Advanced Test Automation Architectures: Beyond Regression Testing

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

Software Quality Methods, LLC. (SQM)

www.SoftwareQualityMethods.com

doug.hoffman@acm.org

CAST 2007

Copyright Notice

The tutorial and these notes may include technical recommendations, but you are not Doug Hoffman's client and Doug is not providing specific advice in the notes or in the tutorial. Even if you ask questions about a specific situation, you must understand that you cannot possibly give enough information in a classroom setting to receive a complete, competent recommendation. I may use your questions as a teaching tool, and answer them in a way that I believe would "normally" be true but my answer could be completely inappropriate for your particular situation. I cannot accept any responsibility for any actions that you might take in response to my comments in this course.

The practices recommended and discussed in this course are useful for testing and test automation, but more experienced testers will adopt additional practices.

Acknowledgment

Much of the material included in the tutorial is based on courses co-developed with Cem Kaner. Without his contributions this tutorial would not exist.

Many of the ideas in this presentation were shared and refined in Los Altos Workshops on Software Testing, The Austin Workshops on Test Automation, and other derivative workshops.

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.

About Doug Hoffman

I see software quality as an engineering discipline and I try to promote the professional practice of software quality assurance by sharing my thoughts and my knowledge in the field.

Software quality assurance, and especially software testing, sometimes has an image 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, and instructor. 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 automation. I enjoy sharing what I've learned with interested people.

Current employment

- QA Program Manager, Hewlett-Packard
- President of Software Quality Methods, LLC. (SQM)
- Adjunct Instructor for UCSC Extension

Education

- MBA, Santa Clara University
- MS in Electrical Engineering, (digital design and information science) UCSB/UCSC
- B.A. in Computer Science, UCSB

Professional

- Fellow, American Society for Quality (ASQ)
- Past Chair, Silicon Valley Section of ASQ
- Founding Member and Past Chair, Santa Clara Valley Software Quality Association (SSQA, 1992-1997)
- Certified in Software Quality Engineering (ASQ-CSQE), ASQ-CQMgr
- Regular speaker at quality conferences
- Previously a Registered ISO 9000 Lead Auditor, (RAB 1993)
- I also actively participate in the Los Altos Workshops (and several others) on Software Testing

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 _____
- One Company _____
- Multiple comps _____

Tutorial Goals

- The relative strengths and weaknesses of manual and automated testing
- What makes a good automated test
- Architectures for automated oracles to establish pass/fail verdicts
- Advanced approaches for non-regression automated tests
- How to go about creating an automation architecture

Advantages of Automated Tests

- Discovery of defects manual testing cannot expose
- Monitoring of information not visible to a person
- Stimulation of the software under test (SUT) through APIs
- Massive numbers of actions not possible manually
- Pseudo random event and value generation to generate variation in the activities of the SUT
- Automated verification of very large numbers of outcomes and potential error conditions not directly associated with the functions being exercised

Outline

AM

- Foundational Concepts in Software Testing
- About Regression Testing
- Automation of Tests
- Test Oracles

PM

- Automation Architecture Approaches
- Advanced Comparison Strategies
- Automation Architectures and High Level Design

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.

Quick & Dirty Automation

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

Software Test Automation:

Foundational Concepts

Why To Automate

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)

Possible Missions for Testing

- Find important bugs fast
- Measure and document product quality
- Verify key features
- Keep up with development
- Reduce the risk of releasing with unknown errors
- 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
- Make tests repeatable

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
- Replicate routine tasks

You Can't Do This Manually!

Extending our reach

- API based testing
- Component testing
- Model based tests
- Data driven tests
- Internal monitoring and control
- Massive input generation
- Large scale result checking
- Use hooks and scaffolding

Multiplying our resources

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

The Mission of Test Automation

What is your test mission?

- What kind of risks are you trying to reduce?
- What kind of bugs are you looking for?
- What concerns are you addressing?
- Who are your shareholders?

Make automation serve your mission.

Expect your mission to change.

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)

Notes

Software Test Automation:

Foundational Concepts

Models and Testing

Why Automate Tests

The creative (and hardest) part of testing is designing good tests

- **All the rest are technical implementation details that can be automated (or not)**

- **Test automation should *reduce* the burden of the technical implementation and maintenance details so the humans can concentrate on creating excellent tests**

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

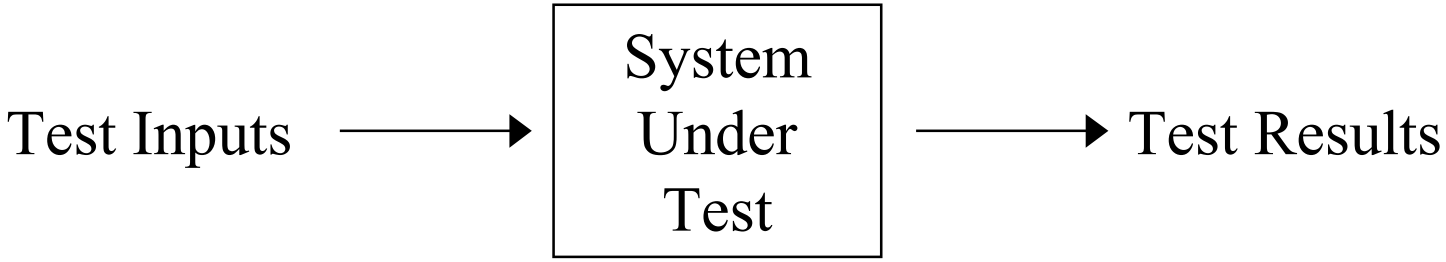
- What makes test cases simple or complex? *(A simple test manipulates one variable at a time.)*
- Advantages of simplicity?
- Advantages of complexity?

- 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

Simple Software Testing Model



Test Inputs

System Under Test

Test Results

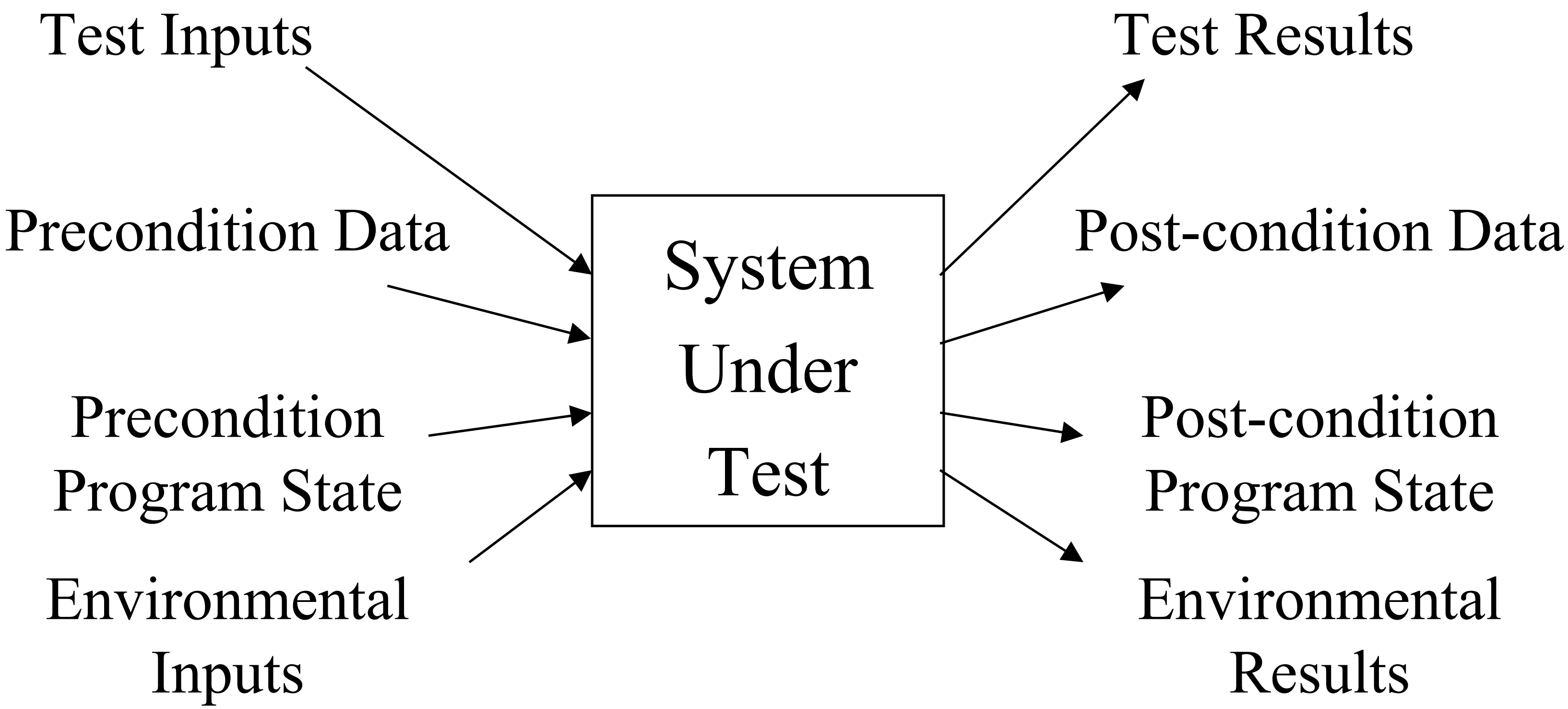
Implications of the Simple Model

- **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 Software Testing Model



Implications of the Expanded Model

We don’t control all inputs

We don’t verify everything

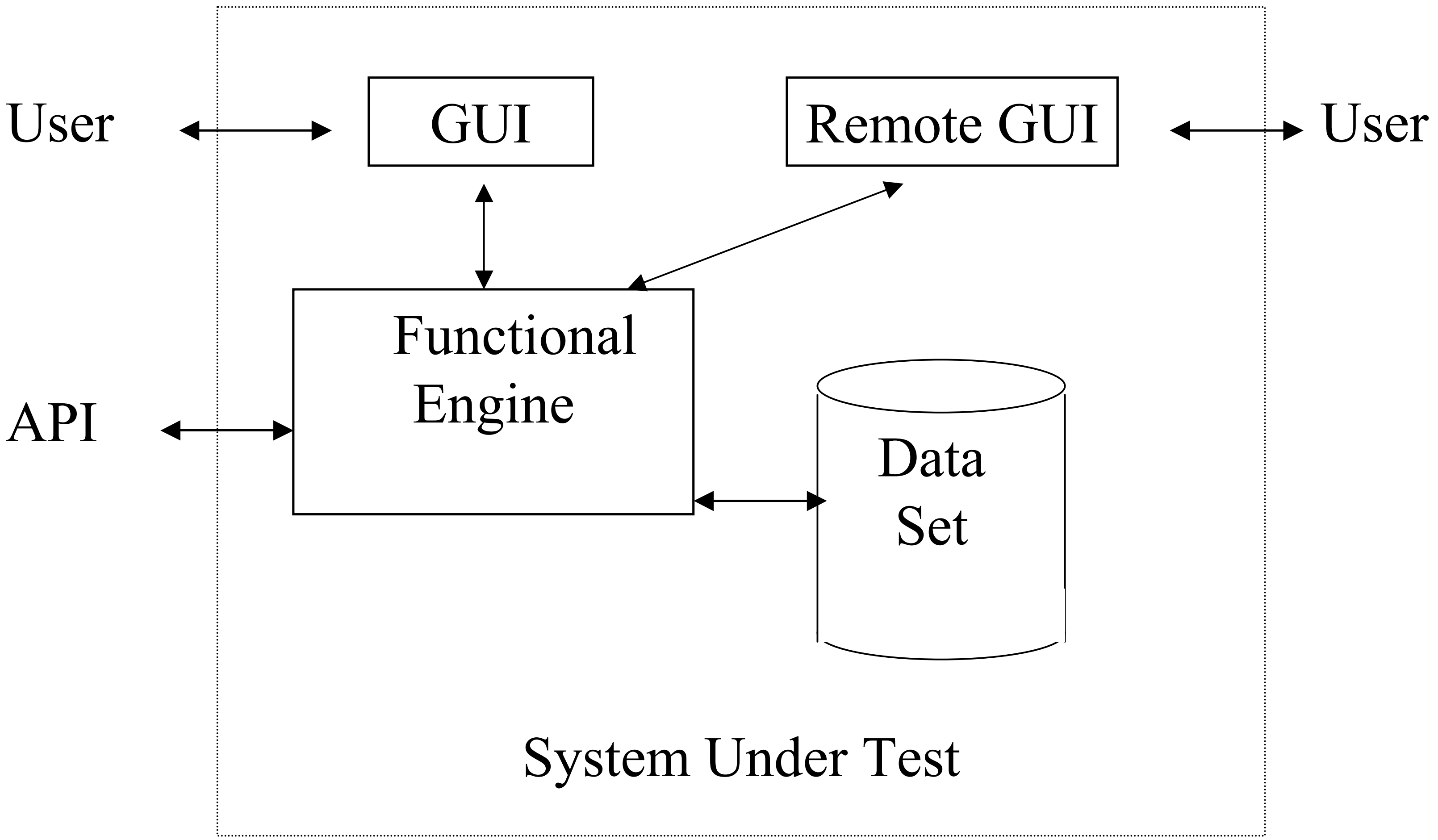
Multiple domains are involved

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



User

GUI

Remote GUI

User

API

Functional Engine

Data Set

System Under Test

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

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

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?

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

About Automated Regression Testing

A Non-Traditional View

General Regression Testing Strategy

Rerun a set of defined tests when anything changes

Usual purpose

- Bug regression (Show that a defect was not fixed)
- Old fix regression (Show that an old fix was broken)
- General functional regression (Show that a change caused some other working area to break)

Strengths

- Reassuring,
- Confidence building,
- Regulator-friendly

Blind spots

- Poor at finding new defects
- Anything not covered in the regression series may be broken
- Maintenance of regression tests 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

Demo Creation Process

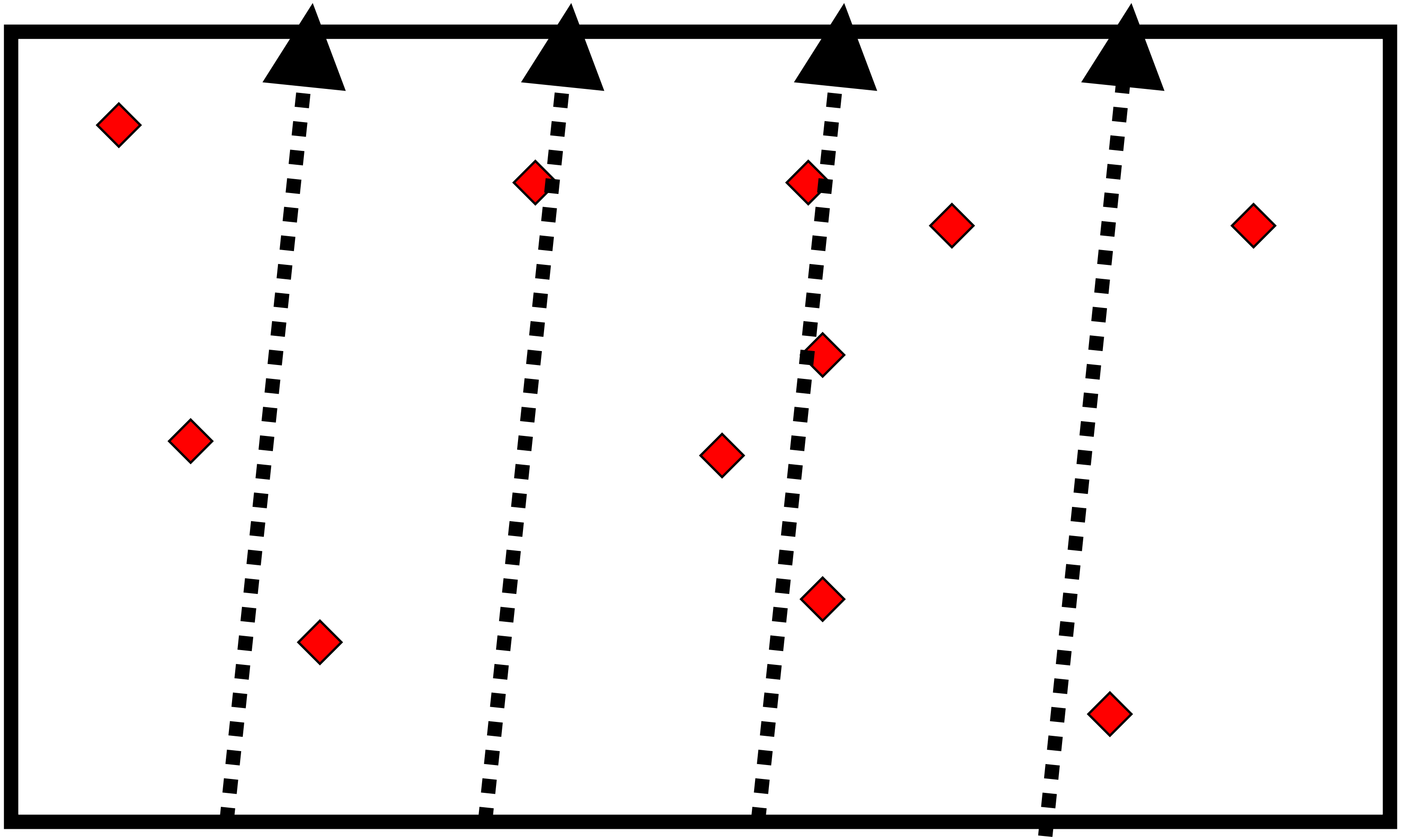
A demo creation technique for new products:

- conceive and create the demo exercise
- run it to see that it demonstrates the desired attributes
- if the program fails, report a defect
- in case of a defect, either wait for a fix or find a different way to do the exercise that doesn't fail
- remember the specific steps and values that work
- in future demos, do not deviate from the steps or try new values to minimize the chance of failure

This is the same as a regression test!

It's designed to MINIMIZE the chance of encountering a defect!

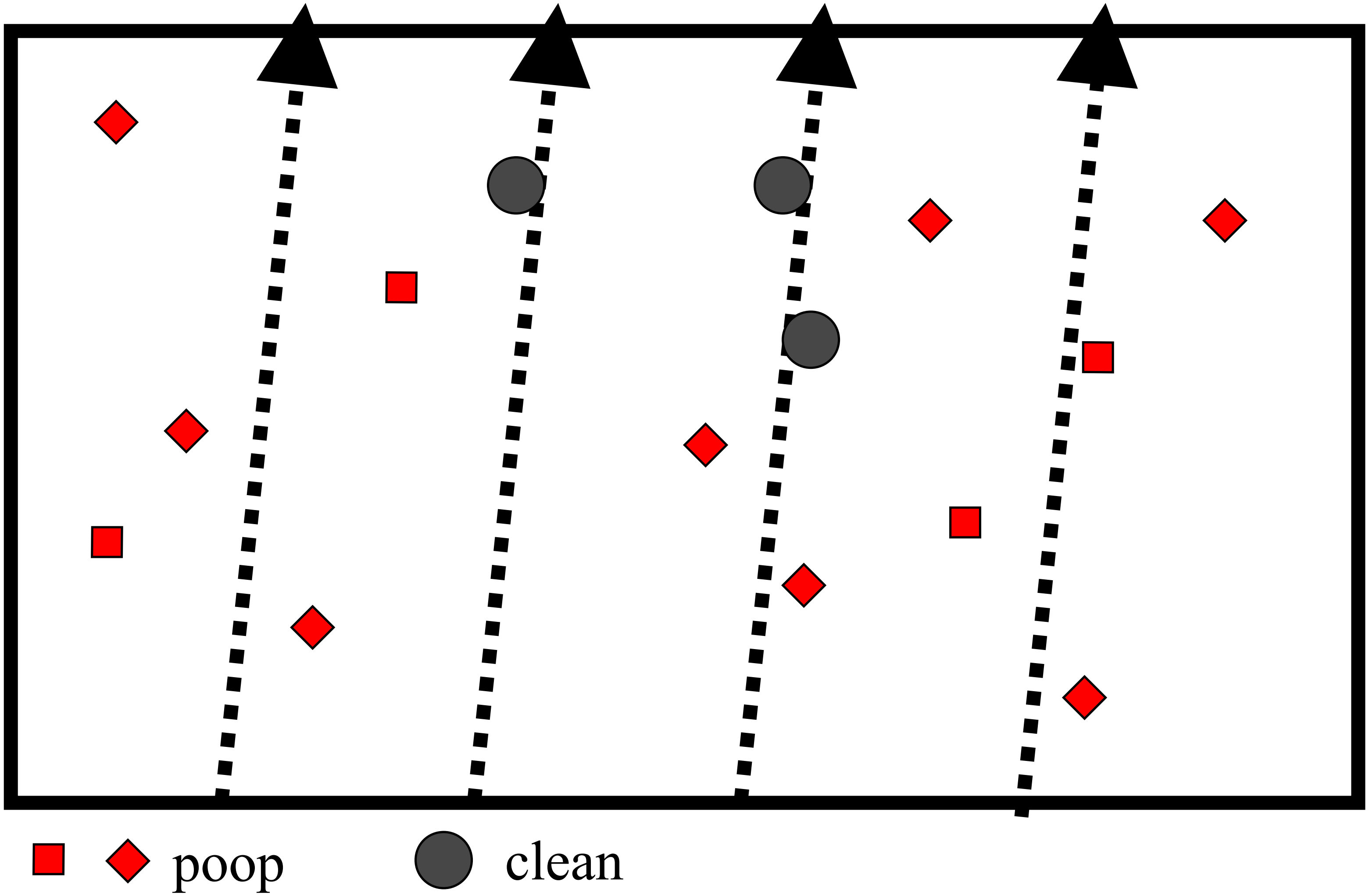
Testing Analogy: Clearing Poop



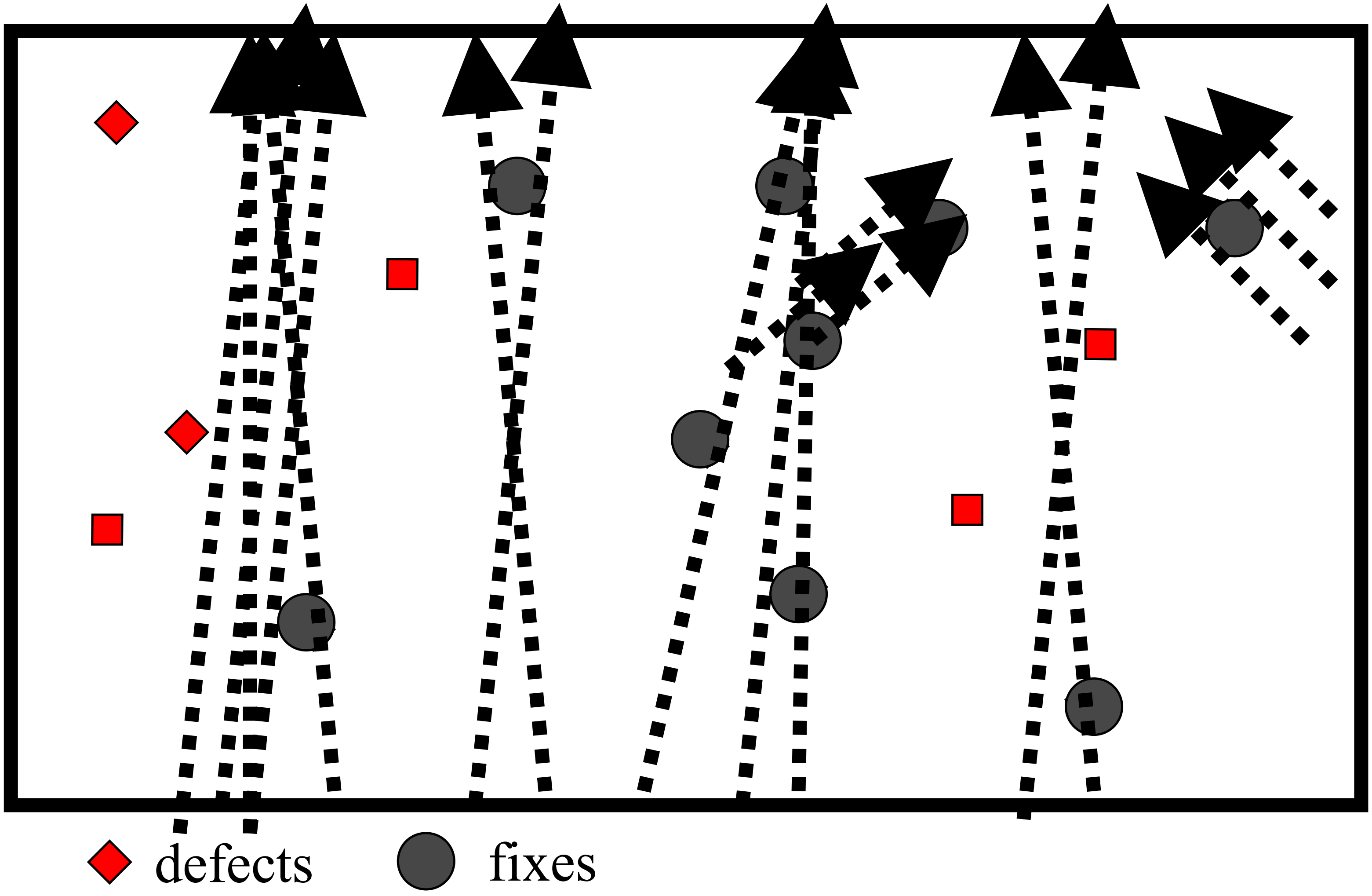
◆ Poop

Thanks to James Bach for the original slides and Eric Nellen for the dog poop analogy.

Following the Same Paths Won't Clear the Poop



Variable Tests are More Effective

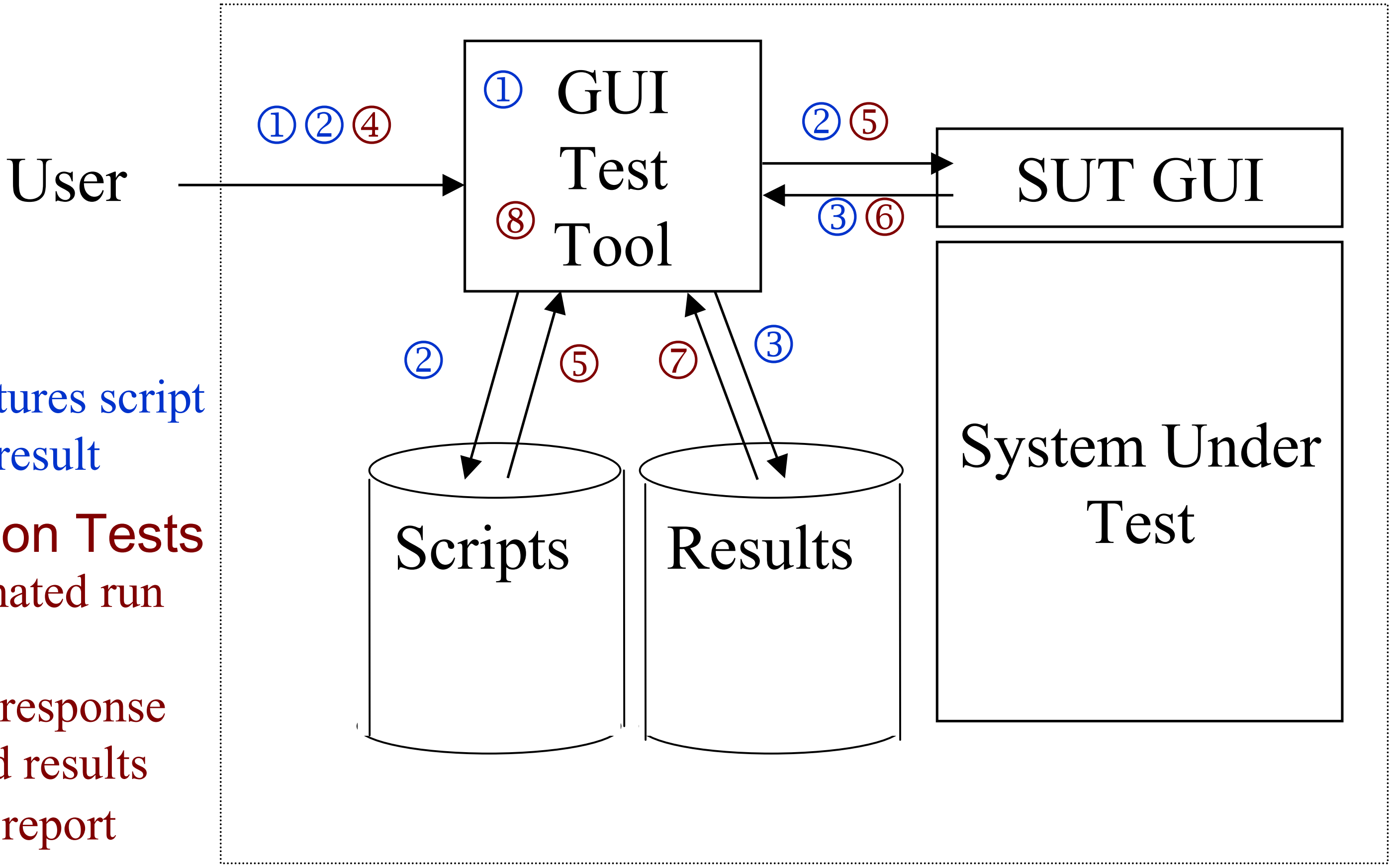


Why Are Regression Tests Weak?

- 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 <u>reduces</u> test variability
- Only verifies things programmed into the test

We *MINIMIZE* the *MINIMAL* chance of finding errors by automating regression testing.

A GUI Regression Automation Model



Setup

① Launch tool

② Test; tool captures script

③ Test; capture result

Run Regression Tests

④ Launch automated run

⑤ Play script

⑥ Capture SUT response

⑦ Read recorded results

⑧ Compare and report

How Automated is Regression Automation?

Analyze product	--	human
Design test	--	human
Run test 1st time	--	human
Evaluate results	--	human
Report 1st bug	--	human
Manage the code	--	human
Save for comparison	--	human
Document test	--	human
Re-run the test	--	MACHINE
Evaluate result	--	MACHINE
(But, a human is needed if there's any mismatch)		
Maintain the test	--	human

We may get the machine to do a lot of our work,

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)

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.)

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.

More about Regression Testing later

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

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*

Notes

Software Test Automation:

Foundational Concepts

Test Oracles I

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 usually use the term.)***
- ***Reference and Evaluation Function:* You ask it whether the program passed the test. *(Some oracles work this way)***

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.) *(These often lead to more powerful automation.)***

Reference Functions: Some Typical Examples

Spreadsheet Version N and Version N-1

- Single function comparisons
- Combination testing
- Potential for problems with revised functions

Database management operations

- Same database, comparable functions across DBMs or query languages

Bitmap comparisons with output files

- The problem of random variation

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. $\sin^2(x) + \cos^2(x) = 1$)

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

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

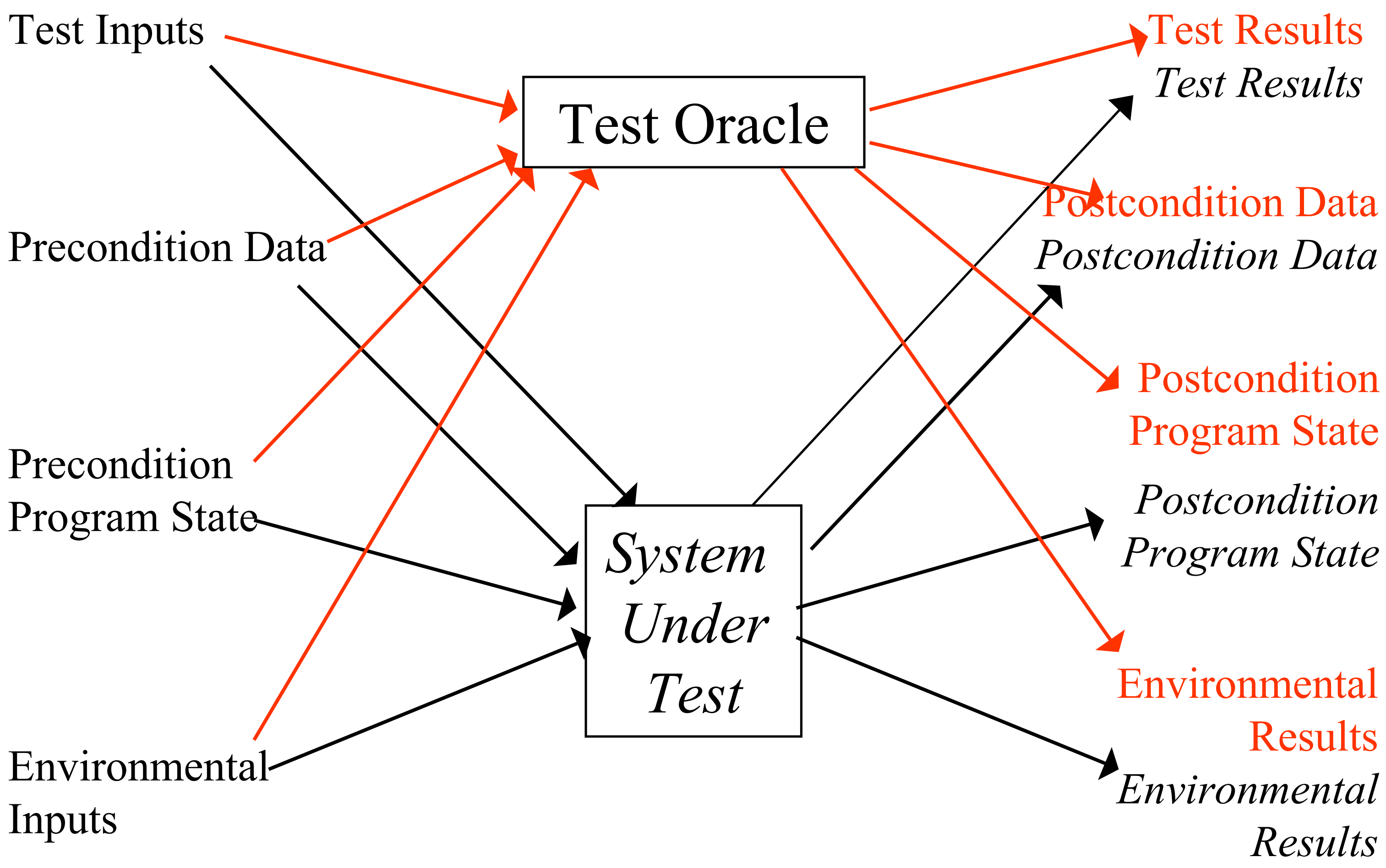
Check (apparently) insufficiently complete attributes

- ZIP Code entries are 5 or 9 digits

Check probabilistic attributes

- X is usually greater than Y

A "Complete" Oracle



Test Result Possibilities

Situation Test Results	No Error	Error
As Expected	Correct	Silent Miss
Unexpected	False Alarm	Caught

True Oracle Example

Simulator

Separate Implementation

Situation / Test Results	No Error	Error
As Expected	**Correct**	**Silent Miss**
Unexpected	**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	Silent Miss
Unexpected	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	Silent Miss
Unexpected	False Alarm	Caught

Incomplete Oracle Example 3

Age Checking

Situation Test Results	No Error	Error
As Expected	Correct	Silent Miss
Unexpected	False Alarm	Caught

Notes

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

Oracle Completeness

Input Coverage

Result Coverage

Function Coverage

Sufficiency

Types of errors possible

SUT environments

There 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 long 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 comparators
- Execution cost
- Cost of comparisons
- Additional analysis of errors
- Cost of misses
- Cost of false alarms

Notes

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

- **Document expectations within tests**
- **Put checks into the tests**
- **Check at the earliest point possible**
- **Gather information as soon as a deviation is detected**
  - Results
  - Other domains
- **Check as many things 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

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 Approaches

Architectures

- Equivalence Testing
- Scripted Testing
- Framework Based
- API Based
- Load/Stress/Performance Testing
- Data Driven Testing
- Stochastic or Random Testing
- Model Based

Automation Architecture Frameworks

Equivalence

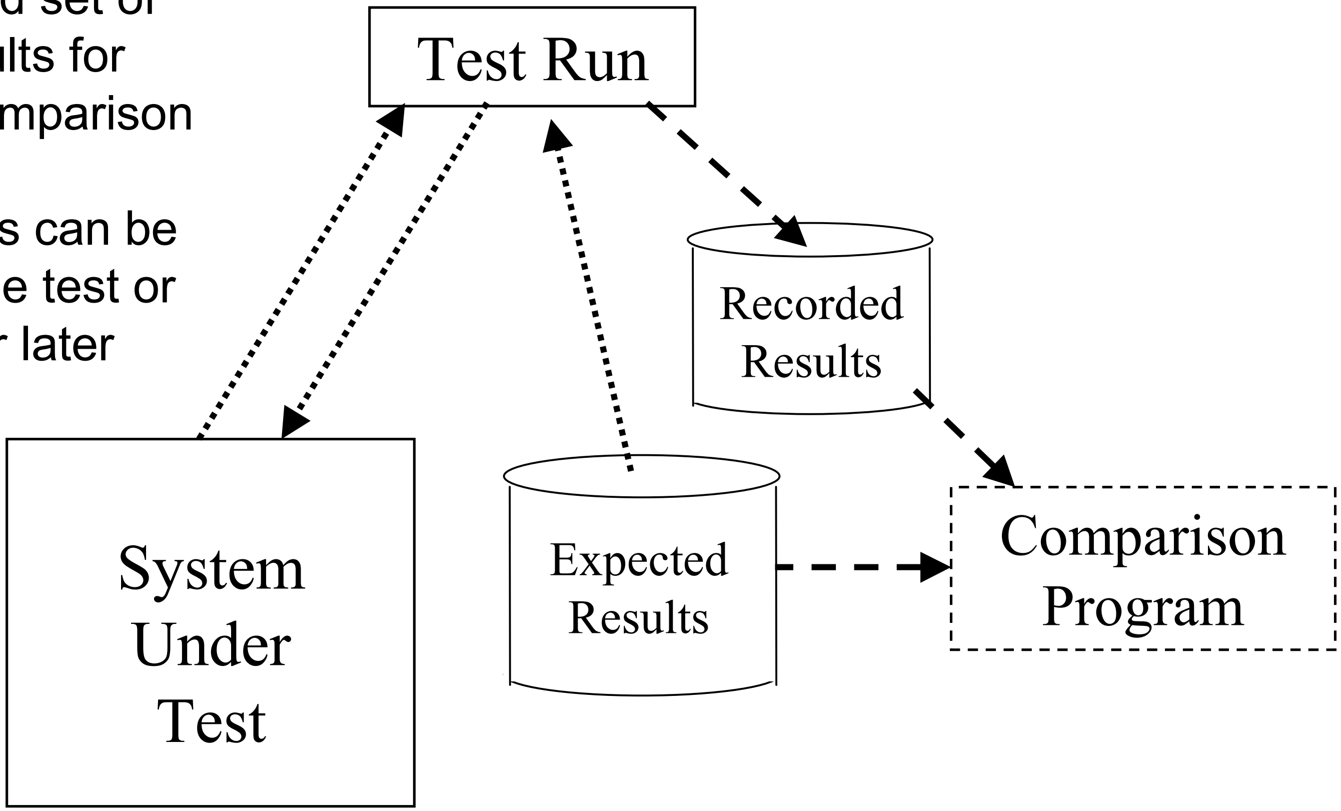
Equivalence Testing

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

A/B Comparisons

An A/B comparison has a recorded set of expected results for automated comparison

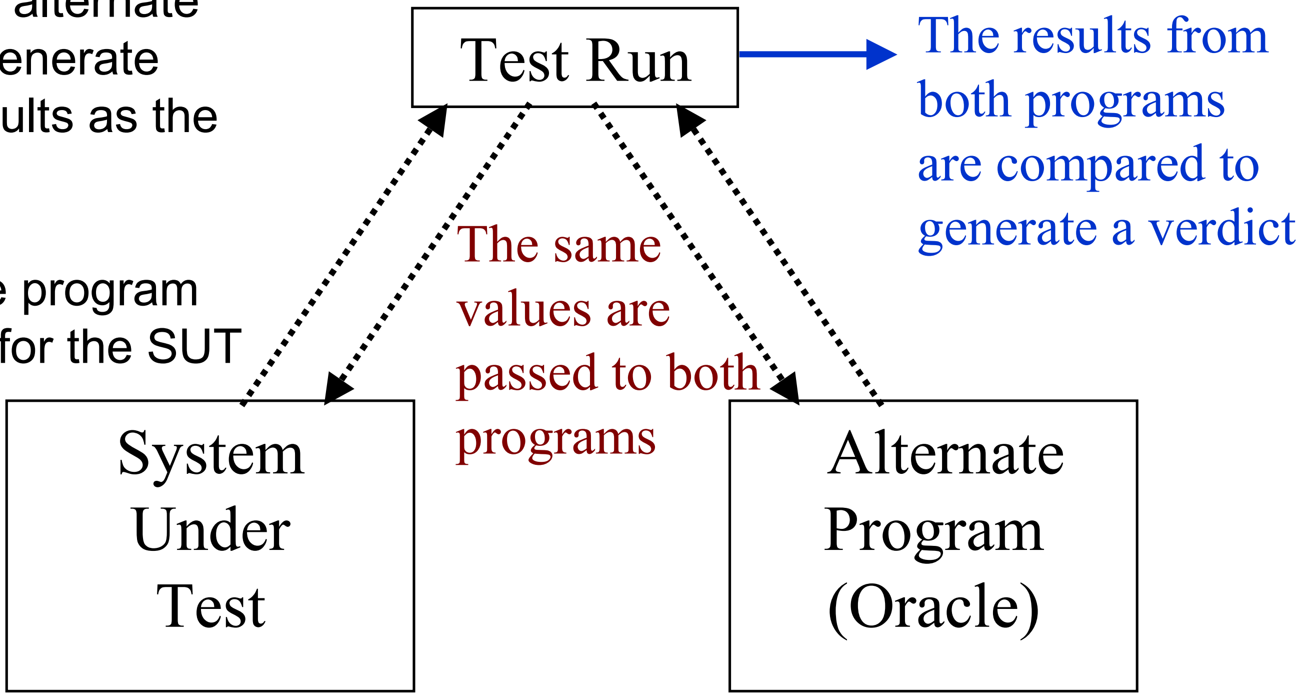
Current results can be checked by the test or put in a file for later comparison



Function Equivalence Testing

A Function Equivalence Test uses an alternate program to generate expected results as the test runs

The alternate program is the oracle for the SUT


Test Run
System Under Test
Alternate Program (Oracle)
The same values are passed to both programs
The results from both programs are compared to generate a verdict

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

Automation Architecture Frameworks

Scripted

Scripted Testing
A Potential for Problems

Methodological

- Fragile tests
- What is “close enough”
- Must prepare for product and user interface changes
- Support for different configurations and environments
- Tracking the state of software under test
- Hard-coded input data limits test reusability

Technical

- Playing catch up with new technologies (e.g., GUI, SOAP)
- Instrumentation is invasive
- Tools can be seriously confused and defective
- Tools require customization and tuning
- Custom controls issues

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 consistency oracle based

A Special Case: MASPAR Example

MASPAR functions: square root tests

- 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?

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

Classic Automated Regression Testing

- Automate existing manual tests
- Add to regression tests over time
- Results verification = file compares
- Automate all tests
- One technique for all tests

Automation Architecture Frameworks

Framework Based

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."

Automation Architecture Frameworks

API Based

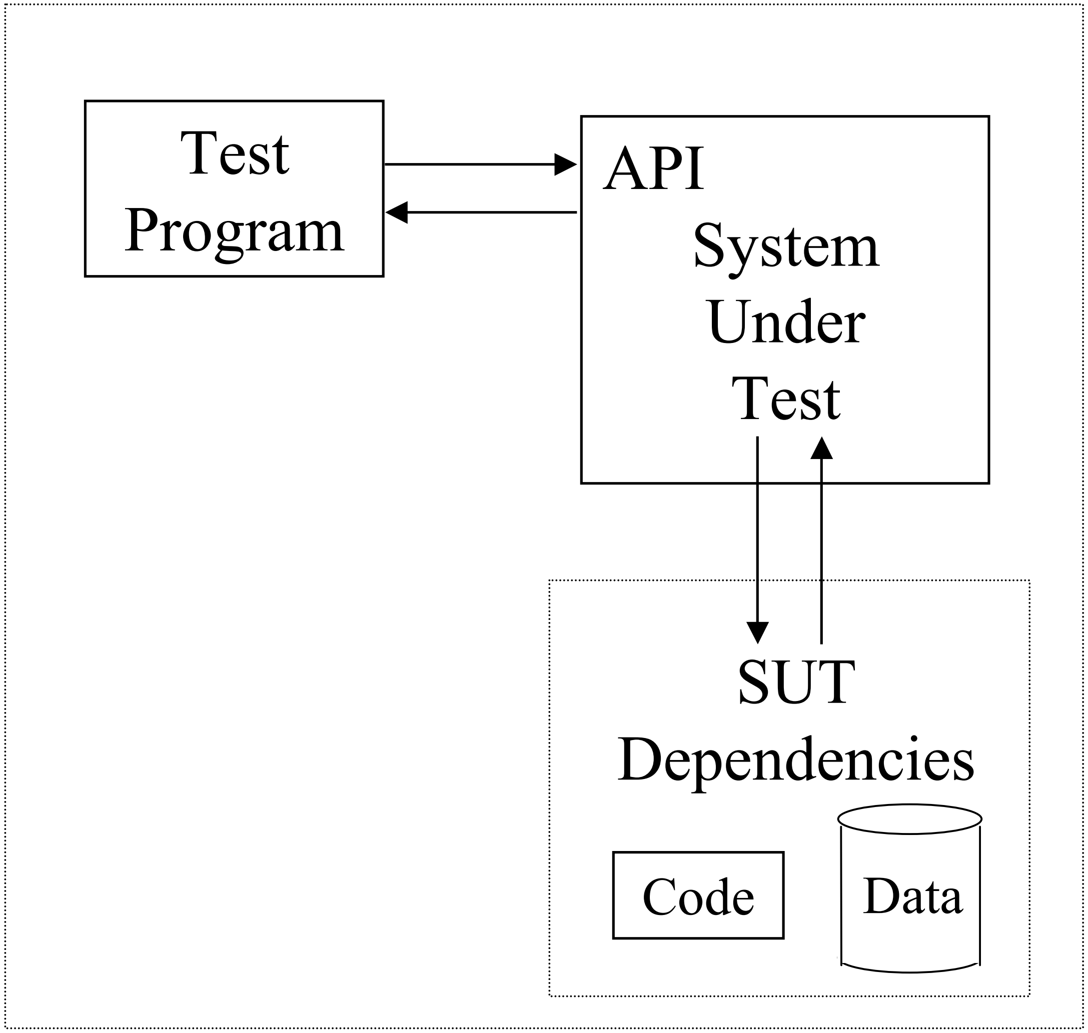
Application Programming Interface (API) Testing

- **Public and Private APIs**
- **Unit, integration, and system level tests**
- **General or specific purpose tests**
  - Hard-coded
  - Data driven
  - Script based
  - Random

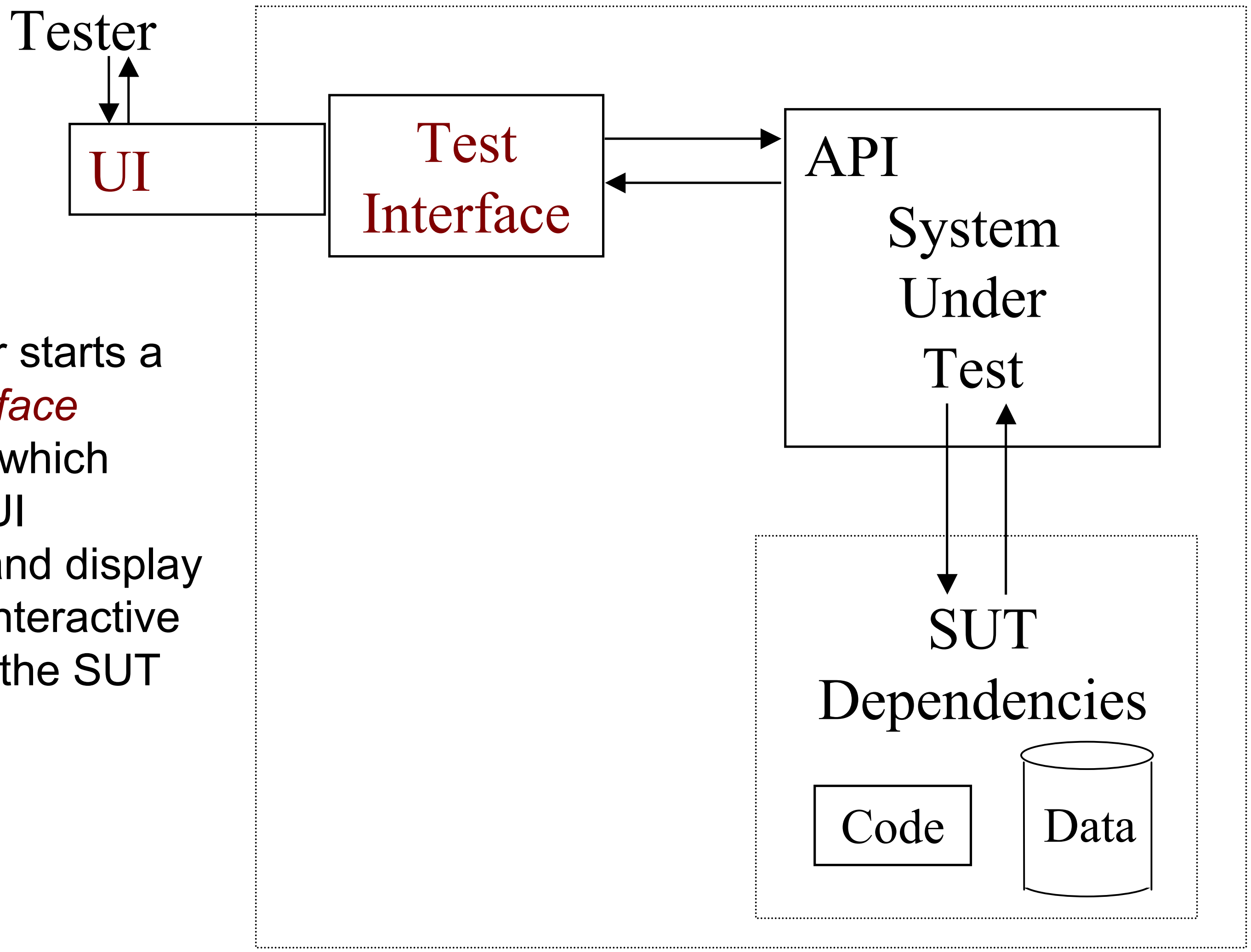
Code Based API Testing

For code based API testing, each test case is a program

The program provides the input and analyzes the result



UI Based API Testing

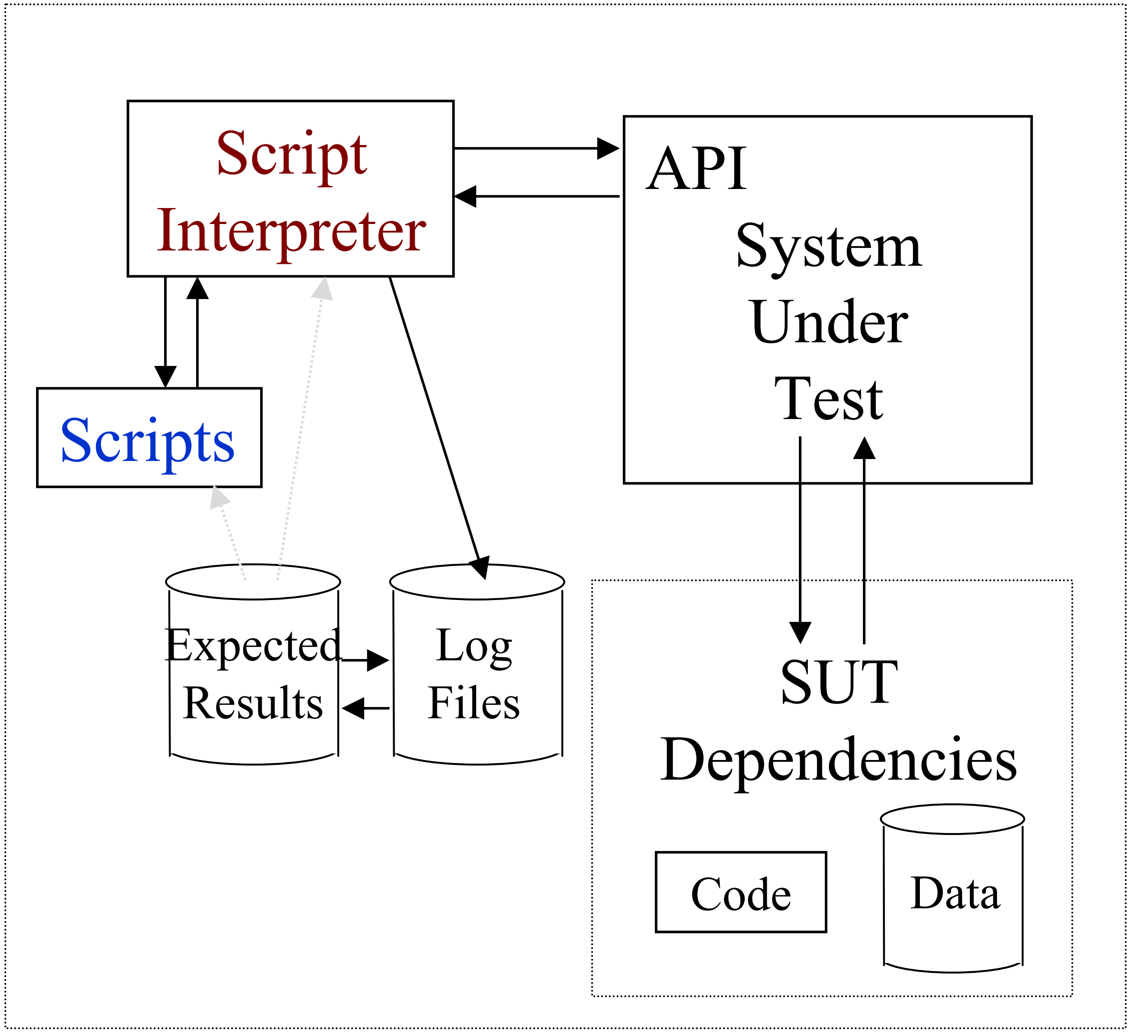


The tester starts a *Test Interface* program, which includes UI for input and display allowing interactive testing of the SUT

Script Based API Testing $_1$

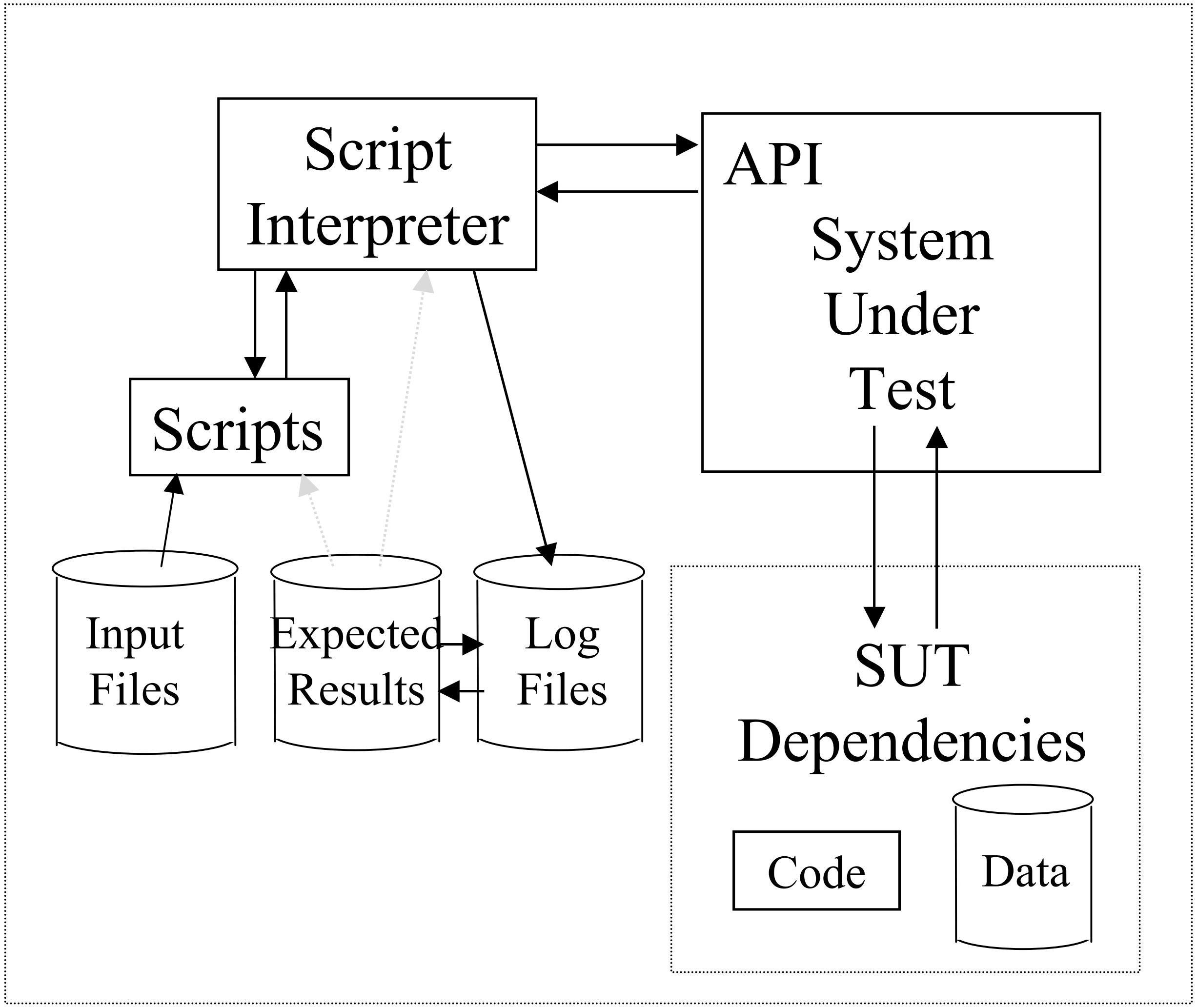
Tester starts a *Script Interpreter* program, which uses *Scripts* (programs) to exercise the SUT

Results are saved in *Log Files* for comparison with *Expected Results* (or compared as the results are generated).



Script Based API Testing$_2$

The *Scripts* read data from *Input Files* to extend their usefulness.



Automation Architecture Frameworks

Load and Stress

Load and Stress Testing

- **Checking for reasonable handling of unreasonable requests**
- **Load: background for other tests**
- **Stress: push it past the limits**
  - Classically: up to and just past the limits
  - More recently: overwhelm the product

Load and Stress Testing

- **Methods for load and stress; increase:**
  - Frequency (how often)
  - Intensity (how much each time)
  - Duration (how long)
- **Reuse automated regression tests**
- **Create load or stress tests**

Notes

Automation Architecture Frameworks

Data Driven

Data-Driven Architecture 1/2

Separate the elements involved in automated Testing

- Test exercises that define the individual tests
- The automation tool that executes the automated test code
- The language and command set for describing the test steps
- The test data (input and results)
- SUT characteristics
  - » The SUT's variables
  - » The SUT's commands
  - » The SUT's UI
  - » The SUT's state machine
- The test environment and system configuration

Store, display, and manage the different data in whatever way is most convenient

These are just implementation details of the test environment and SUT

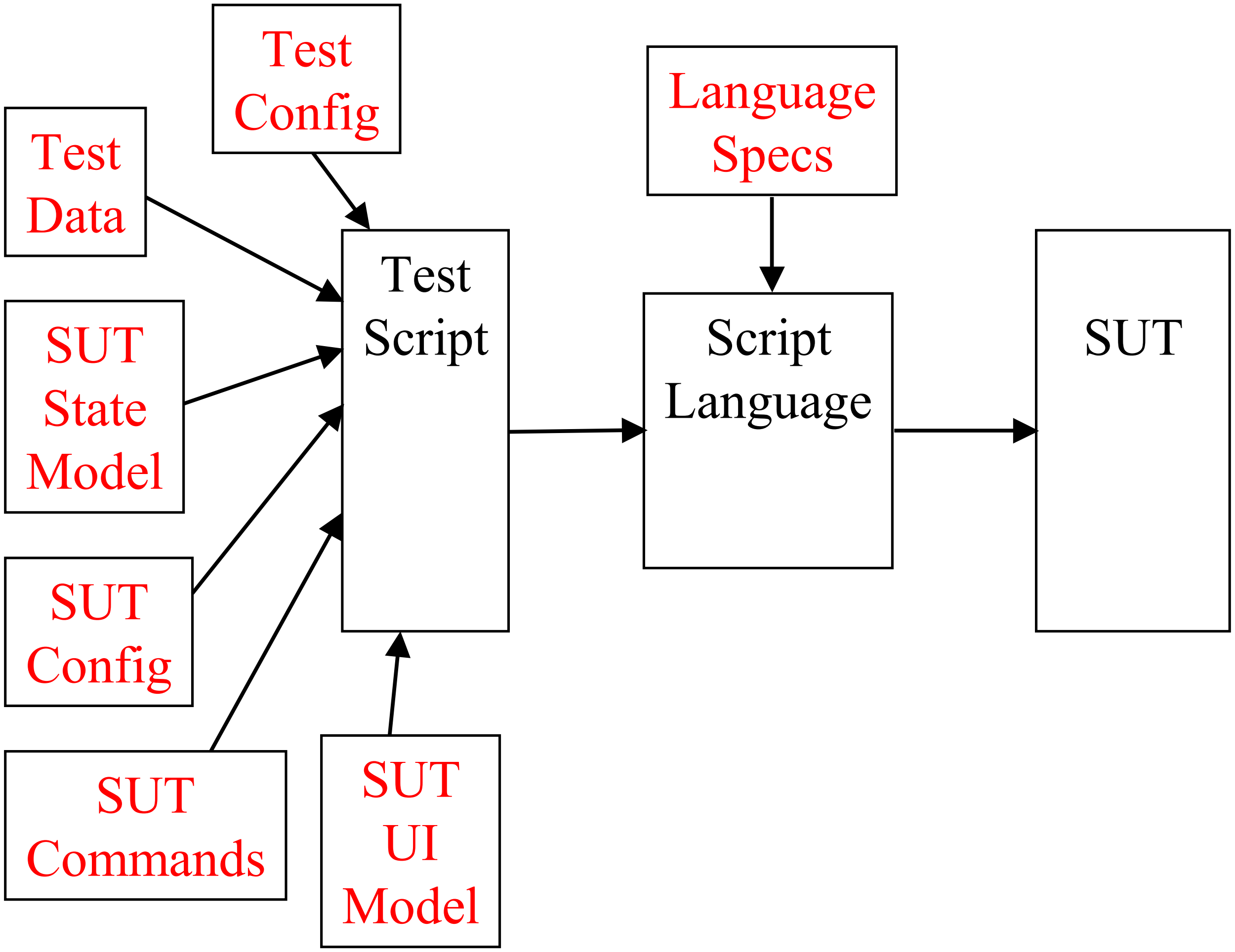
Treat them as values assigned to variables of the automation software

Data-Driven Architecture 2/2

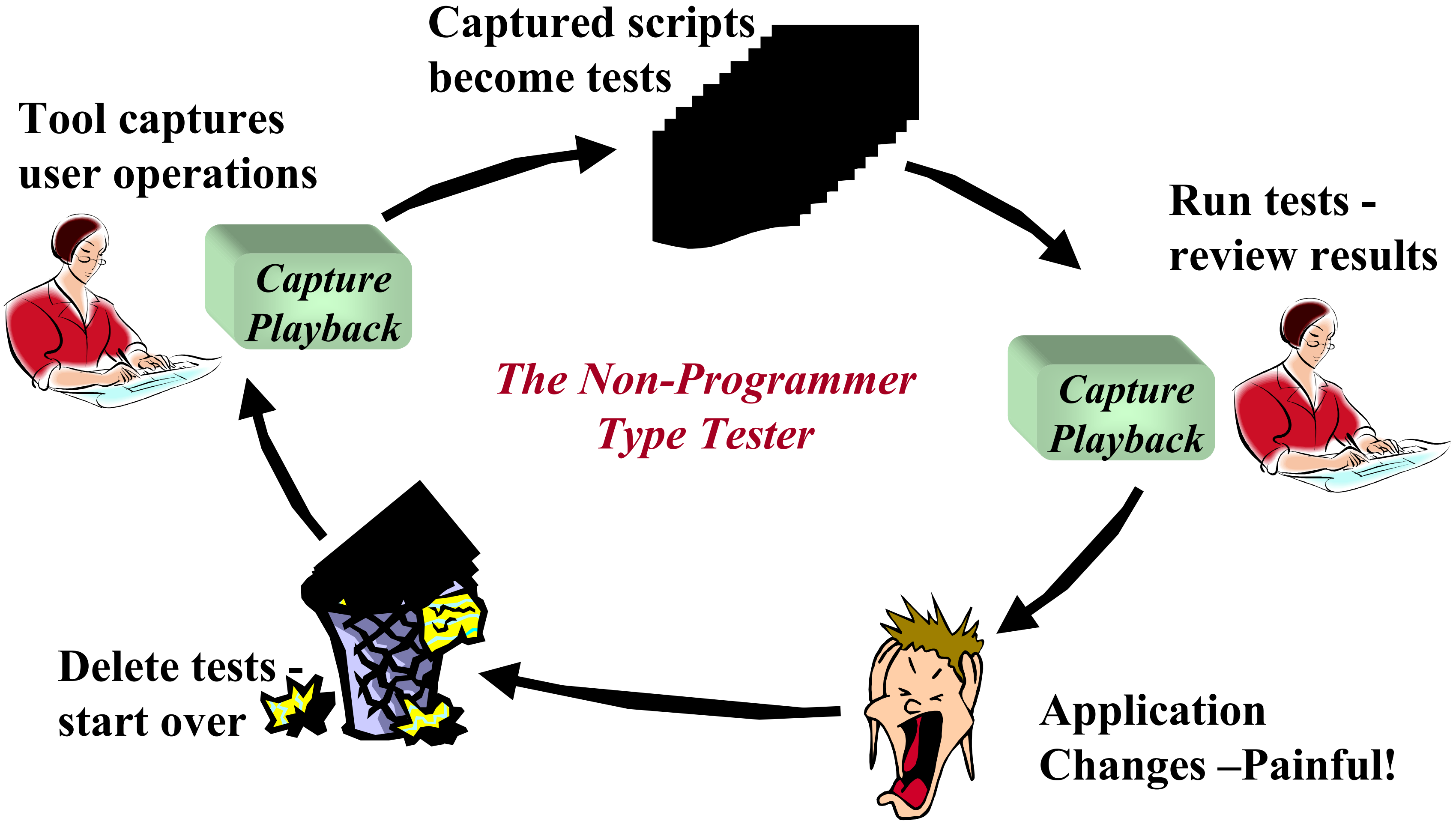
In general, we can benefit from separating the treatment of one type of data from another with an eye to:

- 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
- making the test scripts flexible, broadly applicable, and reusable

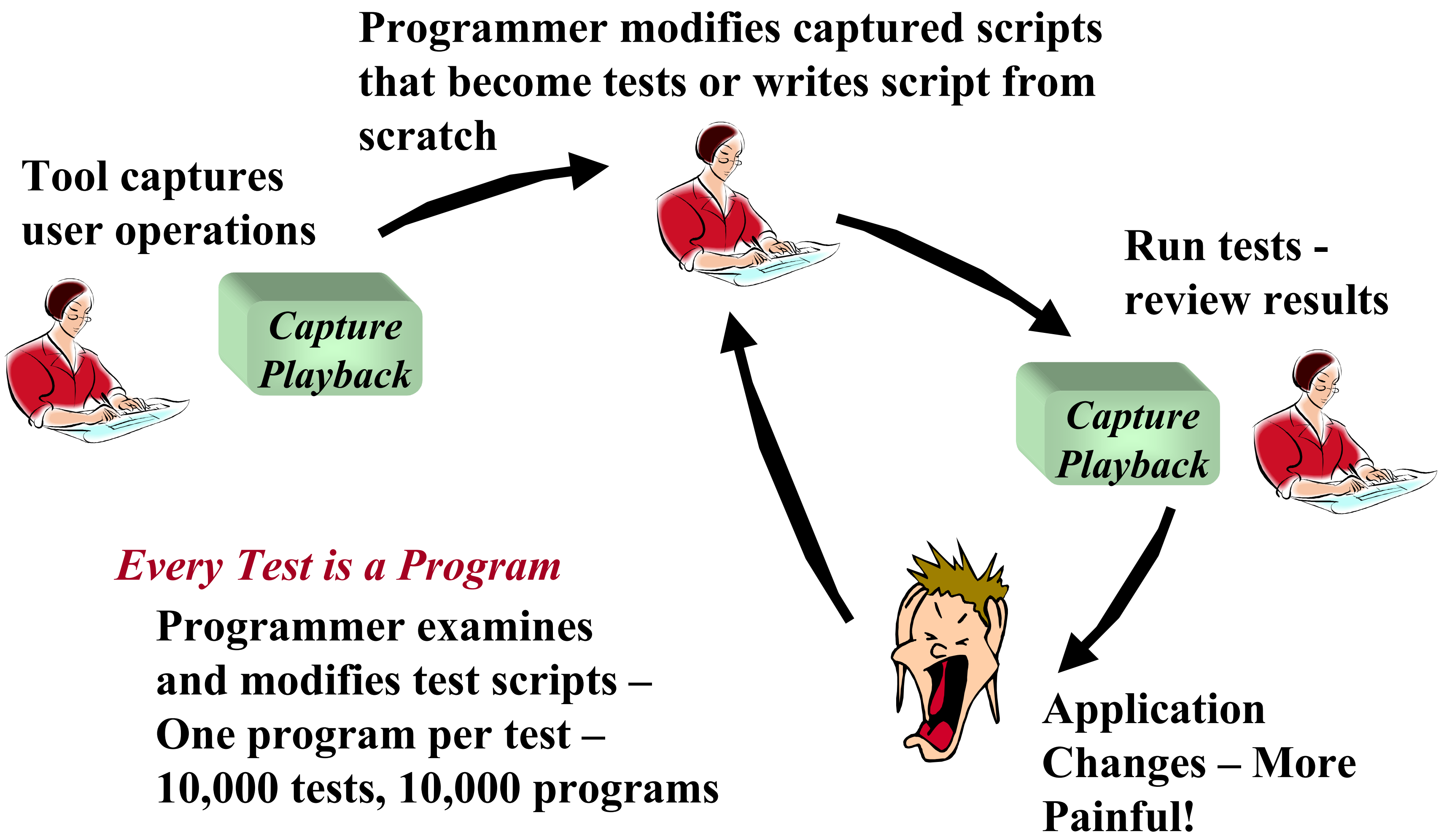
Data Driven Architectures



First Generation Testing Tools



Second Generation Testing Tools



Tool captures user operations

Capture Playback

Programmer modifies captured scripts that become tests or writes script from scratch

Run tests - review results

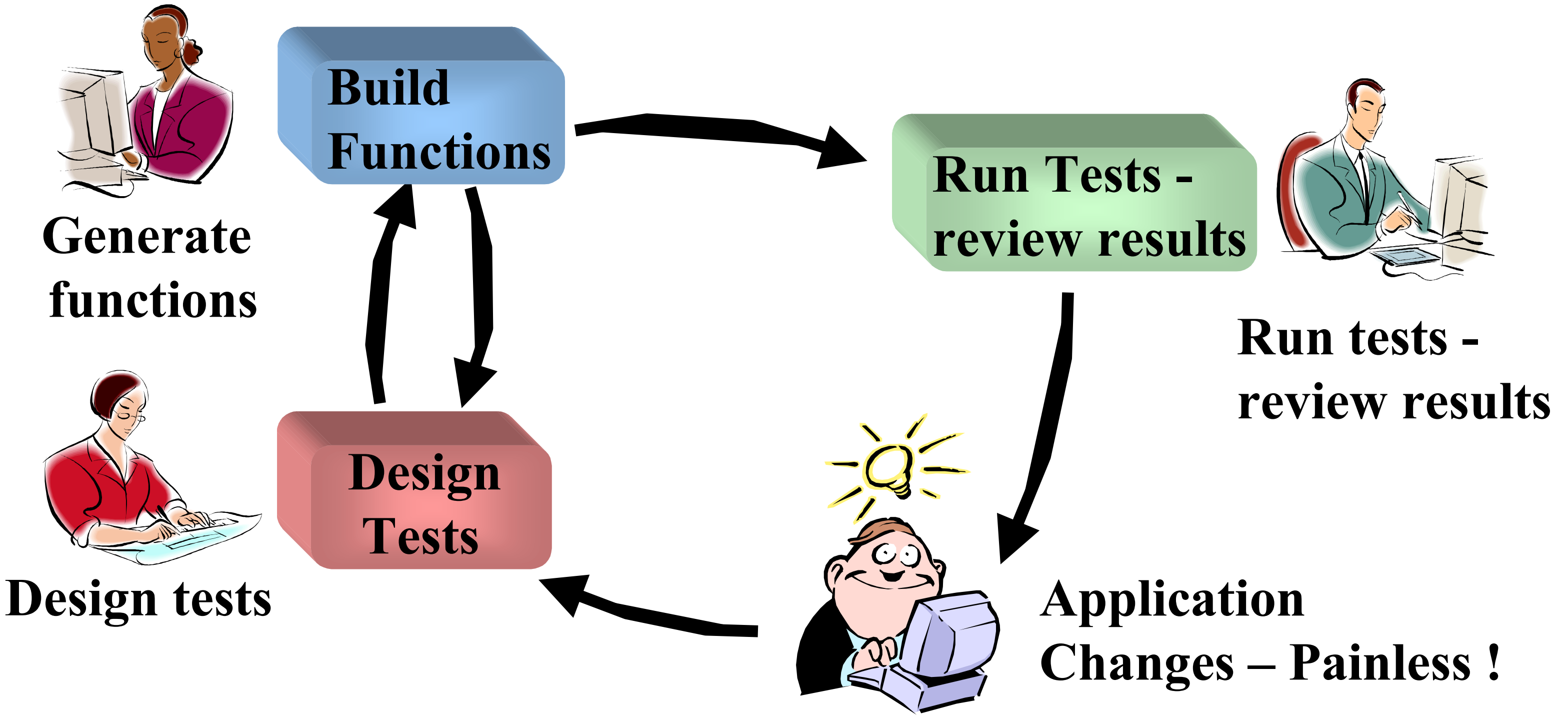
Capture Playback

Application Changes – More Painful!

Every Test is a Program

Programmer examines and modifies test scripts – One program per test – 10,000 tests, 10,000 programs

Role-Based Third Generation TAS



Build Functions

Run Tests - review results

Design Tests

Generate functions

Design tests

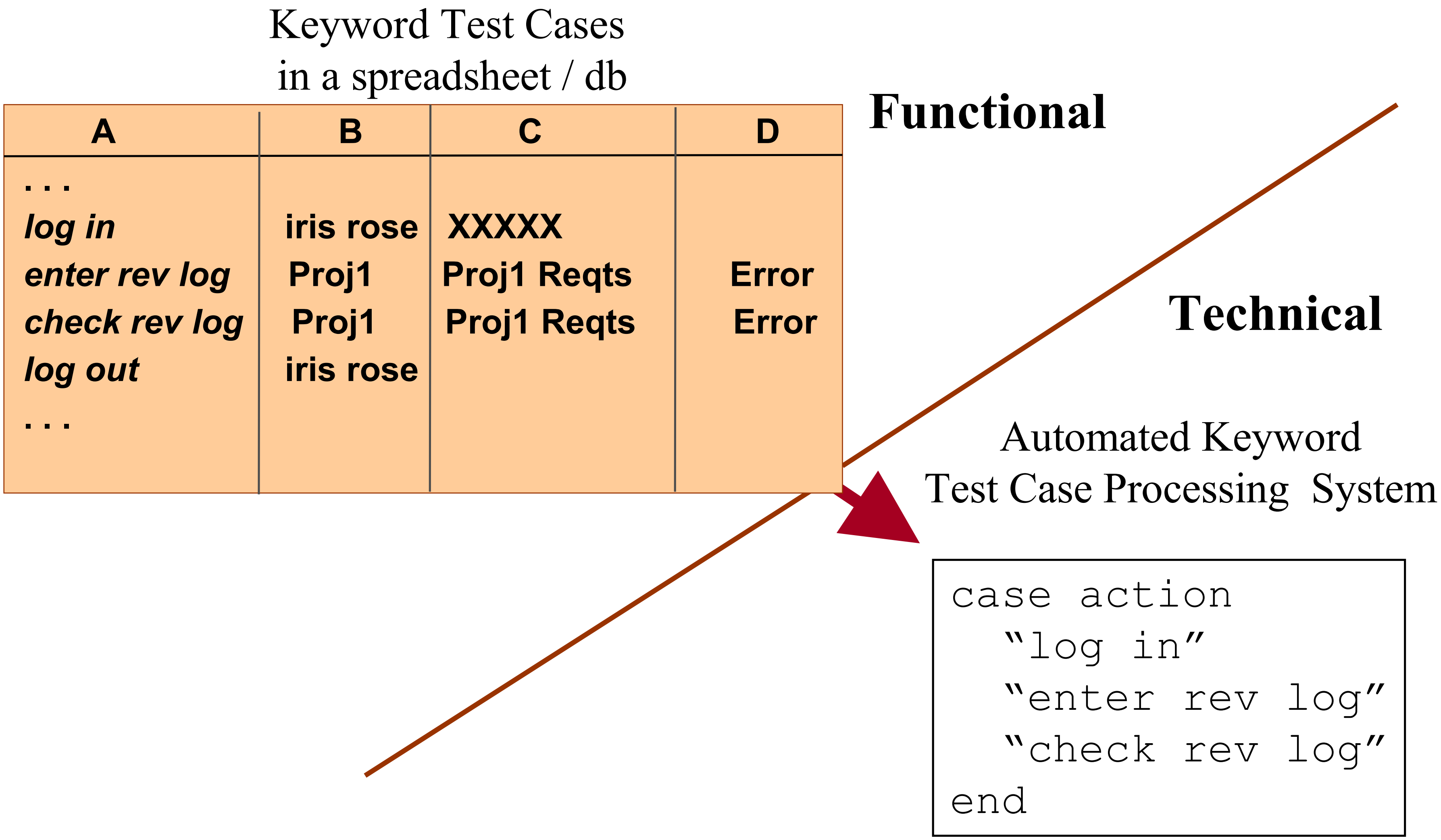
Run tests - review results

Application Changes – Painless !

With effective 3rd Generation Automation Tools

No or very minor modification – Automatically regenerate functions
One program processes all tests – 10,000 tests, 1 program!

Recommendation:
Separate Functional and Technical Tasks



Keyword Test Cases
in a spreadsheet / db

A	B	C	D
. . .			
log in	**iris rose**	**XXXXX**	
enter rev log	**Proj1**	**Proj1 Reqts**	**Error**
check rev log	**Proj1**	**Proj1 Reqts**	**Error**
log out	**iris rose**		
. . .			

Functional

Technical

Automated Keyword
Test Case Processing System

```
case action
  “log in”
  “enter rev log”
  “check rev log”
end
```

Calendar Test Data Example

The test designer defines a test case by providing values for input data in a row

A test driver reads each row and inputs the values to the SUT as the test exercise

		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
Test set 1															
Test set 2															
Test set 3															
Test set 4															
Test set 5															
Test set 6															

Printer Configuration Example

	Paper Sizes	Print Languages	Number of Trays	Initialization Time	Page Orientation	Standard Fonts	Default Font Name	Default Font Size	Optional Fonts	Font Memory	Interface Type	Job Queueing	B&W / Color	Paper Types
Printer 1														
Printer 2														
Printer 3														
Printer 4														
Printer 5														
Printer 6														

Test accepts input of Printer Type and reads in the row of the table that describes the Printer

Test case uses the values from the table as variables to define limits and expected results

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.**

Data-Driven Architecture: Calendar Example

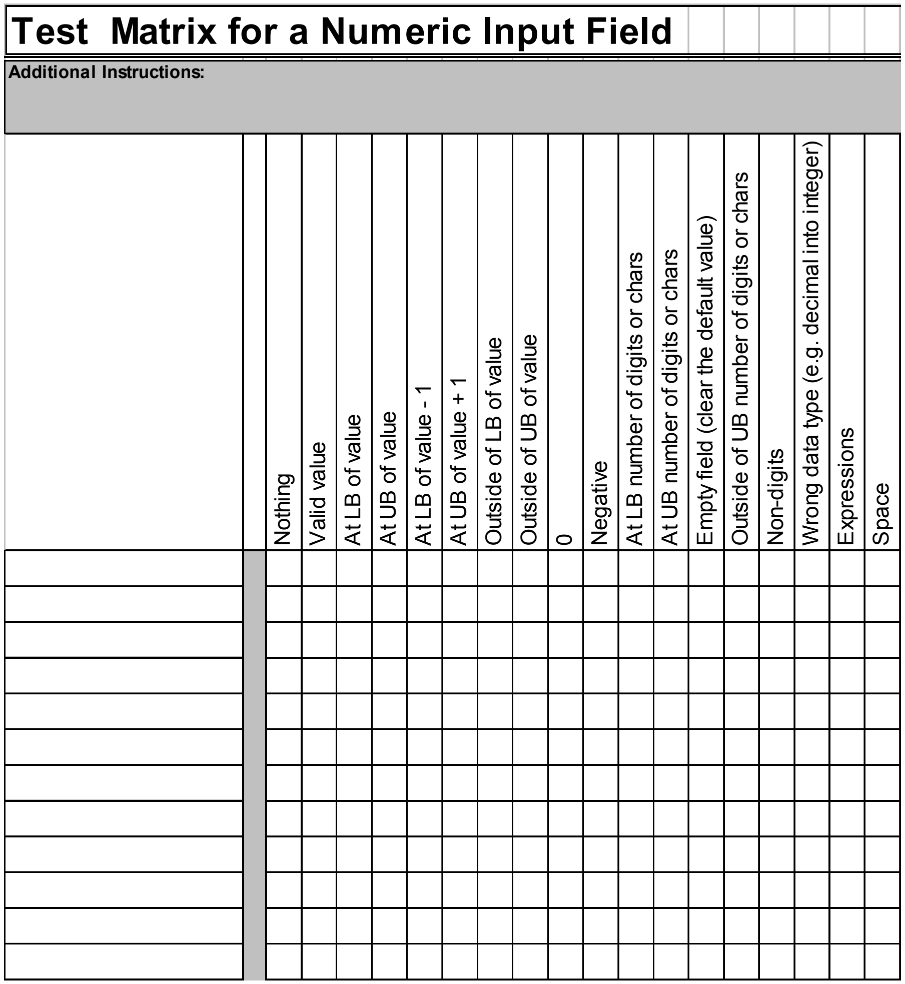
This is a good design from the point of view of optimizing for maintainability because it separates out four types of things that can vary independently:

- *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 can be written as short, separate functions for easy maintenance*
- *The short scripts that map calendar elements to the program functions can call sub-scripts that wrap common program functions*
- *If the tool syntax changes, maintenance involves changing the wrappers' definitions or the sub-scripts rather than the tests*

Application-Independent Data-Driven

- Generic tables of repetitive types
- Rows for instances
- Automation of exercises can be based on selection of which columns to use

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

Automation Architecture Frameworks

Stochastic or Random

Random, Independent, and Stochastic Approaches

Random Testing

- Random (or statistical or stochastic) testing involves generating test cases using a random number generator
- 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 subsequent tests don't matter

Stochastic Testing

- Stochastic process involves a series of "random" events over time
  - » Prices in the 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 [random] tests
- Non-Stochastic tests using self-validating data
- Stochastic tests without a model
- Stochastic tests using a model
- Stochastic tests using other attributes of software under test

Non-Stochastic Random Testing

- The computer runs a large set of essentially independent tests
  - The focus is on the results of each test
  - Tests are designed to minimize sequential interaction among them
- 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
  - Other tests using heuristic oracles

Random Testing With No Model: "Dumb Monkeys"

Dumb Monkey

- Random sequence of events (e.g., key clicks)
- Continue until crash or a diagnostic event occurs
- Continue through crash (Executive Monkey)

The diagnostic pass/fail analysis is based on knowledge of the system, not the internals of the code

Random Testing: “Dumb Monkeys”

Fundamental question or goal

- High volume testing, involving a long sequence of tests
- A typical objective is to evaluate program operation over time
- The distinguishing characteristic of this approach is that the testing software does not use a 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;
- The test 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)

- Clever monkeys
  - » Some knowledge about the system or the environment
  - » Some careful rules of conduct
  - » See Noel Nyman’s Freddy
- Executive monkeys:
  - » Know nothing about the system
  - » Push buttons randomly until the system crashes
- O/S compatibility testing
  - » No model of the software under test
  - » Diagnostics might be available based on the environment
- Early qualification testing
- Life testing
- Load testing

This can be done via an API or command line just as well as using a UI

Random / Statistical Testing: Stochastic, Assert or Diagnostics Based

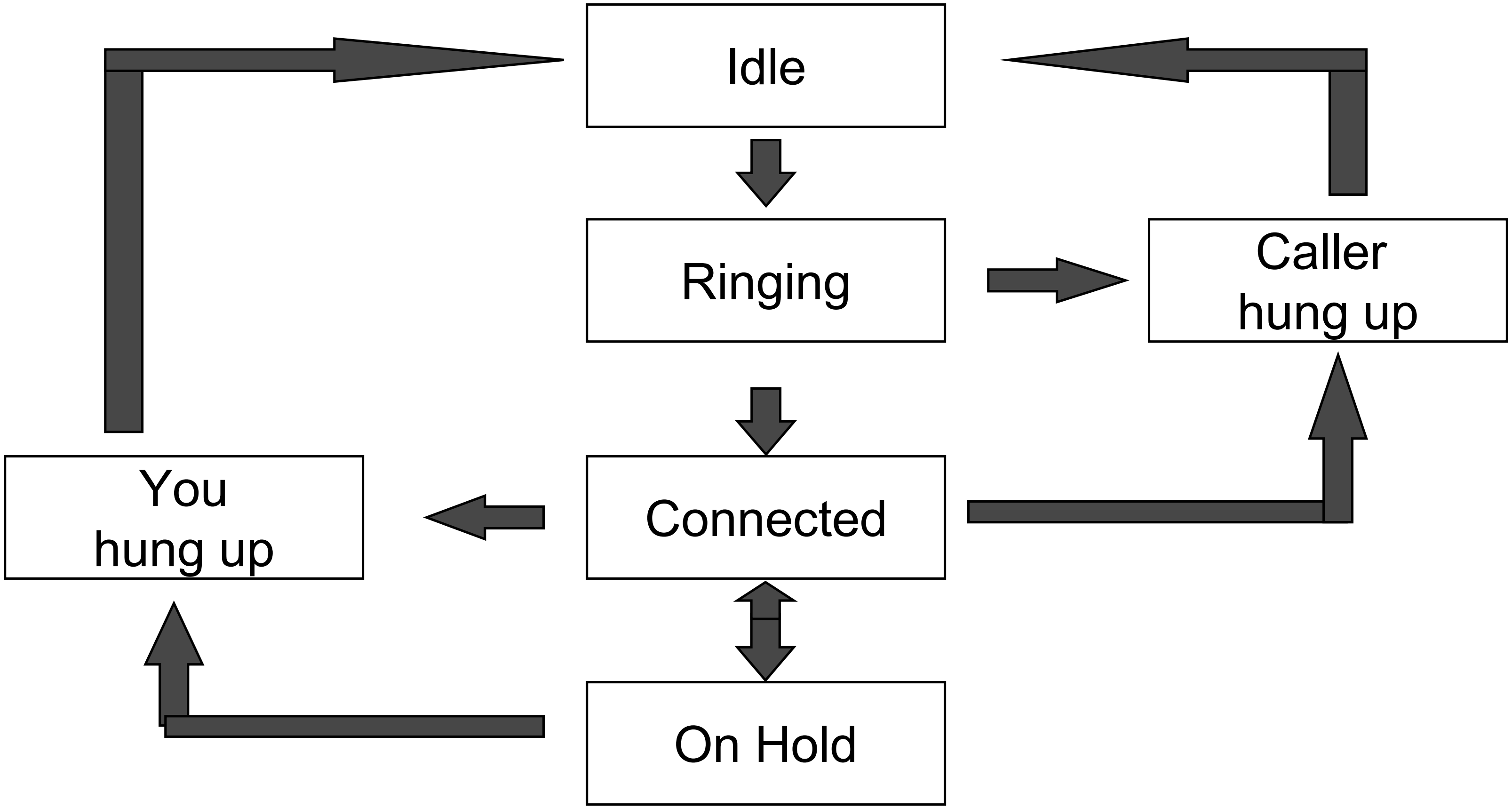
Fundamental question or goal

- 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 seemingly random, irreproducible crashes at a beta site
- All of the individual functions worked
- We had tested all lines and branches
- We had inspected all the code
- The stochastic testing was done using an event 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
- Load testing
- Search for specific types of long-sequence defects

Random Testing:
Stochastic, Regression-Based

- **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
  - Design the tests so that the automation can identify failures
- **Run the tests in random order over a long sequence**
  - You get pass/fail info for every test, but without having to achieve the same depth of understanding of the software
  - You probably have worse coverage, less awareness of your actual coverage, and less opportunity to stumble over bugs
- **This is a low-mental-overhead alternative to model-based testing**
- **Unless this is very carefully managed, there is a serious risk of non-reproducibility of failures**

Random Testing: Sandboxing of 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
- For each test
  - run it on its own many times in one long series
  - randomly switch as many other environmental or systematic variables during this random sequence as our tools allow
- This is the “sandbox” series; each test 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
  - to avoid creating a big random test series that crashes only because one test doesn’t like being run again
  - to weed out these simple causes of failure.
  - to avoid spending a fortune trying to control this risk

Stochastic Testing: Regression Based

Testing with Sequence of Passed Tests

- Collect a large set of regression tests, edit them so that they don't reset system state
- Sandbox qualify each test
- Randomly run the tests in a long series including checking expected against actual results
- Watch for failures even though each of the tests pass individually

Random / Statistical Testing: A Regression Test Pool Approach

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.

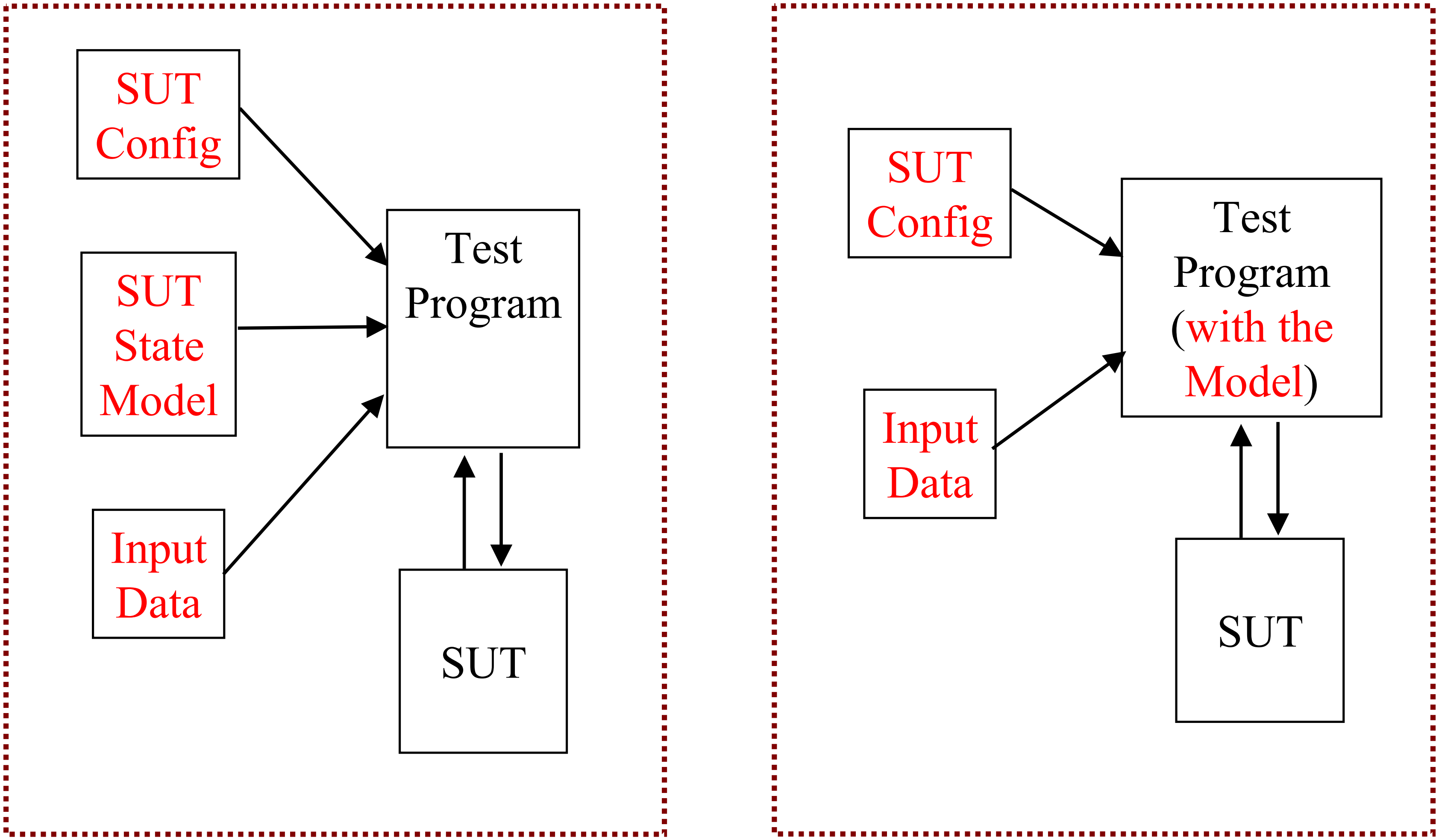
Automation Architecture Frameworks

Model Based

Model Based Automation

- **Test embodies rules for activities**
    - Stochastic process event generation
    - Well formed random values
- **Possible monitors**
    - Code assertions
    - Event logs
    - State transition maps
    - Other oracles

Model Based Architectures



Model Based Stochastic Tests

The General Approach

Build a state model of the software

- » The analysis will reveal several defects in itself
- » For any state
  - *list the actions the user can take*
  - *list the results of each action (what new state*
  - *identify what can indicate that we went to the correct new state*

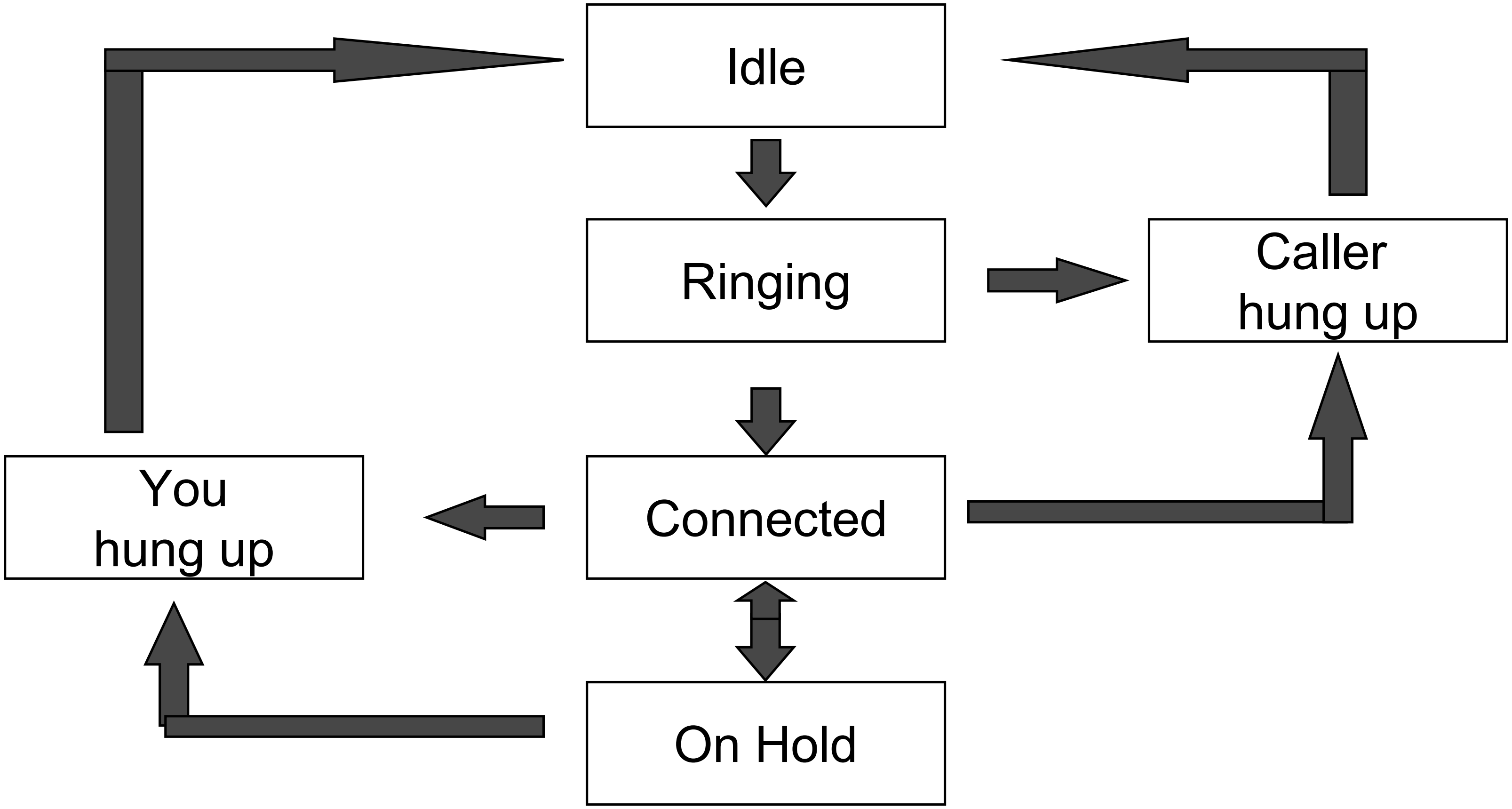
- Generate random event inputs to the program
- When the program responds by moving to a new state, check whether the program has reached the expected state

Random / Statistical Testing: Model-based Stochastic Tests

Some 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 or VCR)
- In general, well suited to a limited-functionality client that will not be reset very often

An Example State Machine



Refer to Testing Computer Software, pages 20-21

State Machine Matrix

Current State	Event	Next State	Activities
Idle	Incomming Call	Ringing	Busy 1 line
Ringing	Caller Hang Up	Caller Hung Up	Free 1 line
Ringing	Pick up Handset	Connected	Busy 1 line Busy Handset
Connected	You Hang Up	You Hung Up	Free 1 line
Connected	Put on Hold	On Hold	Free Handset
Connected	Caller Hang Up	Caller Hung Up	Free 1 line Free Handset
On Hold	Pick up Handset	Connected	Busy Handset
On Hold	You Hang Up	You Hung Up	Free 1 line Free Handset
You Hung Up	<time>	Idle	Free All
Caller Hung Up	<time>	Idle	Free All

Tabular form of the Telephone State Machine

Model Based Non-Stochastic Tests

The General Approach

- Input values conform to a model of the rules
  - » Mathematical expressions
  - » Function call selection and parameters
  - » Input value within ranges, data types
- Selection of events as inputs to the SUT
- May include positive and negative test values
- Parallel threads

Advanced Results Comparison Strategies

Test Oracles II

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 impractically 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

Self-Referential Strategies

- Embed expected results in the data

- Cyclic algorithms

- Shared keys with algorithms

Self Verifying Data

1. Generate a coded identifier when the test data is created (index or seed)

2. Attach the identifier to the data

3. Later, verify the data using the identifier

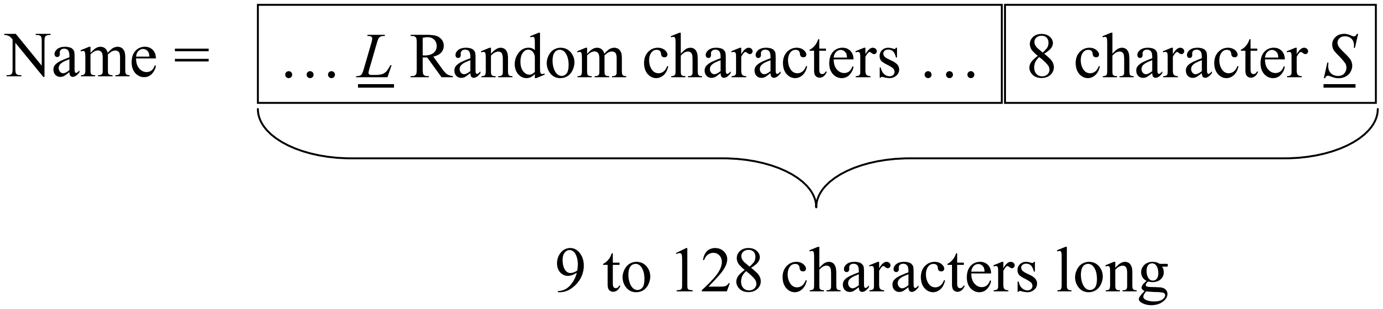
SVD Name Example 1/3

Generate a random name:

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

SVD Name Example 2/3

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


Name = … *L* Random characters … 8 character *S*

9 to 128 characters long

SVD Name Example 3/3

To verify the names:

- Extract the 8 character $\underline{S}$
- Use RANDOM($\underline{S}$) to generate the random name length $\underline{L}$
- Generate random string $\underline{N'}$ of length $\underline{L}$
- Compare the name $\underline{N}$ in the record to the new random string $\underline{N'}$

Non-Embedded SVD Examples

Shared value fields

- last names
- job titles
- Company name

Non-string data

- numeric values
- date fields

Limited length

- first name
- state

Add a new field to the data set for each record

Cyclic Algorithm Examples

1. Bit shifting

2. Pseudo-Random numbers

3. Adding or multiplying by a constant

Cyclic Algorithm
Data Generation Example

Generate random data packets

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

Shared Key and Algorithm Example

To verify the data packets

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

Self-Referential Oracle Examples

Data base

- embedded linkages

Data communications

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

Noel Nyman's "Self Verifying Data"*

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

Rule of thumb

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

Level 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 the data or SUT

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

Math functions

- sine function example

Heuristic Oracle Relationships

Well behaved

- follows heuristic rule for some range of values
- ranges are knowable
- few or no gaps to be excluded

Predictable

- identifiable patterns

Simple

- easy to compute or identify
- requires little or no information as input

Heuristic Oracle Drawbacks

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

Where Do We Fit In The Oracle?

- Identify what to verify
- How do we distinguish the “right answer”
- How close to “right” do we need
- Decide when to generate the expected results
- Decide how and where to verify results
- Acquire (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 oracles

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*

Automation Architecture and High-Level Design

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.

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 extremely useful

Key Automation Factors

- **Elements of the 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 Tools

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:

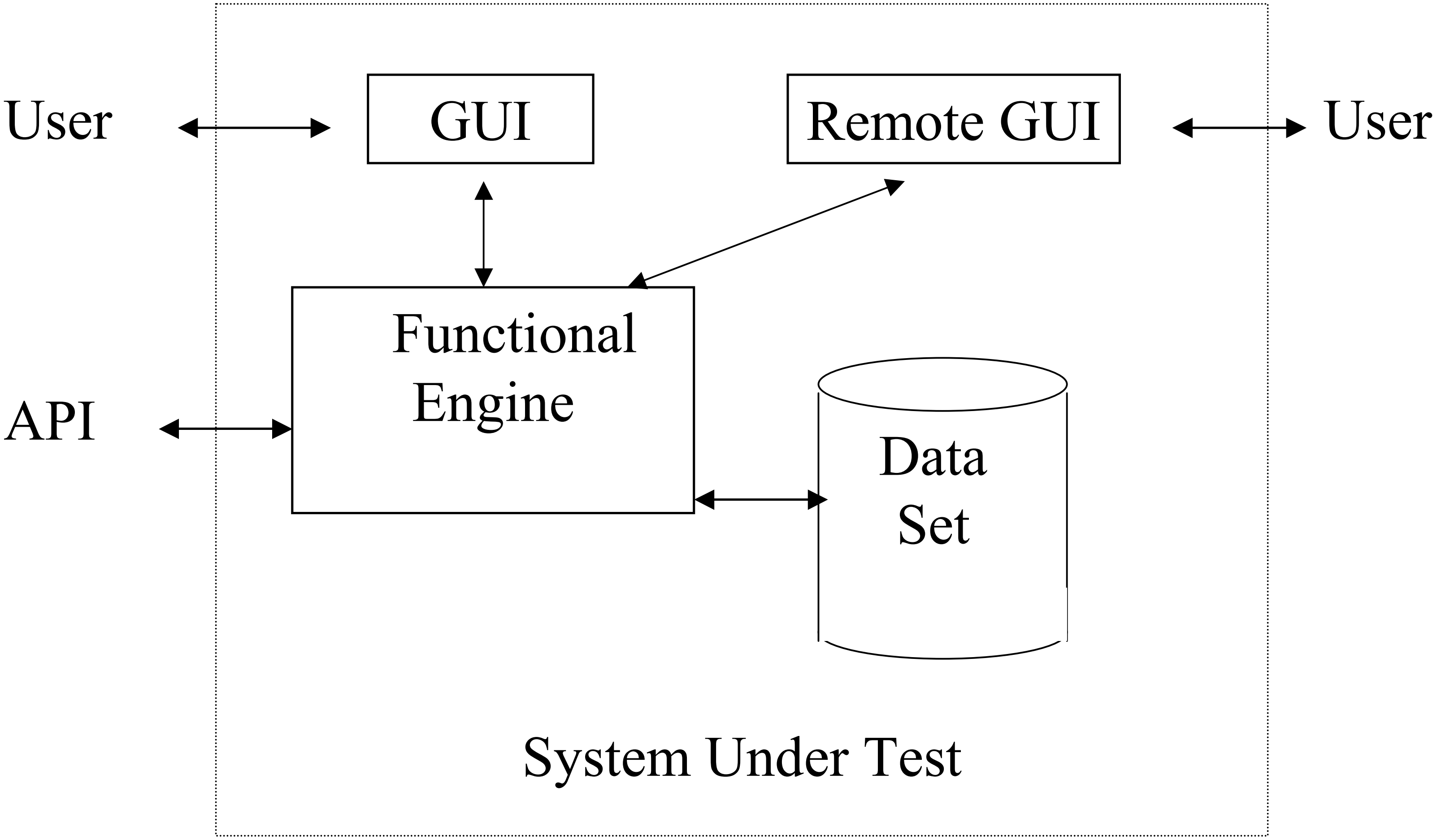
- capture/replay for easy manual creation of tests
- execution of test scripts
- recording of test events
- compare the test results with expected results
- report test results

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



User

GUI

Remote GUI

User

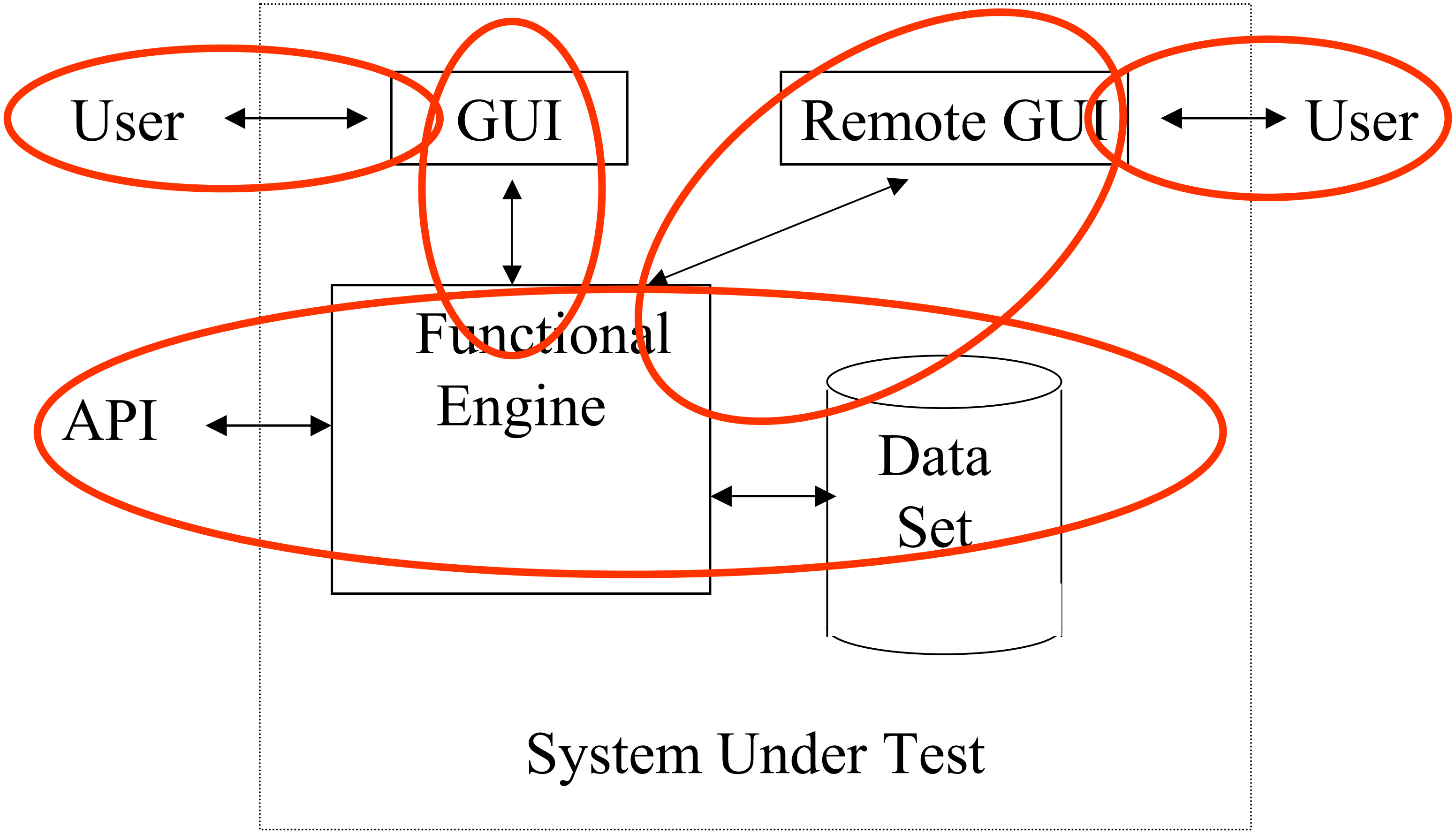
Functional Engine

API

Data Set

System Under Test

Breaking Down The Testing Problem



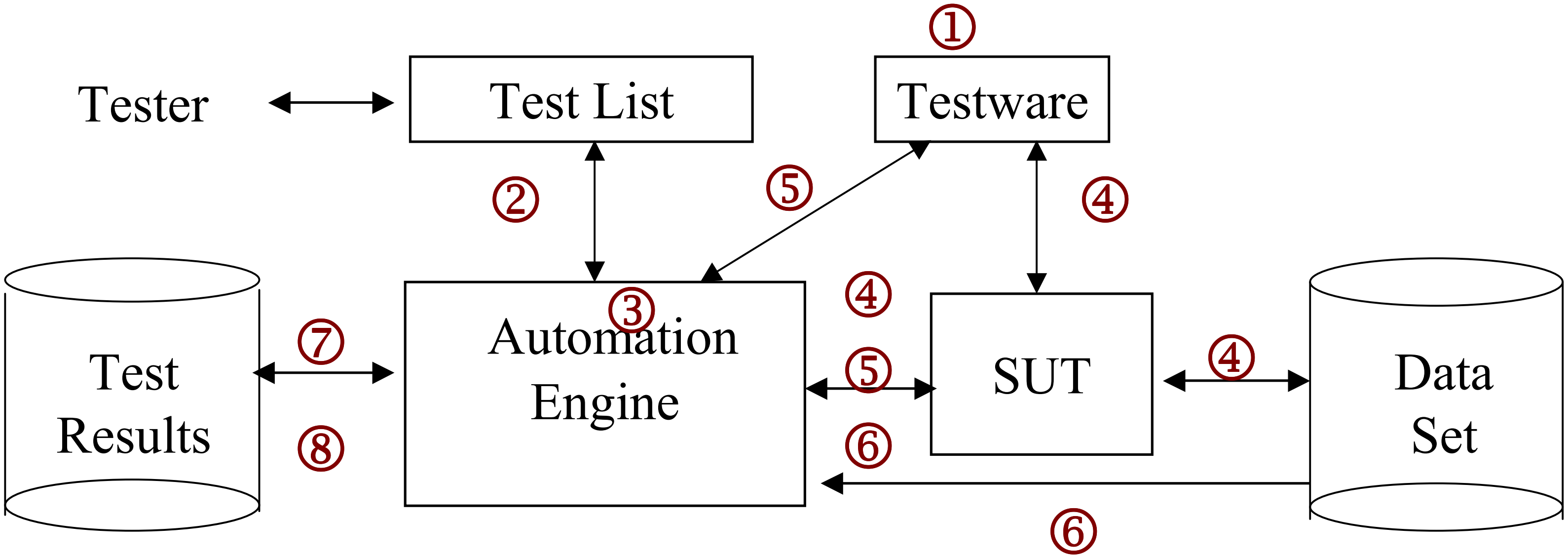
Identify Where To Monitor and Control

- 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 the computer to extend your reach

- Setting conditions
- Monitoring activities
- Controlling the 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

Using Pseudo Random Numbers

Input as an optional parameter

```
> run testprog
> run testprog -seed 4767
```

Check for “seed” value in the test

```
IF (Seed parameter present)
   seed = <input value>
ELSE
   seed = INTEGER(N * RANDOM()) + 1
   PRINT (seed)
ENDIF
   Rand = RANDOM(seed)
```

[where ***N*** is the largest seed value]

Mutating Automated Tests

Closely tied to instrumentation and oracles

Using pseudo random numbers

Positive and negative cases possible

Possibly including 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)

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*

Recap

Topics:

- Foundational Concepts in Software Testing
- About Regression Testing
- Automation of Tests
- Test Oracles
- Automation Architecture Approaches
- Advanced Comparison Strategies
- Automation Architectures and High Level Design

Summary

Take away:

- Software test automation is serious programming
- Automation can extend our reach way beyond manual testing
- Test oracles are key for success
- There are many automation approaches
- Advanced comparison strategies provide even more powerful possibilities
- Engineer the automation architecture

Notes

Test Automation

Testing Resources on the Net

Testing Resources on the Net
Personal Web Sites

Doug Hoffman **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 **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.pettichord.com**

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, articles, and checklists

Karen Johnson **www.KarennJohnson.com**

Testing services, training, and defect tracking products

Testing Resources on the Net
Personal Web Sites

Elisabeth Hendrickson	**www.QualityTree.com**

Consulting and training in software quality and testing

Mike Kelly	**www.MichaelDKelly.com**

Writes and speaks about topics in software quality and testing

Robert Sabourin	**www.AmiBug.com**

Testing services to help companies, and individuals, succeed in developing quality software solutions in a timely, and cost effective manner

Scott Barber	**www.PerfTestPlus.com**

Offers advising, consulting and training services as well as resources to bring software testing expertise and thought-leadership to organizations

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.

RBSC (Bob Binder)	**www.rbsc.com**

A different approach to object-oriented development and testing.

Testing Resources on the Net
Professional Not-For Profit Web Sites

SSQA **www.ssqa-sv.org**

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**

CSTER **http://www.TestingEducation.org**

The Center for Software Testing Education and Research is a Not-for-Profit organization that supports the teaching and self-study of software testing

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. Search engine for standards

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
Professional Not-For Profit Web Sites

The Association For Software Testing **www.AssociationforSoftwareTesting.org**

Not-for-Profit professional service organization dedicated to advancing the understanding and practice of software testing

LaRS at JPL **http://eis.jpl.nasa.gov/lars/**

The Laboratory for Reliable Software's mission is to achieve long term improvements in the reliability of JPL's mission critical software systems

SDForum **www.Center.org**

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

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

Society For Technical Communication **www.STC.org**

Links to research material on documentation process and quality

Software Publishers Association **www.SPA.org**

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

Testing Resources on the Net
Professional For Profit Web Sites

Software Research Inc.	**www.soft.com**

Also a consulting and tool building firm. Publishes the TTN-Online newsletter.

Quality Assurance Institute	**www.QAI.com**

Quality control focused on the needs of Information Technology groups.

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.softwaretestinginstitute.com**

Quality publications, industry research and online services for the software development and testing professional.

Software Quality Engineering	**www.sqe.com**
www.stqe.com	**www.stickyminds.com**

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

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"

Quality.Org	**www.quality.org**

Links to quality control source materials

Testing Resources on the Net
Other Interesting 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.

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.

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

Home of Ratbert, black box tester from Heck.

Notes