BIO
PRESENTATION

W2

10/18/2006 11:30:00 AM

BACK TO THE BEGINNING - TESTING AXIOMS REVISITED

Erik Petersen
Emprove

International Conference on
Software Testing Analysis and Review
October 16-20, 2006
Anaheim, CA USA

Erik Petersen

Erik Petersen is based in Melbourne, Australia and has moved through many SDLC roles since the mid 1980s, focusing on software testing and QA since the early 90s, working on projects for small software houses, custom software providers, large companies and government organizations. For most of his career, he has been involved in custom software development. He has consulted on, worked on and managed dozens of projects in areas as diverse as Hong Kong's international airport terminal scheduling to Japanese sales management systems to a retail and distribution system involving 3 companies that went to nearly every retail outlet across Australia and collected tens of millions of dollars daily. Erik is currently mixes test consultling roles with training testers across Australia and New Zealand.

Erik has worked with more than a dozen methodologies from ISO certified through to agile methods. He is a thought leader in agile and exploratory testing, and was one of the first people to formally propose the idea of paired exploratory testing in 2001.

Erik has been an active participant in several software testing discussion groups since 1996, having email contact with many European and North American testers. Erik was a reviewer of the Software Engineering Body of Knowledge (SWEBOK) 2001 stone draft and 2003 iron man draft, and has reviewed many books and articles including Brian Marick's classic "Classic Testing Mistakes", the highly acclaimed "Lessons Learned in Software Testing" by Kaner, Bach and Pettichord, and Rex Black's "Critical Testing Processes". Recently Erik reviewed the draft IEEE 829 Test Documentation Standard, and many of his suggested changes were adopted, including his accidental naming of the Master Test Report, and the use of bug clusters.

Since 2001, he has been presenting at conferences and user groups, where he is becoming recognized for his mix of practical knowledge and project experience (and excessive numbers of slides!). As well as speaking at Australian, New Zealand, Asian and American conferences, he has also been privileged to win presentation awards for his ideas on new approaches to software testing and quality.

Visit the Software Testing Spot at www.testingspot.net for a useful mix of links to improve your testing and other SDLC skills.

Contact Erik by email via emprove@gmail.com

Back to the Beginning: Testing axioms revisited

Erik Petersen emprove@gmail.com

www.testingspot.net

In the beginning

- In 1976, Glenford Myers' "**Software Reliability – Principles and Practices**" included a section on software testing, and a series of testing axioms (i.e established or accepted truths)
- One of the first books on testing, it can still be bought today. Traditionalist uni courses teach most (9 of 16) axioms as "a useful guide to good testing practice."
- Now 30 years later, let's revisit the original axioms (plus some newer ones) and look at their recognition and relevance today
- <Aside: the published order of the axioms has been rearranged for this presentation, and not all axioms are included. All axioms are covered in the paper.>

Mid 1970s computing

- Myers worked for IBM, which dominated computing, and mainframes dominated IBM. Many of his ideas echo IBM culture of the time.
- Testing was part of development and not a standalone discipline
- Green screens, text based and batch oriented, operator centric, rerunning jobs, delivering printouts etc
- Large mainframe computers had 128K of memory but were very expensive (64K was the norm). One 30 gigabyte music player is more than 200 million times larger!

Programs then and now

- Size increased exponentially
  - Vi UNIX text editor 73 kb *vs* MS-Word >10,000 kb
- Control flow more complex
  - Character based control flow mostly sequential, all keyboard based
  - Unpredictable GUI based control (mouse, keyboard, tablet), even worse for web apps, and voice recognition soon
- Ability to design custom interface makes usability much more important
- Unpredictable delivery mechanisms, unpredictable load and dynamic infrastructure are a challenge to system stability and reliability
- Also more common to buy programs now rather than build, but still need to configure and define parameters

Reuse creative programmers

- *Assign your most creative programmers to testing*
- Myers disputed the common view of testing as mundane and boring work, saying experience shows opposite is more accurate
- Some uni courses paraphrase “most creative” to “best”

Today's view

- Narrow programmer vs. broad tester mindset recognized, so functional testing now a distinct role
- While writing test cases requires creativity, running scripted test cases can be very boring and mundane! Exploratory testing is very creative
- New testsmith roles are mix of dev and test, creating new test tools typically with scripting languages like Ruby
- This axiom is dead for system testing (apart from IT shops without system testers!) but is coming back into vogue in agile projects, particularly for unit testing (more on this later …)

Variations on the theme

- *Testing work is creative and difficult*
  – Bill Hetzel 1988
- *Software testing is a disciplined technical profession requiring training*
  - Ron Patton 2000

Show rigour with results

- *Thoroughly inspect the results of each test*
- Tests are useless if results only glanced at
- Testing means more than just executing set number of cases
- Programmers may react to new error with "I tested that but somehow didn't see that in the output" (does this happen today? ☺)

Today’s view

- This doesn’t seem to be a problem with system testers, but often time to write new cases is limited or non-existent
- It may be a problem with immature developers, but bigger issue is developer coverage and moving beyond valid tests to invalid ones.
- TDD (see later) not only inspects results automatically but fails the build if any are found

Once and once only

- *The design of a system should be such that each module is integrated into the system only once*
- Myers explains this is basically about design simplicity, utilizing small modules that perform single functions

Today’s view

- This doesn’t really fit as a testing axiom, but anticipated incremental development styles
- If this was paraphrased into an axiom of software design, it would still be current. Uni course paraphrases include “Ensure that system design allows for straight forward testing” & “Get the design right and you will have less code to test”

Un-self-ish testing

- *It is impossible to test your own program*

Today's view

- Myers felt programmers shouldn't do any testing, even unit testing. Was this ever followed for unit testing? Programmer typically seen as best person to do unit testing.
- Programmers traditionally haven't liked system testing, but may have to on small projects. New agile approaches are involving programmers in more formal unit testing and they are looking to testers for guidance

TDD

- The previous “integrating modules” axiom related to design simplicity in terms of modules focussing on single functions
- Test driven development (TDD) forces design simplicity during coding by having coders repeatedly add an automated true/false unit test then write matching code, and re-executing the tests with each build
- System compiles very quickly (< 5 minutes) so common to have several daily builds of all progs. Build fails if any test fails (red status vs. green pass status).

Variations on the theme

- *Testing requires independence.*
  - Bill Hetzel
- This could be independent of
  - knowledge of the mechanics of the software
  - a reporting line through time-based (development or project) managers

Clumpy bugs

- *As the number of detected errors in a piece of software increases, the probability of the existence of more undetected errors also increases*
- Counter-intuitive phenomenon: bugs cluster!
- If a particular part of the system appears highly error prone during testing, extra testing efforts should be focussed there
- Uni course paraphrase: “Bugs are not hermits, it is likely to have friends nearby”

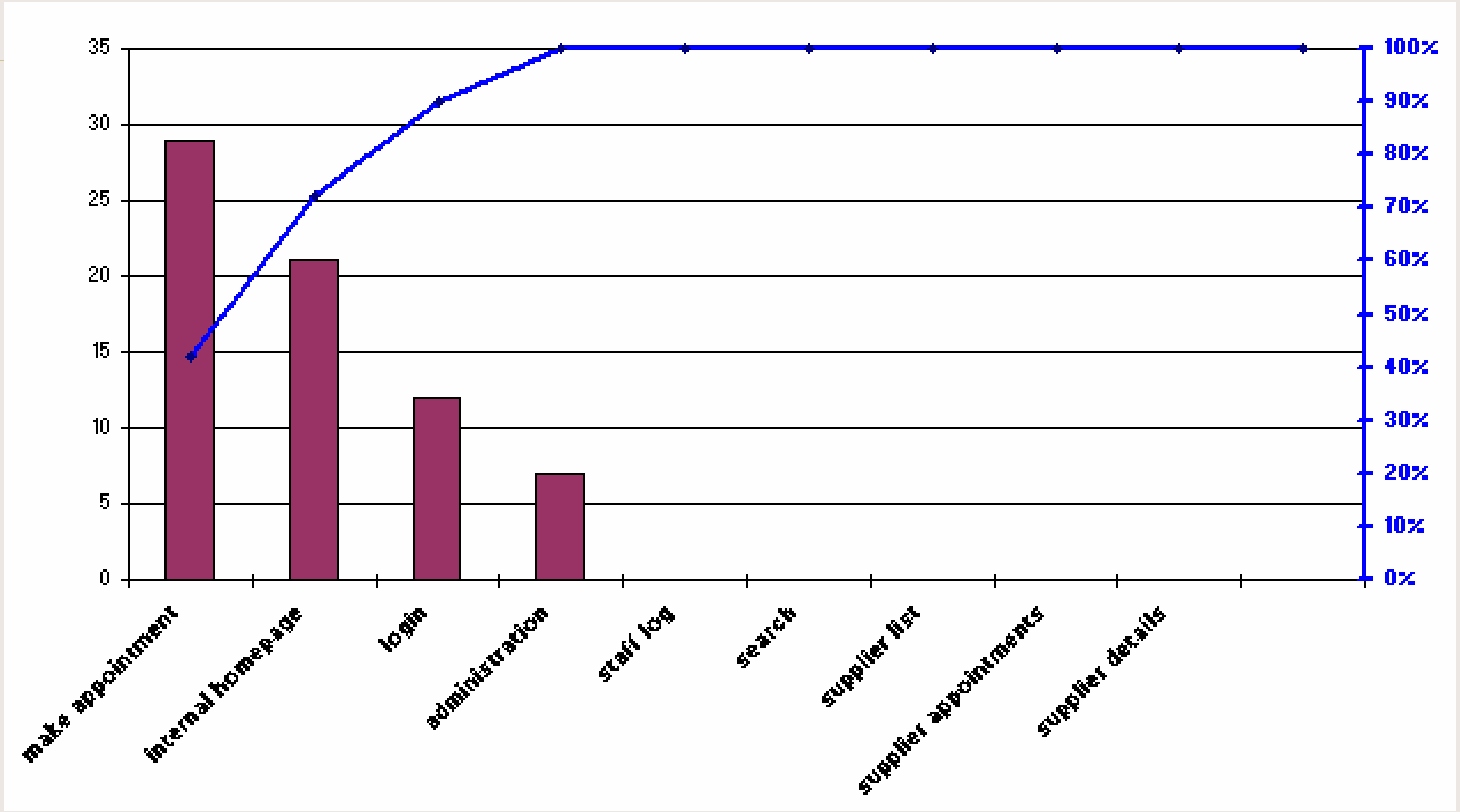
Today’s view

- This is new to many testers today!
- Common approach today ignores this axiom, basing testing instead on engineering methods (preplan detailed scripted tests for everything). Why not use scientific method? (use feedback to determine areas of focus)
- Now getting new recognition, and explicit mention in final draft of new 829 test doco std.
- Myer’s approach of pausing project to write scripted cases once clusters found rarely occurs in industry

Pareto rules

- Special 80:20 rule flavour for this axiom:
- About 80% of the defects come from 20% of the modules (**standard Pareto**) and about half the modules are defect free (**unique to software!**)
  - (*range is 60-90%, with 80% median*)
  - (*load testing: about 40% of modules have 60% defects*)
- About 90% of the downtime comes from at most 10% of the defects

Typical bug Pareto chart



All bugs are concentrated in about ½ the functions!

Variations on the theme

- *The more bugs you find in an area, the more bugs you expect*
  - Ron Patton
- *A small number of modules contain the majority of bugs found by testing, and also in the field*
  – ISTQB Principle

Encourage testing deja vu

- *Avoid non-reproducible or on-the-fly testing*
- Testing without scripted cases needs tests to be redeveloped for retesting
- Testing is already expensive enough without this
- Only do this for throw-away programs
- Document the tests to allow them to be reused by anyone

Today's view

- Writing tests to be reused by anyone is very difficult with the size of modern systems, requiring much effort to achieve detailed tests. Higher level tests more practical
- Pareto bug rule indicates some tests may never find bugs, so focus on broad coverage of tests. If you are scripting, don't have too many detailed tests until an area is confirmed to have bugs

Variations on the theme

- *Testing must be planned*
  - Bill Hetzel

ET

- Do tests need to be described to step level first before running? No. Can link design, direction and discovery dynamically to create tests.
- Exploratory Testing (ET) techniques follow scientific approach and utilize feedback – result of the last test may determine the next test
- Informal opinion indicates big bugs found with ET not scripted tests
- Create list of test ideas, risk rank and choose 1st
- Keep notes/records during ET for retests; track coverage and completion rate to predict end time

More ET

- May need oracles, e.g to validate mathematical or statistical results, or verify under the hood behaviour, e.g rec count retrieved for query
- ET usage varies from investigating bug fixes thru to ET sessions. Try to spend some test execution doing ET if GUI based, non-regulated system
- Session based techniques to co-ordinate groups including James & Jon Bach and James Lyndsay. Also attack techniques of James Whittaker, or quicktests of James Bach & Michael Bolton.
- Variations include paired exploratory testing

Positively Negative

- *A good test case has a high probability of detecting an undiscovered error, not a test case that shows that the program works correctly*

Positively negative overkill

- “Unfortunate consequences of taking Myers literally, … the (nameless) IBM lab set up a test team with the very aggressive mission of proving that the programmers committed errors. … relationships between programmers and testers reached the heights of non-co-operativeness, & the quality of the programmers work plummeted” - *Don Mills*

Negatively positive overkill

- “I was … speechless when a fellow professional said …she was going to discard the majority of her regression tests because they had failed to find errors. … I asked why she was considering this to which she confidently replied, well so-and-so says that tests that don’t find problems aren’t worthwhile” - *Linda Hayes*

Pesticide Paradox

- Another so-and-so (Boris Beizer) has discovered a reason to regularly revise scripted tests.
- Bugs get resistant to the same tests, like pesticide. Repeating the same tests (after fixes made) reduces their ability to find bugs. While regression tests are usually designed to prove correct functionality is unchanged, altering them will increase their chances of finding bugs.
- This is an ISTQB principle.

Today’s view

- Glass half full or half empty? Risk (**find bugs**) vs confidence (**check it works**). One uni thesis claimed goal of testing is “find bugs”, and aim of testing is “check it works” (?!)
- Test case “goodness” is based on context
  - Information objectives (e.g find bugs, test to spec)
  - Test attributes (e.g atypical, easier to evaluate)
  - Testing style (e.g stress, user, domain, scenario)
    - (see Cem Kaner’s “What IS a good test case?” for more)

What is a test case?

- A test case may be
  - A test idea
    - Formalized by varying degrees into a elaborated test case
    - Expanded to varying degrees into an exploratory session
  - An elaborated test case, defined step by step
- Test execution may be by test case, or exploratory session (based on a test idea, or spontaneous, or a detour from exploring another test idea, or user manual, error message list etc)

Validity allsorts

- *Write test cases for invalid as well as valid input test conditions*

Today's view

- Still a core axiom of testing today, & often the distinguishing factor between system testing and other forms of testing that may neglect invalid cases (developer testing, user testing)
- Invalid cases cover simple mistakes. Ed Kit later extended axiom to unexpected and expected inputs, and outputs too. Also test for feasible scenarios of incoherent cases in input, control flow, preconditions, environment, etc, where crazy/confused actions can be taken by hackers or "lost" users (a.k.a misuse cases)

What’s expected

- *A necessary part of every test case is a description of the expected output or result*
- Eliminates the problem of “the eye seeing what it wants to see”
- Tests should be self-checking, or test tool should be able to automatically check results
- Not possible in some cases, e.g mathematical software, reliability tests involving system disruption.
- Should be able to anticipate “acceptable” behaviour for test cases or action in an exploratory session.

Today's view

- A core axiom, but "result" could be output, post condition data & program state, & envirnmt, with matching pre-test info. Automated tools only check results they are told to look for, so humans better at verifying simple visible results.
- Typically just have action, predicted result then tester (or tool) tracks actual result during test
- ET pairs have more eyes to see results.
- Self-checking (e.g check delete worked) good practice in ET or scripted; may need technical knowledge (to check below GUI, e.g in DB).

Expected, Anticipated, Acceptable

Test Case has:	**Verify Against:**	**Verify what/ When determined**
Explicit data	Expected output	Data-when created
Implicit data	Anticipated result	Data-when run
Elaborated steps, defined results	Expected result	Action-when created
Elaborated steps, undefined results, e.g test security or stability (e.g disruption of infrastructure)	Acceptable behaviour, e.g security not compromised, system consistent & not corrupted after outage	Action-when run
Test Idea (for exploration)	Anticipated result or acceptable behaviour	Action-when run

The Testability Quest

- *Ensure that testability is a key objective in your software design*

- “Software has primarily been developed with 3 primary considerations in mind: time to market, budget and functionality. ... What hasn’t been on the list but is finally starting to surface is testability.”
  - *Linda Hayes*

Today's view

- Most of us have heard about testability but still coming to grips with it
- Studies have shown 2/3rds of software lifetime cost is repairs after release
- Testability can reduce this cost, but should be included at build/buy time
- Best to include testability requirements with business/ functional requirements during design, but this is still not widely accepted practice.

Concrete Testability

- Extra testing information on the screen, snapshots, logging, etc to make testing faster and easier
- Tools to create, filter, refresh, reconstruct or reveal information; establish security hierarchies or reference tables; or roll dates forward or back
- Design for automation (TDD, etc)

When to End

- *One of the most difficult problems in testing is knowing when to stop*
- Impossible to test everything
- Economic problem of choosing finite number of test cases to maximize bug discovery for a given investment

Today's view

- Risk based approaches focus on finding big bugs first, by starting with high risk tests first, or using exploratory techniques to investigate anticipated or actual anomaly clusters
- Once sufficient functional coverage achieved (typically by broad range of scripted tests), defect detection rates are good indicative measure of end point of that technique, but what happens if you change your focus or try another technique?
- Still major issue of having to predict testing duration before testing starts in dev projects (how many retests should we schedule?)

Variations on the theme

- *Complete testing is not possible* – Bill Hetzel
- *Testing is a risk based exercise* – Bill Hetzel, Ron Patton
- *Exhaustive testing is impossible, except for trivial examples, so use risk and priorities to focus testing* – ISTQB Principle

An original opinion

- "A reader will not agree with all of the author's ideas and may even feel that some of his ideas are not adequately supported. … I expect that many of the topics discussed … will become second nature to us within … several years. Some will probably be refuted by later research and experiences. … It would be a wise investment [to buy]. - *J.P Langer*, *1977*

Final view

- Read the paper for more detail and some other axioms
- While we've seen a mix of possibly "good, bad & ugly", the core axioms still do what we'd reasonably expect them to do given technology & SDLC changes. Some of the "ugly" ones still have wide following. Others have become opinions only, e.g devs cannot test own work
- We need to use the "forgotten" axioms that can really help us do our jobs better, i.e bug clusters, testability, and push new practices, i.e TDD, ET

Thank you, & thanks to Glenford Myers and the other authors too !

References

- "Software Reliability, Principles & Practices", Glenford Myers, 1976
- Don Mills quote: RE mailing list, Mar 18 2003, referring to "Secrets of Software Quality"
- Cost of repairs after release: various sources including Sue Scully, SEA
- Pareto defect rules: "Software Defect Reduction Top 10 List", Basili & Boehm, IEEE Computer, Jan 2001. Rules 4 & 5
- Brian Marick story: draft version of one of his papers
- Bill Hetzel's axioms from his software testing book
- Ron Patton's axioms from his book "Software Testing"
- Ed Kit's axioms from his book "Software Testing in the real world"
- Cem Kaner's "What IS a good test case?"and many of his writings!
- Linda Hayes quotes, web articles Jan 2001 and Dec 2003
- ET refs at satisfice.com, workroom-productions.com, stickyminds.com
- Testability refs, see Bret Pettichord's articles
- TDD refs at agile testing sites, try searching with Kartoo!
- Assorted uni lecture notes, for other axioms. etc
- J.P.Langer book review, IBM Systems Journal, No 3, 1977

Back to the Beginning : Testing Axioms revisited

Erik Petersen emprove@gmail.com

In 1976, Glenford Myers published his book "**Software Reliability – Principles and Practices**" which was the first entire book devoted to the subject. It included a section on software testing, and a series of testing axioms (i.e established or accepted truths, more typically known today as principles). Realizing there was a demand for similar material, Myers followed this up in 1979 with a book, "The Art of Software Testing". We will look at the axioms from the original book.
It is one of the first books on testing, and it can still be bought second hand. It is still seen in some academic circles as a valuable resource and traditionalist university courses teach most (9 of 16) axioms as "a useful guide to good testing practice."
Now 30 years later, let's revisit the original axioms (plus some newer ones) and look at their recognition and relevance today. Note that the published order of the axioms has been rearranged for this paper.

Of Changing Times

In 1969, man landed on the moon and that was only possible because of computers. What sort of computer was that? It was basically a box that displayed a handful of numbers. A user then had to look up a book to determine what the output meant. That is so primitive compared to our current notion of a computer it is almost impossible to comprehend, even if it had a million dollar price tag when it was built. So how similar is this to Myer's notion of a computer? Luckily it is very different.

Myers worked for IBM, which dominated the computing industry, and mainframes dominated IBM. There were other computers around, but these were many levels of complexity simpler. Computers were based around "Green" screens, text based and batch oriented, needing many people to contribute to a system that could rely on program and job control languages needed to create something that a computer could execute. Operators were heavily involved, running and rerunning jobs, delivering printouts etc

Large mainframe computers had 128K of memory but were very expensive (64K was the norm) because of the restrictions of metal core rings strung around wires. It is amazing to think of the progress of technology since then. One 30 gigabyte music player is more than 200 million times larger than a large mainframe computer from the mid 1970s, so it probably has more storage than all the mainframe computers in existence at that time!

PCs were still in research labs; awaiting green screen PCs in early 80s; then GUIs, mice, etc in the late 1980s, then networking then internet then wireless, voice, etc and who knows what else…..

Many of Myers' ideas echo IBM culture of the time. Testing was part of development and not a standalone discipline. So has this surge of technological progress been matched by a surge in program size? Most definitely. Program sizes have increased exponentially, for example Vi the UNIX text editor is 73 kb *vs* MS-Word the graphical user interface editor which is more than 10,000 kb, excluding external function libraries.

With character based programs, control flow mostly was mostly sequential, and all input was keyboard based. Control flow today is much more complex with GUI based controls (using mouse, keyboard, tablet), when a user can perform random navigations as part of any input. This gets even worse with web applications, and will be more complex with voice recognition systems that will soon be common place. As well as this, there are often unpredictable delivery mechanisms, unpredictable load and dynamic infrastructures now used to implement systems are a challenge to system stability and reliability.

On top of this, we now have the ability to design custom interface makes usability much more important. While it is more common to buy programs now rather than build them, there is still the need to configure and define parameters which can a very complex exercise.

So within this framework of incredible change, how many of the axioms are we still familiar with? How many are still relevant? Are there missing axioms that Myers did not consider?

The Axioms

Axiom the first

Assign your most creative programmers to testing
Myers disputed the common view of testing as mundane and boring work, saying experience shows the opposite is more accurate. Some university courses paraphrase “most creative” to “best”. At the time this would have been a very adventurous claim.

Today, we recognize testing as a separate skill set focussed typically on functional testing. The narrow programmer mindset vs. the broader tester mindset has been recognized, so testing is now a distinct role

While writing test cases requires creativity, running scripted test cases can be very boring and mundane! Some places still restrict juniors to test execution but this attitude is slowly becoming extinct. Exploratory testing with its continual mix of planning and execution is a very creative skill.

Projects now often include testsmith roles to create new tools to support testing. These are typically using a mix of dev and test skills, using scripting languages like Ruby. This axiom is dead for system testing (apart from IT shops without system testers!) but is coming back into vogue in agile projects, particularly for unit testing (more on this later …)

Later authors paraphrased and expanded the idea, dropping the reference to programming.
Testing work is creative and difficult – Bill Hetzel, 1998
Software testing is a disciplined technical profession requiring training - Ron Patton, 2000

Axiom the second

Thoroughly inspect the results of each test

Myers said testing was an activity that required creative thinking, not just rote process following. He said tests are useless if the results only glanced at and need to be examined in detail. Testing was more than just executing set number of cases. Programmers may react to new error with "I tested that but somehow didn't see that in the output" (does this happen today? ☺)

This doesn't seem to be a problem with system testers, but often the time to write new cases is limited or non-existent.

It may be a problem with immature developers who do not pay proper attention to their testing, but a bigger issue is developer coverage of all relevant situations and moving beyond valid tests to invalid ones (which Myers mentions below).

The issue of coverage is being addressed by Test Driven Development or TDD (see later) which not only inspects results automatically but fails the build if any are found

A related issue to "just execute a set number of test cases" is ending a phase of testing with poor exit criteria of "100% of tests passed" which discourages use of difficult tests and ignores issue of deferred bugs fixed in later releases. Exit criteria are much better if based on quality, e.g. all high priority defects closed.

Axiom the third

The design of a system should be such that each module is integrated into the system only once

This rather technical axiom is relevant to testing when viewed as part of development. Myers explains this is basically about design simplicity, utilizing small modules that perform single functions. This doesn't really fit as a testing axiom, but anticipated incremental development styles which deliver functionality as standalone chunks of functionality that gradually integrate together into a complete system over a series of independent releases.

If this was paraphrased into an axiom of software design, it would still be current. University course paraphrases include "Ensure that system design allows for straight forward testing" & "Get the design right and you will have less code to test" (which doesn't exactly follow, but is still true especially if mistake proofing is taken into consideration, e.g. forcing users to select from a list rather than allowing free text input)

Technology and networking have meant it is much more common today to combine multiple systems, so the integration of these is a major issue. The new glue of Extended Markup Language XML and set industry XML formats are simplifying this considerably, e.g recipeML. Soon systems that support an accepted industry XML format will be able to exchange information with other systems that support the same formats and extract the relevant information without additional programming! Now if we could only get similar advances across the board!

Axiom the fourth

It is impossible to test your own program

Myers felt programmers shouldn't do any testing, even unit testing. Was this ever followed for unit testing? This was and still is quite a controversial idea.

Today the programmer is typically seen as the best person to do unit testing, if for no other reason that it keeps communication issues to a minimum. Studies have shown the more people involved in creating software the more bugs are created, so the idea of independent unit testers could create more problems than it solves. Programmers traditionally haven't liked system testing, but may have to on small projects if system testers aren't available. The natural aversion of programmers to testing is reducing significantly, as new agile approaches are involve programmers in more formal repeatable unit testing and they are looking to system testers for guidance.

The previous "integrating modules" axiom related to design simplicity in terms of modules focussing on single functions. Test driven development (TDD) forces design simplicity during coding by having coders repeatedly add an automated true/false unit test then write matching code, and re-executing the tests with each build of their program. The system compiles very quickly (< 5 minutes) so common to have several daily builds of all programs. The build fails if any test fails (red status vs. green pass status). While it is possible to write all the tests first then write the program, this does not allow feedback between the test and code process which may include natural redesign of the code during the process.

In this case programmers have to unit test their own code, though the "testing" is a key part of the stepwise design. It should really be called design-driven during development, but test-driven during repair. The regression unit test suite of automated tests always runs (and as a bonus finds most of those mysterious bugs that system testers struggle to find)

An updated and widely accepted version of the axiom is *Testing requires independence*. Bill Hetzel

The implication in this is it is referring to Functional testing. This could be independent of knowledge of the mechanics of the software, or independent of a reporting line through time-based (development or project) managers

Axiom the fifth

All programs should be shipped with a small number of test cases that detect installation errors

Myers' suggested cases included checking for all files and hardware, the contents of the first record of each file, & whether all parts of the software are present.

Has this <u>ever</u> happened as described by Myers? This sounds sensible and may occur to some degree in custom software installations but hasn't been adopted as a general practice.

While there is some great specialist installation software, a lot of the nightmares of PC installations (and errors like old external function libraries overwriting newer already installed ones!) helped push the adoption of web interfaces to simplify installations.

There is an interesting parallel with the Google approach of using lower quality domestic PCs as hardware and including a test of the box's health before requesting that it perform a relevant function. If the test fails, the box is removed from the pool and a process commenced to replace it.

Axiom the sixth

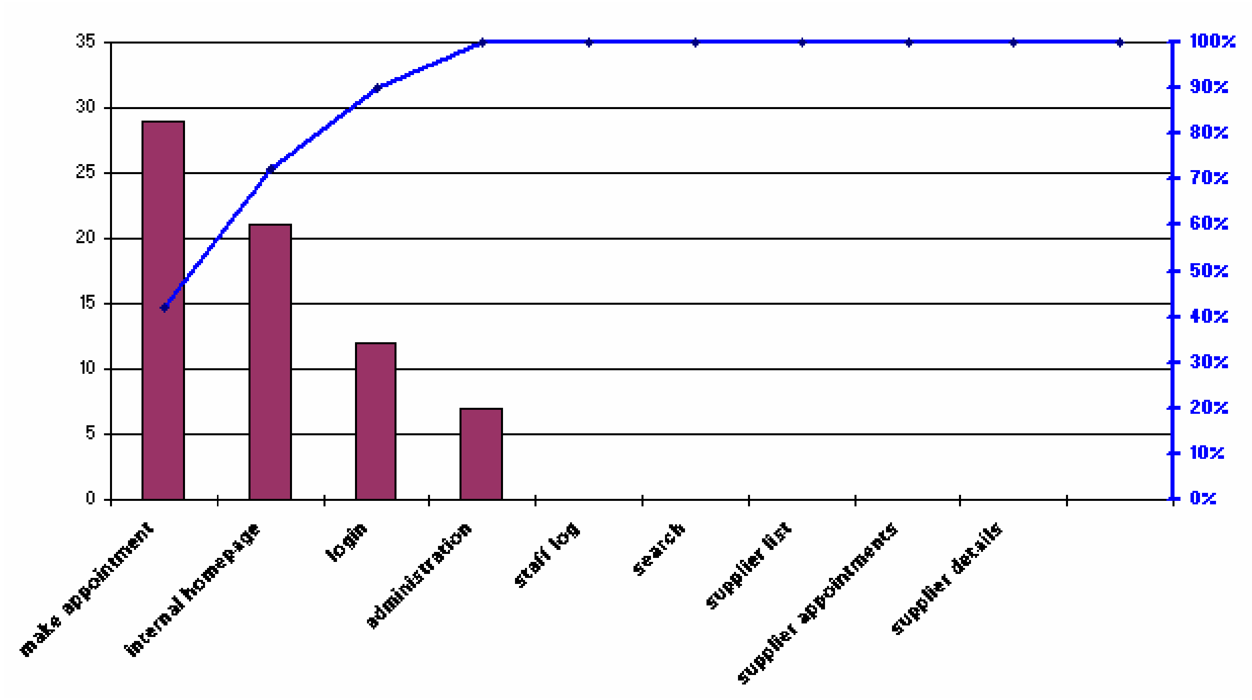
As the number of detected errors in a piece of software increases, the probability of the existence of more undetected errors also increases
Myers said this is a counter-intuitive phenomenon: bugs cluster! If a particular part of the system appears highly error prone during testing, extra testing efforts should be focussed there. A university course paraphrase is: "Bugs are not hermits, it is likely to have friends nearby".

All of the axioms, this is probably the most useful at increasing tester efficiency and helping to find bugs faster. Why is it that it is mostly unknown amongst testers who have learnt on the job, though they have may have discovered it intuitively? The common approach today ignores this axiom, basing testing instead on engineering methods (where we pre-plan detailed scripted tests for everything). Why not use the scientific method? Use feedback to determine areas of focus based on where we find the bugs, particularly the high priority ones.

While this was also included by Ron Patton in his axioms published in 2000 (*The more bugs you find in an area, the more bugs you expect* - Ron Patton), it has recently been included as a principle in the ISTQB principles so is now getting new recognition. (*A small number of modules contain the majority of bugs found by testing, and also in the field* – ISTQB Principle, 2005).
Readers familiar with my presentations will know it is one of my favourite themes, and I can now report that thanks to my efforts there is explicit mention of bug clusters in the relevant documents in the final draft of new IEEE 829 test documentation standard.

One thing that still has not gained acceptance is Myer's approach of pausing the test effort to write scripted cases once clusters are located.

There is also a special 80:20 rule (a.k.a Pareto Principle) flavour for this axiom: When unit and integration testing has been properly performed, effective system testing will find about 80% of the defects in 20% of the modules (**standard Pareto**) and about half the modules are typically defect free (**unique to software!**) This may range from 60-90%, with 80% as a median value. For load testing, about 40% of modules have 60% defects. About 90% of the production downtime comes from at most 10% of the defects



This real Pareto chart from a simple web system shows all bugs are concentrated in about ½ the functions!

Axiom the seventh

Avoid non-reproducible or on-the-fly testing
Myers said testing without scripted cases needs tests to be redeveloped for retesting. He claimed testing was already expensive enough without this, but it could be done for throw-away programs. He also insisted that the tests be documented to allow them to be reused by anyone.

While the defect clusters axiom has the potential to be very helpful in testing efforts but has been largely unknown, this axiom has been widely acknowledged and reflects the attitudes of many traditionalist testers today. How can we reconcile it with Myer's bug cluster axiom, which requires feedback from testing to focus test efforts? Myer's assumed wrongly that initial broad testing would be followed by additional test creation in the bug cluster areas, but this does not happen. Instead of this, the traditional approach is deep testing of all areas, without acknowledging bug cluster areas, though these may receive extra attention if they match any risk analysis of the functionality. The Pareto bug rule indicates some tests may never find bugs, so it is much better to focus on broad coverage of tests, ordered according to risk. If you are scripting, don't have too many detailed tests until an area is confirmed to have bugs.

Writing tests to be reused by anyone is very difficult with the size of modern systems, requiring much effort to achieve accurate detailed tests. This may have been possible with the smaller programs that Myers was familiar with. Higher level tests are more practical, and more resistant to minor changes of functionality.
A more recent paraphrase is *Testing must be planned* – Bill Hetzel

A new axiom

So why is the seventh axiom an axiom at all? It comes straight from engineering approaches that say everything needs to be defined in advance before it has value. Do we insist that scientists provide detailed explanations of all their experiments on their way to a scientific discovery? No, because the experiments they choose to perform next are based on the results of the experiments already performed.

While fully planned testing may find many defects, it may take time to detect them, and may not detect closely related bugs. I will offer an axiom of my own at this point, ***Efficient testing must be directed by feedback***. Efficiency is related to return in terms of effort, bang for your bug bucks. These just-in-time approaches are also recognized by the ISTQB syllabus which describes experience-based testing as a viable testing technique.

So, does test creation need tests to be fully described to step level first? No. We can link design, direction and discovery dynamically to create tests. The main experience-based technique is Exploratory Testing (ET), following the scientific approach and utilizing feedback – the result of the last test may determine the next test. Informal opinion indicates the most severe bugs are found with ET not scripted tests.

If multiple testers are involved, it is good to create a list of test ideas used as a high level plan to guide a test session to reduce testing overlap. While it may not be a requirement to exactly reproduce a test, it should be an option if required. Keep notes/records during ET to facilitate retests; track coverage and completion rate to predict end time. By listing the anticipated time to complete the investigation of each test idea, the total time or time remaining can be estimated.

Tools can be very useful to record actions performed, e.g screen shot software such as gadwin printscreen which takes shots when triggered by the user, or timesnapper which takes automatic screen shots which can then replay like a movie. Oracle (result predicting) tools may also be needed to verify results, e.g to validate mathematical or statistical results, or verify under the hood behaviour, e.g correct record count retrieved for a particular query.

ET usage varies from investigating functionality around bug fixes thru to full ET sessions. A session may last from 30 to 90 minutes. All projects should try to spend a fair proportion of test execution doing ET if testing a GUI based, non-regulated system. Session based techniques to co-ordinate group testing efforts include the approaches of James & Jon Bach and James Lyndsay. Also see the attack techniques of James Whittaker, or quicktests of James Bach & Michael Bolton. Variations include paired exploratory testing, where two testers work together.

Axiom the eighth

A good test case has a high probability of detecting an undiscovered error, not a test case that shows that the program works correctly

Here are some quotes describing how this axiom has been misused.
“Unfortunate consequences of taking Myers literally, … the (nameless) IBM lab set up a test team with the very aggressive mission of proving that the programmers committed errors. … relationships between programmers and testers reached the

heights of non-co-operativeness, & the quality of the programmers work plummeted"
- *Don Mills*

"I was … speechless when a fellow professional said …she was going to discard the majority of her regression tests because they had failed to find errors. … I asked why she was considering this to which she confidently replied, well so-and-so says that tests that don't find problems aren't worthwhile" - *Linda Hayes*

As an aside, another so-and-so (Boris Beizer) has discovered a reason to regularly revise scripted tests if it is important to continue finding bugs. The Pesticide Paradox states that bugs get resistant to the same tests, like pesticide. Repeating the same tests (after fixes are made) reduces their ability to find bugs. While regression tests are usually designed to prove correct functionality is unchanged, altering them will increase their chances of finding bugs. This is also an ISTQB principle. This also seems to weaken the case for axiom seven, that tests must be planned in advance. Goodness is related to purpose though. If tests are designed to be regression testing that functions that should be unchanged are not changed, it may not be relevant that they do not detect new bugs after a while.

So how do we measure if a test case is "good"? Is the glass half full or half empty? We have two equivalent notions, Risk (**find bugs**) vs confidence (**check it works**). At different times, it may be better to talk about one in preference to the other, but both are related. One university thesis attempted to claim the goal of testing is to "find bugs", and the aim of testing is "check it works" (?!)

Test case "goodness" is based on context. What are the:

- Information objectives (e.g find bugs, test to spec)
- Test attributes (e.g atypical, easier to evaluate)
- Testing type (e.g stress, user, domain, scenario)
  (see Cem Kaner's "What IS a good test case?" for more)

A test case may be

- A test idea
  - Formalized by varying degrees into a elaborated test case
  - Expanded to varying degrees into an exploratory session
- An elaborated test case, defined step by step

Test execution may be by test case, or exploratory session (based on a test idea, or spontaneous, or a detour from exploring another test idea, or user manual, error message list etc)

Axiom the ninth

Write test cases for invalid as well as valid input test conditions

Still a core axiom of testing today, this is often the distinguishing factor between system testing and other forms of testing that may neglect invalid cases (typically developer testing and user testing)

Invalid cases cover simple mistakes. Ed Kit later extended the axiom (in 1995) to unexpected and expected inputs, and outputs too. We should also test for feasible scenarios of "disturbed" cases in input, control flow, preconditions, environment, etc, where crazy/confused actions can be taken by hackers or "lost" users (a.k.a misuse cases)

Axiom the tenth

A necessary part of every test case is a description of the expected output or result
Myers said this eliminates the problem of "the eye seeing what it wants to see". He said tests should be self-checking, or a test tool should be able to automatically check results, though he acknowledged it was not possible in some cases, e.g mathematical software,

This is a core axiom of modern system testing practice, but "result" could be output, post condition data & program state, database state & environment, etc with matching pre-test information. Typically we only record the output state, though at some stage all results should be checked.

Typically the test describes an action to take, with a predicted result then tester (or tool) tracks actual result during test. Automated tools only check results they are told to look for, so humans are better at verifying simple visible results (though a lot slower and less reliable). Debugging of failed automated tests can be very complicated and time consuming.

Scientific method talks about the importance of thought experiments to anticipate outcomes of an experiment. This is also very important in exploratory testing. ET pairs have more eyes to see results, as well.

Self-checking (e.g after a delete test, check that the item can no longer be retrieved) is a good practice in ET or scripted testing. This also should be checked structurally at some stage but may require technical knowledge (to check below GUI, e.g in database rather than just on screen).

Test Case has:	**Verify Against:**	**Verify what/ When determined**
Explicit data	Expected output	Data-when created
Implicit data	Anticipated result	Data-when run
Elaborated steps, defined results	Expected result	Action-when created
Elaborated steps, undefined results, e.g test security or stability (e.g disruption of infrastructure)	Acceptable behaviour, e.g security not compromised, system consistent & not corrupted after outage	Action-when run
Test Idea (for exploration)	Anticipated result or acceptable behaviour	Action-when run

The table above shows the different sorts of tests and the way expected results are defined.

Axiom the eleventh

Never alter the program to make testing easier
The context of this was temporarily altering the program to test it, then restoring it to original form to release.

This doesn't seem to happen today, except for one bizarre case Brian Marick relates where the programmers asked to see the test cases and hard wired the code to return the correct results for the tests (allowing all the tests to pass) and fail for other input! They collected their money and ran before the users realized what had happened. Effectively they built a stub instead of a system.

This is a common practice with data, modifying it to create specific data conditions that may not be in production data or filtering production data to allow testing to occur with smaller amounts of data instead allowing reports and other processes to run faster. It can also occur with simplified maintenance tables, security groupings, etc.

Axiom the twelfth

Ensure that testability is a key objective in your software

"Software has primarily been developed with 3 primary considerations in mind: time to market, budget and functionality. ... What hasn't been on the list but is finally starting to surface is testability."
 - *Linda Hayes*

Most of us have heard about testability but are still coming to grips with it. Studies have shown 2/3rds of software lifetime cost is repairs after release. Testability can reduce this cost, but should be included at build/buy time. It is best to include testability requirements with business/ functional requirements during design, but this is still not widely accepted practice. Some agile teams appear to resist functional testability features on the grounds that it is a YAGNI (You Aint Gonna Need It) for the users!

Some examples of testability features include:

- Extra testing information on the screen, snapshots, logging, etc to make testing faster and easier
- Tools to create, filter, refresh, reconstruct or reveal information; establish security hierarchies or reference tables; or roll dates forward or back
- Design for automation (TDD, etc)

Axiom the thirteenth

If you do not have written objectives for your product, or if your objectives are unmeasurable, then you cannot perform a system test.

Myers then says “real” system tests should be done in production or near production (**validation**); using test env is poor cousin (**verification**)!

Who system tests in production?!!! Apart from pre-production performance testing of the first release, the production environment is off limits.
The terms have now changed meanings as well, and are part of ISQTB syllabus.
Verification: are we building the product right?
Software should conform to specification
Validation: are we building the right product?
Software should do what user requires.
Both terms are in ISTQB syllabus.
ISTQB has a principle to cover validation.
Absence of errors: Testing against a model does not ensure that it is the correct model for users or usable by them
- ISTQB Principle (paraphrased)

Axiom the fourteenth

Testing, as almost every other activity, must start with objectives
Myer’s view is very low level, e.g number of paths or conditional branches to be executed, and relative % of bugs to be detected in each phase. This is a sensible definition for him as testing was part of development,

Today a typical goal is functional or business reqs coverage, & checking non-func reqs met. Does anyone measure the relative bug detected % at each phase of the development lifecycle?
Boris Beizer & other traditionalists have long pushed 100% code and branch coverage despite size of modern programs. Typically the cost of coverage tools has limited their usage, but they now come bundled with coding tools. Will this increase their use? Well, spell checkers are in email programs but are they always used? And has anyone worked on projects that have hit this goal? Even the top industry average appears to be only 85%. And code coverage doesn’t find all bugs (e.g variations in data domains, function calls, messages etc)! Coverage methods are included in the ISTQB syllabus. This should be supplemented with the other coverage methods, e.g functional, or data and message flows.
This can easily reach overkill as a customer requirement, e.g 100% path coverage (implying testing all path combinations, inc. fatal errors?!)
This axiom is similar to an ISTQB principle, also covering test involvement from reviews onward related to a Hetzel axiom.
Testing should begin early in the life cycle, focusing on defined objectives – ISTQB Principle
An important reason for testing is to prevent bugs from occurring – Bill Hetzel
There doesn’t seem to be any principles that deal specifically with coverage alone. The closest is an anonymous saying, *Test 1000 things once not one thing 1000 times*. This is quite surprising given the modern appreciation of the importance of risk to highlight the most important areas to test in depth, but this needs to be balanced with broad testing as well.

Axiom the fifteenth

Use your own product in production before you expect others to use it

This is still a good idea, typically followed by enlightened software producers ”eating their own dog food”.

Axiom the sixteenth

One of the most difficult problems in testing is knowing when to stop
Myers realized it was impossible to test everything, so he said it became an economic problem of choosing a finite number of test cases to maximize bug discovery for a given investment

This seems to be moving towards our modern risk based approaches without actually mentioning risk. Myers did not include the option of reducing the proposed feature set as an option for increasing the level of quality. Risk based approaches focus on finding big bugs first, by starting with high risk tests first, or using exploratory techniques to investigate anticipated or actual bug clusters
Once sufficient functional coverage achieved (typically by broad range of scripted tests), defect detection rates are a good indicative measure of end point of that technique, but what happens if you change your focus or try another technique?
We still have a major issue of having to predict testing duration before testing starts in dev projects (how many retests should we schedule?) This is made easier if we have previous projects to base our estimated quality levels on.

More recent principles have explicitly included risk.
Complete testing is not possible – Bill Hetzel
Testing is a risk based exercise – Bill Hetzel, Ron Patton
Exhaustive testing is impossible, except for trivial examples, so use risk and priorities to focus testing – ISTQB Principle

Other axioms

So have we covered everything that is considered important today? ISTQB includes 2 more principles we haven’t mentioned yet. One was defined by Edgard Dijkstra in 1969 then quoted by Ed Kit then expanded by Ron Patton, and is more for the education of management than for system testers: *Testing shows presence of defects but cannot prove their absence, though it does reduce the probability of undiscovered defects.* – Ron Patton

The last ISTQB principle is the basis of the context based school of testing, but an obvious principle : *Testing is context dependent*. In two words, how do you test something :It depends!

So are there any more published axioms? There are probably many. Ron Patton has several very perceptive ones which are quite recent, from the year 2000. These reflect the reality of software testing, *Not all bugs found are fixed, It is difficult to say when a bug is a bug, Product specifications are never final, Software testers aren’t the most popular members of a project team.*

What we've seen

"A reader will not agree with all of the author's ideas and may even feel that some of his ideas are not adequately supported. … I expect that many of the topics discussed … will become second nature to us within … several years. Some will probably be refuted by later research and experiences. … It would be a wise investment [to buy].
- *J.P Langer, 1977*

This quote from a contemporary review of the "**Software Reliability – Principles and Practices"** book indicates that it was controversial at the time it was published. People may have similar issues with this paper. While some axioms became second nature, they are not necessarily the more useful ones, in fact the opposite sometimes held with some of the now-opinionated axioms, the push for planned tests and "good" tests that had to find errors and programmers who couldn't do their own unit testing. Given that defect clusters vanished from our common knowledge as well, Myers' contention that extra planned tests should be written after finding defect clusters was lost as well.

One of Myers' statements was "A software error is present when the program does not do what its end user reasonably expects it to do". This certainly has relevance today. He may have had a different definition of validation but he had good understanding. He was also talking about risk in all but name.

While we've seen a mix of possibly "good, bad & ugly" axioms, the other core axioms still do what we'd reasonably expect them to do given technology & SDLC changes. Complexity in current program sizes & environments, etc have refuted others. Others have become opinions only. We now realize that circumstances do not always allow detailed test planning down to step level all the time, but exploratory testing techniques have proven that may not be a weakness. Because of the feedback involved, we can use these just-in-time approaches to focus on bug cluster areas and find bugs more efficiently. For whatever reason, many testers are unfamiliar with the concept of bug clusters, but this is being addressed.

We need to use the forgotten axioms that can really help us do our jobs better, i.e bug clusters, testability, and push new practices, i.e TDD, ET. Consider that axioms from other sources, and don't forget, ***Efficient testing must be directed by feedback!***

Will the currently valid axioms lose their validity also? Probably not, but only time will tell……..

Note: Know of other important principles that weren't mentioned? Know who originally said any of these unattributed axioms or if some are wrongly attributed? Want to disagree with my claims? Feel free to send me a note! I have a feeling this paper is only the first version of many!!!! Erik

References:

- "Software Reliability, Principles & Practices", Glenford Myers, 1976
- Don Mills quote: RE mailing list, Mar 18 2003, referring to "Secrets of Software Quality"
- Cost of repairs after release: various sources including Sue Scully, SEA
- Pareto defect rules: "Software Defect Reduction Top 10 List", Basili & Boehm, IEEE Computer, Jan 2001. Rules 4 & 5
- Brian Marick story: draft version of one of his papers
- Bill Hetzel's axioms from his software testing book
- Ron Patton's axioms from his book "Software Testing"
- Ed Kit's axioms from his book "Software Testing in the real world"
- Cem Kaner's "What IS a good test case?"and many of his writings, at kaner.com!
- Linda Hayes quotes, web articles Jan 2001 and Dec 2003
- ET refs at satisfice.com, workroom-productions.com, stickyminds.com
- Testability refs, see Bret Pettichord's articles
- TDD refs at agile testing sites, try searching with Kartoo!
- Assorted uni lecture notes, for msoft coverage %, other axioms
- J.P.Langer book review, IBM Systems Journal, No 3, 1977
- Assorted conversations, lectures and assorted tidbits of information gleaned from many sources.