

# ***Test Acceleration***

## ***Part 1.5***

April 22, 2010  
Ottawa Software Quality Association

Kevin Burr  
kevinburr@canada.com  
613-253-4257



## **Who am I?**

- Over 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?

- Attended/viewed talk last year
- Involved in Integration and up-wards?
- Manual Testing?
- Automated Testing?
- Process Improvement?
- Design & Development?
- Management, high-level types?
- Implementation, low-level types?
  - ...sorry. Talk to me later

## 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

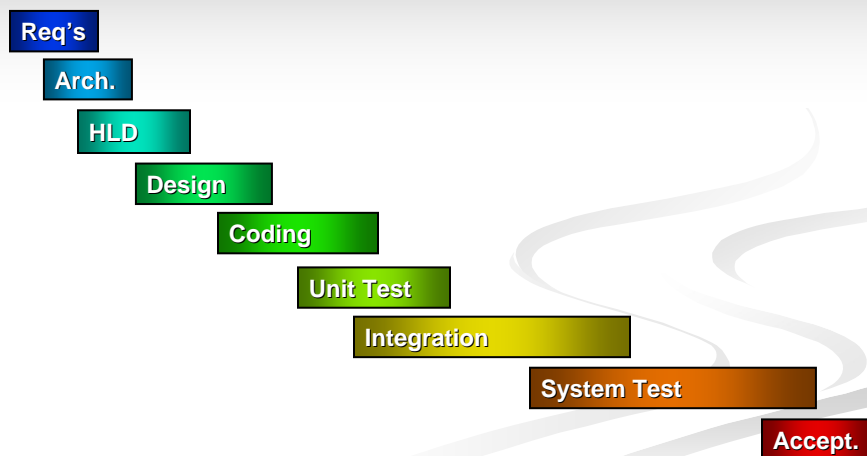
## Purpose

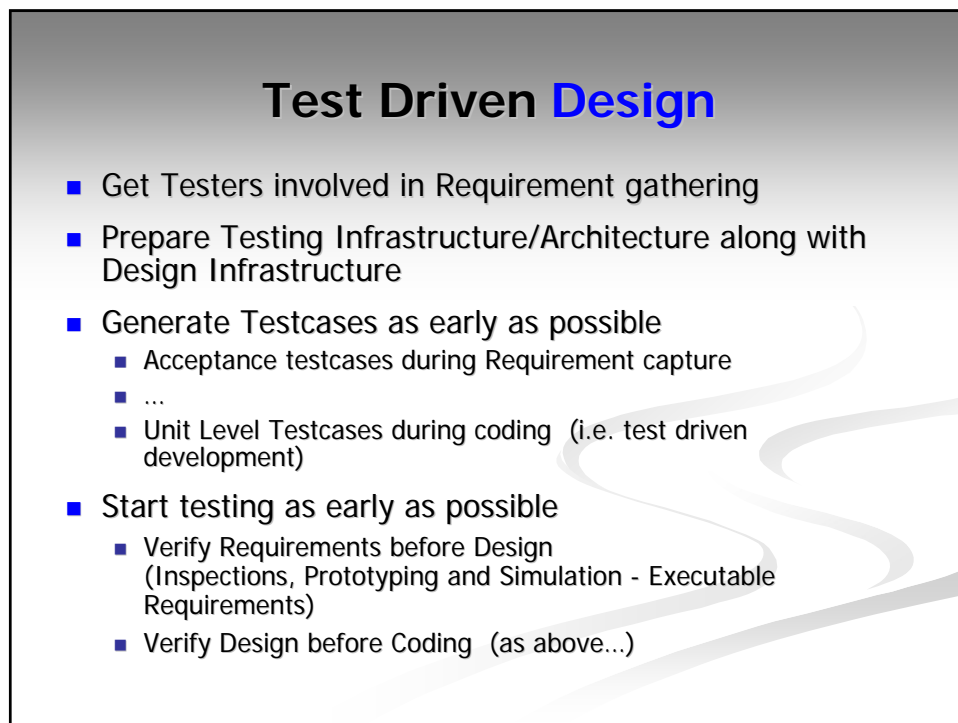
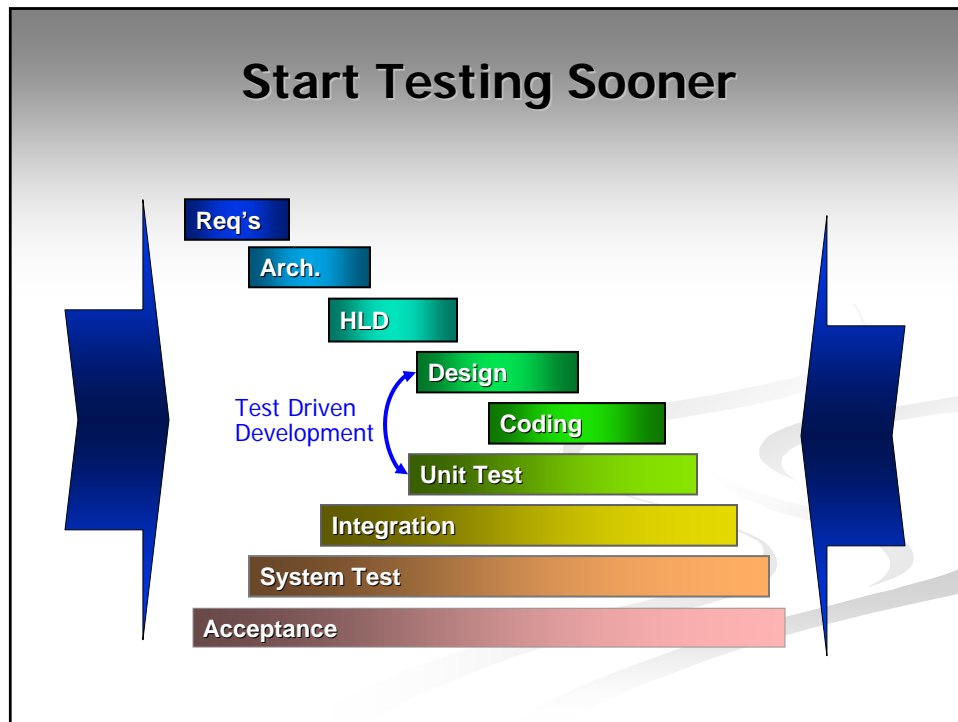
Provide set of techniques to accelerate testing that \*I\* (and friends) have used in real projects

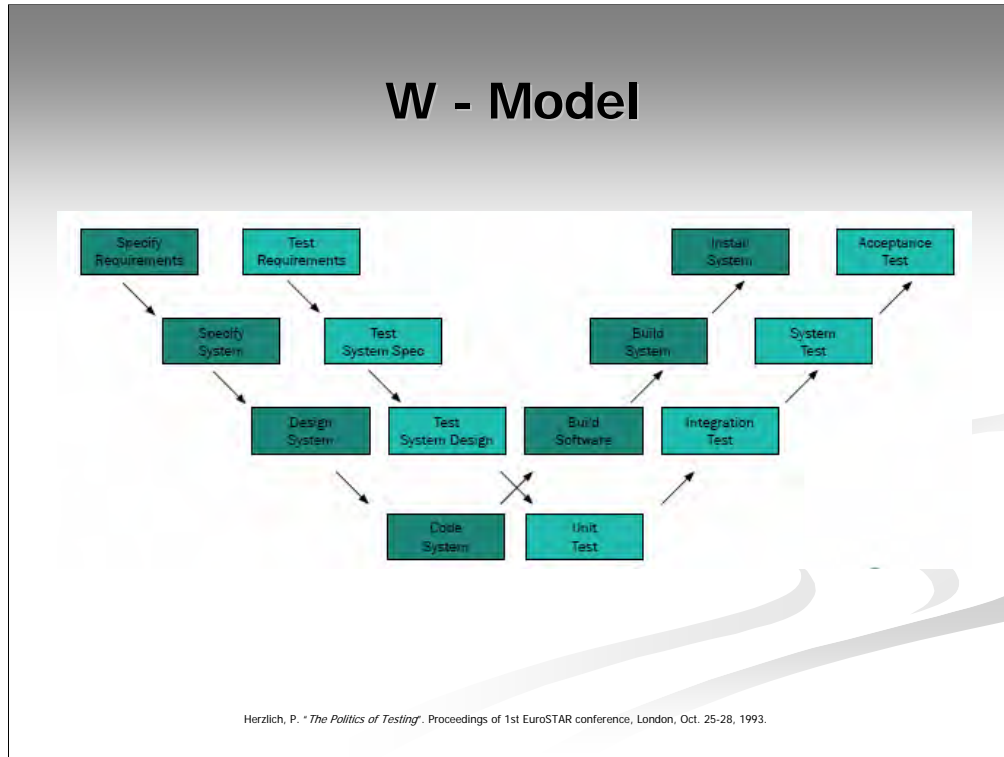
## Agenda

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

## How do you make testing faster?





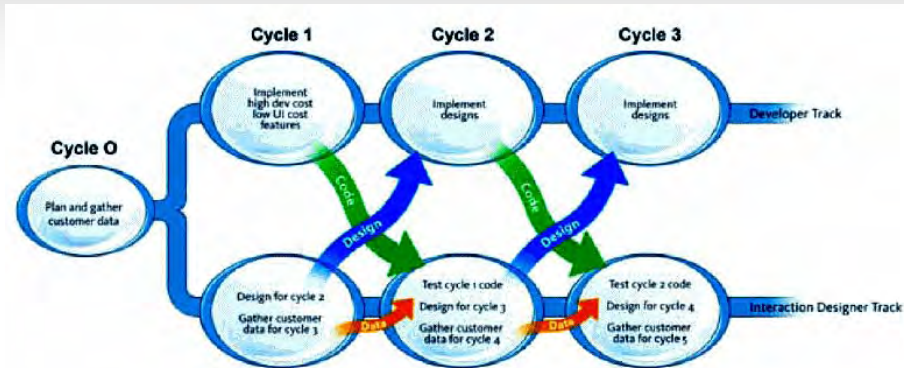


the V model

W model - Herzlich, P. (1993). "*The Politics of Testing*". Proceedings of 1st EuroSTAR conference, London, Oct. 25-28, 1993.

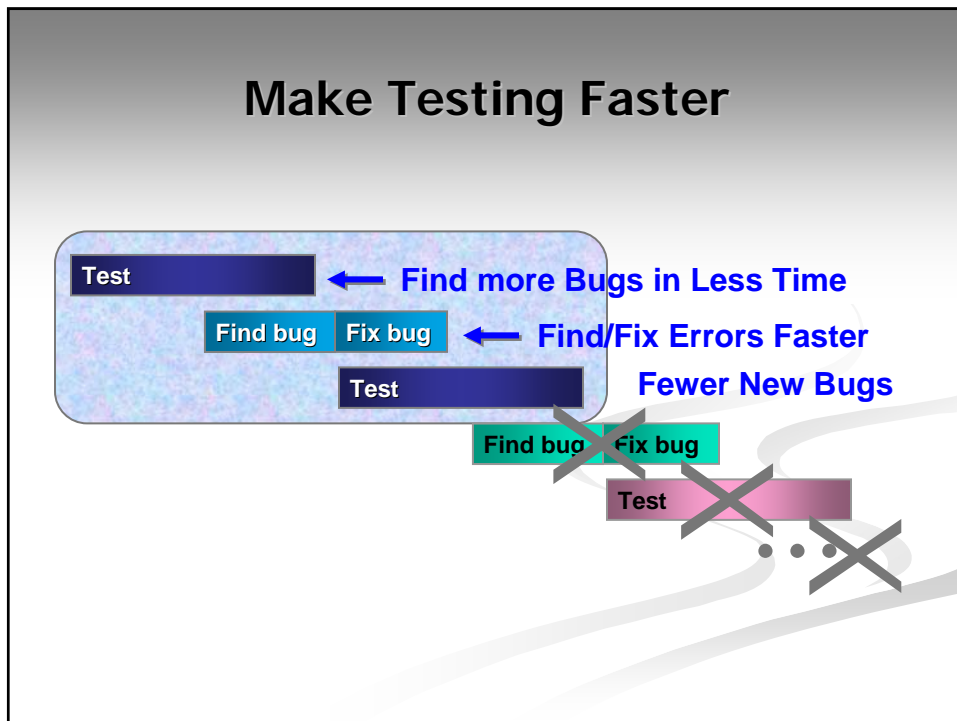
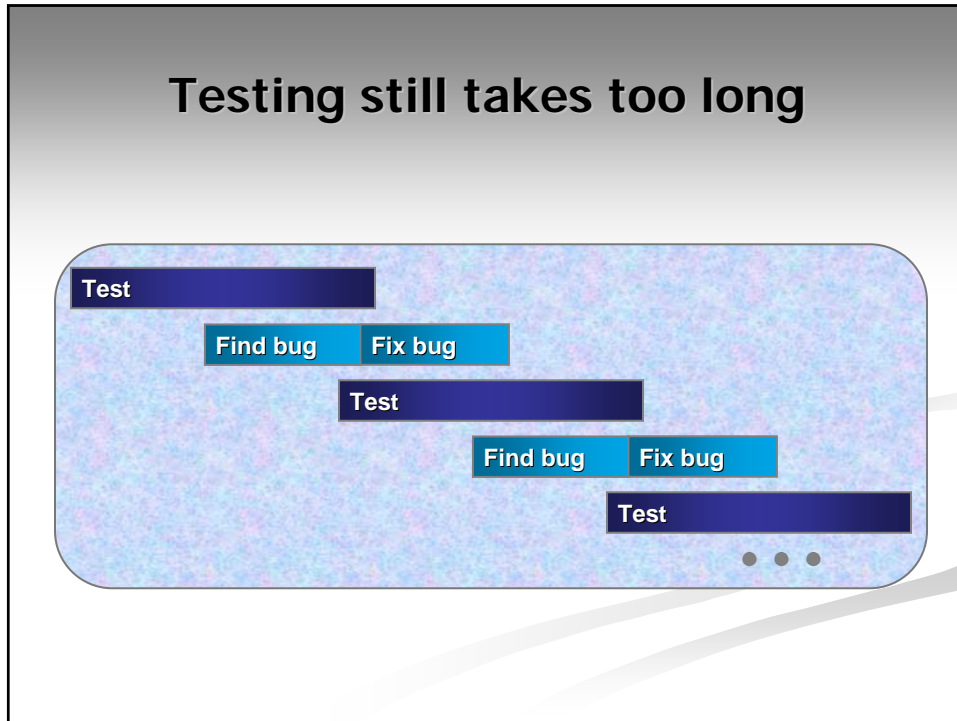
Christie, James. 2008, *The Seductive and Dangerous V-Model* p73 *Testing Experience* April 2008, [www.testingexperience.com](http://www.testingexperience.com)

## Example: Agile and Usability



© Lynn Miller, ALIAS Wavefront.  
[http://wiki.fluidproject.org/download/attachments/1704207/Autodesk\\_WUD2006\\_UCDandAgile\\_lmiller.pdf?version=1](http://wiki.fluidproject.org/download/attachments/1704207/Autodesk_WUD2006_UCDandAgile_lmiller.pdf?version=1)

*Alain Désilets, NRC, OSQA May 2007* **Building the Right Thing Through Agile User-Centered Development.**



## Proven Practices (shopping list)

- **Reduce Testing Time**
  - Less testcases to write, automate, run, verify and maintain (Testcase generation)
  - Faster execution/validation (Testcase automation)
  - Only re-test the parts of the code that changed (Churn)
  - Know when to stop (Software Reliability Engineering, Risk Driven)
- **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)
  - Design for Testability
  - Error Simulation
  - Continuous Integration / Nightly Loadbuilds – automated Sanity
  - Built in Self Test
- **Fix bugs faster and create fewer new bugs**
  - Understand how the code works (Reverse Engineering)
  - Test Functionality as soon as it becomes available
  - Improve Maintainability
  - Rapid Root Cause Analysis

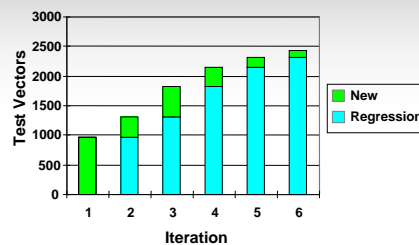
## To Be Discussed

- **Reduce Testing Time**
  - Less testcases to write, automate, run, verify and maintain (Testcase generation)
  - Faster execution /validation (Testcase automation)
  - Only re-test the parts of the code that changed (Churn)
  - Know when to stop (Software Reliability Engineering, Risk Driven)
- **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)
  - Design for Testability
  - Error Simulation
  - Continuous Integration / Nightly Loadbuilds – automated Sanity
  - Built in Self Test
- **Fix bugs faster and create fewer new bugs**
  - Understand how the code works (Reverse Engineering)
  - Test Functionality as soon as it becomes available (Iterate)
  - Improve Maintainability
  - Rapid Root Cause Analysis

## What was in Part 1?

- **Reduce Testing Time**
  - Less testcases to write, automate, run, verify and maintain (Testcase generation)
  - Faster execution/validation (Testcase automation)
  - Only re-test the parts of the code that changed (Churn)
  - Know when to stop (Software Reliability Engineering, Risk Driven)
- **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)
  - Design for Testability
  - Error Simulation
  - Continuous Integration / Nightly Loadbuilds – automated Sanity
  - Built in Self Test
- **Fix bugs faster and create fewer new bugs**
  - Understand how the code works (Reverse Engineering)
  - Test Functionality as soon as it becomes available (Iterate)
  - Improve Maintainability
  - Rapid Root Cause Analysis

## 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

## Automated Test Execution

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

## 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.

```

graph TD
    Start[Create and Set timer T2] --> T2_active[T2 active]
    Start --> T2_inactive_top[T2 inactive]
    T2_active --> Cancel_T2[Cancel T2]
    T2_inactive_top --> Error_issued_1[Error is issued]
    Cancel_T2 --> T2_inactive_bot[T2 inactive]
    Cancel_T2 --> T2_active_bot[T2 active]
    T2_inactive_bot --> Reset_T2[Reset T2]
    T2_active_bot --> Error_issued_2[Error is issued]
    
```

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

## 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
- E.g. SAFSDEV, STAF, & Behaviour-Based

### Keyword Test Automation Framework

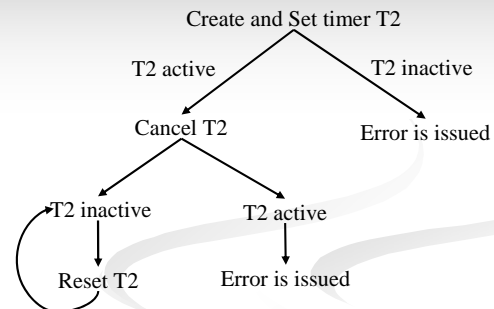
**Kai Chiu,**

**IBM Canada**

QUEST Toronto 2008

## Automated Test Creation

- Use model of system to automatically generate automated test code
- Types: State diagrams (including UML, SDL, etc.), Sets, Grammars, Combinations



- Tools to validate model, find minimum test paths, etc.
- Best of all: manage the model and not the test code

## 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

<http://www.pairwise.org/>

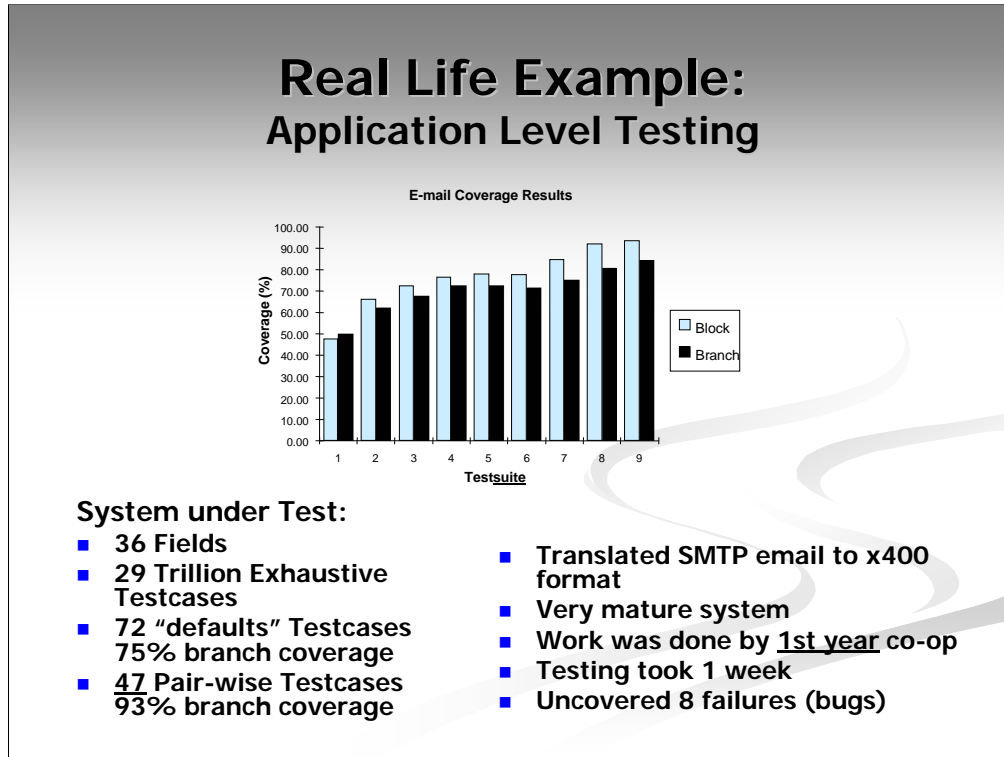
## Pair-wise Test Cases: 13 Fields, 3 Inputs, 15 Test cases

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**

Field 1 & 2 Exhaustive 9 testcases

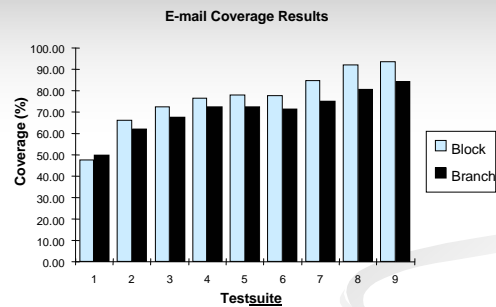
Fields 1 – 3, would need 18 exhaustive testcases



Manually execute 20 to 30 testcases per week, Automate 3 – 5 per week.

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](http://aetgweb.argreenhouse.com/aetg_nortel.pdf)

## Real Life Example: Application Level Testing



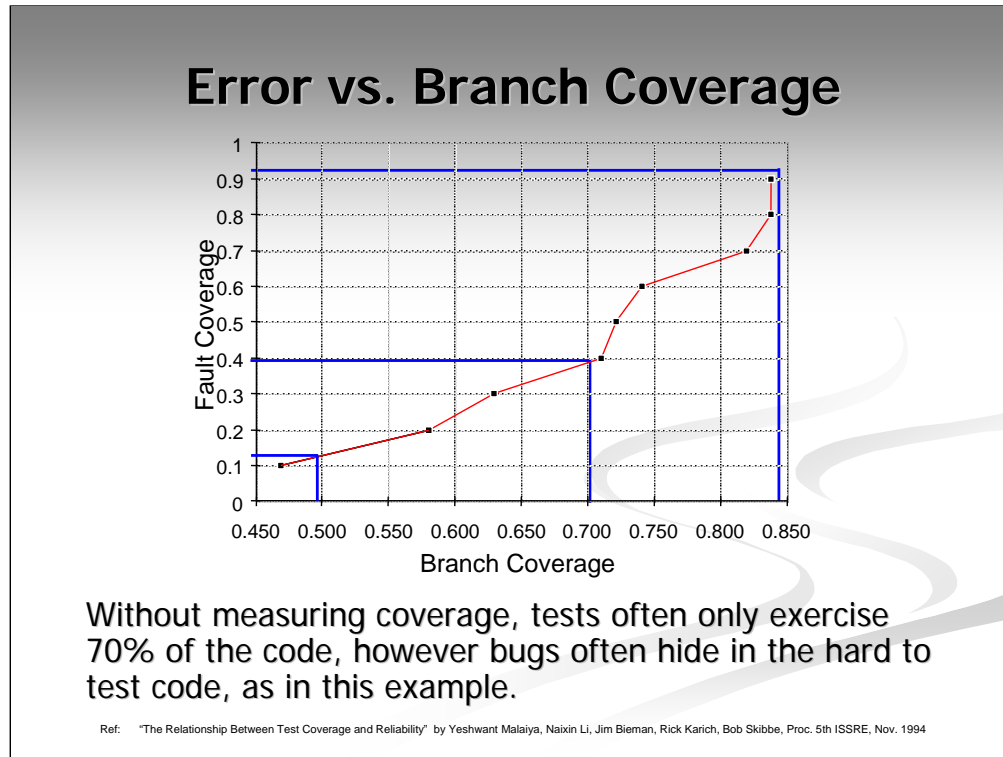
### Benefits:

- 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

Manually execute 20 to 30 testcases per week, Automate 3 – 5 per week.

## 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**



This graph is from a study trying to demonstrate the relationship between reliability and code coverage during component test. [Malaiya, 94]

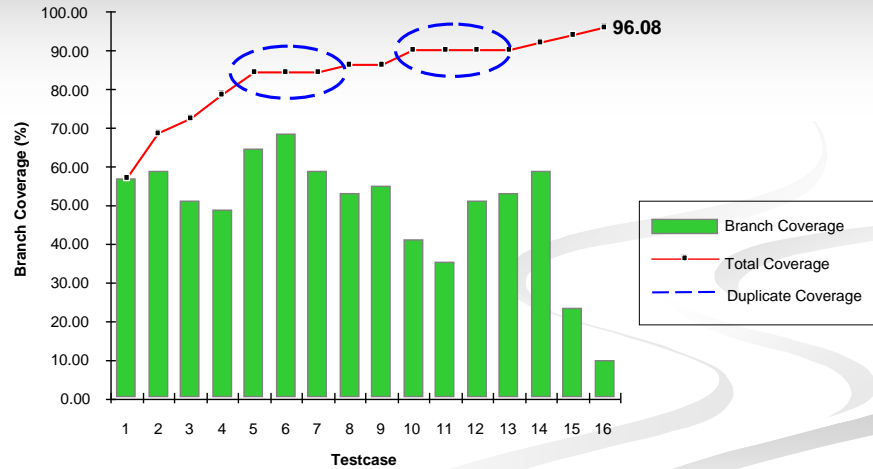
Typically, the results would lay between this curve and the diagonal.

Note: 50% branch coverage only provides 15% fault coverage.

Graph doesn't represent effort involved

"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
<b>TOTAL</b>	<b>9324</b>	<b>1282</b>	<b>14</b>	<b>12257</b>	<b>1693</b>	<b>14</b>

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
<b>TOTAL</b>	<b>39</b>	<b>26</b>	<b>67</b>	<b>39</b>	<b>34</b>	<b>87</b>

The names of software areas were proprietary so have been removed.  
Shows functional improvement in Sanity Suite coverage

Can't get higher than this (Config mgmt data was used to track procedures to subsystems)

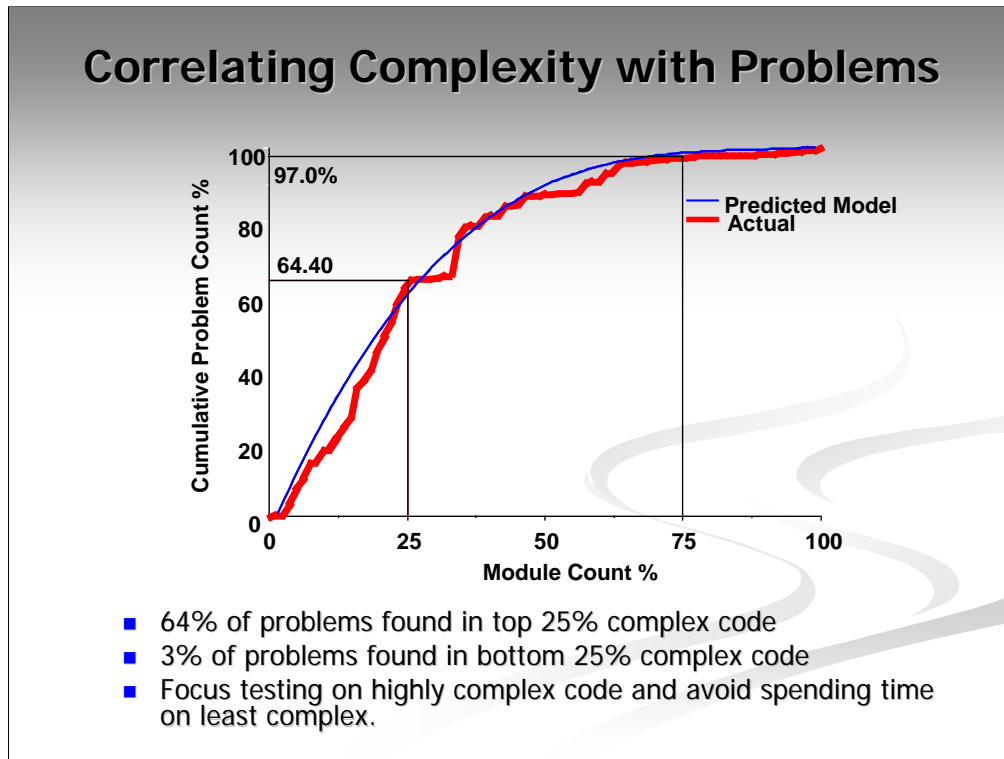
Hitting a subsystem may mean a single procedure was called and exited.

Final missed subsystems were involved in booting system which this code coverage method could not measure

Testers may not know what individual procedures do, but missed classes/files, subsystems, etc. probably have names that can give you a hint to what functionality was missed.

## 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.

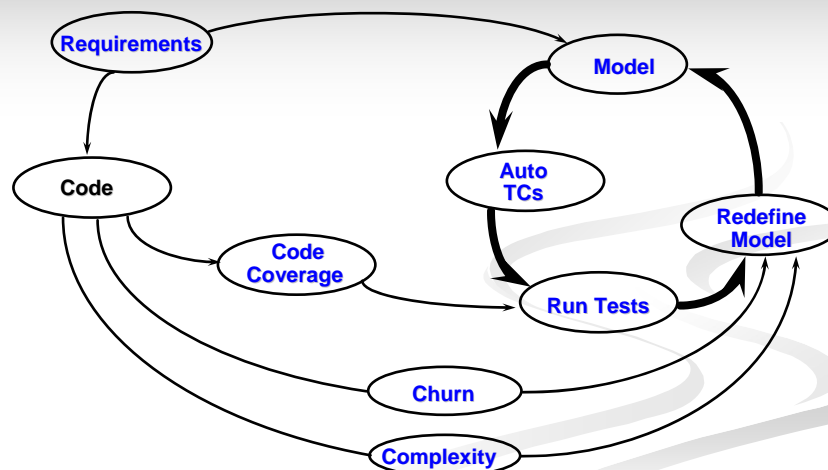


Malaiya, Y., Li, N., Bieman, J., Karich, R., & Skibbe, B., 1994, *The Relationship Between Test Coverage and Reliability*, Proc. 5th ISSRE, Nov. 1994

## 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

## One Bubble at a Time

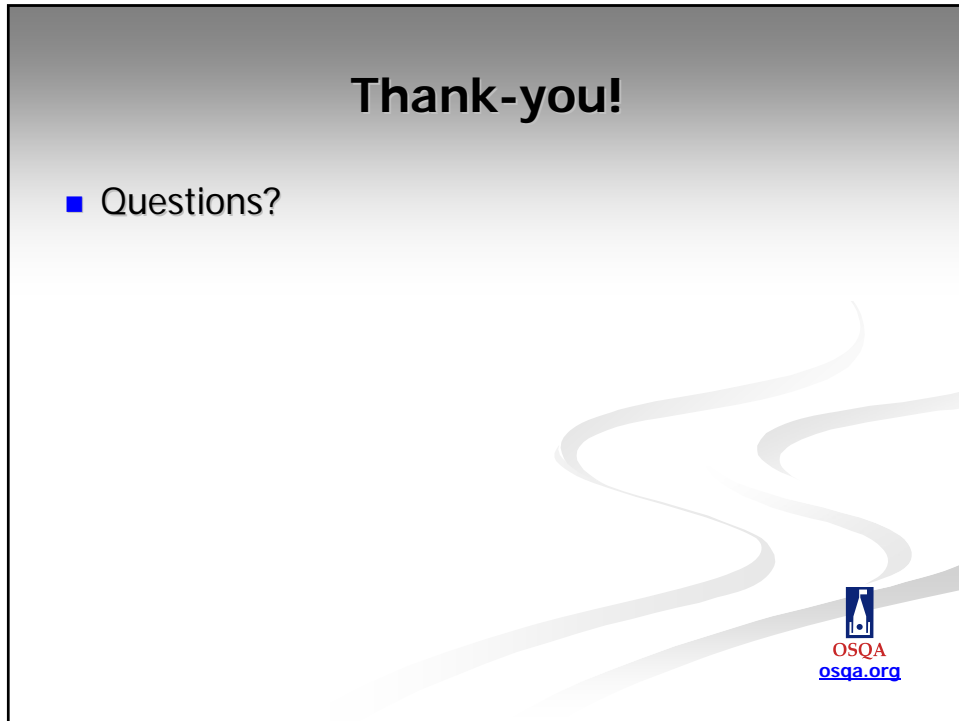


## Baby steps and Continuous Improvement

- 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


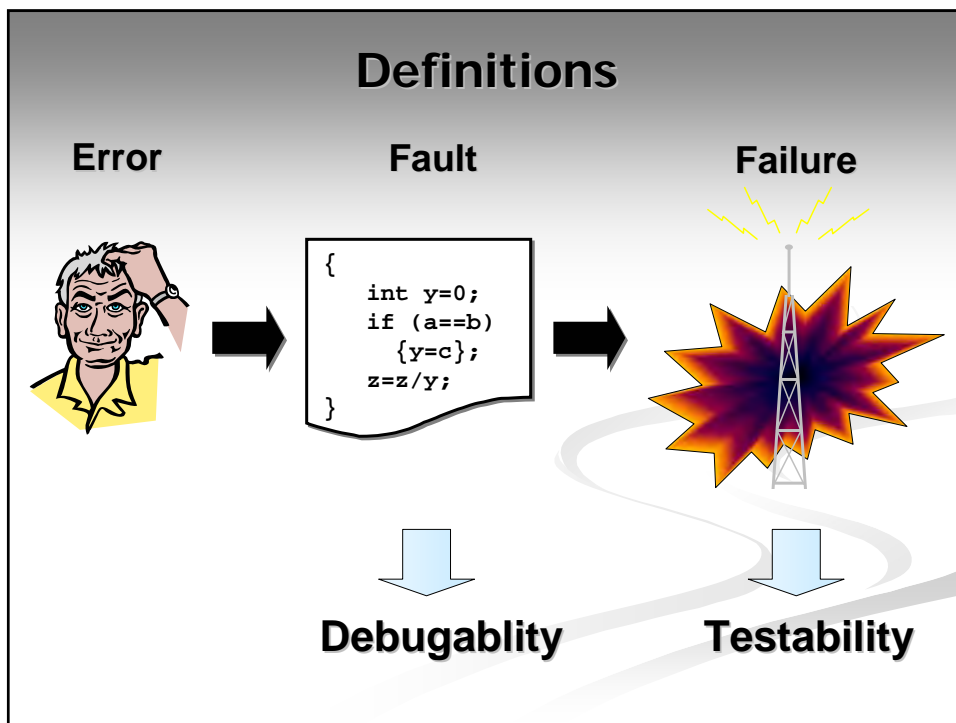
## Conclusion

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



## Automated Test Validation (aka Oracle)

- Hard coded
  - Explicit "Asserts" or coded interaction with SUT (e.g. through API)
  - **Created by model!**
    - SUT does not follow test case's logic
    - Generate verification code as well as execution code
- Capture Replay
  - Compare test results to results from a previous version
    - Previous version becomes the "Oracle"
    - **Previous version could also be a model** (e.g. prototype)
  - Filter out or clean up the changeable bits
  - Bitmaps bad, **text and numbers good**

## Design for Testability

For a failure to take place requires:

- Execution:** An input must cause a fault to be executed
- Infection:** As a consequence of **Execution**, the internal data state must become incorrect
- Propagation:** As a consequence of **Infection**, the data state error must become detectable in an output.

DFT makes it easier for these conditions to be met.

See Jeff Vos's work on Testability

## Design for Testability

- Make the bugs easier to exercise
  - Built in Unit Test
  - Error/Output Simulation
  - Rules of Thumb
- Make bugs easier to see
  - Assertions
  - Trace
- Needs to be “Designed” in, but makes it easier for developers to find, fix, and change code as well.

### ***Make it easier to exercise code:***

#### **1. Built-in Unit Test**

Every class or module is able to test itself

A common Smalltalk idea

Test code is co-located with functional code making maintenance easier

ServiceBuilder in UK had each class check itself and its components during loadbuild and decide itself whether it should be included in the build.

An architectural decision

#### **2. Error/Output Simulation**

Where one level of software can be told to return an output value on demand (e.g. an error value).

Makes fault path testing much easier

Don't have to build separate test harnesses to exercise code. Code becomes its own stubs.

Again an architectural decision.

#### **3. Rules of Thumb**

Design Guidelines

Keep the code simple - need at least 1 test for every path through the code (Cyclomatic Complexity)

A function/procedure/method should only do one thing (Cohesion)



Some examples from:  
Burr, Kevin. 1998, Test Acceleration, IEEE Ottawa, June 1998