SOFTWARE QUALITY

Software Engineering

Dr. Raj Singh

QUALITY CONCEPTS

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.
- Software adheres to the requirements.
- Produces expected results.
- Easy to use
- Error free
- Easy to maintain and extend

EFFECTIVE SOFTWARE PROCESS

An effective software process establishes the infrastructure that supports to build 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.

USEFUL PRODUCT

A useful product delivers the content, functions, and features that the end-user desires

It delivers these assets in a reliable, error free way.

Always satisfies the requirements that have been explicitly stated by stakeholders.

It is ease of use and maintain.

QUALITY ADDS VALUE

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:

- greater software product revenue
- better profitability when an application supports a business process, and/or
- improved availability of information that is crucial for the business

QUALITY DIMENSIONS

Performance:

- Does the software deliver all content, functions, and features that provide value to the end-user?

Feature:

- Does the software provide features that surprise and delight first-time end-users?

Reliability:

- Does the software deliver all features and capability without failure, reliable, and is error free?

Conformance:

- Does the software conforms to software design and standards?

Durability:

- Can the software be maintained and corrected without side effects?

MEASURING QUALITY

General quality dimensions and factors are not adequate for assessing the quality of an application in concrete terms

Project teams need to develop a set of targeted questions to assess the degree to which each application quality factor has been satisfied

Subjective measures of software quality may be viewed as little more than personal opinion

Software metrics represent indirect measures of some manifestation of quality and attempt to quantify the assessment of software quality

COST OF QUALITY

Prevention costs include

- quality planning
- formal technical reviews
- test equipment
- Training

Internal failure costs include

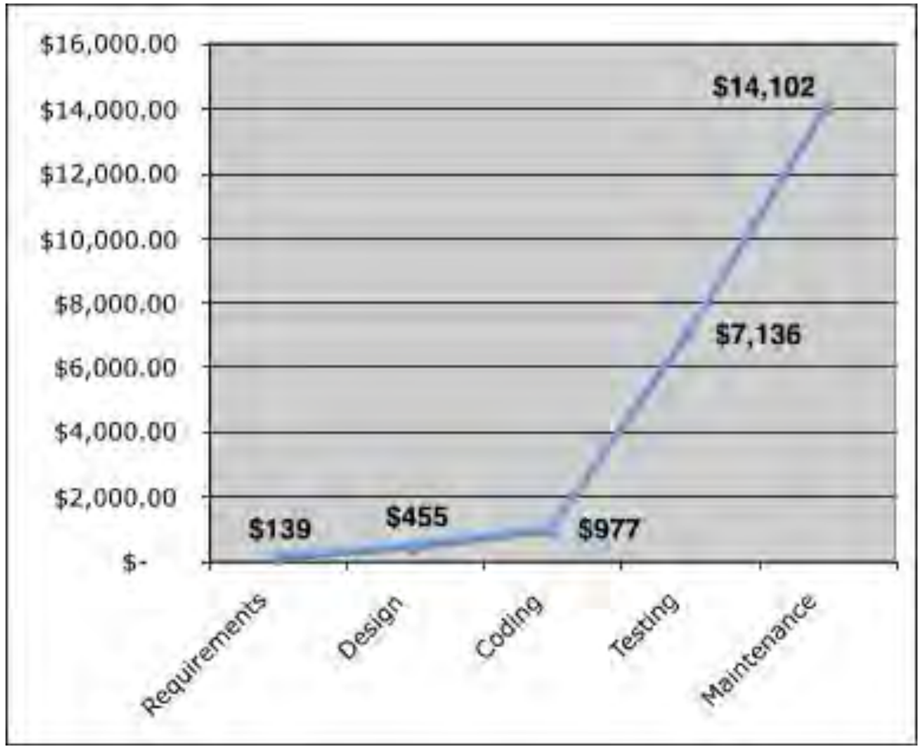
- rework
- repair
- failure mode analysis

External failure costs are

- complaint resolution
- product return and replacement
- help line support
- warranty work

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.



POOR QUALITY SOFTWARE

- Poor quality increases risks for both developers and end-users
- When systems are delivered late, fail to deliver functionality, and does not meet customer expectations litigation ensues
- Poor quality software is easier to hack and can increase the security risks for the application once deployed
- A secure system cannot be built without focusing on quality (security, reliability, dependability) during the design phase
- Low quality software is liable to contain architectural flaws as well as implementation problems (bugs)

IMPACTS OF QUALITY

End users

Developers

Management

Revenue

Profit

Company

ACHIEVING SOFTWARE QUALITY

- Good quality is the result of effective project management and engineering practices
- Understand the problem to be solved and be able of creating a quality design that meets the requirements
- Eliminating architectural flaws during design can improve quality
- Project plan must include explicit techniques for quality and change management
- Series of inspections, reviews, and tests must be used to ensure quality

REVIEW

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

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: a quality problem found before the software is released to end users

Defects: a quality problem found only after the software has been released to end-users

Errors and defects have very different economic, business, psychological, and human impact

METRICS

- The total review effort and the total number of errors discovered are defined as:
  - Ereview = Ep + Ea + Er
  - Errtot = Errminor + Errmajor
- Defect density represents the errors found per unit of work product reviewed.
  - Defect density = Errtot / WPS
- where …

METRICS

- Preparation effort, Ep — the effort (in person-hours) required to review a work product prior to the actual review meeting
- Assessment effort, Ea — the effort (in person-hours) that is expending during the actual review
- Rework effort, Er — 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, Errminor — the number of errors found that can be categorized as minor (requiring less than some pre-specified effort to correct)
- Major errors found, Errmajor — the number of errors found that can be categorized as major (requiring more than some pre-specified effort to correct)

INFORMAL REVIEWS

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.

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.

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)

CONDUCTING THE REVIEW

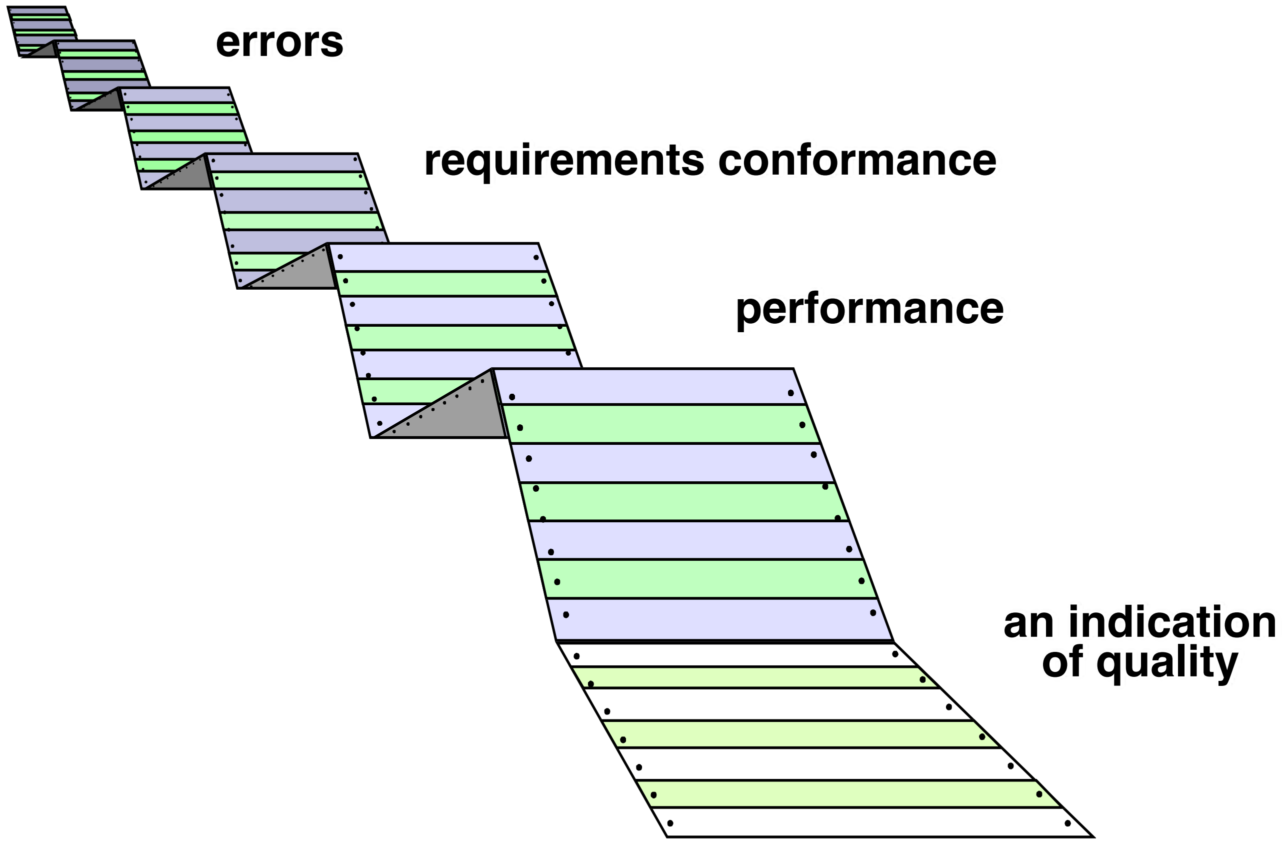
Product	Review the product, not the producer.
Agenda	Set an agenda and maintain it.
Limit	Limit debate and rebuttal, and # of participants.
Enunciate	Enunciate problem areas, but don't attempt to solve every problem noted.
Notes	Take written notes.
Checklist	Develop a checklist for each product that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

TESTING

SOFTWARE TESTING

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

WHAT TESTING SHOWS



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:

Are we building the product right?

refers to the set of tasks that ensure that software correctly implements a specific function

Validation:

Are we building the right product?

refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

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

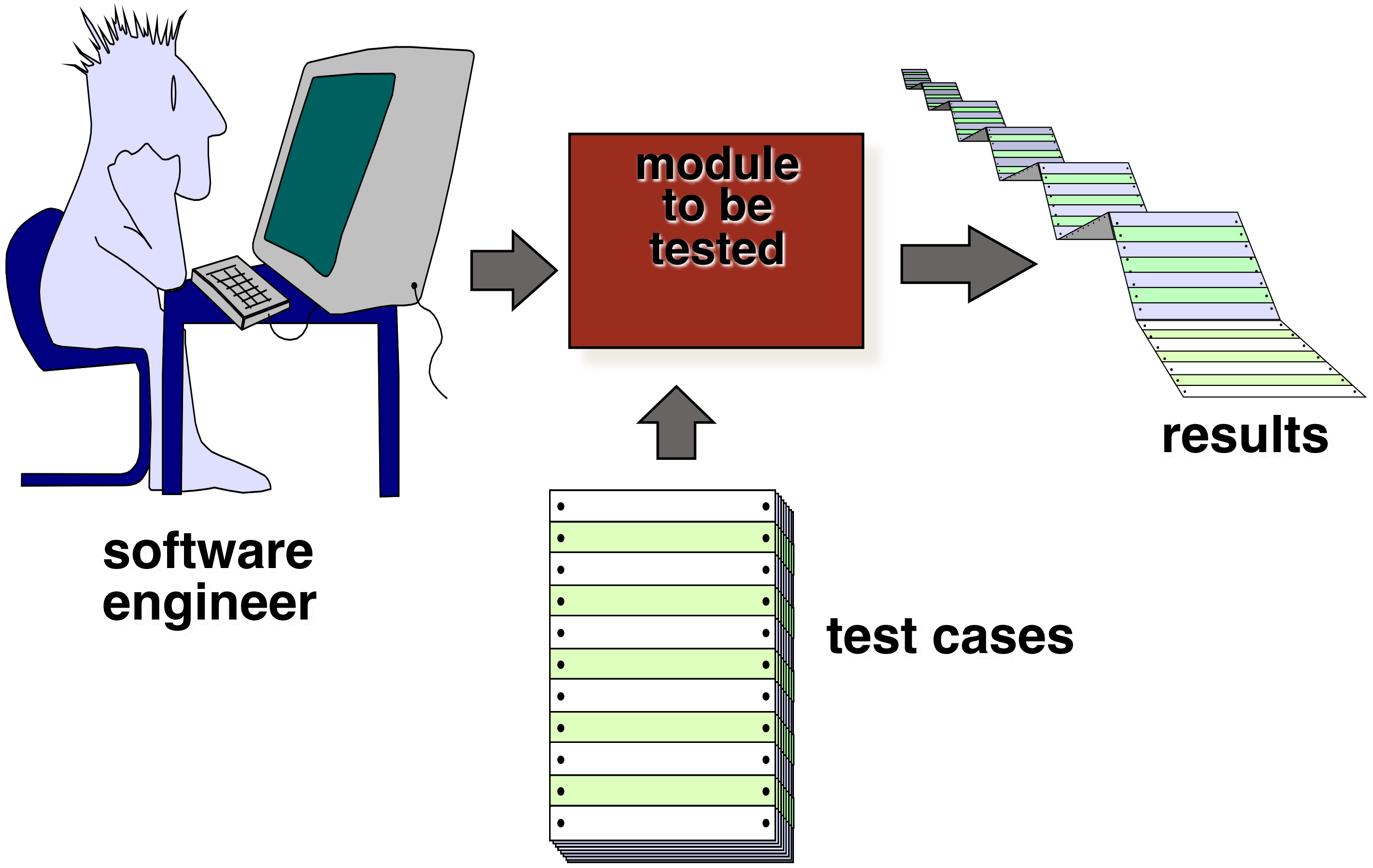
Integration of modules follows

For OO software

an OO class that encompasses attributes and operations and implies communication and collaboration

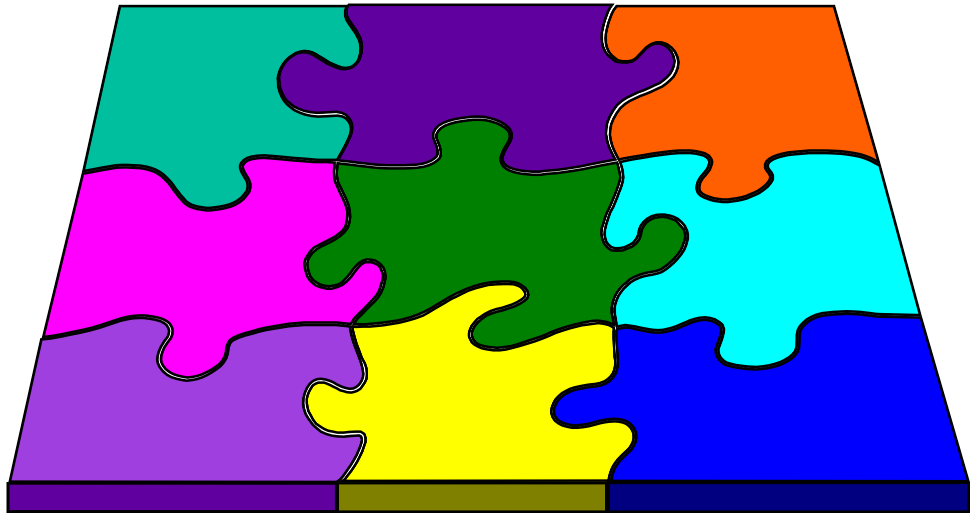
UNIT TESTING

- Test individual units as you code.



INTEGRATION TESTING STRATEGIES

- Put all modules together and test again.



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

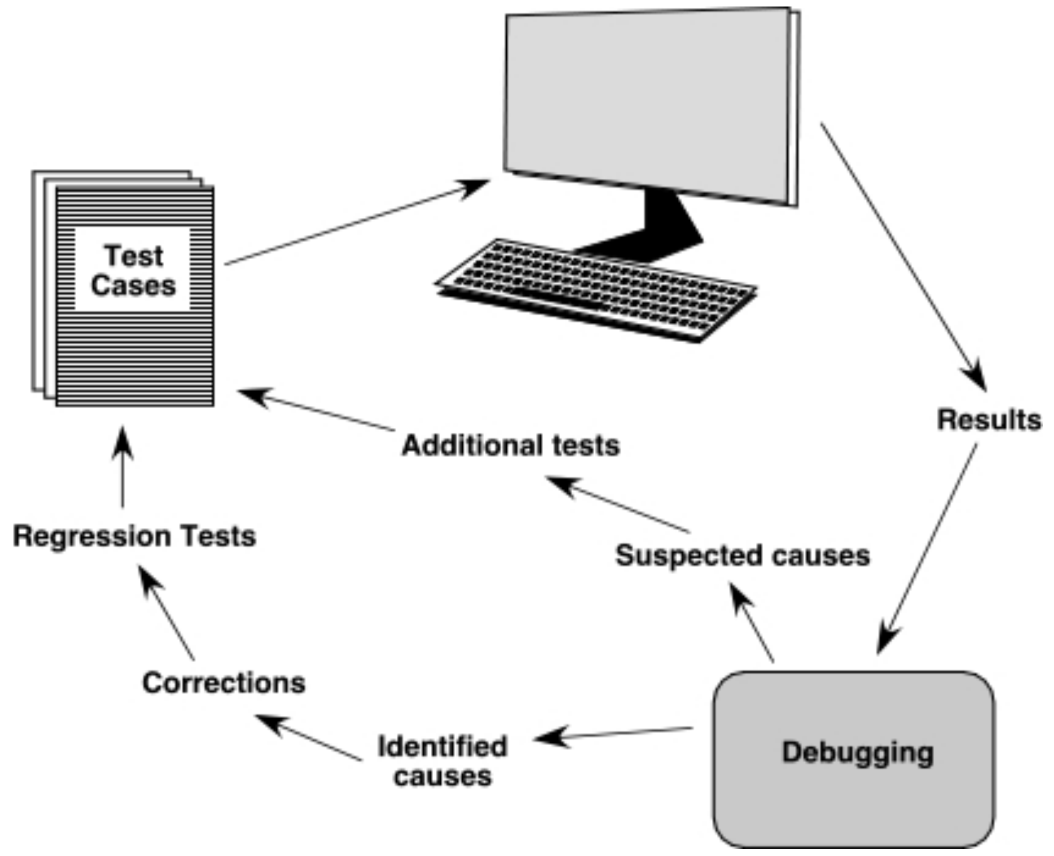
Software components that have been translated into code are integrated into a "build."	• A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
A series of tests is designed to expose errors that will keep 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.

GENERAL TESTING CRITERIA

Interface integrity	internal and external module are tested as each module is added to the software
Functional validity	test to uncover functional defects in the software
Information content	test for errors in local or global data structures
Performance	verify specified performance bounds are tested

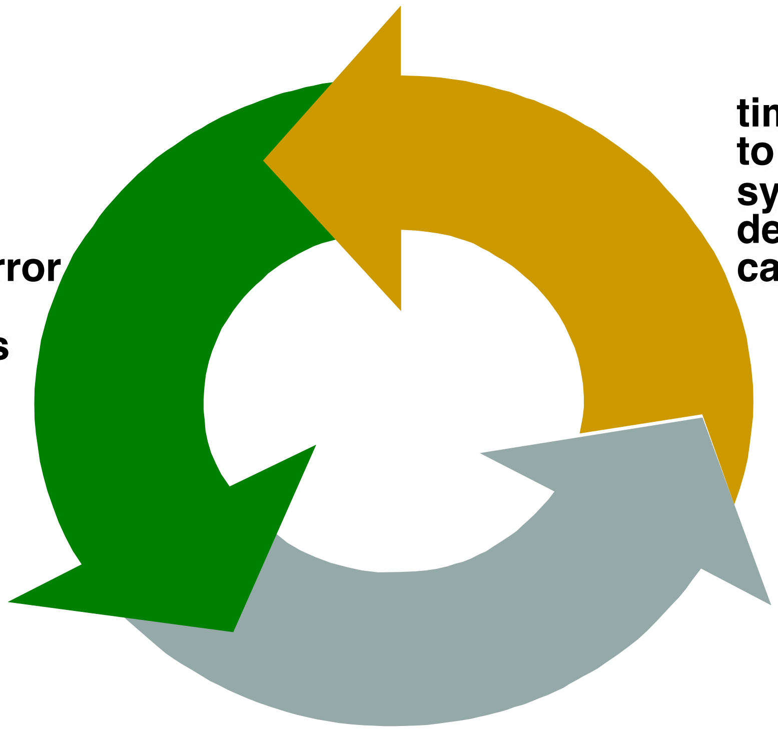
HIGH ORDER TESTING

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



THE DEBUGGING PROCESS

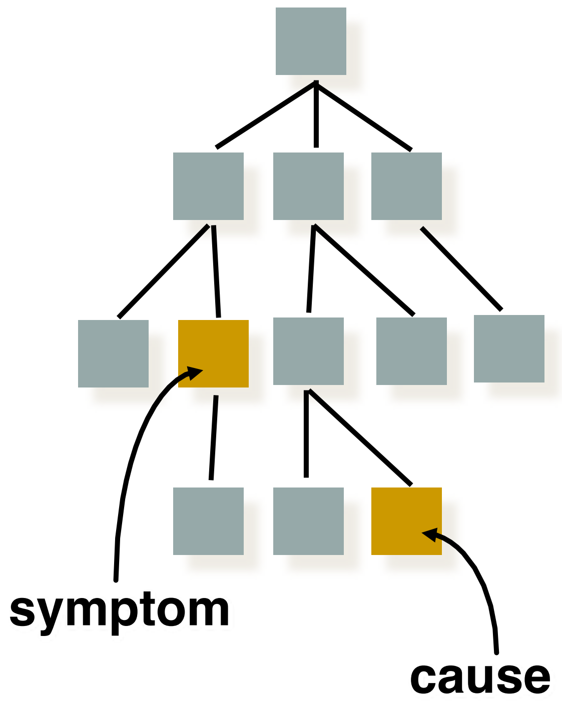
DEBUGGING EFFORT



time required to diagnose the symptom and determine the cause

time required to correct the error and conduct regression tests

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

DEBUGGING TECHNIQUES

Brute force	Review the logs. Test everything if nothing is clear.
Backtracking	Step-by-step go over each line of code in problem area. Back track the problem cause.
Induction	Start with the symptoms of the error, possibly in the result of one or more test cases, the error is often uncovered
Deduction	Processes of elimination and refinement, to arrive at a conclusion.
Other	Interactive, print, remote, post-mortem

CORRECTING THE ERROR

- Is the cause of the bug reproduced in another part of the program?
- Before the correction is made, the source code and design should be evaluated to assess the impact on other areas.
- What could we have done to prevent this bug in the first place?
- 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

REFERENCE

- Roger Pressman, Software Engineering: A Practitioner's Approach, 8th edition, McGraw Hill, ISBN 0078022126