Test Automation at Discount Store Prices

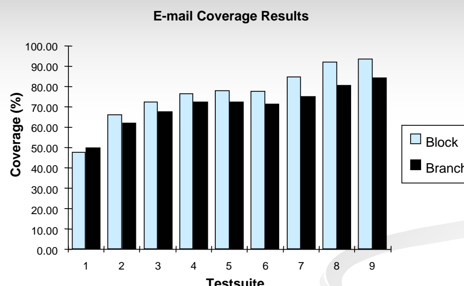
March 18, 2009
Ottawa Software Quality Association
www.osqa.org

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

Myths about Automation

- Test automation only saves you time during Regression testing
- You can't automate tests until…
  - the product is stable
  - you have a product to test.
  - until the requirements are stable

Real Life Examples:
Application Level Testing

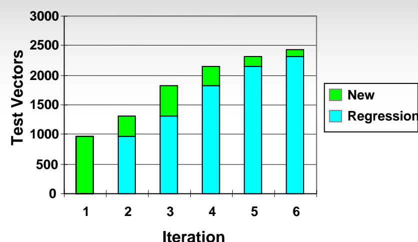


System under Test:

- 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)

[More later]

More Real Life Examples:
Unit Level Testing



System under Test:

- Unit testing of Controllers and Device Drivers on Real-time Embedded System
- First iteration was done manually
- Everything after the second was automated
- Effort here is measured in test vectors. A test vector is an individual test stimulus

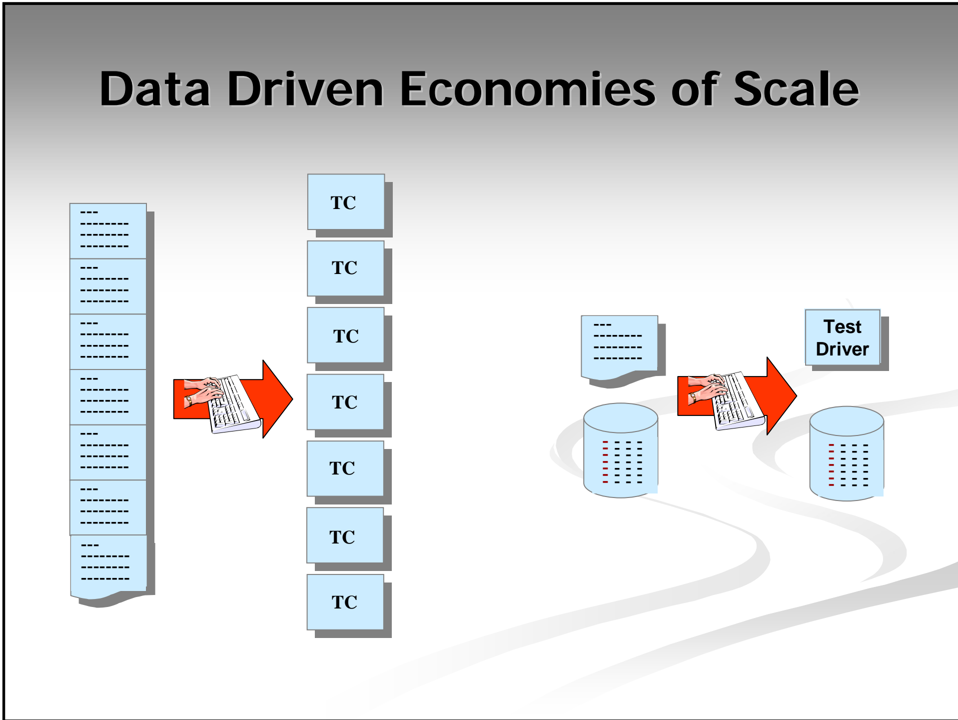
- Iteration 1 represents the number of manual test vectors that can be run in 2 weeks..
- By iteration 6 productivity essentially increased 2.5 times !
- Over 5,000 more test vectors were run than would have been possible manually!
- This is equivalent to 10 weeks extra testing for free.
- During the project, the tester was also able to take a 2 week vacation and work on other assignments, all which would not have been possible using the original manual approach

And More Real Life Examples:

- SOA System Performance Testing
  - 100 testcases, 1 hour per testcase, on 2 testbeds, in 1 week by 1 tester
  - Up to 100 simulated users with multiple roles, multiple event patterns, etc.
  - Tester was also writing a test tool at the same time.
- Web Application System Testing
  - 70 testcases created and executed in 1 week.
- Reduced 700 test cases to 130 Integration Testcases
- Automated Installation testing to test every possible configuration for routing framework.
- Etc.

Agenda

- Myths vs. Reality
- Types of Automation
  1. Automated Test Creation
  2. Automated Test Execution
  3. Automated Test Validation
- When Automation Goes Wrong
- Baby Steps
- Reducing Test Time Further
- Conclusion

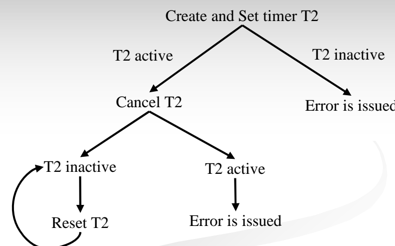


2.0 Automated Test Execution

Approach	Pros	Cons
Record and Playback	▪ Fast and easy to create ▪ Requires little to no coding experience	▪ Breaks easily ▪ Hopeless to maintain
Structured Code	▪ More robust ▪ Intermediate coding skills	▪ Needs more planning ▪ Maintenance is a problem ▪ Leads to code duplication ▪ Slowest to create (unless using cut and paste)
Shared code	▪ Uses shared libraries for common functions (i.e. code re-use) ▪ Faster to create, easier to maintain & more robust ▪ Supports mix of coding skills	▪ Requires good programmers to support re-useable code.
Data Driven	▪ Uses parameters or data files to drive testcases ▪ Fastest to create testcases ▪ Less code, less maintenance	▪ Requires better coding skills ▪ Test scenarios may be complex in order to take advantage of common test driver
Keyword Driven	▪ Simple keywords capture actions ▪ Requires testers to have least coding experience	▪ Requires experienced programmer to maintain/create parser ▪ Keyword "scripts" have to be maintained

2.1.1 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

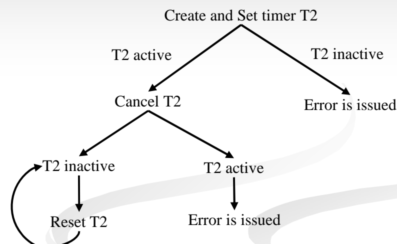
2.2 Keyword Driven Automation

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

1.0 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

1.1.1 Pair-wise test generation

- Many systems suffer from a combinatorial explosion of possible testcases
- Pair-wise test generation based originally on Design of Experiments Theory
  - DOE is an excellent way to create performance characterization tests for performance estimation.
  - Do not need the balanced number of pairs in DOE for regular testing.

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

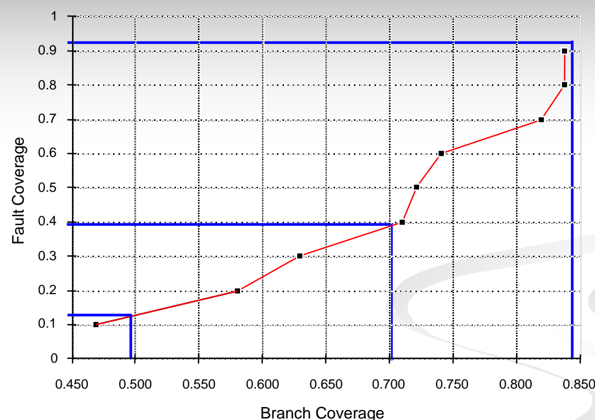
1.1.1 Pair-wise test generation

- 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

1.2 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

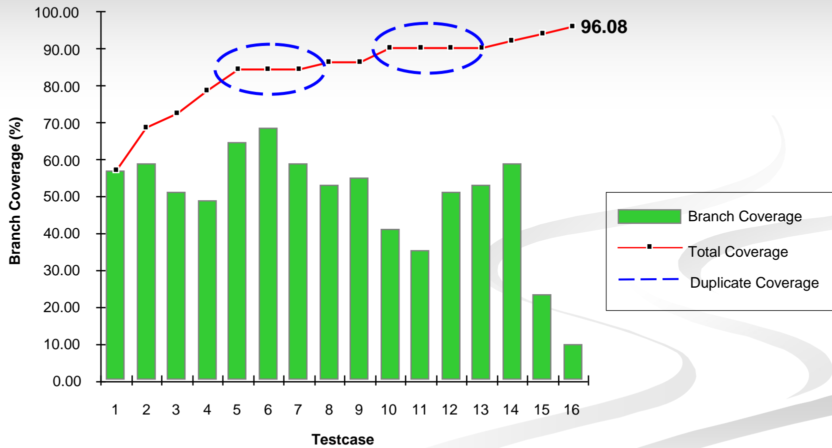
Error vs. Branch Coverage



Without measuring coverage, tests often only exercise 70% of the code, however bugs 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

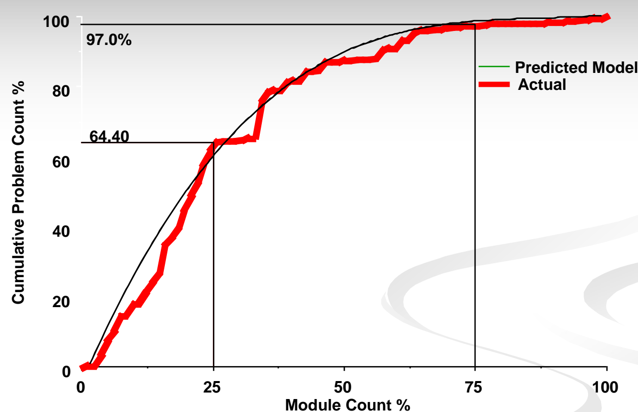
	V.56			V.87		
Area	Procedures	Hits	Percent Hit	Procedures	Hits	Percent Hit
	7	0	0	7	0	0
	0	0	0	88	13	15
	810	33	4	895	36	4
	102	26	25	114	38	33
	5545	893	16	7454	1117	15
	1736	304	18	2302	325	14
	570	24	4	606	65	11
	407	2	0	424	32	8
	13	0	0	321	67	21
	46	0	0	46	0	0
TOTAL	9324	1282	14	12257	1693	14

Area	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

2.1 Risk Based Testing

- 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.
- **Many types – just going to mention automated.**

Complexity Correlation with Problem Reports



- 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.

3.0 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**

When Automation Goes Wrong

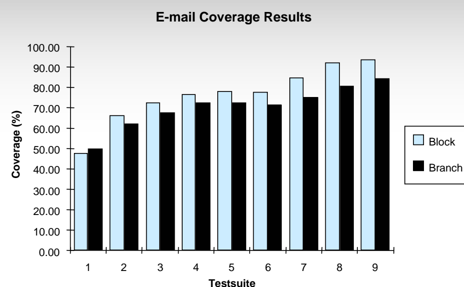
- Find false positives – no-one trusts results
- Does not find any bugs
- Runs slower than manual testing
- Tests become un-maintainable after a couple releases

When Automation Goes Wrong

- Find false positives – no one trusts results
- Does not find any bugs
- Runs slower than manual testing
- Tests become un-maintainable after a couple releases

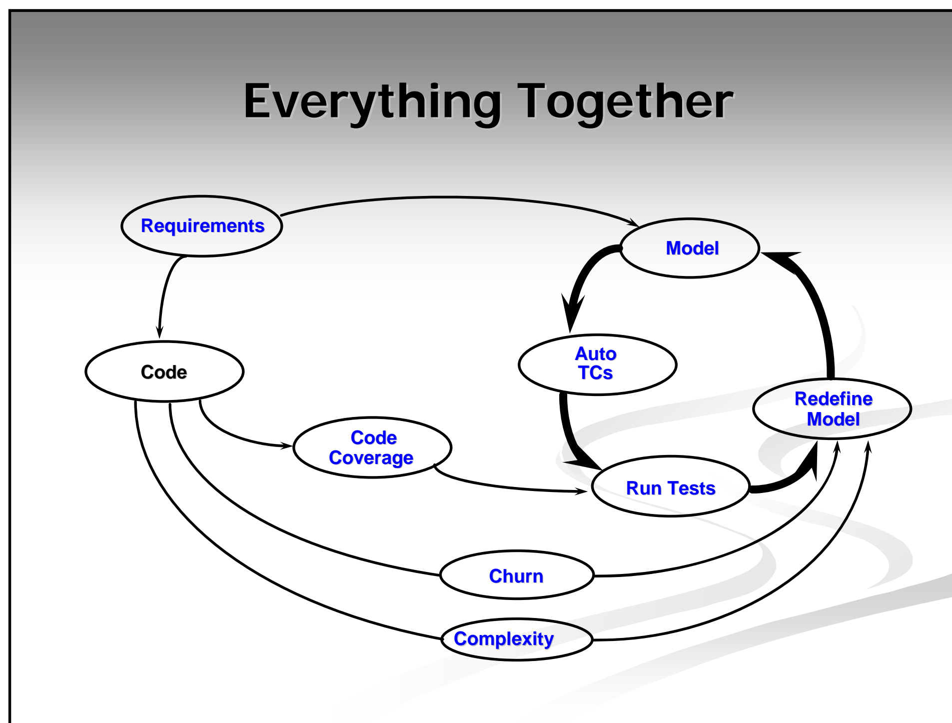
- Test Automation/Tools have to be treated like real code.
  - Needs to be code reviewed, tested, etc.
  - Needs to be maintainable
  - Garbage in Garbage out.
- No technique will find all bugs – use a variety
- If product bugs escape, find out why and fix the test code

Real Life Examples:
Application Level Testing



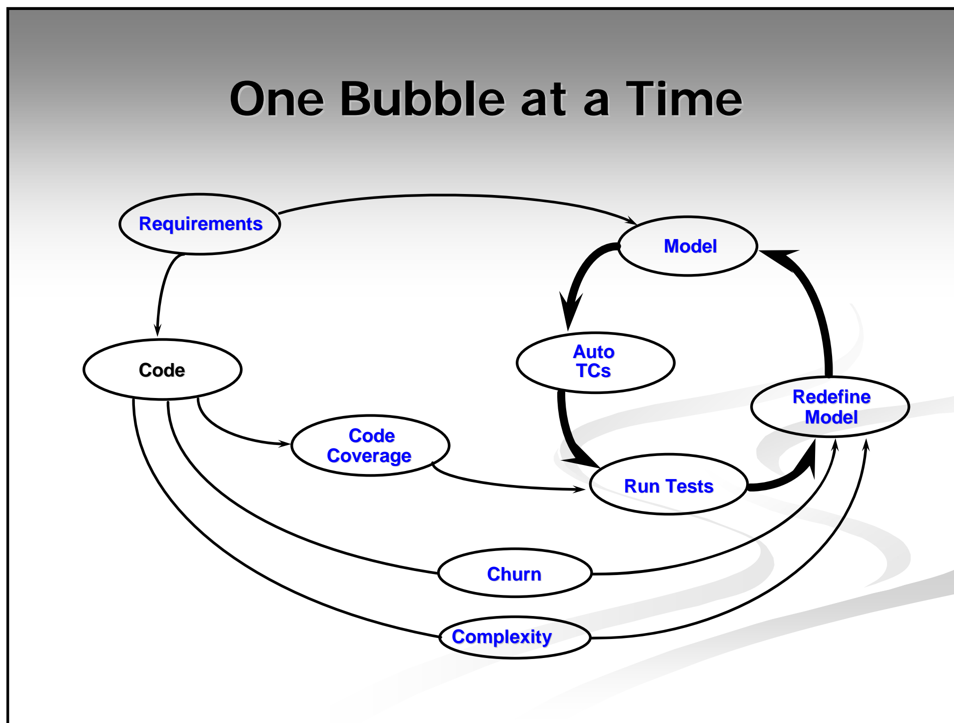
- 36 Fields
- 29 Trillion Exhaustive Testcases
- 72 "defaults" Testcases
- 75% branch coverage
- 47 Pair-wise Testcases
- 93% branch coverage

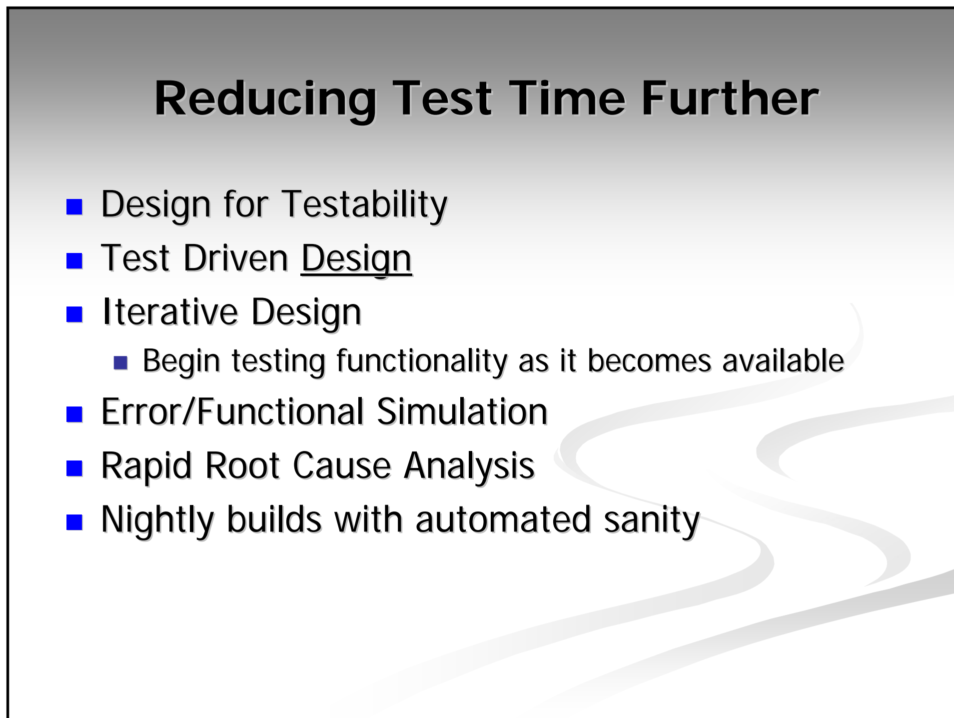
Benefits:
- **Fewer testcases: less time to test and verify results**
- **High level of code coverage**
- **Incorporates boundary testing**
- **Both success and failure testcases generated**
- **Ease of adapting testcase suite to changing requirements**
  - **We were given the wrong requirements initially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



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 Practioners
- Measure and evaluate progress


One Bubble at a Time
Requirements
Model
Code
Auto TCs
Redefine Model
Code Coverage
Run Tests
Churn
Complexity


Reducing Test Time Further
Design for Testability
Test Driven Design
Iterative Design
Begin testing functionality as it becomes available
Error/Functional Simulation
Rapid Root Cause Analysis
Nightly builds with automated sanity

Myths about Automation

- Test automation only saves you time during Regression testing

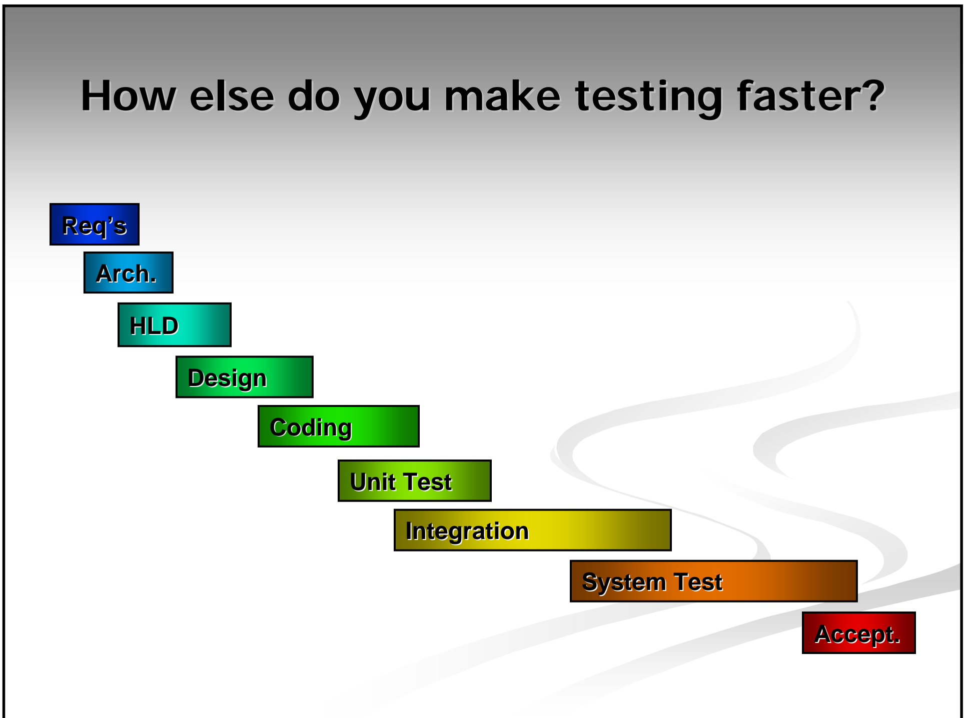
Depending on technique, you can generate and execute automated test cases faster than manual test cases

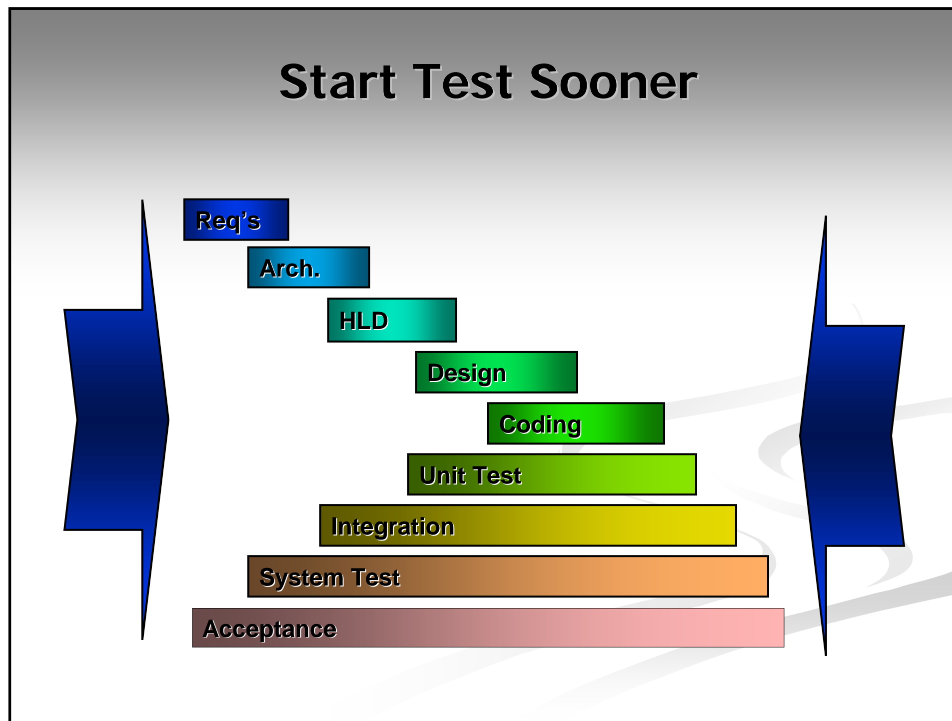
- You can't automate tests until…
  - the product is stable
  - you have a product to test.
  - until the requirements are stable

Models and/or Test drivers can be written in parallel with product code and tweaked as needed.

Thank-you!

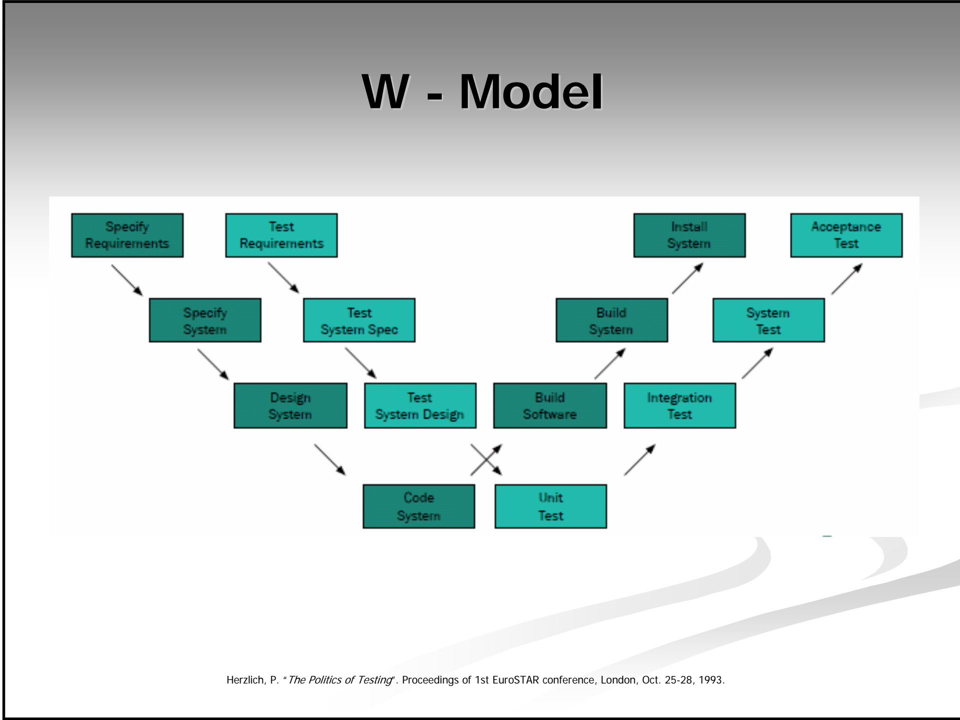
- Questions?

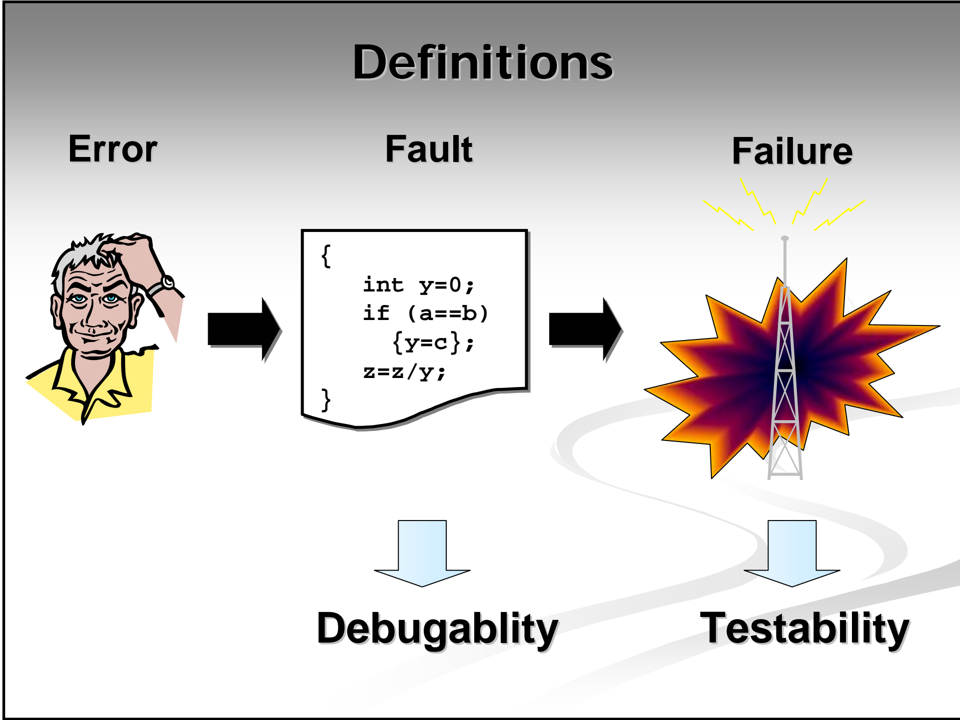

How else do you make testing faster?
Req's
Arch.
HLD
Design
Coding
Unit Test
Integration
System Test
Accept.



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
  - System testcases during High Level Design
  - Integration testcases during Detailed Design
  - 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...)


W - Model
Specify Requirements
Test Requirements
Install System
Acceptance Test
Specify System
Test System Spec
Build System
System Test
Design System
Test System Design
Build Software
Integration Test
Code System
Unit Test
Herzlich, P. "The Politics of Testing". Proceedings of 1st EuroSTAR conference, London, Oct. 25-28, 1993.


Definitions
Error
Fault
Failure
{
int y=0;
if (a==b)
{y=c};
z=z/y;
}
Debugablity
Testability

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.

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.

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