A Course on Software Test Automation Design

Doug Hoffman, BA, MBA, MSEE, ASQ-CSQE

Software Quality Methods, LLC. (SQM)

www.SoftwareQualityMethods.com

doug.hoffman@acm.org

Winter 2003

Copyright Notice

About Doug Hoffman

I advocate and provide advice and services in software testing and quality assurance.

Software quality assurance, and especially software testing, have a reputation of being where failed programmers or programmer "wanta be's" congregate. I don't believe it's true, and it's through courses like this that we can change the perception. I gravitated into quality assurance from engineering. I've been a production engineer, developer, support engineer, tester, writer, instructor, and I've managed manufacturing quality assurance, software quality assurance, technical support, software development , and documentation. Along the way I have learned a great deal about software testing and measurement. I enjoy sharing what I've learned with interested people.

Current employment

- President of Software Quality Methods, LLC. (SQM)
- Management consultant in strategic and tactical planning for software quality.
- Adjunct Instructor for UCSC Extension.

Education

- MBA, 1982.
- MS in Electrical Engineering, (digital design and information science) 1974.
- B.A. in Computer Science, 1972.

Professional

- Past Chair, Silicon Valley Section, American Society for Quality (ASQ).
- Founding Member and Past Chair, Santa Clara Valley Software Quality Association (SSQA, 1992-1997)
- Certified in Software Quality Engineering (ASQ, 1995).
- Previously a Registered ISO 9000 Lead Auditor, (RAB 1993).
- I also participate in the Los Altos Workshops on Software Testing.

Acknowledgment to Cem Kaner (Original Co-author)

He's in the business of improving software customer satisfaction.

He has worked as a programmer, tester, writer, teacher, user interface designer, software salesperson, organization development consultant, as a manager of user documentation, software testing, and software development, and as an attorney focusing on the law of software quality. These have provided many insights into relationships between computes, software, developers, and customers.

Current employment

- Professor of Software Engineering, Florida Institute of Technology
- Private practice in the Law Office of Cem Kaner

Books

- ***Testing Computer Software*** *(*1988; 2nd edition with Hung Nguyen and Jack Falk,1993). This received the *Award of Excellence* in the Society for Technical Communication's *Northern California Technical Publications Competition* and has the lifetime best sales of any book in the field.
- ***Bad Software: What To Do When Software Fails*** (with David Pels). Ralph Nader called this book "a how-to book for consumer protection in the Information Age."
- ***Lessons Learned in Software Testing*** (2002, with James Bach and Bret Pettichord) Doug describes the chapter on test automation better than any book on the subject available today.

Education

- J.D. (law degree, 1993). Elected to the American Law Institute, 1999.
- Ph.D. (experimental psychology, 1984) (trained in *measurement theory* and in *human factors*, the field concerned with making hardware and software easier and safer for humans to use).
- B.A. (primarily mathematics and philosophy, 1974).
- Certified in Quality Engineering (American Society for Quality, 1992). Examiner (1994, 1995) for the California Quality Awards.
- He also co-founded and/or co-host the Los Altos Workshops on Software Testing, the Software Test Managers' Roundtable, the Austin Workshop on Test Automation, the Workshop on Model-Based Testing, and the Workshop on Heuristic & Exploratory Techniques.

Acknowledgment

Many of the ideas in this presentation were presented and refined in Los Altos Workshops on Software Testing and The Austin Workshops on Test Automation.

LAWST 5 focused on oracles. Participants were Chris Agruss, James Bach, Jack Falk, David Gelperin, Elisabeth Hendrickson, Doug Hoffman, Bob Johnson, Cem Kaner, Brian Lawrence, Noel Nyman, Jeff Payne, Johanna Rothman, Melora Svoboda, Loretta Suzuki, and Ned Young.

LAWST 1-3 focused on several aspects of automated testing. Participants were Chris Agruss, Tom Arnold, Richard Bender, James Bach, Jim Brooks, Karla Fisher, Chip Groder, Elizabeth Hendrickson, Doug Hoffman, Keith W. Hooper, III, Bob Johnson, Cem Kaner, Brian Lawrence, Tom Lindemuth, Brian Marick, Thanga Meenakshi, Noel Nyman, Jeffery E. Payne, Bret Pettichord, Drew Pritsker, Johanna Rothman, Jane Stepak, Melora Svoboda, Jeremy White, and Rodney Wilson.

Bret Pettichord organized and led AWTA 1, 2, and 3.

Demographics: *How long have you worked in:*

- software testing
  - 0-3 months ____ 3-6 months ____
  - 6 mo-1 year ____ 1-2 years ____
  - 2-5 years ____ > 5 years ____
- programming
  - » Any experience _____
  - » Production programming _____
- test automation
  - » Test development _____
  - » Tools creation _____
- management
  - » Testing group _____
  - » Any management _____
- marketing _____
- documentation _____
- customer care _____
- traditional QC _____

Outline

Day 1
- Automation Example
- Foundational Concepts
- Some Simple Automation Approaches
- Automation Architectures
- Patterns for Automated Software Tests

Day 2
- Quality Attributes
- Costs and Benefits of Automation
- Test Oracles
- Context, Structure, and Strategies

Starting Exercise

Before I start talking about the different types of automation, I'd like to understand where you are and what you're thinking about (in terms of automation).

So . . . .

Please take a piece of paper and write out what you think automation would look like in your environment.

Automation in Your Environment

An Example to Introduce the Challenges

Automated GUI Regression Tests

The Regression Testing Strategy

Summary

- "Repeat testing after changes."

Fundamental question or goal

- Manage the risks that (a) a bug fix didn't fix the bug, (b) an old bug comes back or (c) a change had a side effect.

Paradigmatic cases

- <u>Bug regression</u> (Show that a bug was not fixed.)
- <u>Old fix regression</u> (Show that an old bug fix was broken.)
- <u>General functional regression</u> (Show that a change caused a working area to break.)

Strengths

- Reassuring, confidence building, regulator-friendly.

Blind spots

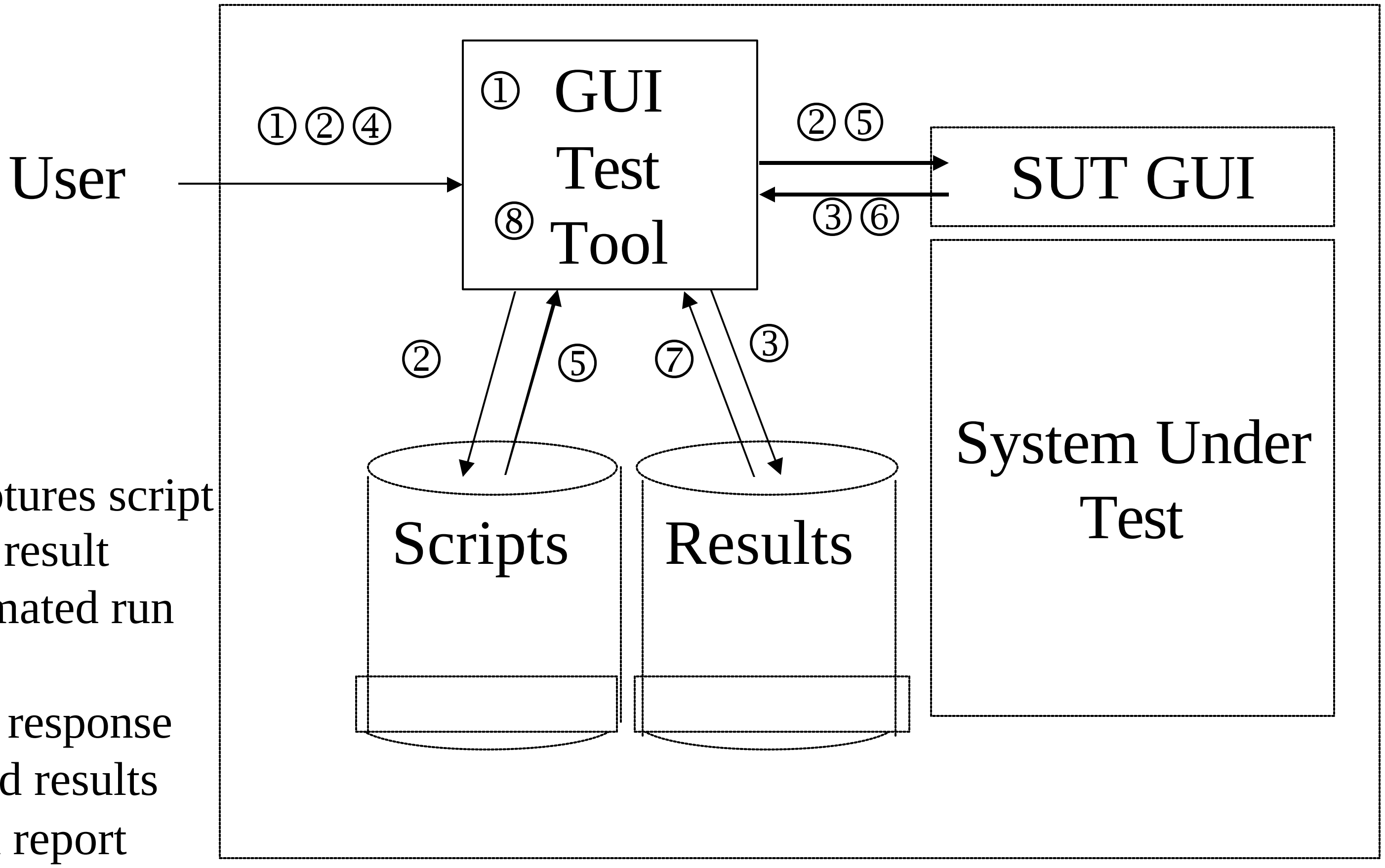
- Anything not covered in the regression series.
- Maintenance of this test set can be extremely costly.

Automating Regression Testing

The most common regression automation technique:

- conceive and create a test case
- run it and inspect the output results
- if the program fails, report a bug and try again later
- if the program passes the test, save the resulting outputs
- in future tests, run the program and compare the output to the saved results
- report an exception whenever the current output and the saved output don't match

A GUI Regression Test Model



① Launch tool
② Test; tool captures script
③ Test; capture result
④ Launch automated run
⑤ Play script
⑥ Capture SUT response
⑦ Read recorded results
⑧ Compare and report

But, Is This Really Automation?

Analyze product	**--**	**human**
Design test	**--**	**human**
Run test 1st time	**--**	**human**
Evaluate results	**--**	**human**
Report 1st bug	**--**	**human**
Save code	**--**	**human**
Save result	**--**	**human**
Document test	**--**	**human**
Re-run the test	**--**	**MACHINE**
Evaluate result	**--**	**MACHINE**

(plus* human *is needed if there's any mismatch)

Maintain result	**--**	**human**

We really get the machine to do a *whole lot* of our work!

(Maybe, but not this way.)

Automated Regression Pros and Cons

Advantages

- **Dominant automation paradigm**
- **Conceptually simple**
- **Straightforward**
- **Same approach for all tests**
- **Fast implementation**
- **Variations are easy**
- **Repeatable tests**

Disadvantages

- **Breaks easily (GUI based)**
- **Tests are expensive**
- **Pays off late**
- **Prone to failure because:**
  - **difficult financing,**
  - **architectural, and**
  - **maintenance issues**
- **Low power even when successful (finds few defects)**

Scripting

COMPLETE SCRIPTING is favored by people who believe that repeatability is everything and who believe that with repeatable scripts, we can delegate to cheap labor.

1 ____ Pull down the Task menu

2 ____ Select First Number

3 ____ Enter 3

4 ____ Enter 2

5 ____ Press return

6 ____ The program displays 5

Scripting: The Bus Tour of Testing

- **Scripting is the Greyhound Bus of software testing:**

  "Just relax and leave the thinking to us."

- ***To the novice, the test script is the whole tour. The tester goes through the script, start to finish, and thinks he's seen what there is to see.***
- ***To the experienced tester, the test script is a tour bus. When she sees something interesting, she stops the bus and takes a closer look.***
- ***One problem with a bus trip. It's often pretty boring, and you might spend a lot of time sleeping.***

GUI Automation is Expensive

- Test case creation is expensive. Estimates run from 3-5 times the time to create and manually execute a test case (Bender) to 3-10 times (Kaner) to 10 times (Pettichord) or higher (LAWST).
- You usually have to increase the testing staff in order to generate automated tests. Otherwise, how will you achieve the same breadth of testing?
- Your most technically skilled staff are tied up in automation
- Automation can delay testing, adding even more cost (albeit hidden cost.)
- Excessive reliance leads to the 20 questions problem. (Fully defining a test suite in advance, before you know the program's weaknesses, is like playing 20 questions where you have to ask all the questions before you get your first answer.)

GUI Automation Pays off Late

- GUI changes force maintenance of tests
  - » May need to wait for GUI stabilization
  - » Most early test failures are due to GUI changes
- Regression testing has low power
  - » Rerunning old tests that the program has passed is less powerful than running new tests
  - » Old tests do not address new features
- Maintainability is a core issue because our main payback is usually in the next release, not this one.

Maintaining GUI Automation

- GUI test tools must be tuned to the product and the environment
- GUI changes break the tests
  - » May need to wait for GUI stabilization
  - » Most early test failures are due to cosmetic changes
- False alarms are expensive
  - » We must investigate every reported anomaly
  - » We have to fix or throw away the test when we find a test or tool problem
- Maintainability is a key issue because our main payback is usually in the next release, not this one.

GUI Regression Automation Bottom Line

Extremely valuable under <u>some</u> circumstances

THERE ARE MANY ALTERNATIVES THAT MAY BE MORE APPROPRIATE AND MORE VALUABLE.

If your only tool is a *hammer*, every problem looks like a nail.

Brainstorm Exercise

I said:

- Regression testing has low power because:
  - » Rerunning old tests that the program has passed is less powerful than running new tests.

OK, is this always true?

When is this statement more likely to be true and when is it less likely to be true?

Notes

GUI Regression Strategies: Some Papers of Interest

Chris Agruss, *Automating Software Installation Testing*

James Bach, *Test Automation Snake Oil*

Hans Buwalda, *Testing Using Action Words*

Hans Buwalda, *Automated testing with Action Words: Abandoning Record & Playback*

Elisabeth Hendrickson, *The Difference between Test Automation Failure and Success*

Cem Kaner, *Avoiding Shelfware: A Manager's View of Automated GUI Testing*

John Kent, *Advanced Automated Testing Architectures*

Bret Pettichord, *Success with Test Automation*

Bret Pettichord, *Seven Steps to Test Automation Success*

Keith Zambelich, *Totally Data-Driven Automated Testing*

Software Test Automation:

Foundational Concepts

Why To Automate

The Mission of Test Automation

What is your test mission?

- What kind of bugs are you looking for?
- What concerns are you addressing?
- Who is your audience?

Make automation serve your mission.

Expect your mission to change.

Possible Missions for Test Automation

- Find important bugs fast
- Measure and document product quality
- Verify key features
- Keep up with development
- Assess software stability, concurrency, scalability…
- Provide service

Possible Automation Missions

Efficiency

- Reduce testing costs
- Reduce time spent in the testing phase
- Automate regression tests
- Improve test coverage
- Make testers look good
- Reduce impact on the bottom line

Service

- Tighten build cycles
- Enable “refactoring” and other risky practices
- Prevent destabilization
- Make developers look good
- Play to computer and human strengths
- Increase management confidence in the product

Possible Automation Missions

Extending our reach

- API based testing
- Use hooks and scaffolding
- Component testing
- Model based tests
- Data driven tests
- Internal monitoring and control

Multiply our resources

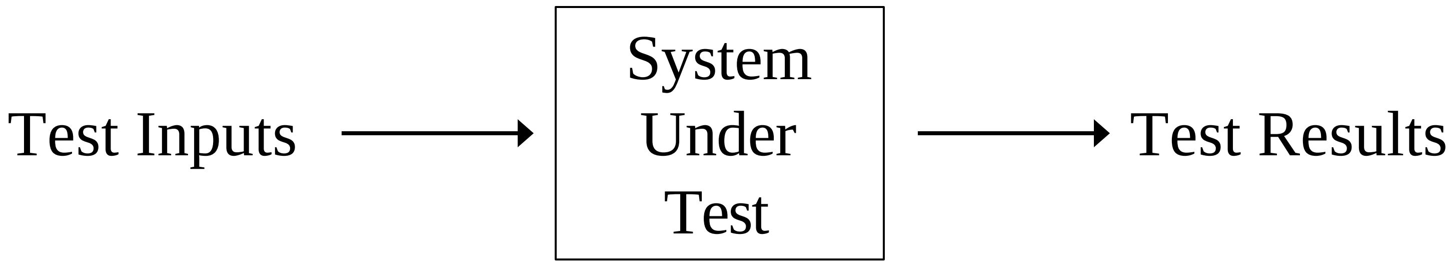
- Platform testing
- Configuration testing
- Model based tests
- Data driven tests

Software Test Automation:

Foundational Concepts

Testing Models

Simple [Black Box] Testing Model



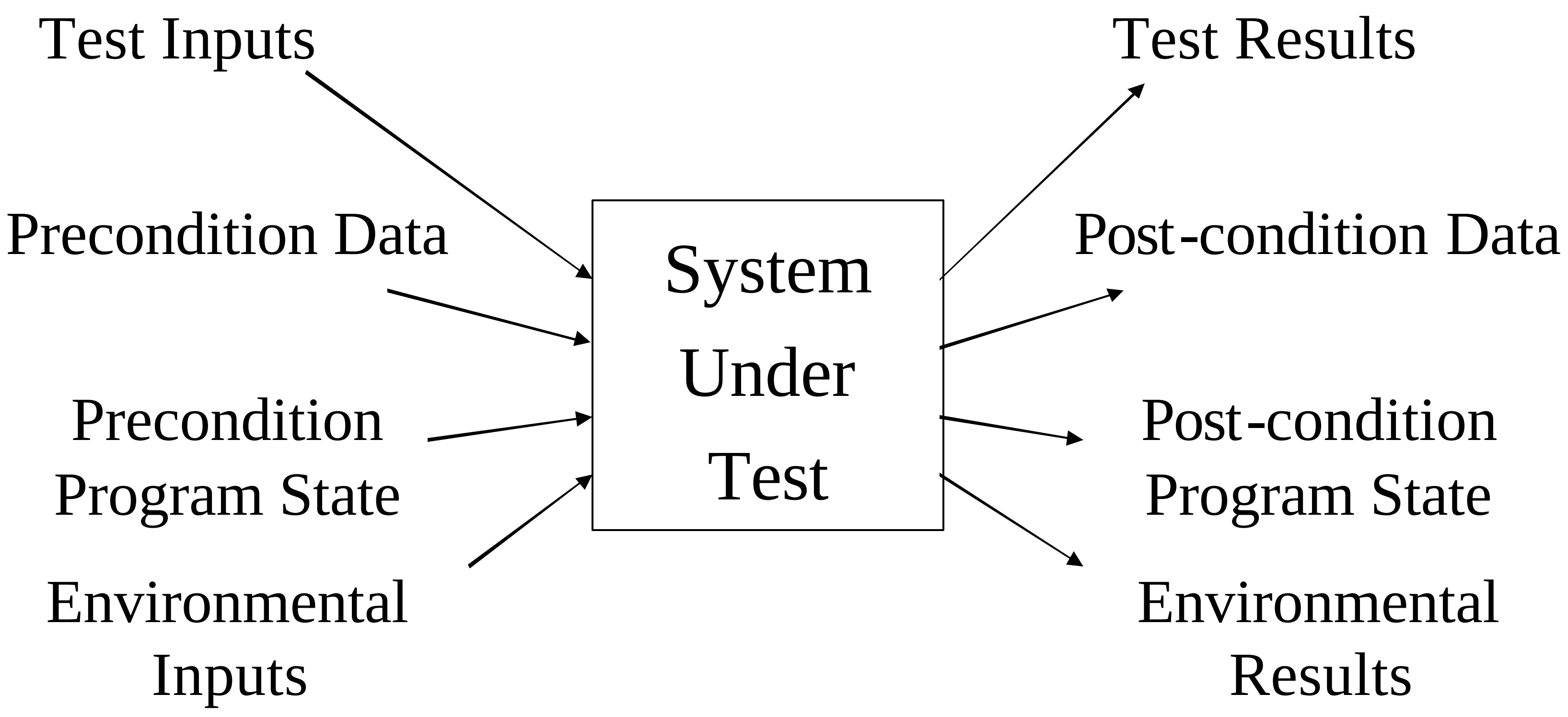
Implications of the Simple Model

- **We control the inputs**
- **We can verify results**

But, we aren’t dealing with all the factors

- Memory and data
- Program state
- System environment

Expanded Black Box Testing Model



Implications of the Expanded Model

We don’t control all inputs

We don’t verify everything

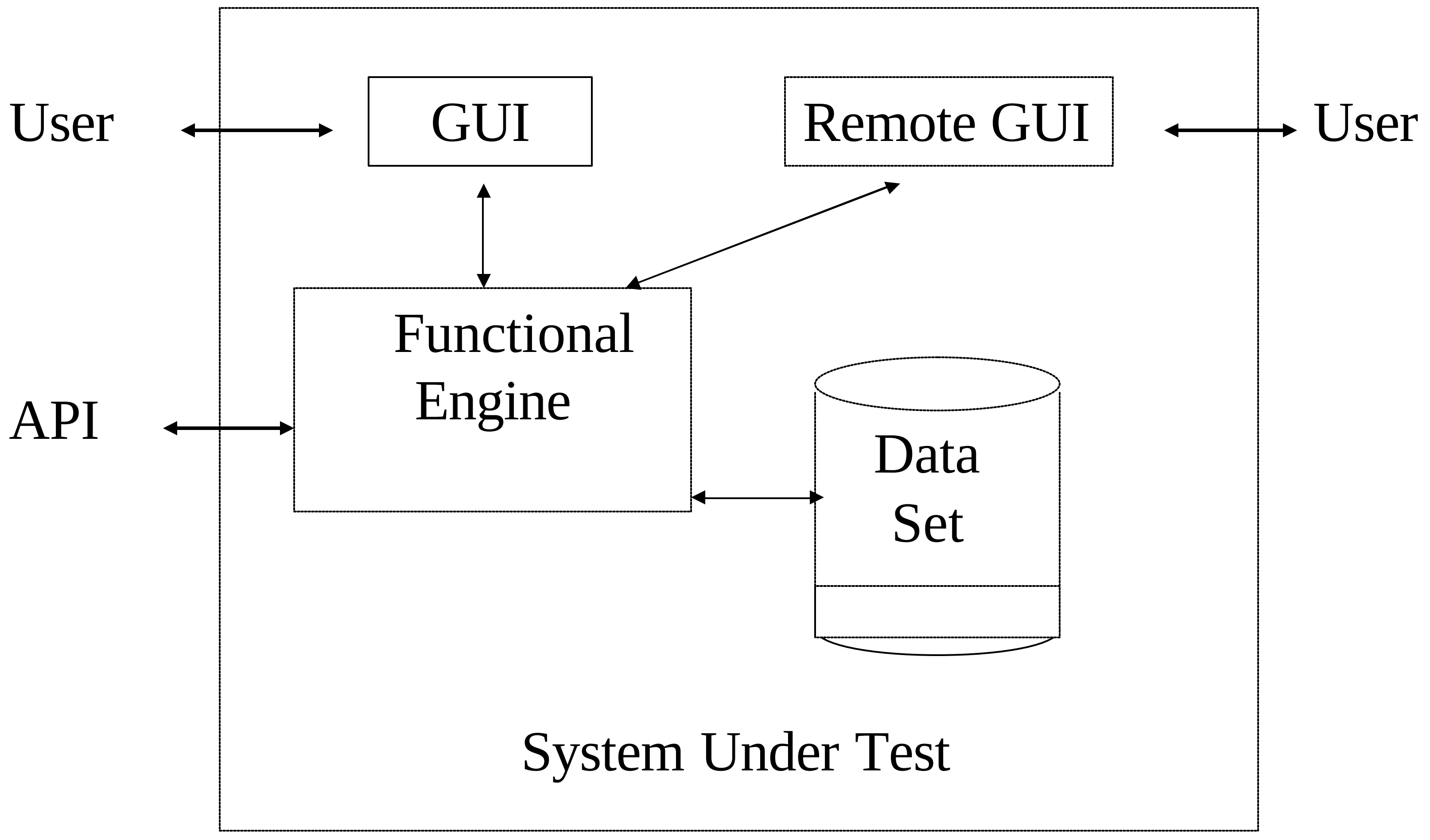
Multiple domains are involved

The test exercise may be the easy part

We can’t verify everything

We don’t know all the factors

An Example Model For SUT



Notes

Software Test Automation:

Foundational Concepts

The Power of Tests

Size Of The Testing Problem

- Input one value in a 10 character field
- 26 UC, 26 LC, 10 Numbers
- Gives 62^{10} combinations
- How long at 1,000,000 per second?

What is <u>your</u> domain size?

We can only run a vanishingly small portion of the possible tests

A Question of Software Testability

Ease of testing a product

Degree to which software can be exercised, controlled and monitored

Product's ability to be tested vs. test suite's ability to test

Separation of functional components

Visibility through hooks and interfaces

Access to inputs and results

Form of inputs and results

Stubs and/or scaffolding

Availability of oracles

An Excellent Test Case

- **Reasonable probability of catching an error**
- **Not redundant with other tests**
- **Exercise to stress the area of interest**
- **Minimal use of other areas**
- **Neither too simple nor too complex**
- **Makes failures obvious**
- **Allows isolation and identification of errors**

Good Test Case Design: Neither Too Simple Nor Too Complex

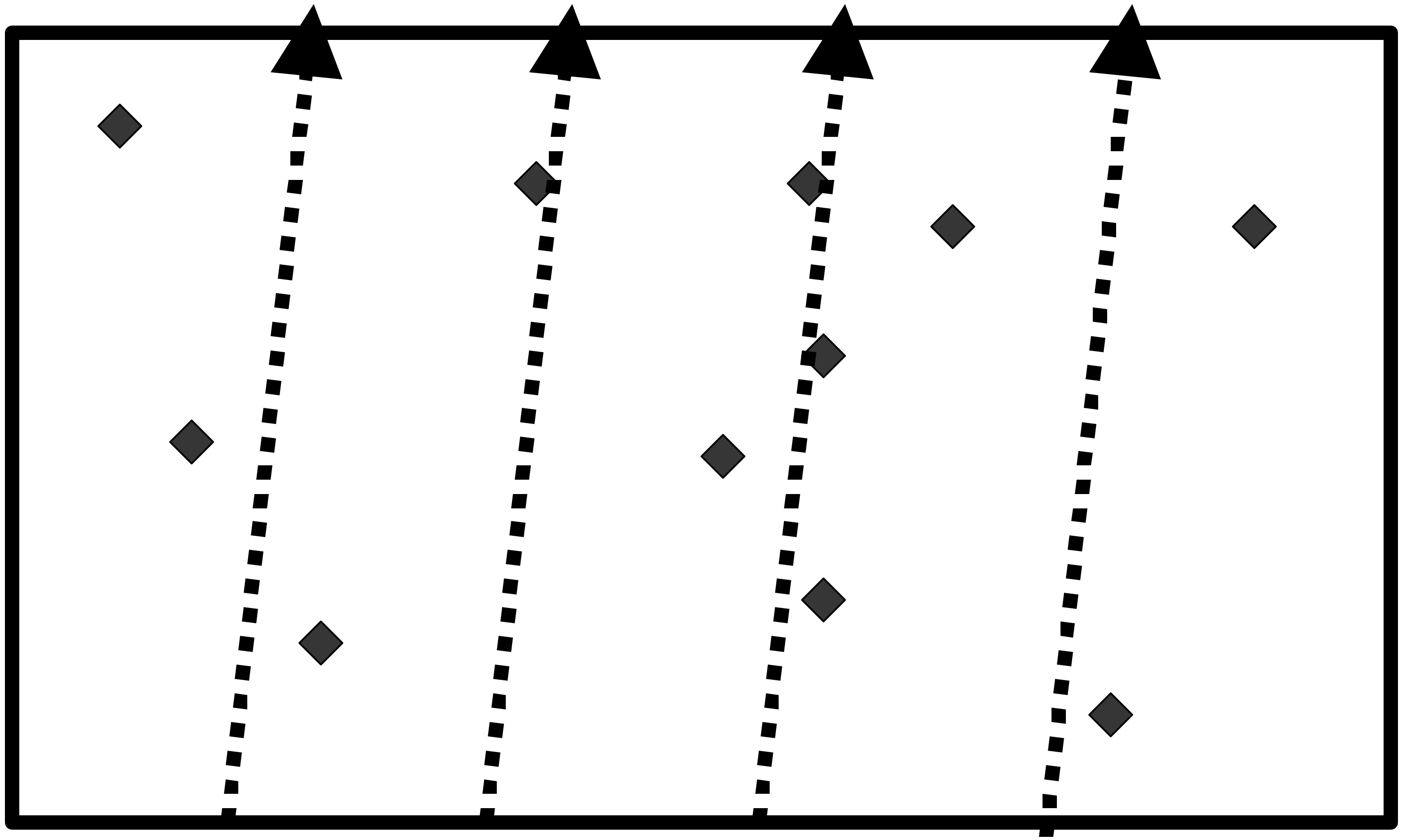
- What makes test cases simple or complex? *(A simple test manipulates one variable at a time.)*
- Advantages of simplicity?
- Advantages of complexity?

- Transition from simple cases to complex cases *(You should increase the power and complexity of tests over time.)*

- Automation tools can bias your development toward overly simple or complex tests

Refer to Testing Computer Software, pages 125, 241, 289, 433

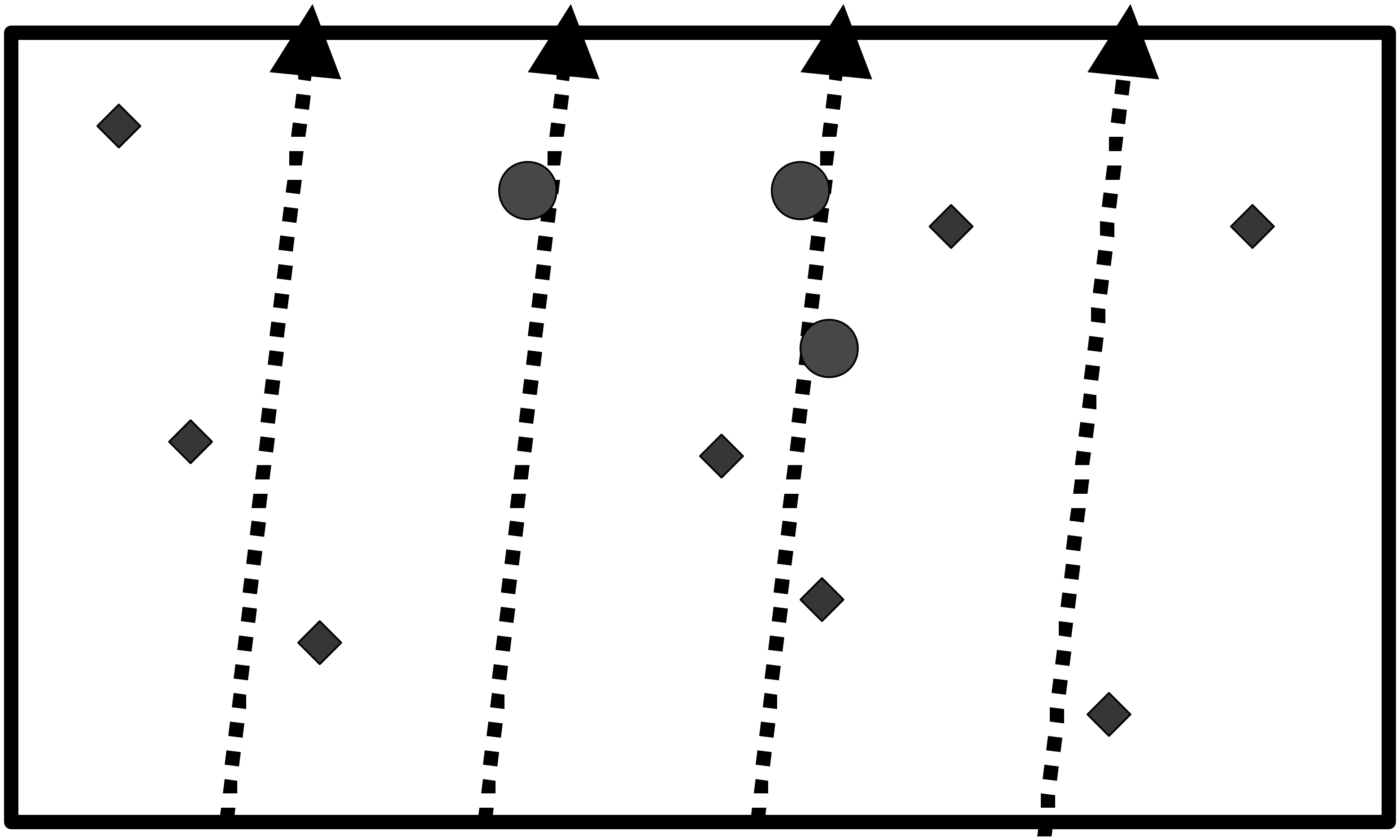
Testing Analogy: Clearing Weeds



◆ weeds

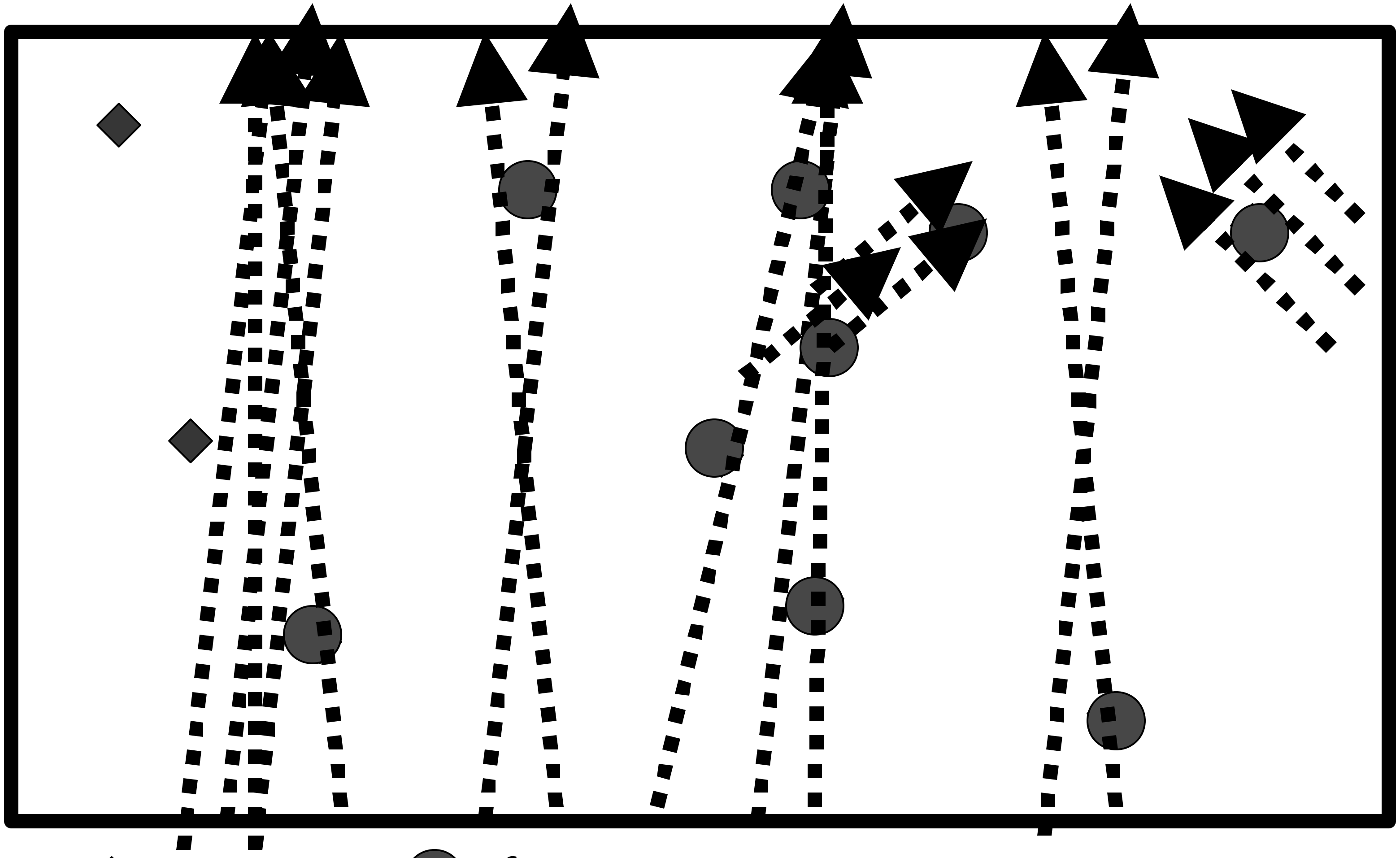
Thanks to James Bach for letting us use his slides.

Totally repeatable tests won't clear the weeds



◆ weeds ● fixes

Variable Tests are Often More Effective



◆ weeds ● fixes

Why Are Regression Tests Weak?

- **Does the same thing over and over**
- **Most defects are found during test creation**
- **Software doesn’t break or wear out**
- **Any other test is equally likely to stumble over unexpected side effects**
- **Automation reduces test variability**
- **Only verifies things programmed into the test**

Regression Testing: Some Papers of Interest

Brian Marick's, *How Many Bugs Do Regression Tests Find?* presents some interesting data on regression effectiveness.

Brian Marick's *Classic Testing Mistakes* raises several critical issues in software test management, including further questions of the places of regression testing.

Cem Kaner, Avoiding Shelfware: A Manager's View of Automated GUI Testing

Software Test Automation:

Foundational Concepts

Automation of Tests

Common Mistakes about Test Automation

The paper (Avoiding Shelfware) lists 19 “Don’ts.” For example,

Don’t expect to be more productive over the short term.

- The reality is that most of the benefits from automation don’t happen until the second release.
- It takes 3 to 10+ times the effort to create an automated test than to just manually do the test. Apparent productivity drops at least 66% and possibly over 90%.
- Additional effort is required to create and administer automated test tools.

Test Automation is Programming

Win NT 4 had 6 million lines of code, and 12 million lines of test code

Common (and often vendor-recommended) design and programming practices for automated testing are appalling:

- **Embedded constants**
- No modularity
- ***No source control***
- No documentation
- `No requirements analysis`

No wonder we fail

Designing Good Automated Tests

- **Start with a known state**
- **Design variation into the tests**
- **Check for errors**
  - Put your analysis into the test itself
  - Capture information when the error is found (not later)
- **Don’t encourage error masking or error cascades**

Start With a Known State

Data

- Load preset values in advance of testing
- Reduce dependencies on other tests

Program State

- External view
- Internal state variables

Environment

- Decide on desired controlled configuration
- Capture relevant session information

Design Variation Into the Tests

- **Dumb monkeys**
- **Variations on a theme**
- **Configuration variables**
- **Data driven tests**
- **Pseudo-random event generation**
- **Model driven automation**

Check for Errors

- **Put checks into the tests**
- **Document expectations in the tests**
- **Gather information as soon as a deviation is detected**
  - Results
  - Other domains
- **Check as many areas as possible**

Error Masks and Cascades

- **Session runs a series of tests**
- **A test fails to run to normal completion**
  - Error masking occurs if testing stops
  - Error cascading occurs if one or more downstream tests fails as a consequence
- **Impossible to avoid altogether**
- **Should not design automated tests that unnecessarily cause either**

Good Test Case Design: Make Program Failures Obvious

Important failures have been missed because they weren't noticed after they were found.

Some common strategies:

- **Show expected results.**
- **Only print failures.**
- **Log failures to a separate file.**
- **Keep the output simple and well formatted.**
- **Automate comparison against known good output.**

Refer to Testing Computer Software, pages 125, 160, 161-164

Notes

Some Simple Automation Approaches

Getting Started With Automation of Software Testing

Six Sometimes-Successful "Simple" Automation Architectures

- Quick & dirty
- Equivalence testing
- Frameworks
- Real-time simulator with event logs
- Simple Data-driven
- Application-independent data-driven

Quick & Dirty

- Smoke tests
- Configuration tests
- Variations on a theme
- Stress, load, or life testing

Equivalence Testing

- **A/B comparison**
- **Random tests using an oracle (Function Equivalence Testing)**
- **Regression testing is the weakest form**

Framework-Based Architecture

Frameworks are code libraries that separate routine calls from designed tests.

- modularity
- reuse of components
- hide design evolution of UI or tool commands
- partial salvation from the custom control problem
- independence of application (the test case) from user interface details (execute using keyboard? Mouse? API?)
- important utilities, such as error recovery

For more on frameworks, see Linda Hayes' book on automated testing, Tom Arnold's book on Visual Test, and Mark Fewster & Dorothy Graham's excellent new book "Software Test Automation."

Real-time Simulator

- **Test embodies rules for activities**
- **Stochastic process**
- **Possible monitors**
    - Code assertions
    - Event logs
    - State transition maps
    - Oracles

Data-Driven Architecture

In test automation, there are (at least) three interesting programs:

- The software under test (SUT)
- The automation tool that executes the automated test code
- The test code (test scripts) that define the individual tests

From the point of view of the automation software, we can assume

- The SUT's variables are data
- The SUT's commands are data
- The SUT's UI is data
- The SUT's state is data
- The test language syntax is data

Therefore it is entirely fair game to treat these implementation details of the SUT as values assigned to variables of the automation software.

Additionally, we can think of the externally determined (e.g. determined by you) test inputs and expected test results as data.

Additionally, if the automation tool's syntax is subject to change, we might rationally treat the command set as variable data as well.

Data-Driven Architecture

In general, we can benefit from separating the treatment of one type of data from another with an eye to:

- optimizing the maintainability of each
- optimizing the understandability (to the test case creator or maintainer) of the link between the data and whatever inspired those choices of values of the data
- minimizing churn that comes from changes in the UI, the underlying features, the test tool, or the overlying requirements

You store and display the different data can be in whatever way is most convenient for you

Table Driven Architecture: Calendar Example

Imagine testing a calendar-making program.

The look of the calendar, the dates, etc., can all be thought of as being tied to physical examples in the world, rather than being tied to the program. If your collection of cool calendars wouldn’t change with changes in the UI of the software under test, then the test data that define the calendar are of a different class from the test data that define the program’s features.

- **Define the calendars in a table. This table should not be invalidated across calendar program versions. Columns name features settings, each test case is on its own row.**
- **An interpreter associates the values in each column with a set of commands (a test script) that execute the value of the cell in a given column/row.**
- **The interpreter itself might use “wrapped” functions, i.e. make indirect calls to the automation tool’s built-in features.**

Calendar Example

		Year	Start Month	Number of Months	Page Size	Page Orientation	Monthly Title	Title Font Name	Title Font Size	Picture Location	Picture File Type	Days per Week	Week Starts On	Date Location	Language	

Data-Driven Architecture: Calendar Example

This is a good design from the point of view of optimizing for maintainability because it separates out four types of things that can vary independently:

- ***The descriptions of the calendars themselves come from real-world and can stay stable across program versions.***
- ***The mapping of calendar element to UI feature will change frequently because the UI will change frequently. The mappings (one per UI element) are written as short, separate functions that can be maintained easily.***
- ***The short scripts that map calendar elements to the program functions probably call sub-scripts (think of them as library functions) that wrap common program functions. Therefore a fundamental change in the software under test might lead to a modest change in the program.***
- ***The short scripts that map calendar elements to the program functions probably also call sub-scripts (library functions) that wrap functions of the automation tool. If the tool syntax changes, maintenance involves changing the wrappers' definitions rather than the scripts.***

Data Driven Architecture

Note with the calendar example:

- we didn’t run tests twice
- we automated execution, not evaluation
- we saved SOME time
- we focused the tester on design and results, not execution.

Other table-driven cases:

- automated comparison can be done via a pointer in the table to the file
- the underlying approach runs an interpreter against table entries

Hans Buwalda and others use this to create a structure that is natural for non-tester subject matter experts to manipulate.

Application-Independent Data-Driven

- Generic tables of repetitive types
- Rows for instances
- Automation of exercises

Reusable Test Matrices

Test Matrix for a Numeric Input Field

Additional Instructions:

		Nothing	Valid value	At LB of value	At UB of value	At LB of value - 1	At UB of value + 1	Outside of LB of value	Outside of UB of value	0	Negative	At LB number of digits or chars	At UB number of digits or chars	Empty field (clear the default value)	Outside of UB number of digits or chars	Non-digits	Wrong data type (e.g. decimal into integer)	Expressions	Space

Notes

Think About:

- Automation is software development.
- Regression automation is expensive and can be inefficient.
- Automation need not be regression--you can run new tests instead of old ones.
- Maintainability is essential.
- Design to your requirements.
- Set management expectations with care.

Automation Architecture and High-Level Design

What Is Software Architecture?

"As the size and complexity of software systems increase, the design and specification overall system structure become more significant issues than the choice of algorithms and data structures of computation. Structural issues include the organization of a system as a composition of components; global control structures; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives. This is the *software architecture* level of design."

"Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among those components. Such a system may in turn be used as a (composite) element in a larger design system."

Software Architecture, M. Shaw & D. Garlan, 1996, p.1.

What Is Software Architecture?

"The quality of the architecture determines the conceptual integrity of the system. That in turn determines the ultimate quality of the system. Good architecture makes construction easy. Bad architecture makes construction almost impossible."

- Steve McConnell, *Code Complete*, p 35; see 35-45

"We've already covered some of the most important principles associated with the design of good architectures: coupling, cohesion, and complexity. But what really goes into making an architecture good? The essential activity of architectural design . . . is the partitioning of work into identifiable components. . . . Suppose you are asked to build a software system for an airline to perform flight scheduling, route management, and reservations. What kind of architecture might be appropriate? The most important architectural decision is to separate the business domain objects from all other portions of the system. Quite specifically, a business object should not know (or care) how it will be visually (or otherwise) represented . . ."

- Luke Hohmann, *Journey of the Software Professional: A Sociology of Software Development*, 1997, p. 313. See 312-349

Automation Architecture

1. **Model the SUT in its environment**
2. Determine the goals of the automation and the capabilities needed to achieve those goals
3. Select automation components
4. Set relationships between components
5. Identify locations of components and events
6. Sequence test events
7. **Describe automation architecture**

Issues Faced in A Typical Automated Test

- What is being tested?
- How is the test set up?
- Where are the inputs coming from?
- What is being checked?
- Where are the expected results?
- How do you know pass or fail?

Automated Software Test Functions

- **Automated test case/data generation**
- **Test case design from requirements or code**
- **Selection of test cases**
- **No intervention needed after launching tests**
- **Set-up or records test environment**
- **Runs test cases**
- **Captures relevant results**
- **Compares actual with expected results**
- **Reports analysis of pass/fail**

Hoffman's Characteristics of "Fully Automated" Tests

- **A set of tests is defined and will be run together.**
- **No intervention needed after launching tests.**
- **Automatically sets-up and/or records relevant test environment.**
- **Obtains input from existing data files, random generation, or another defined source.**
- **Runs test exercise.**
- **Captures relevant results.**
- **Evaluates actual against expected results.**
- **Reports analysis of pass/fail.**

Not all automation is full automation.
Partial automation can be very useful.

Key Automation Factors

- **Components of SUT**
  - Important features, capabilities, data
- **SUT environments**
  - O/S versions, devices, resources, communication methods, related processes
- **Testware elements**
  - Available hooks and interfaces
    - » Built into the software
    - » Made available by the tools
  - Access to inputs and results
- **Form of inputs and results**
  - Available bits and bytes
  - Unavailable bits
  - Hard copy or display only

Functions in Test Automation

Here are examples of automated test tool capabilities:

- Analyze source code for bugs
- Design test cases
- Create test cases (from requirements or code)
- Generate test data
- Ease manual creation of test cases
- Ease creation/management of traceability matrix
- Manage testware environment
- Select tests to be run
- Execute test scripts
- Record test events
- Measure software responses to tests (Discovery Functions)
- Determine expected results of tests (Reference Functions)
- Evaluate test results (Evaluation Functions)
- Report and analyze results

Capabilities of Automation Tools

Automated test tools combine a variety of capabilities. For example, GUI regression tools provide:

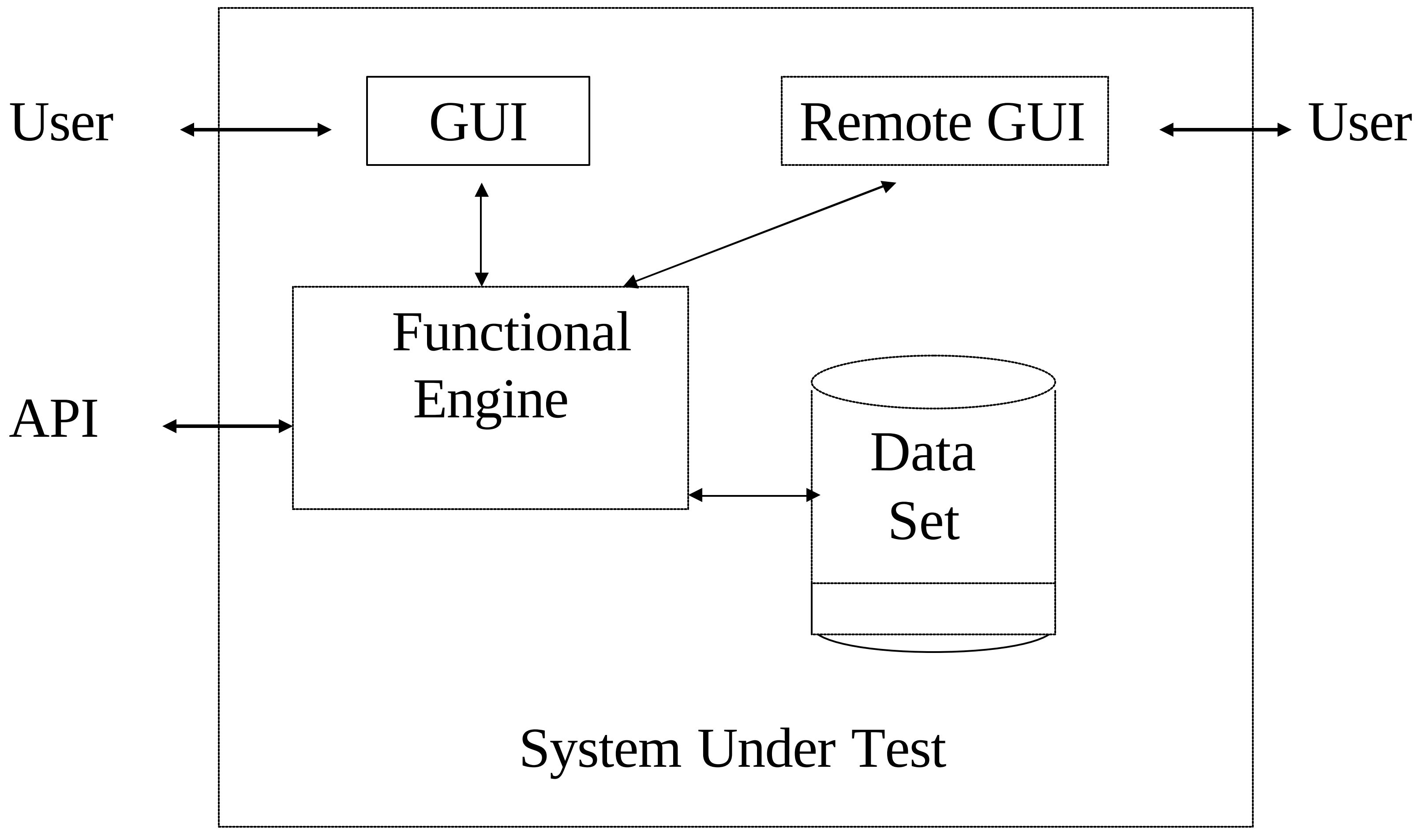
- capture/replay for easy manual creation of tests
- execution of test scripts
- recording of test events
- compare the test results with expected results
- report test results

Some GUI tools provide additional capabilities, but no tool does everything well.

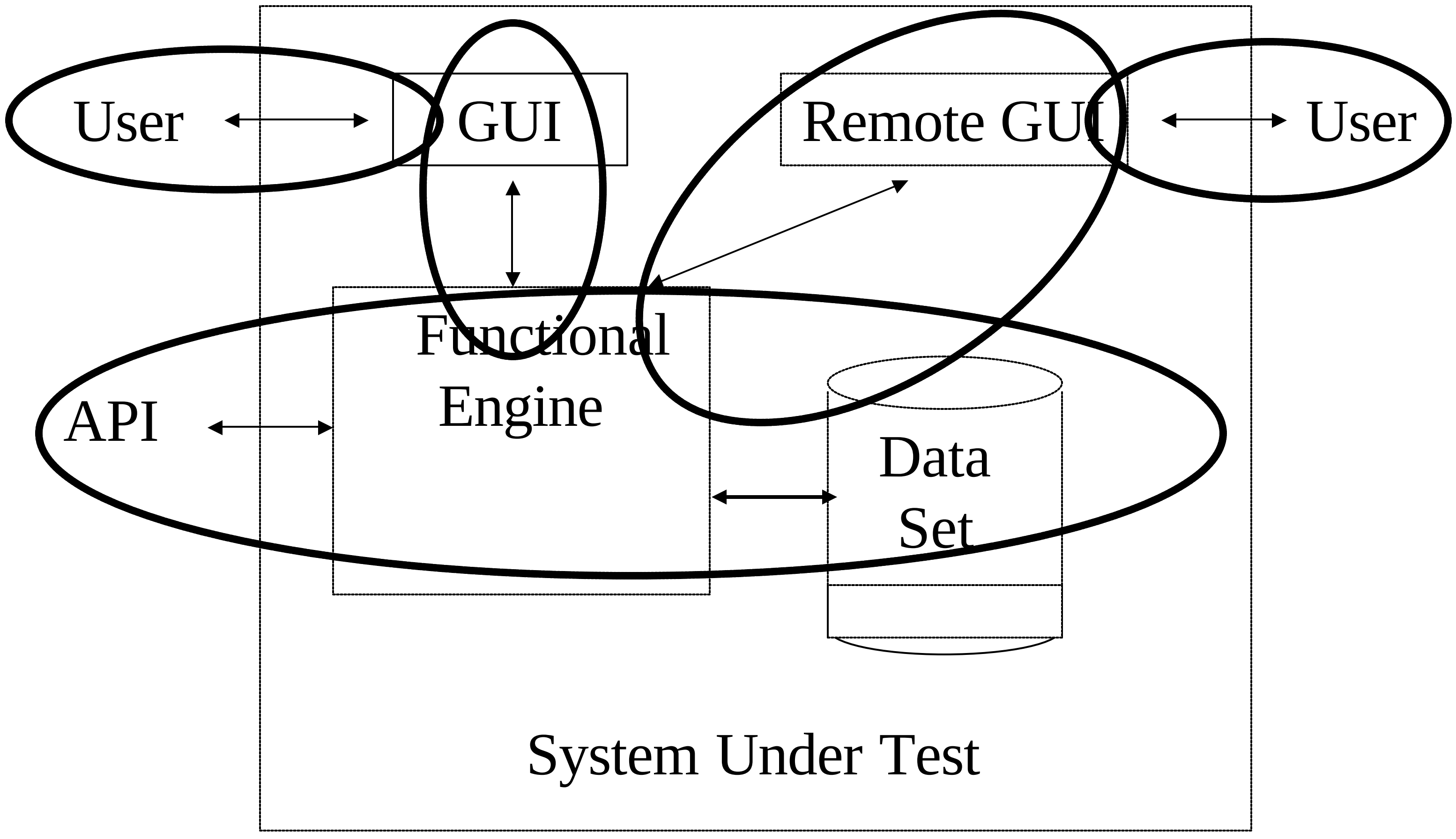
Tools for Improving Testability by ***Providing Diagnostic Support***

- **Hardware integrity tests.** Example: power supply deterioration can look like irreproducible, buggy behavior.
- **Database integrity.** Ongoing tests for database corruption, making corruption quickly visible to the tester.
- **Code integrity.** Quick check (such as checksum) to see whether part of the code was overwritten in memory.
- **Memory integrity.** Check for wild pointers, other corruption.
- **Resource usage reports**: Check for memory leaks, stack leaks, etc.
- **Event logs.** See reports of suspicious behavior. Probably requires collaboration with programmers.
- **Wrappers.** Layer of indirection surrounding a called function or object. The automator can detect and modify incoming and outgoing messages, forcing or detecting states and data values of interest.

An Example Model For SUT



Breaking Down The Testing Problem



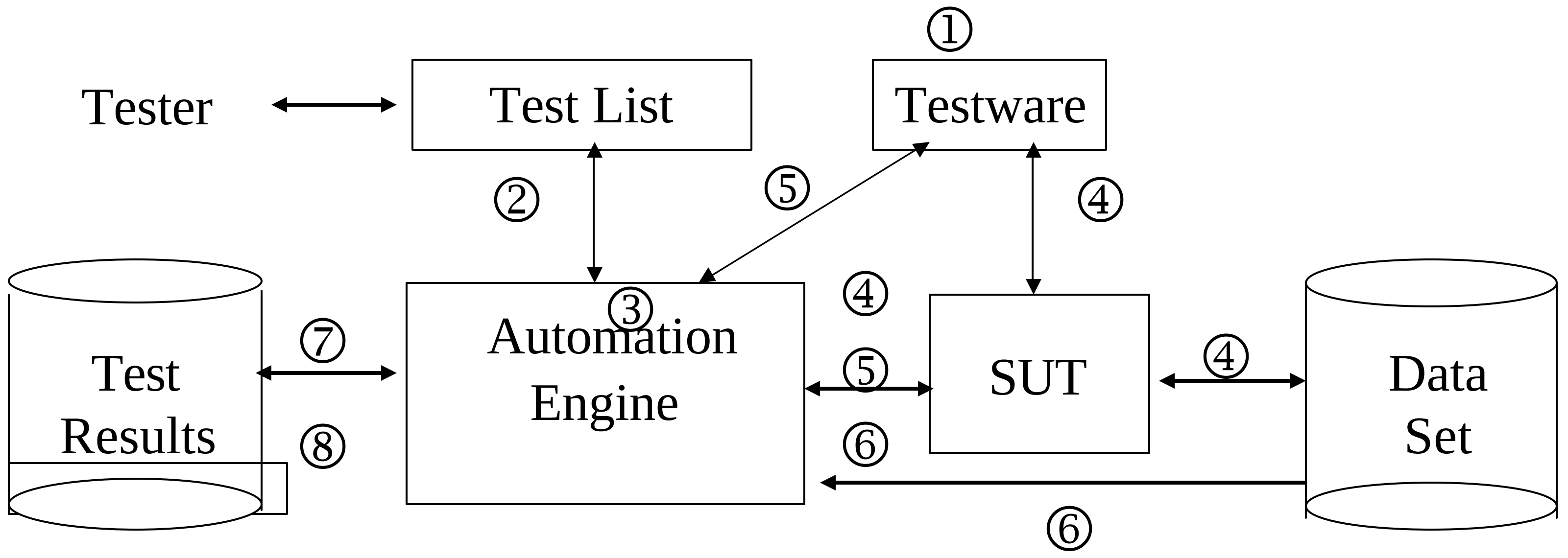
Identify Where To Monitor and Control

- **Natural break points**
- **Ease of automation**
- **Availability of oracles**
- **Leverage of tools and libraries**
- **Expertise within group**

Location and Level for Automating Testing

- **Availability of inputs and results**
- **Ease of automation**
- **Stability of SUT**
- **Project resources and schedule**
- **Practicality of Oracle creation and use**
- **Priorities for testing**

Automated Software Testing Process Model Architecture



1. Testware version control and configuration management
2. Selecting the subset of test cases to run
3. Set-up and/or record environmental variables
4. Run the test exercises
5. Monitor test activities
6. Capture relevant results
7. Compare actual with expected results
8. Report analysis of pass/fail

Automation Design Process

1. List the sequence of automated events
2. Identify components involved with each event
3. Decide on location(s) of events
4. Determine flow control mechanisms
5. Design automation mechanisms

Making More Powerful Exercises

Increase the number of combinations

More frequency, intensity, duration

Increasing the variety in exercises

Self-verifying tests and diagnostics

Use computer programming to extend your reach

- Set conditions
- Monitor activities
- Control system and SUT

Random Selection Among Alternatives

Pseudo random numbers

Partial domain coverage

Small number of combinations

Use oracles for verification

Pseudo Random Numbers

Used for selection or construction of inputs

- With and without weighting factors
- Selection with and without replacement

Statistically "random" sequence

Randomly generated "seed" value

Requires oracles to be useful

Mutating Automated Tests

Closely tied to instrumentation and oracles

Using pseudo random numbers

Positive and negative cases possible

Diagnostic drill down on error

Mutating Tests Examples

Data base contents (Embedded)

Processor instruction sets (Consistency)

Compiler language syntax (True)

Stacking of data objects (None)

Architecture Exercise

There are two important architectures (Software Under Test and Automation Environment) to understand for good test automation. These may or may not be articulated in your organization.

So . . . .

Please take a piece of paper and sketch out what you think the automation (or SUT) architecture might look like in your environment.

Automation Architecture: Some Papers of Interest

Doug Hoffman, *Test Automation Architectures: Planning for Test Automation*

Doug Hoffman, *Mutating Automated Tests*

Cem Kaner & John Vokey: *A Better Random Number Generator for Apple's Floating Point BASIC*

John Kent, *Advanced Automated Testing Architectures*

Notes

Notes

Notes

Notes

Notes

Alternate Paradigms of Black Box Software Testing

This material was prepared jointly by Cem Kaner and James Bach.

We also thank Bob Stahl, Brian Marick, Hans Schaefer, and Hans Buwalda for several insights.

Automation Requirements Analysis

Automation requirements are not just about the software under test and its risks. To understand what we're up to, we have to understand:

- Software under test and its risks
- The development strategy and timeframe for the software under test
- How people will use the software
- What environments the software runs under and their associated risks
- What tools are available in this environment and their capabilities
- The regulatory / required record keeping environment
- The attitudes and interests of test group management.
- The overall organizational situation

Automation Requirements Analysis

Requirement: "Anything that drives design choices."

The paper (Avoiding Shelfware) lists 27 questions. For example,

Will the user interface of the application be stable or not?

- Let's analyze this. The reality is that, in many companies, the UI changes late.
- Suppose we're in an extreme case. Does that mean we cannot automate cost effectively? No. It means that we should do only those types of automation that will yield a faster return on investment.

Data Driven Architectures

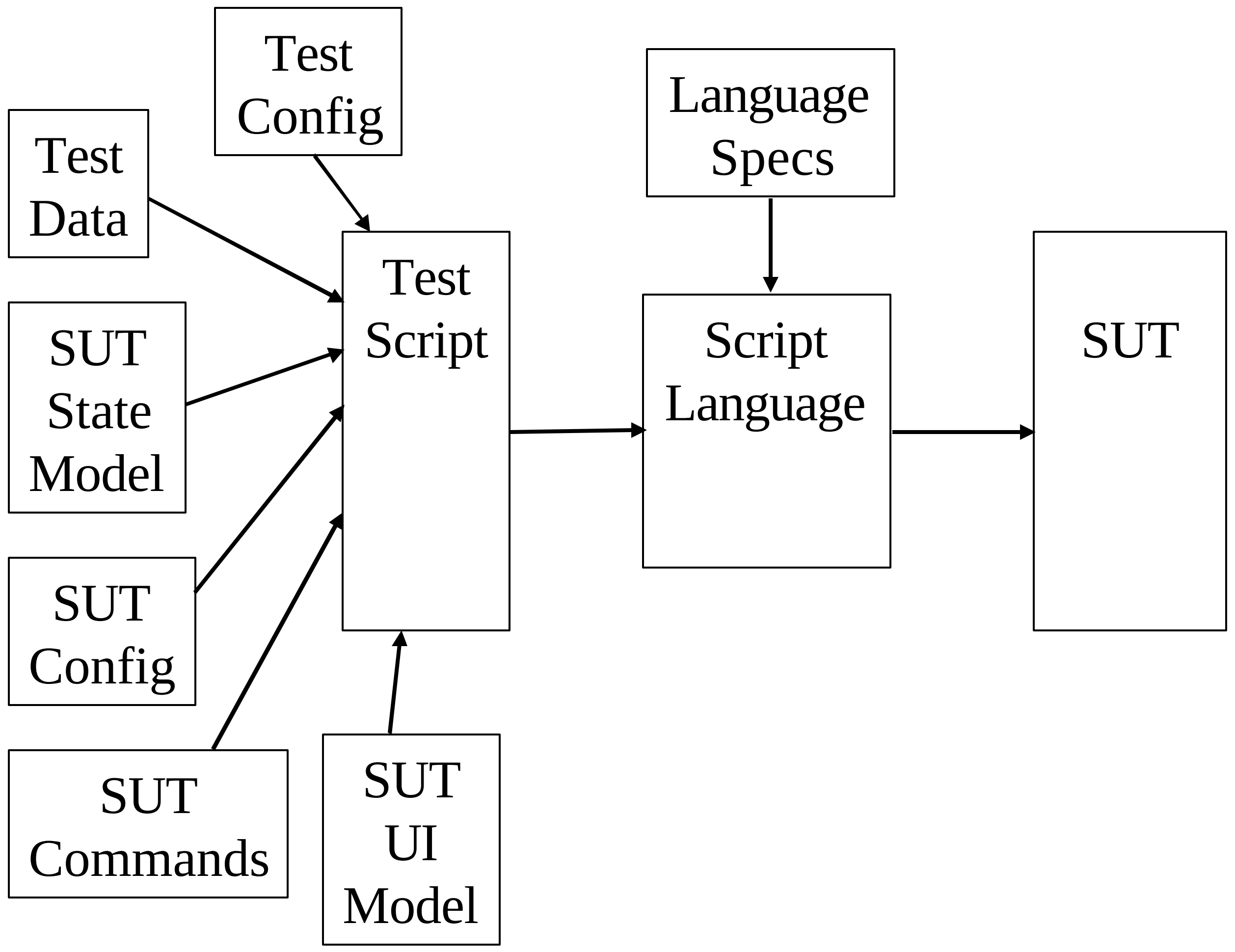
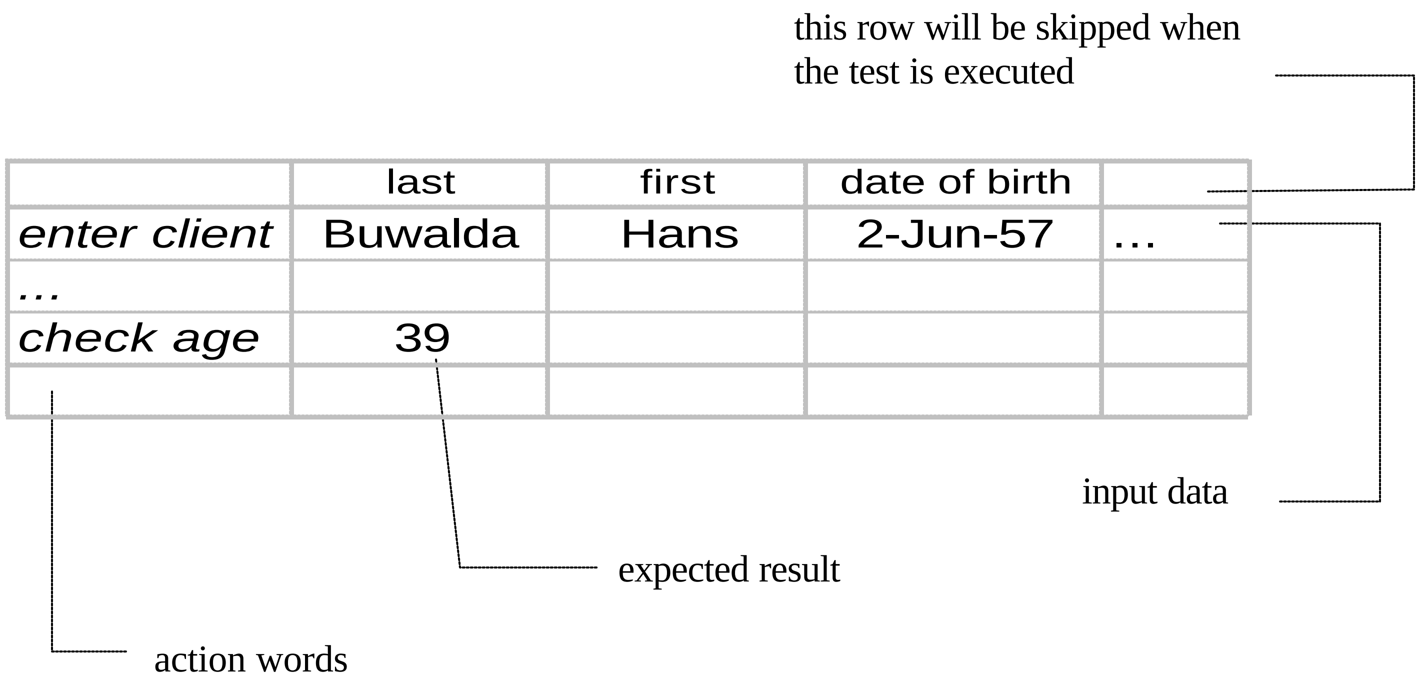


Table Driven Automation



	last	first	date of birth	
enter client	Buwalda	Hans	2-Jun-57	...
...				
check age	39			

Hans Buwalda, *Automated Testing with Action Words*

© CMG Finance BV

Tables: Another Script Format

Step #	**Check ?**	**What to do**	**What to see**	**Design notes**	**Observation notes**
1.	____	Pull down task menu	Task menu down	This starts the blah blah test, with the blah blah goal	

Capture Replay: A Nest of Problems

Methodological

- Fragile tests
- What is “close enough”
- Must prepare for user interface changes
- Running in different configurations and environments
- Must track state of software under test
- Hard-coded data limits reuse

Technical

- Playing catch up with new technologies
- Instrumentation is invasive
- Tools can be seriously confused
- Tools require customization and tuning
- Custom controls issues

Capture Replay: An Approach Without a Context

When could Capture Replay work?

- User interface is defined and frozen early
- Programmers use late-market, non-customized interface technologies

These situations are rare -

- There are very few reports of projects actually using Capture Replay successfully
  - » Monitoring exploratory sessions
  - » Quick and dirty configuration tests

Automated Test Paradigms

- **Regression testing**
- **Function/Specification-based testing**
- **Domain testing**
- **Load/Stress/Performance testing**
- **Scenario testing**
- **Stochastic or Random testing**

Automated Test Mechanisms

- **Regression approaches**
- **Grouped individual tests**
- **Load/Stress/Performance testing**
- **Model based testing**
- **Massive (stochastic or random) testing**

Regression Testing

- **Automate existing tests**
- **Add regression tests**
- **Results verification = file compares**
- **Automate all tests**
- **One technique for all**

Parochial and Cosmopolitan Views

Cosmopolitan View

- **Engineering new tests**
- **Variations in tests**
- **Outcome verification**
- **Extend our reach**
- **Pick techniques that fit**

Regression Testing

Tag line

- "Repeat testing after changes."

Fundamental question or goal

- Manage the risks that (a) a bug fix didn't fix the bug or (b) the fix (or other change) had a side effect.

Paradigmatic case(s)

- <u>Bug regression</u> (Show that a bug was not fixed)
- <u>Old fix regression</u> (Show that an old bug fix was broken)
- <u>General functional regression</u> (Show that a change caused a working area to break.)
- Automated GUI regression suites

Strengths

- Reassuring, confidence building, regulator-friendly

Regression Testing

Blind spots / weaknesses

- Anything not covered in the regression series.
- Repeating the same tests means not looking for the bugs that can be found by other tests.
- Pesticide paradox
- Low yield from automated regression tests
- Maintenance of this standard list can be costly and distracting from the search for defects.

Domain Testing

Tag lines

- "Try ranges and options."
- "Subdivide the world into classes."

Fundamental question or goal

- A stratified sampling strategy. Divide large space of possible tests into subsets. Pick best representatives from each set.

Paradigmatic case(s)

- Equivalence analysis of a simple numeric field
- Printer compatibility testing

Domain Testing

Strengths

- Find highest probability errors with a relatively small set of tests.
- Intuitively clear approach, generalizes well

Blind spots

- Errors that are not at boundaries or in obvious special cases.
- Also, the actual domains are often unknowable.

Function Testing

Tag line

- “Black box unit testing.”

Fundamental question or goal

- Test each function thoroughly, one at a time.

Paradigmatic case(s)

- Spreadsheet, test each item in isolation.
- Database, test each report in isolation

Strengths

- Thorough analysis of each item tested

Blind spots

- Misses interactions, misses exploration of the benefits offered by the program.

A Special Case: Exhaustive

Exhaustive testing involves testing all values within a given domain, such as:

- all valid inputs to a function
- compatibility tests across all relevant equipment configurations.

Generally requires automated testing.

This is typically oracle based and consistency based.

A Special Case: MASPAR Example

<u>MASPAR functions: square root tests</u>

- **32-bit arithmetic, built-in square root**
  - » **2^32 tests (4,294,967,296)**
  - » **65,536 processor configuration**
  - » **6 minutes to run the tests with the oracle**
  - » **Discovered 2 errors that were not associated with any obvious boundary (a bit was mis-set, and in two cases, this affected the final result).**

- **However:**
  - » **Side effects?**
  - » **64-bit arithmetic?**

Domain Testing: Interesting Papers

- Thomas Ostrand & Mark Balcer, *The Category-partition Method For Specifying And Generating Functional Tests*, Communications of the ACM, Vol. 31, No. 6, 1988.
- Debra Richardson, et al., *A Close Look at Domain Testing*, IEEE Transactions On Software Engineering, Vol. SE-8, NO. 4, July 1982
- Michael Deck and James Whittaker, *Lessons learned from fifteen years of cleanroom testing*. STAR '97 Proceedings (in this paper, the authors adopt boundary testing as an adjunct to random sampling.)

Domain Testing: Some Papers of Interest

Hamlet, Richard G. and Taylor, Ross, *Partition Testing Does Not Inspire Confidence*, Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, IEEE Computer Society Press, 206-215, July 1988

abstract = { Partition testing, in which a program's input domain is divided according to some rule and test conducted within the subdomains, enjoys a good reputation. However, comparison between testing that observes partition boundaries and random sampling that ignores the partitions gives the counterintuitive result that partitions are of little value. In this paper we improve the negative results published about partition testing, and try to reconcile them with its intuitive value. Partition testing is show to be more valuable than random testing only when the partitions are narrowly based on expected faults and there is a good chance of failure. For gaining confidence from successful tests, partition testing as usually practiced has little value.}

From the STORM search page: http://www.mtsu.edu/~storm/bibsearch.html

Stress Testing

Tag line

- "Overwhelm the product."

Fundamental question or goal

- Learn about the capabilities and weaknesses of the product by driving it through failure and beyond. What does failure at extremes tell us about changes needed in the program's handling of normal cases?

Paradigmatic case(s)

- Buffer overflow bugs
- High volumes of data, device connections, long transaction chains
- Low memory conditions, device failures, viruses, other crises.

Strengths

- Expose weaknesses that will arise in the field.
- Expose security risks.

Blind spots

- Weaknesses that are not made more visible by stress.

Stress Testing: Some Papers of Interest

Astroman66, *Finding and Exploiting Bugs* 2600

Bruce Schneier, *Crypto-Gram*, May 15, 2000

James A. Whittaker and Alan Jorgensen, *Why Software Fails*

James A. Whittaker and Alan Jorgensen, *How to Break Software*

Specification-Driven Testing

Tag line:

- “Verify every claim.”

Fundamental question or goal

- Check the product’s conformance with every statement in every spec, requirements document, etc.

Paradigmatic case(s)

- Traceability matrix, tracks test cases associated with each specification item.
- User documentation testing

Strengths

- Critical defense against warranty claims, fraud charges, loss of credibility with customers.
- Effective for managing scope / expectations of regulatory-driven testing
- Reduces support costs / customer complaints by ensuring that no false or misleading representations are made to customers.

Blind spots

- Any issues not in the specs or treated badly in the specs /documentation.

Specification-Driven Testing: Papers of Interest

Cem Kaner, *Liability for Defective Documentation*

Scenario Testing

Tag lines

- "Do something useful and interesting"
- "Do one thing after another."

Fundamental question or goal

- Challenging cases that reflect real use.

Paradigmatic case(s)

- Appraise product against business rules, customer data, competitors' output
- Life history testing (Hans Buwalda's "soap opera testing.")
- Use cases are a simpler form, often derived from product capabilities and user model rather than from naturalistic observation of systems of this kind.

Scenario Testing

The ideal scenario has several characteristics:

- It is realistic (e.g. it comes from actual customer or competitor situations).
- There is no ambiguity about whether a test passed or failed.
- The test is complex, that is, it uses several features and functions.
- There is an influential stakeholder who will protest if the program doesn't pass this scenario.

Strengths

- Complex, realistic events. Can handle (help with) situations that are too complex to model.
- Exposes failures that occur (develop) over time

Blind spots

- Single function failures can make this test inefficient.
- Must think carefully to achieve good coverage.

Scenario Testing: Some Papers of Interest

Hans Buwalda, *Testing With Action Words*

Hans Buwalda, *Automated Testing With Action Words, Abandoning Record & Playback*

Hans Buwalda on Soap Operas (in the conference proceedings of STAR East 2000)

Random / Statistical Testing

Tag line

- "High-volume testing with new cases all the time."

Fundamental question or goal

- Have the computer create, execute, and evaluate huge numbers of tests.
  - » The individual tests are not all that powerful, nor all that compelling.
  - » The power of the approach lies in the large number of tests.
  - » These broaden the sample, and they may test the program over a long period of time, giving us insight into longer term issues.

Random / Statistical Testing

Paradigmatic case(s)

- Some of us are still wrapping our heads around the richness of work in this field. This is a tentative classification
  - » **NON-STOCHASTIC [RANDOM] TESTS**
  - » **STATISTICAL RELIABILITY ESTIMATION**
  - » **STOCHASTIC TESTS (NO MODEL)**
  - » **STOCHASTIC TESTS USING A MODEL OF THE SOFTWARE UNDER TEST**
  - » **STOCHASTIC TESTS USING OTHER ATTRIBUTES OF SOFTWARE UNDER TEST**

Random Testing: Independent and Stochastic Approaches

Random Testing

- Random (or statistical or stochastic) testing involves generating test cases using a random number generator. Because they are random, the individual test cases are not optimized against any particular risk. The power of the method comes from running large samples of test cases.

Independent Testing

- For each test, the previous and next tests don’t matter.

Stochastic Testing

- Stochastic process involves a series of random events over time
  - » Stock market is an example
  - » Program typically passes the individual tests: The goal is to see whether it can pass a large series of the individual tests.

Random / Statistical Testing: Non-Stochastic

Fundamental question or goal

- The computer runs a large set of essentially independent tests. The focus is on the results of each test. Tests are often designed to minimize sequential interaction among tests.

Paradigmatic case(s)

- *Function equivalence testing:* Compare two functions (e.g. math functions), using the second as an oracle for the first. Attempt to demonstrate that they are not equivalent, i.e. that the achieve different results from the same set of inputs.
- Other test using fully deterministic oracles (see discussion of oracles, below)
- Other tests using heuristic oracles (see discussion of oracles, below)

Independent Random Tests: Function Equivalence Testing

Hypothetical case: Arithmetic in Excel

Suppose we had a pool of functions that worked well in a previous version.

For individual functions, generate random numbers to select function (e.g. log) and value in Excel 97 and Excel 2000.

- **Generate lots of random inputs**
- **Spot check results (e.g. 10 cases across the series)**

Build a model to combine random functions into arbitrary expressions

- **Generate and compare expressions**
- **Spot check results**

Notes

Random / Statistical Testing: Statistical Reliability Estimation

Fundamental question or goal

- Use random testing (possibly stochastic, possibly oracle-based) to estimate the stability or reliability of the software. Testing is being used primarily to qualify the software, rather than to find defects.

Paradigmatic case(s)

- Clean-room based approaches

Random Testing: Stochastic Tests--No Model: “Dumb Monkeys”

Dumb Monkey

- Random sequence of events
- Continue through crash (Executive Monkey)
- Continue until crash or a diagnostic event occurs. The diagnostic is based on knowledge of the system, not on internals of the code. (Example: button push doesn’t push—this is system-level, not application level.)

Random Testing: “Dumb Monkeys”

Fundamental question or goal

- High volume testing, involving a long sequence of tests.
- A typical objective is to evaluate program performance over time.
- The distinguishing characteristic of this approach is that the testing software does not have a detailed model of the software under test.
- The testing software might be able to detect failures based on crash, performance lags, diagnostics, or improper interaction with other, better understood parts of the system, but it cannot detect a failure simply based on the question, “Is the program doing what it is supposed to or not?”

Random Testing: "Dumb Monkeys"

Paradigmatic case(s)

- Executive monkeys: Know nothing about the system. Push buttons randomly until the system crashes.
- Clever monkeys: More careful rules of conduct, more knowledge about the system or the environment. See Freddy.
- O/S compatibility testing: No model of the software under test, but diagnostics might be available based on the environment (the NT example)
- Early qualification testing
- Life testing
- Load testing

Note:

- Can be done at the API or command line, just as well as via UI

Random / Statistical Testing: Stochastic, Assert or Diagnostics Based

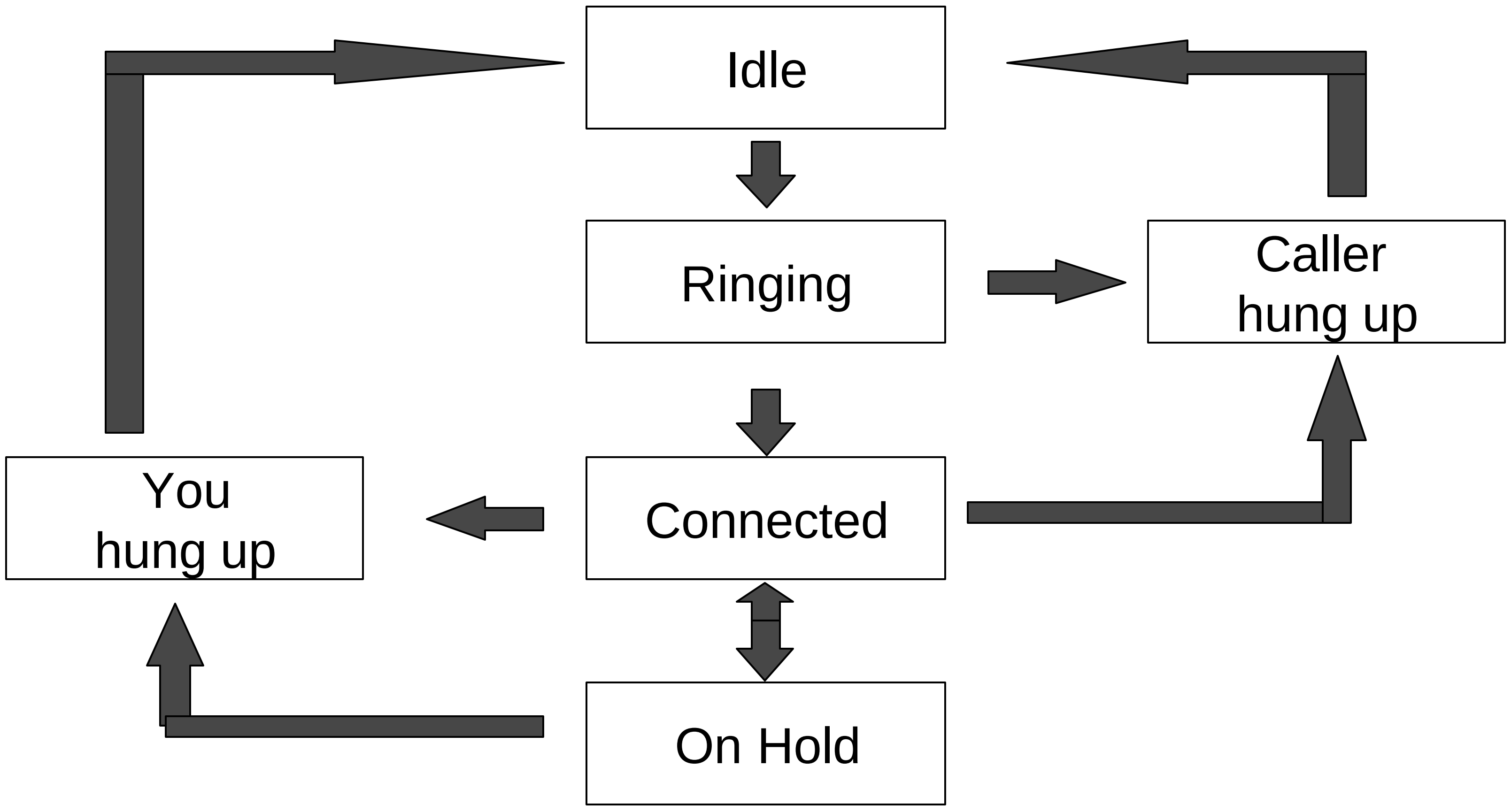
Fundamental question or goal

- High volume random testing using random sequence of fresh or pre-defined tests that may or may not self-check for pass/fail. The primary method for detecting pass/fail uses assertions (diagnostics built into the program) or other (e.g. system) diagnostics.

Paradigmatic case(s)

- Telephone example (asserts)
- Embedded software example (diagnostics)

The Need for Stochastic Testing: An Example



Refer to Testing Computer Software, pages 20-21

Stochastic Test Using Diagnostics

Telephone Sequential Dependency

- Symptoms were random, seemingly irreproducible crashes at a beta site
- All of the individual functions worked
- We had tested all lines and branches
- Testing was done using a simulator, that created long chains of random events. The diagnostics in this case were assert fails that printed out on log files

Random Testing: Stochastic, Regression-Based

Fundamental question or goal

- High volume random testing using random sequence of pre-defined tests that can self-check for pass/fail.

Paradigmatic case(s)

- Life testing
- Search for specific types of long-sequence defects.

Random Testing: Stochastic, Regression-Based

Notes

- Create a series of regression tests. Design them so that they don’t reinitialize the system or force it to a standard starting state that would erase history. The tests are designed so that the automation can identify failures. Run the tests in random order over a long sequence.
- This is a low-mental-overhead alternative to model-based testing. You get pass/fail info for every test, but without having to achieve the same depth of understanding of the software. Of course, you probably have worse coverage, less awareness of your actual coverage, and less opportunity to stumble over bugs.
- Unless this is very carefully managed, there is a serious risk of non-reproducibility of failures.

Random Testing: Sandboxing the Regression Tests

Suppose that you create a random sequence of standalone tests (that were not sandbox-tested), and these tests generate a hard-to-reproduce failure.

You can run a sandbox on each of the tests in the series, to determine whether the failure is merely due to repeated use of one of them.

Random Testing: Sandboxing

- In a random sequence of standalone tests, we might want to qualify each test, T1, T2, etc, as able to run on its own. Then, when we test a sequence of these tests, we know that errors are due to interactions among them rather than merely to cumulative effects of repetition of a single test.
- Therefore, for each Ti, we run the test on its own many times in one long series, randomly switching as many other environmental or systematic variables during this random sequence as our tools allow.
- We call this the "sandbox" series—Ti is forced to play in its own sandbox until it "proves" that it can behave properly on its own. (This is an 80/20 rule operation. We do want to avoid creating a big random test series that crashes only because one test doesn't like being run or that fails after a few runs under low memory. We want to weed out these simple causes of failure. But we don't want to spend a fortune trying to control this risk.)

Stochastic Test: Regression Based

<u>Testing with Sequence of Passed Tests</u>

- Collect a large set of regression tests, edit them so that they don’t reset system state.
- Randomly run the tests in a long series and check expected against actual results.
- Will sometimes see failures even though all of the tests are passed individually.

Random / Statistical Testing: Sandboxing the Regression Tests

In a random sequence of standalone tests, we might want to qualify each test, T1, T2, etc, as able to run on its own. Then, when we test a sequence of these tests, we know that errors are due to interactions among them rather than merely to cumulative effects of repetition of a single test.

Therefore, for each Ti, we run the test on its own many times in one long series, randomly switching as many other environmental or systematic variables during this random sequence as our tools allow. We call this the "sandbox" series—Ti is forced to play in its own sandbox until it "proves" that it can behave properly on its own. (This is an 80/20 rule operation. We just don't want to create a big random test series that crashes only because one test doesn't like being run one or a few times under low memory. We want to weed out these simple causes of failure.)

=============

In a random sequence of standalone tests (that were not sandbox-tested) that generate a hard-to-reproduce failure, run the sandbox on each of the tests in the series, to determine whether the failure is merely due to repeated use of one of them.

Notes

Random / Statistical Testing: Model-based Stochastic Tests

<u>The Approach</u>

- **Build a state model of the software. (The analysis will reveal several defects in itself.) For any state, you can list the actions the user can take, and the results of each action (what new state, and what can indicate that we transitioned to the correct new state).**
- **Generate random events / inputs to the program or a simulator for it**
- **When the program responds by moving to a new state, check whether the program has reached the expected state**

  - **See <u>www.geocities.com/model_based_testing/online_papers.htm</u>**

Random / Statistical Testing: Model-based Stochastic Tests

The Issues

- Works poorly for a complex product like Word
- Likely to work well for embedded software and simple menus (think of the brakes of your car or walking a control panel on a printer)
- In general, well suited to a limited-functionality client that will not be powered down or rebooted very often.

Random / Statistical Testing: Model-based Stochastic Tests

The applicability of state machine modeling to mechanical computation dates back to the work of Mealy [Mealy, 1955] and Moore [Moore, 1956] and persists to modern software analysis techniques [Mills, *et al*., 1990, Rumbaugh, *et al*., 1999]. Introducing state design into software development process began in earnest in the late 1980’s with the advent of the cleanroom software engineering methodology [Mills, *et al.*, 1987] and the introduction of the State Transition Diagram by Yourdon [Yourdon, 1989].

A deterministic finite automata (DFA) is a state machine that may be used to model many characteristics of a software program. Mathematically, a DFA is the quintuple, $M = (Q, S, d, q_0, F)$ where *M* is the machine, *Q* is a finite set of states, *S* is a finite set of inputs commonly called the “alphabet,” *d* is the transition function that maps *Q* x *S to Q,*, q_0 is one particular element of *Q* identified as the initial or stating state, and *F Í Q* is the set of final or terminating states [Sudkamp, 1988]. The DFA can be viewed as a directed graph where the nodes are the states and the labeled edges are the transitions corresponding to inputs.

When taking this state model view of software, a different definition of *software failure* suggests itself: “The machine makes a transition to an unspecified state.” From this definition of software failure a *software defect* may be defined as: “Code, that for some input, causes an unspecified state transition or fails to reach a required state.”

Alan Jorgensen, *Software Design Based on Operational Modes,* Ph.D. thesis, Florida Institute of Technology

Random / Statistical Testing: Model-based Stochastic Tests

…

Recent developments in software system testing exercise state transitions and detect invalid states. This work, [Whittaker, 1997b], developed the concept of an "operational mode" that functionally decomposes (abstracts) states. Operational modes provide a mechanism to encapsulate and describe state complexity. By expressing states as the cross product of operational modes and eliminating impossible states, the number of distinct states can be reduced, alleviating the state explosion problem.

Operational modes are not a new feature of software but rather a different way to view the decomposition of states. All software has operational modes but the implementation of these modes has historically been left to chance. When used for testing, operational modes have been extracted by reverse engineering.

Alan Jorgensen, *Software Design Based on Operational Modes,*
Ph.D. thesis, Florida Institute of Technology

Random / Statistical Testing: Thoughts Toward an Architecture

We have a population of tests, which may have been sandboxed and which may carry self-check info. A test series involves a sample of these tests.

We have a population of diagnostics, probably too many to run every time we run a test. In a given test series, we will run a subset of these.

We have a population of possible configurations, some of which can be set by the software. In a given test series, we initialize by setting the system to a known configuration. We may reset the system to new configurations during the series (e.g. every 5th test).

We have an execution tool that takes as input

- a list of tests (or an algorithm for creating a list),
- a list of diagnostics (initial diagnostics at start of testing, diagnostics at start of each test, diagnostics on detected error, and diagnostics at end of session),
- an initial configuration and
- a list of configuration changes on specified events.

The tool runs the tests in random order and outputs results

- to a standard-format log file that defines its own structure so that
- multiple different analysis tools can interpret the same data.

Random / Statistical Testing

Strengths

- Testing doesn’t depend on same old test every time.
- Partial oracles can find errors in young code quickly and cheaply.
- Less likely to miss internal optimizations that are invisible from outside.
- Can detect failures arising out of long, complex chains that would be hard to create as planned tests.

Blind spots

- Need to be able to distinguish pass from failure. Too many people think “Not crash = not fail.”
- Executive expectations must be carefully managed.
- Also, these methods will often cover many types of risks, but will obscure the need for other tests that are not amenable to automation.
- Testers might spend much more time analyzing the code and too little time analyzing the customer and her uses of the software.
- Potential to create an inappropriate prestige hierarchy, devaluating the skills of subject matter experts who understand the product and its defects much better than the automators.

Random Testing: Some Papers of Interest

Larry Apfelbaum, *Model-Based Testing*, Proceedings of Software Quality Week 1997 (not included in the course notes)

Michael Deck and James Whittaker, *Lessons learned from fifteen years of cleanroom testing*. STAR '97 Proceedings (not included in the course notes).

Doug Hoffman, *Mutating Automated Tests*

Alan Jorgensen, *An API Testing Method*

Noel Nyman, *GUI Application Testing with Dumb Monkeys*.

Harry Robinson, *Finite State Model-Based Testing on a Shoestring*.

Harry Robinson, *Graph Theory Techniques in Model-Based Testing*.

Paradigm Exercise

Do any of the paradigms listed reflect a dominant approach in your company? Which one(s)?

Looking at the paradigms as styles of testing, which styles are in use in your company? (List them from most common to least.)

Of the ones that are not common or not in use in your company, is there one that looks useful, that you think you could add to your company's repertoire? How?

Costs & Benefits of

Software Test Automation

Return on Investment?

Classic equation

$$E_n = A_a/A_m = (V_a + n*D_a)/ (V_m + n*D_m)$$

- Subscript “a” stands for automated, “m” stands for manual
- V_a: Expenditure for test specification and implementation
- V_m: Expenditure for test specification
- D_a: Expenditure for test interpretation after automated testing
- D_m: Expenditure for single, manual test execution
- n: number of automated test executions

Linz, T, Daigl, M. “GUI Testing Made Painless. Implementation and results of the ESSI Project Number 24306”, 1998.
Analysis in Case Study: Value of Test Automation Measurement, p. 52+ of Dustin, et. al., *Automated Software Testing*, Addison-Wesley, 1999

Return on Investment?

It is unrealistic to compare N automated test runs against the same number of manual test runs.

- Manual tests have built-in variance, and reruns of passed tests are weak.
- It can’t be five times as valuable to run an automated test daily as to run the same test manually once in a week.
- What should be compared is the number of times the automated test is run, and the actual cost of running it those times, versus the actual cost of running the manual test the number of times we would run it.

This doesn’t make automation look as good as an investment, but it better reflects actual value.

Falsely Expected Benefits

- All tests will be automated
- Immediate payback from automation
- Automation of existing manual tests
- Zero ramp up time
- Automated comprehensive test planning
- Capture/Play back for regression testing
- One tool that fits perfectly
- Automatic defect reporting (without human intervention)

Intangibles

Automation may have positive or negative effects on the following:

- Professionalism of the test organization
- Perceived productivity of the test organization
- Expansion into advanced test issues
- Quality of tests
- Willingness to experiment and change on the part of the test team
- Trust between testers and management
- Ability of the corporation to run many builds quickly through testing (e.g. for silent patch releases or localization testing)
- Testing coverage
- Residual ability of the test group to do exploratory testing

Conceptual Background

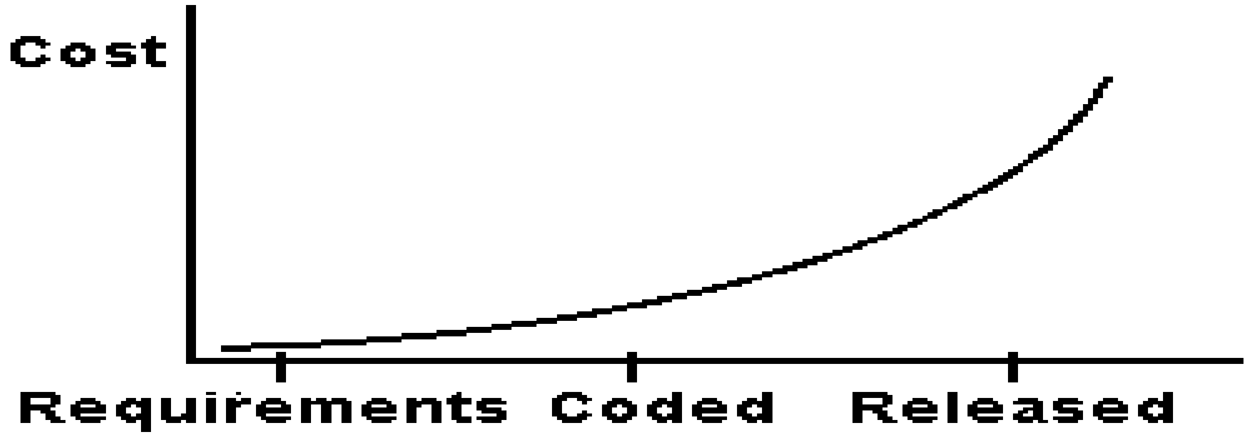
Time vs cost curve

Bugs found late are more expensive than bugs found early

The paradoxes of automation costing:

- Techniques to find bugs later that are cheaper are more expensive
- Techniques to find bugs earlier that are more expensive are cheaper

Cost vs time



Conceptual Background to Costing

Most automation benefits come from discipline in analysis and planning

Payback from automation is usually in the next project or thereafter

Automating usually causes significant negative schedule and performance impacts at introduction

Automated tests are more difficult to design and write, and require more programming and design skills from testers

Automated tests frequently require maintenance

Software metrics aren’t unbiased statistics

ROI Computations

$$E_n = A_a/A_m = (V_a + n*D_a) / (V_m + n*D_m)$$ †

$$E_n = A_a/A_m = (V_a + n_1*D_a) / (V_m + n_2*D_m)$$

$ROI_{automation}$(in time *t*) = (Savings from automation) / (Costs of automation)

$ROI_{automation}$(in time *t*) = Δ(Savings from automation) / Δ(Costs of automation)

† Linz, T, Daigl, M. "GUI Testing Made Painless. Implementation and results of the ESSI Project Number 24306", 1998. Analysis in Case Study: Value of Test Automation Measurement, p. 52+ of Dustin, et. al., *Automated Software Testing*, Addison-Wesley, 1999.

Costs and Benefits: Some Papers of Interest

Doug Hoffman, *Cost Benefits Analysis of Test Automation*

Linz, *GUI Testing Made Painless*

Brian Marick, *When Should a Test Be Automated?*

Test Oracles

The Test Oracle

Two slightly different views on the meaning of the word

- ***Reference Function:*** **You ask it what the "correct" answer is.** ***(This is how I use the term.)***
- ***Reference and Evaluation Function:*** **You ask it whether the program passed the test.**

Using an oracle, you can compare the program's result to a reference value (predicted value) and decide whether the program passed the test.

- ***Deterministic oracle*** **(mismatch means program fails)** ***(This is the commonly analyzed case.)***
- ***Probabilistic oracle*** **(mismatch means program probably fails.)** ***(Hoffman analyzes these in more detail.)***

Reference Functions: Some Typical Examples

Spreadsheet Version N and Version N-1

- Single function comparisons
- Combination testing
- What about revised functions?

Database management operations

- Same database, comparable functions across DBMs or query languages

Bitmap comparisons (output files)

- The problem of random variation

Deterministic Reference Functions

Saved result from a previous test.

Parallel function

- previous version
- competitor
- standard function
- custom model

Inverse function

- mathematical inverse
- operational inverse (e.g. split a merged table)

Useful mathematical rules (e.g. $\sin^2(x) + \cos^2(x) = 1$)

Expected result encoded into data

Test Result Possibilities

Situation / Test Results	No Error	Error
As Expected	Correct	Missed It
Red Flag	False Alarm	Caught

True Oracle Example

Simulator

Separate Implementation

Situation / Test Results	No Error	Error
As Expected	Correct	Missed It
Red Flag	False Alarm	Caught

Incomplete Oracle Example 1

Zip Code check of 5/9 digits

$\text{Sine}^2(x) = 1 - \text{Cosine}^2(x)$

Situation / Test Results	No Error	Error
As Expected	Correct	Missed It
Red Flag	False Alarm	Caught

Incomplete Oracle Example 2

Profile of Orders by Zip Code

Filter Testing (round-tripping)

Situation / Test Results	No Error	Error
As Expected	Correct	Missed It
Red Flag	False Alarm	Caught

Incomplete Oracle Example 3

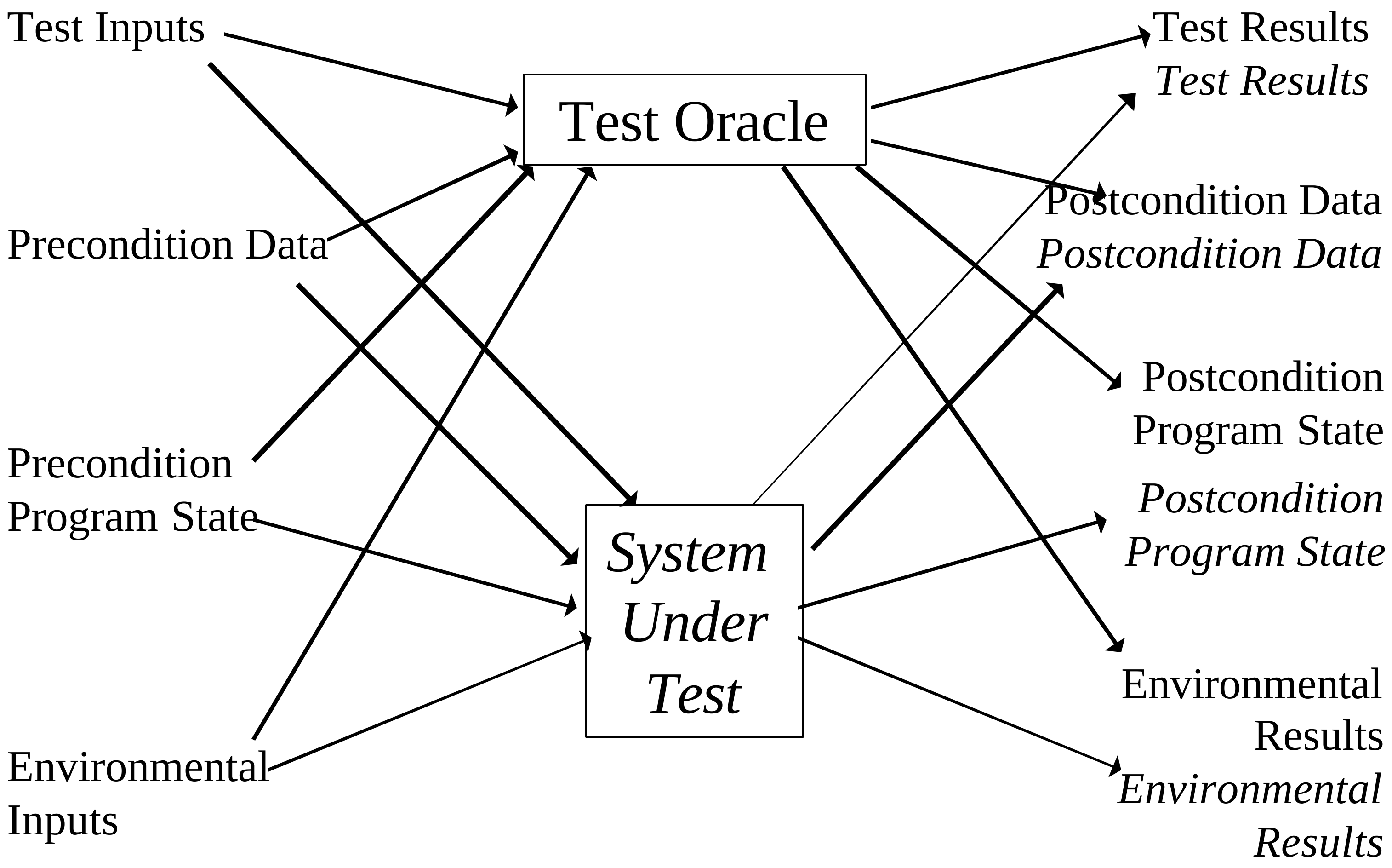
Age Checking

Test Results \ Situation	No Error	Error
As Expected	Correct	Missed It
Red Flag	False Alarm	Caught

Oracles: Challenges

- Completeness of information
- Accuracy of information
- Usability of the oracle or of its results
- Maintainability of the oracle
- May be as complex as SUT
- Temporal relationships
- Costs

A “Complete” Oracle



Notes

Oracle Completeness

- **Input Coverage**
- **Result Coverage**
- **Function Coverage**
- **Sufficiency**
- **Types of errors possible**
- **SUT environments**

May be more than one oracle for the SUT

Inputs may affect more than one oracle

Oracle Accuracy

How similar to SUT

- Arithmetic accuracy
- Statistically similar

How independent from SUT

- Algorithms
- Sub-programs & libraries
- System platform
- Operating environment

Close correspondence makes common mode faults more likely and reduces maintainability

How extensive

- The more ways in which the oracle matches the SUT, i.e. the more complex the oracle, the more errors

Types of possible errors

Oracle Usability

Form of information

- Bits and bytes
- Electronic signals
- Hardcopy and display

Location of information

Data set size

Fitness for intended use

Availability of comparators

Support in SUT environments

Oracle Maintainability

COTS or custom

- Custom oracle can become more complex than the SUT
- More complex oracles make more errors

Cost to keep correspondence through SUT changes

- Test exercises
- Test data
- Tools

Ancillary support activities required

Oracle Complexity

Correspondence with SUT

Coverage of SUT domains and functions

Accuracy of generated results

Maintenance cost to keep correspondence through SUT changes

- Test exercises
- Test data
- Tools

Ancillary support activities required

Temporal Relationships

- How fast to generate results
- How fast to compare
- When is the oracle run
- When are results compared

Oracle Costs

- **Creation or acquisition costs**
- **Maintenance of oracle and comparitors**
- **Execution cost**
- **Cost of comparisons**
- **Additional analysis of errors**
- **Cost of misses**
- **Cost of false alarms**

Evaluation Functions: Heuristics

Compare (apparently) sufficiently complete attributes

- compare calculated results of two parallel math functions (but ignore duration, available memory, pointers, display)

An almost-deterministic approach: Statistical distribution

- test for outliers, means, predicted distribution

Compare incidental but informative attributes

- durations

Check (apparently) insufficiently complete attributes

- ZIP Code entries are 5 or 9 digits

Check probabilistic attributes

- X is usually greater than Y

Results Comparison

Strategies

Comparison Functions

Data Comparisons (Oracle based)

- Previous version
- Competitor
- Standard function
- Custom model

Computational or Logical Modeling

- Inverse function
  - » mathematical inverse
  - » operational inverse (e.g. split a merged table)
- Useful mathematical rules (e.g. $\sin^2(x) + \cos^2(x) = 1$)

Oracle Strategies for Verification

	No Oracle	True Oracle	Consistency	Self Referential	Heuristic Strategy
Definition	-Doesn't check correctness of results, (only that some results were produced)	-Independent generation of all expected results	-Verifies current run results with a previous run (Regression Test)	-Embeds answer within data in the messages	-Verifies some values, as well as consistency of remaining values
Advantages	-Can run any amount of data (limited only by the time the SUT takes)	-No encountered errors go undetected	-Fastest method using an oracle -Verification is straightforward -Can generate and verify large amounts of data	-Allows extensive post-test analysis -Verification is based on message contents -Can generate and verify large amounts of complex data	-Faster and easier than True Oracle -Much less expensive to create and use
Disadvantages	-Only spectacular failures are noticed.	-Expensive to implement -Complex and often time-consuming when run	-Original run may include undetected errors	-Must define answers and generate messages to contain them	-Can miss systematic errors (as in *sine* wave example)

'No Oracle' Strategy

- **Easy to implement**
- **Tests run fast**
- **Only spectacular errors are noticed**
- **False sense of accomplishment**

“True” Oracle

Independent implementation

Complete coverage over domains

- Input ranges
- Result ranges

“Correct” results

Usually expensive

Consistency Strategy

A / B compare

Checking for changes

Regression checking

- Validated
- Unvalidated

Alternate versions or platforms

Foreign implementations

Consistency Strategy

Consistency-based testing involves comparing the results of today's test with a prior result. If the results match (are consistent), the program has "passed" the test.

Prior result can be from:

- Earlier version of SUT.
- Version of SUT on another platform.
- Alternate implementation (Oracle, Emulator, or Simulator).
- Alternative product.

More generally, A/B comparison where the set {B} is a finite set of saved reference data, not a program that generates results.

Typical case: Traditional automated regression test.

Consistency Example: Regression Automation

Run a test manually. If the program passes the test, automate it.

- Create a script that can replay the test procedure, create a reference file containing screen output or result data.
- Then rerun the script, and compare the results to the reference file.

Only becomes interesting when the results are different:

- Something was just fixed.
- Something is now broken.
- We’re comparing data that can validly change.

Self-Referential Strategies

Embed results in the data

Cyclic algorithms

Shared keys with algorithms

Self Verifying Results

1. Generate a coded identifier when the test data is created

2. Attach the identifier to the data

3. Verify data using the identifier

Simple SVD Example 1

Create a random name:

- Generate and save random number Seed (<u>*S*</u>)
- Use the first random value using RAND(<u>*S*</u>) as the Length (<u>*L*</u>)
- Generate random Name (<u>*N*</u>) with L characters
- Concatenate the Seed (<u>*S*</u>) to name

Simple SVD Example 1

Assume the Seed ($\underline{S}$) is 8 bytes, and
Name ($\underline{N}$) field is maximum of 128 characters
Generate a name with Length ($\underline{L}$) random characters
(a maximum of 120)

$$\text{Name} = \underbrace{\boxed{\ldots\ \underline{L}\ \text{Random characters}\ \ldots}\boxed{\text{8 character }\underline{S}}}_{\text{9 to 128 characters long}}$$

Simple SVD Example 1

To verify the names:

- Extract the 8 character *S*
- Use RAND(*S*) to generate the random name length *L*
- Generate random string *N'* of length *L*
- Compare the name *N* in the record to the new random string *N'*

Simple SVD Example 2

Create random data packets

- Generate Random values for
  - » Start ($\underline{S}$),
  - » Increment ($\underline{I}$), and
  - » Character count ($\underline{C}$)
- First data ($\underline{V}_1$) = $\underline{S}$
- Next data ($\underline{V}_{i+1}$) = $\text{Mod}_8(\underline{V}_i + \underline{I})$
- Generate until $\underline{V}_C = \text{Mod}_8((\underline{V}_{C-1}) + \underline{I})$

Simple SVD Example 2

To verify the data packets

- First data $\underline{V_1}$ => $\underline{S}$
- Next data $\text{Mod}_8(256 + \underline{V}_2 - \underline{V}_1)$ => $\underline{I}$
- Verify each next data $\underline{V}_i = \text{Mod}_8((\underline{V}_{i-1}) + \underline{I})$
- Count the number of values => $\underline{C}$
- Return values of Start ($\underline{S}$), Increment ($\underline{I}$), and Count of values ($\underline{C}$)

Non-Unique SVD Fields

- Shared value fields
  - last names
  - job titles
  - company
- Non-string data
  - numeric values
  - date fields
- Limited length
  - first name
  - state

Add a new field to the data set for each record

Self-Referential Oracle Examples

Data base

- embedded linkages

Data communications

- value patterns (start, increment, number of values)

Noel Nyman's "Self Verifying Data"*

Notes

Heuristics

"Heuristics are criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal. They represent compromises between two requirements: the need to make such criteria simple and, at the same time, the desire to see them discriminate correctly between good and bad choices.

"A heuristic may be a rule of thumb that is used to guide one's actions. For example, a popular method for choosing rip cantaloupe involves pressing the spot on the candidate cantaloupe where it was attached to the plant . . . This . . . Does not guarantee choosing only ripe cantaloupe, nor does it guarantee recognizing each ripe cantaloupe judged, but it is effective most of the time. . . .

"It is the nature of good heuristics both that they provide a simple means of indicating which of several courses of action is to be preferred, and that they are not necessarily guaranteed to identify the most effective course of action, but do so sufficiently often."

Judea Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving* (1984).

Heuristic Oracles

Heuristics are rules of thumb that support but do not mandate a given conclusion. We have partial information that will support a probabilistic evaluation. This won’t tell you that the program works correctly but it can tell you that the program is broken. This can be a cheap way to spot errors early in testing.

Example:

- History of transactions → Almost all transactions came from New York last year.
- Today, 90% of transactions are from Wyoming. Why? Probably (but not necessarily) the system is running amok.

Choosing / Using a Heuristic

Rules of thumb

- similar results that don’t always work
- low expected number of false errors, misses

Levels of abstraction

- General characteristics
- Statistical properties

Simplify

- use subsets
- break down into ranges
- step back (20,000 or 100,000 feet)
- look for harmonic patterns

Other relationships not explicit in SUT

- date/transaction number
- one home address
- employee start date

Strategy: Heuristic

Predict a characteristic and check it against a large random sample or a complete input or output domain. This won’t tell you that the program works correctly but it can tell you that the program is probably broken. (Note that most heuristics are prone to both Type I and Type II errors.) This can be a cheap way to spot errors early in testing.

- Check (apparently) insufficient attributes
  - ZIP Code entries are 5 or 9 digits
- Check probabilistic attributes
  - X is <u>usually</u> greater than Y
- Check incidental but correlated attributes
  - durations
  - orders
- Check consistent relationships
  - Sine similar to a sawtooth wave
  - $\sin(x)^2 + \cos(x)^2 = 1$

Heuristic Oracle Examples

Data base

- selected records using specific criteria
- selected characteristics for known records
- standard characteristics for new records
- correlated field values (time, order number)
- timing of functions

Data communications

- value patterns (start, increment, number of values)
- CRC

Sine function example

Heuristic Oracle Relationships

Nice

- follow heuristic rule for some range of values
- ranges are knowable
- few or no gaps

Predictable

- identifiable patterns

Simple

- easy to compute or identify
- require little information as input

Heuristic Oracle Drawbacks

- **Inexact**
  - will miss specific classes of errors
  - may miss gross systematic errors
  - don’t cover entire input/result domains
- **May generate false errors**
- **Can become too complex**
  - exception handling
  - too many ranges
  - require too much precision
- **Application may need better verification**

Where Do We Fit In The Oracle?

- **Identify what to verify**
- **How do we know the “right answer”**
- **How close to “right” do we need**
- **Decide when to generate the expected results**
- **Decide how and where to verify results**
- **Get or build an oracle**

Choosing an Oracle Strategy

- **Decide how the oracle fits in**
- **Identify the oracle characteristics**
- **Prioritize testing risks**
- **Watch for combinations of approaches**

Oracles:
Some Papers of Interest

Doug Hoffman, *Heuristic Test Oracles*

Doug Hoffman, *Oracle Strategies For Automated Testing*

Noel Nyman, *Self Verifying Data - Validating Test Results Without An Oracle*

Designing of Test Sets

A Classification Scheme for Test Sets

<u>Source of test cases</u>

- Old
- Intentionally new
- Random new

<u>Size of test pool</u>

- Small
- Large
- Exhaustive

<u>Serial dependence among tests</u>

- Independent
- Sequence is relevant

A Classification Scheme for Test Sets

Evaluation strategy

- Comparison to saved result
- Comparison to an oracle
- Comparison to a computational or logical model
- Comparison to a heuristic prediction.
  (NOTE: All oracles are heuristic.)
- Crash
- Diagnostic
- State model

Examples:

-

Regression Testing

Source of test cases

- **Old**

Size of test pool

- **Small**

Serial dependence among tests

- **Independent**

Evaluation strategy

- **Comparison to saved result**

Examples:

- **GUI based, Capture/Playback**

Independent Random Tests: ***Function Equivalence Testing***

Source of test cases

- **Random new**

Size of test pool

- **Large**

Serial dependence among tests

- **Independent**

Evaluation strategy

- **Comparison to an oracle**

Examples

- **Arithmetic in Excel**

Stochastic Test: Random Inputs

Source of test cases

- **Random new**

Size of test pool

- **Large**

Serial dependence among tests

- **Sequence is relevant**

Evaluation strategy

- **Crash or Diagnostics**

Examples

- **Dumb Monkeys**

Stochastic Test: Model Based

<u>Source of test cases</u>

- **Random new**

<u>Size of test pool</u>

- **Large, medium or small (different substrategies)**

<u>Serial dependence among tests</u>

- **Sequence is relevant**

<u>Evaluation strategy</u>

- **State model or crash**

<u>Examples</u>

- **Navigation through windows**

Stochastic Test: Saved Tests Based

Source of test cases

- **Old**

Size of test pool

- **Large**

Serial dependence among tests

- **Sequence is relevant**

Evaluation strategy

- **Saved results or Crash or Diagnostics**

Examples

- **Sandboxed tests**

Stochastic Test: Using Diagnostics

Source of test cases

- **Random new**

Size of test pool

- **Large**

Serial dependence among tests

- **Sequence is relevant**

Evaluation strategy

- **Diagnostics in code**

Examples

- **Telephone system Hold function**

Notes

Sorting it Out: Structure and Strategies of Automation

Strategies for Automation

The following slides are not complete. They are a structure for thinking about your situation. (For us, they are a work in progress, and we'll fill in new items as we think of them, but they will always be incomplete.)

Consider them in the context of the questions on the previous slides, and list:

- more of the relevant characteristics (ones relevant to your situation)
- more examples of the strategies (e.g. more heuristic rules, more items for consistency comparison, etc.)

Please note that factors that are favorable to one strategy or another are just that, "favorable." They might or might not be necessary and they are not sufficient. They simply push you in one direction or another.

Evaluation of Strategies for Automation

What characteristics of the

- goal of testing
- level of testing (e.g. API, unit, system)
- software under test
- environment
- generator
- reference function
- evaluation function
- users
- risks

would support, counter-indicate, or drive you toward

- consistency evaluation
- small sample, pre-specified values
- exhaustive sample
- random (a.k.a. statistical)
- heuristic analysis of a large set
- embedded, self-verifying data
- model-based testing

Favorable Conditions: Consistency

Goal of Testing

- Smoke testing
- Port or platform testing
- Demo to regulators
- Next version tests
-

Level of Testing (e.g. API, unit, system)

-

Software Under Test

- For a GUI-based test, uses standard controls, not custom controls.
- Hooks provided (e.g. API) for testing below the UI level.
- Stability of design / result set [if unstable, unsuitable for consistency testing].
- Must be repeatable output, e.g. postscript output and dithered output are unsuitable.
-

Environment

- Some embedded systems give non-repeatable results.
- Real time, live systems are usually not repeatable.
-

Favorable Conditions: Consistency

Generator

- Expensive to run tests in order to create reference data. Therefore it is valuable to generate test results once and use them from archives.
-

Reference Function

- captured screen, captured state, captured binary output file, saved database.
- duration of operation, amount of memory used, exiting state of registers, or other incidental results.
- finite set of reference data against which we can compare current behavior.
- It's nutty to compare 2 *screens* in order to see whether a sorted file compares to a previously sorted file. If you want to check the sorting, compare the files not the displays. Capture the essence of the things you want to compare.
-

Evaluation Function

-

Users

- Non-programmers are typical users (and are the normal targets of vendors of capture/playback tools).
-

Risks

- Tests a few things (sometimes well), does nothing with the rest.

Confounding Factors: Consistency

- The displayed (or printed) value may not be the same as that generated by the SUT. [Interface defects]
- Assumptions made may not be valid and need to be reconfirmed during and after testing.
- Smart tools limit visibility into actual SUT behaviors (smart tools –> less tester control).
- Small sample consistency testing -> see the discussion of automated regression testing weaknesses.
- Often mistaken for complete or true oracle comparisons.

Evaluation: Consistency

Advantages

- Straightforward.
- The program can serve as its own oracle.
- Easily used at an API.
- Effective when test cases are very expensive or when the software design is very stable.

Disadvantages

- Every time the software changes, tests that relied on that characteristic of the software must change.
- Unless the test code is carefully architected, the maintenance cost is impossibly high.
- Common mode of failure errors won’t be detected.
- Legacy errors won’t be detected.

Bottom line

-

Strategy: Small Sample

The small sample strategy is about limiting the number of tests used to exercise a product. Typically we use pre-specified values and compare results against some type of oracle.

Examples

	Small Sample	**Large Sample**
Unique	**Soap Opera**	**Stochastic**
Regression	**Silk / GUI regression**	**API-based tests; Function equivalence**

Examples: Small Sample

Equivalence and boundary analysis follow this approach. We divide a large population of possible tests into subsets and choose a few values that are representative of each set.

Scenario tests are often expensive and complex. Some companies create very few of them. In UI-intense situations, scenarios and exploratory tests might be manual. However, other applications are most naturally tested by writing code and creating sample data. Thus, an exploratory test or a one-use scenario test might be automated.

Favorable Conditions: Small Sample

Goal of Testing

- Destructive testing (can't test often if your test is designed to break the machine every time).
- Enormously long, repetitive test (too boring and tedious and time consuming to make a human run it even once).
-

Level of Testing (e.g. API, unit, system)

-

Software Under Test

- Regular function or any other input or output domain that is well-tested by a small group of representative values (such as boundary values).
-

Environment

- Environment or data cost high (e.g. Beizer's report of costs of Y2K time machine tests).
- High cost of renting machine.
- Mainframe (only have one, must share it with everyone else).
- Live system, can't feed much artificial data to it because you have to take it down each time or do special accounting stuff each time.
-

Favorable Conditions: Small Sample

Generator

- High cost to generate test cases (e.g., no automated generator).
-

Reference Function

- High cost to generate comparison data (e.g., no oracle).
- Huge comparison cost (e.g. the 1 terabyte database).
-

Evaluation Function

- Automated evaluation is slow, expensive.
-

Users

- Tolerant of errors.
- Intolerant of errors, but at a point at which we have done extensive function and domain testing and are now doing extremely complex tests, such as high-power soap operas.
-

Risks

- Low risk.

Evaluation: Small Sample

Advantages

- Can be fast to create and run.
- Identifies results of changes.
- Automation can be customized.
- Automated comparisons are straightforward.
- Product can be oracle for itself.

Disadvantages

- Saved results may contain unrecognized errors.
- Doesn’t necessarily consider specific, key data values, especially special cases not at visible boundaries.
- False security if domains are not correctly analyzed.
- Already-missed errors will remain undetected by repeated regression tests.
- If this testing is done before SW is cooked, then the code becomes tailored to the tests.

The fundamental problem is, it only checks a few values (we don’t know anything about the rest).

Favorable Conditions: Exhaustive

Goal of Testing

-

Level of Testing (e.g. API, unit, system)

-

Software Under Test

- Limited input domain.
-

Environment

- The range of environments is limited: embedded software or system configuration that is fully controlled by vendor.
- The important parameters (key elements of the environment) can be identified and are known.
-

Generator

- Easy to create tests.
-

Favorable Conditions: Exhaustive

Reference Function

- Oracle available.
-

Evaluation Function

- Evaluation function available.
-

Users

- ?
-

Risks

- Safety-critical or business-critical.
-

Evaluation: Exhaustive

Advantages

- Complete management of certain risks.
- Discover special case failures that are not visible at boundaries or suggested by traditional test design approaches.
-

Disadvantages

- Expensive.
- Often impossible.
-

Bottom line

Strategy: Random

Examples:

NON-STOCHASTIC RANDOM TESTS

- **Function Equivalence Testing.**
- **Data value generation using a statistical profile.**
- **Heuristic data profiles.**

STATISTICAL RELIABILITY ESTIMATION

- **Clean Room.**

STOCHASTIC TESTS (NO MODEL)

- **Dumb monkeys, such as early analysis of product stability, O/S compatibility testing.**

Strategy: Random

Examples (continued):

STOCHASTIC TESTS USING ON A MODEL OF THE SOFTWARE UNDER TEST

- **Random transition from state to state. Complex simulations, involving long series of events or combinations of many variables. Check whether the program has actually reached the expected state.**

STOCHASTIC TESTS USING OTHER ATTRIBUTES OF SOFTWARE UNDER TEST

- **Random transition from state to state. Complex simulations, involving long series of events or combinations of many variables. Check for assertion fails or other debug warning messages.**

Favorable Conditions: Random

Goal of Testing

- Load / life test
- Qualify embedded software (simple state machines that run for long periods)
- Statistical quality control.

Level of Testing (e.g. API, unit, system)

-

Software Under Test

- knockoff of a successful competitor
- upgrade from a working program
- conditions under test are very complex
-

Environment

-

Generator

- Random inputs through a generator function, such as creating random formulas for a spreadsheet

Favorable Conditions: Random

Reference Function

- Need *some* way to evaluate pass or fail. For example, compute the value of a formula from a reference spreadsheet.
-

Evaluation Function

- Must be available.
-

Users

-

Risks

- Significant errors that involve complex sequences of states or combinations of many inputs
-

Evaluation: Random

Advantages

- Can run a huge number of test cases
- Few or no evaluation errors

Disadvantages

- Doesn’t consider specific, key data values (no special allowance for boundaries, for example).

Risks

- People sometimes underestimate the need for a good oracle. They run so many tests that they think they are doing powerful work even though they are merely testing for crashes.
- Some of the random models generate sequences that make it impossible to reproduce a bug.
- Risk of false negatives (i.e. bug is missed when it is there) (oracle has same errors as software under test, so no bug is discovered)(see Leveson’s work on common mode errors).
- Risk of overestimating coverage--miss need for *other* types of tests to check for risks not tested for by this series of tests. E.G., might test individual functions but miss need to check combinations.

Favorable Conditions: Heuristic

Goal of Testing

- Early testing for plausibility of results

Level of Testing (e.g. API, unit, system)

-

Software Under Test

- Nice
  - » follows heuristic rule for some range of values
  - » ranges are knowable
  - » few or no gaps
- Predictable
  - » identifiable patterns
- Simple
  - » easy to compute or identify
  - » requires little information as input

Environment

Generator

-

Favorable Conditions: Heuristic

Reference Function

- **Usually there are multiple choices for oracles (can select "best" for the circumstances).**
-

Evaluation Function

-

Users

-

Risks

- **The risks that you manage by this type of testing are based on your knowledge of any testable fact about code or data that might be proved false by testing across a large set of data.**
-

Evaluation: Heuristic

Advantages

- May allow exhaustive testing of values of inputs (or results).
- Handy, powerful for detection early in testing
- Heuristic oracles are often reusable.

Disadvantages

- The results are not definitive.
  - » will miss specific classes of errors
  - » may miss gross systematic errors
  - » might not cover entire input/result domains
  - » may generate false errors
- Can become too complex
  - » exception handling
  - » too many ranges
  - » require too much precision
- Application may need better verification

Bottom line

- Handy, powerful for early detection, but should not be the only test type that you use.

Favorable Conditions: Embedded, SVD

Goal of Testing
- Independent verification after testing
-

Level of Testing (e.g. API, unit, system)
-

Software Under Test
- Persistent data
- Packetized data
-

Environment
-

Generator
-

Reference Function
- Inverse function

Evaluation Function
- Inverse function

Users
-

Risks
- Software subject to hidden errors identifiable from data
-

Evaluation: Embedded, SVD

Advantages

- Can uncover subtle side effects.
- Allows random data generators.
-

Disadvantages

- Some data types not conducive to SVD.
  - » dates
  - » numeric values
  - »
- May require additional data fields.

Bottom line

- Useful technique for some tests.

Favorable Conditions: Model Based

Goal of Testing

- Testing of a state machine
-

Level of Testing (e.g. API, unit, system)

-

Software Under Test

- Identified state machine model

Environment

-

Generator

- State machine based

Reference Function

- State machine model

Evaluation Function

- State verification

Users

-

Risks

- State transition errors possible

Evaluation: Model Based

Advantages

- Can find state transition errors.
- Allows random walks.
- Can be designed as generalized model tester
-

Disadvantages

- Only applicable to state based SUT.
- May require significant work to keep model in sync with SUT.
- Works poorly for a complex or changing product
-

Bottom line

- Useful technique for some SUT.

?

Testing Resources on the Net

Testing Resources on the Net
Various Web Sites

DOUG HOFFMAN'S HOME PAGE **www.SoftwareQualityMethods.com**

Consulting and training in strategy and tactics for software quality. Articles on software testing, quality engineering, and management. **Look here for updated links.**

CEM KANER'S HOME PAGE **www.kaner.com**

Articles on software testing and testing-related laws

JAMES BACH **www.satisfice.com**

Several interesting articles from one of the field's most interesting people.

BRETT PETTICHORD **www.io.com/~wazmo/qa.html**

Several interesting papers on test automation. Other good stuff too.

BRIAN LAWRENCE **www.coyotevalley.com**

Project planning from Brian Lawrence & Bob Johnson.

BRIAN MARICK **www.testing.com**

Brian Marick wrote an interesting series of papers for CenterLine. This particular one is a checklist before automating testing. The CenterLine site has a variety of other useful papers.

ELISABETH HENDRICKSON **www.QualityTree.com**

Consulting and training in software quality and testing.

JOHANNA ROTHMAN **www.jrothman.com**

Consulting in project management, risk management, and people management.

HUNG NGUYEN **www.logigear.com**

Testing services, training, and defect tracking products.

Testing Resources on the Net
Various Web Sites

LESSONS LEARNED IN SOFTWARE TESTING **www.testinglessons.com**

This is the home page for Cem', James', and Bret's book, *Lessons Learned in Software* Testing.

BAD SOFTWARE HOME PAGE **www.badsoftware.com**

This is the home page for Cem's book, *Bad Software*. Material on the law of software quality and software customer dissatisfaction.

SOFTWARE QUALITY ENGINEERING **www.sqe.com**

Several interesting articles on current topics.

SOFTWARE QUALITY ENGINEERING **www.stqe.com** **www.stickyminds.com**

Articles from STQE magazine, forum for software testing and quality engineering.

QA DUDE'S QUALITY INFO CENTER **www.dcez.com/~qadude**

"Over 200 quality links" -- pointers to standards organizations, companies, etc. Plus articles, sample test plans, etc.

QUALITY AUDITOR **www.geocities.com/WallStreet/2233/qa-home.htm**

Documents, links, listservs dealing with auditing of product quality.

THE OBJECT AGENCY **www.toa.com**

Ed Berard's site. Object-oriented consulting and publications. Interesting material.

RBSC (BOB BINDER) **www.rbsc.com**

A different approach to object-oriented development and testing.

DILBERT **www.unitedmedia.com/comics/dilbert.**

Home of Ratbert, black box tester from Heck.

Testing Resources on the Net
Various Web Sites

SSQA	**www.ventanatech.com/ssqa**

Silicon Valley Software Quality Association is a local professional software QA organization with monthly meetings, newsletter, more.

AMERICAN SOCIETY FOR QUALITY (ASQ)	**www.asq.org**

National/international professional QA organization.

SILICON VALLEY SECTION OF (ASQ)	**www.asq-silicon-valley.org**
ISO	**www.iso.ch**

Describes ISO (International Organization for Standardization), with links to other standards organizers

AMERICAN NATIONAL STANDARDS INSTITUTE	**www.ansi.org**
NSSN	**www.nssn.org**

National Standards Systems Network. Find / order various standards. Lots of links to standards providers, developers and sellers.

IEEE Computer Society	**www.computer.org**

Back issues of IEEE journals, other good stuff.

SOFTWARE ENGINEERING INSTITUTE	**www.sei.cmu.edu**

SEI at Carnegie Melon University. Creators of CMM and CMMI.

Testing Resources on the Net Various Web Sites

CENTER FOR SOFTWARE DEVELOPMENT	**www.center.org**

Non-profit in San Jose with a big test lab and various other support facilities.

RELIABLE SOFTWARE TECHNOLOGIES	**www.rstcorp.com**

Consulting firm. Hot stuff on software reliability and testability. Big archive of downloadable papers. Lots of pointers to other software engineering sites.

SOFTWARE TESTING INSTITUTE	**www.ondaweb.com/sti**

Membership-funded institute that promotes professionalism in the industry. BIG list of pointers to resources in the industry (the Online STI Resource Guide).

SOFTWARE PRODUCTIVITY CENTRE	**www.spc.ca**

Methodology, training and research center that supports software development in the Vancouver BC area.

CENTRE FOR SOFTWARE ENGINEERING	**www.cse.dcu.ie**

"Committed to raising the standards of quality and productivity within Ireland's software development community."

EUROPEAN SOFTWARE INSTITUTE	**www.esi.es**

Industry organization founded by leading European companies to improve the competitiveness of the European software industry. Very interesting for the Euromethod contracted software lifecycle and documents.

Testing Resources on the Net
Various Web Sites

QUALITY.ORG **www.casti.com**

Links to quality control source materials.

CSST (CLIENT-SERVER SOFTWARE TESTING) **www.cse.dcu.ie**

D. J. Mosley's home page. Lots of client / server publications.

FORMAL TECHNICAL REVIEW ARCHIVE **www.ics.Hawaii.edu/~johnson/FTR**

Documents, links, listservs dealing with auditing of product quality.

SOCIETY FOR TECHNICAL COMMUNICATION **www.stc.org**

Links to research material on documentation process and quality.

BUGNET **www.bugnet.com**

Lists of bugs and (sometimes) fixes. Great source for data.

TRY THIS SOMETIME -- With a product you own, look up bugs in BugNet that doesn't list a workaround or a bugfix release, and replicate it on your computer. Then call the publisher's tech support group and ask if they have an upgrade or a fix for this bug. Don't tell them that you found it in BugNet. The question is, what is the probability that your publisher's support staff will say, "Gosh, we've never heard of that problem before."

JPL TEST ENGINEERING LAB **tsunami.jpl.nasa.gov**

Jet Propulsion Lab's Test Engineering Laboratory. Includes the comp.software.testing archives.

Additionally, there are many sites for specialized work, such as sites for HTML compatibility tests of browsers. The usual search tools lead you to the key sites (the list changes weekly.)

Testing Resources on the Net Various Web Sites

SOFTWARE RESEARCH INC. **www.soft.com**

Also a major consulting and toolbuilding firm. Organizes the Quality Week conference. Publishes the TTN-Online newsletter. Excellent links.

QUALITY ASSURANCE INSTITUTE **www.qai.com**

Quality control focused on the needs of Information Technology groups.

AETG WEBSITE **http://aetgweb.argreenhouse.com/**

Home page for the AETG combinatorial testing product. Includes articles describing the theory of the product.

UNIFORM COMMERCIAL CODE ARTICLE 2B **www.law.upenn.edu/bll/ulc/ulc.htm**

These hold the drafts of the proposed Article 2B (UCITA), which will govern all sales of software. This will become the legal foundation of software quality. Currently, the foundation looks like it will be made of sand, Jell-O, and invisible ink.

SOFTWARE PUBLISHERS ASSOCIATION **www.spa.org**

Software & Information Industry Association is the main software publishers' trade association.

Testing Resources on the Net: Some News Groups

comp.software.testing

This covers issues of general interest to software testers. Plagued by too much spamming, this is not quite as interesting a spot as it used to be.

comp.human-factors

User interface issues, usability testing, safety, man-machine reliability, design tools.

comp.software.international

Internationalization and localization issues

comp.software.config-mgmt

Various configuration management issues.

comp.software-eng

Discussions of all areas of software engineering, including design, architecture, testing, etc. The comp.software-eng FAQ is the one that lists sources of bug tracking systems, for example. (You'd think it would be in comp.software.testing, but comp.software-eng got there first.)

comp.software.measurement

Not much there, but the goal is picking up metrics / measurements / data.

Testing Resources on the Net: Some News Groups

comp.answers

The home of many, many FAQs (documents that answer Frequently Asked Questions).

comp.jobs, comp.jobs.offered, ba.jobs, misc.jobs, etc.

A good way to check out market values for testers (or to find your new home when you need one).

comp. risks

Daily discussions of newsworthy bugs.

comp.dcom.modems

Lots and lots and lots of discussion of modems.

misc.industry.quality

Various discussions of quality control paradigms and experiences.

alt.comp.virus

This covers viruses, explaining things at end-user and more technical levels. The group often has very-up-to-date stuff. And like most alt-based groups, it seems to have a lot of spam mixed in. . .

Testing Resources on the Net Mailing Lists, etc.

swtest-discuss@convex.convex.com

> **Mail swtest-discuss-digest-request@rstcorp.com with "subscribe" in the body of the message to subscribe.**

baldrige, qs9000, many others

> ***contact* bill casti The Quality Czar <help@quality.org>**

DEMING-L

> ***contact* LISTSERV@UHCCVM.UHCC.HAWAII.EDU**

testing technology newsletter

> ***contact* ttn@soft.com, software research assocs**