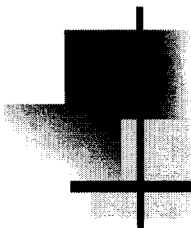


# Software Audit Observer Training

Version 1.0



---

Prepared by the  
Center for Nuclear Regulatory Waste  
Analyses  
Quality Assurance

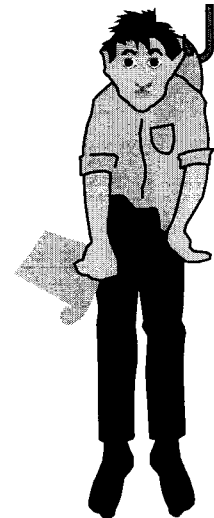
# Software Observer Auditor Training - Course Agenda

- Life Cycle Models
- Design, Development and Testing
- Peer Reviews
- Configuration Management
- Validation
- Risk Management
- Maintenance

# About Your Instructor

Randy Folck

- Consultant: process improvement
- Lead Auditor: Telecom, Aerospace, Automotive, Commercial Nuclear
- Nine years software QA
- University instructor
- Twenty five years quality management system experience



# Myths or Facts?

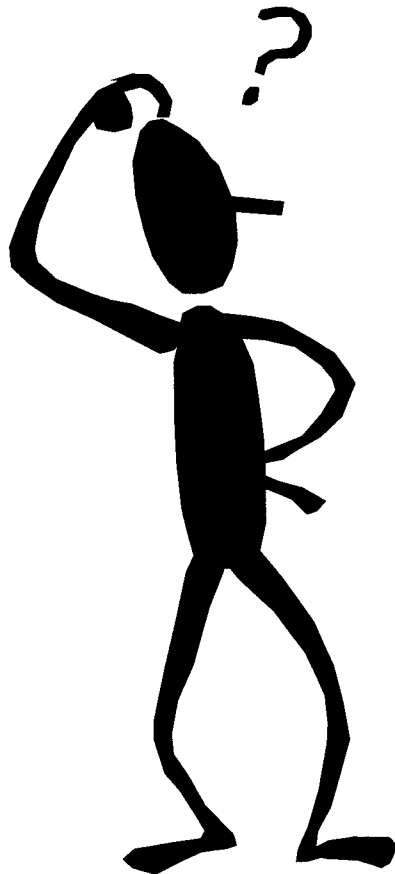
- Quality means goodness; it cannot be defined. ●
- Because it cannot be defined, quality cannot be measured.
- The trouble with quality is that workers don't really care.
- Quality is fine, but we can't afford it. ●
- Data Processing is different, error is inevitable.

# Software Quality Defined

- The degree to which software **meets specified requirements.**
- The degree to which software **meets customer or user needs or expectations.**

[IEEE-STD-610]

# Is Satisfying Requirements Software Quality?



- Getting the requirements right
- Getting the right requirements

# What is software 'quality'?



- Reasonably bug-free software ●
- Meets requirements and expectations
- Maintainable

Quality may be defined by a set of attributes! ●



# Software Quality as Attributes

---

- Quality is the degree of excellence of something.
- We measure the excellence of software via a **set of attributes**.

[Glass, Robert L., *Building Quality Software*, Prentice Hall, Englewood Cliffs, NJ, 1992]



# Quality Attributes



- Portability
- Reliability
- Efficiency
- User Friendliness
- Testability
- Understandability
- Modifiability

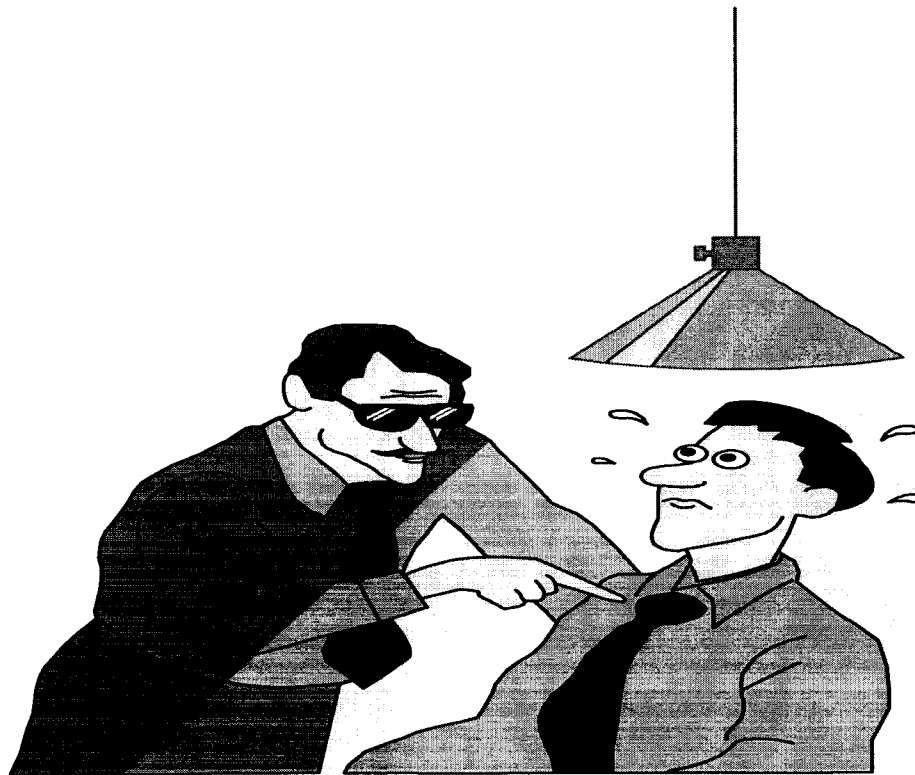
# If quality is made up of attributes then...

How do we achieve them?

- Trade-off analysis
  - Prioritized list of attributes
- Testing, testing, testing
- Focus on satisfying requirements



# Questions?

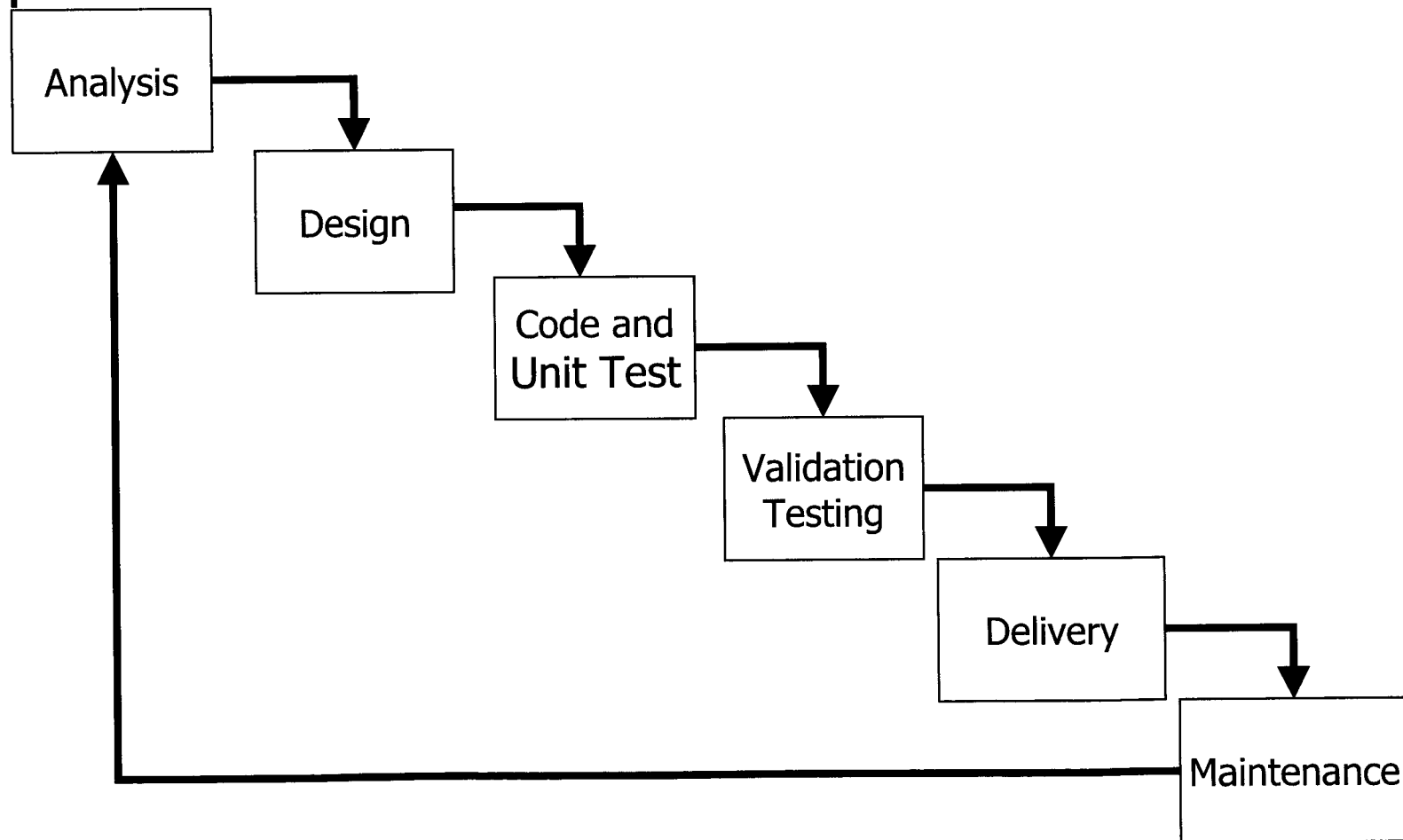


# Software Lifecycle Development Models



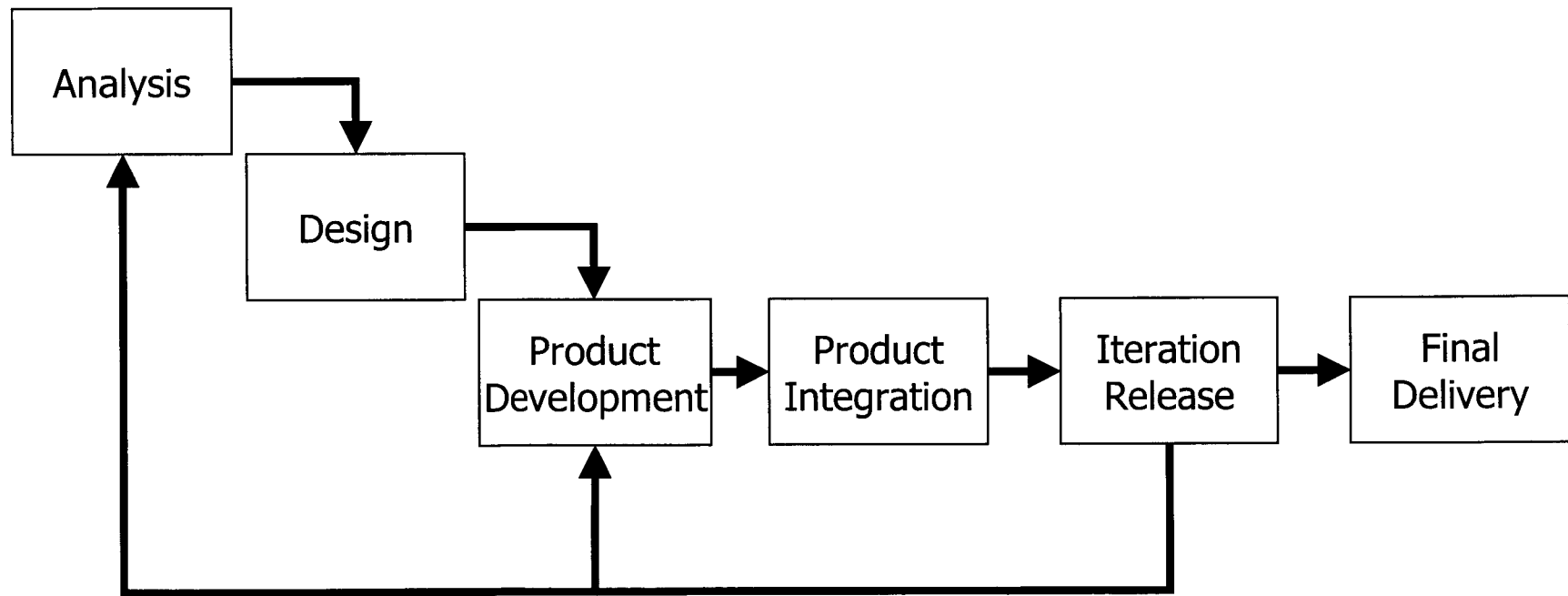
- The Waterfall Software Development Model
- The Iterative Software Development Model
- The Prototype Development Model
- Others

# Waterfall Model

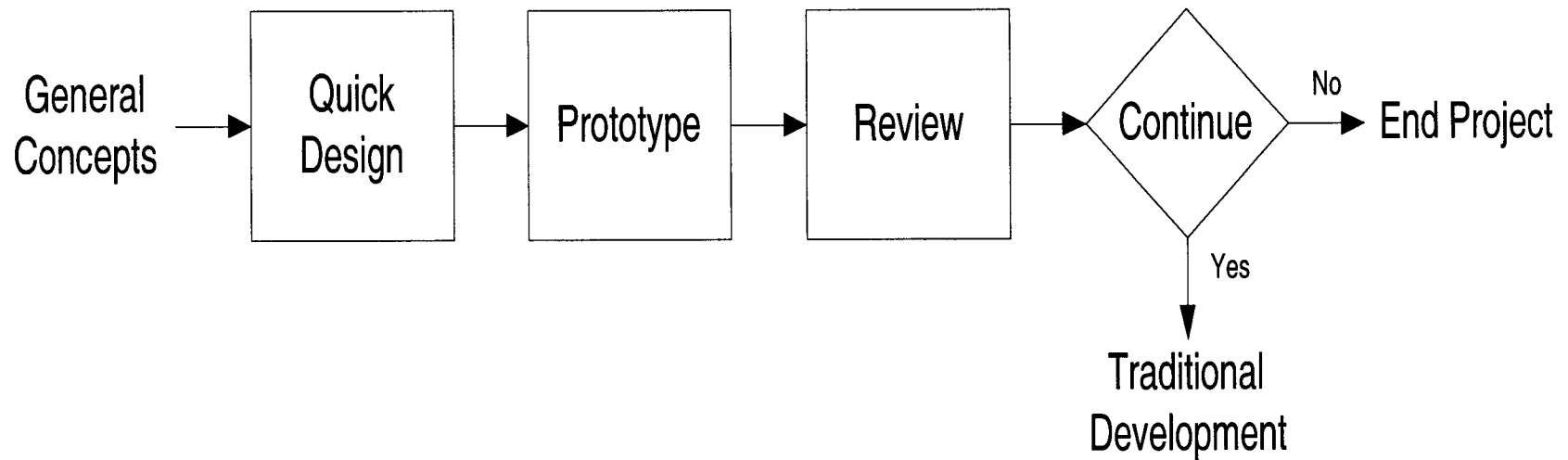


Version 1.0

# Iterative Model



# Prototype Model

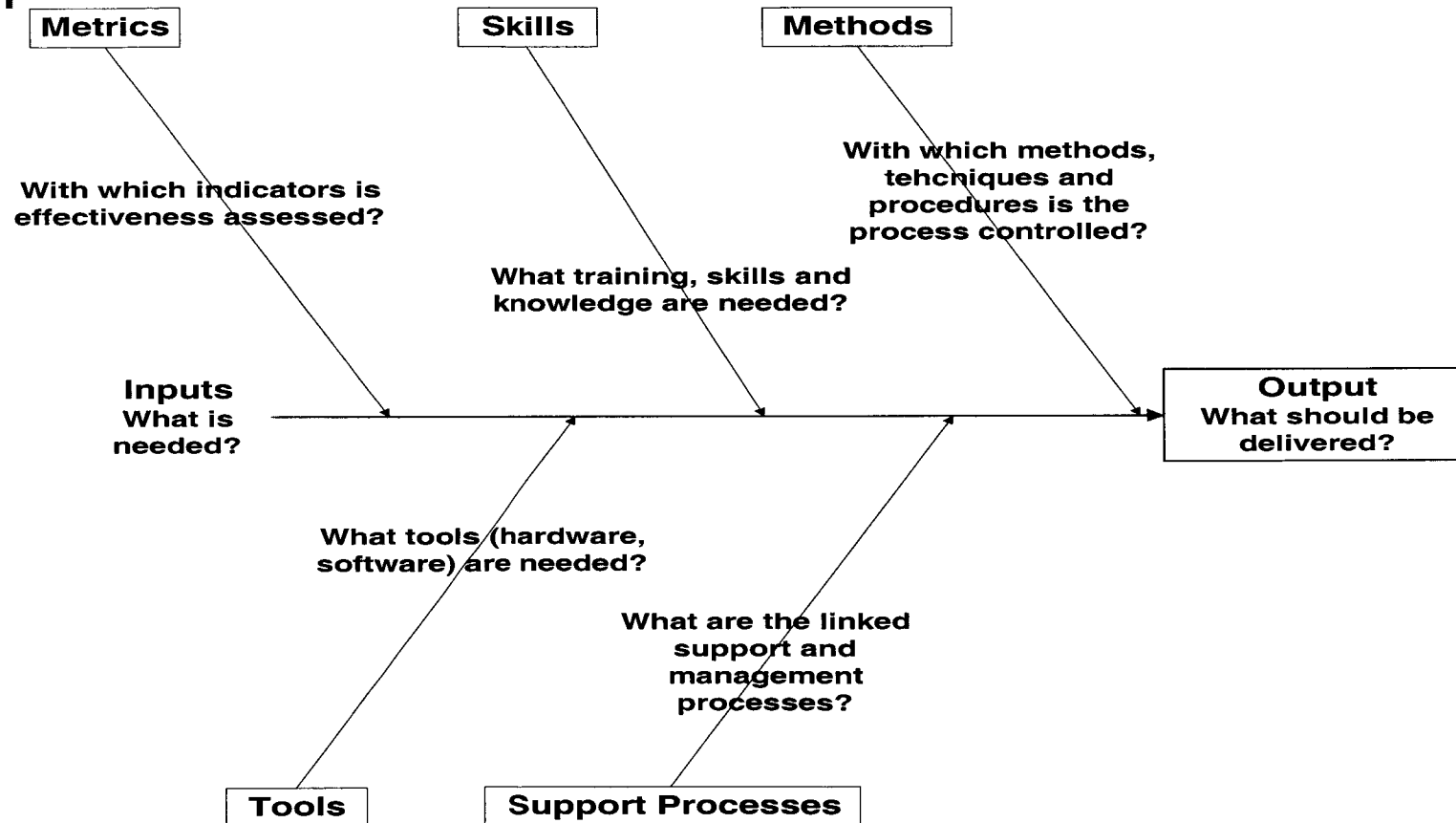


# Questions?

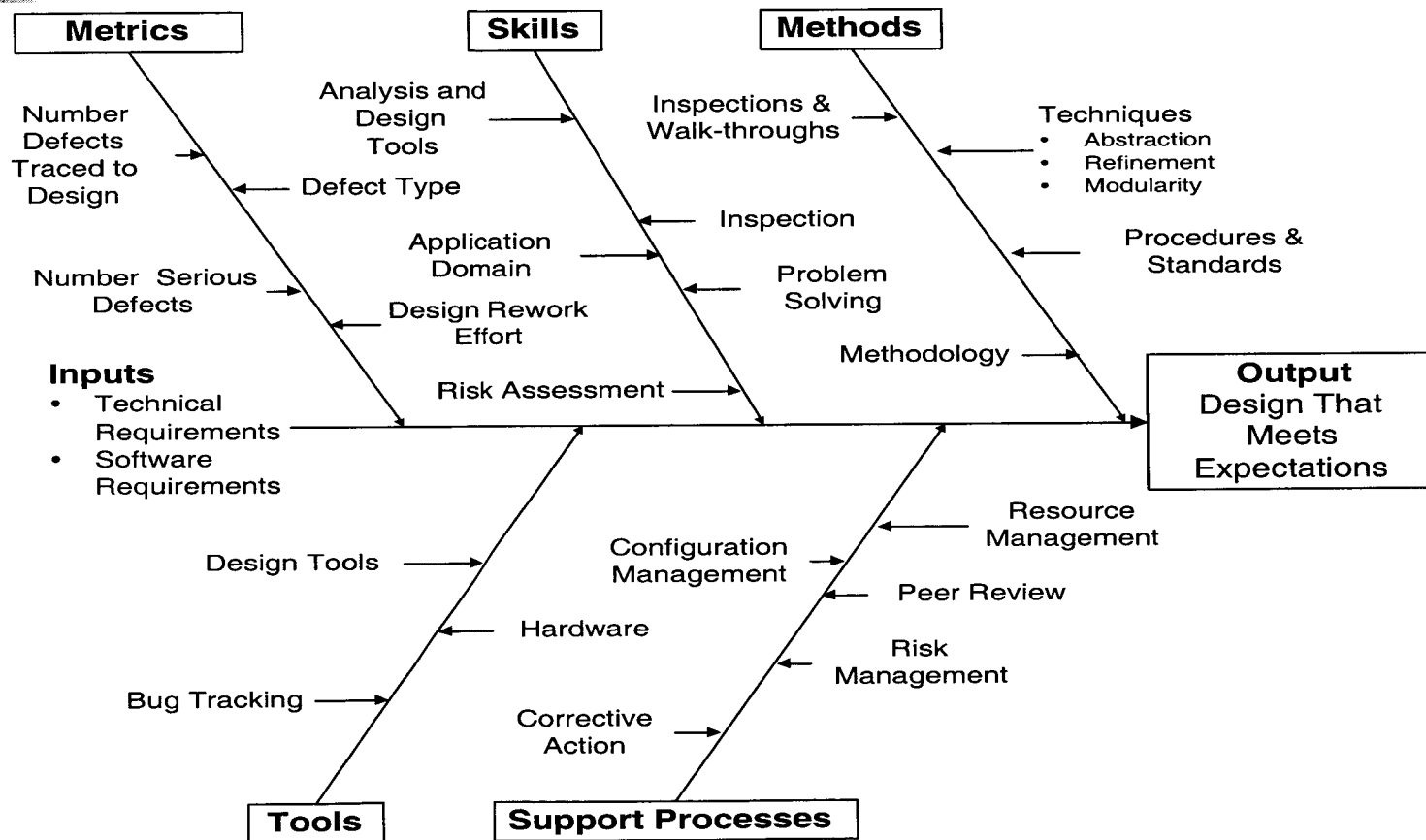




# 4 Questions About a Process



# Software Analysis & Design





# Analysis and Design Methods

---

- Structured Analysis and Design
- Functional Decomposition
- Object-oriented Analysis and Design
- Others



# Structured Analysis & Design

---

- Systems with stable requirements
- Complex systems
- Concurrent systems



# Functional Decomposition

---

- Distinct input-process-output view of Requirements ●
- Top-down decomposition
- Systems with stable requirements
- Small systems
- Systems with simple interfaces ●

# Object-Oriented Analysis & Design

- Uses an object model with classes and objects, attributes, operations, and messages
- Dozens of object-oriented analysis & design methods
- Prototypes, iterative systems, and evolutionary systems
- Data-intensive systems



# A Word About Requirements

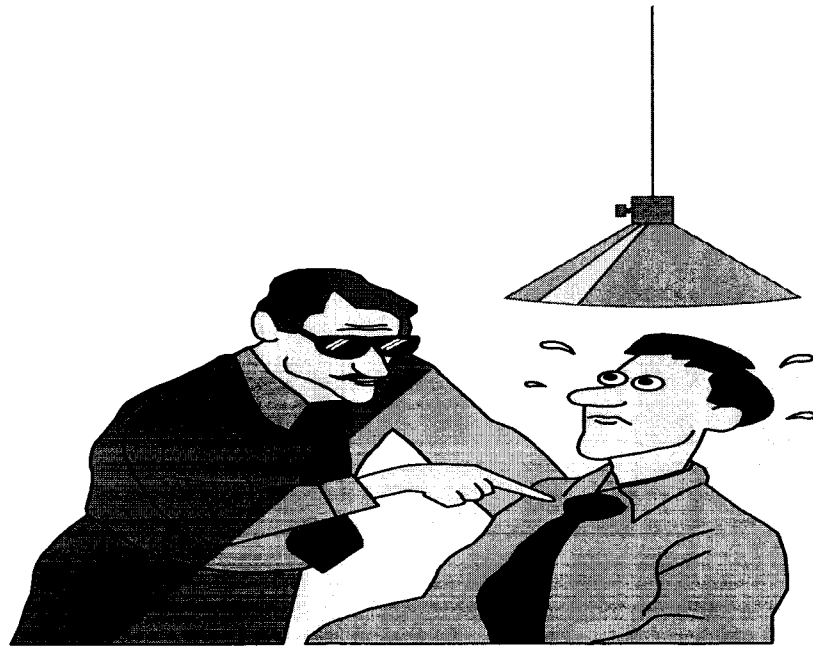
---

- Requirements analysis will focus on "what", not "how," i.e. what data, what functions, what interfaces, and what constraints.

Requirements should be:

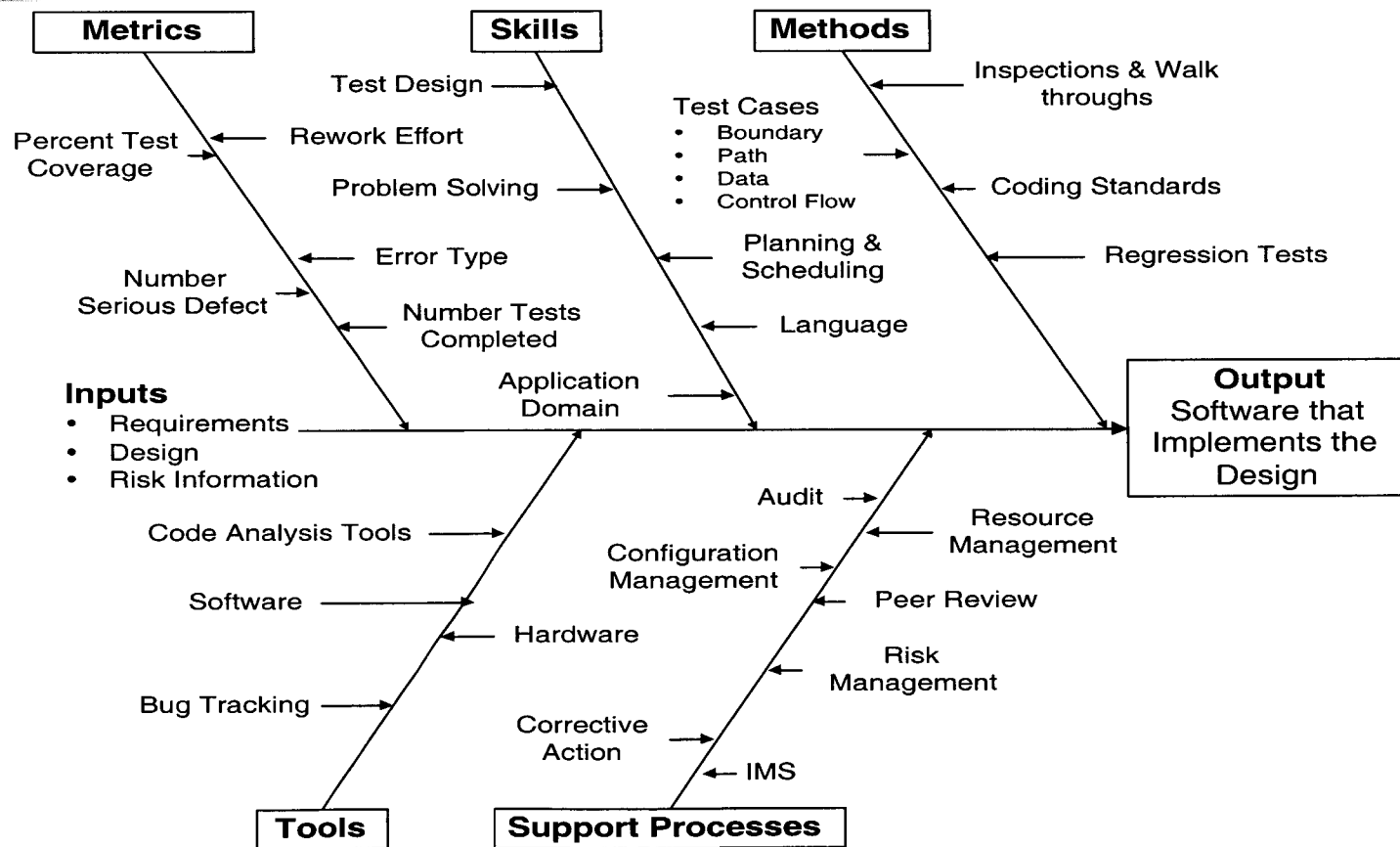
- Feasible and appropriate
- Clear and properly stated
- Proper level
- Testable

# Questions?

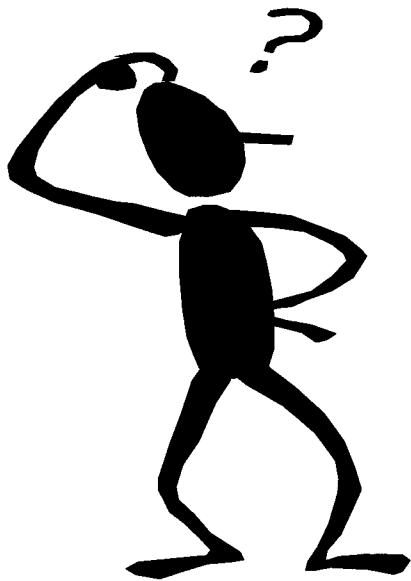




# Code and Unit Test



# Coding Standards



- What is a coding standard?
  - Conventions for use and formats
  - Not required by the computer language
  - Benefits the human
  - Rules and Recommendations
  - How to write code: descriptions and examples

# What's in a Coding Standard?

- Typical table of contents
  - File formats, comments, header information
  - Spacing and indentation, brace style
  - Names, declarations, statements
  - Classes, methods, fields





# Unit Level Testing (White Box)

- Test cases are derived from knowledge of the **internal structure** of the module or unit ●
- Test cases can be derived to exercise:
  - ⇒ **Independent paths** within a unit
  - ⇒ **Logical decisions** on their true and false bounds
  - ⇒ **Loops** at their boundaries and within their operational bounds ●
- Also termed logic-driven or glass box testing



# White Box Testing Methods

---

- Static Analysis
  - Used to identify potential errors such as unreachable code, uninitialized variables, unused variables, etc.
- Loop Testing
  - To force loops to execute a varied numbers of iterations.
- Data Flow Testing
  - To exercise all instructions that define or use a particular variable.



# Basis Path Testing

---

- A strategy for generating test cases that can achieve **100% path (code) coverage** for a single module. ●
- A way to ensures that **all statements** within a module are exercised at least once and **all logical decisions** are exercised on their true and false sides. ●



# Basis Path Testing Approach

---

- Determine the **cyclomatic complexity** of a module ●
- Determine a **basis set** of independent paths through the module
- Preparing test cases that will force execution of **each path** in the basis set ●

# Cyclomatic Complexity Metric, $V(G)$

- Defines the number of **independent paths** through a module/program
  - $V(G)$  = number of regions in a decision-to-decision graph
  - $V(G)$  = number of predicates (decisions) + 1
- Determines the **maximum** number of tests that must be conducted to ensure that **all statements** have been executed at least once in a given module/program



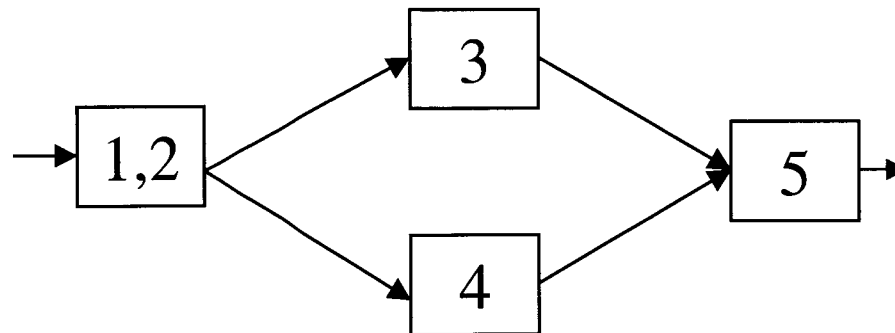
# Example #1

## (Simple Compare Program)

BEGIN

1. READ X AND Y (Both defined as signed integers)
2. IF  $X > Y$
3.        THEN print "X is bigger"
4.        ELSE print "X is not bigger"
5. ENDIF

END



# Example #1 continued

## (Skeletal Decision Table)

		Test Case 1	Test Case 2
Inputs	X	$X_1$	$X_2$
	Y	$Y_1$	$Y_2$
Decisions	$X > Y?$	yes	no
Expected Outputs	Message "X is bigger"	yes	no
	Message "X is not bigger"	no	yes

Skeletal means that input values have not been selected for the test case.

What is the significance of having two (2) test cases?

# Example #1 continued

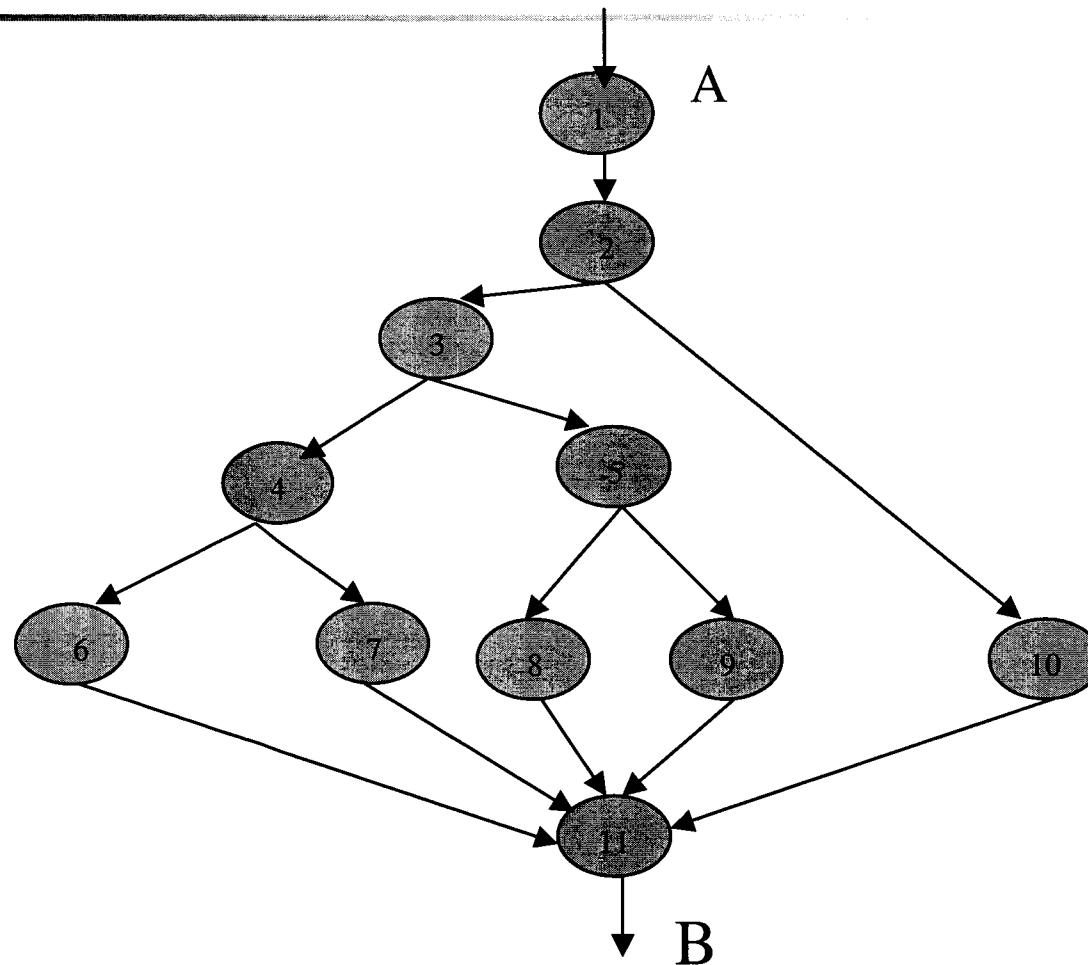
## (Skeletal Decision Table)

---

- Inputs include variable names (X, Y) and the test case conditions ( $X > Y?$ )
- Each case in one column, covering one path through program
  - Test Case 1 covers path 1-2-3-5
  - Test Case 2 covers path 1-2-4-5

# Basis Path Testing

## Example #2



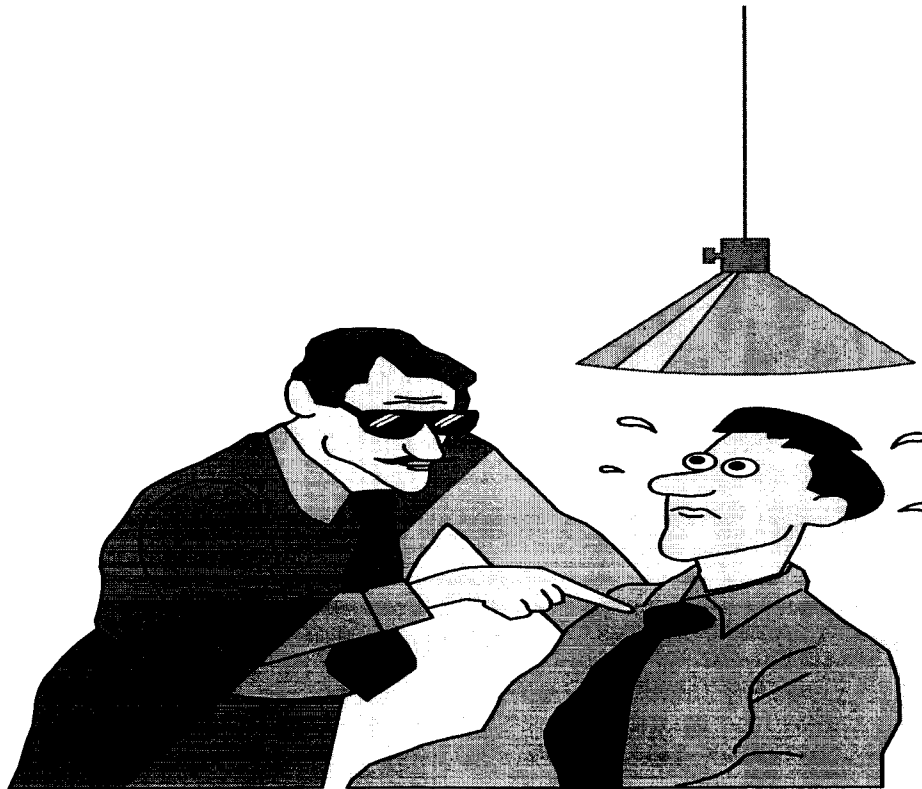
# Basis Path Testing

## Example #2

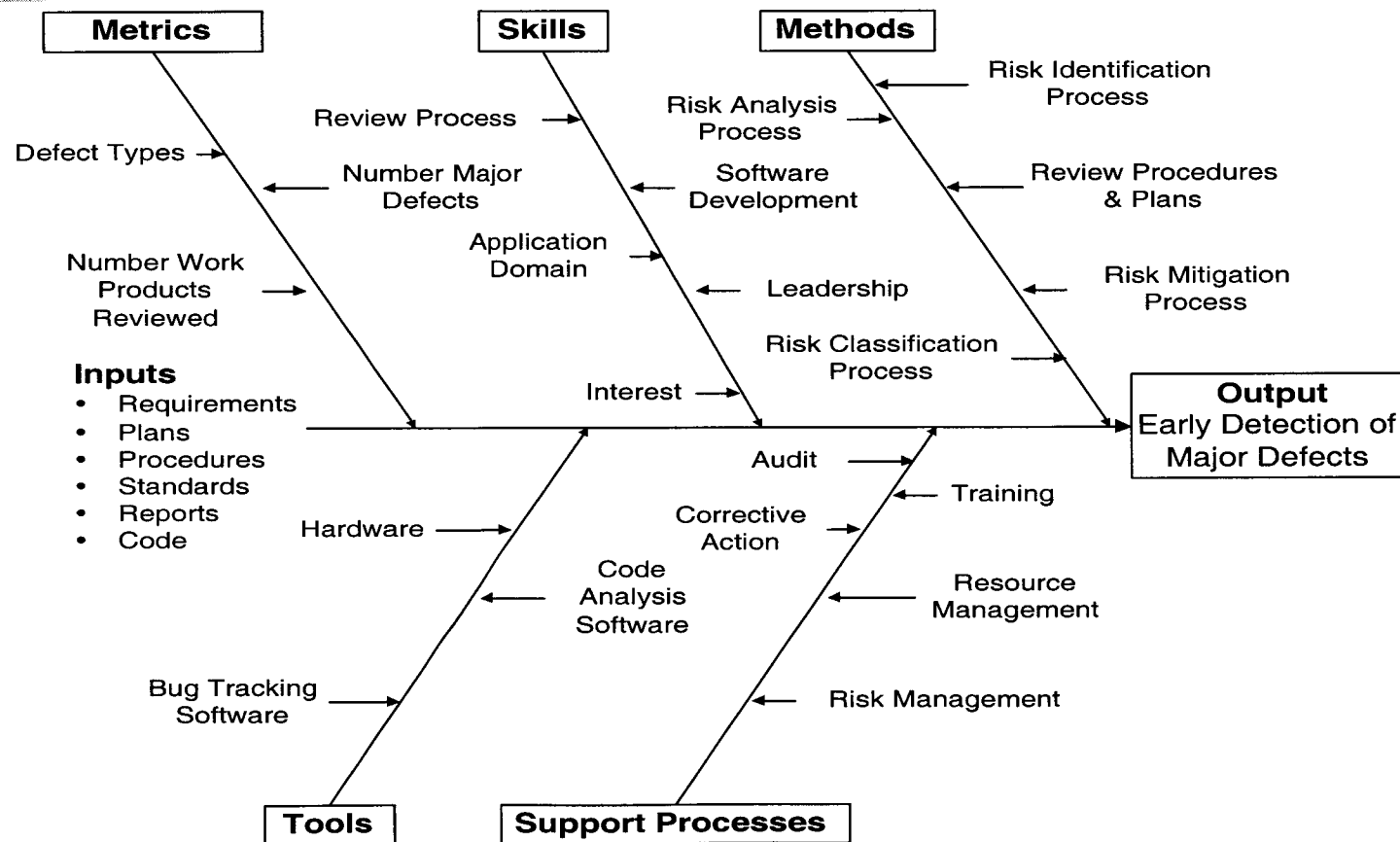
---

- Complexity:\_\_\_\_\_ ●
- Number of independent paths:\_\_\_\_\_
- Number of test cases to ensure 100% code coverage, i.e. every (reachable) statement is executed at least once:\_\_\_\_\_ ●

# Questions?



# Peer Review





# Types of Reviews

---

- **Status Reviews**
  - Project Issues
    - (Schedules, problems, resources)
  - Product Issues
    - (Progress, problems)
- **Peer Reviews**
  - Product and Process Issues
    - (Quality of work products)





# Status Reviews

---

- By leader(s) or team member(s)
- For leader(s) or buyer(s)
- Examples
  - Design reviews
  - Customer interface meetings
  - Development team meeting





# Peer Reviews

- “The purpose of **Peer Reviews** is to remove defects from the software work products **early and efficiently**. An important corollary effect is to develop a better understanding of the software work products and of defects that might be prevented.
- Peer Reviews involve a **methodical** examination of software work products by the **producers' peers** to identify defects and areas where changes are needed.”

# Commonly Reviewed Work Products



- Software plans
  - Requirements specifications
  - Design documents
  - Test plans and procedures
  - Code
  - Procedures and Methods
- 
- 



# Informal Peer Reviews

---

- Poorly defined review process
- Unspecified reviewer responsibilities
- Used for:
  - Low risk products
  - Small products
  - Products of low complexity
- **No Follow-up**

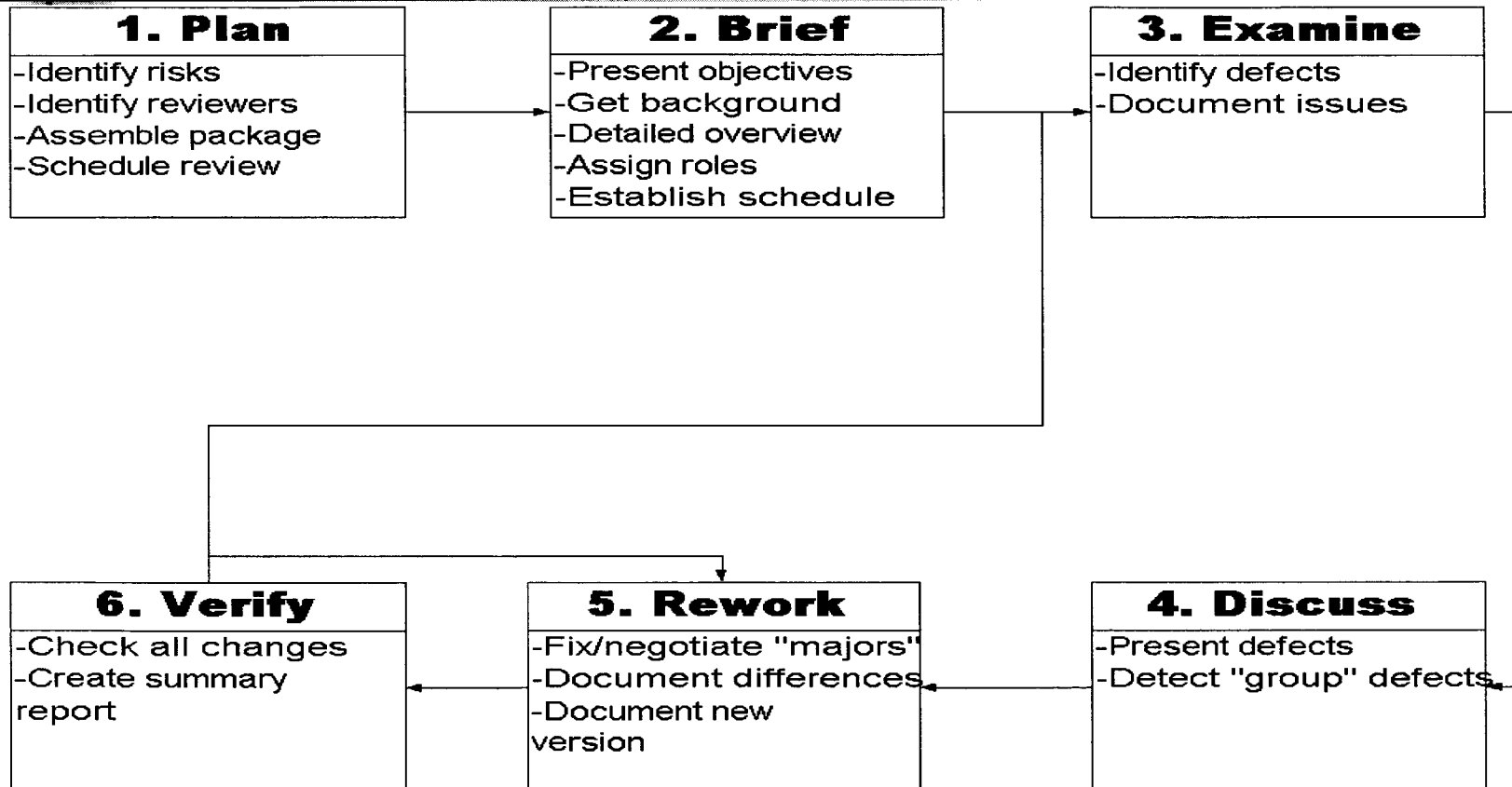


# Formal Peer Reviews

---

- Well-defined “visible” review process ●
- Specified reviewer responsibilities
- Written records
- Used for:
  - Risky products
  - Large and/or complex products ●
  - Early work products
- **Follow-up**

# Formal Peer Review Process



# Defect Categories

- Major defect
  - Potential to cause “big” failure or costly to fix
    - Seriously impairs maintainability
    - Fails to satisfy a requirement
    - Inaccurate statements
    - Exclusion of vital information
    - \_\_\_\_\_
- Minor defect
  - Defect that is not a major

# Types of Defects

- Ambiguous
- Unnecessary
- Untestable
- \_\_\_\_\_
- \_\_\_\_\_
- \_\_\_\_\_
- Missing
- Inconsistent
- Nonconforming
- Incorrect
- Unclear
- \_\_\_\_\_
- \_\_\_\_\_





# Issues

---

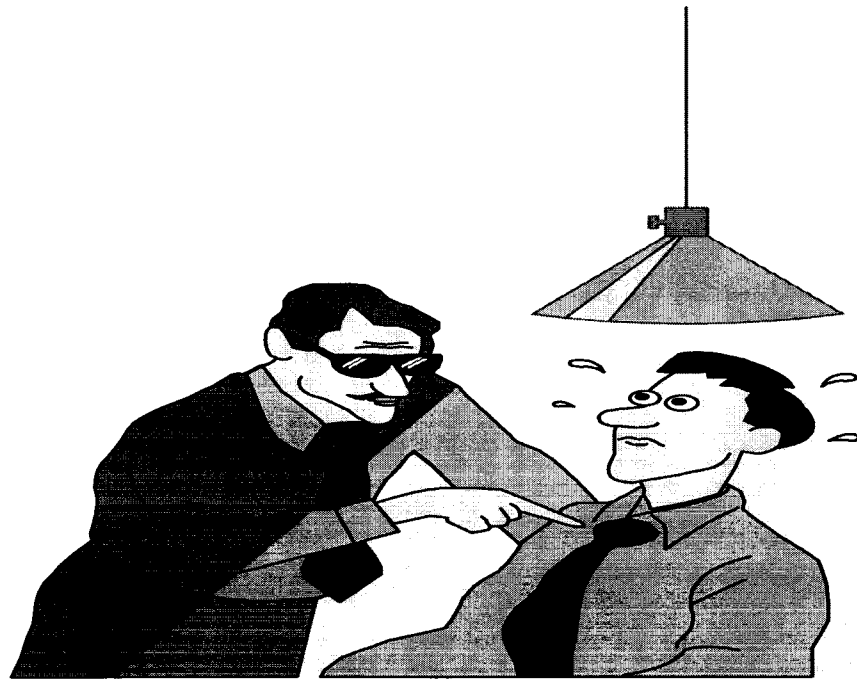
- Any issue requiring effort outside the peer review process:
  - Problems with the standard
  - Problems with the process
  - Problems with a specification



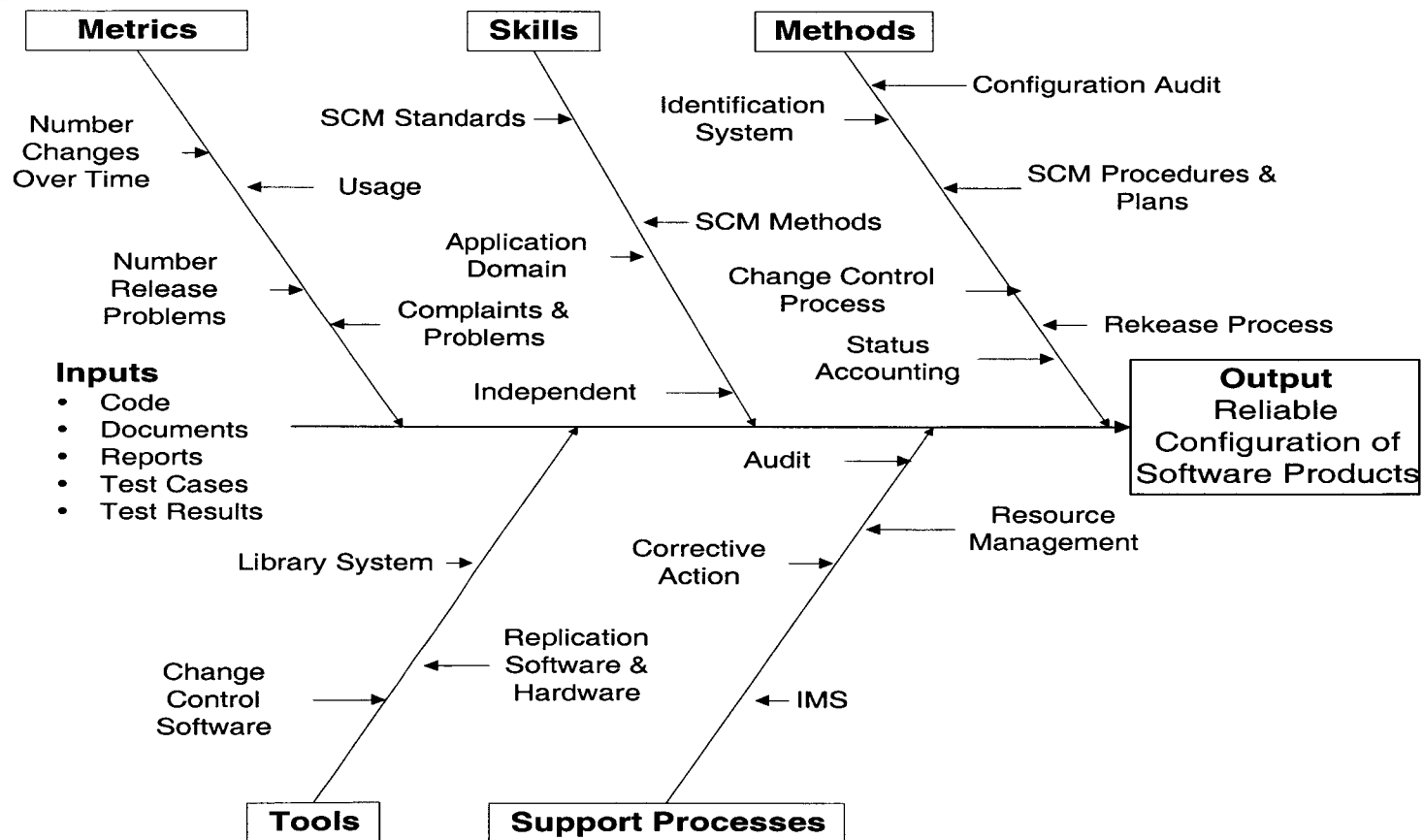
# Why Do Peer Reviews?

- **Find Problems** in the short run
- **Prevent problems** in the long run
- **Better** technical work
- **Communicate** technical information
- **Educate** participants
- **Detect Defects Early** resulting in:
  - Lower costs
  - Lower risk
  - Higher quality

# Questions?



# Configuration Management





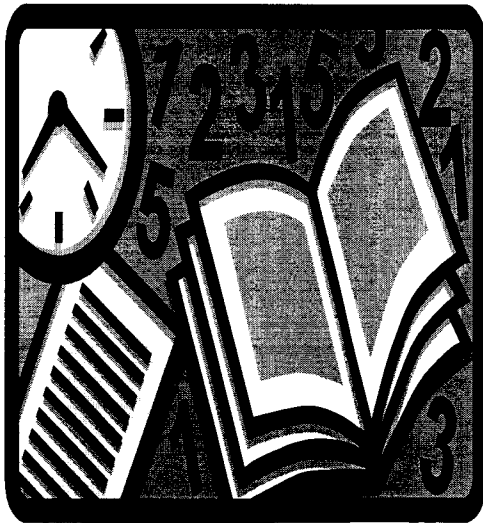
# Configuration Management

---

The process used during software development and maintenance to identify, control, and report functional and physical configurations of software products (e.g., source code, executable code, databases, test scenarios and data, and documentation).

# Components of Configuration Management

- Identify
- Control
- Status
- Audit





# Baseline

---

- A particular version of a document, software release, or system configuration which status and content are known, which is reproducible, and which has some particular and specified designation or reason for existence. For example, a software baseline might be a release incorporating some set of new features that the previous release did not have.



# Configuration Identification

---

- The selection of configuration items (CI) ●
- The issuance of numbers and other identifiers affixed to the CI's and to the technical documentation that defines the CI's configuration
- The release of CI's and their associated configuration documentation ●
- The establishment of configuration baselines for CI's

[MIL-STD-973]



# Configuration Control

- The systematic proposal, justification, evaluation, coordination, and approval or disapproval of proposed changes, and the implementation of all approved changes in the configuration of a Configuration Item (CI) after establishment of the baseline(s) for the CI.

[MIL-STD-973]



# Status Accounting

---

- The recording and reporting of information needed to manage configuration items (CI) effectively, including:
  - A record of the approved configuration documentation and identification numbers.
  - The status of proposed changes, deviations, and waivers to the configuration.
  - The implementation status of approved changes.
  - The configuration of all units of the CI in the operational inventory.

[MIL-STD-973]

Version 1.0



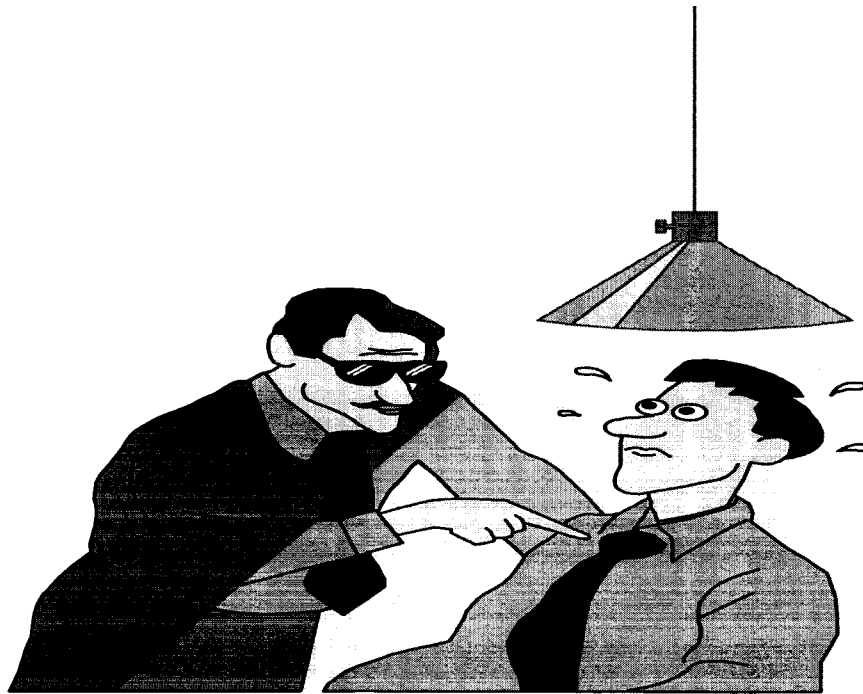
# Audit

---

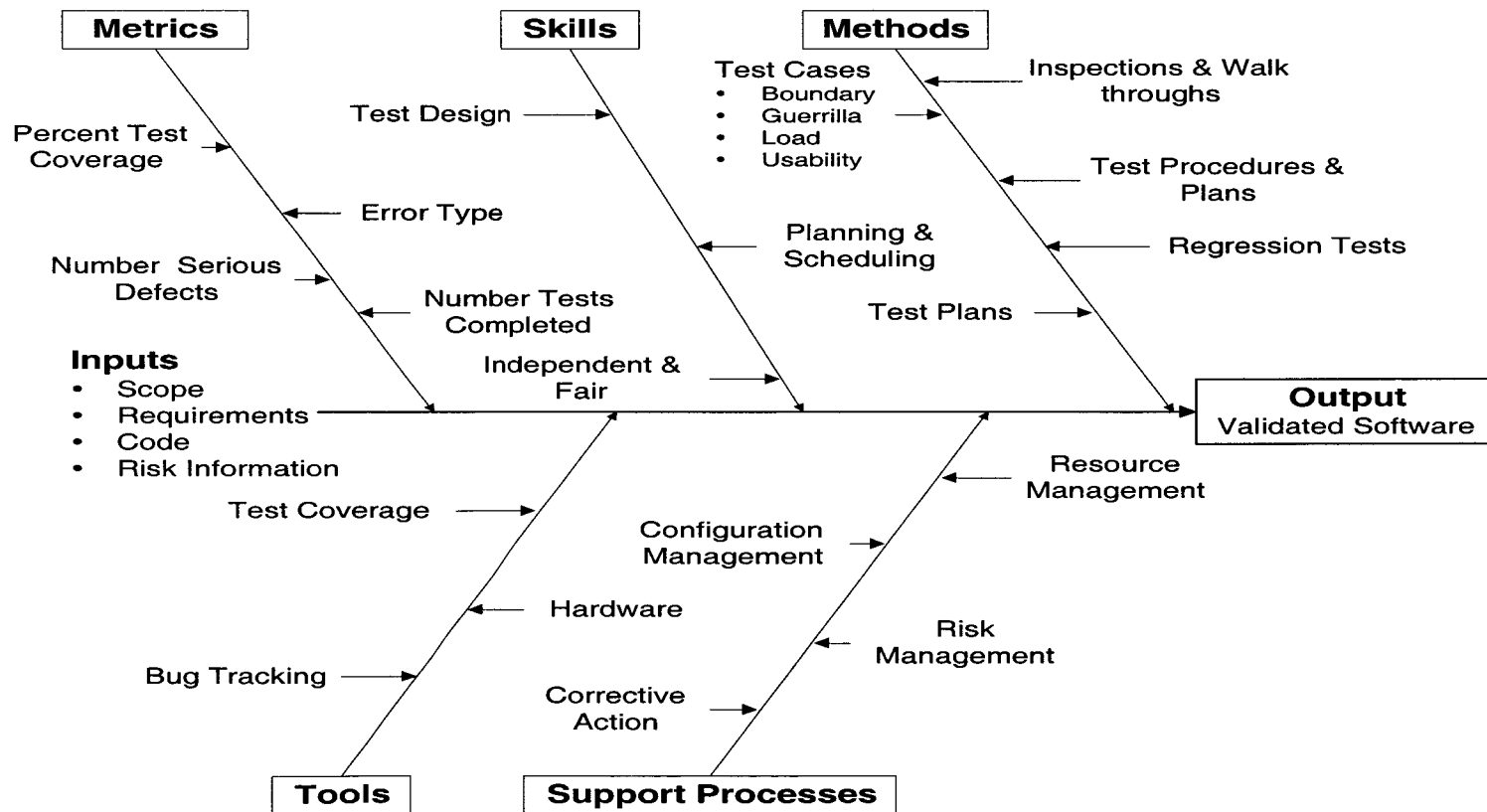
- An independent examination of a work product or set of work products to assess compliance with specifications, standards, contractual agreements, or criteria.

(CMU/SEI-93-TR-25, IEEE-STD-610)

# Questions?



# Software Validation





# The Goal of Software Testing

---

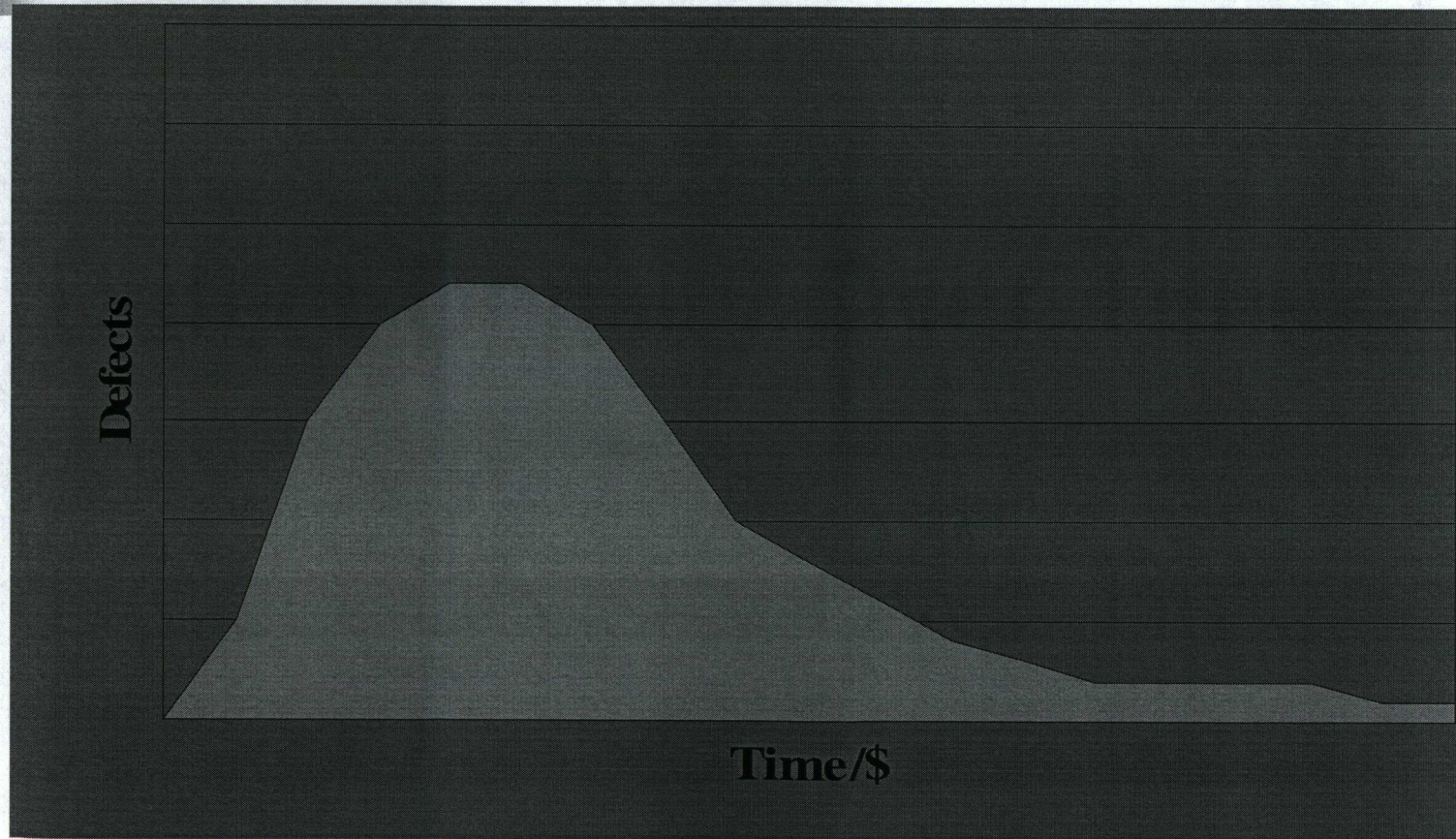
- How do the following statements “**add value?**”
  - Testing is the process of demonstrating that errors are not present.
  - The purpose of testing is to show that a program performs its intended function correctly.
  - Testing is the process of establishing confidence that a program does what it is supposed to do.
  - Testing is the process of executing a program with the intent of finding errors.

# Software Testing Defined

- Software testing is the process of executing a software system:
  - In order to **identify errors**
  - To verify **conformance to requirements**



# Defect Density Over Time





# Testing Concepts

- Testing is the process of executing a program with the intent of **finding** error. ●
- A **good** test case is one that has a high probability of detecting an as-yet undiscovered error.
- A **successful** test case is one that detects an as-yet undiscovered error. ●
- If defects are present, **debugging** determines where and why.

Myers, Glenford J., The Art of Software Testing, John Wiley & Sons, Inc.. New York, 1979.

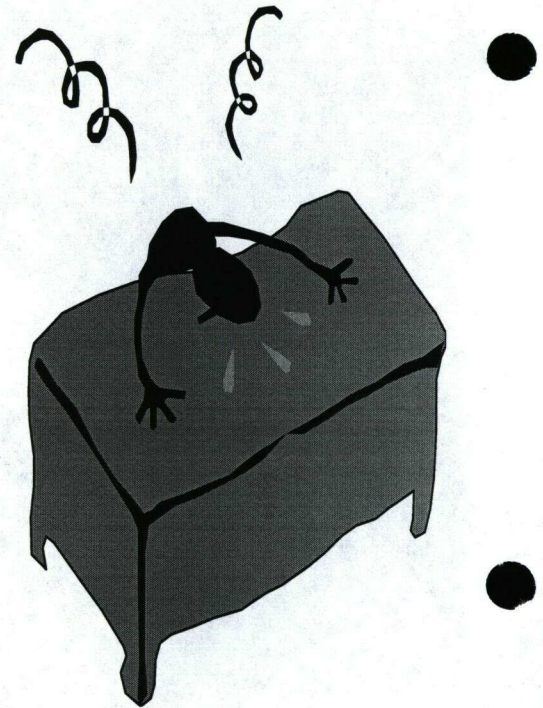
# The (Potential) Cost of Inadequate Testing



- Loss of Life
- Property damage
- Loss of business
- Lost opportunity
- Reduced market share
- Cost of repair
- Any others?

# Exhaustive Testing

- Ideally, testing should exhaustively exercise all program logical paths by invoking the system with all possible input values and combinations.
- To achieve 100 percent confidence through exhaustive testing is impossible.



# Exhaustive Testing

## Example #1

---

- Program analyzes string of ten uppercase alphabetic characters.
  - Exhaustive testing entails  $26^{10} = 1.4 \times 10^{14}$  combinations
  - Would take 4,500 years at one millisecond per test

Learning Tree International, course number 316, p. 316-1-7

# Exhaustive Testing

## Example #2

- Unit has 10 - 20 statements with a DO loop that iterates up to 20 times and 4 nested IF statements
- The number of unique logic paths is  $10^{14} = 5^{20} + 5^{19} + \dots + 5^1$
- Exhaustive testing would take about 1 billion years at one test case developed per five minutes

Myers, Glenford J., The Art of Software Testing, p. 10

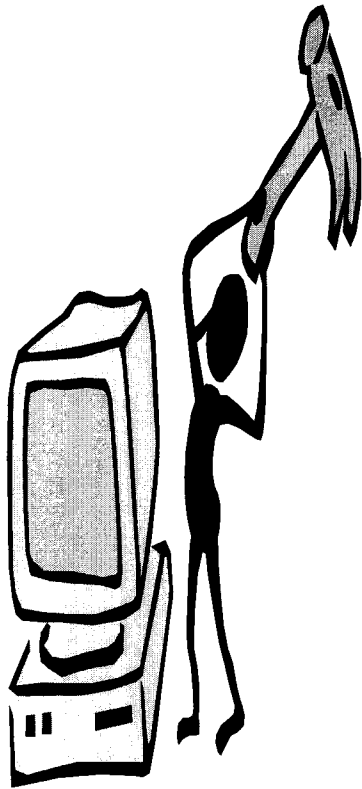


# Exhaustive Testing

---

Testing can be used to show the presence of defects, **but never their absence!**

# Characteristics of a “Good” Software Tester



- **Attitude**
  - What am I going to break today?
- **Creativity**
  - Derive those corner cases.
- **Interpersonal Skills**
  - A team player.
- **Tenacity**
  - Don't give up.
- **Technical skills**
  - Product, testing techniques and tools.

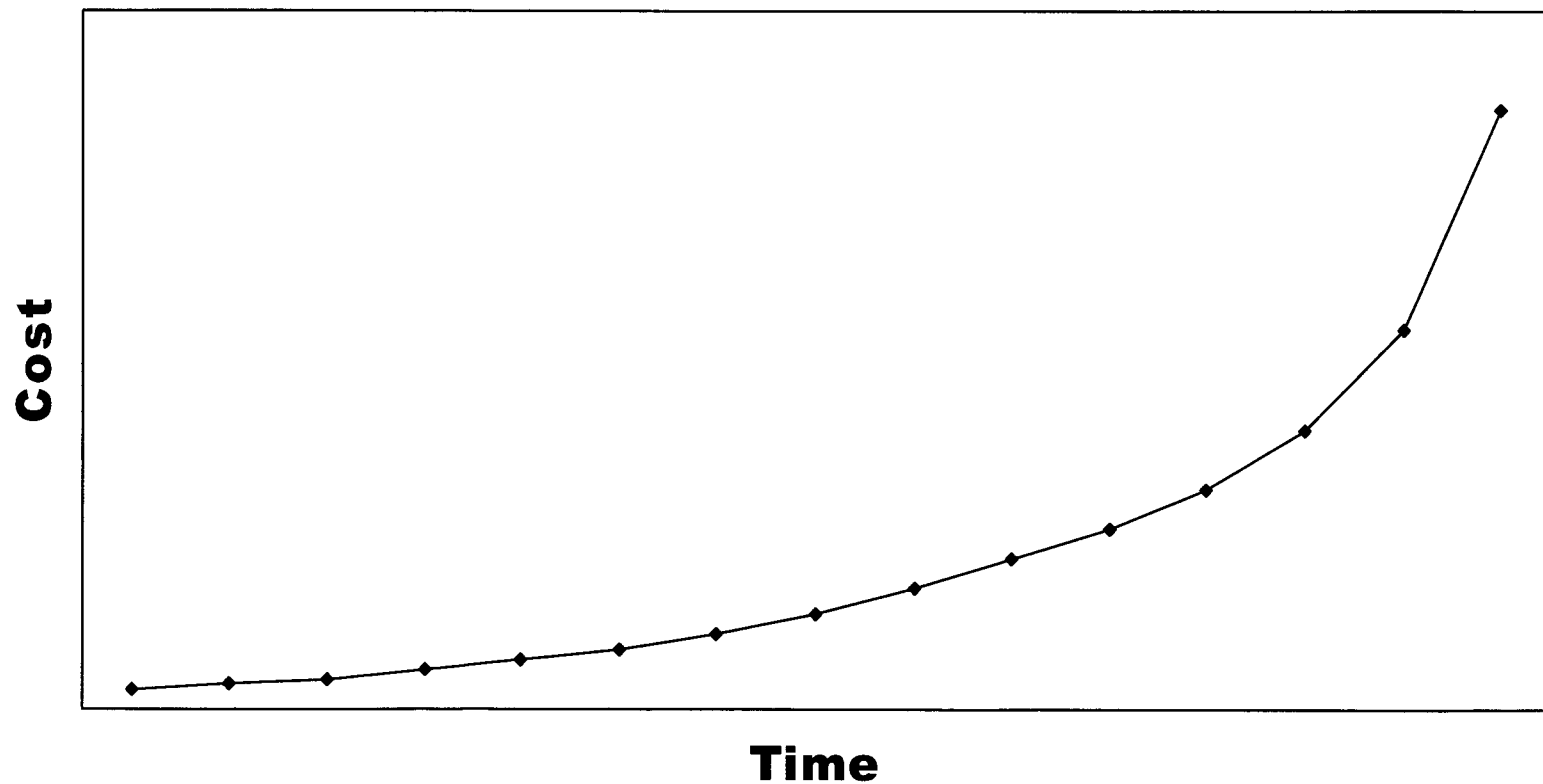
# Testing Benefits and Costs

- A software development organization can expend between **30 and 40 percent** of the total project effort on testing.\* ●
- Testing of life critical software can cost **three to five times** as much as all other software engineering activities combined.\*
- If earlier development phases slip, **extend** delivery date to enable full testing and reevaluate costs as appropriate ●

\*Pressman, Roger S., Software Engineering A Practitioner's Approach, p. 448



# Cost of Finding and Fixing Software Errors





# Software Testing - Black Box

---

- You cannot see into it.
- Test cases can be derived to determine:
  - If the software is particularly sensitive to certain **input values**
  - What **data rates** and data volume can the software tolerate
  - What effect will specific **combinations of data** have on the software operation
- Also called data-driven or input/output-driven testing.



# Equivalence Classes

---

- If you expect the same result from two tests, you consider them equivalent.
  - They all test the same thing
  - If one test catches a bug the others should
  - If one test does not catch a bug the other probably won't
- Valid input conditions must be documented in a specification.



# Equivalence Partitioning

---

- How do you pick the input values for a **specific** test case?
  - Identify an input condition from the SRS, SDD, etc.
  - Partition the input condition into two or more groups, the **equivalence classes**.
  - Use **one** test case to represent an equivalence class.
- Note that there are **two** types of equivalence classes, **valid** and **invalid**.



# Equivalence Classes

- **Valid** equivalence classes represent valid inputs to the software
- **Invalid** equivalence classes represent all other inputs (e.g., erroneous input values)

# Equivalence Classes (Guidelines)

- A test of **one** input value in an equivalence class **represents** the class
- Should yield results that **represent** responses to **all** class members
  - For any input from a **valid** equivalence class the software should produce a normal, **correct output**
  - For any input from an **invalid** equivalence class the software should generate an error or incorrect output

# Equivalence Classes (Example #1)

- If an input condition specifies a member of a set, identify **one valid** equivalence class and **one invalid** equivalence class

Example:                      Set = {EG6334 students}

Valid class:                      {...,Clem, Bobbie, ...}

Invalid class:                      {...anything else...}

# Equivalence Classes (Example #2)

- If a **must be** condition is required, identify **one valid** equivalence class and **one invalid** class

Example: First character in a PIN must be an numeric

Valid class: {0,1,2,3,4,5,6,7,8,9}

Invalid class: {...not numbers...}



# Equivalence Classes (Example #3)

- If an input condition specifies a **range of values** select **one valid** equivalence class and **two invalid** class

Example:            The item count can be 1 to 999

Valid Class:        1 < item count < 999)

Invalid Class:      Item count < 1 & item count > 999).

# Equivalence Classes (Example #4)

- If an input condition specifies the **number of permissible** values select **one valid** equivalence class and **two invalid** classes ●

Example: One through six owners can be listed for the automobile

Valid Class:  $(1 \leq \text{owners} \leq 6)$

Invalid Class:  $(\text{Owners} = 0 \text{ and } \text{owners} > 6)$  ●

# Boundary Value Analysis

- Assumes that the **greater number** of errors tend to occur at the **boundaries** of the input domain than at the center.
- Tests special case input conditions around the **edges of equivalence** classes with the probability of invoking seldom-executed special case code.
  - Coded  $(a + b \geq c)$  rather than  $(a + b > c)$ .
- **Refines** the input selection process for equivalence partitioning.
- Generally **intuitive** to most developers.

# Boundary Value Analysis

## (Guidelines)

---

- If an input condition specifies a **range** bounded by values a and b, test cases should be designed **with values a and b** and with values **just above** and **just below** a and b.
- If an input condition specifies the number of permissible values, test cases should be designed to exercise the **minimum** and **maximum** numbers and with values **just above** and **just below** the minimum and maximum.



# Boundary Value Analysis

## (Guidelines)

---

- If an output condition is a table, test cases should be designed to create an output report that produces the **maximum** and **minimum** number of **allowable table entries**.
- If internal program data structures have **prescribed boundaries** (e.g. array with a defined limit of 100 entries), test cases should be designed to exercise the data structure at its boundaries.

# Boundary Value Analysis (Guidelines)

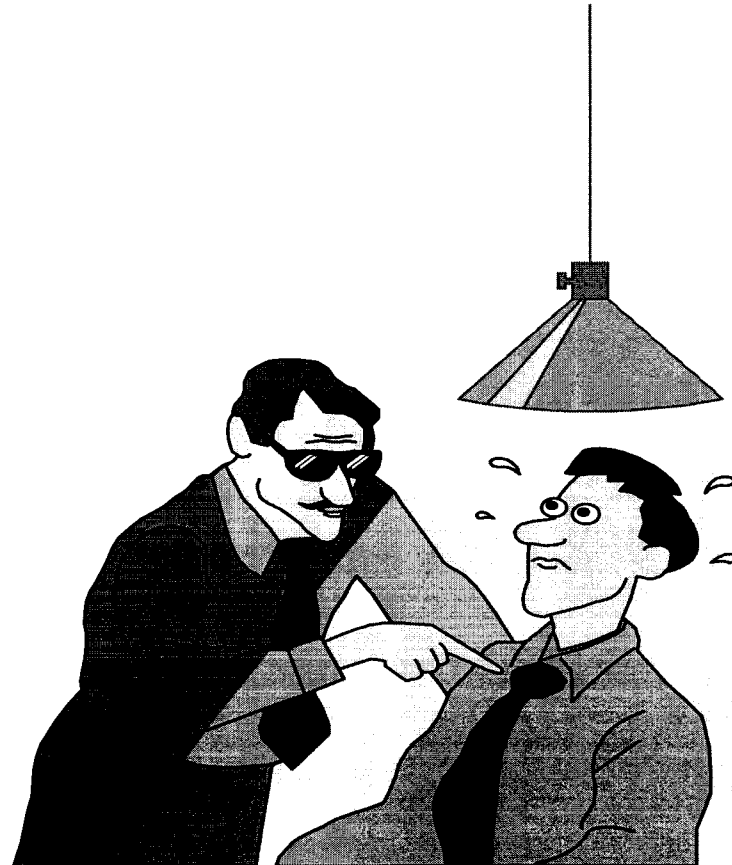
<b>Input Condition</b>	<b>Valid Equivalence Class Values</b>	<b>Invalid Equivalence Class Values</b>
1 to 999	1 and 2 998 and 999	0 1000
1 through 6	1 and 2 5 and 6	0 7



# Cause-Effect Graphing

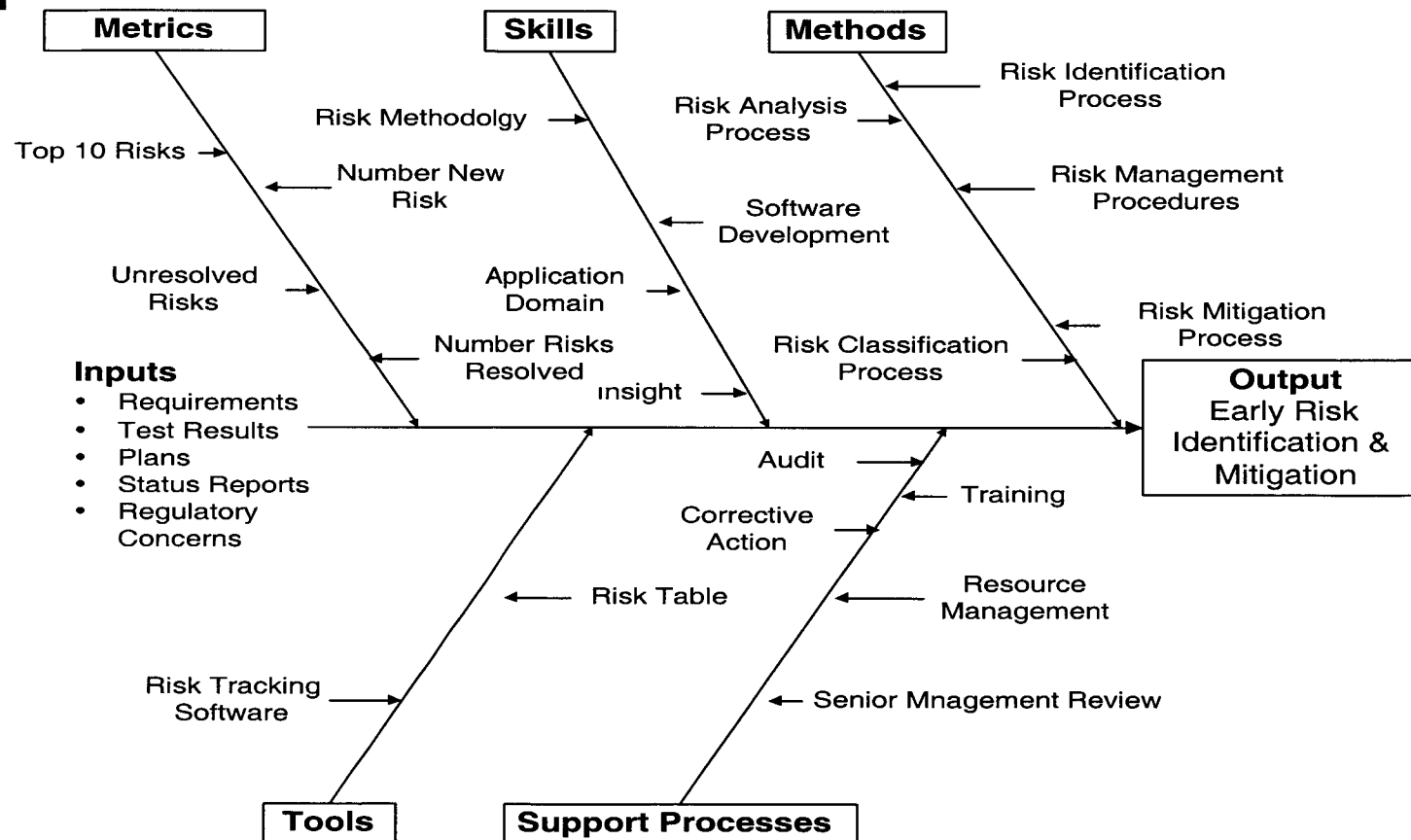
- Test cases are generated based on **combinations of conditions**
  - Example: X is negative & Y is positive in the compare program.
- Equivalence partitioning and boundary value analysis **do not** address **combinations** of input values.
- All test cases derived from equivalence classes for input values A and B pass without error but their product exceeds some limit, e.g. memory.

# Questions?





# Risk Management





# Risk Management

---

- A process of identifying risks and mitigating their effects before these risks disrupt program activities.
  - Risk Identification and Analysis
  - Assigning Risk Criticality
  - Risk Action Planning



# Risk Identification and Analysis


---

- Technical Risks
  - Ambiguous, incomplete requirements
- Environment Risks
  - Training, communication
- Program Constraint Risks
  - Costs, schedule



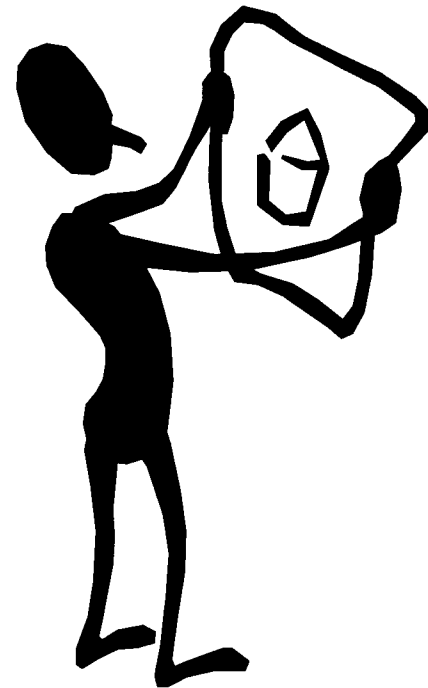
# Assigning Risk Criticality

---

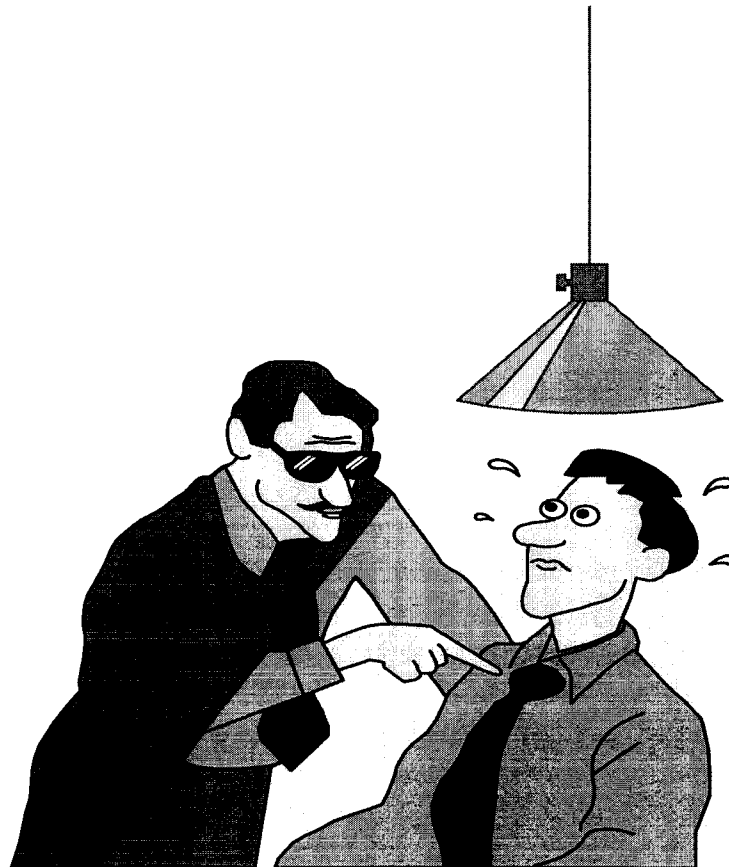
- Impact
    - Negligible
    - Marginal
    - Critical
    - Catastrophic
  - Probability
    - Very high
    - High
    - Medium
    - Low
    - Very low
- 

# Risk Action Planning

- Act Immediately
- Watch
- Transfer
- Delegate
- Strategize

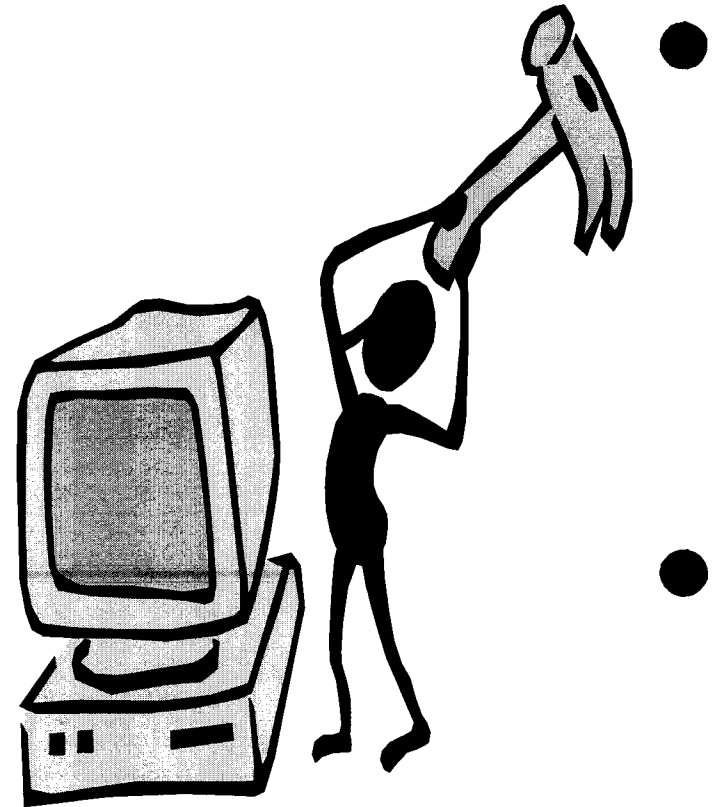


# Questions?



# A Word About Software Maintenance

- Change in software is inevitable
- Hardware deteriorates because of a lack of maintenance
- Software deteriorates because of maintenance



# What is Software Maintenance?



- What does it mean to you? ●
- What does it mean to your organization?
- Is it necessary?
- Are product “versions” defined by maintenance cycles?
- Who should do maintenance? ●





# Kinds of Maintenance

---

- Adaptive
  - Environmental (hardware changes)
- Corrective
  - Fixing errors
- Perfective
  - Making enhancements



# Maintenance Cycle

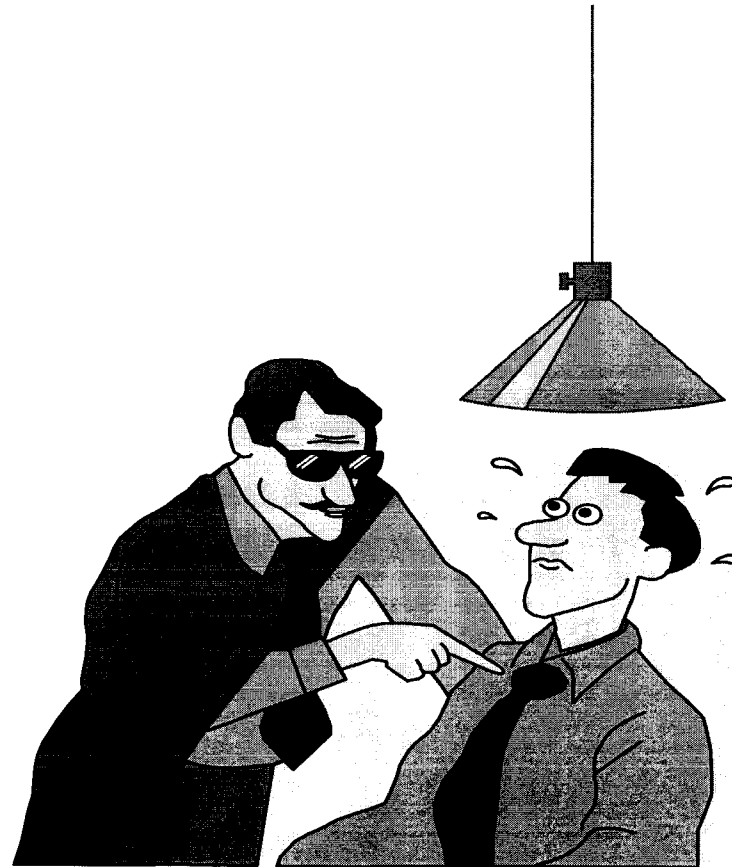
---

- Understand the change: 15%
- Tracing logic: 25%
- Implementing change: 20%
- Testing and debug: 30%
- Reviewing/Updating documentation: 10%

Percentages represent time in phase.

Note similarity to development life cycle!

# Questions?





# Further Reading

---

- Software Engineering Institute, Capability Maturity Model, Peer Reviews ●
- Software Quality Engineering, Technical Reviews & Inspections, Version 4.1
- F.A. Ackerman et al, "Software Inspections: An Effective Verification Process," IEEE Software, May, 1989. ●
- M.E. Fagan, "Advances in Software Inspections," IEEE Transactions on Software Engineering, July, 1996.



# Further Reading

---

- M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, vol. 15, no. 3, 1979.
- D.P. Freedman and G.M. Weinberg, Handbook of Walkthroughs, Inspections, and Technical Reviews, Dorset House Publishing, 1990.
- W.S. Humphrey, A Discipline for Software Engineering, Addison-Wesley Publishing, 1995.