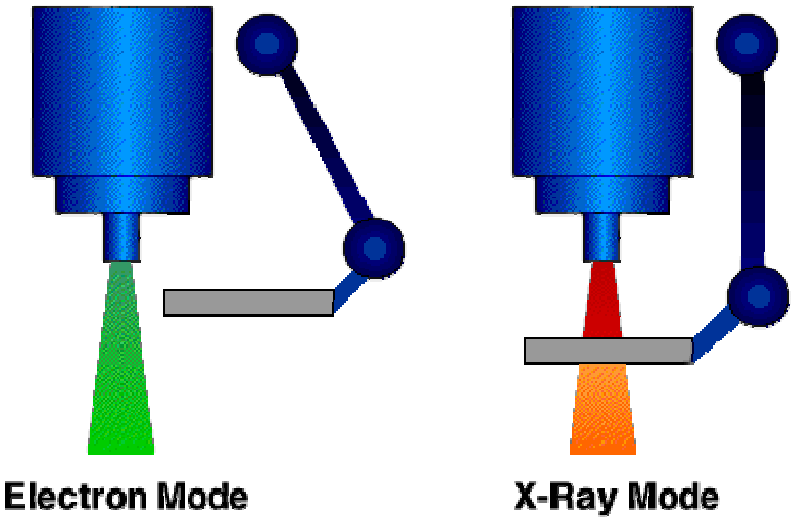
Analysis of Software Artifacts

Introduction to Software Analysis

Jonathan Aldrich

Software Disasters: Therac-25

- Delivered radiation treatment
- 2 modes
  - Electron: low power electrons
  - X-Ray: high power electrons converted to x-rays with shield
- Race condition
  - Operator specifies x-ray, then quickly corrects to electron mode
  - Dosage process doesn't see the update, delivers x-ray dose
  - Mode process sees update, removes shield
- Consequences
  - 3 deaths, 3 serious injuries from radiation overdose



from http://www.netcomp.monash.edu.au/cpe9001/assets/readings/HumanErrorTalk6.gif

source: Leveson and Turner, An Investigation of the Therac-25 Accidents, *IEEE Computer*, Vol. 26, No. 7, July 1993.

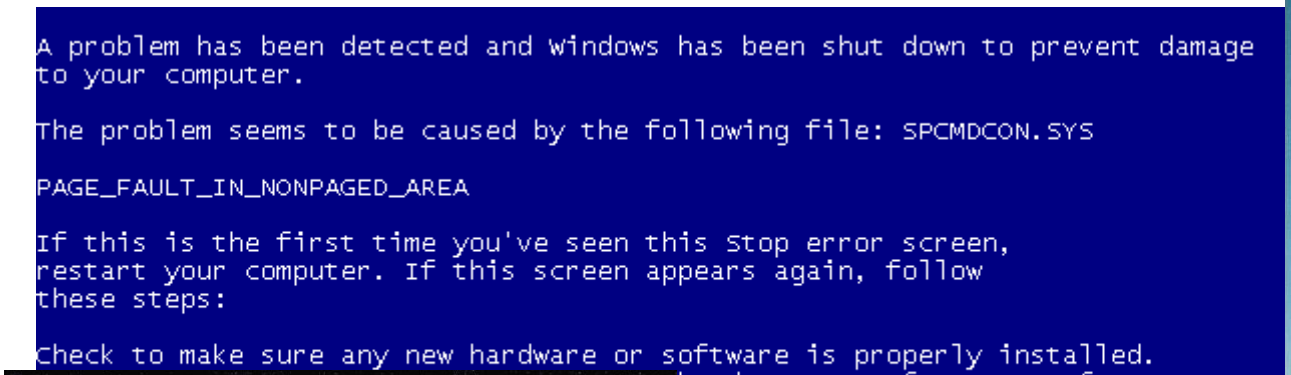
Software Disasters: Ariane 5

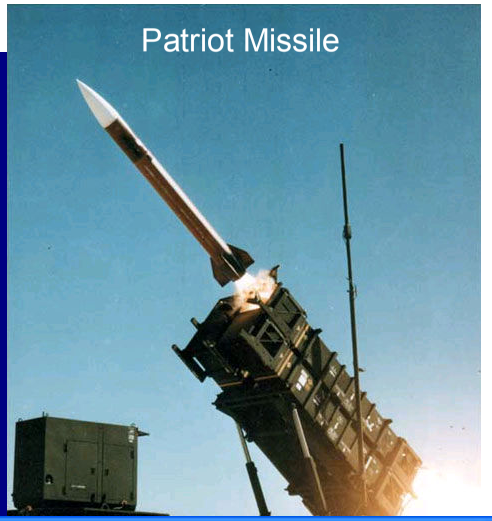
- $7 billion, 10 year rocket development
- Exploded on first launch
  - A numeric overflow occurred in an alignment system
  - Converting lateral velocity from a 64 to a 16-bit format
  - Guidance system shut down and reported diagnostic data
  - Diagnostic data was interpreted as real, led to explosion
- Irony: alignment system was *unnecessary* after launch and should have been shut off
- Double irony: overflow was in code reused from Ariane 4
  - Overflow impossible in Ariane 4
  - Decision to reuse Ariane 4 software, as developing new software was deemed too risky!

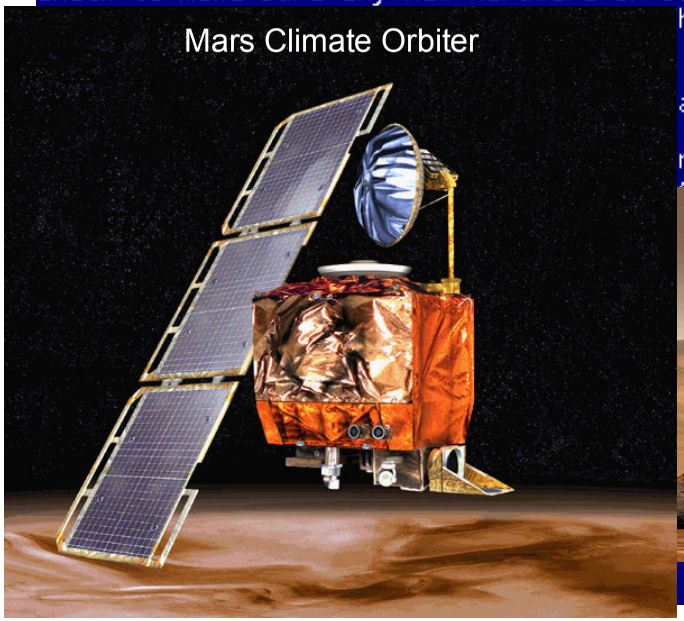
from http://www-user.tu-chemnitz.de/~uro/teaching/crashed-numeric/ariane5/

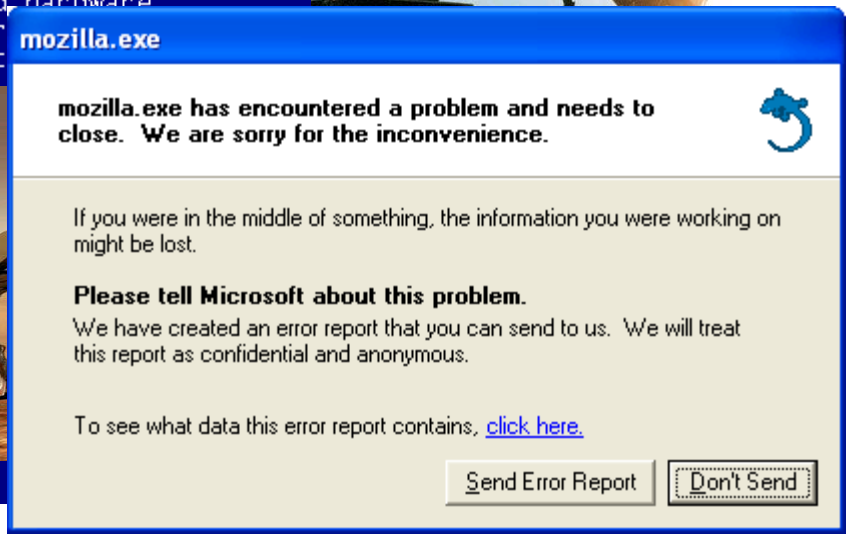
source: Ariane 501 Inquiry Board report

Software Disasters







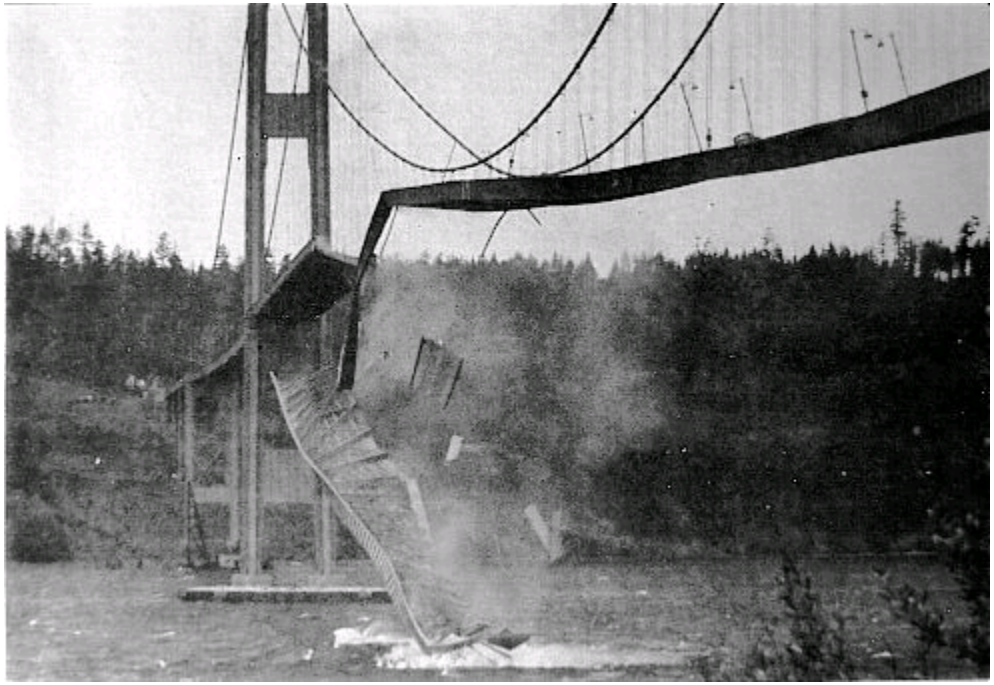


Software Quality Challenges

[adapted from Scherlis]

- **Expense**
  - Testing and evaluation may consume more time and cost in the software engineering process than design and code development
- **Precision**
  - Almost impossible to completely succeed in testing and QA
    - "Very high quality" is rarely achieved, even for critical systems
  - Major gaps in testing and inspection
- **Consequences**
  - NIST report: $60B lost
  - Developers: Holding back features and new capabilities

Hardware Disasters: What's Different?



- Can be equally serious
- But don't seem to be equally common

Why is Building Quality Software Hard?

- For other disciplines we do pretty well
  - Well-understood quality assurance techniques
  - Failures happen, but they are arguably rare
  - Engineers can measure and predict quality

- For software, we aren't doing well
  - Failure is a daily or weekly occurrence
  - How many cars get recalled for a patch once a month?
  - We have relatively poor techniques for measuring, predicting, and assuring quality

Software vs. other Engineering Disciplines

- Every software project is different
  - Classifications of engineering design
    - Routine design: specialize a well-known design to a specific context
      - Most common in engineering projects

Anyone recognize these cars?

Software vs. other Engineering Disciplines

- Every software project is different
  - Classifications of engineering design
    - Routine design: specialize a well-known design to a specific context
      - Most common in engineering projects
    - Innovative design: extend a well-known design to new parameter values
      - Sometimes risky – see Tacoma Narrows Bridge!

Software vs. other Engineering Disciplines

- Every software project is different
  - Classifications of engineering design
    - Routine design: specialize a well-known design to a specific context
      - Most common in engineering projects
    - Innovative design: extend a well-known design to new parameter values
    - Creative design: introduce new parameter values into the design space
      - Involves generating new prototypes
        - Variants of old prototypes, or completely new
      - Relatively unusual, and highly risky



Software vs. other Engineering Disciplines

- Every software project is different
  - Classifications of engineering design
    - Routine design: specialize a well-known design to a specific context
      - Most common in engineering projects
    - Innovative design: extend a well-known design to new parameter values
    - Creative design: introduce new parameter values into the design space
      - Involves generating new prototypes
        - Variants of old prototypes, or completely new
      - Relatively unusual, and highly risky
  - Software
    - Nearly all design is innovative or creative
    - As soon as design is routine, we put it in a library, language or tool!
      - "software manufacturing" will never happen

Software's Unmatched Complexity

- 50 Mloc = 1 million pages
  - What other man-made artifacts have designs this large?
  - We do because software is so flexible and powerful
  - We are limited only by complexity
    - As soon as we manage one level of complexity, the market will push us to add more!
- Worse: *every page matters*
  - Q: Could Windows crash because a third-party device driver has a bug?
  - A: Yes. In fact, that's the biggest cause of Windows crashes.
- *Why?*

Engineering Mathematics

- Continuous mathematics: calculus, etc.
  - Foundation of electrical, mechanical, civil, even chemical engineering

- Some quality strategies
  - Divide and conquer
    - Break a big problem into parts
      - Physical location: floor, room…
      - Conceptual system: frame, shell, wiring, plumbing…
    - Solve those parts separately
  - Overengineer
    - Build two so if one fails the other will work
    - Build twice as strong to allow for failure
  - Statistical analysis of quality
    - Relies on continuous domain

  - These work because the different parts of the system are independent
    - Never completely true, but true enough in practice

Software uses *Discrete* Mathematics

- Old quality strategies fail!
  - Divide and conquer
    - Butterfly effect: small bugs mushroom into big problems
  - Overengineering
    - Build two, and both will fail simultaneously
  - Statistical quality analysis
    - Most software has few meaningful statistical properties

- *Discrete math defeats conventional modularity*
  - Must *leverage discrete math* to analyze software
    - Choose concrete cases based on conceptual categories
      - Functional test coverage
      - Inspection checklists
      - Dynamic analysis
    - Construct proofs based on considering all abstract cases
      - Static analysis
      - Formal modeling
      - Program verification
    - Very *different* from analysis in other engineering disciplines

Questions for Analysis

- How can we ensure a system *does not behave badly?*
- How can we ensure a system *meets its specification?*
- How can we ensure a system *meets the needs of its users?*

Software Analysis, Defined

- *The systematic examination of a software artifact to determine its properties*
  - Systematic
    - Attempting to be comprehensive
      - Test coverage, inspection checklists, exhaustive model checking
  - Examination
    - Automated
      - Regression testing, static analysis, dynamic analysis
    - Manual
      - Manual testing, inspection, modeling
  - Artifact
    - Code, execution trace, test case, design or requirements document
  - Properties
    - Functional: code correctness
    - Quality attributes: evolvability, security, reliability, performance, …

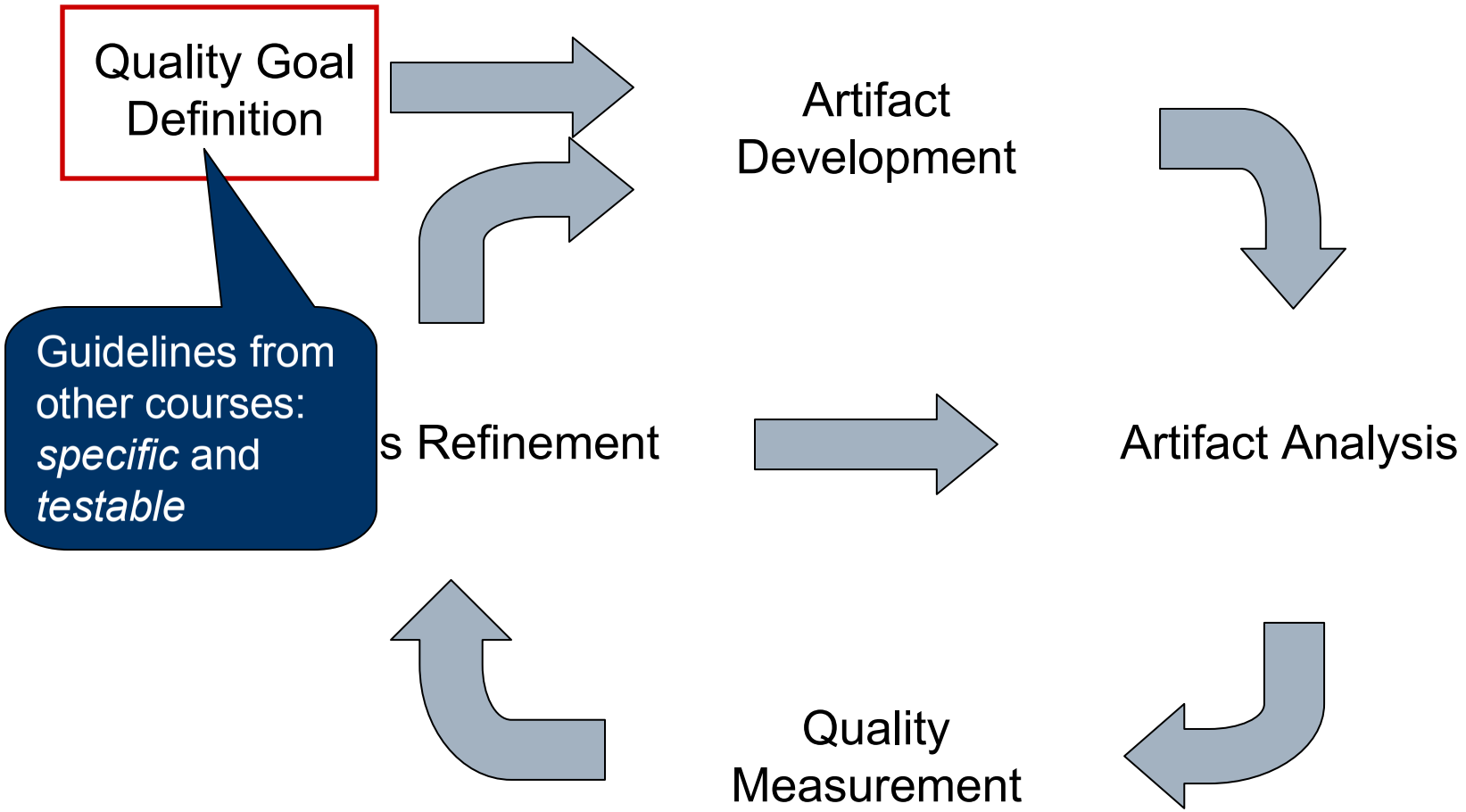
Verification and Validation

[adapted from Scherlis]

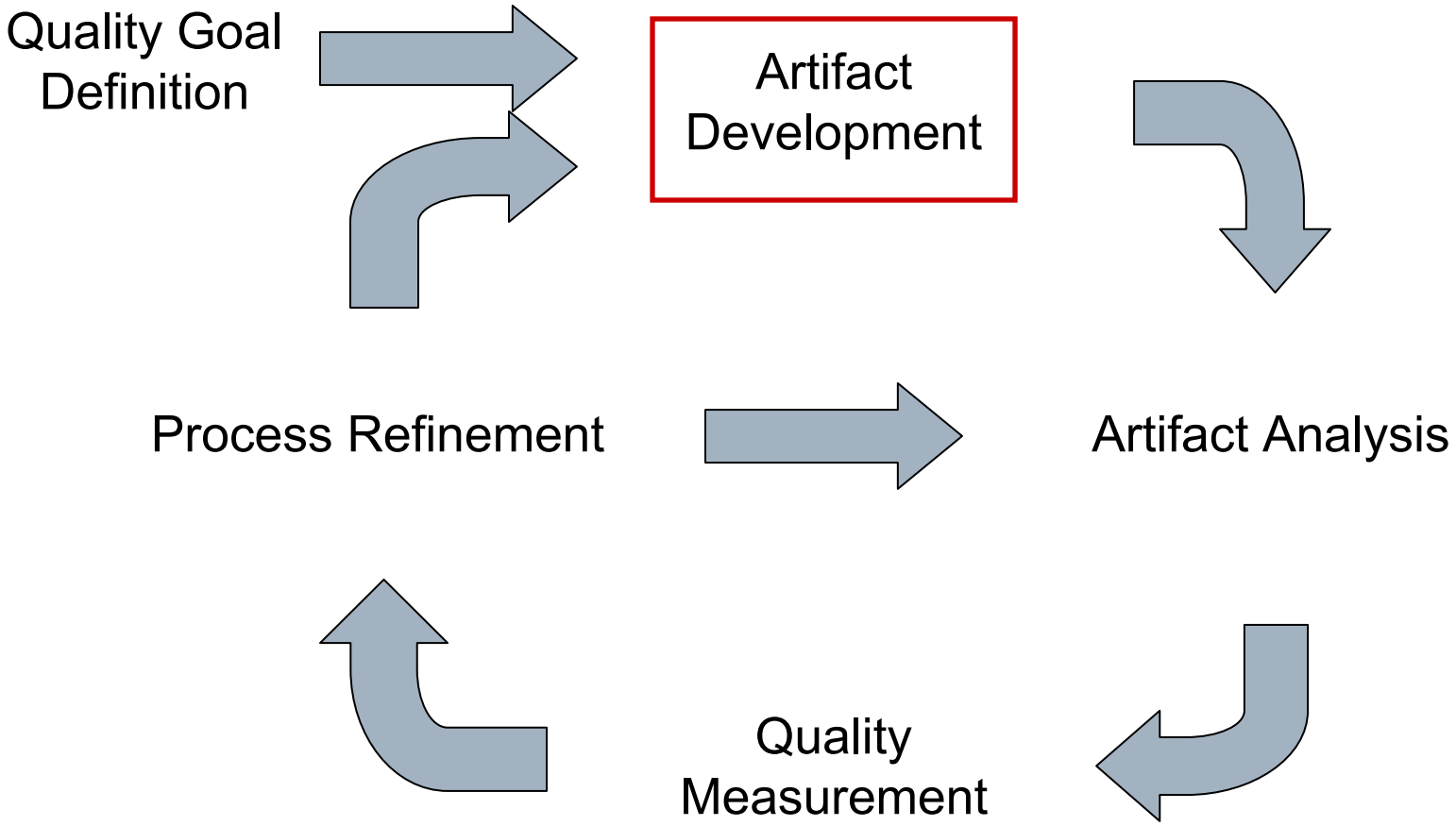
Two kinds of Analysis questions

- Verification
  - Does the system meet its specification?
    - i.e. *did we build the system right?*
  - Flaws in design or code
    - Incorrect design or implementation decisions
- Validation
  - Does the system meet the needs of users?
    - i.e. *did we build the right system?*
  - Flaws in specification
    - Incorrect requirements capture
- We will focus mostly on verification
  - Testing, inspection discussion will touch on validation
  - Other validation approaches beyond scope of course
    - prototyping, interviews, scenarios, user studies

Analysis in a Process Context



Analysis in a Process Context



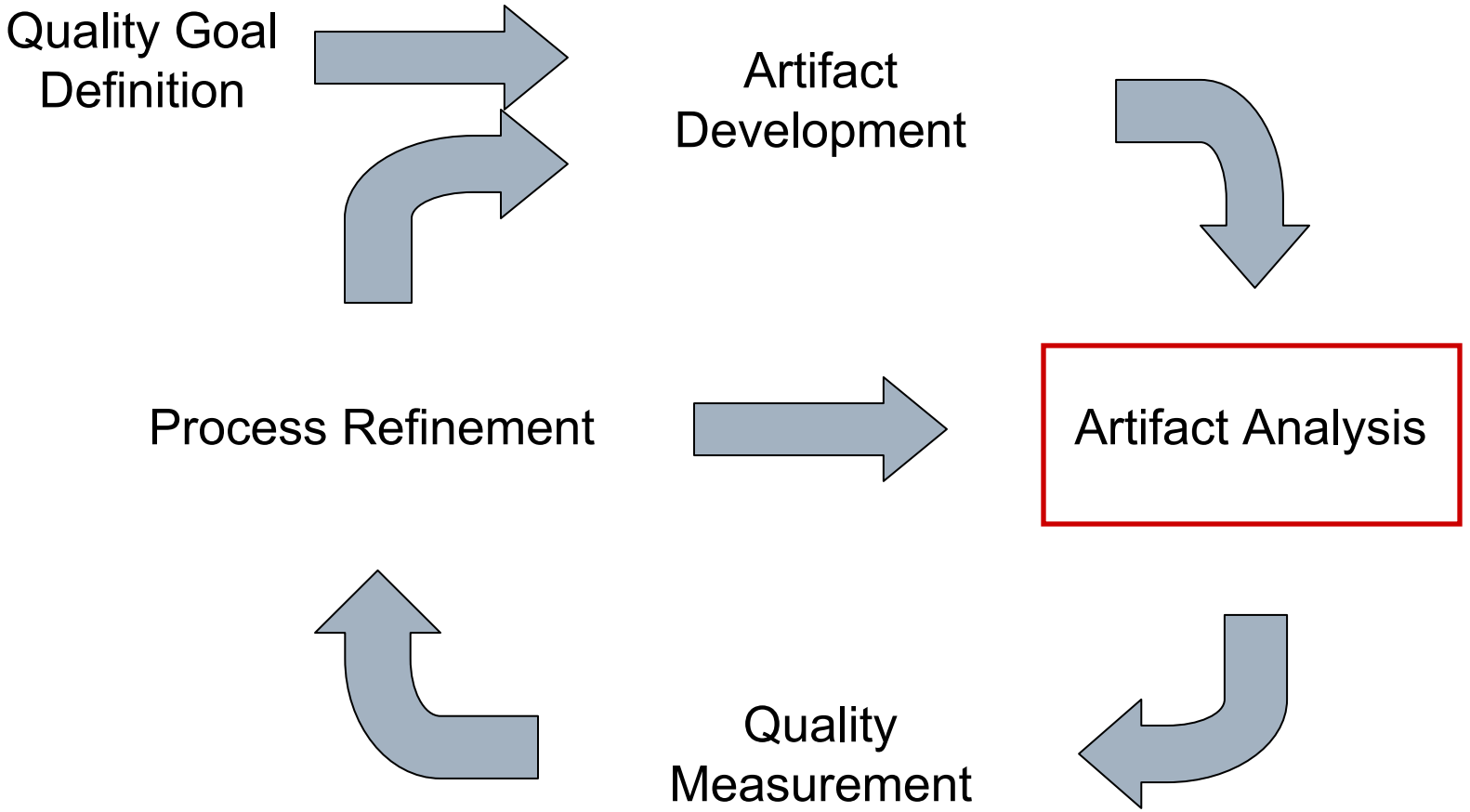
Prevention – Worth a Pound of Cure

- Requirements
  - Involve stakeholders
  - Non-functional attributes
  - Prototyping
- Process
  - Risk management
  - Root cause analysis
  - Continuous improvement
- Design
  - Design patterns
  - Separation of concerns
  - Encapsulation
  - Safe APIs
- Coding
  - Safe languages
  - Safe coding practices

Evaluative techniques like testing are important—but quality cannot be tested in!

[adapted from Scherlis]

Analysis in a Process Context



Principal Evaluative Techniques

[adapted from Scherlis]

- Testing
  - Direct execution of code on test data in a controlled environment
    - Functional and performance attributes
      - Component-level
      - System-level
  - Identify and locate faults – no assurance of complete coverage
- Inspection
  - Human evaluation of code, design documents (specs and models)
    - Structural attributes
      - Design and architecture
      - Coding practices
      - Algorithms and design elements
  - Creation and codification of understanding
- Dynamic analysis
  - Tools extracting data from test runs
    - Finding faults: memory errors
    - Gathering data: performance, invariants
  - Information is precise but does not cover all possible executions

Emerging Evaluative Techniques

[adapted from Scherlis]

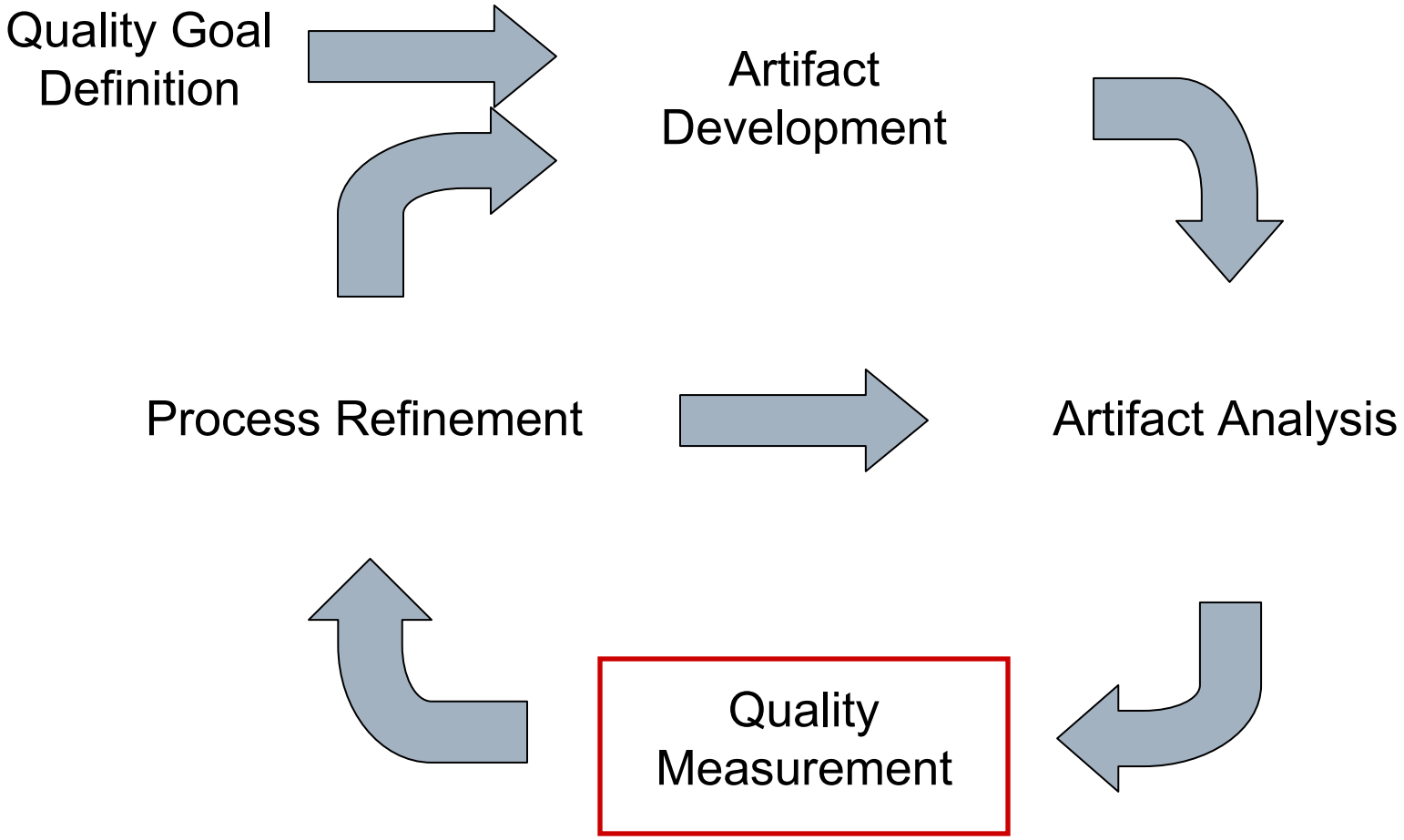
- Modeling
  - Building and analyzing formal models of a system
    - Find design flaws
    - Predict system properties
  - Often tool-supported
- Static analysis
  - Tool-supported direct static evaluation of formal software artifacts
    - Mechanical errors
      - Null references
      - Unexpected exceptions
      - Memory usage
  - Can make (partial) correctness guarantees about all executions
- Formal (i.e. mathematical) verification
  - Formal proof that a program meets its specification
    - Typical focus on functional attributes
  - Some tool support
  - Typically expensive

Criteria for Evaluating Techniques

- Cost
  - Money, time to market
  - Sunk and recurring
- Timeliness
  - Design time
  - During coding
  - During testing
  - After deployment
- Accuracy
  - False positives
  - False negatives
- Development value
  - Is the information actionable?
    - e.g. enough information to fix a bug?
  - Risks of adoption
- Measurability
  - What can we measure about the technique's outcomes?
- Scope: What kinds of defects?
  - Functionality
  - Quality attributes: performance, usability, security, safety, …
  - Design problems

[adapted from Scherlis]

Analysis in a Process Context



Faults, Errors, Failures, Hazards

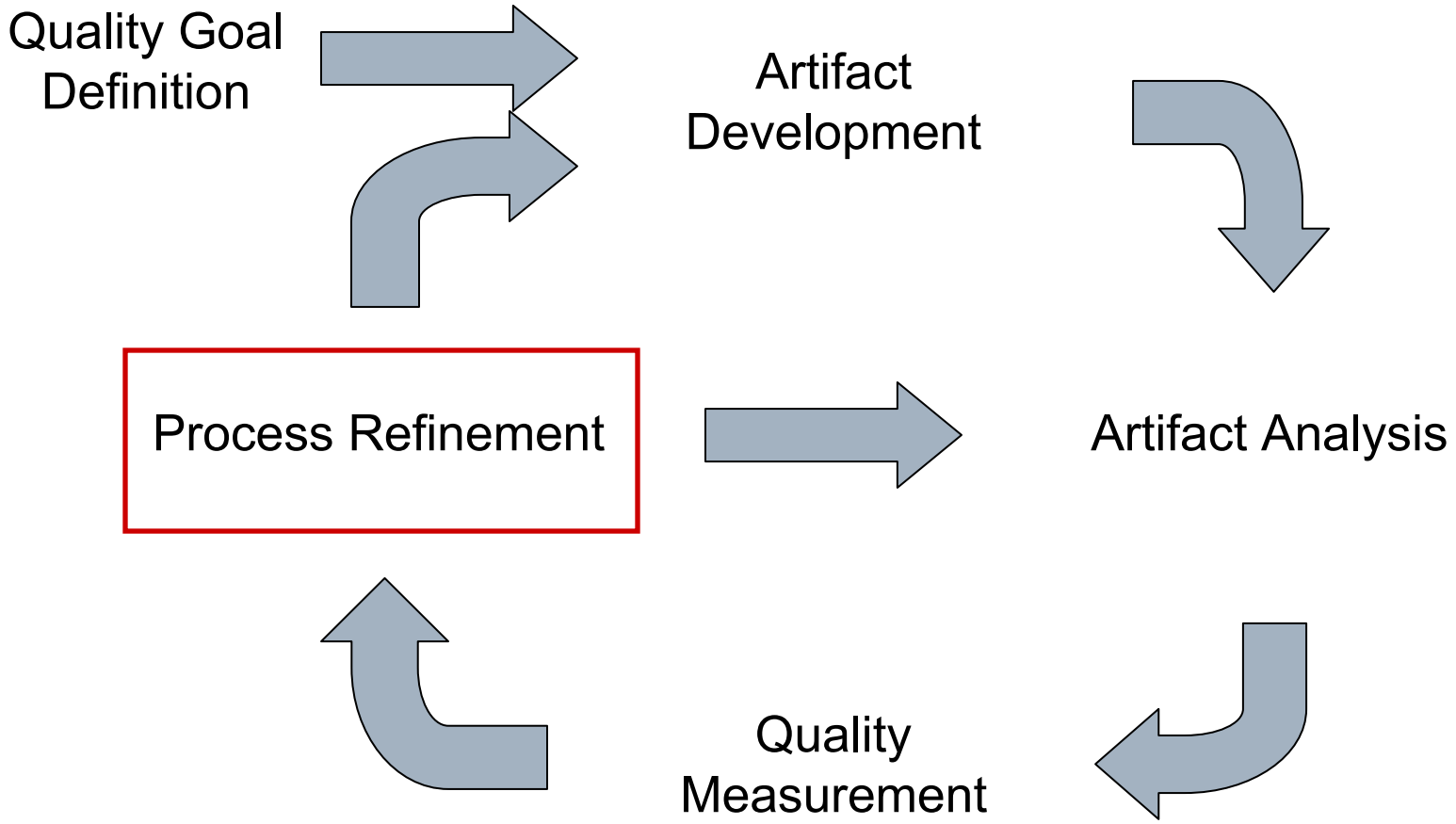
[adapted from Scherlis]

- **Faults, Defects, Bugs**
  - **Fault – a flaw in a physical component**
    - Traditional notion of a fault in hardware reliability theory (physical parts wearing out)
  - **Defect (or Bug) – a static flaw in software code**
    - Syntactically local in code or structurally pervasive
    - Software defects cause errors only when triggered by use.
- **Error – incorrect state at execution time caused by a defect**
  - E.g., buffer overflow, race condition, deadlock, corrupted data
- **Failure – effect of an error on system capability**
  - E.g., program crashes, attacker gains control, program becomes unresponsive, incorrect output
- **Severity – cost of failure to stakeholders**
  - E.g., Loss of life, privacy compromise
- **Hazard – product of failure probability and severity**
  - Equivalent to risk exposure

Robustness / Fault Tolerance

- How does the system behave in the presence of errors in the system or environment?
  - Hardware: memory parity errors, sensor failures, actuator anomalies
  - Software: buffer overflows, null dereferences, protocol violations
  - Environment: network faults, inputs out of range
- Robustness: diminishing the likelihood or severity of **failure** in response to the **defect**
  - Buffer overrun in C == ? in Java
- Strategies for robustness
  - Type systems
  - Run-time system checks
  - Rebooting components
  - Autonomic architectures
  - Self-healing data structures
  - Data validation
  - State estimators

Analysis in a Process Context



Root Cause Analysis at Microsoft

- Gather data on failures
  - Every MSRC bulletin
  - Beta release feedback
  - Watson crash reports
  - Self host
  - Bug databases
- Understand important failures in a deep way
  - Understand why the defect was introduced
    - Not just the incorrect code
  - Understand why it was not caught earlier
    - Process failure
  - Identify patterns in defect data
- Design and adjust the engineering process to ensure that these failures are prevented
  - Developer education
  - Review checklists
  - New static analyses

source: Manuvir Das

Session Summary

- Achieving software quality is difficult
  - Due in part to the discrete nature of software
- Analysis defined
  - *The systematic examination of a software artifact to determine its properties*
- Diversity of analysis techniques
  - Testing, inspection, static and dynamic analysis, model checking, formal verification
  - Each appropriate for different attributes of quality

Analysis of Software Artifacts

Course Overview

Jonathan Aldrich

Course Goals

- Understanding
  - Principles underlying analysis techniques
  - Where different analyses are appropriate
  - Tradeoffs between analysis techniques
  - Theory sufficient to evaluate new analyses
- Experience
  - Applying analysis to software artifacts

Course Outline

- Introduction (this lecture)
  - Introduction to Software Analysis
  - Course Overview
  - Orothogonal Defect Classification
- Traditional analysis techniques
  - Testing
  - Inspection
- Design analysis
  - Design patterns
  - Frameworks
- Program specification and verification
  - Formal specification
  - Proving programs correct
- Static analysis
  - Model checking
  - Dataflow analysis
  - Static analysis applications
- Analysis across the software lifecycle
  - Principles of security analysis; STRIDE
  - Performance analysis: profiling
- Wrap-up

Homeworks and Projects

- Course project: developing board game program
  - Inspect rule specification and implement rules
  - Inspect and test rule implementations
  - Design a generic game framework
  - Inspect the framework design
  - Formally specify the framework
  - Implement the framework and/or plugins to the specification
  - System test the framework and plugins
  - Run a commercial or open-source tool on the system
  - Analyze your experiences using defect data
- Other assignments
  - Prove small programs correct with Hoare logic
  - Check program correctness with the ESC/Java tool
  - Demonstrate your understanding of static analysis and model checking
  - Probe a software system for security violations
  - Measure and tune system performance
  - Develop a quality assurance plan for your studio project

Course Instructor

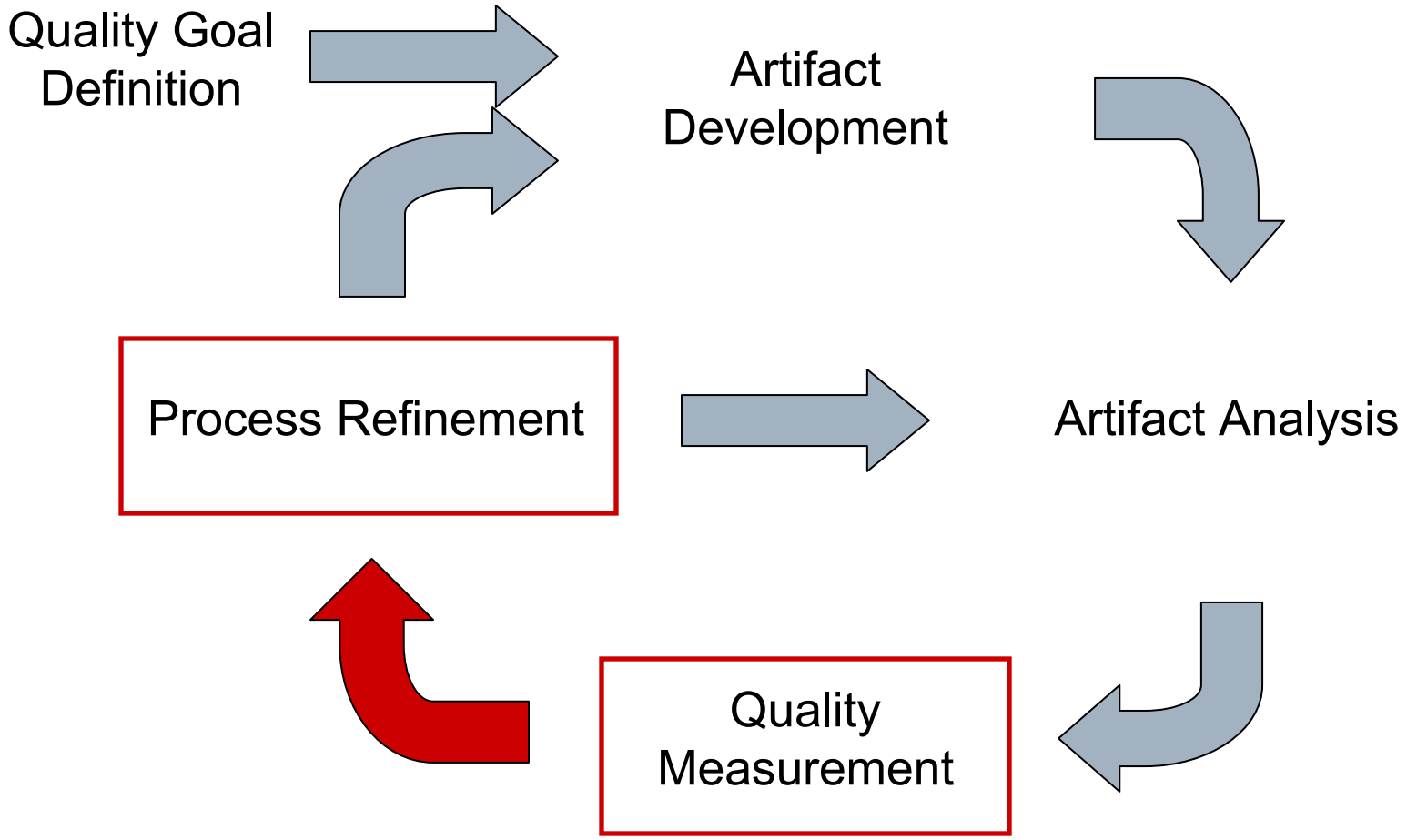
- Jonathan Aldrich, Ph.D.
  - Assistant Professor at Carnegie Mellon University since 2003
  - Active researcher and educator in this field
  - Consultant to companies integrating static analysis into their engineering process
  - Awards for work on static analysis for conformance to a software architecture
    - 2006 NSF CAREER award
    - 2007 Dahl-Nygaard Junior Prize
      - Premier award for early-career researchers who have made technical contributions to the field of Object-Orientation
  - http://www.cs.cmu.edu/~aldrich/

Analysis of Software Artifacts

Orthogonal Defect Classification

Jonathan Aldrich

Analysis in a Process Context



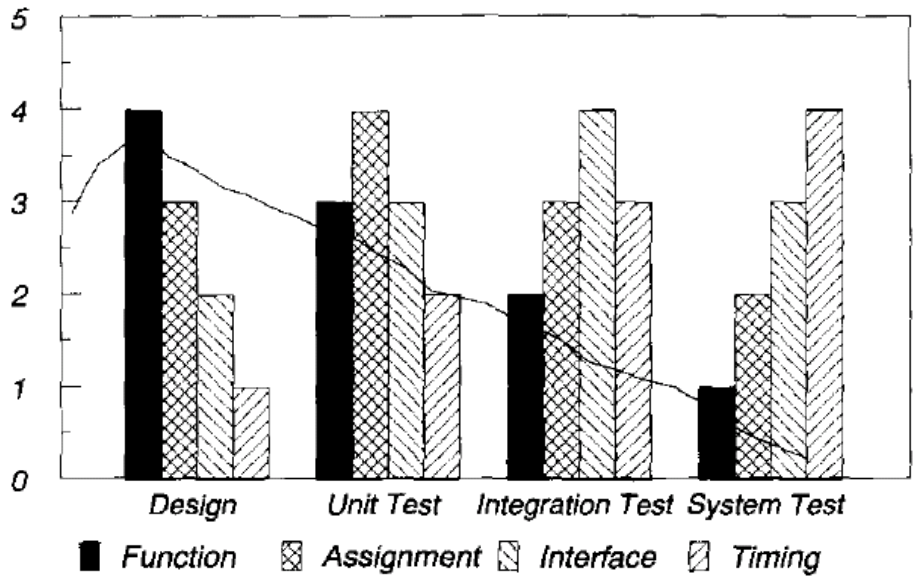
Principled Process Refinement: Orthogonal Defect Classification (ODC)

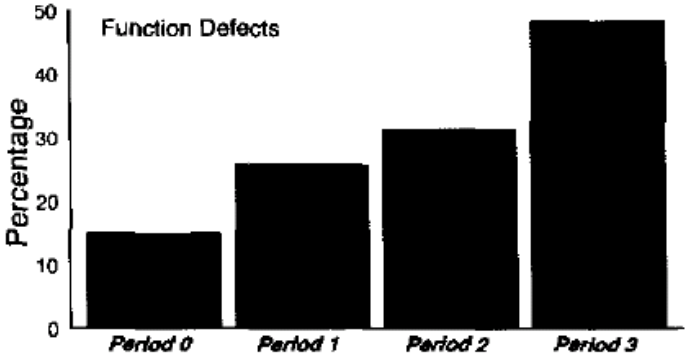
- Analyzing defect data to refine QA process
  - Need to know where defects are introduced and where found
- A defect's type is related to where it was introduced
  - Hypothesis: can estimate defect type with less bias than directly estimate of phase where defect was introduced

ODC Defect Types

Defect Type	Description	Source
Function	Capability affected requiring design change	Design
Interface	Error interacting with other components	Low-level design
Checking	Data not properly validated before use	Low-level design or code
Assignment	Assigned / initialized incorrect value	Code
Timing	Incorrect management of shared resources	Low-level design
Build	Mistakes in libraries, change management	Libraries/tools
Documentation	Incorrect documentation	Publications
Algorithm	Local problems that do not require design change	Low-level design

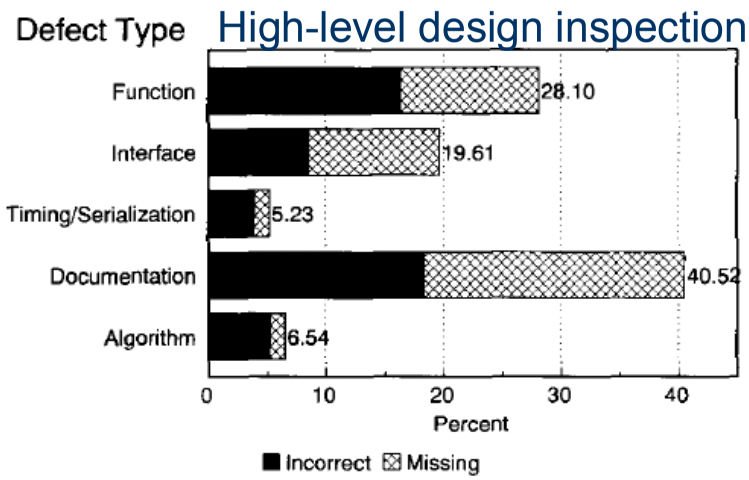
Example: Analyzing Defect Type Distribution over Time

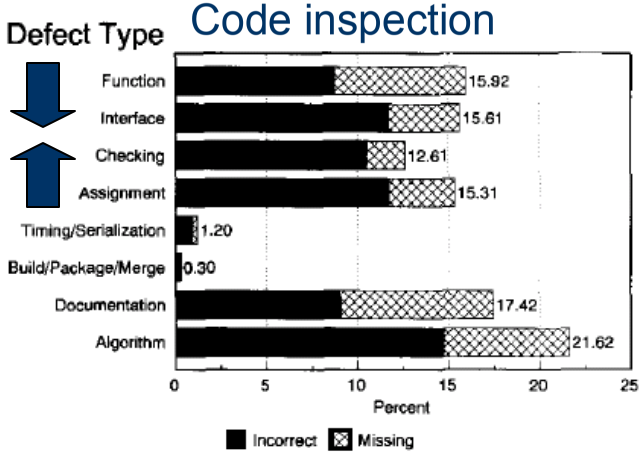


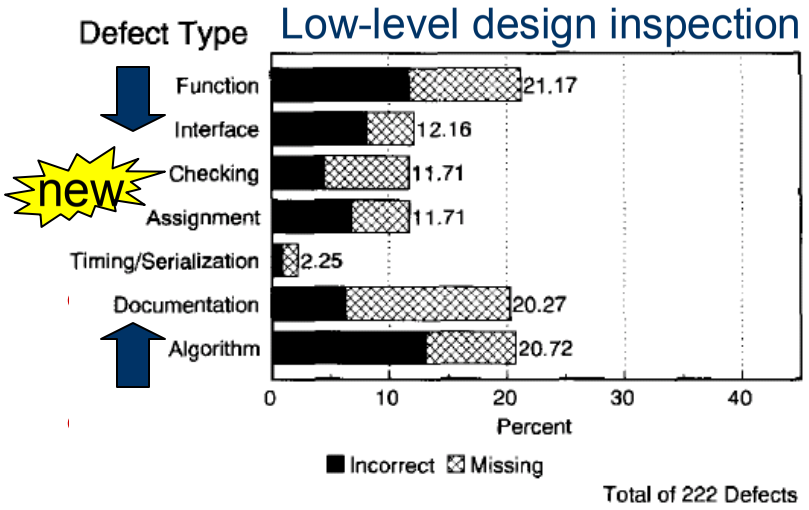


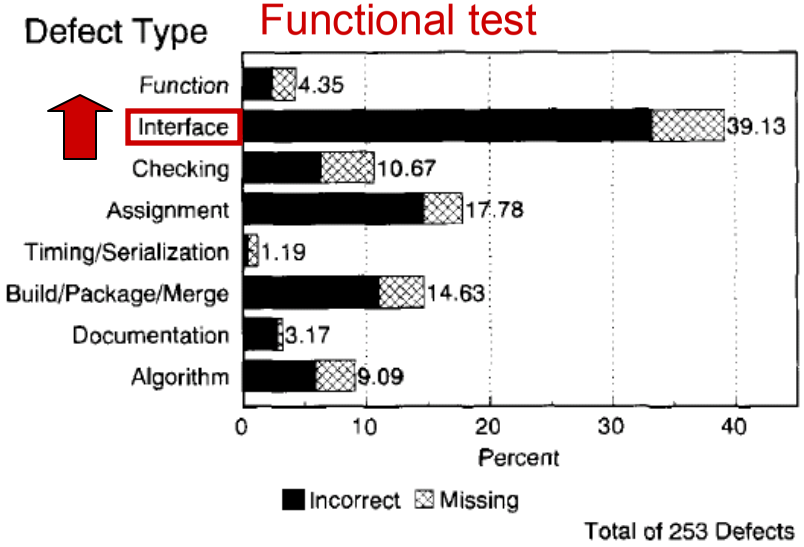
- Ideal curves
  - Function peaks in design inspection
  - Assignment peaks at unit test
  - Interface peaks at integration test
  - Timing peaks at system test
- Actual curve for function increases at each stage!
  - What does this tell us? What would you do about it?
    - Something was missed in design
    - Better to re-do design rather than keep testing

Example:
Analyzing Defect Type Distribution over Time

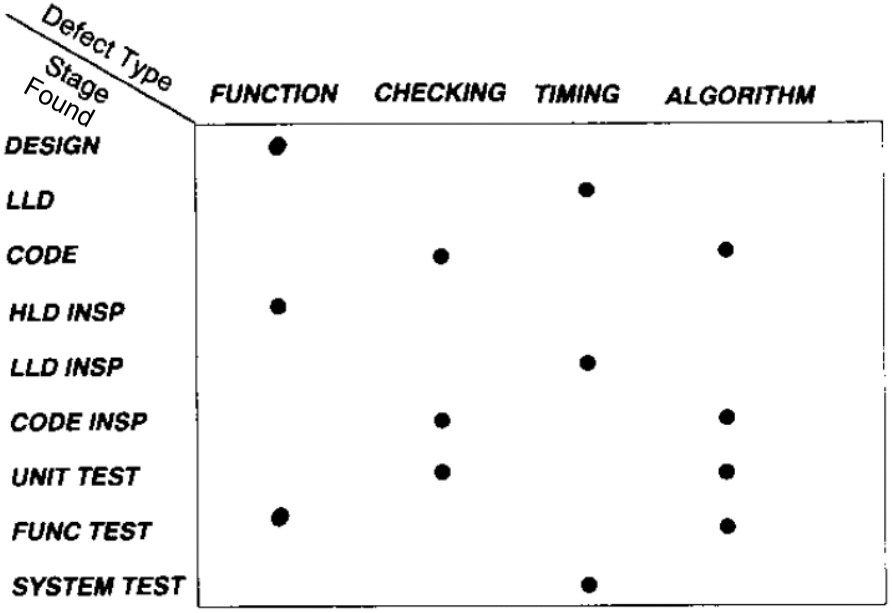


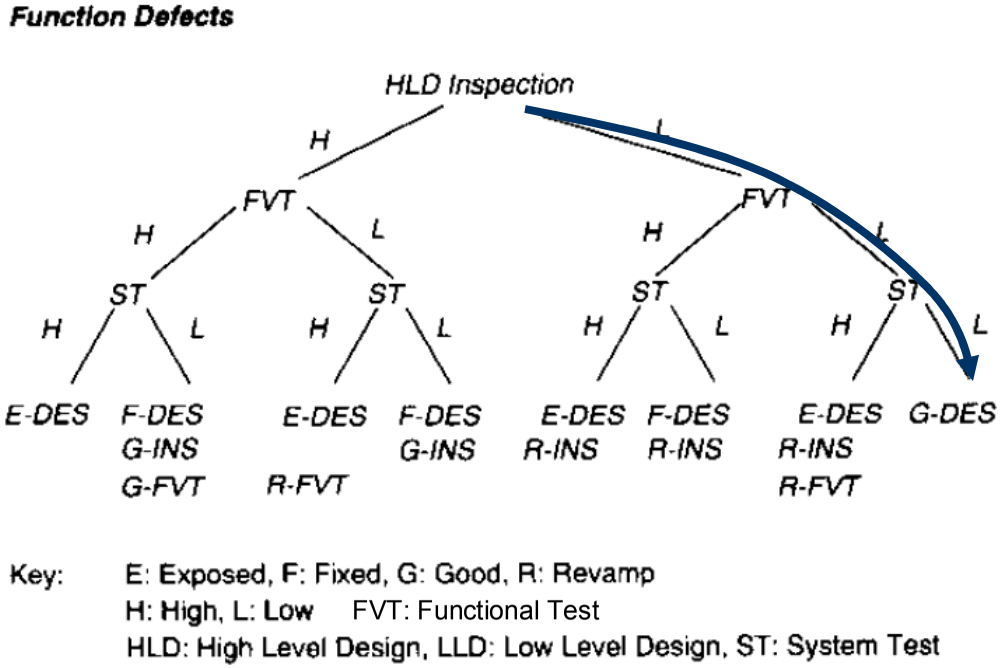






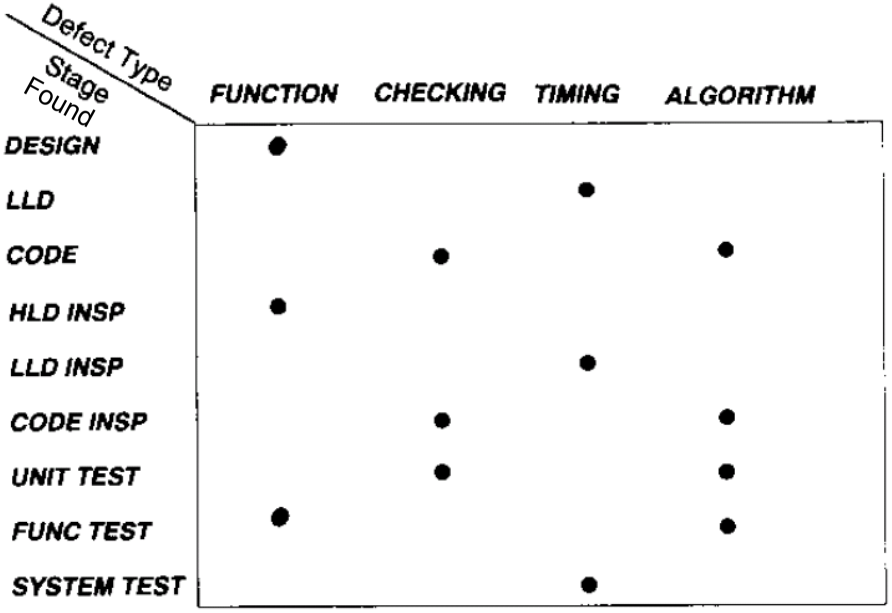
Process Inferences

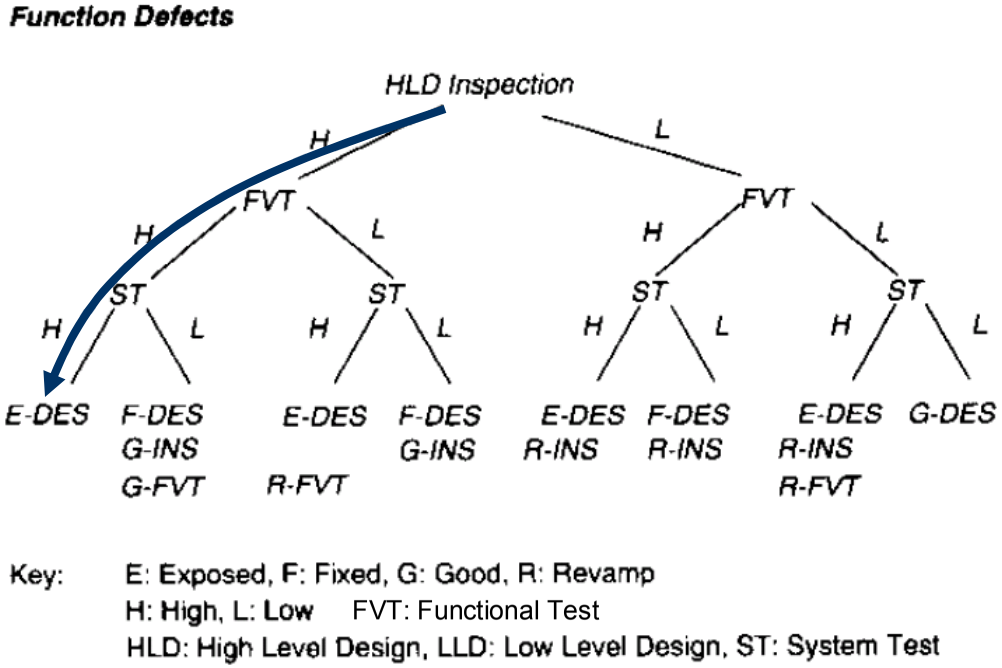




- No errors => good work

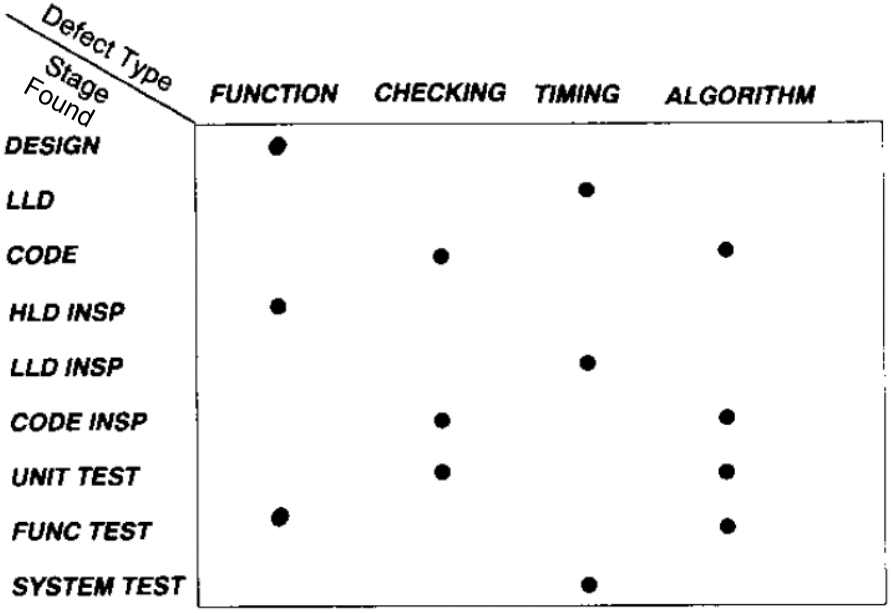
Process Inferences

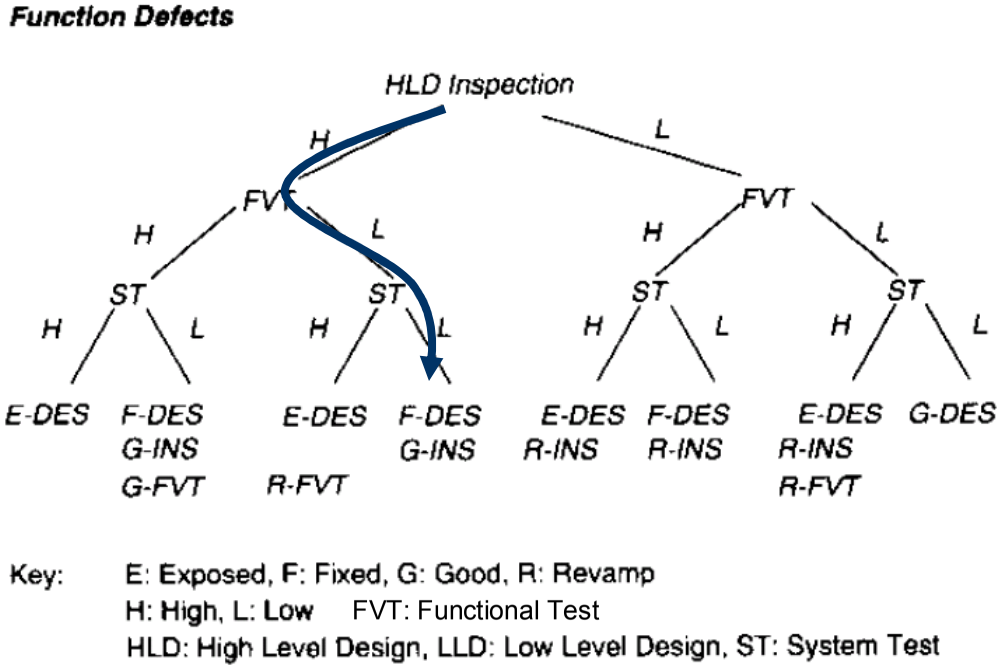




- No errors => good work
- High errors throughout => trouble

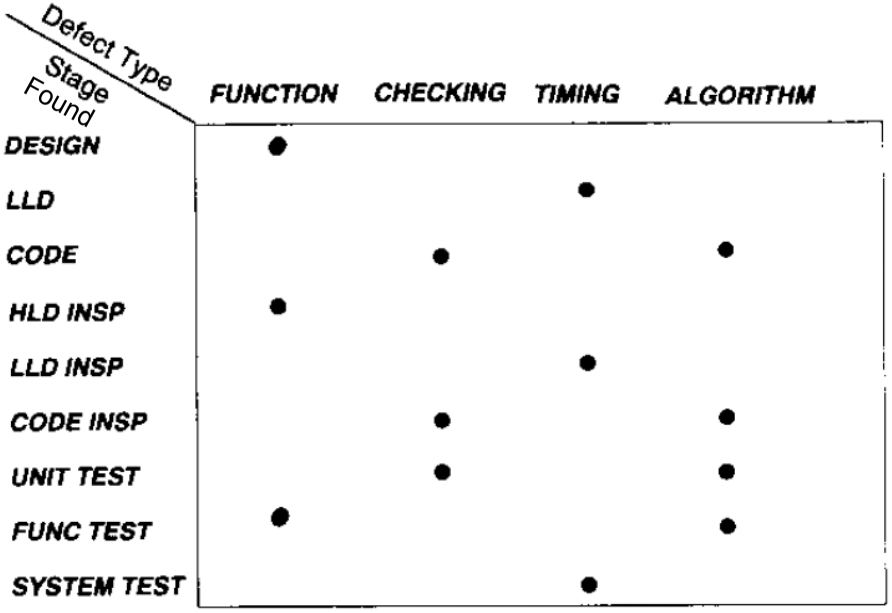
Process Inferences

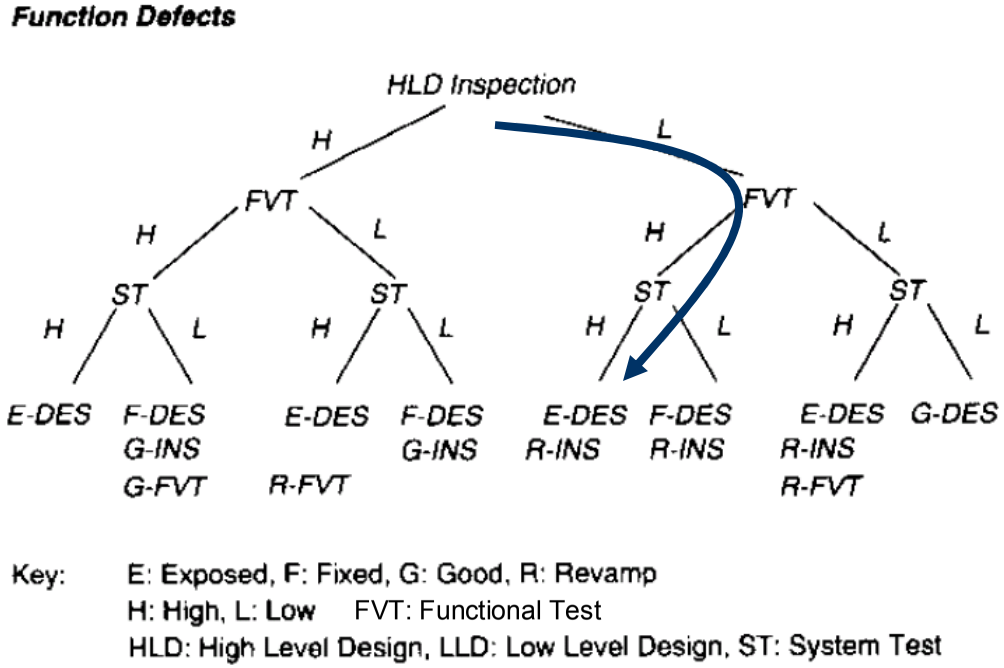




- No errors => good work
- High errors throughout => trouble
- High errors then low => fixed w/ good stage

Process Inferences





- No errors => good work
- High errors throughout => trouble
- High errors then low => fixed w/ good stage
- Low then high => revamp stage

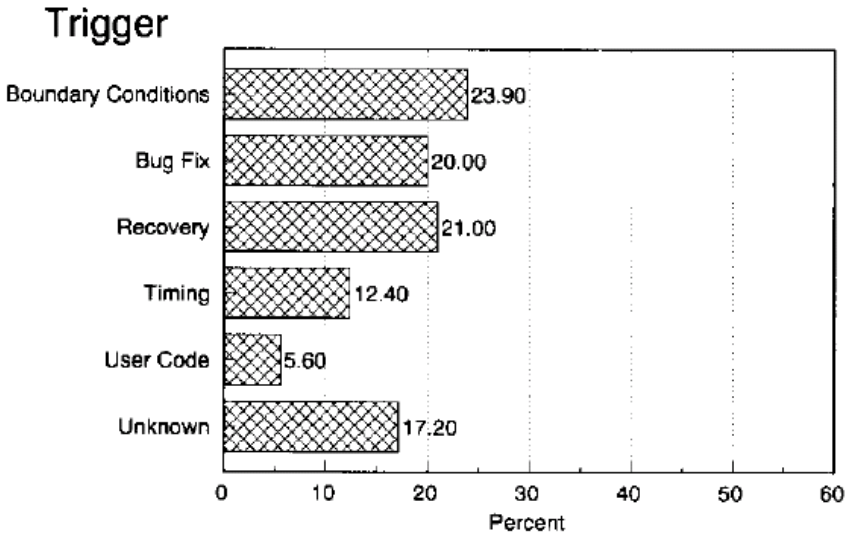
Defect Triggers

- Trigger – a condition that allows a defect to surface

- Examples
  - bug fix
  - boundary conditions
  - exception handling
  - timing
  - workload

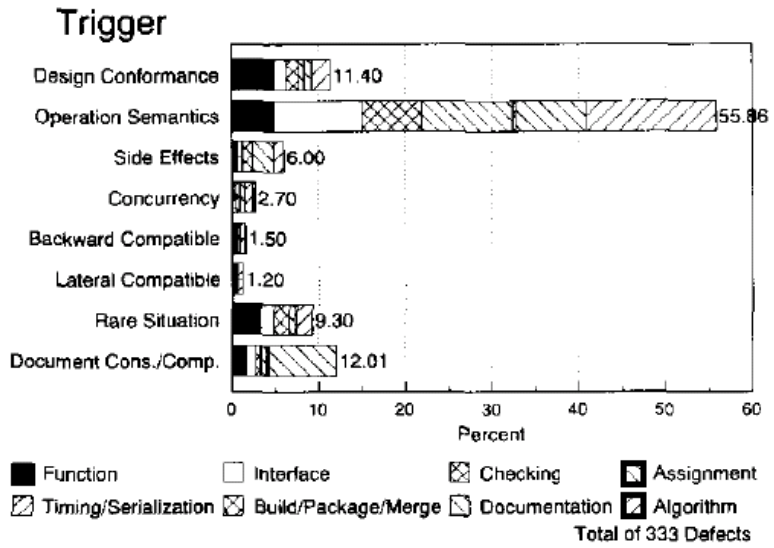
- Why useful?
  - If defects in the field differ from defects found in test, that points out an inadequacy of the test suite

Defect Triggers



- Example from pilot
  - Most defects triggered by boundary conditions
  - Intervention: invest more time in inspections looking at boundary conditions

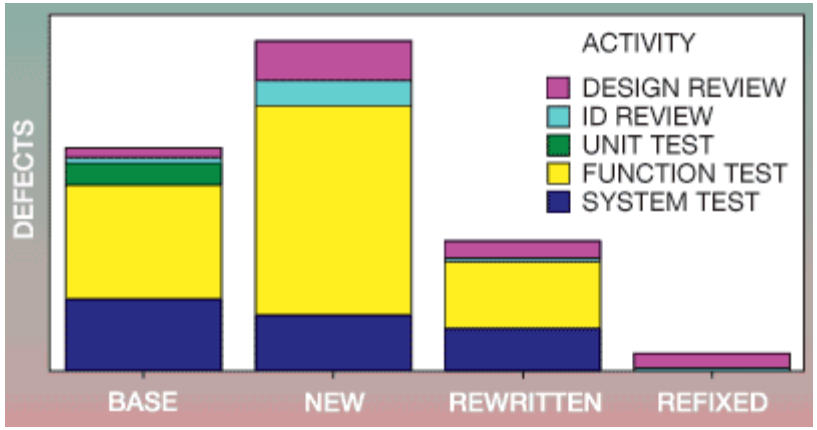
Defect Triggers

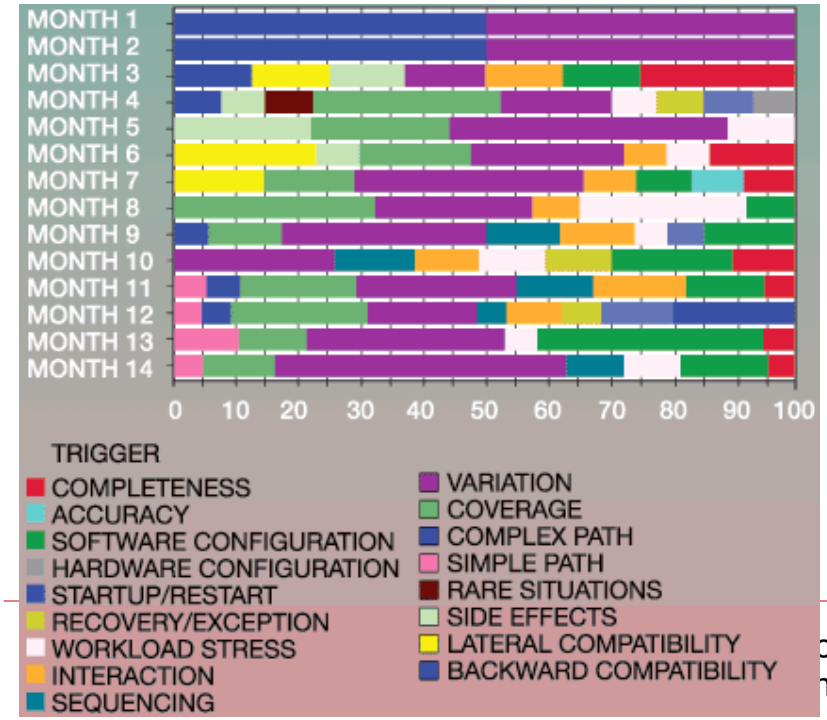


- Another pilot
  - Context: significant interaction with other components
  - Observation: few lateral compatibility triggers
  - Suggests poor review process (or extraordinarily talented team)

ODC: Case Study 1

Source: Butcher, Munro, and Kratschmer. Improving Software Testing via ODC: Three Case Studies. IBM Systems Journal 41(1), 2002.





- Surprisingly many errors in base code
- Majority should have been caught in function test
- Response: more regression testing

- Many field defects had "variation" trigger
  - different inputs for a function
- Response: more variation testing

ODC: Case Study 2 & 3

Source: Butcher, Munro, and Kratschmer. Improving Software Testing via ODC: Three Case Studies. IBM Systems Journal 41(1), 2002.

- Case study 2
  - Good signs led to early entry into system test
    - broad range of triggers & some complex triggers: coverage, variation, sequence, interaction, recovery/exception, startup/restart
  - Field triggers included a lot of software configuration
    - do more configuration testing
  - A lot of "checking" defects from missing code
    - broaden testing with code coverage tool and analyzing test cases for trigger coverage
- Case study 3
  - Assessed a project as not ready for release
    - Fewer than expected defects found in functional testing relative to GUI review
      - functional testing inadequate
    - Simple defects found in functional testing (coverage, variation triggers)
      - more interesting defects not exposed yet!
    - Majority of defects found in old code
      - should have been caught before!

Session Summary

- A principled approach to tracking defects can provide insight into improving process
- Orthogonal Defect Classification
  - Objective defect types used to estimate defect introduction phase
  - Defect trigger distribution used to identify weaknesses in testing
- ODC provided useful feedback in case studies

Questions?

Announcements

- Course information
  - Blackboard: discussion, turn-in
  - Web site: everything else
    - http://www.cs.cmu.edu/~aldrich/courses/654/
- First assignment out in the next couple of days
  - Topic: Specification Inspection, ODC, and Java
  - Mostly a group assignment
  - Due Monday, January 19, at 10:30am
- Next page: Policies

Policies

- Time Management
  - Keep track of time spent on each assignment
- Late Work
  - 5 free late days
    - can be used on non-critical path assignments only
  - No other late work except under extraordinary circumstances
- Collaboration Policy
  - You may discuss the lectures and assignments with others, and help each other with technical problems
  - Your work must be your own. You may not look at other solutions before doing your own. If you discuss an assignment with others, throw away your notes and work from the beginning yourself.
  - You must cite sources if you use or paraphrase any material
  - If you have any questions, ask the instructor or TAs