Black Box Software Testing

Douglas Hoffman
B.A., M.S., M.B.A., ASQ-CSQE, ASQ-CQMgr, ASQ Fellow
Consultant
Software Quality Methods, LLC, (SQM)

Winter, 2004

doug.hoffman@acm.org
www.SoftwareQualityMethods.com

Copyright © 1994-2004 SQM, LLC. All Rights Reserved.
Copyright Notice

These slides are distributed under the Creative Commons License.

In brief summary, you may make and distribute copies of these slides so long as you give the original author credit and, if you alter, transform or build upon this work, you distribute the resulting work only under a license identical to this one.

For the rest of the details of the license, see http://creativecommons.org/licenses/by-sa/2.0/legalcode.
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.

Professional

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

I’m in the business of improving software customer satisfaction.
I’ve 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 computers, software, developers, and customers.

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

Books
- *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* (2001; with James Bach and Bret Pettichord). This is an outstanding book covering practical approaches and techniques for all aspects of software testing.
Acknowledgment to James Bach

I'm in the business of questioning things.

I started in this business as a programmer. I like programming. But I find the problems of software quality analysis and improvement more interesting than those of software production. For me, there's something very compelling about the question "How do I know my work is good?" Indeed, how do I know anything is good? What does good mean? That's why I got into SQA, in 1987.

Today, I work with project teams and individual engineers to help them plan SQA, change control, and testing processes that allow them to understand and control the risks of product failure. I also assist in product risk analysis, test design, and in the design and implementation of computer-supported testing. Most of my experience is with market-driven Silicon Valley software companies like Apple Computer and Borland, so the techniques I've gathered and developed are designed for use under conditions of compressed schedules, high rates of change, component-based technology, and poor specification.

Current employment

• Principal at Satisfice

Book

• Lessons Learned in Software Testing (2001; with Cem Kaner and Bret Pettichord). This is an outstanding book covering practical approaches and techniques for all aspects of software testing.
## Contents

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>Introductory exercises</td>
<td>309 Paradigm: user</td>
</tr>
<tr>
<td>2</td>
<td>39</td>
<td>Completeness is impossible</td>
<td>313 Paradigm: scenario</td>
</tr>
<tr>
<td>3</td>
<td>55</td>
<td>Domain &amp; Boundaries</td>
<td>321 Paradigm: risk-based</td>
</tr>
<tr>
<td>4</td>
<td>105</td>
<td>Black Box Test Design I</td>
<td>347 Paradigm: stress</td>
</tr>
<tr>
<td>5</td>
<td>123</td>
<td>Overview of development</td>
<td>351 Paradigm: volume random</td>
</tr>
<tr>
<td>6</td>
<td>147</td>
<td>Black box testing group</td>
<td>372 Paradigm: state-models</td>
</tr>
<tr>
<td>7</td>
<td>159</td>
<td>Reusable Test Matrices</td>
<td>381 Paradigm: exploratory</td>
</tr>
<tr>
<td>8</td>
<td>167</td>
<td>Relationship Tables</td>
<td>11 441 Exploratory Testing</td>
</tr>
<tr>
<td>9</td>
<td>179</td>
<td>Bug Advocacy</td>
<td>12 499 Test Documentation</td>
</tr>
<tr>
<td>10</td>
<td>255</td>
<td>Testing Paradigms</td>
<td>13 505 Scripting Test Cases</td>
</tr>
<tr>
<td>270</td>
<td></td>
<td>Paradigm: domain testing</td>
<td>14 512 Requirements for Test Docs</td>
</tr>
<tr>
<td>277</td>
<td></td>
<td>Paradigm: function testing</td>
<td>15 527 Testing User Documentation</td>
</tr>
<tr>
<td>281</td>
<td></td>
<td>Paradigm: regression</td>
<td>16 537 Managing Test Automation</td>
</tr>
<tr>
<td>293</td>
<td></td>
<td>Paradigm: spec-driven</td>
<td>17 575 Test Strategy Planning</td>
</tr>
<tr>
<td>Page</td>
<td>Number</td>
<td>Chapter Title</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>----------------------------------</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>595</td>
<td>Planning Testing Projects</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>609</td>
<td>Metrics</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>647</td>
<td>Defect Life Cycles</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>657</td>
<td>Impossible Deadlines</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>671</td>
<td>Career planning for testers</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>679</td>
<td>Job Seeking</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>707</td>
<td>Recruiting testers</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>729</td>
<td>Learning styles</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>741</td>
<td>Resources on the Net</td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 1994-2004 Software Quality Methods, LLC. All Rights Reserved.
About This Course

This course is an introduction to black box software testing. Black box testing involves testing software from the customer’s view, without knowledge of the underlying code. The alternative to black box testing is glass box (design tests on the basis of your knowledge of the code). Both approaches find defects, and there are significant patterns in the kinds of defects that they find.

This course is based on Cem’s book (with Jack Falk and Hung Quoc Nguyen), *Testing Computer Software* (2nd edition). These lecture notes update the book. As an update, they provide much more material than we can cover in class.

This class is designed to be customized. From about 10 days worth of material, we’ll select a 3-day subset for classroom use. The rest is for you to study in your spare time.

A final note about the intent of the class. Because it is impossible to test everything, you will always have to make tradeoffs. For example, what is the right ratio of time spent documenting tests vs. actually doing testing? I’ll try to help you see the factors that questions like this depend on, and I’ll suggest specific answers for some specific circumstances. But at work, you’ll have to make your own evaluations of your own situation, and come up with your own answer. The course’s goal is to help you improve your ability to make judgments like these.

We revise this course frequently. Suggestions for improvement are always welcome, and they make a big difference. Thanks for coming, and for your feedback.
**Demographics:**

*How long have you worked in:*

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>software testing</td>
<td>0-3 months  ____  3-6 months  ____</td>
</tr>
<tr>
<td></td>
<td>6 mo-1 year  ____  1-2 years  ____</td>
</tr>
<tr>
<td></td>
<td>2-5 years  ____  &gt; 5 years  ____</td>
</tr>
<tr>
<td>programming</td>
<td>Any experience  ____</td>
</tr>
<tr>
<td></td>
<td>Production programming  ____</td>
</tr>
<tr>
<td>test automation</td>
<td>Test development  ____</td>
</tr>
<tr>
<td></td>
<td>Tools creation  ____</td>
</tr>
<tr>
<td>management</td>
<td>Testing group  ____</td>
</tr>
<tr>
<td></td>
<td>Any management  ____</td>
</tr>
<tr>
<td>marketing</td>
<td>____</td>
</tr>
<tr>
<td>documentation</td>
<td>____</td>
</tr>
<tr>
<td>customer care</td>
<td>____</td>
</tr>
<tr>
<td>traditional QC</td>
<td>____</td>
</tr>
</tbody>
</table>
Customization

Please vote for your top 10 topics

<table>
<thead>
<tr>
<th>Managing testing groups</th>
<th>Test strategy planning</th>
<th>Job Seeking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software life cycles</td>
<td>Black box automation</td>
<td>Recruiting software testers</td>
</tr>
<tr>
<td>Role of test group</td>
<td>Testing user documentation</td>
<td></td>
</tr>
<tr>
<td>Test case design</td>
<td>Metrics</td>
<td></td>
</tr>
<tr>
<td>Bug advocacy</td>
<td>Status reporting</td>
<td></td>
</tr>
<tr>
<td>Specification analysis</td>
<td>Defect life cycle</td>
<td></td>
</tr>
<tr>
<td>Exploratory testing</td>
<td>Planning / scheduling the testing project</td>
<td>Beta testing (in-house courses only)</td>
</tr>
<tr>
<td>Test planning (documentation)</td>
<td>Career paths for testers</td>
<td>Bug tracking system design (in-house)</td>
</tr>
</tbody>
</table>
Themes of the Course

The Themes

1. Management (not Testing) heads up QA.
2. Many development methods work.
3. Testing is done in context.
4. The program doesn’t work. Therefore, your task is to find errors.
5. Complete testing requires a nearly infinite series of tests.
6. Be methodical.
7. Look for powerful representatives of classes of tests.
8. Groups often develop a better range of insights than individuals.
10. Think in terms of quality costs.
11. Automation is software development.
12. Communication must be effective.
13. Change is normal.
14. Test planning can be evolutionary.
Themes of the Course

1. **Management (not Testing) heads up QA.**
A product’s head of QA is the person who makes the decisions that determine its quality. That happens long before the start of testing. Test a lousy program forever and you’ll end up with an expensive, well-tested, lousy program.

Watch out for project managers who tell you that YOU are responsible for the quality of the product (not them). It’s a game that can cause you excessive and undeserved stress. Your task is to find and clearly report software errors. (See *Testing Computer Software*, Chapter 15, and the Appendix paper on *Negotiating Testing Resources.*)

2. **Many development methods work.**
School training might lead you to believe that there are only a few “proper” ways to manage projects. In practice, there have been many successful approaches. Companies and project managers have their own styles. Some are much less formal than others. Some are quite formal, but aren’t in the traditional “waterfall” mold.

Some companies are appallingly disorganized, and some companies’ quality standards are terrible (whether their processes are organized or not).

You will certainly want to form your own opinion of the situations that you’re in, but take some time in forming your judgments and look carefully at the actual results of your company’s efforts. (See *Testing Computer Software*, page 255 onward.)
(Themes, continued . . .)

3. Testing is done in context

Rather than looking for The One True Way to run a test group, we should recognize that the testing group is a service organization and that its work is done in a context. Life cycles, testing group missions, testing techniques must fit within the overall organization. We should look at/for relevant factors (perform a requirements analysis) to guide our decisions.

4. The program doesn’t work. Therefore, your task is to find errors.

Nobody would pay you to test if their program didn’t have bugs. All programs have bugs. Any change to a program can cause new bugs, and any aspect of the program can be broken.

When someone tells you that your task is to “verify that the program is working” then they’re telling you that you’re going to fail in your task every time you find a bug. If they’re serious:

• you must be doing a final release test on files to go onto the master disks; OR
• the person who’s telling you this doesn’t understand your job; OR
• the programmer/project manager who’s telling you this is talking you out of doing your job; OR
• you should talk further with your manager to understand the charter of your group.

If you set your mind to showing that a program works correctly, you’ll be more likely to miss problems than if you want and expect the program to fail.

Copyright © Cem Kaner, All rights reserved.  Page 13
5. **Complete testing requires an infinite series of tests.**

There’s a nearly infinite number of paths through any non-trivial program, and a virtually infinite set of combinations of data that you can feed the program. Multiply that by the infinite number of possible configurations. You can’t test them all. No one can test them all.

Therefore, your task is to hunt for bugs -- not to find *all* the bugs (you won’t) and not to verify that the program has no bugs (it does).

Certainly, you want to find as many bugs as possible, you want to find the most serious bugs, and you want to find them as early as possible.

Your challenges will require judgment, tradeoffs, discipline, and efficiency.

6. **Be methodical.**

Black box testing isn’t just banging away at the keyboard. To have any hope of doing an efficient job you must be methodical:

- Break the testing project into tasks and subtasks so that you have a good idea of what has to be done. (It’s handy to break down work into 4-hour-sized jobs.)
- Track what you’ve done so you can avoid:
  - duplicated effort
  - misunderstood depth and extent of testing
  - missing areas or testing them late.
- Prioritize the tasks.
7. Look for powerful representatives of classes of tests.

Two tests belong to the same *equivalence class* if you expect the same result (pass / fail) of each. An equivalence class might include thousands (or trillions) of test cases. You don’t have time to test them all. Instead, pick a few representative members of the class.

*Boundaries* mark a point or zone of transition from one equivalence class to another (such as the largest number you are allowed to enter into a field, and the number that is 1 larger than the largest valid value). This is where the program is most likely to fail, so these are your best selections from an equivalence class.

Some classes have no firm boundaries (imagine compatibility testing, and the class of LaserJet 4-compatible printers). In this case, pick representative cases, but look for worst cases (such as an almost-compatible).

8. Groups often develop a better range of insights than individuals.

Testers working together in pairs often find more bugs and more interesting bugs than they would find on their own. Testers brainstorming as a group generate a wider range of dimensions or risks to investigate. In general, group work can facilitate the development of creative ideas, while keeping the group focused on the task at hand. Testing in pairs is also effective for training testers who are new or new to the project.
9. **Cost-justify your processes.**

Test groups are often asked to do several tasks that don’t directly help them find and report bugs. Every minute that you spend on such tasks is a minute that you aren’t finding bugs, reporting bugs, or finding a more efficient way to find or report bugs. These tasks might have great value, but you must evaluate them carefully. This includes the detailed documentation of your testing process.

You might not be the person who does the cost/benefit analysis, but you can still suggest ways to improve your efficiency to your manager, and you should realize that reasonable cost/benefit considerations might have led your company to drop paperwork that testing courses or books might have led you to expect.

10. **Think in terms of quality costs.**

Testing accounts for only *some* of the money that your company spends on quality. Quality-related costs include:

- **Prevention costs:** Cost of preventing customer dissatisfaction, including errors or weaknesses in software, design, documentation, and support.
- **Appraisal costs:** Cost of inspection (testing, reviews, etc.).
- **Internal failure costs:** Cost of dealing with errors discovered during development and testing.
- **External failure costs:** Cost of dealing with errors that affect your customers, after the product is released.

Analyzing situations in these terms will give you a broader understanding, and will often help you make your arguments more effectively or find allies for your position.
11. Automation is software development.

Test automation is like any form of automation.

Many test groups attempt to automate with minimal planning, inadequate budgets, absurd schedules, no underlying architecture, and no documentation. These efforts fail -- if not immediately, then when the test group tries to reuse the tests and discovers they provide an unknown level of coverage and are unmaintainable.

When you automate, use the development methods you expect of the development teams whose work you test.

12. Communication must be effective.

From your first day on the job, you are writing reports that could end up on the company president’s desk, or in court.

Your bug reports advise people of difficult situations -- the problems they report can affect the project schedule, hurt the company’s cash flow, get someone fired. The clearer your reports are, the more likely it is that the company will make reasonable business decisions about them.

Computer Science majors are often poorly trained in persuasive writing, technical writing, oral argument, and face-to-face negotiation. These are core skills for your job.
(Themes, continued . . .)

13. Change is normal.
Projects’ requirements change. The market changes. And we come to understand the product more thoroughly as we build it.

Some companies accept late changes into their products as a matter of course. As a new tester, you might decide quickly that this is a poor way to do business. It might be, and it might not be. Get to know this company and its business before you let yourself get negative.

If your company does change specifications and design as it goes, there is no point standing in front of the train screaming “Stop changing!” You have to take steps to make yourself more effective in dealing with late changes.

14. Test planning can be evolutionary.

Rather than designing all of your test cases and then running your tests, you can develop the test plan gradually, in parallel with the testing effort.

If you write your full plan, then develop all your cases, then test, much of the planning and development might be made obsolete by program design changes before you run a single test on the changed part of the program.

An evolutionary approach can help you save you a lot of wasted time if the product’s design changes during the alpha and beta test phases, because your test plan immediately leads to test cases, which are immediately run.
Black Box Testing Defined

Black Box Testing is based on externally observed characteristics of the program we are testing. The tests are designed without regard to how the program works internally.

Glass Box Testing (a.k.a. White Box) draws on knowledge of the internal characteristics of the program. The tests are designed to exercise specific parts of the code using that knowledge.
Part 1.

Introductory Exercises
Triangle Problem

The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

» From Glen Myers, The Art of Software Testing

- Write a set of test cases that would adequately test this program.
- Please write your name on your answer so that we can hand it back to you. Hand it in when you are done.
Notes about the Triangle Problem

Several classes of issues were missed by most students. For example:

• Few students checked whether they were producing valid triangles. (1,2,3) and (1,2,4) cannot be the lengths of any triangle.

  » Knowledge of the subject matter of the program under test will enable you to create test cases that are not directly suggested by the specification. If you lack that knowledge, you will miss key tests. (This knowledge is sometimes called “domain knowledge,” not to be confused with “domain testing.”)

• Few students checked non-numeric values, bad delimiters, or non-integers.

• The only boundaries tested were at MaxInt or 0.

We will revisit this example in Domain Testing
Does font size work in Open Office?

What’s your oracle?
An oracle is the principle or mechanism by which you recognize problems.

“..it works”

Means:

“...it appeared to meet some requirement to some degree.”
OK, so what about WordPad?
What if we compared WordPad to Word?
What does this tell us?

WordPad

Word
Testing is about ideas.

Heuristics give you ideas.

• A heuristic is a fallible idea or method that may help solve a problem.
• You don’t comply with a heuristic; you apply it. Heuristics can hurt you when elevated to the status of authoritative rules.
• Heuristics represent wise behavior only in context. They do not contain wisdom.
• Your relationship to a heuristic is the key to applying it wisely.

“Heuristic reasoning is not regarded as final and strict but as provisional and plausible only, whose purpose is to discover the solution to the present problem.”

- George Polya, How to Solve It
Some useful oracle heuristics

**Consistent with History:** Present function behavior is consistent with past behavior.

**Consistent with our Image:** Function behavior is consistent with an image that the organization wants to project.

**Consistent with Comparable Products:** Function behavior is consistent with that of similar functions in comparable products.

**Consistent with Claims:** Function behavior is consistent with documented or advertised behavior.

**Consistent with Specifications or Regulations:** Function behavior is consistent with claims that must be met.

**Consistent with User’s Expectations:** Function behavior is consistent with what we think users want.

**Consistent within Product:** Function behavior is consistent with behavior of comparable functions or functional patterns within the product.

**Consistent with Purpose:** Function behavior is consistent with apparent purpose.
Some font size testing issues…

What to cover?

- Every font size (to a tenth of a point).
- Every character in every font.
- Every method of changing font size.
- Every user interface element associated with font size functions.
- Interactions between font sizes and other contents of a document.
- Interactions between font sizes and every other feature in Wordpad.
- Interactions between font sizes and graphics cards & modes.
- Print vs. screen display.

What’s your oracle?

- What do you know about typography?
- Definition of “point” varies. There are as many as six different definitions (http://www.oberonplace.com/dtp/fonts/point.htm)
- Absolute size of characters can be measured, but not easily (http://www.oberonplace.com/dtp/fonts(fontsize.htm)
- How closely must size match to whatever standard is chosen?
- Heuristic approaches: relative size of characters; comparison to MS Word.
- Expectations of different kinds of users for different uses.
A Thought Experiment

(I’ve never tried anything like this on Quicken. This example is purely hypothetical.)

• Imagine testing Quicken. I understand that it can handle an arbitrarily large number of checks, limited only by the size of the hard disk. So, being a tester, you go buy the biggest whopping hard disk on the market (about 120 gig at the moment, I think) and then write a little program to generate a Quicken-like data file with many checks.

• Now, open Quicken, load that file, and add 1 more check. Let’s pretend that it works — sort of. The clock spins and spins while Quicken sorts the new check into place. At the end of four days, you regain control.

• So you write a bug report— “4 days to add a check is a bit too long.”

• The programmers respond— “Not a bug. Only commercial banks manage a billion checks. If you’re Citibank, use a different program.”

Suppose that you decided to follow this up because you thought the programmers are dismissing a real problem without analysis. What would you do?
Risk as a simplifying factor

For *Wordpad*, we don’t care if font size meets precise standards of typography!

In general it can vastly simplify testing if we focus on whether the product has a problem that matters, rather than whether the product merely satisfies all relevant standards.

Effective testing requires that we understand standards as they relate to how our clients value the product.

Instead of thinking **pass** vs. **fail**,
Consider thinking **problem** vs. **no problem**.
Risk as a simplifying factor

What if we applied the same evaluation approach

• that we applied to WordPad
• to Open Office or MS Word or Adobe PageMaker?

The same tests or results lead to different conclusions in different contexts
The oracle problem and test automation

We often hear that all testing should be automated. Automated testing depends on our ability to programmatically detect when the software under test fails a test.

Our ability to automate testing is fundamentally constrained by our ability to create and use oracles.
The oracle problem and automation

One common way to define an oracle is as a source of expected results.

- Under this view, you compare your results to those obtained from the oracle. If they match, the program passes. If they don’t match, the program fails.
- The comparison between MS WordPad and MS Word illustrates this approach.
- *It is important to recognize that this evaluation is heuristic:*
  - **We can have false alarms:** A mismatch between WordPad and Word might not matter.
  - **We can miss bugs:** A match between WordPad and Word might result from the same error in both programs.
Automated tests narrow the oracle's scope

An automated test is not equivalent to the most similar manual test:

- The mechanical comparison is typically more precise (and will be tripped by irrelevant discrepancies)
- The skilled human comparison will sample a wider range of dimensions, noting oddities that one wouldn't program the computer to detect.
Black Box Software Testing

Part 2.

The Impossibility of Complete Testing

Selected Readings:

- Hoffman, Bug Report (Exhausting Your Test Options)
- Kaner, The Impossibility of Complete Testing
- Kaner, Software Negligence & Testing Coverage
The Impossibility of Complete Testing

If you test completely, then at the end of testing, there cannot be any undiscovered errors. (After all, if there are more bugs, you can find them if you do more testing. So testing couldn't yet be "complete.") This is impossible because:

1. The domain of possible inputs is too large.
2. There are too many combinations of data to test.
3. There are too many possible paths through the program to test.
4. And then there are the user interface errors, the configuration/compatibility failures, and dozens of other dimensions of analysis.
1. **Too Many Inputs**

The domain of possible inputs is too large

- Valid inputs
  - Many variables
  - Huge number of possible values
  - In some cases, complete testing makes sense.

At MASPAR we made a Massively Parallel computer, up to 65K parallel processors. To test the 32-bit integer square root function, we checked all values (all 4,294,967,296 of them). This took the computer about 6 minutes to run the tests and compare the results to an oracle. There were 2 (two) errors, neither of them near any boundary. (The underlying error was that a bit was sometimes mis-set, but in most of these cases, there was no effect on the final calculated result.) Without an exhaustive test, these errors probably wouldn’t have shown up. But what about the 64-bit integer square root? How could we find the time to run all of these?
1. Too Many Inputs

- Invalid inputs
  » Don’t forget Easter Eggs.
- Edited inputs
  » These can be quite complex. How much editing is enough?
- Variations on input timing
  » Enter data very quickly or very slowly.
  » Try testing one event just before, just after, and in the middle of processing a second event.
- Now, what about all the error handling that you can trigger with "invalid" inputs?
  » Think about Whittaker & Jorgensen's constraint-focused attacks (Whittaker, How Software Fails)
  » Think about Jorgensen's hostile data stream attacks
1. 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 your domain size?

We can only run a vanishingly small portion of the possible tests
2. Too Many Combinations

Variables interact.

- For example, a program crashes when attempting to print preview a high resolution (say, 600x600 dpi) image on a high resolution screen. The option selections for printer resolution and screen resolution are interacting.
- For example, a program fails when the sum of a series of variables is too large.

Suppose there are N variables. Suppose the number of choices for the variables are $V_1$, $V_2$, through $V_N$. The total number of possible combinations is $V_1 \times V_2 \times \ldots \times V_N$. This is huge.

We saw 39,601 combinations of just two variables whose values could range only between -99 and 99.

Here’s a case that isn’t so trivial. There are 318,979,564,000 possible combinations of the first four moves in chess.
3. Too Many Paths

Here’s an example that shows that there are too many paths to test in even a fairly simple program. This one is from Myers, *The Art of Software Testing*.
3. Too Many Paths

The program starts at A.
From A it can go to B or C
From B it goes to X
From C it can go to D or E
From D it can go to F or G
From F or from G it goes to X
From E it can go to H or I
From H or from I it goes to X
From X the program can go to
EXIT or back to A. It can go back
to A no more than 19 times.

One path is ABX-Exit. There are 5 ways to get to X and then to the EXIT in one pass.
Another path is ABXACDFX-Exit. There are 5 ways to get to X the first time, 5 more to get back to X the second time, so there are 5 x 5 = 25 cases like this.
3. *Too Many Paths*

There are $5^1 + 5^2 + \ldots + 5^{19} + 5^{20} = 10^{14}$

= 100 trillion paths through the program to test, or approximately one billion years to try every path if one could write, execute and verify a test case every five minutes.
3. Sequences –
The Telephone Example

- Idle
- Ringing
- Connected
- On Hold
- Caller hung up
- You hung up
3. Sequences –
   The Telephone Example

Why are we spending so much time on this crazy example?
Because it illustrates several important points:

- Simplistic approaches to path testing can miss critical defects.
- Critical defects can arise under circumstances that appear (in a test lab) so specialized that you would never intentionally test for them.
- When (in some future course or book) you hear a new methodology for combination testing or path testing, I want you to test it against this defect. If you have no suspicion that there is a stack corruption problem in this program, will the new method lead you to find this bug?

This example also lays a foundation for our introduction to random/statistical testing.
The Telephone Example

- Idle
- Ringing
- Connected
- On Hold
- Caller hung up

You hung up
Let's consider the nature of the infinite set of tests

There are enormous numbers of possible tests. To test everything, you would have to:

• Test every possible input to every variable (including output variables and intermediate results variables).
• Test every possible combination of inputs to every combination of variables.
• Test every possible sequence through the program.
• Test every hardware / software configuration, including configurations of servers not under your control.
• Test every way in which any user might try to use the program.
Complete coverage?

Some people (attempt to) simplify away the problem of complete testing by advocating "complete coverage."

What is coverage?

- Extent of testing of certain attributes or pieces of the program, such as statement coverage or branch coverage or condition coverage.
- Extent of testing completed, compared to a population of possible tests.

Typical definitions are oversimplified. They miss, for example,

- Interrupts and other parallel operations
- Interesting data values and data combinations
- Missing code

The number of variables we might measure is stunning. I (Kaner) listed 101 examples in Software Negligence & Testing Coverage.
**Expanded Black Box Testing Model**

- **Test Inputs**
  - Precondition Data
  - Precondition Program State
  - Environmental Inputs

- **System Under Test**

- **Test Results**
  - Post-condition Data
  - Post-condition Program State
  - Environmental Results
Conclusion

Complete testing is impossible

• *There is no simple answer for this.*

• *There is no simple, comprehensive oracle to deal with it.*

• *Therefore testers live and breathe tradeoffs.*

Our job isn’t to design and do all the tests - it’s to choose and do the most important tests.
Black Box Software Testing

Part 3.
Domain Testing

Selected Readings:

- Cem Kaner, Liability for Product Incompatibility
- 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

• AKA partitioning, equivalence analysis, boundary analysis

• Fundamental question or goal:
  • This confronts the problem that there are too many test cases for anyone to run. This is a stratified sampling strategy that provides a rationale for selecting a few test cases from a huge population.

• General approach:
  • Divide the set of possible values of a field into subsets, pick values to represent each subset. The goal is to find a “best representative” for each subset, and to run tests with these representatives. Best representatives of ordered fields will typically be boundary values.
  • Multiple variables: combine tests of several “best representatives” and find a defensible way to sample from the set of combinations.

• Paradigmatic case(s)
  • Equivalence analysis of a simple numeric field.
  • Printer compatibility testing (multidimensional variable, doesn’t map to a simple numeric field, but stratified sampling is essential.)
Example Test Series

Here is the program’s specification:

• This program is designed to add two numbers, which you will enter
• Each number should be one or two digits
• The program will print the sum. Press Enter after each number
• To start the program, type ADDER

Before you start testing, do you have any questions about the spec?

Refer to Testing Computer Software, Chapter 1, page 1
Here’s a basic strategy for dealing with new code:

1. Start with mainstream-level tests. *Test the program with easy-to-pass values that will be taken as serious issues if the program fails.*

2. Test each function sympathetically. *Learn why this feature is valuable before you criticize it.*

3. Test broadly, rather than deeply. *Check out all parts of the program quickly before focusing.*

4. Look for more powerful tests. *Once the program can survive the easy tests, put on your thinking cap and look systematically for challenges.*

5. Pick boundary conditions. *There will be too many good tests. You need a strategy for picking and choosing.*

6. Do some exploratory testing. *Run new tests every week, from the first week to the last week of the project.*

Refer to Testing Computer Software, Chapter 1
1. A Mainstream Level Test

For the first test, try a pair of easy values, such as 3 plus 7.

Here is the screen display that results from that test.

Are there any bug reports that you would file from this?

Refer to Testing Computer Software, Chapter 1
2. Test each function sympathetically

- Why is this function here?
- What will the customer want to do with it?
- What is it about this function that, once it is working, will make the customer happy?

Knowing what the customer will want to do with the feature gives you a much stronger context for discovering and explaining what is wrong with the function, or with the function's interaction with the rest of the program.
3. Test broadly before deeply

- The objective of early testing is to flush out the big problems as quickly as possible.
- You will explore the program in more depth as it gets more stable.
- There is no point hammering a design into oblivion if it is going to change. Report as many problems as you think it will take to force a change, and then move on.
4. Look for More Powerful Tests

More likely to encounter and recognize errors:

- Do more iterations (more classes per function).
- Do more combinations (interactions between functions).
- Do more things (functional breadth).
- Methodically cover the code (look everywhere).
- Look for specific errors (similar products’ problems).
- Check for more symptoms (more types of errors).
- Try to break it (take a perverse view, get creative).
4a. Look for More Powerful Tests

Brainstorming Rules:

• The goal is to get lots of ideas. You are brainstorming together to discover categories of possible tests.
• There are more great ideas out there than you think.
• Don’t criticize others’ contributions.
• Jokes are OK, and are often valuable.
• Work later, alone or in a much smaller group, to eliminate redundancy, cut bad ideas, and refine and optimize the specific tests.
• Facilitator and recorder keep their opinions to themselves.
4. Look for More Powerful Tests

<table>
<thead>
<tr>
<th>What?</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Refer to Testing Computer Software, pages 4-5, for examples.
5. Equivalence Class & Boundary Analysis

There are 199 x 199 = 39,601 test cases for valid values:

• 99 values: 1 to 99
• 1 value: 0
• 99 values: -99 to –1

__________
199 values per variable
199 x 199 = 39,601 possible tests

So should we test them all?

We tested 3 + 7. Should we also test

• 4 + 7? 4 + 6?
• 2 + 7? 2 + 8?
• 3 + 8? 3 + 6?

Why?
5. Reducing the testing burden

There are too many possible tests.
There are $199 \times 199 = 39,601$ test cases for valid values:
There are infinitely many invalid cases:
- 100 and above
- -100 and below
- anything non-numeric

We cannot afford to run every possible test. We need a method for choosing a few tests that will represent the rest. Equivalence analysis is the most widely used approach.

• refer to Testing Computer Software pages 4-5 and 125-132
5. Classical Equivalence Class and Boundary (Domain) Analysis

- To avoid unnecessary testing, partition (divide) the range of inputs into groups of equivalent values.
- Then treat any input value from the equivalence class as representative of the full group.
- Two tests are equivalent if we would expect that an error revealed by one would be revealed by the other.
- If you can map the input space to a number line, then boundaries mark the point or zone of transition from one equivalence class to another. These are good members of equivalence classes to use since a program is more likely to fail at a boundary.

Myers, Art of Software Testing, 45
5. Traditional Approach (Myers)

One input or output field

- The “valid” values for the field fit within one (1) easily specified range.
- Valid values: -99 to 99
- Invalid values
  - < -99
  - > 99
  - non-numeric
  - expressions
  - long delays between digits
  - too many, too few values
5. Myers’ boundary table

<table>
<thead>
<tr>
<th>Variable</th>
<th>Valid Case Equivalence Classes</th>
<th>Invalid Case Equivalence Classes</th>
<th>Boundaries and Special Cases</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>First number</td>
<td>-99 to 99</td>
<td>&gt; 99</td>
<td>99, 100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt; -99</td>
<td>-99, -100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>non-integer</td>
<td>null entry</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>Second number</td>
<td>same as first</td>
<td>same as first</td>
<td>same</td>
<td></td>
</tr>
</tbody>
</table>

The traditional analysis would look at the potential numeric entries and partition them the way the specification would partition them.
5. **Myers’ boundary table (continued)**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Valid Case Equivalence Classes</th>
<th>Invalid Case Equivalence Classes</th>
<th>Boundaries and Special Cases</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>First number</strong></td>
<td>-99 to 99</td>
<td>&gt; 99</td>
<td>99, 100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; -99</td>
<td>&lt; -99</td>
<td>-99, -100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>non-integer</td>
<td>null entry</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>non-number</td>
<td>2.5</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td></td>
<td>expressions</td>
<td>:</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td><strong>Second number</strong></td>
<td>same as first</td>
<td>same as first</td>
<td>same</td>
<td></td>
</tr>
</tbody>
</table>

It might be useful to consider some other cases, such as special cases that are inside the range (e.g. 0) and errors on a different dimension from your basic "too big" and "too small".
## 5. Myers’ boundary table

<table>
<thead>
<tr>
<th>Variable</th>
<th>Valid Case Equivalence Classes</th>
<th>Invalid Case Equivalence Classes</th>
<th>Boundaries and Special Cases</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>First number</td>
<td>-99 to 99</td>
<td>&gt; 99</td>
<td>99, 100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt; -99</td>
<td>-99, -100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>non-number expressions</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>null entry</td>
<td></td>
</tr>
<tr>
<td>Second number</td>
<td>same as first</td>
<td>same as first</td>
<td>same</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>-198 to 198</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We should also consider other variables, not just inputs. For example, think of output variables, interim results, variables as things that are stored, and variables as inputs to a subsequent process.

---See Whittaker, *How to Break Software*. 
Boundary table as a test plan component

• Makes the reasoning obvious.
• Makes the relationships between test cases fairly obvious.
• Expected results are pretty obvious.
• Several tests on one page.
• Can delegate it and have tester check off what was done. Provides some limited opportunity for tracking.
• Not much room for status.

----------------------------------------

Question, now that we have the table, must we do all the tests? What about doing them all each time (each cycle of testing)?
5. Building the Table (in practice)

- Relatively few programs will come to you with all fields fully specified. Therefore, you should expect to learn what variables exist and their definitions over time.
- To build an equivalence class analysis over time, put the information into a spreadsheet. Start by listing variables. Add information about them as you obtain it.
- The table should eventually contain all variables. This means, all input variables, all output variables, and any intermediate variables that you can observe.
- In practice, most tables that I’ve seen are incomplete. The best ones that I’ve seen list all the variables and add detail for critical variables.
6. Exploratory Testing

Exploratory testing involves simultaneously learning, planning, running tests, and reporting/troubleshooting results.

A common goal of exploration is to probe for weak areas of the program.

Proportions do (and should) vary across the industry. Here is my preference per week:

- 25% of the group’s time developing new tests
- 50% executing old tests (including bug regression)
- 25% on exploratory testing

I recommend running new tests even in the very last days before releasing the product.

Different testers do more or less well at exploratory testing. Don’t assign 25/50/25 to everyone. Some people should do almost no exploration. Others should do it nearly full-time.
Second Cycle of Testing

When you get a new build, look over the responses to the problems you’ve already reported:

• Immediately retest bugs that the programmers say they fixed.
  If you catch a bad fix right away, the programmer is likely to remember what she did. She can deal with it immediately. If you wait a few days, she’ll forget details and wait until she has time to rethink and reinvestigate the problem.

• Plan your responses to deferred and rejected bugs.
  Don’t challenge every deferral. Pick your battles or no one will listen to you. Plan your approach—*If you’re going to fight, win*. Figure out what additional information would convince the other members of the project team that the problem should be fixed. In any case, think about other tests that the deferred bug suggests. The bug you reported might not be the worst of its line.

refer to Testing Computer Software, pages 11-15
Triangle Problem, Revisited

The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

» From Glen Myers, *The Art of Software Testing*

- **Write a set of test cases that would adequately test this program.**

  This is a classic example of domain partitioning.
Myers’ Answer

1. Test case for a valid scalene triangle
2. Test case for a valid equilateral triangle
3. Three test cases for valid isosceles triangles (a=b, b=c, a=c)
4. One, two or three sides are zero length (7 cases)
5. One side has a negative length
6. Sum of two sides equals the third (e.g. 1,2,3) is invalid because it’s not a triangle (3 permutations a+b=c, a+c=b, b+c=a)
7. Sum of two numbers is less than the third (e.g. 1,2,4) (3 permutations)
8. Non-integer value
9. Wrong number of values (too many, too few)
Example Values

1. {5,6,7}
2. {15,15,15}
3. {3,3,4; 5,6,6; 7,8,7}
4. {0,1,1; 2,0,2; 3,2,0; 0,0,9; 0,8,0; 11,0,0; 0,0,0}
5. {3,4,-6}
6. {1,2,3; 2,5,3; 7,4,3}
7. {1,2,4; 2,6,2; 8,4,2}
8. {Q,2,3}
9. {2,4; 4,5,5,6}
Exercise: List 10 tests that you’d run that aren’t in Myers’ list.
Exercise

For each of the following, list

- The variable(s) of interest
- The valid and invalid classes
- The boundary value test cases.

1. FoodVan delivers groceries to customers who order food over the Net. To decide whether to buy more vans, FV tracks the number of customers who call for a van. A clerk enters the number of calls into a database each day. Based on previous experience, the database is set to challenge (ask, “Are you sure?”) any number greater than 400 calls.

2. FoodVan schedules drivers one day in advance. To be eligible for an assignment, a driver must have special permission or she must have driven within 30 days of the shift she will be assigned to.
What is equivalence?

4 views of what makes values equivalent. Each has practical implications

• **Intuitive Similarity**: two test values are equivalent if they are so similar to each other that it seems pointless to test both.
  » This is the earliest view and the easiest to teach
  » Little guidance for subtle cases or multiple variables

• **Specified As Equivalent**: two test values are equivalent if the specification says that the program handles them in the same way.
  » Testers complain about missing specifications may spend enormous time writing specifications
  » Focus is on things that were specified, but there might be more bugs in the features that were un(der)specified
Understanding domain testing

What is equivalence?

- **Equivalent Paths:** two test values are equivalent if they would drive the program down the same path (e.g. execute the same branch of an IF)
  - Tester should be a programmer
  - Tester should design tests from the code
  - Some authors claim that a complete domain test will yield a complete branch coverage.
  - No basis for picking one member of the class over another.
  - Two values might take program down same path but have very different subsequent effects (e.g. timeout or not timeout a subsequent program; or e.g. word processor's interpretation and output may be the same but may yield different interpretations / results from different printers.)
Understanding domain testing

What is equivalence?

- **Risk-Based**: two test values are equivalent if, given your theory of possible error, you expect the same result from each.
  - Subjective analysis, differs from person to person. It depends on what you expect (and thus, what you can anticipate).
  - Two values may be equivalent relative to one potential error but non-equivalent relative to another.
Understanding domain testing

Test which values from the equivalence class?

Most discussions of domain testing start from several assumptions:

- The domain is continuous  
  [This is easily relaxed -- CK]
- The domain is linearizable (members of the domain can be mapped to the number line) or, at least, the domain is an ordered set (given two elements, one is larger than the other or they are equal)
- The comparisons that cause the program to branch are simple, linear inequalities

“It is possible to move away from these assumptions, but the cost can be high.” --- Clarke, Hassell, & Richardson, p. 388

If we think in terms of paths, can we use any value that drives the program down the correct path? This approach is common in coverage-focused testing. Unfortunately, it doesn't yield many failures. See Hamlet & Taylor

If you can map the input space to a number line, then boundaries mark the point or zone of transition from one equivalence class to another. These are said to be good members of equivalence classes to use because the program is more likely to fail at a boundary.
Understanding domain testing

Test which values from the equivalence class?
The program is more likely to fail at a boundary?

• Suppose program design:
  » INPUT < 10 result: Error message
  » 10 <= INPUT < 25 result: Print "hello"
  » 25 >= INPUT result: Error message

• Some error types
  » Program doesn't like numbers
    • Any number will do
  » Inequalities mis-specified (e.g. INPUT <= 25 instead of < 25)
    • Detect only at boundary
  » Boundary value mistyped (e.g. INPUT < 52, transposition error)
    • Detect at boundary and any other value that will be handled incorrectly

• Boundary values (here, test at 25) catch all three errors
• Non-boundary values (consider 53) may catch only one of the three errors
Understanding domain testing

Test which values from the equivalence class?

• The emphasis on boundaries is inherently risk-based

• But the explicitly risk-based approach goes further
  » Consider many different risks
  » Partitioning driven by risk
  » Selection of values driven by risk:
    • *A member of an equivalence class is a best representative (relative to a potential error) if no other member of the class is more likely to expose that error than the best representative.*

  **Boundary values are often best representatives**
  **We can have best representatives that are not boundary values**
  **We can have best representatives in non-ordered domains**
**Expanding the Notion of Equivalence**

Consider these cases. Are these paired tests equivalent?

<table>
<thead>
<tr>
<th>If you tested</th>
<th>Would you test</th>
</tr>
</thead>
<tbody>
<tr>
<td>51+52</td>
<td>52+53</td>
</tr>
<tr>
<td>53+54</td>
<td>54+55</td>
</tr>
<tr>
<td>55+56</td>
<td>56+57</td>
</tr>
<tr>
<td>57+58</td>
<td>58+59</td>
</tr>
<tr>
<td>59+60</td>
<td>60+61</td>
</tr>
<tr>
<td>61+62</td>
<td>62+63</td>
</tr>
<tr>
<td>63+64</td>
<td>64+65</td>
</tr>
<tr>
<td>65+66</td>
<td>66+67</td>
</tr>
<tr>
<td>67+68</td>
<td>68+69</td>
</tr>
</tbody>
</table>
Another Example:

Boundaries May Not Be Obvious

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>47</td>
</tr>
<tr>
<td>lower bound</td>
<td>0 48</td>
</tr>
<tr>
<td></td>
<td>1 49</td>
</tr>
<tr>
<td></td>
<td>2 50</td>
</tr>
<tr>
<td></td>
<td>3 51</td>
</tr>
<tr>
<td></td>
<td>4 52</td>
</tr>
<tr>
<td></td>
<td>5 53</td>
</tr>
<tr>
<td></td>
<td>6 54</td>
</tr>
<tr>
<td></td>
<td>7 55</td>
</tr>
<tr>
<td></td>
<td>8 56</td>
</tr>
<tr>
<td>upper bound</td>
<td>9 57</td>
</tr>
<tr>
<td></td>
<td>: 58</td>
</tr>
<tr>
<td></td>
<td>A 65</td>
</tr>
<tr>
<td></td>
<td>a 97</td>
</tr>
</tbody>
</table>

Refer to Testing Computer Software, pages 9-11
Another example of non-obvious boundaries

Still in the 99+99 program
Enter the first value
Wait N seconds
Enter the second value

Suppose our client application will time out on input delays greater than 600 seconds. Does this affect how you would test?

Suppose our client passes data that it receives to a server, the client has no timeout, and the server times out on delays greater than 300 seconds.

• Would you discover this timeout from a path analysis of your application?
• What boundary values should you test? In whose domains?
In sum: equivalence classes and representative values

Two tests belong to the same *equivalence class* if you expect the same result (pass / fail) of each. Testing multiple members of the same equivalence class is, by definition, redundant testing.

In an ordered set, *boundaries* mark the point or zone of transition from one equivalence class to another. The program is more likely to fail at a boundary, so these are the best members of (simple, numeric) equivalence classes to use.

More generally, you look to subdivide a space of possible tests into relatively few classes and to run a few cases of each. You’d like to pick the most powerful tests from each class. We call those most powerful tests the *best representatives* of the class.
A new boundary and equivalence table

<table>
<thead>
<tr>
<th>Variable</th>
<th>Risk (potential failure)</th>
<th>Classes that should not trigger the failure</th>
<th>Classes that might trigger the failure</th>
<th>Test cases (best representatives)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>First input</td>
<td>Fail on out-of-range values</td>
<td>-99 to 99</td>
<td>MinInt to -100, 100 to MaxInt</td>
<td>-100, 100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Doesn't correctly discriminate in-range from out-of-range</td>
<td></td>
<td></td>
<td>-100, -99, 100, 99</td>
<td></td>
</tr>
<tr>
<td>Misclassify digits</td>
<td>Non-digits</td>
<td>0 to 9</td>
<td>0 (ASCII 48)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9 (ASCII 57)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Misclassify non-digits</td>
<td>Digits 0 - 9</td>
<td>ASCII other than 48 - 57</td>
<td>/ (ASCII 47) ; (ASCII 58)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that we’ve dropped the issue of “valid” and “invalid.” This lets us generalize to partitioning strategies that don’t have the concept of “valid” -- for example, printer equivalence classes. (For discussion of device compatibility testing, see Kaner et al., Chapter 8.)
Examples of ordered sets

So many examples of domain analysis involve databases or simple data input fields that some testers don't generalize. Here's a sample of other variables that fit the traditional equivalence class / boundary analysis mold.

ranges of numbers
character codes
how many times something is done
  ▪ (e.g. shareware limit on number of uses of a product)
  ▪ (e.g. how many times you can do it before you run out of memory)
how many names in a mailing list, records in a database, variables in a spreadsheet, bookmarks, abbreviations

size of the sum of variables, or of some other computed value (think binary and think digits)
size of a number that you enter (number of digits) or size of a character string
size of a concatenated string
size of a path specification
size of a file name
size (in characters) of a document
Examples of ordered sets

- Size of a file (note special values such as exactly 64K, exactly 512 bytes, etc.)
- Size of the document on the page (compared to page margins) (across different page margins, page sizes)
- Size of a document on a page, in terms of the memory requirements for the page. This might just be in terms of resolution x page size, but it may be more complex if we have compression.
- Equivalent output events (such as printing documents)
- Amount of available memory (> 128 meg, > 640K, etc.)
- Visual resolution, size of screen, number of colors
- Operating system version
- Equivalent event times (when something happens)
- Timing: how long between event A and event B (and in which order—races)
- Length of time after a timeout (from JUST before to way after) -- what events are important?
Examples of ordered sets

- speed of data entry (time between keystrokes, menus, etc.)
- speed of input—handling of concurrent events
- number of devices connected / active
- system resources consumed / available (also, handles, stack space, etc.)
- date and time
- transitions between algorithms (optimizations)
  (different ways to compute a function)
- most recent event, first event
- input or output intensity (voltage)
- speed / extent of voltage transition (e.g. from very soft to very loud sound)
Non-ordered sets

A sample problem:

- There are about 2000 Windows-compatible printers, plus multiple drivers for each. We can’t test them all.

They are not ordered, but maybe we can form equivalence classes and choose best representatives anyway.

Here are two examples from programs (desktop publishing and an address book) developed in 1991-92.
Non-ordered sets

Primary groups of printers at that time:

- HP - Original
- HP - LJ II
- PostScript Level I
- PostScript Level II
- Epson 9-pin, etc.

LaserJet II compatible printers, huge class (maybe 300 printers, depending on how we define it)

1. Should the class include LJII, LJII+, and LIIP, LJIID-compatible subclasses?

2. What is the best representative of the class?
Non-ordered sets

Example: graphic complexity error handling
  • HP II original was the weak case.

Example: special forms
  • HP II original was strong in paper-handling. We worked with printers that were weaker in paper-handling.

We pick different best representatives from the same equivalence class, depending on which error we are trying to detect.

Examples of additional queries for almost-equivalent printers
  • Same margins, offsets on new printer as on HP II original?
  • Same printable area?
  • Same handling of hairlines? (Postscript printers differ.)
More examples of non-ordered sets

Here are more examples of variables that don't fit the traditional mold for equivalence classes but which have enough values that we will have to sample from them. What are the boundary cases here?

Membership in a common group
- Such as employees vs. non-employees.
- Such as workers who are full-time or part-time or contract.

Equivalent hardware
- such as compatible modems, video cards, routers

Equivalent output events
- perhaps any report will do to answer a simple the question: Will the program print reports?

Equivalent operating environments
- such as French & English versions of Windows 3.1
More examples of non-ordered sets

Rather than thinking about a single variable with a single range of values, a variable might have different ranges, such as the day of the month, in a date:

1-28
1-29
1-30
1-31

We analyze the range of dates by partitioning the month field for the date into different sets:

{February}
{April, June, September, November}
{Jan, March, May, July, August, October, December}

For testing, you want to pick one of each. There might or might not be a “boundary” on months. The boundaries on the days, are sometimes 1-28, sometimes 1-29, etc.

This is nicely analyzed by Jorgensen:
Software Testing--A Craftsman’s Approach.
An example of interaction

Interaction-thinking is important when we think of an output variable whose value is based on some input variables. Here’s an example that gives students headaches on tests:

I, J, and K are integers. The program calculates $K = I \times J$. For this question, consider only cases in which you enter integer values into I and J. Do an equivalence class analysis from the point of view of the effects of I and J (jointly) on the variable K. Identify the boundary tests that you would run (the values you would enter into I and J) if

- I, J, K are unsigned integers
- I, J, K are signed integers
Interaction example

K can run from MinInt (smallest integer) to MaxInt.
For any value of K, there is a set of values of I and J that will yield K

• The set is the null set for impossible values of K
• The set might include only two pairs, such as K = MaxInt, when MaxInt is prime (7 could be a MaxInt)
  » (I,J) ? {(1, MaxInt), (MaxInt, 1)}
• The set might include a huge number of pairs, such as when K is 0:
  » (I,J) ? {(0,0), (1, 0), (2, 0), … (MaxInt,0), (0,1),…,(0, MaxInt,1)}
• A set of pairs of values of I and J can be thought of as an equivalence set (they all yield the same value of K) and so we ask which values of K are interesting (partition number 1) and then which values of I and J would produce those K-values, and do a partitioning on the sets of (I,J) pairs to find best representatives of those.
• As practitioners, we do this type of analysis often, but many of us probably don’t think very formally about it.
Domain Testing

Strengths

• Find highest probability errors with a relatively small set of tests.
• Intuitively clear approach, easy to teach and understand
• Extends well to multi-variable situations

Blind spots or weaknesses

• Errors that are not at boundaries or in obvious special cases.
  » The "competent programmer hypothesis" can be misleading.
• Also, the actual domains are often unknowable.
• Reliance on best representatives for regression testing leads us to overtest these cases and undertest other values that were as, or almost as, good.
Part 4.

Introduction to
Black Box Test Case Design
Test Case Design

We’ll soon see a wide variety of strategies for designing test cases. Each one offers its own definition of excellence in test case design.

This section summarizes some traditional wisdom about good test case design.
Test Case Design

If the purpose of testing is to gain information about the product, then a test case’s function is to elicit information quickly and efficiently.

In information theory, we define “information” in terms of reduction of uncertainty. If there is little uncertainty, there is little information to be gained. A test case that promises no information is poorly designed. A good test case will provide information of value whether the program passes the test or fails it.
Models

A Model is…

• A map of a territory
• A simplified perspective
• A relationship of ideas
• An incomplete representation of reality
• A diagram, list, outline, matrix…

No good test design has ever been done without models.

The trick is to become aware of how you model the product, and learn different ways of modeling.
Simple Black Box 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 Black Box Testing Model

- Test Inputs
  - Precondition Data
  - Precondition Program State
  - Environmental Inputs

- System Under Test

- Test Results
  - Post-condition Data
  - Post-condition Program State
  - Environmental Results
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
The “Complete” Oracle

Test Oracle

System Under Test

Test Inputs

Precondition Data

Precondition Program State

Environmental Inputs

Test Results

Postcondition Data

Postcondition Program State

Environmental Results

Reprinted with permission of Doug Hoffman

Copyright © 1994-2004 Software Quality Methods, LLC. All Rights Reserved.
An Example Model For SUT

System Under Test

User → GUI → Functional Engine → Data Set → Remote GUI ← User

API
Breaking Down The Testing Problem

System Under Test

User → GUI → Data Set → Remote GUI → User

User → API → Functional Engine → User
Good Test Case Design

An excellent test case satisfies the following criteria:

- Reasonable probability of catching an error
- Exercises an area of interest
- Does interesting things
- Doesn’t do unnecessary things
- Neither too simple nor too complex
- Not redundant with other tests
- 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
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
Make Test Output Obvious:
Examples from Printer Testing

Frame graphical output
Put a one-pixel wide frame around the graphic and/or around the edge of the printable page. Off-by-one-pixel errors will often erase one of these borders.

For color output, create color sample charts.
Print a color wheel or some other standard progression (light to dark or rainbow progression) that is visually obvious. Name the colors. Have a comparison page handy.

Draw regular line-art that can show distortions
Draw circles and perfect squares. Look for bowing, stretching, dropout. Use some dashed lines, to see if dashes are drawn unequally long.
Making a Good Test

- Start with a known state
- Design variation into the tests
  » configuration variables
  » specifiable (e.g. table-loadable) data values
- Check for errors
- Put your analysis into the test itself
- Capture information when the error is found (not later)
  » test results
  » environment results
- Don’t encourage error cascades
Black Box Software Testing

Part 5

An Overview of Product Development

Supplementary Reading:

- Coyote Valley Software’s *Product Lifecycle Model*: http://www.coyotevalley.com/plc/builder.htm

- Tom Gilb’s *Evolutionary Results Delivery Method*. http://stsc.hill.af.mil/swtesting/gilb.asp
During development, several groups (or people) act as service providers to the Project Manager. They often play other roles beyond this, but during development, it pays to look at them as members of a managed group that seeks to ship a high quality, salable product on time.
Product Development Organization

**Project Manager:** Responsible for shipping a salable product on time and within budget.

**Architect:** Design the product’s overall (code & data) structure, relationships among components, and relationships with other expected parts of the customers’ system.

**User Interface Designers:** Design to make the program usable and useful.

**Programmers:** Design and write/test the code.

**Glass Box Testers:** Use knowledge of the internals of the code to drive the testing. Along with glass box test techniques, use code inspections and other code-aware methods to find errors.
# Product Development Organization

**Tech Writers: Write the user manual, help, etc.**

**Content Developers:** There may be several different groups here, animating, filming, writing and performing music, writing, etc.

**Multimedia Producer:** Responsible for managing content development, selecting content to include with the product, negotiating rights, etc.

**Marketing:** Manage the product’s overall profit/loss plan. Evaluate competitors, appraise sub-markets (e.g. should we be compatible with a certain printer?), design packaging and advertisements, run trade shows, etc.

**Black Box Testers:** Test from the customer’s perspective.
Product Development Organization

**Customer Service:** Help evaluate the effect of design decisions, bugs, and existing problems on customers.

**Manufacturing:** Buy materials. Help everyone control their contribution to the cost of goods. Drive the late-stage schedule. Make the product.
The Waterfall Model for Software Development

Idea —
Requirements —
Design —
Detailed Design —
Coding —
Test Planning —
Write manual —
Testing —
Customer Beta —
Release product —
What the Waterfall Really Looks Like

Idea

Announce product

Choose team

Order T-shirts

Code

Design

Accept orders

Test

Write manual

Write test plan

Write specs

Release product

Customer complaints
Evolution

Prototyping

- Requirements
- Complete Engineering
- Rapid Design
- Prototype
- Revise Requirements
- User Feedback
- Start
- Stop
Mock-Up Approach

- Revise Requirements
- Revise Prototype
- Evaluate Prototype
- Detailed Design
- Code/Unit Test
- Integration/Product Verification
- Implementation/System Test
- Operation and Maintenance/Revalidation
Spiral Model

Determine objectives, alternatives, constraints

Evaluate Alternatives, Identify and Resolve

Plan Next Phases

Develop, Verify Next Level Product

Risk Analysis

Prototype

Operational Prototype

Detailed Design

Implementation

Integration and Test

Development Plan

Requirements Plan

Concept

Prototype

Risks

Risk Analysis

Risk Analysis

Risk Analysis

Requirements
Fourth Generation

Requirements Gathering

“Design” Strategy

Implementation Using 4GL

Testing
Object-Oriented

1. OOA
2. No Go
3. Customer Evaluation
4. OK
5. Put in Use

No

OOD

Reuse Component

Component Library
Examine for a Reusable Component

Put in Library

Develop New Component

Test
Code
Design
Spec

Develop New Component

Copyright © 1994-2004 Software Quality Methods, LLC. All Rights Reserved.
Evaluating a Life Cycle

When you evaluate a life cycle model, consider the tradeoffs that the project manager must make among

- Features
- Reliability
- Price
- and Time.

Ask yourself what project management system will more effectively manage the risks that your recommended changes are not made.
Product Development Milestones

Milestones are moments of truth for a project. They define synchronization points that help groups work as a team. They do this by collecting lists of prerequisite tasks. (“Complete all these tasks in order to reach the milestone.”) Tasks that trigger events (e.g. UI freeze) become milestones.

- Design and specification complete
- First functionality
- Almost alpha
- Alpha
- Pre-beta
- Beta
- User interface freeze
- Pre-final
- Final
- Release
Product Development Milestones

- **Product design and specification complete**
  We know what is to be built, what it will look like, what benefits it will provide, and how it will work.

- **First functionality**
  The program has just enough capability that you can try it out.

- **Almost alpha**
  The alpha milestone will hit in a few days or weeks. Before the program can be declared “alpha” the Testing Group has to check that it meets the alpha criteria.

- **Alpha**
  A significant (and agreed-upon) portion of the product has been completed. (The “product” includes code, documentation, additional art or other content, etc.) The product is ready for first in-house use.
Product Development Milestones

- **Pre-beta**
  A *beta candidate* has been submitted. If it meets the beta criteria (as verified by Testing), the software is beta.

- **User interface freeze**
  Every aspect of the user interface of the software has been frozen. Some companies except error messages and help screens.

- **Beta**
  Most or all of the product is complete. Some companies send “beta copies” of the product to customers at this milestone.

- **Pre-final**
  A *final candidate* has been submitted. If all the pieces are there and other tests are met, Final.

- **Final**
  Last wave of testing before it gets inflicted on customers.

- **Release**
Variation in Milestone Definitions

Alpha definitions:
   a. All code complete (but not necessarily working)
   b. Customer useful (but not necessarily fully coded)
   c. Core functionality complete (but not complete and not useful)

Beta definitions:
   a. Reasonably predictable as X weeks (e.g. 6) before shipment.
      − All known Level 1 errors corrected.
      − No bugs are known that look so complex that they might not be fixed before shipment
      − X% (such as 90%) of all Level 2 bugs reported have been closed and the total number of remaining bugs is less than K
   b. Customer useful. All critical features coded. No known bugs that could damage an external beta user’s data or embarrass the publisher
   c. Customer useful. Some features are still absent.

The point is not to decide which definition is correct.
The point is to understand which definition is in use.
Variation in Milestones

Milestones are defined quite differently by different development groups.

- Another example of detailed definitions comes from Brian Lawrence and Bob Johnson of Coyote Valley Software.
- The testing group doesn’t define the life cycle model or the milestones. But you can do the project a huge service by listing a wide range of tasks and working with the project manager to sort the tasks into milestones (must be completed by Milestone X.)
A Product Life Cycle (PLC) Model

Introduction

The term "PLC" has stood for both a Product life cycle and a Project life cycle. We believe that a project is many times a subset of a product. Thus, products consist of one or more projects.

Many projects fail because they do not consider the bigger product view: the smallest of projects may need to "sell" their results, announce their completion, and in general do many of the tasks required of larger products being delivered to customers. Therefore, we suggest you keep an open mind about your need to do the activities suggested by these checklists.

We are not blind to the fact that so many PLCs have been slayers of countless trees - behemoth tomes which end up gathering dust on engineers and managers' bookshelves. We hope your effort is better received. Our approach was to follow a simple design objective:

It should be easier to complete my project using this PLC than without it!

With this idea in mind we constructed a set of tools we think is useful, but it's helpful to understand our philosophy when learning how to use them. There is a lot of material presented here. We do not believe every organization needs to implement this entire PLC. Think of this as a shopping list. Look at the sections which you believe will help your organization. Then cross-reference their contents. Provided you are always adding, removing some lists.

Views

There are many ways to build a PLC; here we present three approaches to the problem:

- A Minimal set. This will get you started with checklists for a simple Alpha, Beta, First Customer Ship (FCS) system.
- A Complete set. This is a complete PLC, with checklists for major and minor milestones.
- The Smorgasbord (A work in progress). This is all the Milestones in a beginning to end structure. Included are checklists plus templates of various deliverables.

Getting Help

If you want to save yourself and your organization time and trouble, Coyote Valley Software can provide you with one- or two-day seminars on this PLC model. Please send mail for rates and availability.

Credits

What's presented here is based on the hard work by lots of people over several years at least four different companies, spanning continents, and uncounted bottles of fine wine.

About the Authors

- Brian Lawrence
- Rob Johnson and more Rick
Part 6

The Black Box Testing Organization

Supplementary Reading:

Role of the Testing Group

Many testing groups call themselves “Quality Assurance.” This gives everyone the wrong idea.

A QA group has the resources and authority to set standards and solve problems. A QA group owns the project’s quality and thus the satisfaction of the company’s customers. The role includes prevention as well as late-stage damage control. For example, the real head of QA can send programmers to courses in defensive programming. Normally, the real head of QA in a company is the President or a Senior Vice-President.

A Testing Group provides important technical services:

- We find and report bugs. *This is high skill work, not to be under-rated.*
- We identify weak areas of the program;
- We identify high risk areas of a project;
- We explain our findings in ways that help customer service staff to help customers;
- We explain our findings in ways that help management make reasoned business decisions about each bug.
Software QA vs. Testing

CSQE Body of Knowledge (ASQ)

- General Knowledge (10%)
- Software Quality Management (19%)
- Software Engineering Processes (16%)
- Program and Project Management (15%)
- Software Metrics (15%)
- **Software Verification and Validation (15%)**
- Configuration Management (10%)
Role of the Testing Group

Within a testing group, the typical responsibilities of a tester are:

- **Find problems**
  - find bugs
  - find design issues
  - find more efficient ways to find bugs
- **Communicate problems**
  - report the bugs and design issues
  - report on testing progress
  - evaluate and report the program’s stability
- **Manage / supervise testing projects**
  - Prepare test plans and schedules
  - Estimate testing tasks, resources, time and budget
  - Measure and report testing progress against milestones
  - Teach other testers to find bugs
It is Not Our Job to Verify That a Program Works Correctly

- If you can’t test the program completely, you can’t verify that it works correctly.
- Besides, the program doesn’t work correctly, so no one can verify that it does.
- If your goal is to verify that the program works correctly, then as a tester, you fail every time you find an error.

  If you set your mind to showing that the program works correctly, you’ll be more likely to miss problems than if you want and expect the program to fail.

- This is a critical psychological problem that affects researchers in all areas of science and that shows up even in basic human sensation and perception.
- Read Rosenthal’s *Experimenter Effects in Behavioral Research*, for example.
# Our Mission: Which Group is Better?

<table>
<thead>
<tr>
<th></th>
<th>Found pre-release</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Testing Group 1</strong></td>
<td></td>
</tr>
<tr>
<td>Function A</td>
<td>100</td>
</tr>
<tr>
<td>Function B</td>
<td>0</td>
</tr>
<tr>
<td>Function C</td>
<td>0</td>
</tr>
<tr>
<td>Function D</td>
<td>0</td>
</tr>
<tr>
<td>Function E</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Found pre-release</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Testing Group 2</strong></td>
<td></td>
</tr>
<tr>
<td>Function A</td>
<td>50</td>
</tr>
<tr>
<td>Function B</td>
<td>6</td>
</tr>
<tr>
<td>Function C</td>
<td>6</td>
</tr>
<tr>
<td>Function D</td>
<td>6</td>
</tr>
<tr>
<td>Function E</td>
<td>6</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>74</td>
</tr>
</tbody>
</table>

Two groups test the same program.
- The functions are equally important
- The bugs are equally significant

*This is artificial, but it sets up a simple context for a discussion of tradeoffs.*

From Marick, *Classic Testing Mistakes*
## Our Mission:
### Which Group is Better?

<table>
<thead>
<tr>
<th></th>
<th>Found pre-release</th>
<th>Found later</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function A</td>
<td>100</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Function B</td>
<td>0</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Function C</td>
<td>0</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Function D</td>
<td>0</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Function E</td>
<td>0</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100</strong></td>
<td><strong>48</strong></td>
<td><strong>148</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Found pre-release</th>
<th>Found later</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function A</td>
<td>50</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>Function B</td>
<td>6</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Function C</td>
<td>6</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Function D</td>
<td>6</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Function E</td>
<td>6</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>74</strong></td>
<td><strong>74</strong></td>
<td><strong>148</strong></td>
</tr>
</tbody>
</table>
So? Purpose of Testing?

A testing group has two key priorities, bug finding and assessment. Sometimes, these conflict:

1. Find the bugs (preferably in priority order).
2. Help the project manager make tough decisions.

Assessment is the underlying reason for testing, from management’s viewpoint. But if you aren’t hammering hard on the program, you aren’t doing anyone any favors.
Varying Missions of Test Groups

(Some of these might be impossible)

- Find defects
- Maximize bug count
- Block premature product releases
- Help managers make ship / no-ship decisions
- Assess quality
- Minimize technical support costs
- Conform to regulations
- Minimize safety-related lawsuit risk
- Assess conformance to specification
- Find safe scenarios for use of the product (find ways to get it to work, in spite of the bugs)
- Verify correctness of the product
- Assure quality
A Different Take:
Public vs. Private Bugs

A programmer’s **public bug rate** includes all bugs left in the code when she gives it to someone else (such as a tester.) Rates of one bug per hundred statements are not unusual, and several programmers’ rates are higher (such as three bugs per hundred).

A programmer’s **private bug rate** includes all the bugs that she makes, including the ones she fixes before passing the program to testing.

Estimates of private bug rates have ranged from 15 to 150 bugs per 100 statements. Therefore, programmers must be finding and fixing between 80% and 99.3% of their own bugs before their code goes into test. (Even the sloppy ones find and fix a lot of their own bugs.)

What does this tell us about our task?

It says that we’re looking into the programmer’s (and her tools’) blind spots. Merely repeating the types of tests that the programmers did won’t yield more bugs. That’s one of the reasons that an alternative approach is so valuable.
On Hitting The Wall

What do you do when you’ve been testing the product for a while and you finally hit the wall / lose your edge?

A few suggestions:

• test against the manual for fresh perspective
• rotate program areas
• bring in users for observation
• Pull out top bugs, see where the weak areas of the program are and re-attack
• be systematic (if uncreative)
• create complex documents (use competitors’ demos)
• switch focus to automation or some other useful, technical task.
Part 7

Test Planning:

Reusable Test Matrices
Using Test Matrices for Routine Issues

After testing a simple numeric input field a few times, you’ve learned the drill. The boundary chart is reasonably easy to fill out for this, but it wastes your time. Use a test matrix to show/track a series of test cases that are essentially the same.

• For example, for most input fields, you’ll do a series of the same tests, checking how the field handles boundaries, unexpected characters, function keys, etc.

• As another example, for most files, you’ll run essentially the same tests on file handling.

The matrix is a concise way of showing the repeating tests.

• Put the objects that you’re testing on the rows.

• Show the tests on the columns.

• Check off the tests that you actually completed in the cells.
Typical Uses of Test Matrices

- You can often re-use a matrix like this across products and projects.
- You can create matrices like this for a wide range of problems. Whenever you can specify multiple tests to be done on one class of object, and you expect to test several such objects, you can put the multiple tests on the matrix.
- Mark a cell green if you ran the test and the program passed it.
- Mark the cell red if the program failed and write the bug number of the bug report for this bug.
- Write the automation number (or identifier or file name) in the cell if the test case has been automated.
Potential Problems with Matrices

- What if your thinking gets out of date? (What if this program poses new issues, not covered by the standard tests?)
- Do you need to execute every test every time? (or ever?)
- What if the automation ID number changes? -- We still have a maintenance problem but it is not as obscure.
<table>
<thead>
<tr>
<th>Numeric (Integer) Input Field</th>
<th>Nothing</th>
<th>LB of value</th>
<th>UB of value</th>
<th>LB of value - 1</th>
<th>UB of value + 1</th>
<th>0</th>
<th>Negative</th>
<th>LB number of digits or chars</th>
<th>UB number of digits or chars</th>
<th>Empty field (clear the default value)</th>
<th>Outside of UB number of digits or chars</th>
<th>Non-digits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This includes only the first few columns of a matrix that I’ve used commercially, but it gets across the idea.
### Examples of integer-input tests

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Test Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing</td>
<td></td>
</tr>
<tr>
<td>Valid value</td>
<td></td>
</tr>
<tr>
<td>At LB of value</td>
<td></td>
</tr>
<tr>
<td>At UB of value</td>
<td></td>
</tr>
<tr>
<td>At LB of value - 1</td>
<td></td>
</tr>
<tr>
<td>At UB of value + 1</td>
<td></td>
</tr>
<tr>
<td>Outside of LB of value</td>
<td></td>
</tr>
<tr>
<td>Outside of UB of value</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Negative number</td>
<td></td>
</tr>
<tr>
<td>At LB number of digits or chars</td>
<td></td>
</tr>
<tr>
<td>At UB number of digits or chars</td>
<td></td>
</tr>
<tr>
<td>Empty field (clear the default value)</td>
<td></td>
</tr>
<tr>
<td>Outside of UB number of digits or chars</td>
<td></td>
</tr>
<tr>
<td>Non-digits</td>
<td></td>
</tr>
<tr>
<td>Wrong data type (e.g. decimal into integer)</td>
<td></td>
</tr>
<tr>
<td>Expressions</td>
<td></td>
</tr>
<tr>
<td>Space</td>
<td></td>
</tr>
<tr>
<td>Non-printing char (e.g., Ctrl+char)</td>
<td></td>
</tr>
<tr>
<td>DOS filename reserved chars (e.g., &quot;\ * .:&quot; )</td>
<td></td>
</tr>
<tr>
<td>Upper ASCII (128-254)</td>
<td></td>
</tr>
<tr>
<td>Upper case chars</td>
<td></td>
</tr>
<tr>
<td>Lower case chars</td>
<td></td>
</tr>
<tr>
<td>Modifiers (e.g., Ctrl, Alt, Shift-Ctrl, etc.)</td>
<td></td>
</tr>
<tr>
<td>Function key (F2, F3, F4, etc.)</td>
<td></td>
</tr>
</tbody>
</table>
Error Handling when Writing a File

full local disk
almost full local disk
write protected local disk
donamed (I/O error) local disk
unformatted local disk
remove local disk from drive after opening file
timeout waiting for local disk to come back online
keyboard and mouse I/O during save to local disk
other interrupt during save to local drive
power out during save to local drive

full network disk
almost full network disk
write protected network disk
donamed (I/O error) network disk
remove network disk after opening file
timeout waiting for network disk
timeout waiting for network disk to come back online
keyboard / mouse I/O during save to network disk
other interrupt during save to network drive
local power out during save to network drive
network power out during save to network drive
Part 8

Test Planning:

Relationship Tables
Tabular Format for Data Relationships
Tabular Format for Data Relationships

- Look at this record, from the Timeslips Deluxe time and billing database. In this dialog box, click the arrow next to the Consultant field to edit the Consultant record (my name, billing info, etc.) or enter a new one.
- If I edit it here, will the changes carry over to every other display of this Consultant record?
- Also, note that the End Date for this task is before the Start Date. That’s not possible.
The program checks the End Date against the Start Date and rejects this pair as impossible because the task can’t end before it starts.

*The value of End Date is constrained by Start Date, because End Date can’t be earlier than Start Date.*

*The value of Start Date constrains End Date, because End Date can’t be earlier than Start Date.*
A Tabular Format for Data Relationships

<table>
<thead>
<tr>
<th>Field</th>
<th>Entry Source</th>
<th>Display</th>
<th>Print</th>
<th>Related Variable</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Date</td>
<td></td>
<td></td>
<td></td>
<td>End Date</td>
<td>Constraint to a range</td>
</tr>
<tr>
<td>End Date</td>
<td></td>
<td></td>
<td></td>
<td>Start Date</td>
<td>Constraint to a range</td>
</tr>
</tbody>
</table>

Copyright © 1994-2004 Software Quality Methods, LLC.  All Rights Reserved. 171
Tabular Format for Data Relationships

Once you identify two variables that are related, test them together using boundary values of each or pairs of values that will trigger some other boundary.

- This is not the most powerful process for looking at relationships. An approach like Cause-Effect Graphing is more powerful, if you have or can build a complete specification.
- I started using this chart as an exploratory tool for simplifying my look at relationships in overwhelmingly complex programs. (There doesn’t have to be a lot of complexity to be “overwhelming.”)
**Tabular Format for Data Relationships**

**THE TABLE’S FIELDS**

- **Field**: Create a row for each field (Consultant, End Date, and Start Date are examples of fields.)

- **Entry Source**: What dialog boxes can you use to enter data into this field? Can you import data into this field? Can data be calculated into this field? List every way to fill the field -- every screen, etc.

- **Display**: List every dialog box, error message window, etc., that can display the value of this field. When you re-enter a value into this field, will the new entry show up in each screen that displays the field? (Not always -- sometimes the program makes local copies of variables and fails to update them.)

- **Print**: List all the reports that print the value of this field (and any other functions that print the value).

- **Related to**: List every variable that is related to this variable. (What if you enter a legal value into this variable, then change the value of a constraining variable to something that is incompatible with this variable’s value?)

- **Relationship**: Identify the relationship to the related variable.
Many relationships among data:

- **Independence**
  - Varying one has no effect on the value or permissible values of the other.

- **Causal determination**
  - By changing the value of one, we determine the value of the other.
  - For example, in MS Word, the extent of shading of an area depends on the object selected. The shading differs depending on Table vs. Paragraph.

- **Constrained to a range**
  - For example, the width of a line must be less than the width of the page.
  - In a date field, the permissible dates are determined by the month (and the year, if February).

- **Selection of rules**
  - For example, hyphenation rules depend on the language you choose.
Tabular Format for Data Relationships

Many relationships among data:

- Logical selection from a list
  - processes the value you entered and then figures out what value to use for the next variable. Example: timeouts in phone dialing:
    - 0 on complete call 555-1212 but 95551212?
    - 10 on ambiguous completion, 955-5121
    - 30 seconds incomplete 555-121
- Logical selection of a list:
  - For example, in printer setup, choose:
    - OfficeJet, get Graphics Quality, Paper Type, and Color Options

Look at Marick (Craft of Software Testing) for discussion of catalogs of tests for data relationships.
Complex Data Relationships
Data Relationship Table

Looking at the Word options, you see the real value of the data relationships table. Many of these options have a lot of repercussions. You might analyze all of the details of all of the relationships later, but for now, it is challenging just to find out what all the relationships ARE. The table guides exploration and will surface a lot of bugs.

PROBLEM
Works great for this release. Next release, what is your support for more exploration?