

Test Acceleration

October 27, 2009
Toronto Association of Systems and Software Quality

Kevin Burr
Ottawa Software Quality Association
kevinburr@canada.com
613-253-4257



Who am I?

- Twenty years of software development experience, ten years of management experience in Nortel
 - Performance Engineering - OAM (*Operations, Administration and Maintenance*) Network Engineering Solutions & Services,
 - Manager: Scalability, System Integration, Tools, and Design Support - OAM Framework
 - Product/Design Manager: Application Mgmt - OAM Application Platform
 - Manager: Verification and Infrastructure - Routing Software Framework
 - Manager: [Test Acceleration - Software Engineering Analysis Lab](#)
 - Team Leader: OAM Development - Network Services Framework
 - System Verification - Design Development Management Environment
 - Programmer/Analyst - CAD/CAM Framework
- Currently in the Technology Innovation Management Program at Carleton University

...and Who are You?

- Involved in Integration and up-wards?
- Manual Testing?
- Automated Testing?
- Process Improvement?
- Design & Development?
 - ...sorry. Most everything in talk applies
- Management, high-level types?
- Implementation, low-level types?
 - ...sorry. Talk to me later
- How many attending Testrek?

Problems

- Too many testcases to write
- Take too long to run and verify
- For new builds, which verification tests need to be rerun
- No objective measure of how well the product was tested.
- Cannot depend on help from developers
- Too much to maintain

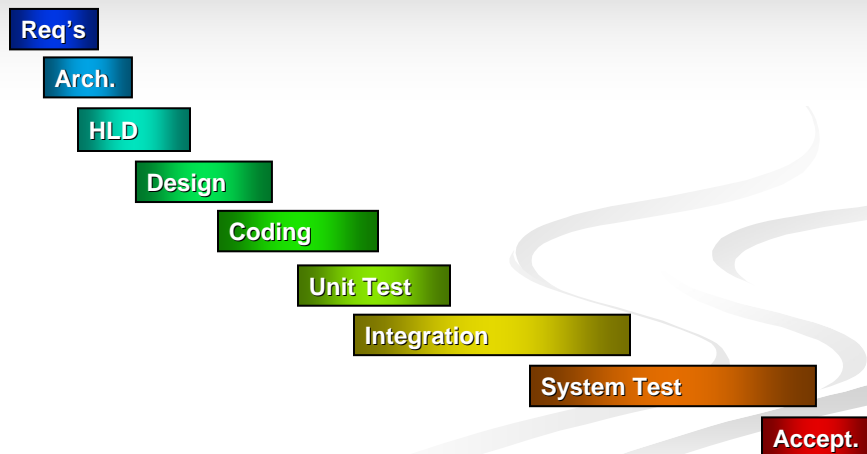
Purpose

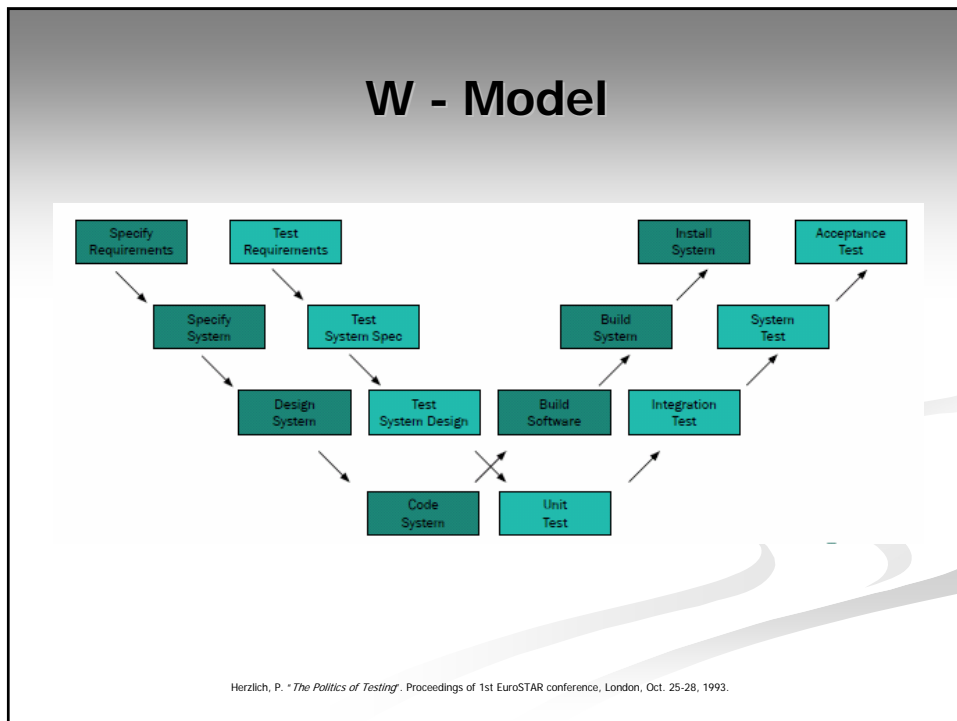
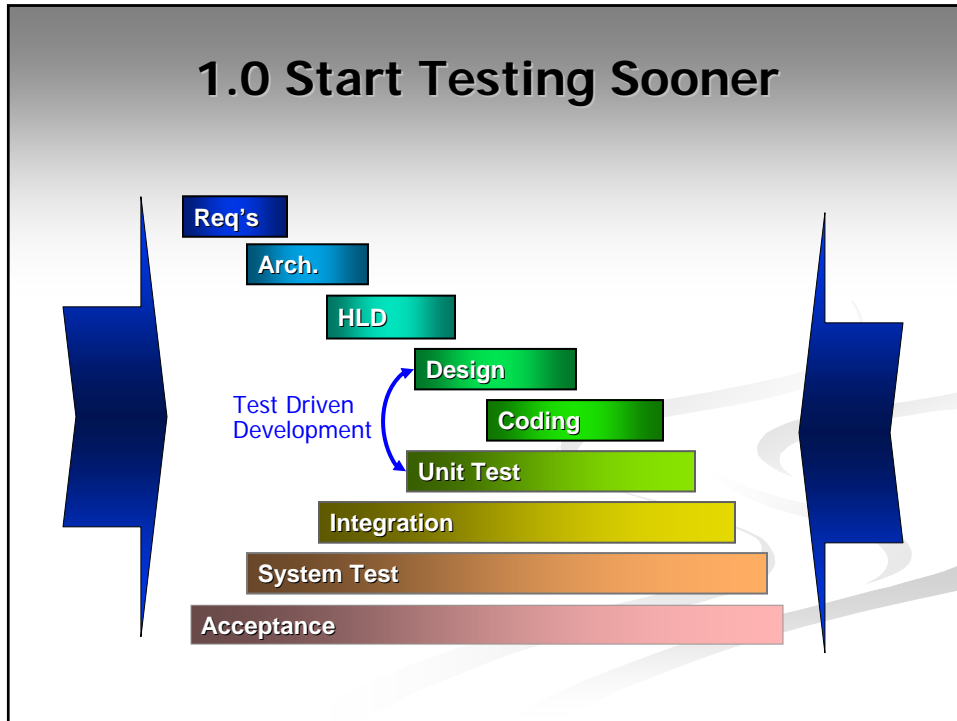
Provide set of techniques to accelerate testing that *I* (and friends) have used in real projects

Agenda

1. Get more time to Test
2. Test Faster
3. Do Less Testing
4. Pulling it off

How do you make testing faster?

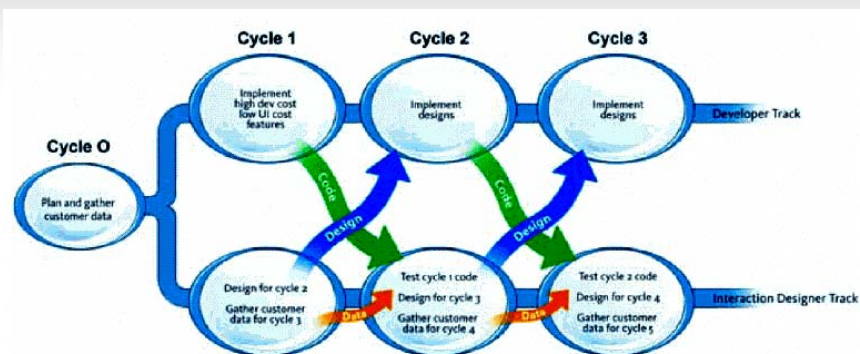




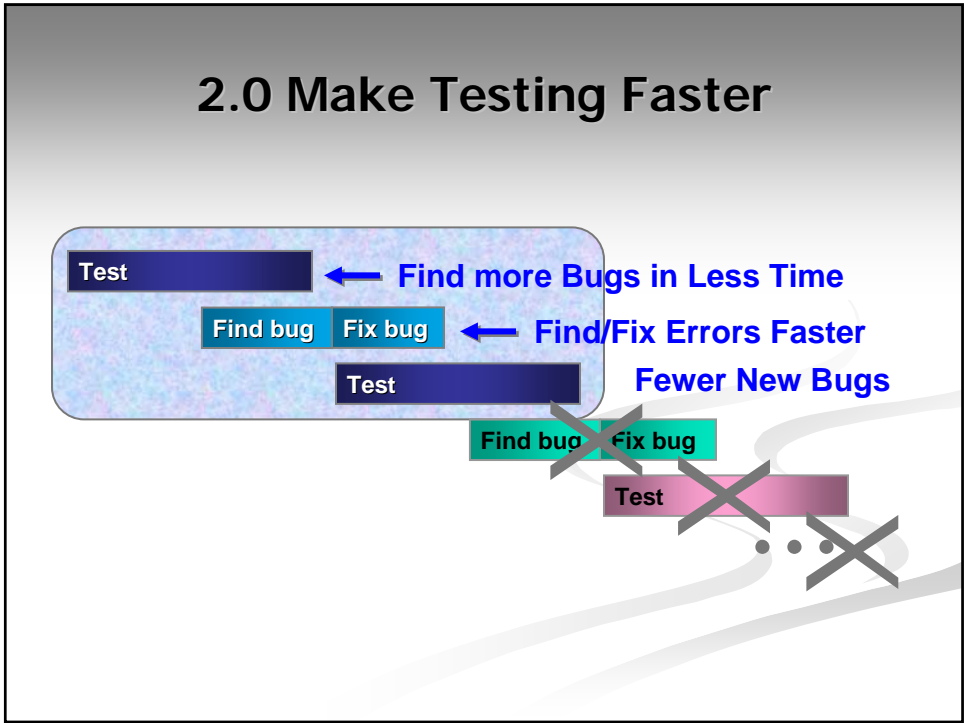
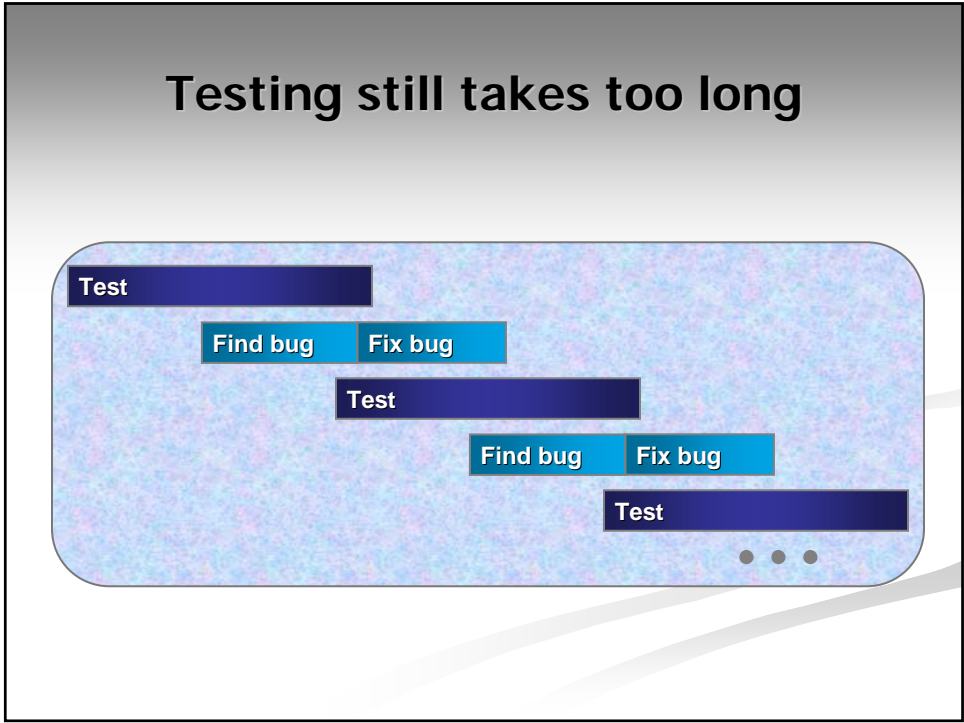
Test Driven Design

- Get Testers involved in Requirement gathering
- Prepare Testing Infrastructure/Architecture along with Design Infrastructure
- Generate Testcases as early as possible
 - Acceptance testcases during Requirement capture
 - ...
 - Unit Level Testcases during coding (i.e. test driven development)
- Start testing as early as possible
 - Verify Requirements before Design (Inspections, Prototyping and Simulation - Executable Requirements)
 - Verify Design before Coding (as above...)

Example: Agile and Usability



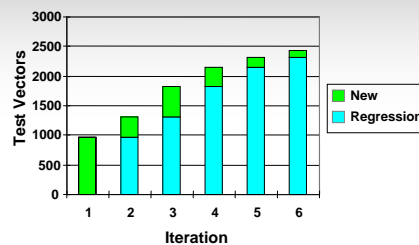
© Lynn Miller, ALIAS Wavefront.
http://wiki.fluidproject.org/download/attachments/1704207/Autodesk_WUD2006_UCDandAgile_lmiller.pdf?version=1



Proven Practices

- **Reduce Testing Time**
 - Faster execution (Testcase automation)
 - Less testcases to write, automate, run, verify and maintain (Testcase generation)
 - Only re-test the parts of the code that changed (Churn)
- **Find bugs faster**
 - Better requirement coverage (Testcase generation)
 - Make sure bugs don't slip by (Code Coverage)
 - How much testing is enough?
 - Only test the code with the important bugs (Risk Analysis)
 - Only test the bugs the customer will find (Operational Profile)
 - Error Simulation
 - Continuous Integration / Nightly Loadbuilds
 - Built in Self Test
- **Fix bugs faster and create fewer new bugs**
 - Understand how the code works (Reverse Engineering)
 - Improve Maintainability

Real Life Example: Unit Level Testing



System under Test:

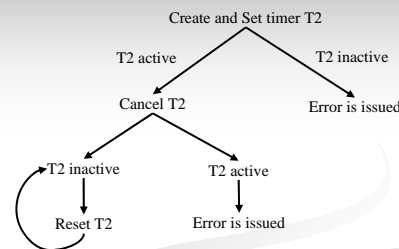
- Unit testing of Controllers and Device Drivers on Real-time Embedded System
 - First iteration was manual
 - Everything after the second was automated
 - Effort is measured in test vectors: a single test input
- Iteration 1 represents the number of manual test vectors that can be run in 2 weeks..
 - By iteration 6 productivity essentially increased 2.5 times !
 - Over 5,000 more test vectors were run than possible manually!
 - This is equivalent to 10 weeks extra testing for free.
 - The tester also able took a 2 week vacation and worked on other assignments, which would have been impossible originally

Types of Test Automation

Approach	Pros	Cons
Record and Playback	<ul style="list-style-type: none"> Fast and easy to create Requires little to no coding experience 	<ul style="list-style-type: none"> Breaks easily Hopeless to maintain
Structured Code	<ul style="list-style-type: none"> More robust Intermediate coding skills 	<ul style="list-style-type: none"> Needs more planning Maintenance is a problem Leads to code duplication Slowest to create (unless using cut and paste)
Shared code	<ul style="list-style-type: none"> Uses shared libraries for common functions (i.e. code re-use) Faster to create, easier to maintain & more robust Supports mix of coding skills 	<ul style="list-style-type: none"> Requires good programmers to support re-useable code.
Data Driven	<ul style="list-style-type: none"> Uses parameters or data files to drive testcases Fastest to create testcases Less code, less maintenance 	<ul style="list-style-type: none"> Requires better coding skills Test scenarios may be complex in order to take advantage of common test driver
Keyword Driven	<ul style="list-style-type: none"> Simple keywords capture actions Requires testers to have least coding experience 	<ul style="list-style-type: none"> Requires experienced programmer to maintain/create parser Keyword "scripts" have to be maintained

2.1 Data Driven Automation

- aka Table driven
- Each test case is 1 line in table
- Test driver automates a use-case
- Use test attributes and expected results to create a data table for test driver
- Behavior is data modeled. Test driver may drive SUT directly or create a test script for each test case
- Test driver may interact with multiple automation tools behind the scenes.
- Test driver can be as complex or simple.



Test Attribute	Possible values	Test values
Value	0,1,2,..., MAX_VALUE	0,1,2,3,20,40,61, 100,900
Granularity	1,2,3,4	1,2,3,4
Type	one shot timer, periodic timer	one shot timer, periodic timer
Create branch	active, inactive	active, inactive
Cancel branch	active, inactive	active, inactive

2.2 Keyword Driven Automation

Window	Action	Object	Value
Main	Click	NextButton	
Main	Click	MenuBar	File
Main	Click	MenuBar	Print...
Print	EnterValue	NumberCopies	2
Print	Select	PrinterName	Adobe PDF
Print	Click	OKButton	

- aka Table Driven
- Single test driver for whole System under Test
- 1 test case per table, 1 action per line
 - e.g. Window, Action, Object, Value
 - Can also validate data
- Depending on your test plan "style" this can become executable documentation
 - Test instructions are the automation
- If you expand concept to simplified scripting languages
 - e.g. Click ("Main", "NextButton")
 - This is the most popular data driven testing approach by far.
 - But not as fast to create as previous

Pair-wise test generation

- Produces a very small (but twisted) set of test cases
- Easily feeds directly into data driven test driver
- Use a tool that uses rules (predicates) to prevent impossible combinations of inputs
 - Modeling around them is too hard

Pair-wise Test Cases: 13 Fields, 3 Inputs, 15 Test cases

TEST	Field1	Field2	Field3	Field4	Field5	Field6	Field7	Field8	Field9	Field10	Field11	Field12	Field13
Case1	1	1	1	1	1	1	1	1	1	1	1	1	1
Case2	1	2	2	2	2	2	2	2	2	2	1	1	1
Case3	1	3	3	3	3	3	3	3	3	3	1	1	1
Case4	2	1	1	2	2	2	3	3	3	1	2	2	1
Case5	2	2	2	3	3	3	1	1	1	2	2	2	1
Case6	2	3	3	1	1	1	2	2	2	3	2	2	1
Case7	3	1	1	3	3	3	2	2	2	1	3	3	1
Case8	3	2	2	1	1	1	3	3	3	2	3	3	1
Case9	3	3	3	2	2	2	1	1	1	3	3	3	1
Case10	1	2	3	1	2	3	1	2	3	1	2	3	2
Case11	2	3	1	2	3	1	2	3	1	2	3	1	2
Case12	3	1	2	3	1	2	3	1	2	3	1	2	2
Case13	1	3	2	1	3	2	1	3	2	1	3	2	3
Case14	2	1	3	2	1	3	2	1	3	2	1	3	3
Case15	3	2	1	3	2	1	3	2	1	3	2	1	3

Maximum possible combinations = 1,594,323

Real Life Example: Application Level Testing

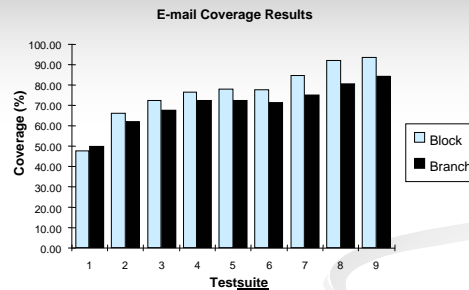
E-mail Coverage Results

Testsuite	Block (%)	Branch (%)
1	65	50
2	68	60
3	70	65
4	72	70
5	75	70
6	78	70
7	80	75
8	90	80
9	95	80

System under Test:

- 36 Fields
- 29 Trillion Exhaustive Testcases
- 72 "defaults" Testcases
75% branch coverage
- 47 Pair-wise Testcases
93% branch coverage
- Translated SMTP email to x400 format
- Very mature system
- Work was done by 1st year co-op
- Testing took 1 week
- Uncovered 8 failures (bugs)

Real Life Example: Application Level Testing



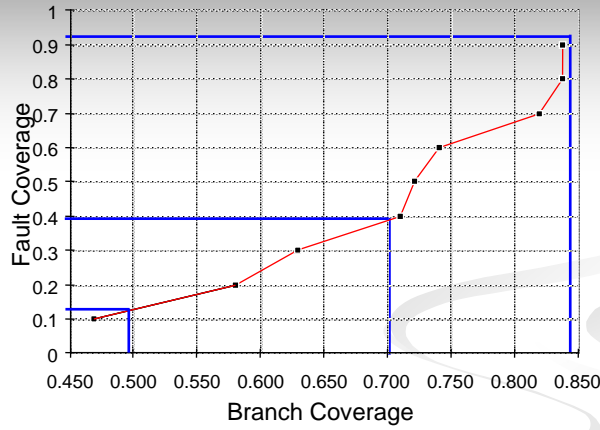
Benefits:

- Fewer testcases: less time to test and verify results
- High level of code coverage
- Incorporates boundary testing
- Both success and failure testcases generated
- Ease of adapting testcase suite to changing requirements
 - We were given the wrong requirements initially and slowly discovered the correct ones

Testing Testing (Feedback Loop)

- How do you know you have tested everything?
 - Undocumented requirements and features
 - Undocumented dependencies
 - Using out of date documentation
 - Missing code
- How do you know you are not wasting time testing some functionality too much?
- Measure what code was not executed (code coverage)
 - Whatever functionality that code was written to implement is missing from the testsuite
 - Does not tell you anything about the code that was executed

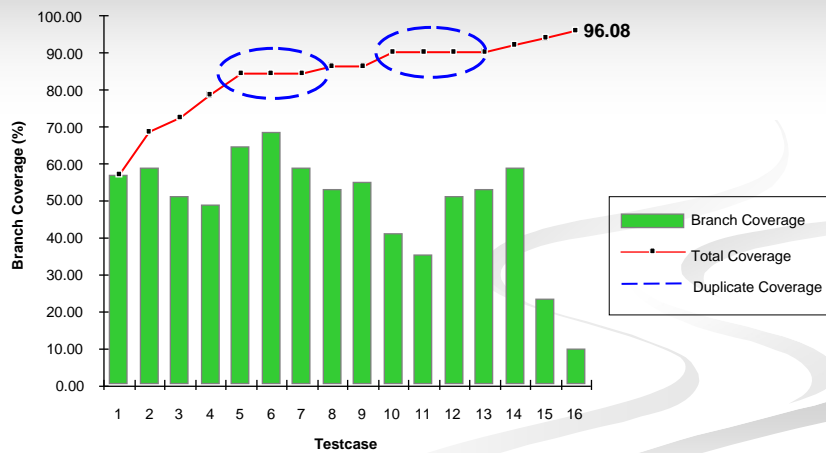
Error vs. Branch Coverage



Without measuring coverage, tests often only exercise 70% of the code, however bugs often hide in the hard to test code, as in this example.

Ref: "The Relationship Between Test Coverage and Reliability" by Yeshwant Malaiya, Naixin Li, Jim Bieman, Rick Karich, Bob Skibbe, Proc. 5th ISSRE, Nov. 1994

Coverage Results - Procedure Test



Redundant code coverage might be wasted testing time

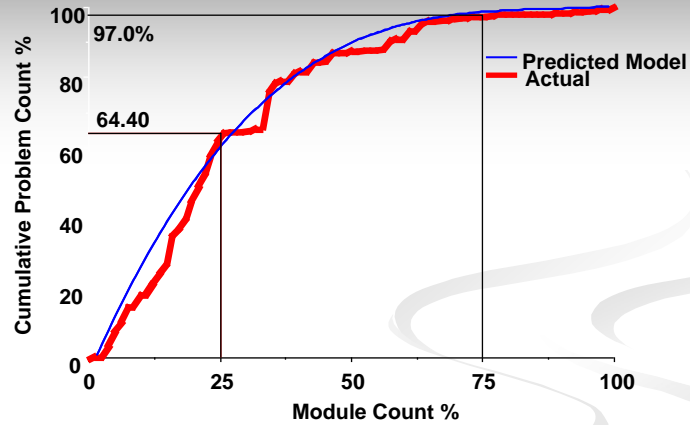
Sanity Subsystem Coverage Code Coverage - not just for Unit Test

Area	V.56			V.87		
	Procedures	Hits	Percent Hit	Procedures	Hits	Percent Hit
	7	0	0	7	0	0
	0	0	0	88	13	15
	810	33	4	895	36	4
	102	26	25	114	38	33
	5545	893	16	7454	1117	15
	1736	304	18	2302	325	14
	570	24	4	606	65	11
	407	2	0	424	32	8
	13	0	0	321	67	21
	46	0	0	46	0	0
TOTAL	9324	1282	14	12257	1693	14

Area	V.56			V.87		
	Subsystems	Hits	Percent Hit	Subsystems	Hits	Percent Hit
	1	0	0	1	0	0
	1	0	0	1	1	100
	1	1	100	1	1	100
	1	1	100	1	1	100
	24	18	75	25	24	96
	4	4	100	4	4	100
	2	1	50	1	1	100
	3	1	33	3	1	33
	1	0	0	1	1	100
	1	0	0	1	0	0
TOTAL	39	26	67	39	34	87

- ### 3.0 Test Less
- Rarely able to test everything
 - Some parts of the code tend to have more bugs than other places
 - New or changed code
 - Complex code
 - Functionality most used by customers.
 - Etc.
 - Use risk metrics to focus testing on the parts of the code most likely to have bugs customers will find.

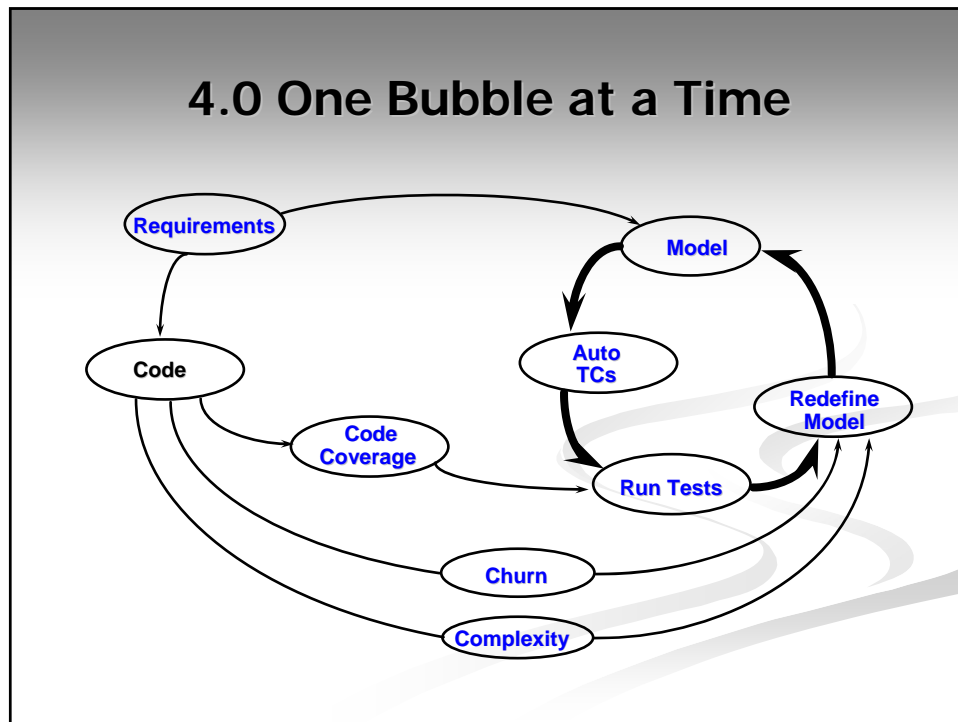
Correlating Complexity with Problems



- 64% of problems found in top 25% complex code
- 3% of problems found in bottom 25% complex code
- Focus testing on highly complex code and avoid spending time on least complex.

Operational Profiles

- Users do not use all the code
 - Test the code that is used the most
 - Test the code that cannot fail
 - Fixing bugs that do not need to be fixed increases the chances of important code breaking
- Rate testcases by impact if they fail, and cut back to management's desired risk level.
- Many examples of code breaking under stress
 - But over engineering means not coding other things
- Measure what users are really doing



Baby steps and Continuous Improvement

- Select a goal and plan how to get there
- Decide how to measure success and measure where you currently are
- Don't try to do too much
- Sell solutions based on needs
- Work with the willing and needy first
- Keep focused on goals and problems
- Align the behaviour of Managers and Practitioners
- Measure and evaluate progress

Conclusion

- Get more time to Test
- Test Faster
- Do Less Testing
- Pulling it off

Thank-you!

- Questions?



References

- Burr, Kevin & Young, William, 1998, *Combinatorial test techniques: Table-based automation, test generation and code coverage*, Proceedings of the Intl. Conf. on Software Testing Analysis and Review, San Diego, CA. October 1998, http://aetgweb.argreenhouse.com/aetg_nortel.pdf
- Burr, Kevin. 1998, *Test Acceleration*, IEEE Ottawa, June 1998
- Christie, James. 2008, *The Seductive and Dangerous V-Model p73 Testing Experience April 2008*, www.testingexperience.com
- <http://www.pairwise.org/>
- Malaiya, Y., Li, N., Bieman, J., Karich, R., & Skibbe, B., 1994, *The Relationship Between Test Coverage and Reliability*, Proc. 5th ISSRE, Nov. 1994