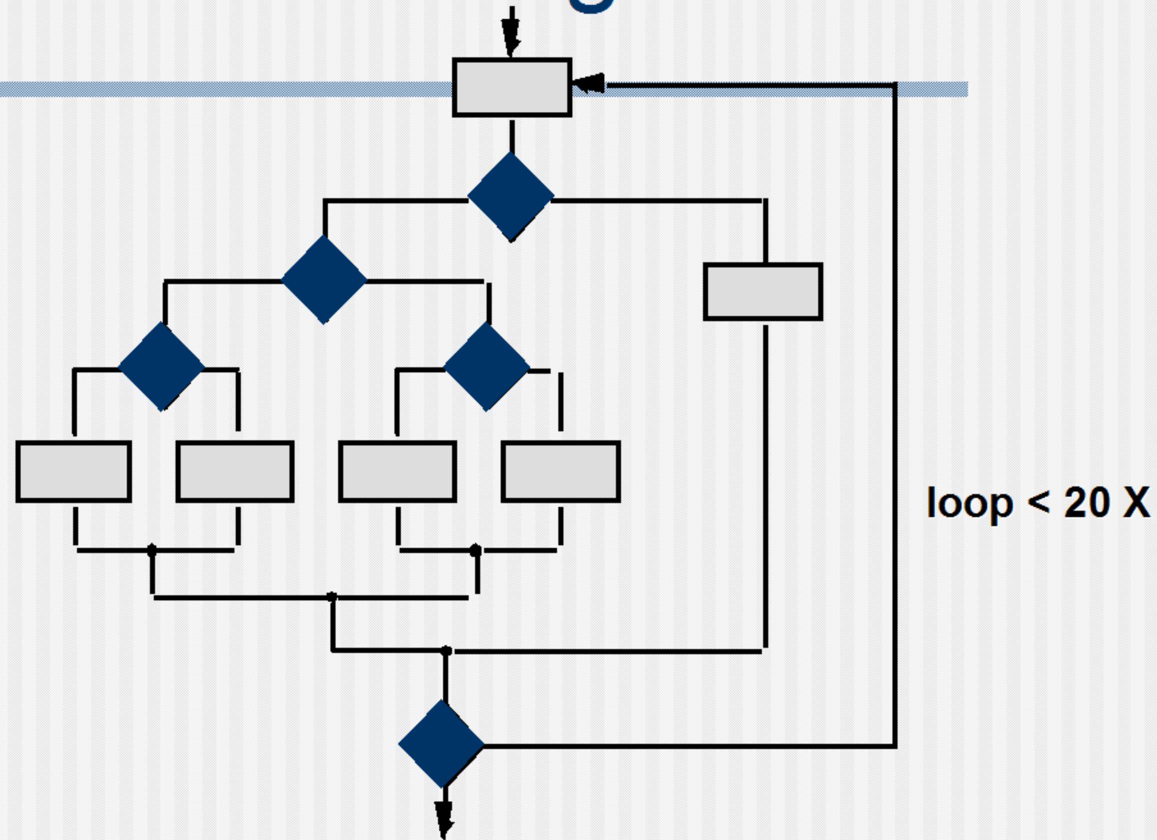
"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer



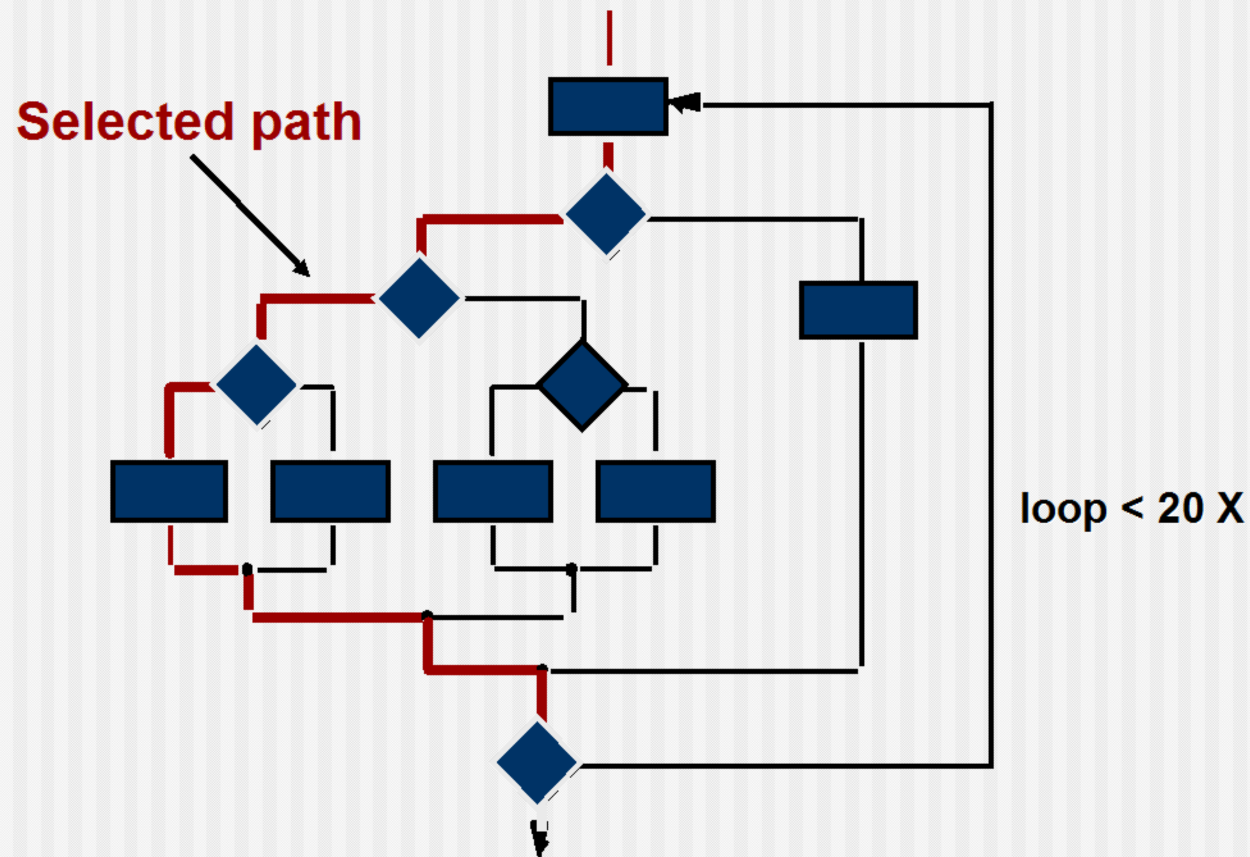
OBJECTIVE	**to uncover errors**
CRITERIA	**in a complete manner**
CONSTRAINT	**with a minimum of effort and time**

Exhaustive Testing



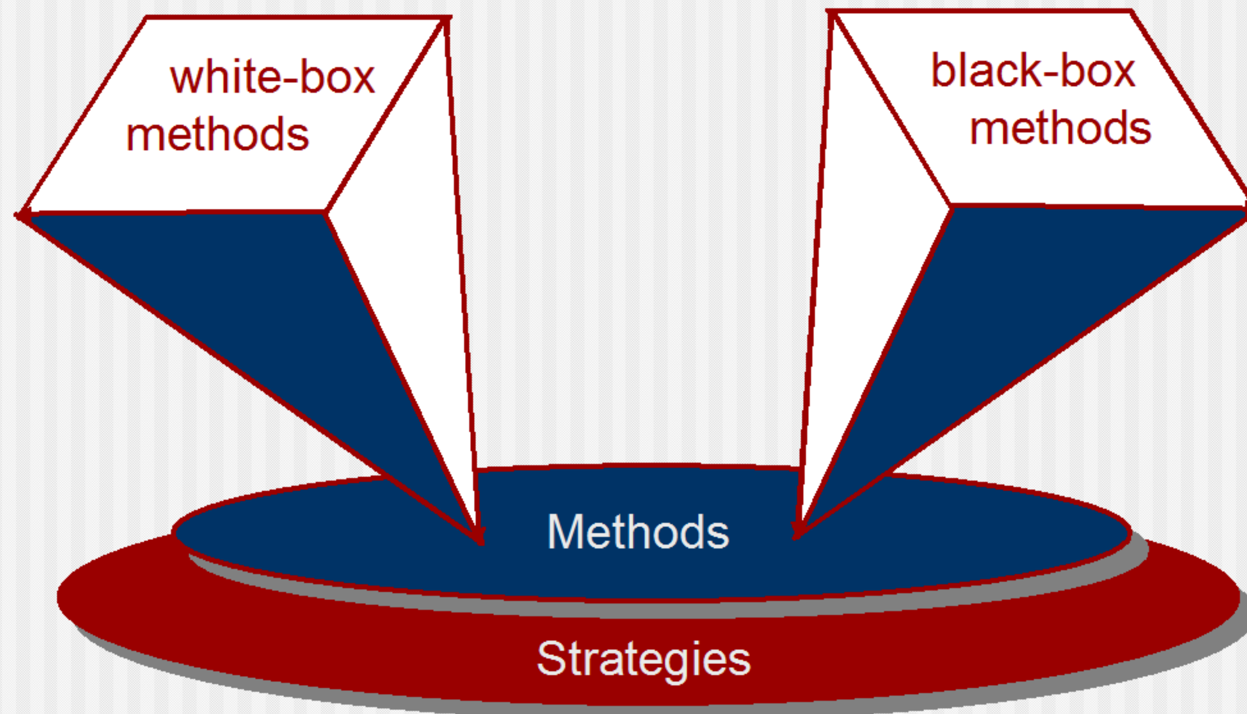
There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

Selective Testing



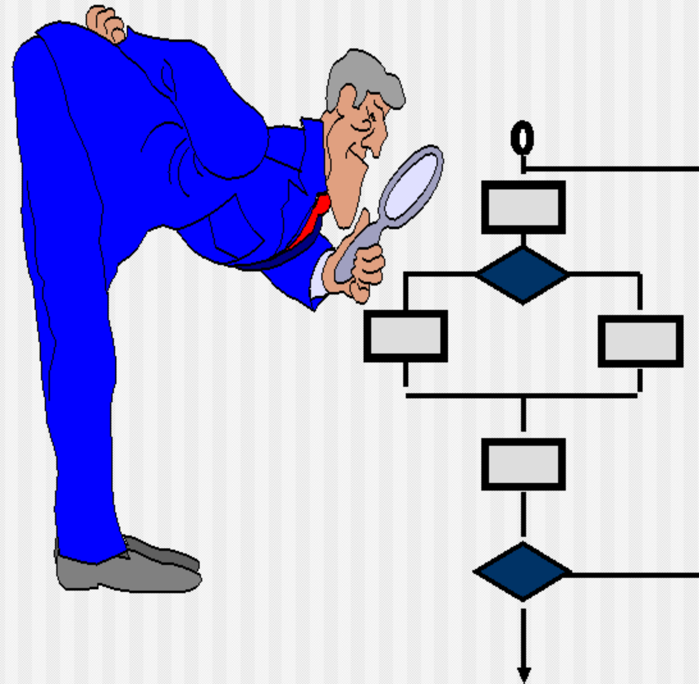
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Software Testing



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

White-Box Testing



... our goal is to ensure that all statements and conditions have been executed at least once ...

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Why Cover?

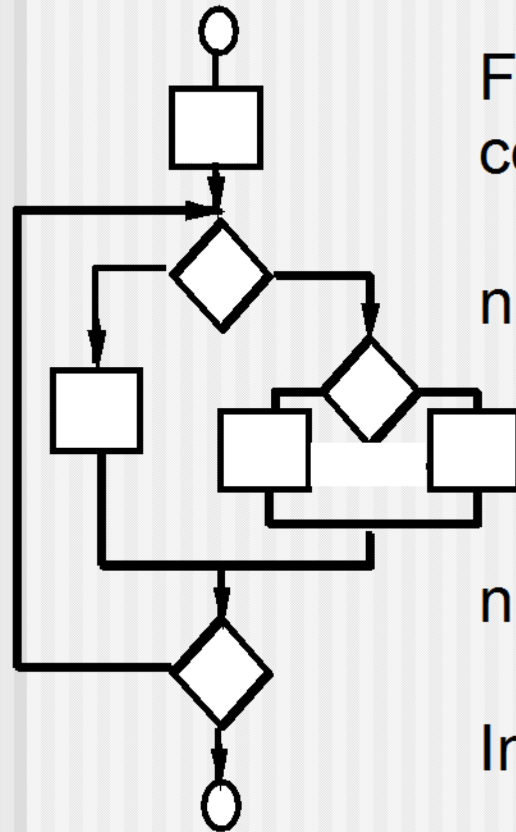
- **logic errors and incorrect assumptions are inversely proportional to a path's execution probability**

- **we often believe that a path is not likely to be executed; in fact, reality is often counter intuitive**

- **typographical errors are random; it's likely that untested paths will contain some**

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Basis Path Testing



First, we compute the cyclomatic complexity:

number of simple decisions + 1

or

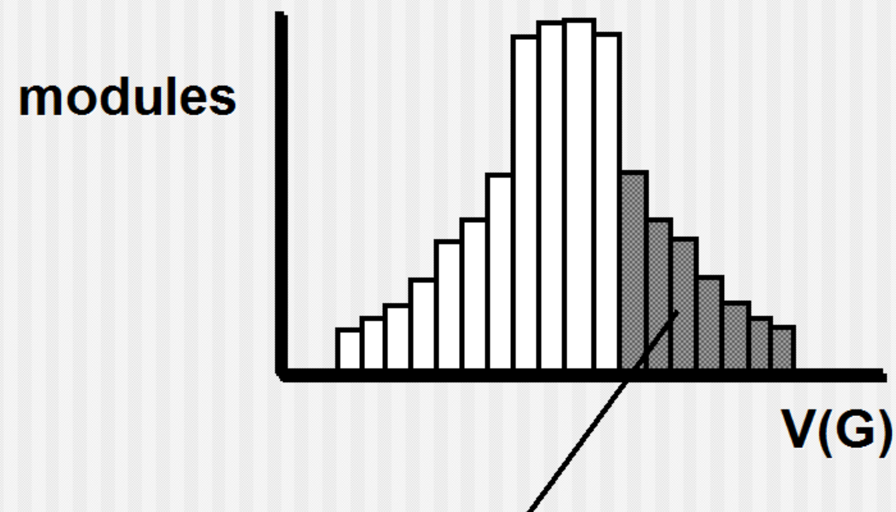
number of enclosed areas + 1

In this case, V(G) = 4

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

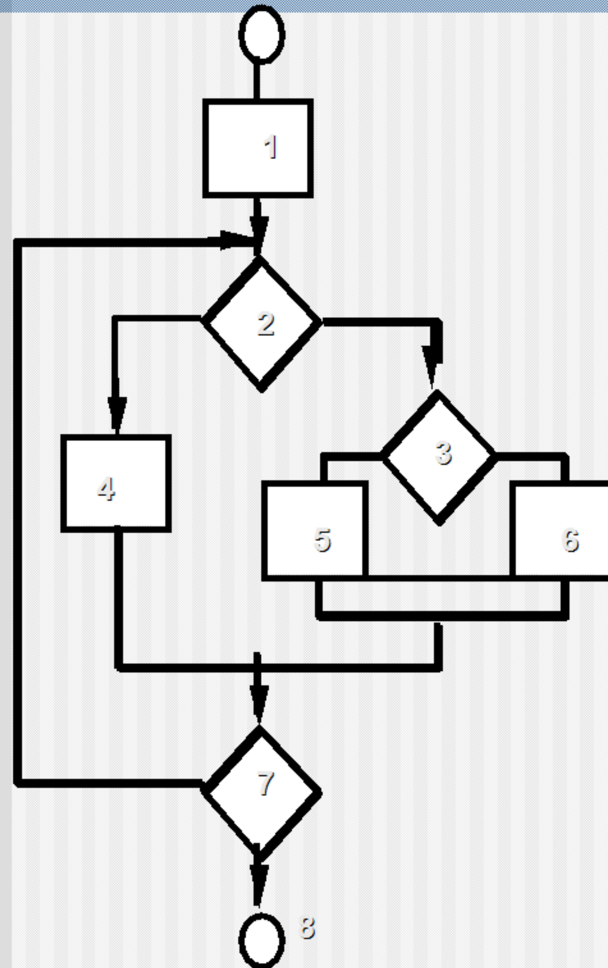
Cyclomatic Complexity

A number of industry studies have indicated that the higher V(G), the higher the probability or errors.



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Basis Path Testing



Next, we derive the independent paths:

Since V(G) = 4, there are four paths

Path 1: 1,2,3,6,7,8

Path 2: 1,2,3,5,7,8

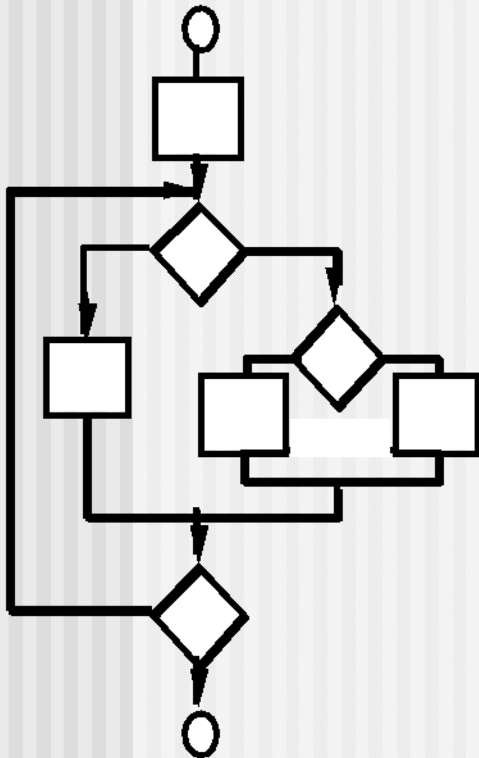
Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Basis Path Testing Notes



- you don't need a flow chart, but the picture will help when you trace program paths
- count each simple logical test, compound tests count as 2 or more
- basis path testing should be applied to critical modules

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Deriving Test Cases

- *Summarizing:*
  - Using the design or code as a foundation, draw a corresponding flow graph.
  - Determine the cyclomatic complexity of the resultant flow graph.
  - Determine a basis set of linearly independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing

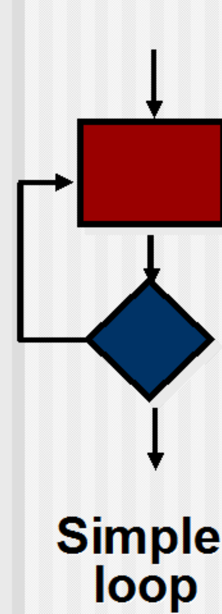
Control Structure Testing

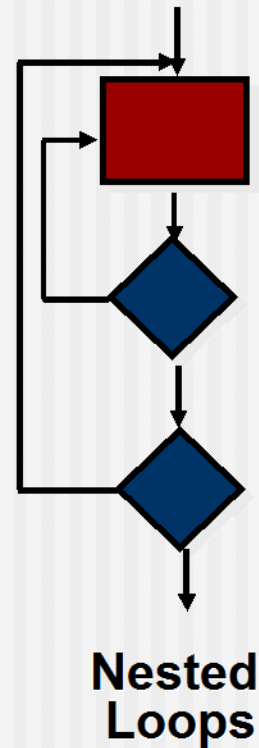
- Condition testing — a test case design method that exercises the logical conditions contained in a program module
- Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program

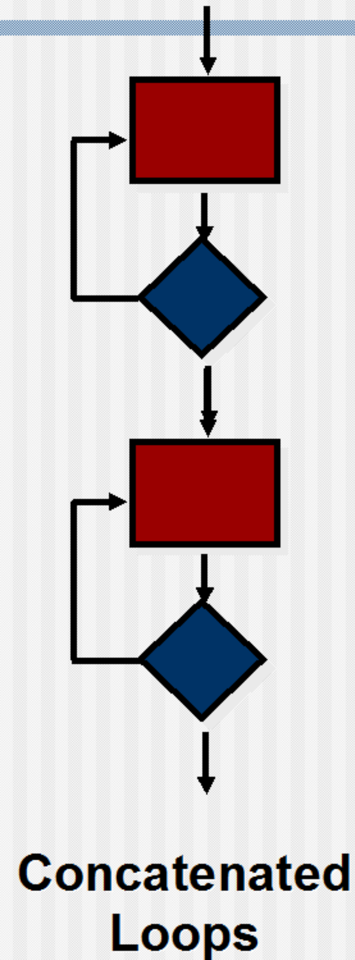
Data Flow Testing

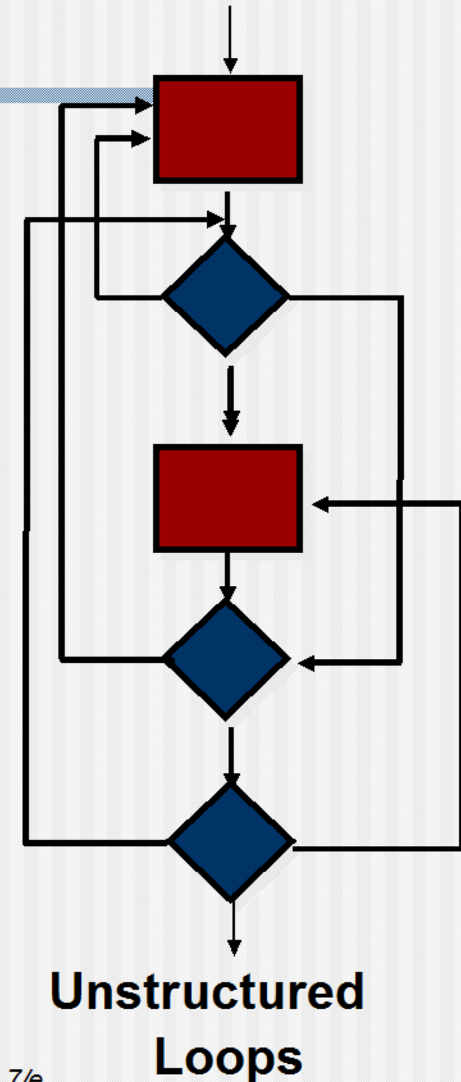
- The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program.
  - Assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with *S* as its statement number
    - DEF(*S*) = {*X* | statement *S* contains a definition of *X*}
    - USE(*S*) = {*X* | statement *S* contains a use of *X*}
  - A *definition-use (DU) chain* of variable X is of the form [*X, S, S'*], where *S* and *S'* are statement numbers, *X* is in DEF(*S*) and USE(*S'*), and the definition of *X* in statement *S* is live at statement *S'*

Loop Testing









These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Loop Testing: Simple Loops

Minimum conditions—Simple Loops

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop $m < n$
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

Loop Testing: Nested Loops

Nested Loops

Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.

Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.

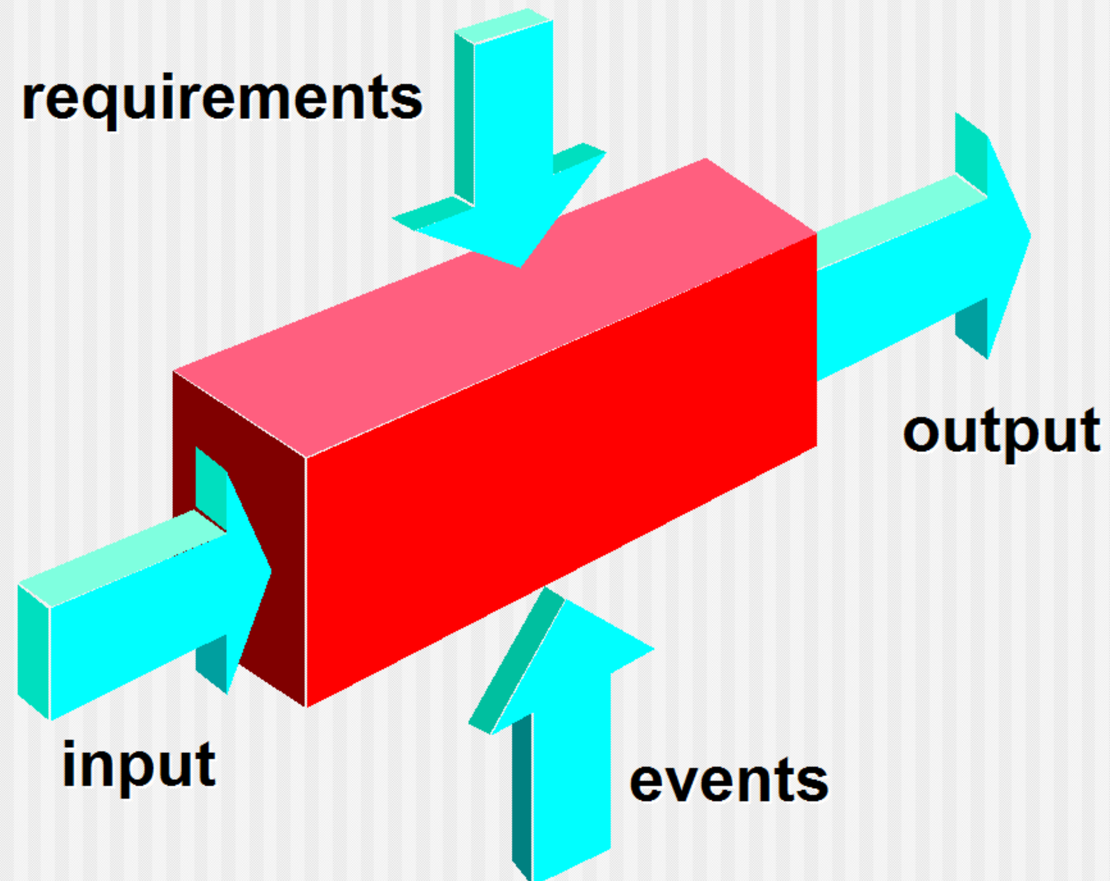
Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

If the loops are independent of one another
then treat each as a simple loop
else* treat as nested loops
endif*

for example, the final loop counter value of loop 1 is used to initialize loop 2.

Black-Box Testing



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

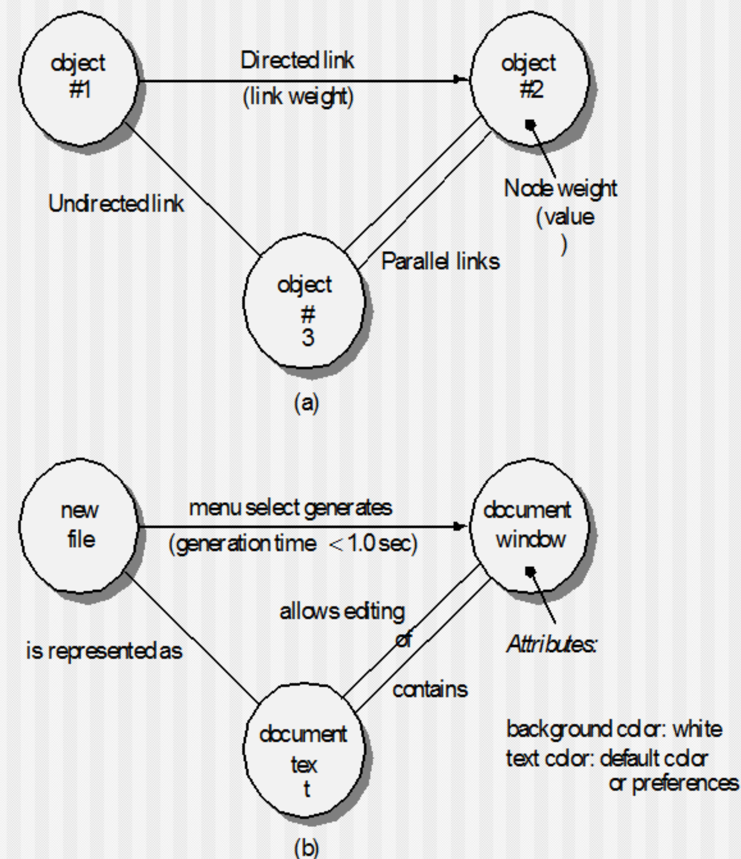
Black-Box Testing

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

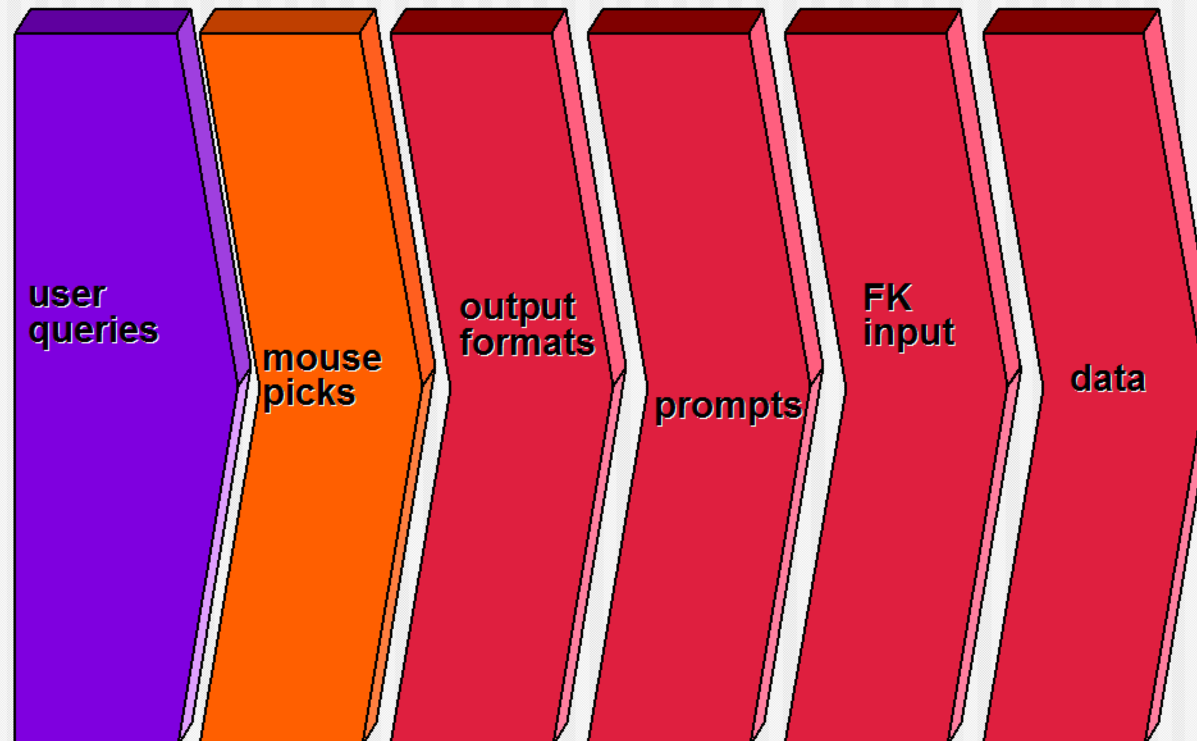
Graph-Based Methods

To understand the objects that are modeled in software and the relationships that connect these objects

In this context, we consider the term "objects" in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.

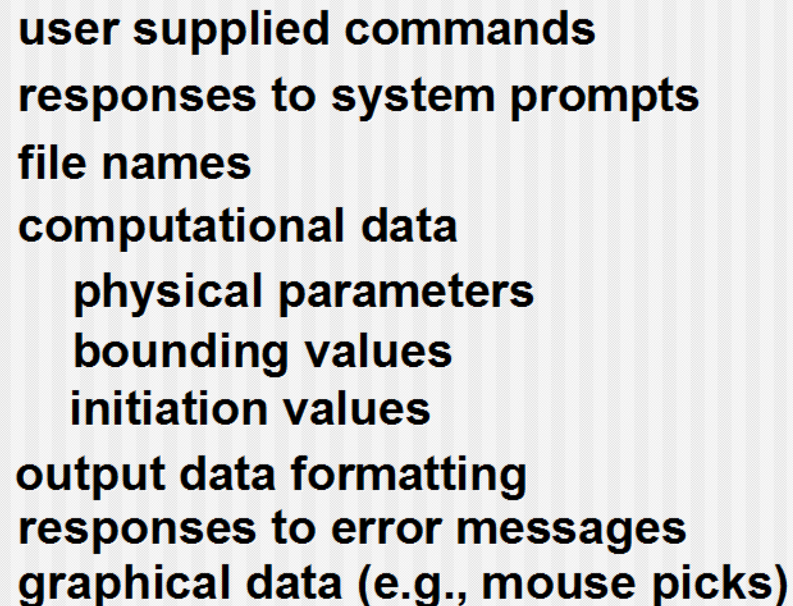


Equivalence Partitioning



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Sample Equivalence Classes



Valid data

user supplied commands

responses to system prompts

file names

computational data

physical parameters

bounding values

initiation values

output data formatting

responses to error messages

graphical data (e.g., mouse picks)

Invalid data

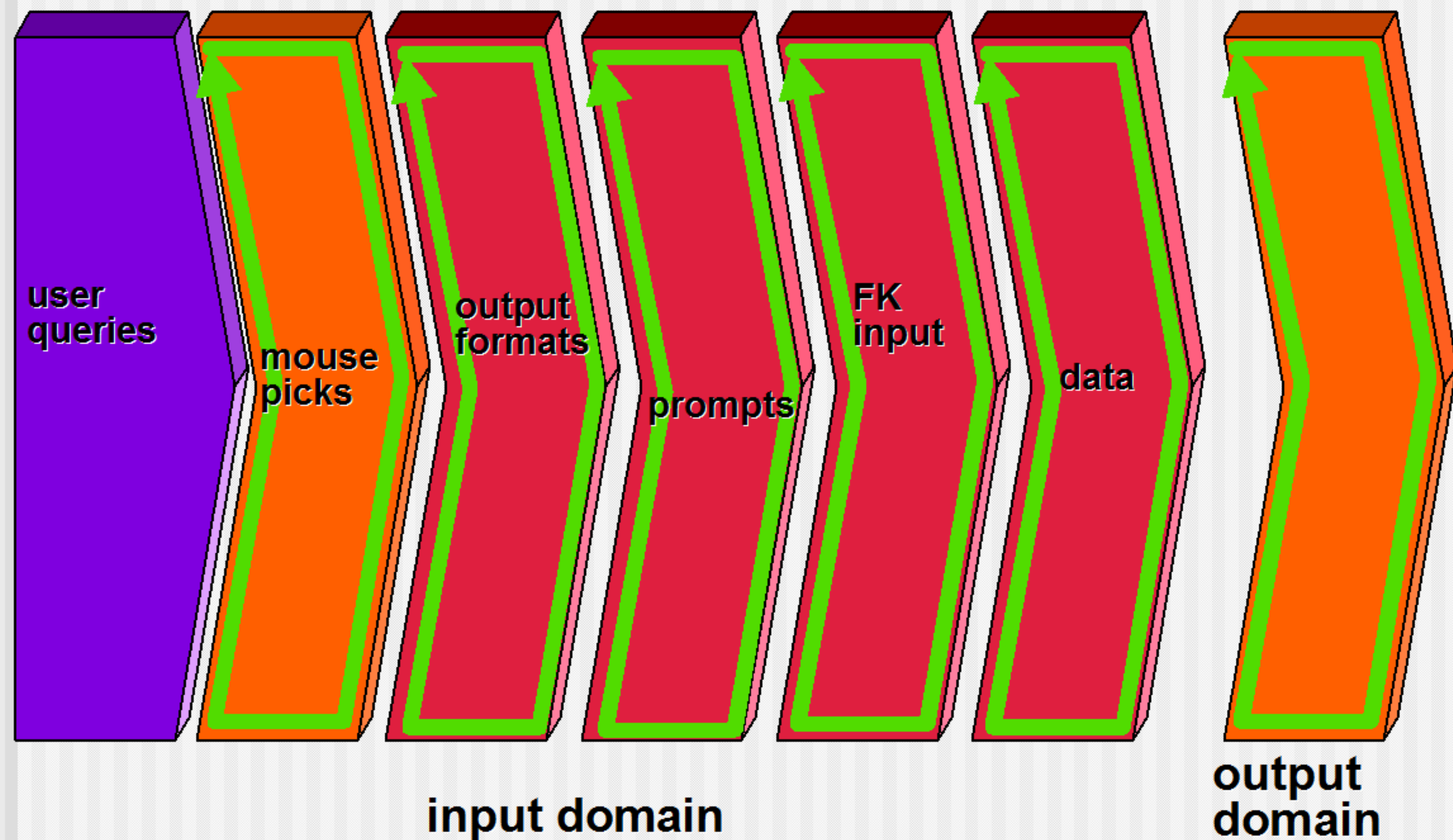
data outside bounds of the program

physically impossible data

proper value supplied in wrong place

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Boundary Value Analysis



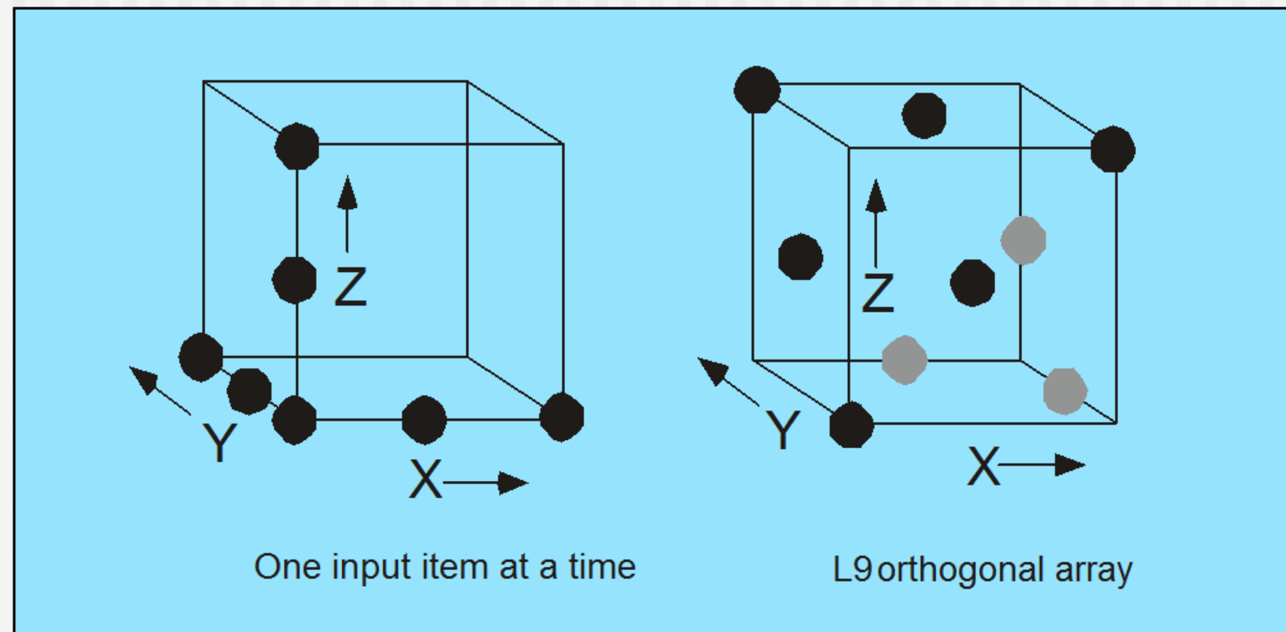
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Comparison Testing

- Used only in situations in which the reliability of software is absolutely critical (e.g., human-rated systems)
  - Separate software engineering teams develop independent versions of an application using the same specification
  - Each version can be tested with the same test data to ensure that all provide identical output
  - Then all versions are executed in parallel with real-time comparison of results to ensure consistency

Orthogonal Array Testing

- Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded



One input item at a time

L9 orthogonal array

Model-Based Testing

- Analyze an existing behavioral model for the software or create one.
  - Recall that a *behavioral model* indicates how software will respond to external events or stimuli.
- Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.
  - The inputs will trigger events that will cause the transition to occur.
- Review the behavioral model and note the expected outputs as the software makes the transition from state to state.
- Execute the test cases.
- Compare actual and expected results and take corrective action as required.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Software Testing Patterns

- Testing patterns are described in much the same way as design patterns (Chapter 12).
- *Example:*
  - *Pattern name:* **ScenarioTesting**
  - *Abstract:* Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The **ScenarioTesting** pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement. [Kan01]

Chapter 19

- **Testing Object-Oriented Applications**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

OO Testing

- To adequately test OO systems, three things must be done:
  - the definition of testing must be broadened to include error discovery techniques applied to object-oriented analysis and design models
  - the strategy for unit and integration testing must change significantly, and
  - the design of test cases must account for the unique characteristics of OO software.

‘Testing’ OO Models

- The review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level
- Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side affects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

Correctness of OO Models

- During analysis and design, semantic correctness can be asesssed based on the model's conformance to the real world problem domain.
- If the model accurately reflects the real world (to a level of detail that is appropriate to the stage of development at which the model is reviewed) then it is semantically correct.
- To determine whether the model does, in fact, reflect real world requirements, it should be presented to problem domain experts who will examine the class definitions and hierarchy for omissions and ambiguity.
- Class relationships (instance connections) are evaluated to determine whether they accurately reflect real-world object connections.

Class Model Consistency

- Revisit the CRC model and the object-relationship model.
- Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
- Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
- Using the inverted connections examined in the preceding step, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
- Determine whether widely requested responsibilities might be combined into a single responsibility.

OO Testing Strategies

- Unit testing
  - the concept of the unit changes
  - the smallest testable unit is the encapsulated class
  - a single operation can no longer be tested in isolation (the conventional view of unit testing) but rather, as part of a class
- Integration Testing
  - *Thread-based testing* integrates the set of classes required to respond to one input or event for the system
  - *Use-based testing* begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) of server classes. After the independent classes are tested, the next layer of classes, called *dependent classes*
  - *Cluster testing* [McG94] defines a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

OO Testing Strategies

- Validation Testing
  - details of class connections disappear
  - draw upon use cases (Chapters 5 and 6) that are part of the requirements model
  - Conventional black-box testing methods (Chapter 18) can be used to drive validation tests

(McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

OOT Methods

Berard [Ber93] proposes the following approach:

1. **Each test case should be uniquely identified and should be explicitly associated with the class to be tested,**
2. **The purpose of the test should be stated,**
3. **A list of testing steps should be developed for each test and should contain [BER94]:**
   a. **a list of specified states for the object that is to be tested**
   b. **a list of messages and operations that will be exercised as a consequence of the test**
   c. **a list of exceptions that may occur as the object is tested**
   d. **a list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)**
   e. **supplementary information that will aid in understanding or implementing the test.**

Testing Methods

- Fault-based testing
  - The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.
- Class Testing and the Class Hierarchy
  - Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.
- Scenario-Based Test Design
  - Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests.

OOT Methods: Random Testing

- Random testing
  - identify operations applicable to a class
  - define constraints on their use
  - identify a minimum test sequence
    - an operation sequence that defines the minimum life history of the class (object)
  - generate a variety of random (but valid) test sequences
    - exercise other (more complex) class instance life histories

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

OOT Methods: Partition Testing

- Partition Testing
  - reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software
  - state-based partitioning
    - categorize and test operations based on their ability to change the state of a class
  - attribute-based partitioning
    - categorize and test operations based on the attributes that they use
  - category-based partitioning
    - categorize and test operations based on the generic function each performs

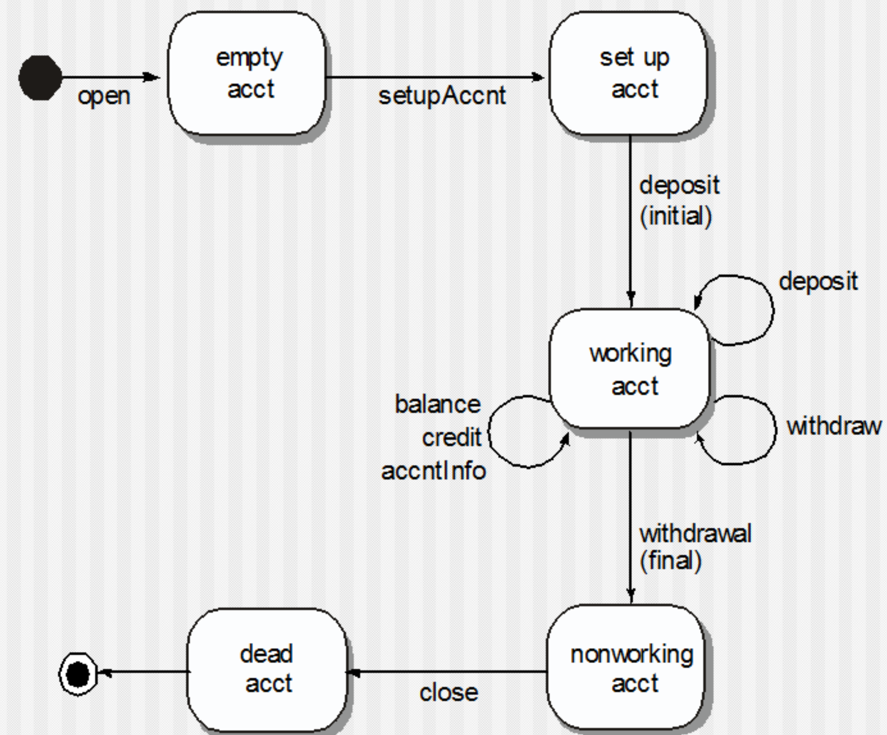
OOT Methods: Inter-Class Testing

- Inter-class testing
  - For each client class, use the list of class operators to generate a series of random test sequences. The operators will send messages to other server classes.
  - For each message that is generated, determine the collaborator class and the corresponding operator in the server object.
  - For each operator in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
  - For each of the messages, determine the next level of operators that are invoked and incorporate these into the test sequence

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

OOT Methods: Behavior Testing

The tests to be designed should achieve all state coverage [KIR94]. That is, the operation sequences should cause the Account class to make transition through all allowable states



Chapter 20

- **Testing Web Applications**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

Testing Quality Dimensions-I

- *Content* is evaluated at both a syntactic and semantic level.
  - syntactic level—spelling, punctuation and grammar are assessed for text-based documents.
  - semantic level—correctness (of information presented), consistency (across the entire content object and related objects) and lack of ambiguity are all assessed.
- *Function* is tested for correctness, instability, and general conformance to appropriate implementation standards (e.g.,Java or XML language standards).
- *Structure* is assessed to ensure that it
  - properly delivers WebApp content and function
  - is extensible
  - can be supported as new content or functionality is added.

Testing Quality Dimensions-II

- *Usability* is tested to ensure that each category of user
  - is supported by the interface
  - can learn and apply all required navigation syntax and semantics
- *Navigability* is tested to ensure that
  - all navigation syntax and semantics are exercised to uncover any navigation errors (e.g., dead links, improper links, erroneous links).
- *Performance* is tested under a variety of operating conditions, configurations, and loading to ensure that
  - the system is responsive to user interaction
  - the system handles extreme loading without unacceptable operational degradation

Testing Quality Dimensions-III

- *Compatibility* is tested by executing the WebApp in a variety of different host configurations on both the client and server sides.
  - The intent is to find errors that are specific to a unique host configuration.
- *Interoperability* is tested to ensure that the WebApp properly interfaces with other applications and/or databases.
- *Security* is tested by assessing potential vulnerabilities and attempting to exploit each.
  - Any successful penetration attempt is deemed a security failure.

Errors in a WebApp

- Because many types of WebApp tests uncover problems that are first evidenced on the client side, you often see a symptom of the error, not the error itself.
- Because a WebApp is implemented in a number of different configurations and within different environments, it may be difficult or impossible to reproduce an error outside the environment in which the error was originally encountered.
- Although some errors are the result of incorrect design or improper HTML (or other programming language) coding, many errors can be traced to the WebApp configuration.
- Because WebApps reside within a client/server architecture, errors can be difficult to trace across three architectural layers: the client, the server, or the network itself.
- Some errors are due to the *static operating environment* (i.e., the specific configuration in which testing is conducted), while others are attributable to the dynamic operating environment (i.e., instantaneous resource loading or time-related errors).

WebApp Testing Strategy-I

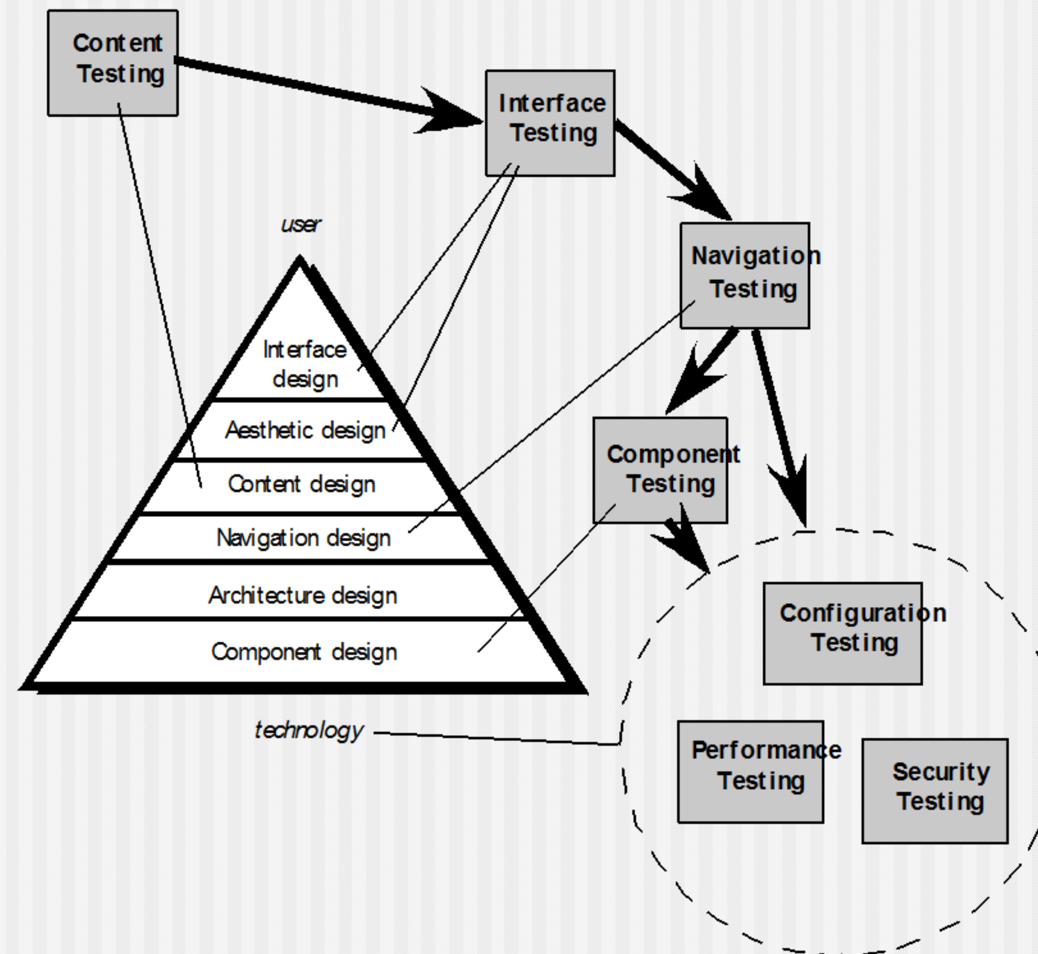
- The content model for the WebApp is reviewed to uncover errors.
- The interface model is reviewed to ensure that all use-cases can be accommodated.
- The design model for the WebApp is reviewed to uncover navigation errors.
- The user interface is tested to uncover errors in presentation and/or navigation mechanics.
- Selected functional components are unit tested.

WebApp Testing Strategy-II

- Navigation throughout the architecture is tested.
- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
- Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
- Performance tests are conducted.
- The WebApp is tested by a controlled and monitored population of end-users
  - the results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

The Testing Process



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Content Testing

- Content testing has three important objectives:
  - to uncover syntactic errors (e.g., typos, grammar mistakes) in text-based documents, graphical representations, and other media
  - to uncover semantic errors (i.e., errors in the accuracy or completeness of information) in any content object presented as navigation occurs, and
  - to find errors in the organization or structure of content that is presented to the end-user.

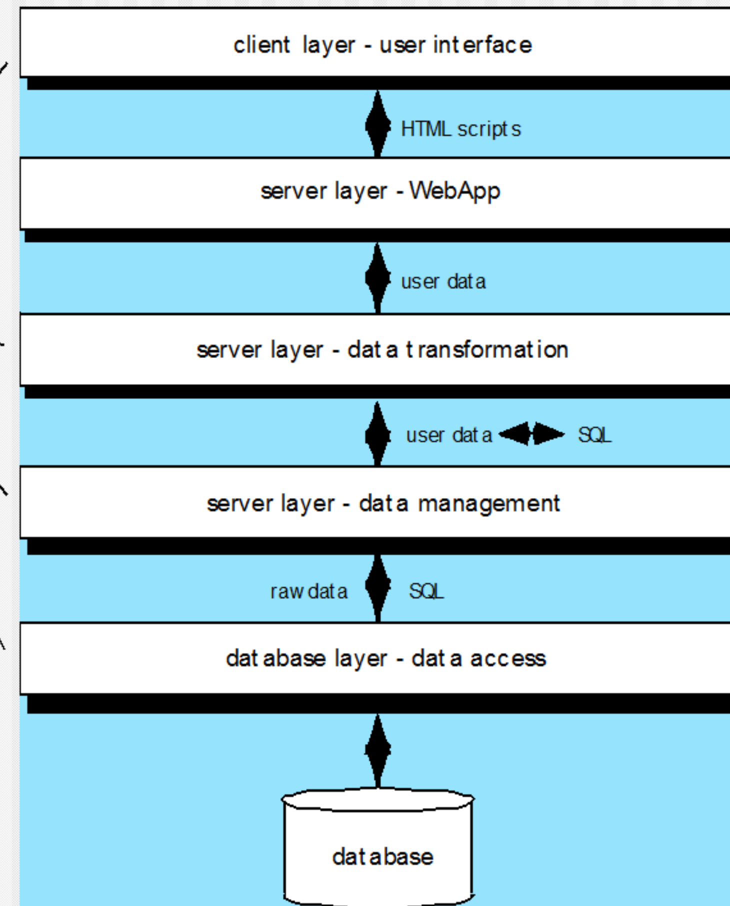
Assessing Content Semantics

- Is the information factually accurate?
- Is the information concise and to the point?
- Is the layout of the content object easy for the user to understand?
- Can information embedded within a content object be found easily?
- Have proper references been provided for all information derived from other sources?
- Is the information presented consistent internally and consistent with information presented in other content objects?
- Is the content offensive, misleading, or does it open the door to litigation?
- Does the content infringe on existing copyrights or trademarks?
- Does the content contain internal links that supplement existing content? Are the links correct?
- Does the aesthetic style of the content conflict with the aesthetic style of the interface?

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Database Testing

Tests are defined for each layer



User Interface Testing

- Interface features are tested to ensure that design rules, aesthetics, and related visual content is available for the user without error.
- Individual interface mechanisms are tested in a manner that is analogous to unit testing.
- Each interface mechanism is tested within the context of a use-case or NSU for a specific user category.
- The complete interface is tested against selected use-cases and NSUs to uncover errors in the semantics of the interface.
- The interface is tested within a variety of environments (e.g., browsers) to ensure that it will be compatible.

Testing Interface Mechanisms-I

- *Links*—navigation mechanisms that link the user to some other content object or function.
- *Forms*—a structured document containing blank fields that are filled in by the user. The data contained in the fields are used as input to one or more WebApp functions.
- *Client-side scripting*—a list of programmed commands in a scripting language (e.g., Javascript) that handle information input via forms or other user interactions
- *Dynamic HTML*—leads to content objects that are manipulated on the client side using scripting or cascading style sheets (CSS).
- *Client-side pop-up windows*—small windows that pop-up without user interaction. These windows can be content-oriented and may require some form of user interaction.

Testing Interface Mechanisms-II

- *CGI scripts*—a common gateway interface (CGI) script implements a standard method that allows a Web server to interact dynamically with users (e.g., a WebApp that contains forms may use a CGI script to process the data contained in the form once it is submitted by the user).
- *Streaming content*—rather than waiting for a request from the client-side, content objects are downloaded automatically from the server side. This approach is sometimes called "push" technology because the server pushes data to the client.
- *Cookies*—a block of data sent by the server and stored by a browser as a consequence of a specific user interaction. The content of the data is WebApp-specific (e.g., user identification data or a list of items that have been selected for purchase by the user).
- *Application specific interface mechanisms*—include one or more "macro" interface mechanisms such as a shopping cart, credit card processing, or a shipping cost calculator.

Usability Tests

- Design by WebE team … executed by end-users
- Testing sequence …
  - Define a set of usability testing categories and identify goals for each.
  - Design tests that will enable each goal to be evaluated.
  - Select participants who will conduct the tests.
  - Instrument participants' interaction with the WebApp while testing is conducted.
  - Develop a mechanism for assessing the usability of the WebApp
- different levels of abstraction:
  - the usability of a specific interface mechanism (e.g., a form) can be assessed
  - the usability of a complete Web page (encompassing interface mechanisms, data objects and related functions) can be evaluated
  - the usability of the complete WebApp can be considered.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Compatibility Testing

- Compatibility testing is to define a set of “commonly encountered” client side computing configurations and their variants
- Create a tree structure identifying
  - each computing platform
  - typical display devices
  - the operating systems supported on the platform
  - the browsers available
  - likely Internet connection speeds
  - similar information.
- Derive a series of compatibility validation tests
  - derived from existing interface tests, navigation tests, performance tests, and security tests.
  - intent of these tests is to uncover errors or execution problems that can be traced to configuration differences.

Component-Level Testing

- Focuses on a set of tests that attempt to uncover errors in WebApp functions
- Conventional black-box and white-box test case design methods can be used
- Database testing is often an integral part of the component-testing regime

 Slides copyright 2009 by Roger Pressman.

Navigation Testing

- The following navigation mechanisms should be tested:
  - *Navigation links*—these mechanisms include internal links within the WebApp, external links to other WebApps, and anchors within a specific Web page.
  - *Redirects*—these links come into play when a user requests a non-existent URL or selects a link whose destination has been removed or whose name has changed.
  - *Bookmarks*—although bookmarks are a browser function, the WebApp should be tested to ensure that a meaningful page title can be extracted as the bookmark is created.
  - *Frames and framesets*—tested for correct content, proper layout and sizing, download performance, and browser compatibility
  - *Site maps*—Each site map entry should be tested to ensure that the link takes the user to the proper content or functionality.
  - *Internal search engines*—Search engine testing validates the accuracy and completeness of the search, the error-handling properties of the search engine, and advanced search features

Testing Navigation Semantics-I

- Is the NSU achieved in its entirety without error?
- Is every navigation node (defined for a NSU) reachable within the context of the navigation paths defined for the NSU?
- If the NSU can be achieved using more than one navigation path, has every relevant path been tested?
- If guidance is provided by the user interface to assist in navigation, are directions correct and understandable as navigation proceeds?
- Is there a mechanism (other than the browser 'back' arrow) for returning to the preceding navigation node and to the beginning of the navigation path.
- Do mechanisms for navigation within a large navigation node (i.e., a long web page) work properly?
- If a function is to be executed at a node and the user chooses not to provide input, can the remainder of the NSU be completed?

Testing Navigation Semantics-II

- If a function is executed at a node and an error in function processing occurs, can the NSU be completed?
- Is there a way to discontinue the navigation before all nodes have been reached, but then return to where the navigation was discontinued and proceed from there?
- Is every node reachable from the site map? Are node names meaningful to end-users?
- If a node within an NSU is reached from some external source, is it possible to process to the next node on the navigation path. Is it possible to return to the previous node on the navigation path?
- Does the user understand his location within the content architecture as the NSU is executed?

Configuration Testing

- Server-side
  - Is the WebApp fully compatible with the server OS?
  - Are system files, directories, and related system data created correctly when the WebApp is operational?
  - Do system security measures (e.g., firewalls or encryption) allow the WebApp to execute and service users without interference or performance degradation?
  - Has the WebApp been tested with the distributed server configuration (if one exists) that has been chosen?
  - Is the WebApp properly integrated with database software? Is the WebApp sensitive to different versions of database software?
  - Do server-side WebApp scripts execute properly?
  - Have system administrator errors been examined for their affect on WebApp operations?
  - If proxy servers are used, have differences in their configuration been addressed with on-site testing?

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Configuration Testing

- Client-side
  - *Hardware*—CPU, memory, storage and printing devices
  - *Operating systems*—Linux, Macintosh OS, Microsoft Windows, a mobile-based OS
  - *Browser software*—Internet Explorer, Mozilla/Netscape, Opera, Safari, and others
  - *User interface components*—Active X, Java applets and others
  - *Plug-ins*—QuickTime, RealPlayer, and many others
  - *Connectivity*—cable, DSL, regular modem, T1
- The number of configuration variables must be reduced to a manageable number

Security Testing

- Designed to probe vulnerabilities of the client-side environment, the network communications that occur as data are passed from client to server and back again, and the server-side environment
- On the client-side, vulnerabilities can often be traced to pre-existing bugs in browsers, e-mail programs, or communication software.
- On the server-side, vulnerabilities include denial-of-service attacks and malicious scripts that can be passed along to the client-side or used to disable server operations

Performance Testing

- Does the server response time degrade to a point where it is noticeable and unacceptable?
- At what point (in terms of users, transactions or data loading) does performance become unacceptable?
- What system components are responsible for performance degradation?
- What is the average response time for users under a variety of loading conditions?
- Does performance degradation have an impact on system security?
- Is WebApp reliability or accuracy affected as the load on the system grows?
- What happens when loads that are greater than maximum server capacity are applied?

Load Testing

- The intent is to determine how the WebApp and its server-side environment will respond to various loading conditions
  - *N*, the number of concurrent users
  - *T*, the number of on-line transactions per unit of time
  - *D*, the data load processed by the server per transaction
- Overall throughput, *P*, is computed in the following manner:
  - $P = N \times T \times D$

Stress Testing

- Does the system degrade 'gently' or does the server shut down as capacity is exceeded?
- Does server software generate "server not available" messages? More generally, are users aware that they cannot reach the server?
- Does the server queue requests for resources and empty the queue once capacity demands diminish?
- Are transactions lost as capacity is exceeded?
- Is data integrity affected as capacity is exceeded?
- What values of *N, T,* and *D* force the server environment to fail? How does failure manifest itself? Are automated notifications sent to technical support staff at the server site?
- If the system does fail, how long will it take to come back on-line?
- Are certain WebApp functions (e.g., compute intensive functionality, data streaming capabilities) discontinued as capacity reaches the 80 or 90 percent level?

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Chapter 21

■ Formal Modeling and Verification

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e

by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

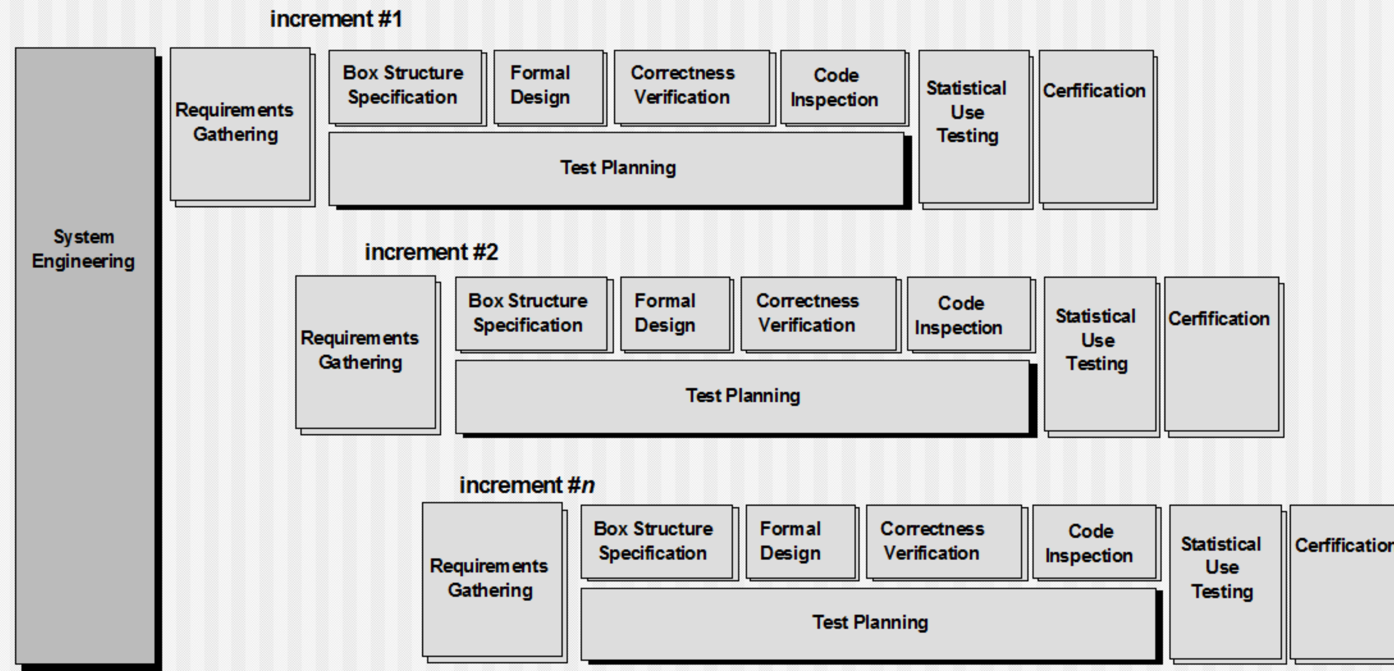
May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

Formal Modeling and Verification

- *Cleanroom software engineering* and *formal methods*
  - Both demand a specialized specification approach and each applies a unique verification method.
  - Both are quite rigorous and neither is used widely by the software engineering community.
- If you must build "bullet-proof" software, these methods can help immeasurably.

The Cleanroom Process Model



The Cleanroom Strategy-I

Increment Planning—adopts the incremental strategy

Requirements Gathering—defines a description of customer level requirements (for each increment)

Box Structure Specification—describes the functional specification

Formal Design—specifications (called “black boxes”) are iteratively refined (with an increment) to become analogous to architectural and procedural designs (called “state boxes” and “clear boxes,” respectively).

Correctness Verification—verification begins with the highest level box structure (specification) and moves toward design detail and code using a set of “correctness questions.” If these do not demonstrate that the specification is correct, more formal (mathematical) methods for verification are used.

Code Generation, Inspection and Verification—the box structure specifications, represented in a specialized language, are transmitted into the appropriate programming language.

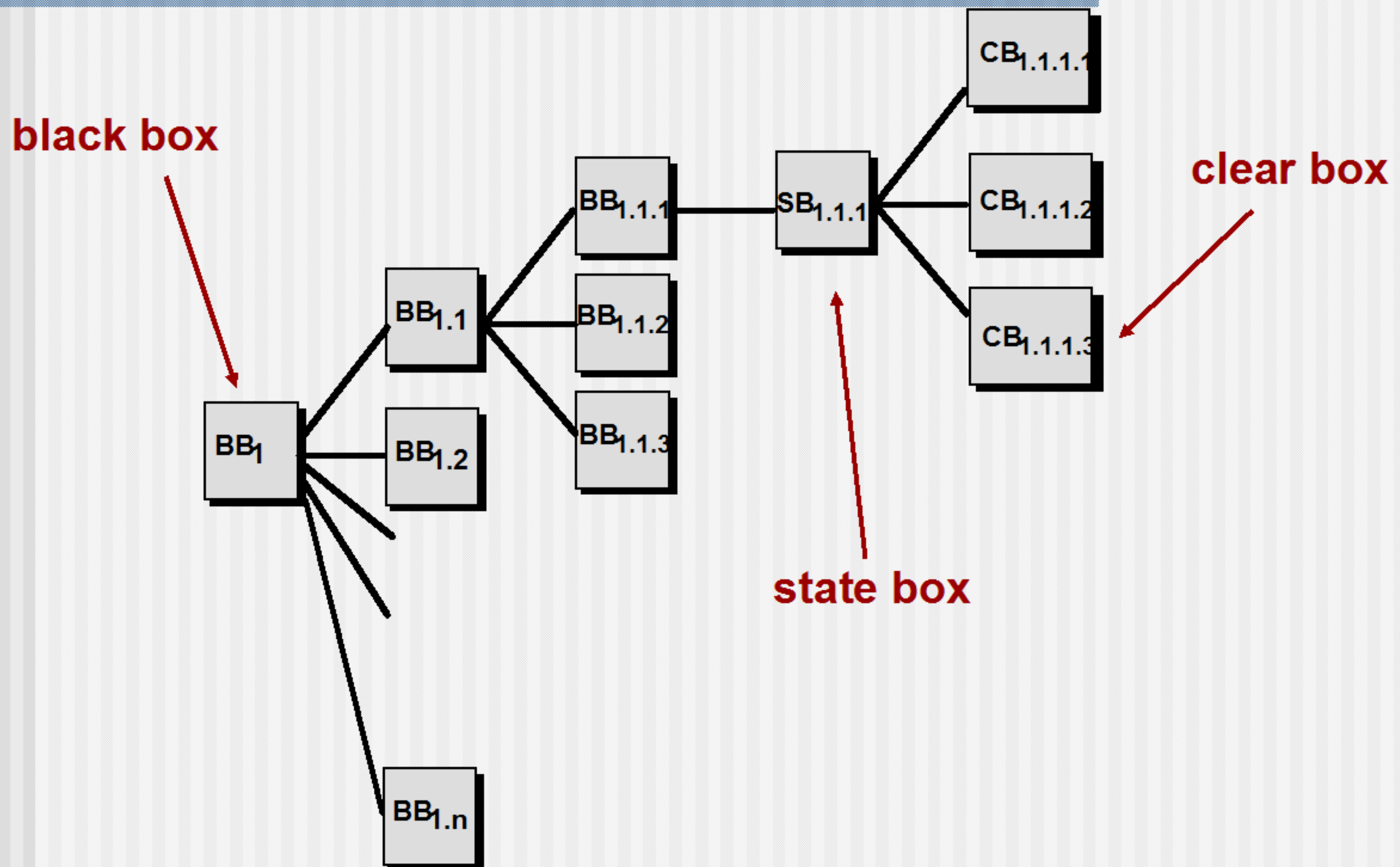
The Cleanroom Strategy-II

Statistical Test Planning—a suite of test cases that exercise of "probability distribution" of usage are planned and designed

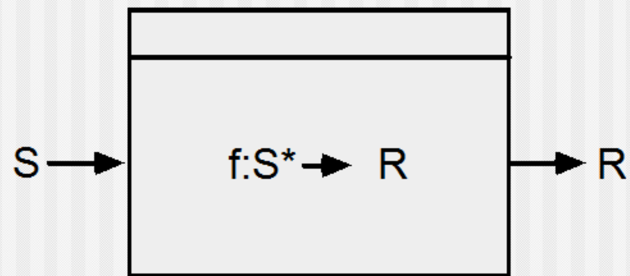
Statistical Usage Testing—execute a series of tests derived from a statistical sample (the probability distribution noted above) of all possible program executions by all users from a targeted population

Certification—once verification, inspection and usage testing have been completed (and all errors are corrected) the increment is certified as ready for integration.

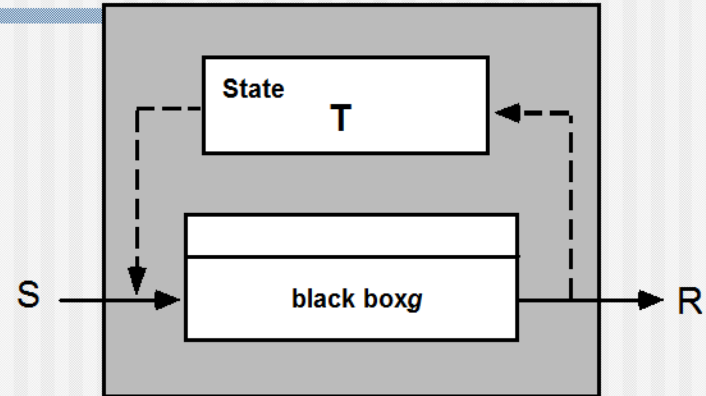
Box Structure Specification



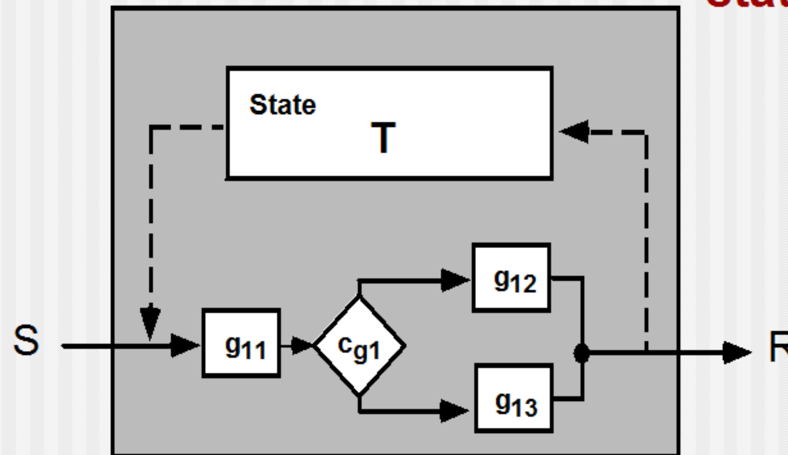
Box Structures



black box



state box



clear box

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Design Refinement & Verification

If a function f is expanded into a sequence g and h, the correctness condition for all input to f is:

- **Does g followed by h do f?**

When a function f is refined into a conditional (if-then-else), the correctness condition for all input to f is:

- **Whenever condition <c> is true does g do f and whenever <c> is false, does h do f?**

When function f is refined as a loop, the correctness conditions for all input to f is:

- **Is termination guaranteed?**
- **Whenever <c> is true does g followed by f do f, and whenever <c> is false, does skipping the loop still do f?**

Advantages of Design Verification

- It reduces verification to a finite process.
- It lets cleanroom teams verify every line of design and code.
- It results in a near zero defect level.
- It scales up.
- It produces better code than unit testing.

Cleanroom Testing

- statistical use testing
  - tests the actual usage of the program
- determine a "usage probability distribution"
  - analyze the specification to identify a set of stimuli
  - stimuli cause software to change behavior
  - create usage scenarios
  - assign probability of use to each stimuli
  - test cases are generated for each stimuli according to the usage probability distribution

Certification

1. Usage scenarios must be created.
2. A usage profile is specified.
3. Test cases are generated from the profile.
4. Tests are executed and failure data are recorded and analyzed.
5. Reliability is computed and certified.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Certification Models

Sampling model. Software testing executes *m* random test cases and is certified if no failures or a specified numbers of failures occur. The value of *m* is derived mathematically to ensure that required reliability is achieved.

Component model. A system composed of *n* components is to be certified. The component model enables the analyst to determine the probability that component *i* will fail prior to completion.

Certification model. The overall reliability of the system is projected and certified.

Formal Methods

- *"Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, and verify systems in a systematic, rather than ad hoc manner."*

  ***The Encyclopedia of Software Engineering* [Mar01]**

- The Problem with conventional specs:
  - contradictions
  - ambiguities
  - vagueness
  - incompleteness
  - mixed levels of abstraction

Formal Specification

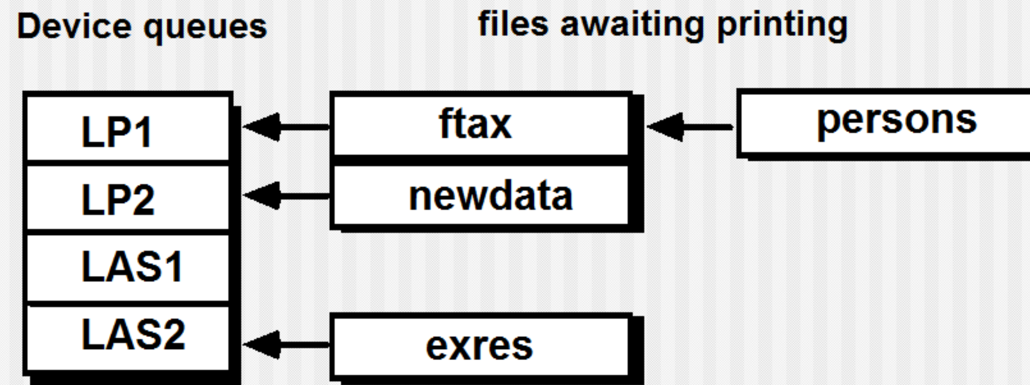
- Desired properties—consistency, completeness, and lack of ambiguity—are the objectives of all specification methods
- The formal syntax of a specification language enables requirements or design to be interpreted in only one way, eliminating ambiguity that often occurs when a natural language (e.g., English) or a graphical notation must be interpreted
  - The descriptive facilities of set theory and logic notation enable clear statement of facts (requirements).
- Consistency is ensured by mathematically proving that initial facts can be formally mapped (using inference rules) into later statements within the specification.

Formal Methods Concepts

- *data invariant*—a condition that is true throughout the execution of the system that contains a collection of data
- *state*
  - Many formal languages, such as OCL (Section 28.5) , use the notion of states as they were discussed in Chapters 7 and 8, that is, a system can be in one of several states, each representing an externally observable mode of behavior.
  - The Z language (Section 28.6)defines a *state* as the stored data which a system accesses and alters
- *operation*—an action that takes place in a system and reads or writes data to a state
  - *precondition* defines the circumstances in which a particular operation is valid
  - *postcondition* defines what happens when an operation has completed its action

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

An Example—Print Spooler



Limits

LP1 -> 750
LP2 -> 500
LAS1 -> 300
LAS2 -> 200

Size

newdata -> 450
ftax -> 650
exres -> 50
persons -> 700

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

States and Data Invariant

The state of the spooler is represented by the four components *Queues, OutputDevices, Limits,* and *Sizes.*

The data invariant has five components:

- Each output device is associated with an upper limit of print lines
- Each output device is associated with a possibly nonempty queue of files awaiting printing
- Each file is associated with a size
- Each queue associated with an output device contains files that have a size less than the upper limit of the output device
- There will be no more than *MaxDevs* output devices administered by the spooler

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Operations

- An operation which adds a new output device to the spooler together with its associated print limit
- An operation which removes a file from the queue associated with a particular output device
- An operation which adds a file to the queue associated with a particular output device
- An operation which alters the upper limit of print lines for a particular output device
- An operation which moves a file from a queue associated with an output device to another queue associated with a second output device

Pre- & Postconditions

For the first operation (adds a new output device to the spooler together with its associated print limit):

Precondition: the output device name does not already exist and that there are currently less than *MaxDevs* output devices known to the spooler

Postcondition: the name of the new device is added to the collection of existing device names, a new entry is formed for the device with no files being associated with its queue, and the device is associated with its print limit.

Mathematical Concepts*

- sets and constructive set specification
- set operators
- logic operators
  - e.g., $\forall$ i, j: $\mathbb{N}$• i > j i^2 => j^2
  - which states that, for every pair of values in the set of natural numbers, if i is greater than j, then i^2 is greater than j^2.
- sequences

*A discussion of sets and constructive specification (slides 20 - 24) is no longer included within SEPA, 7/e, but is included here for those who are unfamiliar with the basic concepts.

Sets and Constructive Specification

- A *set* is a collection of objects or elements and is used as a cornerstone of formal methods.
  - Enumeration
    - {C++, Pascal, Ada, COBOL, Java}
    - #{C++, Pascal, Ada, COBOL, Java} implies *cardinality* = 5
  - Constructive set specification is preferable to enumeration because it enables a succinct definition of large sets.
    - $\{x, y : N \mid x + y = 10 \quad (x, y^2)\}$

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Set Operators

- A specialized set of symbology is used to represent set and logic operations.
  - Examples
    - The P operator is used to indicate membership of a set. For example, the expression
      - x P X
    - The operators , , and # take sets as their operands. The predicate
      - A , B
    - has the value *true* if the members of the set A are contained in the set B and has the value *false* otherwise.
  - The union operator, <, takes two sets and forms a set that contains all the elements in the set with duplicates eliminated.
    - {File1, File2, Tax, Compiler} < {NewTax, D2, D3, File2} is the set
    - {Filel, File2, Tax, Compiler, NewTax, D2, D3}

Logic Operators

- Another important component of a formal method is logic: the algebra of true and false expressions.
  - Examples:
    - V or
    - ¬ not
    - => implies
- Universal quantification is a way of making a statement about the elements of a set that is true for every member of the set. Universal quantification uses the symbol, . An example of its use is
    - $i, j : N \quad i > j \Rightarrow i^2 > j^2$
- which states that for every pair of values in the set of natural numbers, if *i* is greater than *j,* then i^2 is greater than j^2.

Sequences

- Sequences are designated using angle brackets. For example, the preceding sequence would normally be written as
  - k Jones, Wilson, Shapiro, Estavezl
- Catenation, X, is a binary operator that forms a sequence constructed by adding its second operand to the end of its first operand. For example,
  - k 2, 3, 34, 1l X k12, 33, 34, 200 l = k 2, 3, 34, 1, 12, 33, 34, 200 l
- Other operators that can be applied to sequences are *head, tail, front,* and *last*.
  - *head* k 2, 3, 34, 1, 99, 101 l = 2
  - *tail* k 2, 3, 34, 1, 99, 101 l = 73, 34, 1,99, 1018
  - *last* k 2, 3, 34, 1, 99, 101 l = 101
  - *front* k 2, 3, 34, 1, 99, 101 l = 72, 3, 34, 1, 998

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Formal Specification

- The block handler
  - The block handler maintains a reservoir of unused blocks and will also keep track of blocks that are currently in use. When blocks are released from a deleted file they are normally added to a queue of blocks waiting to be added to the reservoir of unused blocks.
    - The state

      used, free: P *BLOCKS*

      BlockQueue: *seq* P *BLOCKS*
    - Data Invariant

      used > free = \

      used < free = *AllBlocks*

      i: dom *BlockQueue* *BlockQueue i # used*

      i, j : dom *BlockQueue* *i ≠ j => BlockQueue i > BlockQueue j* = \
    - Precondition

      #BlockQueue > 0
    - Postcondition

      used' = used \ head BlockQueue

      free' = *free < head BlockQueue*

      BlockQueue' = tail BlockQueue

Formal Specification Languages

- A formal specification language is usually composed of three primary components:
  - a syntax that defines the specific notation with which the specification is represented
  - semantics to help define a "universe of objects" [WIN90] that will be used to describe the system
  - a set of relations that define the rules that indicate which objects properly satisfy the specification
- The syntactic domain of a formal specification language is often based on a syntax that is derived from standard set theory notation and predicate calculus.
- The *semantic domain* of a specification language indicates how the language represents system requirements.

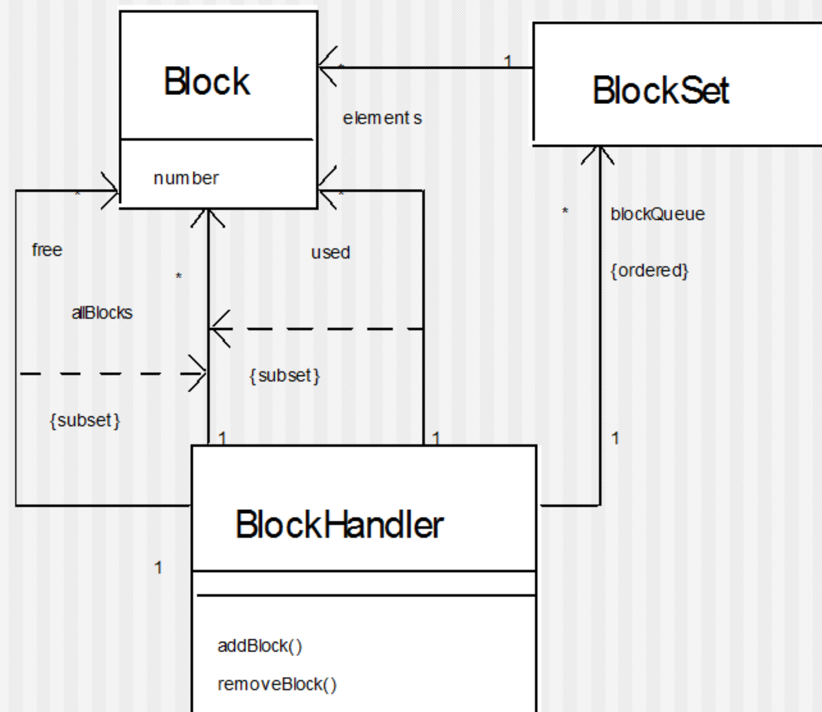
Object Constraint Language (OCL)

- a formal notation developed so that users of UML can add more precision to their specifications
- All of the power of logic and discrete mathematics is available in the language
- However the designers of OCL decided that only ASCII characters (rather than conventional mathematical notation) should be used in OCL statements.

OCL Overview

- Like an object-oriented programming language, an OCL expression involves operators operating on objects.
- However, the result of a complete expression must always be a Boolean, i.e. true or false.
- The objects can be instances of the OCL **Collection** class, of which **Set** and **Sequence** are two subclasses.
- See Table 28.1 for summary of OCL notation

BlockHandler using UML



BlockHandler in OCL

- No block will be marked as both unused and used.
  - ***context*** *BlockHandler inv:*
  - *(self.used->intersection(self.free)) ->isEmpty()*
- All the sets of blocks held in the queue will be subsets of the collection of currently used blocks.
  - ***context*** *BlockHandler inv:*
  - *blockQueue->forAll(aBlockSet | used->includesAll(aBlockSet))*
- No elements of the queue will contain the same block numbers.
  - ***context*** *BlockHandler* ***inv****:*
  - *blockQueue->forAll(blockSet1, blockSet2 |*
  - *blockSet1 <> blockSet2 implies*
  - *blockSet1.elements.number->excludesAll(blockSet2.elements.number))*
  - The expression before *implies* is needed to ensure we ignore pairs where both elements are the same Block.
- The collection of used blocks and blocks that are unused will be the total collection of blocks that make up files.
  - ***context*** *BlockHandler* ***inv****:*
  - *allBlocks = used->union(free)*
- The collection of unused blocks will have no duplicate block numbers.
  - ***context*** *BlockHandler* ***inv****:*
  - *free->isUnique(aBlock | aBlock.number)*
- The collection of used blocks will have no duplicate block numbers.
  - ***context*** *BlockHandler* ***inv****:*
  - *used->isUnique(aBlock | aBlock.number)*

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

The Z Language

- organized into *schemas*
  - defines variables
  - establishes relationships between variables
  - the analog for a "module" in conventional languages
- notation described in Table 21.2

 Slides copyright 2009 by Roger Pressman.

BlockHandler in Z

The following example of a schema describes the state of the block handler and the data invariant:

———BlockHandler——————————————————

used, free : P *BLOCKS*

BlockQueue : seq P *BLOCKS*

——————————————————————————

used > free = \

used < free = *AllBlocks*

i: dom *BlockQueue* *BlockQueue i # used*

i, j : dom *BlockQueue* *i ≠ j => BlockQueue i > BlockQueue j* = \

——————————————————————————

See Section 21.7.2 for further expansion of the specification

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Chapter 22

- **Software Configuration Management**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

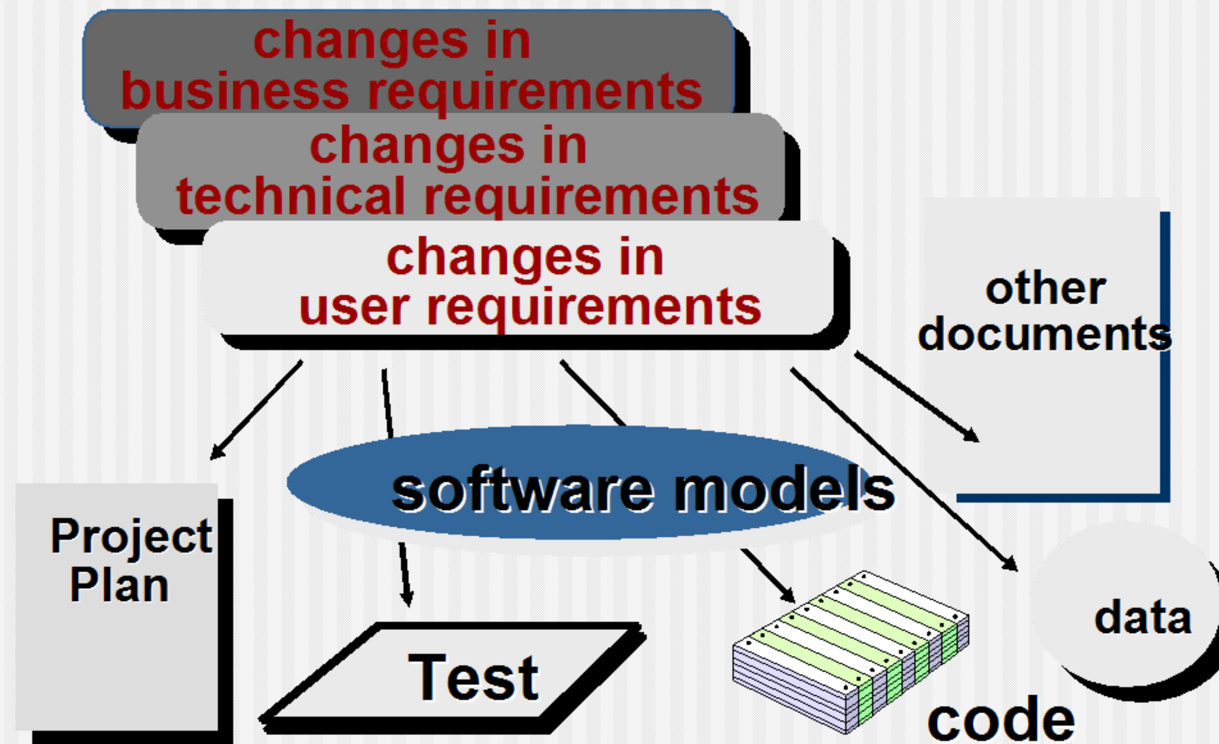
All copyright information MUST appear if these slides are posted on a website for student use.

The "First Law"

No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle.

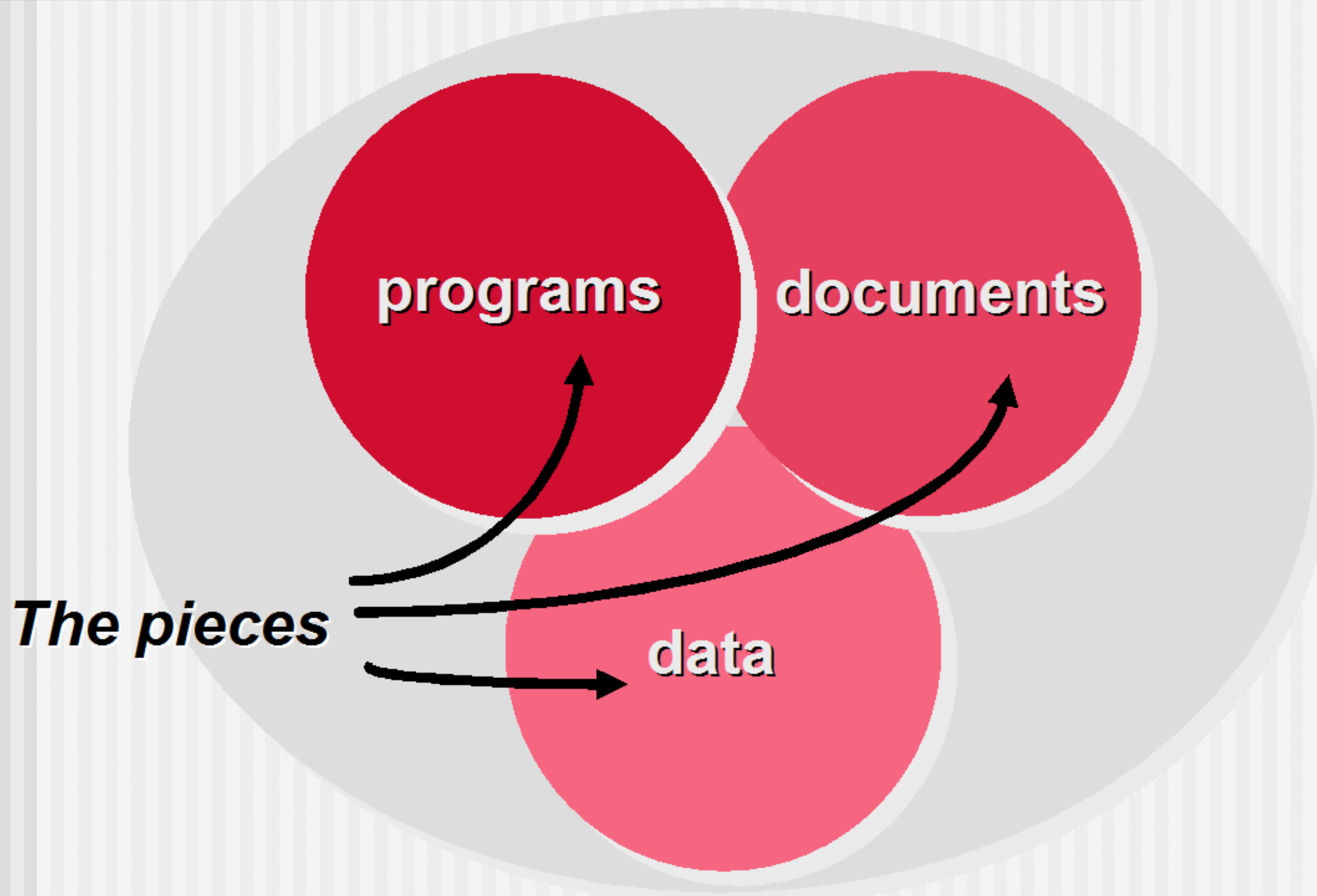
Bersoff, et al, 1980

What Are These Changes?



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

The Software Configuration



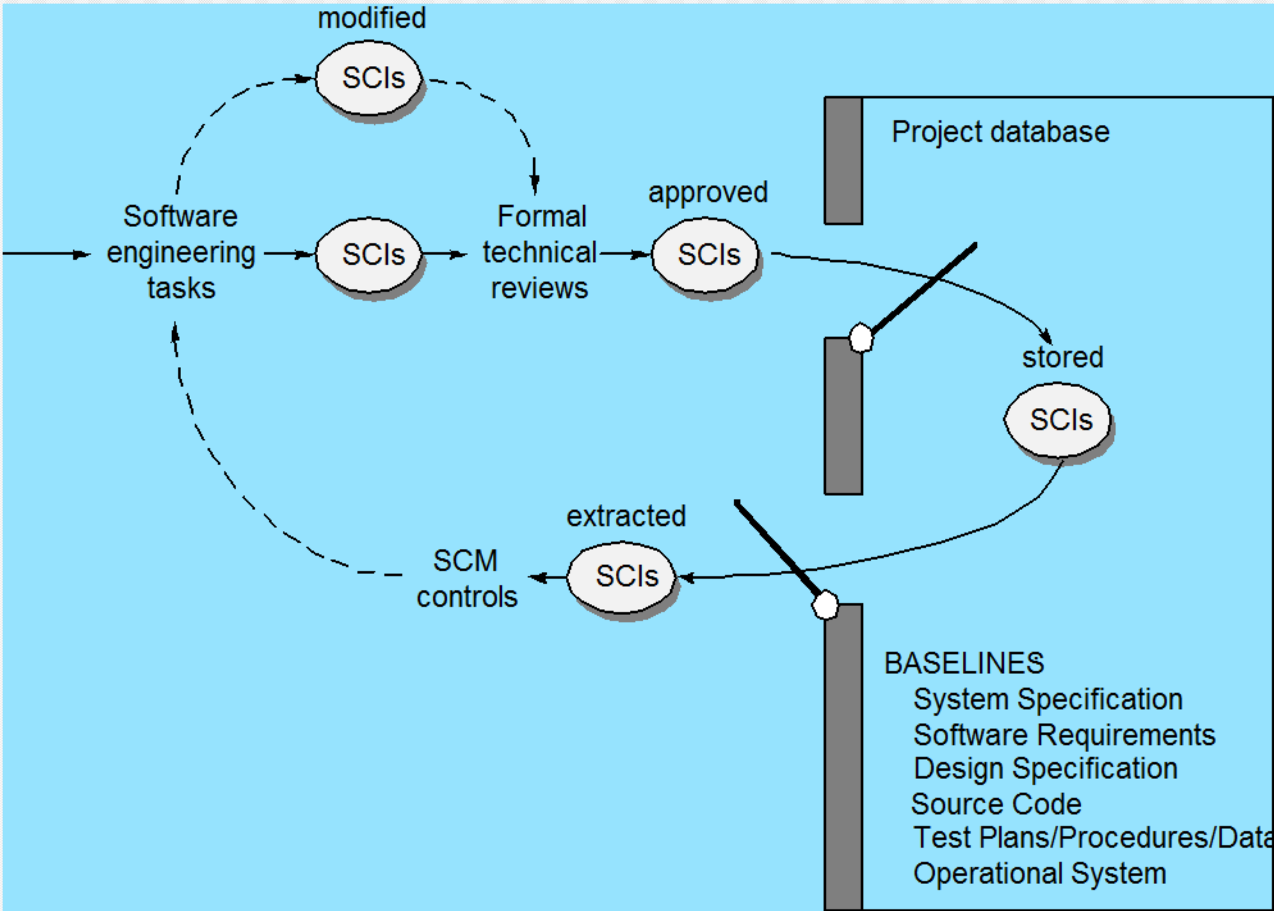
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Baselines

- The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:
  - A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.
- a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review

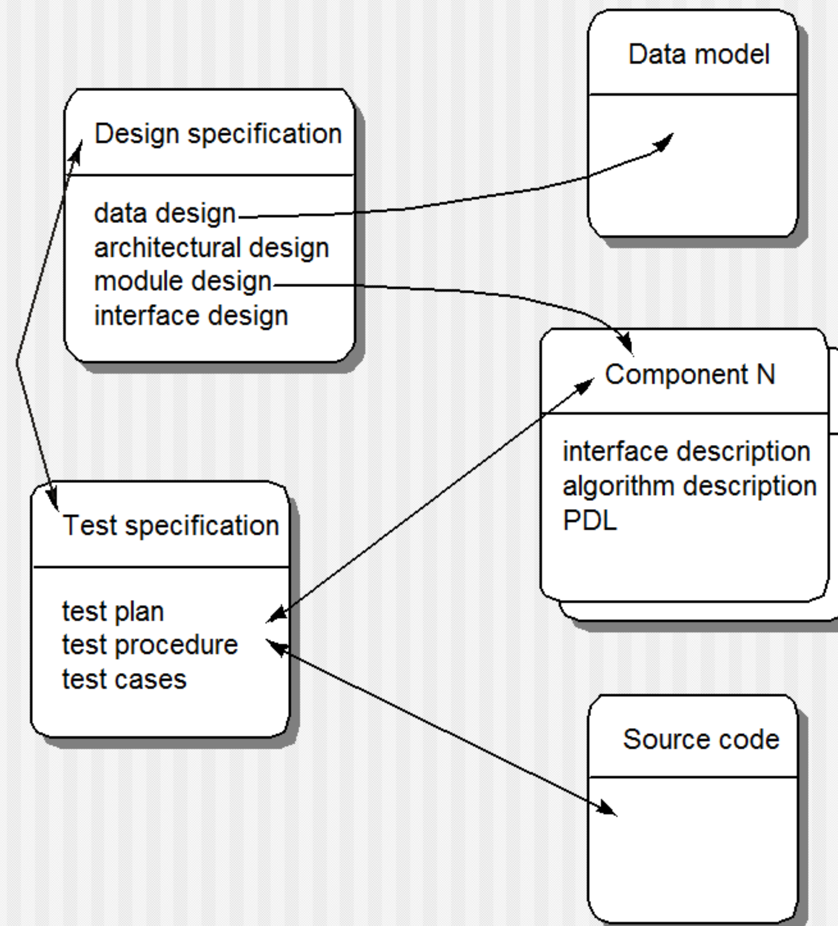
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Baselines



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Software Configuration Objects

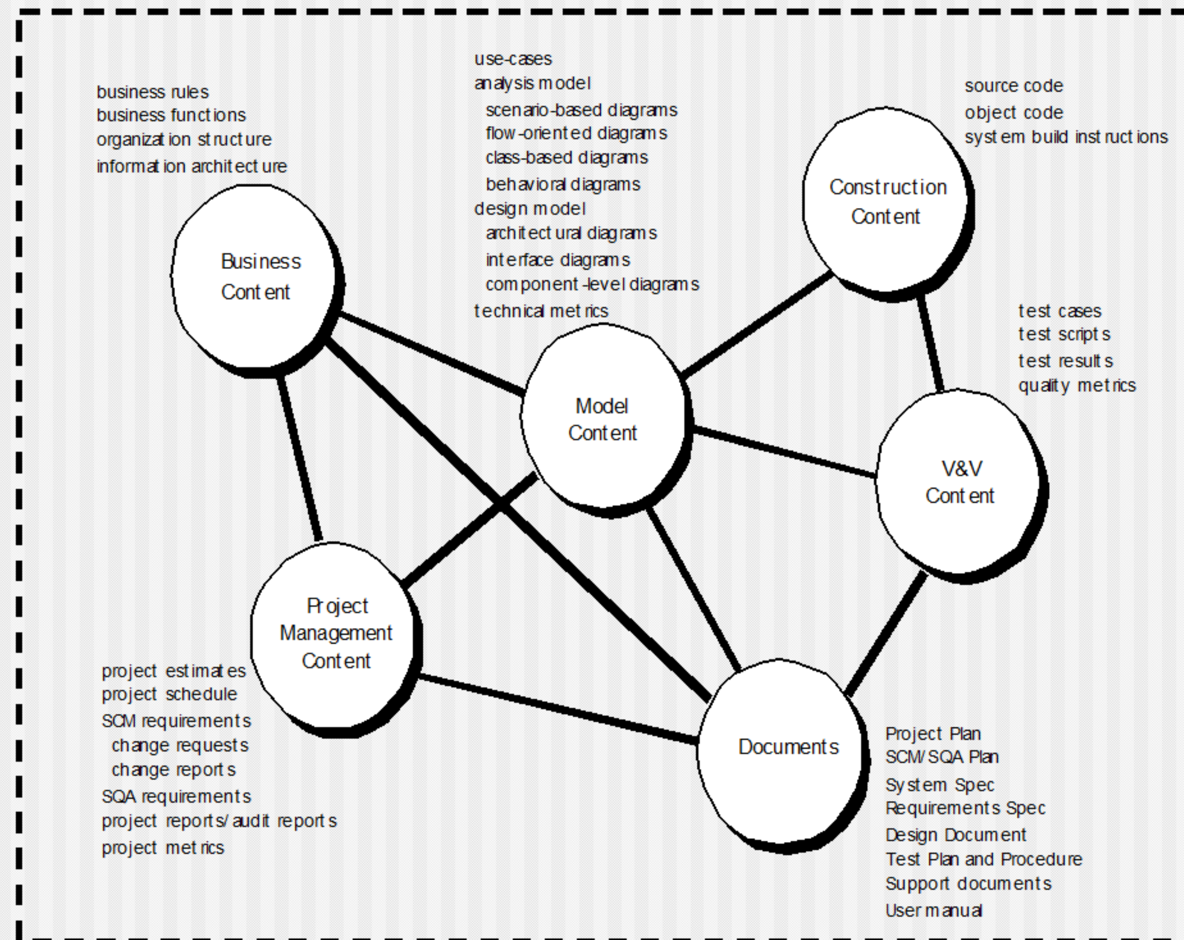


These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

SCM Repository

- The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner
- The repository performs or precipitates the following functions [For89]:
  - Data integrity
  - Information sharing
  - Tool integration
  - Data integration
  - Methodology enforcement
  - Document standardization

Repository Content



Repository Features

- **Versioning.**
  - saves all of these versions to enable effective management of product releases and to permit developers to go back to previous versions
- **Dependency tracking and change management.**
  - The repository manages a wide variety of relationships among the data elements stored in it.
- **Requirements tracing.**
  - Provides the ability to track all the design and construction components and deliverables that result from a specific requirement specification
- **Configuration management.**
  - Keeps track of a series of configurations representing specific project milestones or production releases. Version management provides the needed versions, and link management keeps track of interdependencies.
- **Audit trails.**
  - establishes additional information about when, why, and by whom changes are made.

SCM Elements

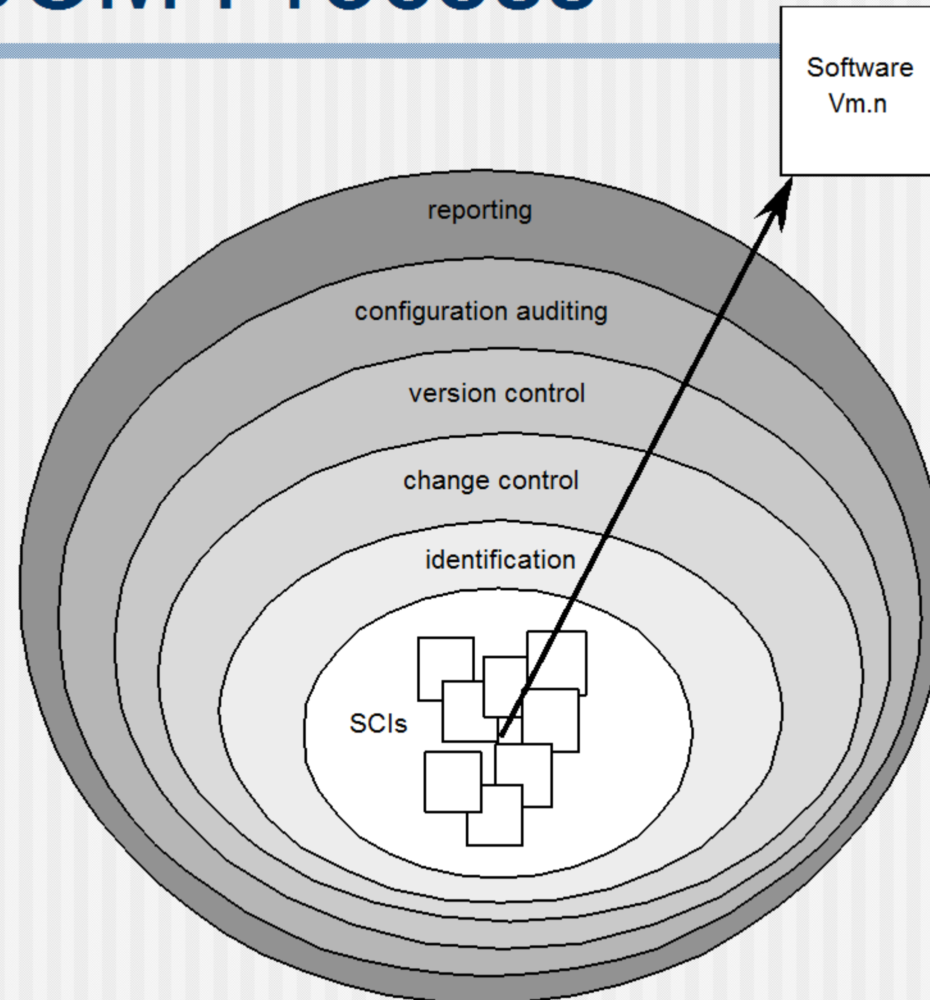
- *Component elements*—a set of tools coupled within a file management system (e.g., a database) that enables access to and management of each software configuration item.
- *Process elements*—a collection of procedures and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering and use of computer software.
- *Construction elements*—a set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) have been assembled.
- *Human elements*—to implement effective SCM, the software team uses a set of tools and process features (encompassing other CM elements)

The SCM Process

Addresses the following questions …

- How does a software team identify the discrete elements of a software configuration?
- How does an organization manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How does an organization control changes before and after software is released to a customer?
- Who has responsibility for approving and ranking changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to appraise others of changes that are made?

The SCM Process

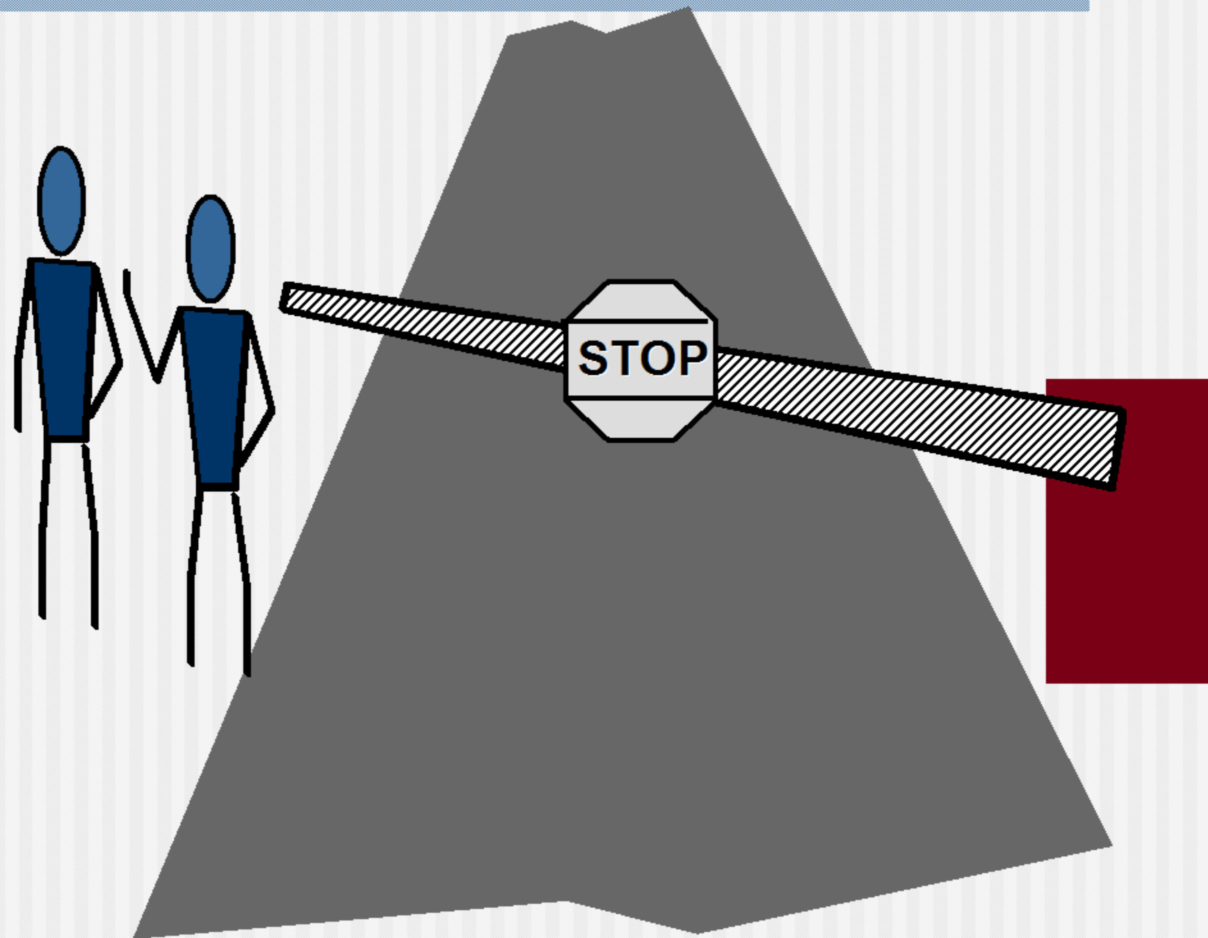


These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Version Control

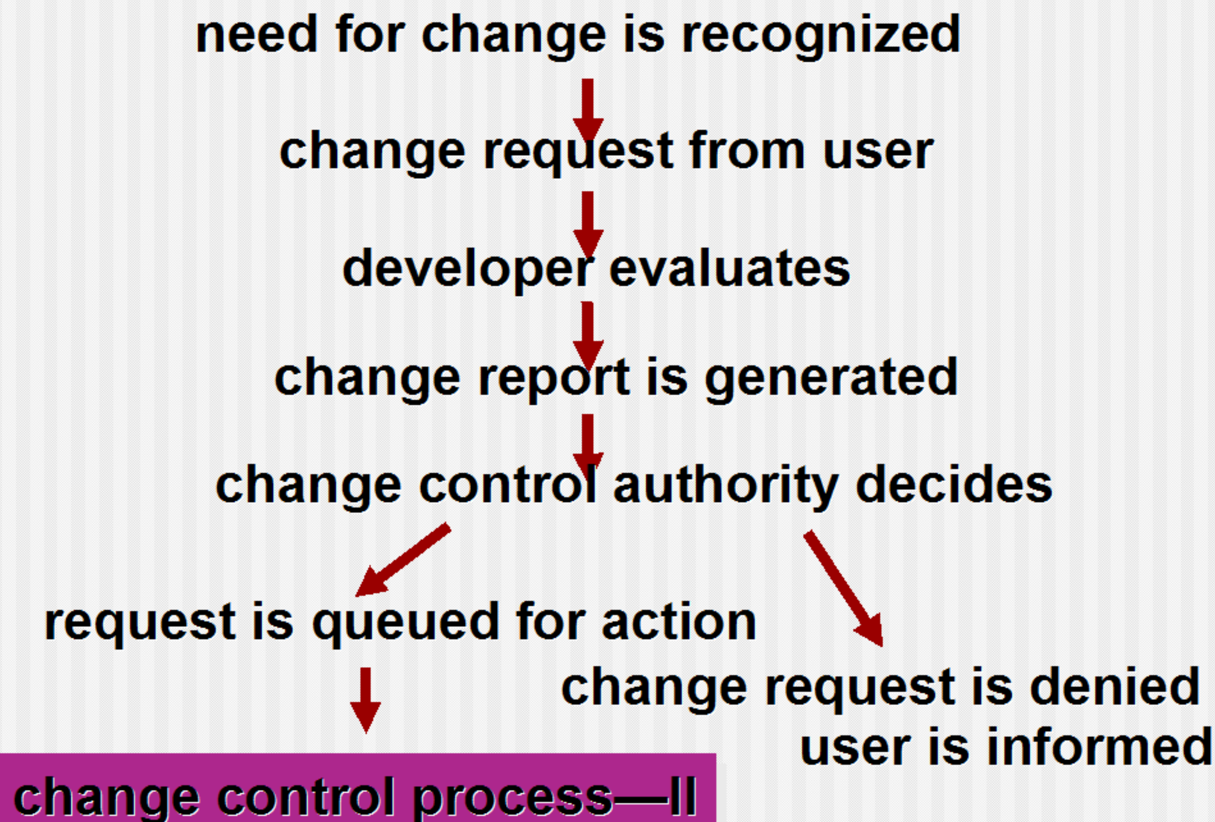
- Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process
- A version control system implements or is directly integrated with four major capabilities:
  - a *project database (repository)* that stores all relevant configuration objects
  - a *version management* capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions);
  - a *make facility* that enables the software engineer to collect all relevant configuration objects and construct a specific version of the software.
  - an *issues tracking* (also called *bug tracking*) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

Change Control



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Change Control Process—I



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Change Control Process-II

assign people to SCIs

check-out SCIs

make the change

review/audit the change

establish a "baseline" for testing

change control process—III

Change Control Process-III

perform SQA and testing activities

↓

check-in the changed SCIs

↓

promote SCI for inclusion in next release

↓

rebuild appropriate version

↓

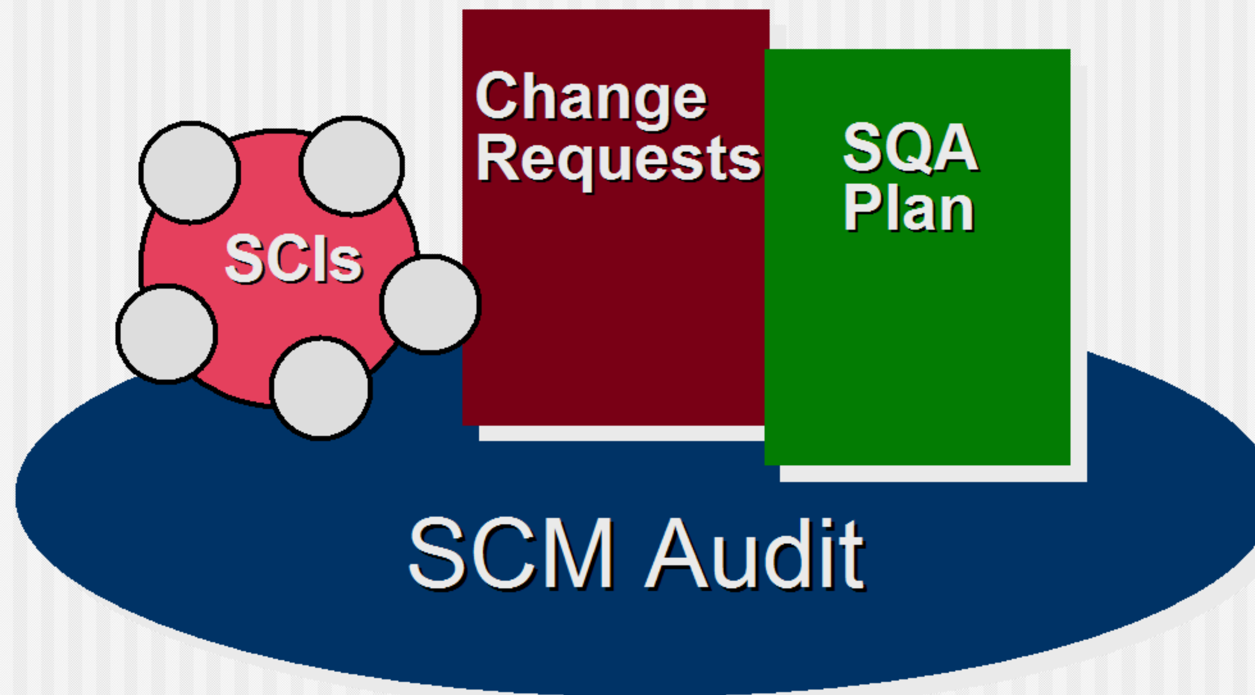
review/audit the change

↓

include all changes in release

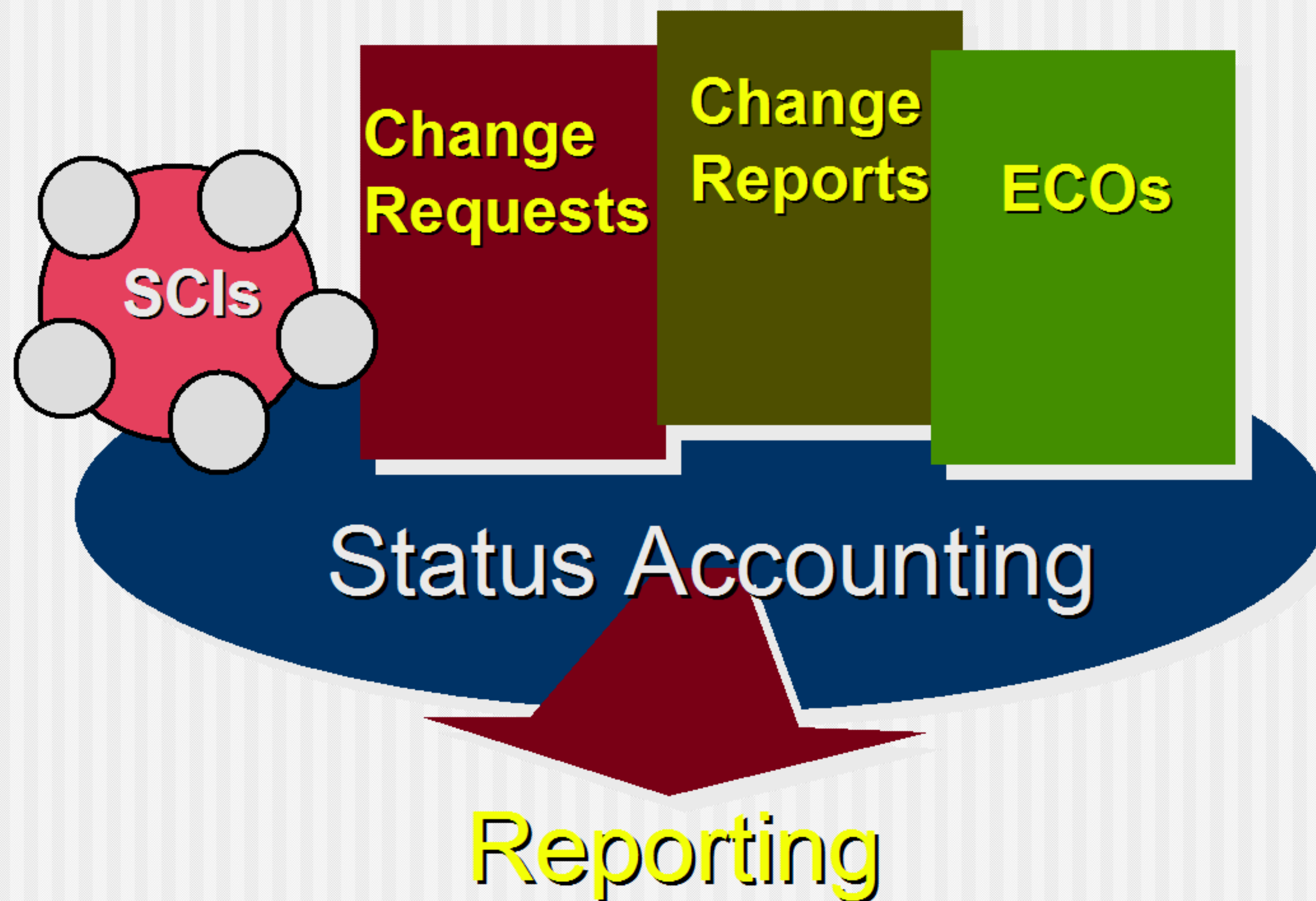
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Auditing



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Status Accounting



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

SCM for Web Engineering-I

- **Content.**
  - A typical WebApp contains a vast array of content—text, graphics, applets, scripts, audio/video files, forms, active page elements, tables, streaming data, and many others.
  - The challenge is to organize this sea of content into a rational set of configuration objects (Section 27.1.4) and then establish appropriate configuration control mechanisms for these objects.
- **People.**
  - Because a significant percentage of WebApp development continues to be conducted in an ad hoc manner, any person involved in the WebApp can (and often does) create content.

SCM for Web Engineering-II

- **Scalability.**
  - As size and complexity grow, small changes can have far-reaching and unintended affects that can be problematic. Therefore, the rigor of configuration control mechanisms should be directly proportional to application scale.
- **Politics.**
  - Who 'owns' a WebApp?
  - Who assumes responsibility for the accuracy of the information on the Web site?
  - Who assures that quality control processes have been followed before information is published to the site?
  - Who is responsible for making changes?
  - Who assumes the cost of change?

Content Management-I

- **The collection subsystem** encompasses all actions required to create and/or acquire content, and the technical functions that are necessary to
  - convert content into a form that can be represented by a mark-up language (e.g., HTML, XML
  - organize content into packets that can be displayed effectively on the client-side.
- **The management subsystem** implements a repository that encompasses the following elements:
  - *Content database*—the information structure that has been established to store all content objects
  - *Database capabilities*—functions that enable the CMS to search for specific content objects (or categories of objects), store and retrieve objects, and manage the file structure that has been established for the content
  - *Configuration management functions*—the functional elements and associated workflow that support content object identification, version control, change management, change auditing, and reporting.

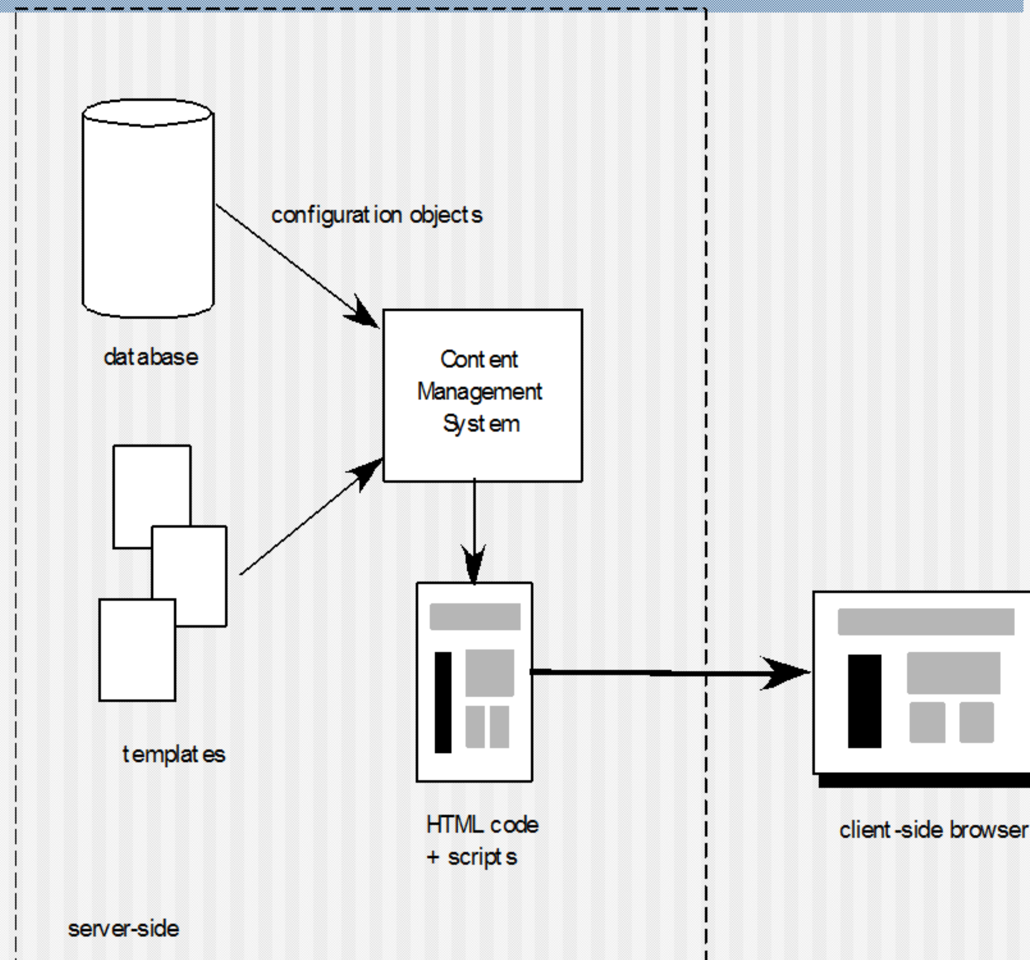
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Content Management-II

- The **publishing subsystem** extracts from the repository, converts it to a form that is amenable to publication, and formats it so that it can be transmitted to client-side browsers. The publishing subsystem accomplishes these tasks using a series of templates.
- Each *template* is a function that builds a publication using one of three different components [BOI02]:
  - *Static elements*—text, graphics, media, and scripts that require no further processing are transmitted directly to the client-side
  - *Publication services*—function calls to specific retrieval and formatting services that personalize content (using predefined rules), perform data conversion, and build appropriate navigation links.
  - *External services*—provide access to external corporate information infrastructure such as enterprise data or "back-room" applications.

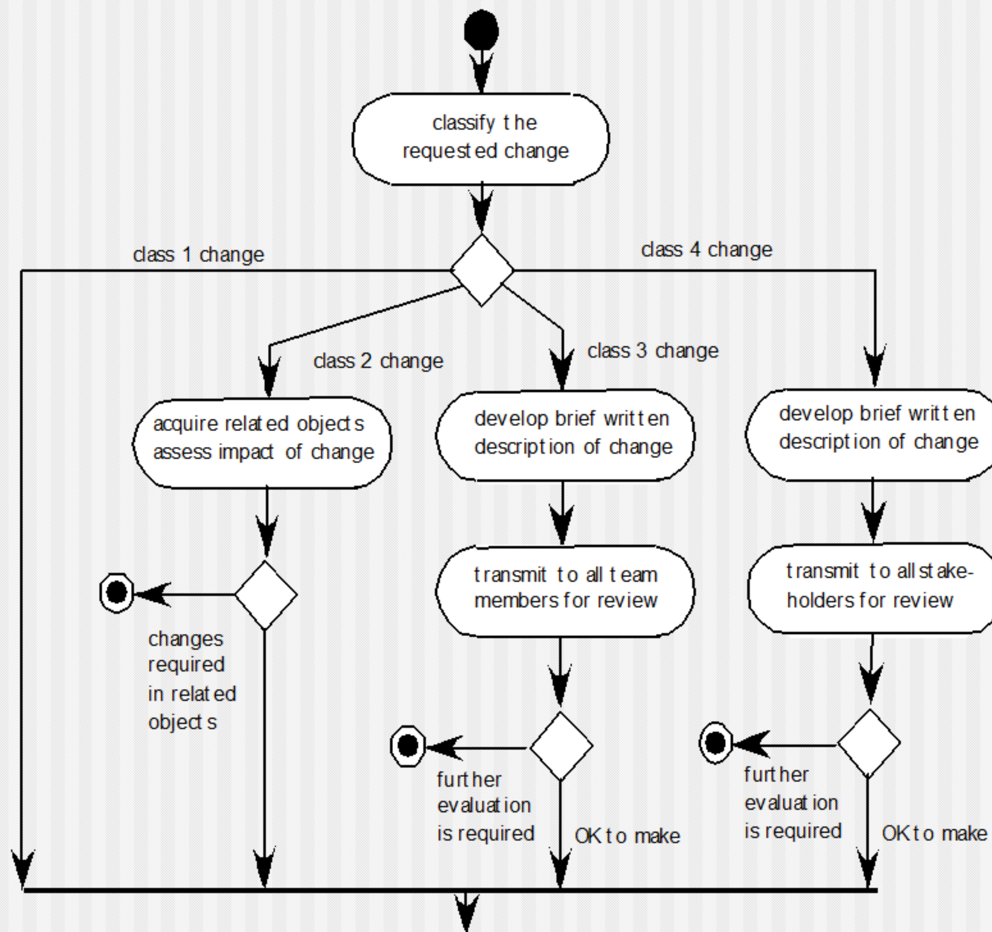
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Content Management



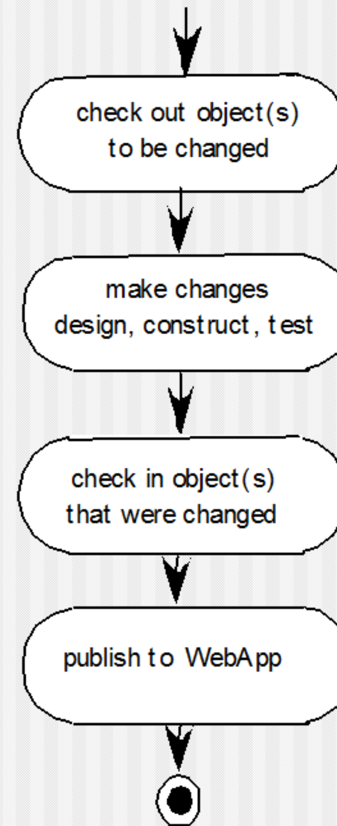
These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Change Management for WebApps-I



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Change Management for WebApps-II



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Chapter 23

- **Product Metrics**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e

by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

McCall's Triangle of Quality

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

A Comment

McCall's quality factors were proposed in the early 1970s. They are as valid today as they were in that time. It's likely that software built to conform to these factors will exhibit high quality well into the 21st century, even if there are dramatic changes in technology.

Measures, Metrics and Indicators

- A *measure* provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process
- The IEEE glossary defines a *metric* as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."
- An *indicator* is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Measurement Principles

- The objectives of measurement should be established before data collection begins;
- Each technical metric should be defined in an unambiguous manner;
- Metrics should be derived based on a theory that is valid for the domain of application (e.g., metrics for design should draw upon basic design concepts and principles and attempt to provide an indication of the presence of an attribute that is deemed desirable);
- Metrics should be tailored to best accommodate specific products and processes [Bas84]

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Measurement Process

- *Formulation.* The derivation of software measures and metrics appropriate for the representation of the software that is being considered.
- *Collection.* The mechanism used to accumulate data required to derive the formulated metrics.
- *Analysis.* The computation of metrics and the application of mathematical tools.
- *Interpretation.* The evaluation of metrics results in an effort to gain insight into the quality of the representation.
- *Feedback.* Recommendations derived from the interpretation of product metrics transmitted to the software team.

Slides copyright 2009 by Roger Pressman.

Goal-Oriented Software Measurement

- The Goal/Question/Metric Paradigm
  - (1) establish an explicit measurement *goal* that is specific to the process activity or product characteristic that is to be assessed
  - (2) define a set of *questions* that must be answered in order to achieve the goal, and
  - (3) identify well-formulated *metrics* that help to answer these questions.
- Goal definition template
  - Analyze {the name of activity or attribute to be measured}
  - for the purpose of {the overall objective of the analysis}
  - with respect to {the aspect of the activity or attribute that is considered}
  - from the viewpoint of {the people who have an interest in the measurement}
  - in the context of {the environment in which the measurement takes place}.

Metrics Attributes

- *Simple and computable.* It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time
- *Empirically and intuitively persuasive.* The metric should satisfy the engineer's intuitive notions about the product attribute under consideration
- *Consistent and objective.* The metric should always yield results that are unambiguous.
- *Consistent in its use of units and dimensions.* The mathematical computation of the metric should use measures that do not lead to bizarre combinations of unit.
- *Programming language independent.* Metrics should be based on the analysis model, the design model, or the structure of the program itself.
- *Effective mechanism for quality feedback.* That is, the metric should provide a software engineer with information that can lead to a higher quality end product

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Collection and Analysis Principles

- Whenever possible, data collection and analysis should be automated;
- Valid statistical techniques should be applied to establish relationship between internal product attributes and external quality characteristics
- Interpretative guidelines and recommendations should be established for each metric

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Metrics for the Requirements Model

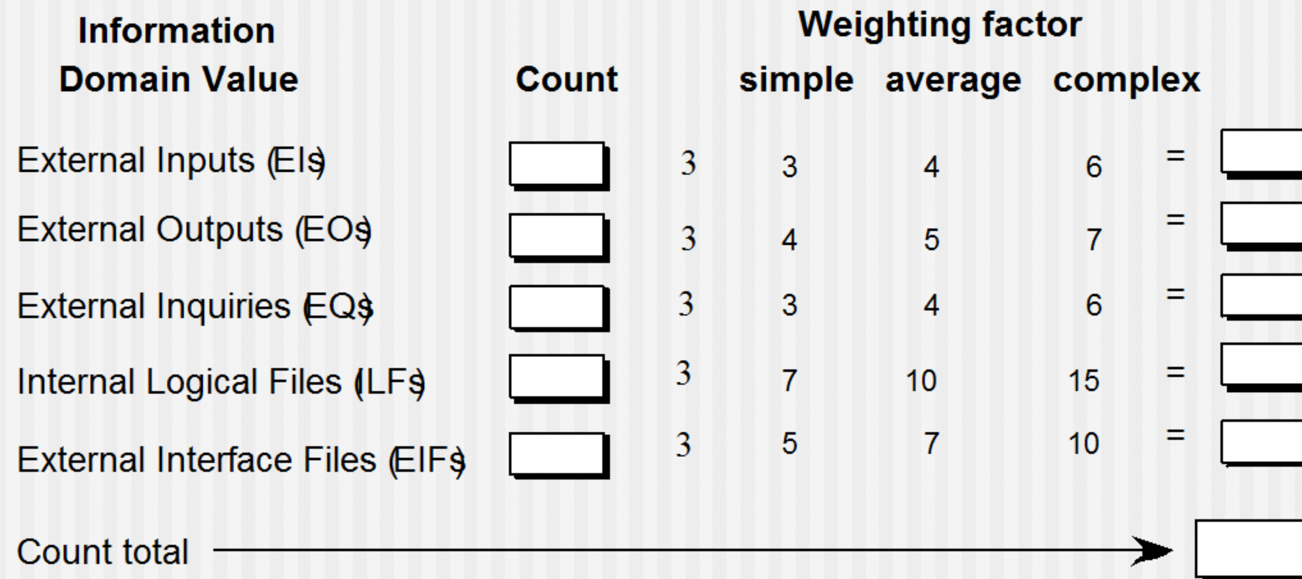
- Function-based metrics: use the function point as a normalizing factor or as a measure of the “size” of the specification
- Specification metrics: used as an indication of quality by measuring number of requirements by type

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Function-Based Metrics

- The *function point metric* (FP), first proposed by Albrecht [ALB79], can be used effectively as a means for measuring the functionality delivered by a system.
- Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity
- Information domain values are defined in the following manner:
  - number of external inputs (EIs)
  - number of external outputs (EOs)
  - number of external inquiries (EQs)
  - number of internal logical files (ILFs)
  - Number of external interface files (EIFs)

Function Points



These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Architectural Design Metrics

- Architectural design metrics
  - Structural complexity = g(fan-out)
  - Data complexity = f(input & output variables, fan-out)
  - System complexity = h(structural & data complexity)
- HK metric: architectural complexity as a function of fan-in and fan-out
- Morphology metrics: a function of the number of modules and the number of interfaces between modules

Metrics for OO Design-I

- Whitmire [Whi97] describes nine distinct and measurable characteristics of an OO design:
  - Size
    - Size is defined in terms of four views: population, volume, length, and functionality
  - Complexity
    - How classes of an OO design are interrelated to one another
  - Coupling
    - The physical connections between elements of the OO design
  - Sufficiency
    - “the degree to which an abstraction possesses the features required of it, or the degree to which a design component possesses features in its abstraction, from the point of view of the current application.”
  - Completeness
    - An indirect implication about the degree to which the abstraction or design component can be reused

Metrics for OO Design-II

- Cohesion
  - The degree to which all operations working together to achieve a single, well-defined purpose
- Primitiveness
  - Applied to both operations and classes, the degree to which an operation is atomic
- Similarity
  - The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose
- Volatility
  - Measures the likelihood that a change will occur

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Distinguishing Characteristics

Berard [Ber95] argues that the following characteristics require that special OO metrics be developed:

- Localization—the way in which information is concentrated in a program
- Encapsulation—the packaging of data and processing
- Information hiding—the way in which information about operational details is hidden by a secure interface
- Inheritance—the manner in which the responsibilities of one class are propagated to another
- Abstraction—the mechanism that allows a design to focus on essential details

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Class-Oriented Metrics

Proposed by Chidamber and Kemerer [Chi94]:

- weighted methods per class
- depth of the inheritance tree
- number of children
- coupling between object classes
- response for a class
- lack of cohesion in methods

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Class-Oriented Metrics

Proposed by Lorenz and Kidd [Lor94]:

- class size
- number of operations overridden by a subclass
- number of operations added by a subclass
- specialization index

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Class-Oriented Metrics

The MOOD Metrics Suite [Har98b]:

- Method inheritance factor
- Coupling factor
- Polymorphism factor

Operation-Oriented Metrics

Proposed by Lorenz and Kidd [Lor94]:

- average operation size
- operation complexity
- average number of parameters per operation

Component-Level Design Metrics

- Cohesion metrics: a function of data objects and the locus of their definition
- Coupling metrics: a function of input and output parameters, global variables, and modules called
- Complexity metrics: hundreds have been proposed (e.g., cyclomatic complexity)

Interface Design Metrics

- Layout appropriateness: a function of layout entities, the geographic position and the "cost" of making transitions among entities

Design Metrics for WebApps

- Does the user interface promote usability?
- Are the aesthetics of the WebApp appropriate for the application domain and pleasing to the user?
- Is the content designed in a manner that imparts the most information with the least effort?
- Is navigation efficient and straightforward?
- Has the WebApp architecture been designed to accommodate the special goals and objectives of WebApp users, the structure of content and functionality, and the flow of navigation required to use the system effectively?
- Are components designed in a manner that reduces procedural complexity and enhances the correctness, reliability and performance?

Code Metrics

- Halstead's Software Science: a comprehensive collection of metrics all predicated on the number (count and occurrence) of operators and operands within a component or program
  - It should be noted that Halstead's "laws" have generated substantial controversy, and many believe that the underlying theory has flaws. However, experimental verification for selected programming languages has been performed (e.g. [FEL89]).

Metrics for Testing

- Testing effort can also be estimated using metrics derived from Halstead measures
- Binder [Bin94] suggests a broad array of design metrics that have a direct influence on the “testability” of an OO system.
  - Lack of cohesion in methods (LCOM).
  - Percent public and protected (PAP).
  - Public access to data members (PAD).
  - Number of root classes (NOR).
  - Fan-in (FIN).
  - Number of children (NOC) and depth of the inheritance tree (DIT).

Maintenance Metrics

- IEEE Std. 982.1-1988 [IEE94] suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:
  - M_T = the number of modules in the current release
  - F_c = the number of modules in the current release that have been changed
  - F_a = the number of modules in the current release that have been added
  - F_d = the number of modules from the preceding release that were deleted in the current release
- The software maturity index is computed in the following manner:
  - $SMI = [M_T - (F_a + F_c + F_d)]/M_T$
- As SMI approaches 1.0, the product begins to stabilize.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Chapter 24

- **Project Management Concepts**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

The Four P's

- People — the most important element of a successful project
- Product — the software to be built
- Process — the set of framework activities and software engineering tasks to get the job done
- Project — all work required to make the product a reality

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Stakeholders

- *Senior managers* who define the business issues that often have significant influence on the project.
- *Project (technical) managers* who must plan, motivate, organize, and control the practitioners who do software work.
- *Practitioners* who deliver the technical skills that are necessary to engineer a product or application.
- *Customers* who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
- *End-users* who interact with the software once it is released for production use.

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Software Teams

How to lead?

How to organize?

How to collaborate?

How to motivate?

How to create good ideas?

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Team Leader

- The MOI Model
  - **Motivation.** The ability to encourage (by “push or pull”) technical people to produce to their best ability.
  - **Organization.** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
  - **Ideas or innovation.** The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Software Teams

The following factors must be considered when selecting a software project team structure ...

- the difficulty of the problem to be solved
- the size of the resultant program(s) in lines of code or function points
- the time that the team will stay together (team lifetime)
- the degree to which the problem can be modularized
- the required quality and reliability of the system to be built
- the rigidity of the delivery date
- the degree of sociability (communication) required for the project

Organizational Paradigms

- closed paradigm—structures a team along a traditional hierarchy of authority
- random paradigm—structures a team loosely and depends on individual initiative of the team members
- open paradigm—attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm
- synchronous paradigm—relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves

suggested by Constantine [Con93]

Avoid Team “Toxicity”

- A frenzied work atmosphere in which team members waste energy and lose focus on the objectives of the work to be performed.
- High frustration caused by personal, business, or technological factors that cause friction among team members.
- “Fragmented or poorly coordinated procedures” or a poorly defined or improperly chosen process model that becomes a roadblock to accomplishment.
- Unclear definition of roles resulting in a lack of accountability and resultant finger-pointing.
- “Continuous and repeated exposure to failure” that leads to a loss of confidence and a lowering of morale.

These slides are designed to accompany *Software Engineering: A Practitioner’s Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Agile Teams

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.
- Team is “self-organizing”
  - An adaptive team structure
  - Uses elements of Constantine’s random, open, and synchronous paradigms
  - Significant autonomy

Team Coordination & Communication

- *Formal, impersonal approaches* include software engineering documents and work products (including source code), technical memos, project milestones, schedules, and project control tools (Chapter 23), change requests and related documentation, error tracking reports, and repository data (see Chapter 26).
- *Formal, interpersonal procedures* focus on quality assurance activities (Chapter 25) applied to software engineering work products. These include status review meetings and design and code inspections.
- *Informal, interpersonal procedures* include group meetings for information dissemination and problem solving and "collocation of requirements and development staff."
- *Electronic communication* encompasses electronic mail, electronic bulletin boards, and by extension, video-based conferencing systems.
- *Interpersonal networking* includes informal discussions with team members and those outside the project who may have experience or insight that can assist team members.

The Product Scope

- Scope
  - **Context.** How does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context?
  - **Information objectives.** What customer-visible data objects (Chapter 8) are produced as output from the software? What data objects are required for input?
  - **Function and performance.** What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?
- Software project scope must be unambiguous and understandable at the management and technical levels.

Problem Decomposition

- Sometimes called *partitioning* or *problem elaboration*
- Once scope is defined …
  - It is decomposed into constituent functions
  - It is decomposed into user-visible data objects

  or
  - It is decomposed into a set of problem classes
- Decomposition process continues until all functions or problem classes have been defined

The Process

- Once a process framework has been established
  - Consider project characteristics
  - Determine the degree of rigor required
  - Define a task set for each software engineering activity
    - Task set =
      - Software engineering tasks
      - Work products
      - Quality assurance points
      - Milestones

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

Melding the Problem and the Process

COMMON PROCESS FRAMEWORK ACTIVITIES	communication	planning	modeling	construction
Software Engineering Tasks				
Product Functions				
Text input				
Editing and formatting				
Automatic copy edit				
Page layout capability				
Automatic indexing and TOC				
File management				
Document production				

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill, 2009). Slides copyright 2009 by Roger Pressman.

The Project

- *Projects get into trouble when …*
  - Software people don't understand their customer's needs.
  - The product scope is poorly defined.
  - Changes are managed poorly.
  - The chosen technology changes.
  - Business needs change [or are ill-defined].
  - Deadlines are unrealistic.
  - Users are resistant.
  - Sponsorship is lost [or was never properly obtained].
  - The project team lacks people with appropriate skills.
  - Managers [and practitioners] avoid best practices and lessons learned.

Common-Sense Approach to Projects

- *Start on the right foot.* This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations.
- *Maintain momentum. The* project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.
- *Track progress.* For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity.
- *Make smart decisions.* In essence, the decisions of the project manager and the software team should be to "keep it simple."
- *Conduct a postmortem analysis.* Establish a consistent mechanism for extracting lessons learned for each project.

To Get to the Essence of a Project

- Why is the system being developed?
- What will be done?
- When will it be accomplished?
- Who is responsible?
- Where are they organizationally located?
- How will the job be done technically and managerially?
- How much of each resource (e.g., people, software, tools, database) will be needed?

Barry Boehm [Boe96]

Critical Practices

- Formal risk management
- Empirical cost and schedule estimation
- Metrics-based project management
- Earned value tracking
- Defect tracking against quality targets
- People aware project management

 Slides copyright 2009 by Roger Pressman.