Developer Based Pre-emptive Testing

PMI ISSIG 2006
Professional Development Symposium

Dr. Richard Bechtold
Abridge Technology: www.abridge-tech.com

Presentation Overview

- Typical Test Motivation and Objectives
- Typical Test Strategies
- Controversial Test Strategies
- Typical Developer Attitudes about Testing
- Developer Based Pre-emptive Testing Approach
- Strategies for Developer Based Pre-emptive Testing
- Prioritizing Strategies
- Developer Based Pre-emptive Testing Advantages

Tester Psychology

Testing is the process of
comparing the invisible
to the ambiguous
so as to avoid the unthinkable
happening to the anonymous.
– James Bach

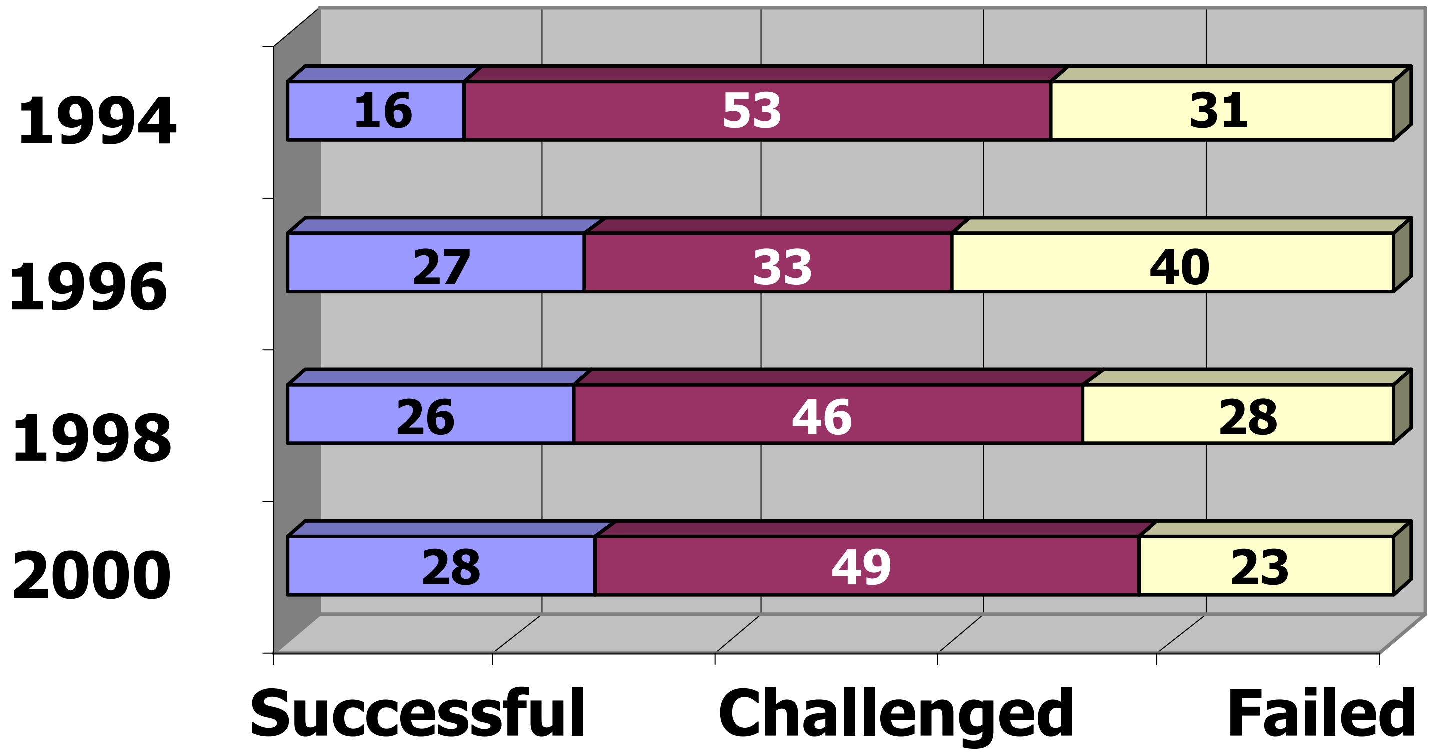
Tester Psychology

Software Testers: Depraved minds, usefully employed.

– Rex Black

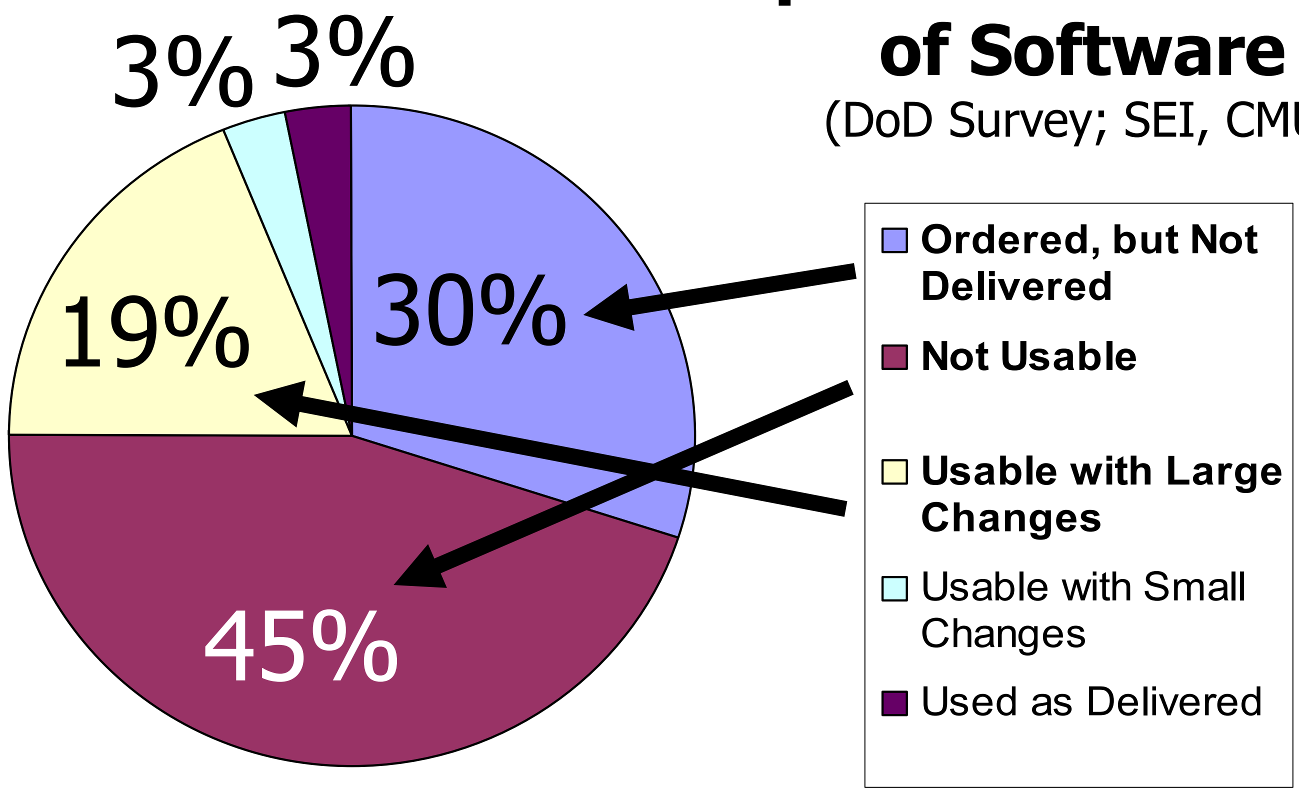
Typical Test Motivation

Standish Group; IT/IS Project Success Rates:



Typical Test Objectives

Requested Functionality of Software
(DoD Survey; SEI, CMU)



rbechtold@abridge-tech.com

Typical Test Strategies

- Black Box vs. Gray Box vs. White Box
- Test Cases, Suites, Scripts, and Scenarios
- Use Case or Role-Based Scenarios
- Alpha, Beta, and Gamma Testing
- Piloting and Limited Releases
  - Gold, Platinum, Titanium
- Manual, Automated, and Hybrid

Typical Test Strategies

- Focus on identifying (ISO 9126)
  - Functionality (Accuracy, Security, etc.)
  - Reliability (Recoverability, Fault Tolerance, etc.)
  - Usability (Learnability, Operability, etc.)
  - Efficiency (Time Behavior, Resource Behavior)
  - Maintainability (Analyzability, Changeability, Testability, etc.)
  - Portability (Installability, Replaceability, etc.)

Controversial Test Strategies

- Agile Testing: Write tests first; expect constant change and uncertainty
- Exploratory Testing: Simultaneous learning, test design, test execution; investigative
- Test Automation: Typically expensive; often highly reusable; strategic investment
- Developer Based Pre-emptive Testing: *Aren't these people too expensive to perform testing?*

Typical Developer Attitudes about Testing

- "Testing is boring."
- "An author cannot proof-read their own work, or test their own software."
- "Testing is often where people start their careers, and I've moved beyond that."
- "Testing should only be done by test specialists, and I only specialize in design/development."
- "We're too expensive to perform testing!"

Developer Based Pre-emptive Testing (DBPT) Approach

- With DBPT, test time is deliberately planned (e.g., 4 hours this Friday, covering the following strategies…)
- Developers do not immediately fix what they find; instead they simply log, fix later
- Testing can be deliberately designed (by the developers) to be more interesting

DBPT Approach

- As will be seen when discussing strategies, developers *are* the proper specialists for this type of testing
- With DBPT, developers continue to perform their usual sub-unit testing (typically, numerous times throughout each day)
- DBPT planning should be a minimum of 10% to 15% of DBPT time (i.e., if 4 hours/week set aside for DBPT, spend the first 30 minutes planning)

Strategies for DBPT

Underlying principle for each of the 8 strategies we are about to examine:

There are things that developers know intimately well about their software, and they can leverage that knowledge when planning and performing testing

Strategies for DBPT

- System Interaction Testing
- Software Architecture Testing
- Boolean Complexity Testing
- Algorithmic Performance Testing
- Exception Detection and Management Testing
- Harness and Diagnostic Log Testing
- High Index of Suspicion Testing
- Diagnostic Software Architectures

System Interaction Testing

- Depending upon your environment, a given system may need to interact with numerous other systems
- Often, systems test well in isolation, then fail during integration test
- Developers usually know exactly how, where, and when their code exchanges data or messages with other systems

System Interaction Testing

- Developers first test interactions with systems directly accessible from the developers’ environment
- Developers can also test system interactions in an alpha test environment
- Focus on integration and interaction with other systems or subsystems undergoing active upgrades or modernization

Software Architecture Testing

- In addition to horizontal interaction with neighboring systems, a developer's code may integrate architecturally as a part of numerous virtual software layers
  - Graphical user interface
  - Data management
  - User authentication, permission management, access and security control

Software Architecture Testing

- Developers specifically focus on testing code where
  - The developer has used, or relied upon, the greatest amount of other in-house or third party software components
  - The developer has made the greatest number of assumptions regarding the other software components' detailed functions and characteristics

Boolean Complexity Testing

- Developers concentrate on testing the most deeply nested Boolean constructs they've created
  - Nested binary logic: if, then, else, switch
  - Nested looping constructs: while, for, repeat, do…until
- Test in a manner that ensures the most deeply nested logic is reached and executed

Boolean Complexity Testing

- Generally, even with only moderately complex logical structures, testing every possible path is infeasible
- This focus of this strategy is not path coverage (although that’s desirable, if achievable in a practical way) but is instead an investigation of the integrity of the logical structure itself

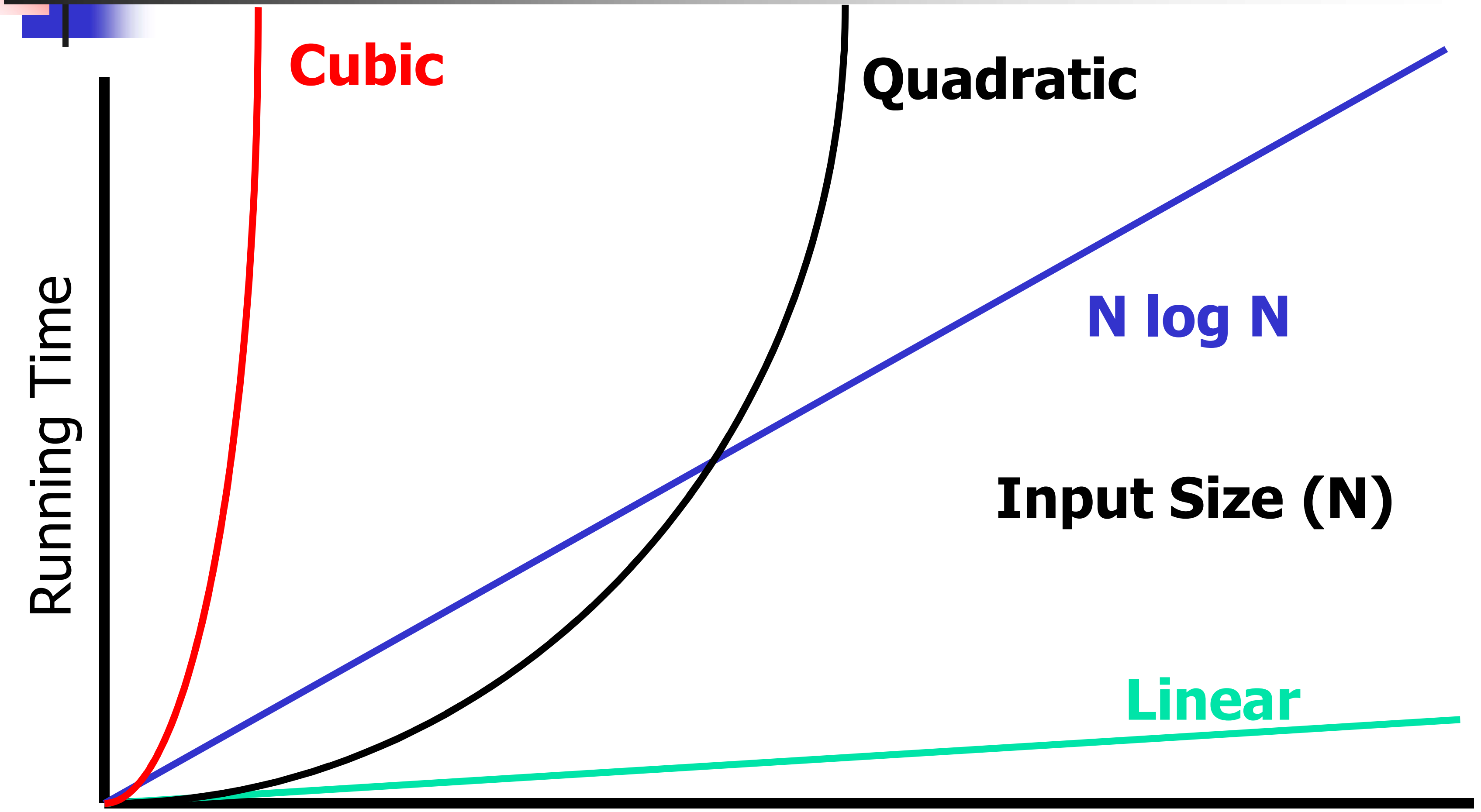
Algorithmic Performance Testing

- Similar to before, the focus is again on looping constructs
- However, now we are particularly interested in execution time *implications* resulting from nested looping constructs
    - Logarithmic time
    - Linear time
    - Quadratic time
    - Cubic time
    - Exponential time

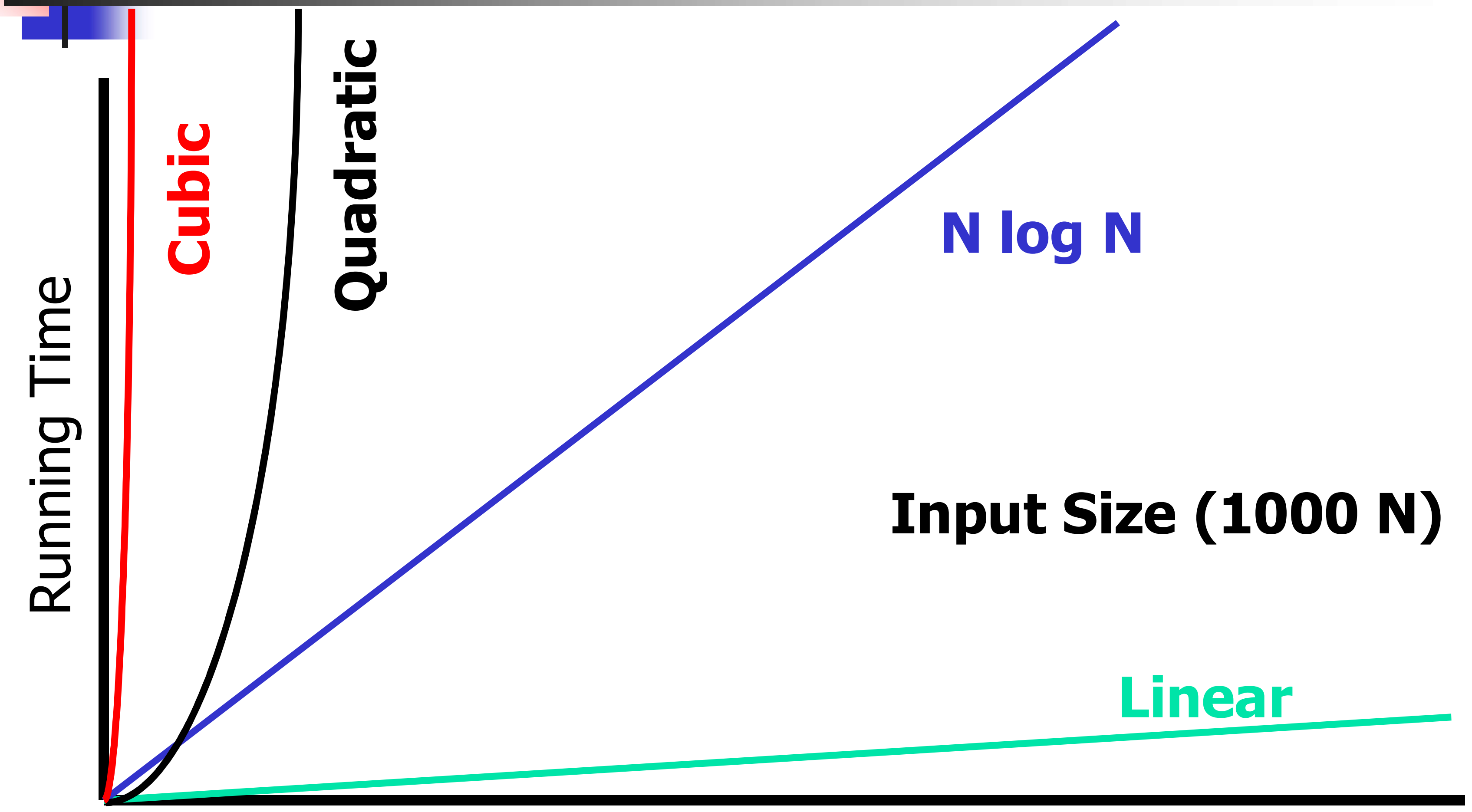
Algorithmic Performance Testing

- Generally, the *dominant term* determines the running time of an algorithm
- Consider: (10*N^3)+N^2+(40*N)+80
- For the value of N = 1000
  - Function returns: 10,001,040,080
  - Of that, 10,000,000,000 is due to (**10*N^3**)
  - The remaining 1,040,080 is less than 0.01 percent of the original total value

Algorithmic Performance Testing



Algorithmic Performance Testing



Algorithmic Performance Testing

n =	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 century
3^n	.059 second	58 minutes	6.5 years	3855 century	2x10^8 century	1.3x10^ 13 cent.

Exception Detection and Management Testing

- Modern programming languages usually offer advanced features for exception detection and management
  - Java: throw, catch, final
- Also, carefully examine the role of non-standard early terminations
  - Java: break, continue, break outermost

Exception Detection and Management Handling

- In some cases, it *is* reasonable to try for full coverage of all exception trapping and escalation logic (at least in a particular code segment)
- Rethink and consider, what are some potential "worst case" situations (e.g., disappearing files, etc.) and how will the software behave?

Harness and Diagnostic Log Testing

- In this strategy, developers write additional separate code to help with their testing process
  - Add minor logic to interior algorithms so that selected inputs, temporary values, and outputs are written to one or more diagnostic files
  - Write a small utility that examines the diagnostic files for correctness

Harness and Diagnostic Log Testing

- Similarly, small utility algorithms can be written that produce huge sets of simulated input data
- Other utilities can capture output data from standard interfaces, and check correctness
- Although expensive at first, harness and diagnostic utilities are normally highly reusable, and excellent for regression testing

High Index of Suspicion Testing

- As a general rule, developers are well aware of where the logic is that scares them the most, for example
  - By type of logic (e.g., pointer manipulation)
  - By type of function (e.g., any of the routines performing Federal, State, and County tax calculations)
  - By when created (any code originally developed for version 3.2 or earlier)

High Index of Suspicion Testing

- When—not if—the help desk starts receiving calls about defects, what are the most likely locations in your source?
- Also, what are the most likely types of potential problems?
  - Logical
  - Mathematical
  - File I/O
  - Error handling
  - Data corruption

Diagnostic Software Architectures

- Although almost no one does it, systems can be *designed* to support diagnostics
- Premise: you want to reduce the time of
  - Detections
  - Corrections
  - Recovery
- Your systems/software should *never* keep secrets from you

Diagnostic Software Architectures

- As with automobiles, the ideal is to design your software so that a variety of very fine-grained detection capabilities exist
- Diagnostic capabilities can be activated and deactivated by type, depth of investigation, location, etc. (or any combination)
- Performance penalties can be temporarily acceptable
- Typically, fixing a defect is relatively fast and easy—the vast majority of *rework* time and cost actually occur trying to find the defect location

Prioritizing Strategies

- Generally, allow developers to decide how much time to set aside for
  - DBPT planning
  - DBPT performance
- Allow the developer to select the primary strategy, or combination of strategies, he or she intends to use
- Strategies should generally reflect the developer's strengths and insights (*and unstated concerns*)

Prioritizing Strategies

- Ask for *very simple* record keeping
  - What was found
  - Where was it found
  - Time spent detecting (relative to plan)
- Use the resulting data, and data from the independent test group, to recommend adjustments to future DBPT planning and performance

DBPT Advantages

- DBPT generally avoids duplicating testing performed by the independent test group (e.g., role-based testing)
- Facilitates discovery of defects that might otherwise be very hard to find prior to release
- Augments, but does not replace, standard day to day sub-unit testing

DBPT Advantages

- Defects are discovered and fixed when the overall code is clearest in the developer's mind (not weeks after testing, *or months after release*)
- Can potentially result in substantially reduced time and cost in
  - Development
  - Rework
  - Testing
  - Customer support
  - Defect recovery actions

Summary and Conclusions

- In principle, no testing should be necessary—developers should write perfect software
- In practice, an independent test group is often essential for achieving pre-release quality objectives
- However, test time, and test budgets, are often used as buffers for project over-runs

Summary and Conclusions

- Engaging developers in *some* level of systematic and planned investigations into anomalous software behavior can save significant time and costs
- As with anything, the more developers practice DBPT, the more adept they'll become at the various strategies
- Plot trends in defect discovery to confirm that DBPT is successful in *your* context

Contact Information

Dr. Richard Bechtold
President; Senior Consultant
Abridge Technology; Ashburn VA
703.729.6085
rbechtold@abridge-tech.com
rbechtold@rbechtold.com
www.abridge-tech.com

Biographical Highlights

Dr. Bechtold is a senior consultant for Abridge Technology, a Virginia-based company he founded in 1996. Abridge Technology is an SEI Partner and is authorized to provide licensed training and appraisal services. Dr. Bechtold provides consulting, training, and support services in the areas of project management, process improvement, process definition, measurement, and risk management. Dr. Bechtold has assisted government and industry with implementing the Software CMM since 1992, the Acquisition CMM since 1996, and the CMMI since 2000. Dr. Bechtold's expertise spans organizations of all types and sizes, from multi-billion dollar companies and agencies, to companies with less than 20 personnel.