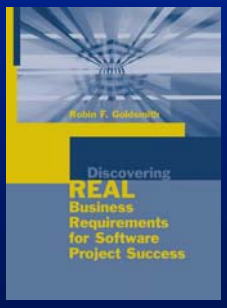
Proactive Testing™:
Puts Agile Test-Driven (and Other) Development on Steroids



Robin F. Goldsmith, JD
GO PRO MANAGEMENT, INC.
SYSTEM ACQUISITION & DEVELOPMENT
QUALITY/TESTING
PRODUCTIVITY
BUSINESS ENGINEERING
TRAINING
22 CYNTHIA ROAD
NEEDHAM, MA 02494-1412
INFO@GOPROMANAGEMENT.COM
WWW.GOPROMANAGEMENT.COM
(781) 444-5753 *VOICE/FAX*

Objectives

- Describe the strengths and (often unrecognized) limitations of typical test-first development
- Explain key truly agile concepts of Proactive Testing™ that further enable quicker, cheaper, *and* better software development
- Show a variety of Proactive Testing™ techniques which reveal numerous otherwise overlooked test conditions which then can be addressed selectively based on risk

Traditional Testing Tends to Be Reactive and Relatively Ineffective/Inefficient

- Tends to come late when defects are hardest and most expensive to find/fix
- Developers make more errors that they realize and find/fix fewer than they believe
- Testers lack sufficient knowledge or time to test as thoroughly as needed

Agile eXtreme Programming Includes Important Test-Driven Techniques

- Pair programming provides repeated reviews of code
- Test-first development assures code is unit tested, and regression tested, automatically
- Involved user defines automated "acceptance tests" of somewhat larger business functions/integrations

Planning , though not specifically test planning, actually is part of XP too!

Test-First Development Surely Beats Traditional Test-Last (or Never) Coding

- Developer(s) decide how to test that code works and write code in the program being developed (Software Under Test—SUT) to perform the tests
- Then write program's regular, functional code
- Code works when included tests are passed

Included tests are re-executed for every change

Test-First Development Is Good; but Has Some Seldom-Recognized Limitations

- Programmer/code-centric view can easily miss the bigger, more important issues to test
- Developer's (even the pair's) mindset defining tests is likely to be largely same as for the code
  - Mainly testing what is (going to be) written
  - Won't catch what developer doesn't understand adequately or overlooks
  - Developer still is unlikely to have a testing "break it" mindset or systematic test planning and design methods, so probably overlooks many conditions needing testing
- Agile's fanatical resistance to writing anything other than executable code, including tests, is high-effort with relatively low leverage payback

Plus the religious-like "How dare you question my Agile techniques?"

Coding Is Smallest Source of Errors

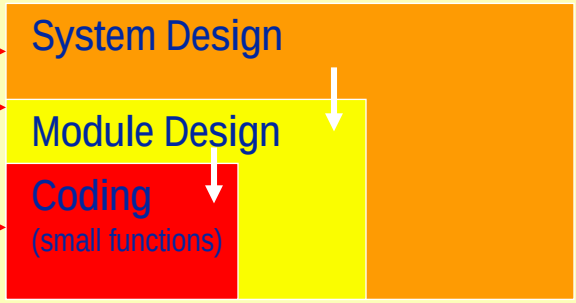
Focusing here →

Coding
(small functions)

Coding Is Smallest Source of Errors

Diverts attention from here →

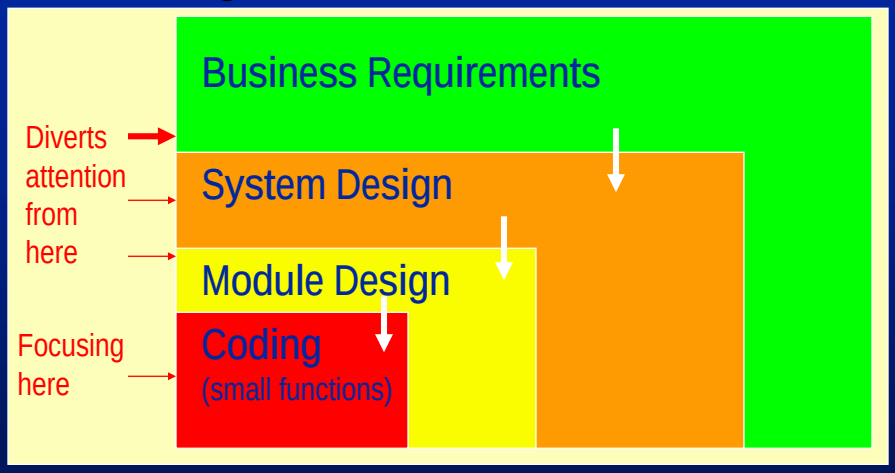
Focusing here →



2/3 of errors in delivered code are in the design.

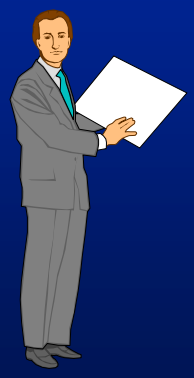
Does essentially having no design increase, decrease, or just mask that?

Coding Is Smallest Source of Errors



Missed/incorrect/unclear business requirements are biggest source of design problems

Don't Active User Involvement, User Stories, and "Acceptance Tests" Eliminate Requirements Issues?



- Just because a user says it, doesn't make it *business* requirements
- Programmer-centric focus increases likelihood that
  - User story "requirements" are really design
  - Acceptance tests actually are after the fact and from perspective of how code is/will be written
- Automated tests are likely to miss some issues that would be apparent to a real user

Two Types of Requirements:

Business/User

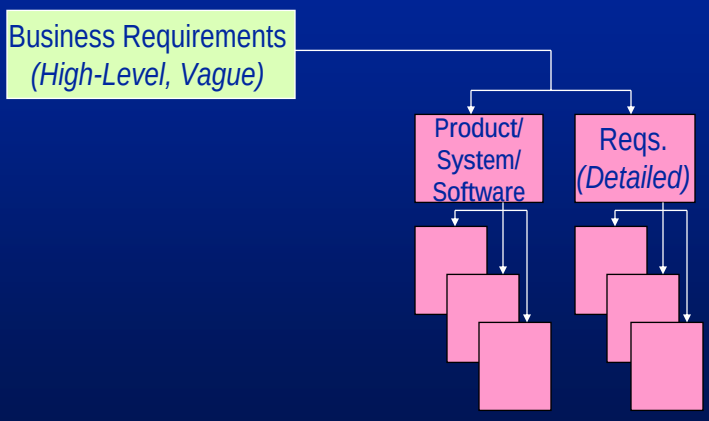
- Business/user language & view, conceptual; *exists* within the business environment
- Serves business objectives
- ***What*** business results must be delivered to solve a business need (problem, opportunity, or challenge) and provide value when delivered/satisfied/met

Many possible ways to accomplish

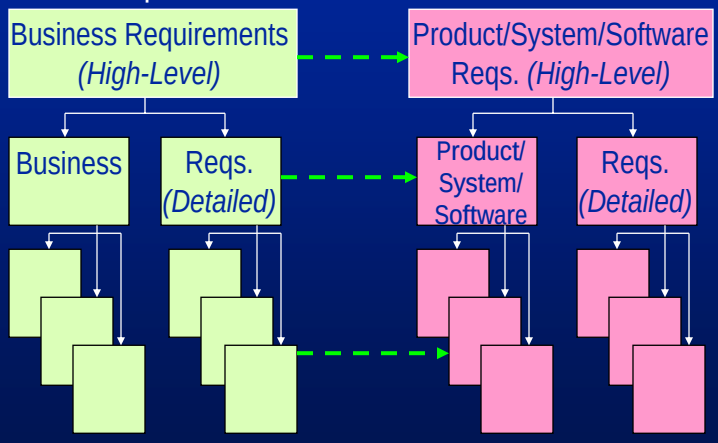
Product/System/Software

- Language & view of a *human-defined product/system*
- **One of the possible ways *How*** (design) presumably to accomplish the presumed business requirements
- Often phrased in terms of external functions each piece of the product/system must perform to work as designed (Functional Specifications)

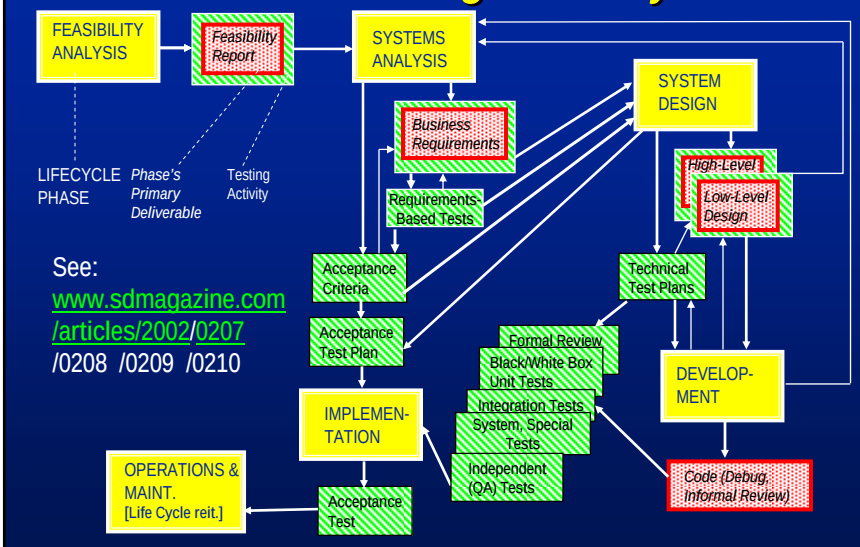
Even Requirements "Experts" Think the Difference is Detail



When Business/User Requirements Are Detailed First, Creep Is Reduced



Proactive Testing™ Life Cycle



See:
www.sdmagazine.com
/articles/2002/0207
/0208 /0209 /0210

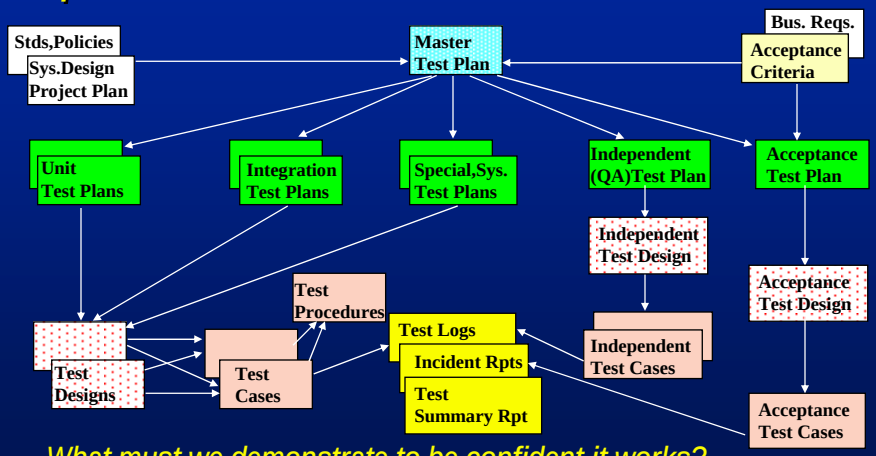
Key Proactive Testing™ Concepts 1 of 2

- Develop iteratively—whatever size piece is coded should be designed and responsive to adequately defined REAL, *business* requirements, which in turn both should be tested
- Define more complete true user acceptance tests proactively at start, *keep independent of design*
- Use Proactive Testing™, in conjunction with more effective discovery and specification, techniques to improve the accuracy, completeness, and clarity/testability of the requirements and design
  - Write enough to help—but no more, and no less
  - Catch big-picture issues, keep refocusing based on risk

Key Proactive Testing™ Concepts 2 of 2

- Let higher-level (than just code) testing planning/design thought processes drive development to
  - Economically anticipate and avert larger consequences of design issues that ordinarily cause rework
  - Plan for coding/testing early to avoid biggest rework risks as well as implementing immediately useful functionality
  - Increase awareness of more of the frequently-overlooked conditions that code/tests must address
- Plan/design tests early, prioritize, promote reuse
  - Concisely define, detect issues top-down at varying levels
  - Create and apply reusable test designs and test cases
  - Implement selectively based on risk

Testware--Test (Plan) Documentation per ANSI/IEEE Std. 829-1998



What must we demonstrate to be confident it works?

Testing Structure's Advantages: Organize Thoughts, Not Pump Paper

- ✓ *True Agility: Write no more than is helpful, but no less*
- ✓ Break big things into manageable pieces (but within the overall context, not just isolated detail)
- ✓ Manage and ease re-creating large set of test cases
- ✓ Show the choices for meaningful prioritization
  - ✓ Focus first on larger issues, drill down later to detail
  - ✓ Successively spot overlooked conditions to test
  - ✓ Test the biggest risks more thoroughly *and earlier*
- ✓ Enable identification of reusable Test Design Specifications and Test Case Specifications

❶ Master Test Plan

- States positively how system will be tested
  - Defines detailed test plans which taken together demonstrate that full system works
  - Sets test priorities and strategy to address risk
  - Establishes defaults, e.g., entry and exit criteria

United States

- Project plan for the testing (sub)project, becomes part of project plan
- Management agreement between customer and technical executives, understandable to both

❷ Detailed Test Plan

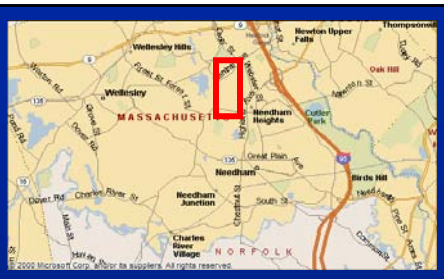
- States positively how piece will be tested
  - Defines set of features, functions, and capabilities (can be a Test Design Specification for each) which taken together demonstrate that it works
  - Identifies exceptions to Master Test Plan defaults
  - Sequences, data sources

Massachusetts

- One per unit, integration, special, system, independent QA, and user acceptance test
- Technical document
- Basis for detailed workplan and estimates

❸ Test Design Specification

- States positively how feature etc. will be tested
  - Defines conditions that must be demonstrated to be confident it works
  - Identifies set of Test Cases that taken together demonstrate conditions
  - May define procedures
  - Can be formal or informal



Needham

- One per feature, function, and capability—can be consolidated for economy and practicality
- Valuable intermediate level, often overlooked
- Potential for reuse

❹ Test Case Specification

- States positively how test will be executed
  - Identifies input/condition (environment), expected results, and (preferably separate) procedure
  - Includes Specification (in words) and Data Values (preferably separate)
  - Common formats are script (especially when automated) and matrix

Cynthia Road

- Lowest-level, can be simple (one input, one result) or complex (series of inputs-results); black box or white box
- Actually executed
- Potential for reuse

We Need to Plan/Design *Tests—Identify the Needed Set of Test Cases, Because*

- Focusing mainly on test cases detail often misses many more conditions than we tend to be aware of
  - We won't find the errors in the things we don't know need to be tested
  - Emphasizing detail first tends to obscure awareness of other, often bigger issues
  - Such areas are more likely to have errors because development-view overlooks them too
- Meaningful prioritization requires *comparing* choices--we've first got to know as fully as possible what the choices are
- Choices depend on risks of each usage

Scale test thinking at a variety of levels, biggest value is at higher levels

Example: Proactive Testing™ Master Test Planning Risk Analysis for *Web Quote Personal Auto Insurance*

(as described in Software Testing class by attendees)

- For use by independent agents
- 1. Ascertain who client is, kind of cars, drivers, driving records, location, marital, sex, age, VIN, usage, driver training, grades, types of coverage, deductibles.
- 2. If passes initial scrutiny, find out about liens on the vehicle, additional insured, billing plan, payment type.
- Calculate and provide premium quote.
- Print application form to be signed, returned with payment.

Risks to the System in Operation that Testing Should Address

[*identified by author*]
1 car
1 driver
More cars than drivers
More drivers than cars
Age groups
Accidents and tickets

Risks to the System in Operation that Testing Should Address

[*identified by author*]
1 car
1 driver
More cars than drivers
More drivers than cars
Age groups
Accidents and tickets
Order Motor Vehicle Record
Rates
Agents' use
Flow to in-house system

Risks to the System in Operation that Testing Should Address

[*identified by author*]
1 car, 1 driver
More cars than drivers; more drivers than cars
Age groups; accidents and tickets

Order Motor Vehicle Record Rates Agents' use
Flow to in-house system

[*added by others*]
Data validation and editing
Lose connections, session continuity
Hardware capacity and performance
Compatibility—browser, O/S, platform Viruses

Risks to the System in Operation that Testing Should Address

[*identified by author*]
1 car, 1 driver; more cars than drivers; more drivers than cars
Age groups; accidents and tickets
Order Motor Vehicle Record Rates Agents' use Flow to in-house system
[*added by others*] Data validation and editing Lose connections, session continuity
Hardware capacity and performance Compatibility—browser, O/S, platform Viruses

[from author & others]
Printing quotes and app forms Underwriting rules
Send in signed printed application;
check accompanies if paying by check
Track applications, tie back to ones not sent in
Minimum set of data Calculations
Security Firewalls, anti virus
Order credit scores, receive back for calculations
Validating payment with credit card, not approved
3rd party system down

New customers,
Existing custs, mult D/B records
Multiple requests for quotes
Reports on types of quotes,
quotes vs. purchases
multiple quotes for same person
Compare web applications to
phone, mail applications
Purging records
Interactions with other systems

These represent a combination of Detailed (unit, integration, or special) Test Plans and Test Design Specifications—Also Identify Design Issues

How Many Ordinarily Would Be Overlooked? What's the Impact?

- Are these things developers are likely to think to code, let alone *unit* test?
- With a focus on small code pieces, how many would be overlooked by "acceptance tests" too?
- If they go wrong in the delivered system
  - How many would be showstoppers?
  - How much redesign and rework

Would Agile developers benefit from economically discovering these issues early?

Is this the way testing typically works? *Reactive or Proactive?*

Functionality Matrix: Systematically Identify Parts of Unit Needing Tests

User view, step-by-step (Use Case)
Technical view, what's happening "under the covers"

- Physical I-O, **Create**, **Retrieve**, **Update**, **Delete**
- **Communicate** with an external device
- **Interface** to another piece of software
- Perform **logic** or calculations
- **Change state**
- Meet a specified **performance** level
- Comply with an external **constraint**

Each user/technical view intersection should be addressed in a Test Design Specification (can split or consolidate)

Functionality Matrix

Technical View

User View (Use Cases)	Create	Retrieve	Update	Delete	Commun.	Interface	Logic	ChgState	PerfLevel	Constraint

Functionality Matrix

Technical View

User View (Use Cases)	Create	Retrieve	Update	Delete	Commun.	Interface	Logic	ChgState	PerfLevel	Constraint
Find applicant by driver's license		X			X				X	
Found and confirmed					X	X		X		
Found but not confirmed					X	X		X		
Not found					X	X		X		
Search for applicant by name		X			X		X		X	
Search for applicant by address		X			X		X		X	
Select applicant from search list *					X	X		X		
Quit the search *					X	X		X		
Add applicant to database	X				X	X	X	X	X	X
Quit					X	X		X		

How Many Ordinarily Would Be Overlooked? What's the Impact?

Detailed Test Planning, such as with the Functionality Matrix, deals with smaller pieces. Use Cases are a common format for specifying functionality, with the by-product of seemingly translating fairly readily into tests. XP's "User Stories" often are similar to use cases.

Would Agile developers benefit from economically discovering these issues early?

- Would any of the use case steps be likely to be overlooked? Which ones?
- Are any of the technical view issues things developers are likely not to think to code, let alone *unit* test?
- How many would be overlooked by "acceptance tests" too?

Proactive?

Test Design: What Must We Demonstrate to Be Confident "Find an applicant by driver's license" Works?

Assumptions: License number is fixed-length number

Valid
Actual number for my state
Actual number for a different state

Invalid
Invalid length, too long, too short
Number of proper length for my state, not a license
Number of proper length for a different state, not a license
Valid number for my state but indicated for a different state where not a license
No state, invalid state
Alphabetic, special characters

✓ Checklists of Added Conditions: Data Field Formats & Contents

- Field length and type, inputs and outputs
  - Alpha vs. numeric; mixed data types; case
  - Font, pointsize, color, visibility; focus
  - Special characters; packed and binary; control keys; control characters (special meaning to O/S)
  - Initialization; nulls; defaults; repetition; editing
  - Dates, formats and Julian/Gregorian
- Calculations and algorithms, zero, negative, integers, intermediate results, leading zeros, justification

Valid
Fonts, sizes, color/bkgrnd
Field initially empty, filled
Edit input
Repeat with same, diff no.
Invalid
Mixed alpha and numeric
Blank, null
Zeros, leading blanks
*. , $ + - / * % ( ) [ ]*
Other special characters
Control keys, characters

✓ Application Functionality: Data and Process Models

- Demonstrate each output and input
  - Displayed, transmitted, printed, stored, passed, error messages and indicators; human/machine readable
  - Data modification (and add, delete), loading, and reloading
  - Number of tables, elements, records, device types/locations
- Navigate all routes usage is likely to take
- Environments--multiple concurrent users, browsers, O/Ss, access constraints, degradation, geography

Valid
Newly added, modified number
Reloaded file/DB
Key in, paste in, scan in
To field: Tab, back tab, prior, next Arrows, link, Enter, automatic
Data on user's hard drive, CD server, Web
Single, multiple users
Invalid
Clicks outside indicated fields
Double clicks on fields
Paste in graphic
Deleted number
No access (security)
DB, network error

✓ Boundary Testing--The Single Most Likely Way to Detect Errors

Within Equivalence Class:

- Accept
  - Normally occurring value
  - Minimum, optionally plus one
  - Maximum, optionally minus one
- Reject
  - Minimum minus one
  - Maximum plus one
  - Optionally, extremes high/low

Valid

Lowest number in DB
Next higher number in DB
Highest number in DB
Next lower number in DB
Proper number for state with most digits
Proper number for state with least digits
Proper number for first state in DB
Proper number for last state in DB

Invalid

Just lower than lowest number in DB
Just higher than highest number in DB
Very small number (e.g., .000000001)
Very large number
Proper length minus 1, plus 1
Very long number

How Many Ordinarily Would Be Overlooked? What's the Impact?

Test Design Specifications identify at the lowest level the set of Test Cases that taken together would demonstrate the feature, function, or capability works. Having a more complete definition of the set of possible Test Cases allows more accurate prioritization based on risk.

- Without systematically and consciously asking, what percent of the initial brainstormed Test Cases would be likely to be overlooked? What percent of the Test Cases prompted by the checklists would be overlooked?
- Would developers be likely to think to code, let alone *unit* test these overlooked details?
- Would "acceptance tests" overlook them too?

Would Agile developers benefit from economically discovering these issues early? *Reuse? Proactive?*

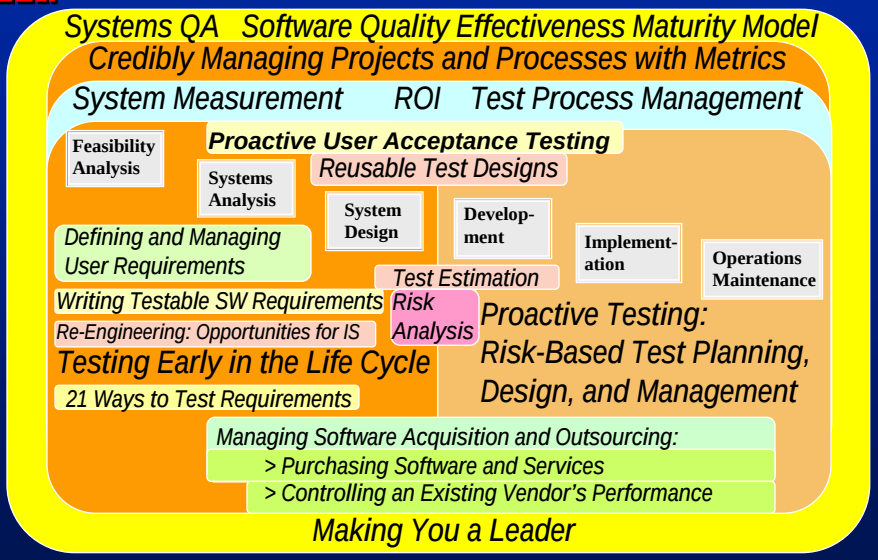
Proactive Testing™ Can Greatly Increase Test-First Effectiveness

- Small amount of effort spots things that ordinarily are overlooked and would take much more effort to fix/provide afterward
  - Bigger parts of the software unit—features, functions, and capabilities (subject of Test Design Specifications)
  - Their conditions that must be demonstrated to be confident they work (Test Cases)
- Structured proactive test planning/design enables reuse of test cases and especially test designs
- Address selectively based on risk

Summary

- Typical Agile test-driven development has advantages compared to traditional test-last (or never) development and reactive testing but also has (often unrecognized) limitations due to its narrow programmer-based focus
- Proactive Testing™ enables truly Agile quicker, cheaper, *and* better software development by feeding low-overhead high-leverage test planning and design information into development throughout the life cycle
- A variety of Proactive Testing™ techniques efficiently reveal numerous otherwise overlooked test conditions *at varying levels* which then can be addressed selectively based on risk and often can be reused

Go Pro Management, Inc. Seminars--Relation to Life Cycle



Robin F. Goldsmith, JD

robin@gopromanagement.com (781) 444-5753

www.gopromanagement.com

- President of Go Pro Management, Inc. consultancy since 1982, working directly with and training professionals in business engineering, requirements analysis, software acquisition, project management, quality and testing.
- Previously a developer, systems programmer/DBA/QA, and project leader with the City of Cleveland, leading financial institutions, and a "Big 5" consulting firm.
- Degrees: Kenyon College, A.B.; Pennsylvania State University, M.S. in Psychology; Suffolk University, J.D.; Boston University, LL.M. in Tax Law.
- Published author and frequent speaker at leading professional conferences.
- Formerly International Vice President of the Association for Systems Management and Executive Editor of the *Journal of Systems Management*.
- Founding Chairman of the New England Center for Organizational Effectiveness.
- Member of the Boston SPIN and SEPG'95 Planning and Program Committees.
- Chair of BOSCON 2000 and 2001, ASQ Boston Section's Annual Quality Conferences.
- Member ASQ Software Division Methods Committee.
- Member IEEE Std. 829 for Software Test Documentation Standard Revision Committee
- Admitted to the Massachusetts Bar and licensed to practice law in Massachusetts.
- Author of book: *Discovering REAL Business Requirements for Software Project Success*