“Forensic Software Engineering: Take the guesswork out of testing”

by

Les Hatton

Professor of Forensic Software Engineering,
CISM, Kingston University, UK
l.hatton@kingston.ac.uk

Version 1.21: 06/Dec/2007

This presentation is available at http://www.leshatton.org/

Oh bother …

User testing is strengthened after project runs into trouble

Passport agency goes public on test errors

Tony Collins
tony.collins@rbi.co.uk

The Identity and Passport Service is strengthening its testing programme for IT projects after an investigation into the failure of a project to process passport applications online found that insufficient testing and checks were partly to blame.

And in response to Computer Weekly's campaign for greater openness, the government agency has published a report on the lessons it has learned from this and its other key IT projects in 2006.

Bernard Herdan, executive director of service delivery, said the agency had changed its approach to testing on IT projects in the light of the internal inquiry's findings. And he challenged other central departments to follow suit by publishing lessons learned from specific

Open book: project lessons were published after Computer Weekly challenge

KEY POINTS

- Identity and Passport Service goes public with lessons of key IT projects
- Testing boosted after inquiry into online passport applications system
- Report says the agency relied too heavily on supplier for testing
- Director urges other departments to publish lessons learned from projects

Computer Weekly

17-Jan-2007

My tester of the year …

My Alamo hirecar

15-May-2007

It goes on …

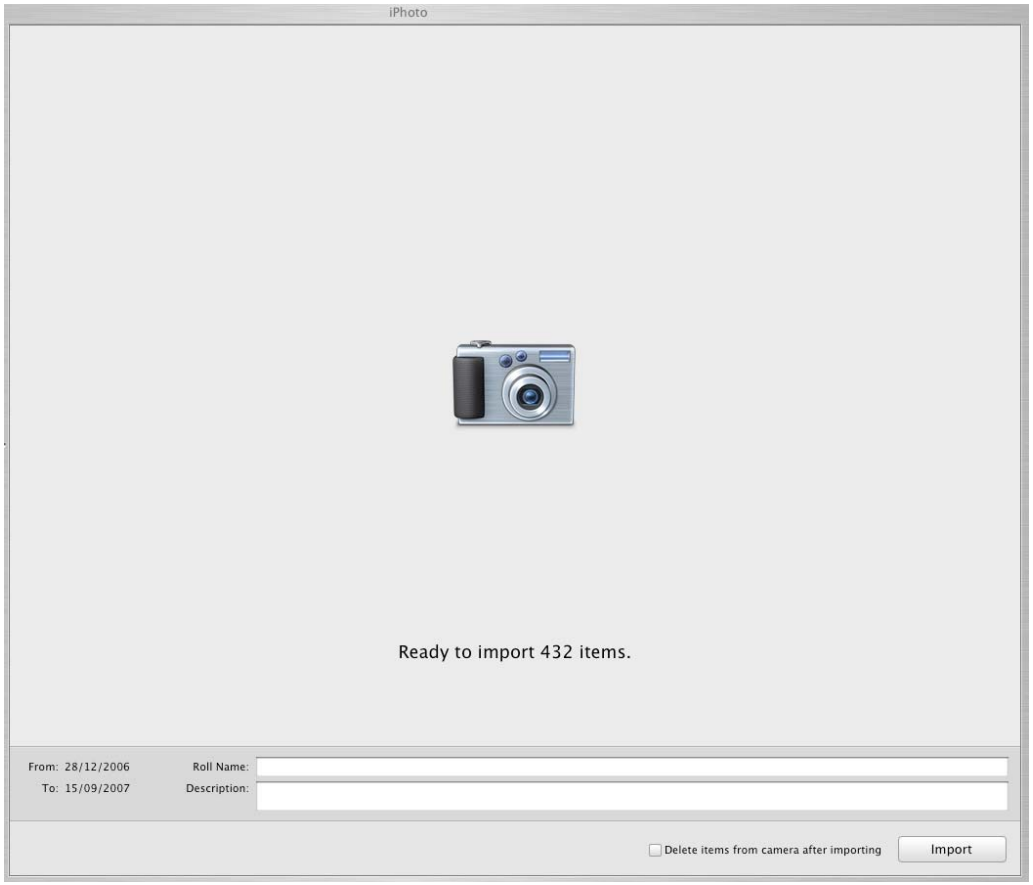
17/July/2007

- Errors in 13 million tax records in UK (lack of field checking software)
- Database problems in driver licensing computer in Honolulu stop drivers receiving licences "… may take 6 months to fix …"
- Software calculates prison sentences incorrectly releasing dangerous criminals in New Zealand

Computer weekly

Testers are there to remind developers what humans look like …

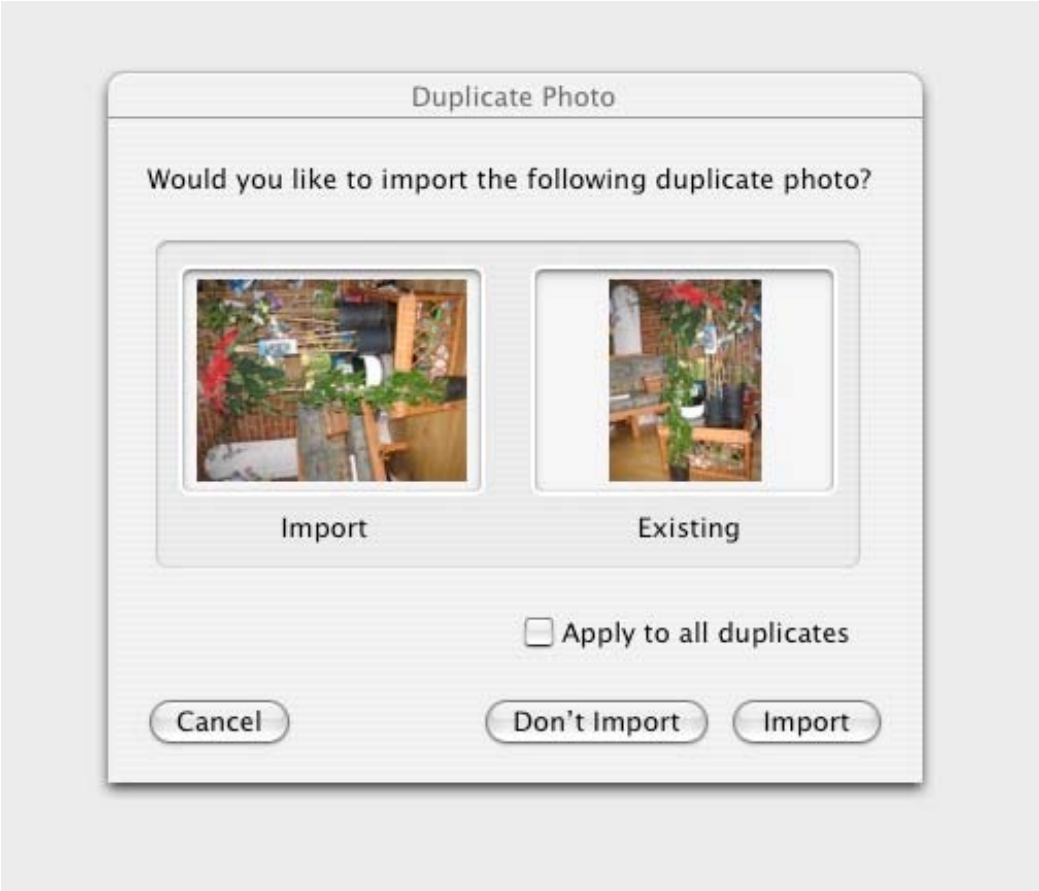
iPhoto ™, ready for action



Testers are there to remind developers what humans look like …

What does this mean ? Note the lack of grouping and causality.

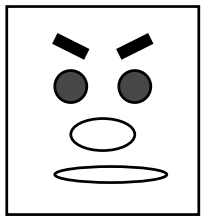
You could test this against requirements but should you ?

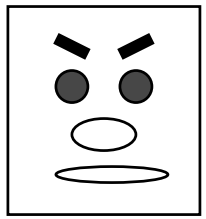


Testers are there to remind developers what humans look like …

This photo has already been imported to your computer

Be explicit …



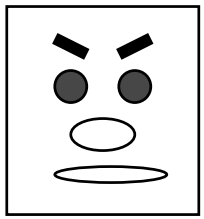


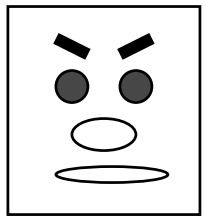
Would you …

- **Like to import it again because you don't trust me**
- **Like to skip it because you do trust me**
- **Like to import this and any other duplicates so I don't have to ask again**
- **Like to skip this and any other duplicates so I don't have to ask again**
- **Like to forget the whole thing and maybe practise the trombone**

Testers are there to remind developers what humans look like …

This photo has already been imported to your computer





Or be causal

Would you …
- **Like to import it again because you don't trust me**
- **Like to skip it because you do trust me**
- **Like to forget the whole thing and maybe practise the trombone**

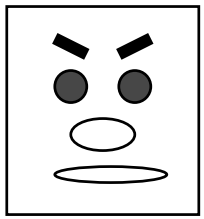
Followed by …

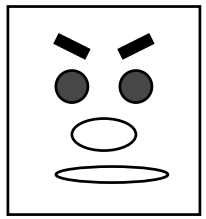
- **Do you want to (skip | accept) subsequent duplicates ?**
- **Do you want me to carry on asking ?**

Testers are there to remind developers what humans look like …

Would you like to import the following duplicate photo ?

Or lay it out properly …





- **Do not import**
- **Import**

- **Treat all subsequent duplicates the same way**

- **Forget the whole thing and maybe play the trombone**

Forensic engineering

To improve a process, you must

- Define a measurable criterion for improvement
- Analyse the measurements regularly to understand what process changes will lead to improvement

For software testing, this means

- We have to have good objective measures. Since a successful test finds defects, *released defects* is an ideal candidate.
- We have to analyse all defects to link them to deficiencies in the test process, if any.

How good is good ?

A useful criterion

- Define a defect as a fault that has failed
- Define an executable line of code (XLOC) as any line of code which generates an executable statement
- Define asymptotic defect density as the *upper bound* of the total number of defects ever found in the product's entire life-cycle divided by the lines of code.

If your asymptotic defect density is < 1 defect per KXLOC (thousand executable lines of code), you are doing about as well as has ever been achieved.

How bad is bad ?

NIST (US National Institute of Standards and Technology)

- 2002 report estimating costs of software failure in US alone at $60 billion per year
- 80% of software development costs are finding and fixing defects
  - Economist Science Technology Quarterly 19/Jun/2003

Royal Academy of Engineering (UK) 2004, reported

- Only 16% of projects in the UK were considered successful
- This suggests that around GBP 17 billion will be wasted in 2003/2004 alone.
  - “The challenges of complex IT projects”, 22/Apr/2004

Extracting patterns

- **Where do we start ?**
  - Aristotleans v. Babylonians: the role of measurement
  - Some examples
- **Complicating factors**
- **Measurements and how to test them**
- **Searching in unstructured texts**

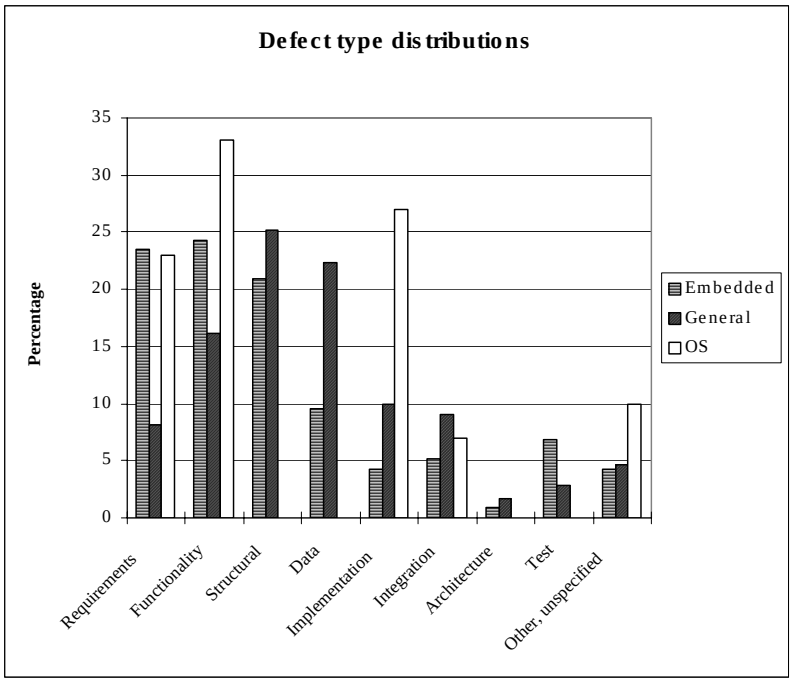
Aristotleans v. Babylonians

- **Aristotleans …**
  - Decide everything by deep thought
- **Babylonians …**
  - Learn the hard way by sticking their fingers in light sockets.

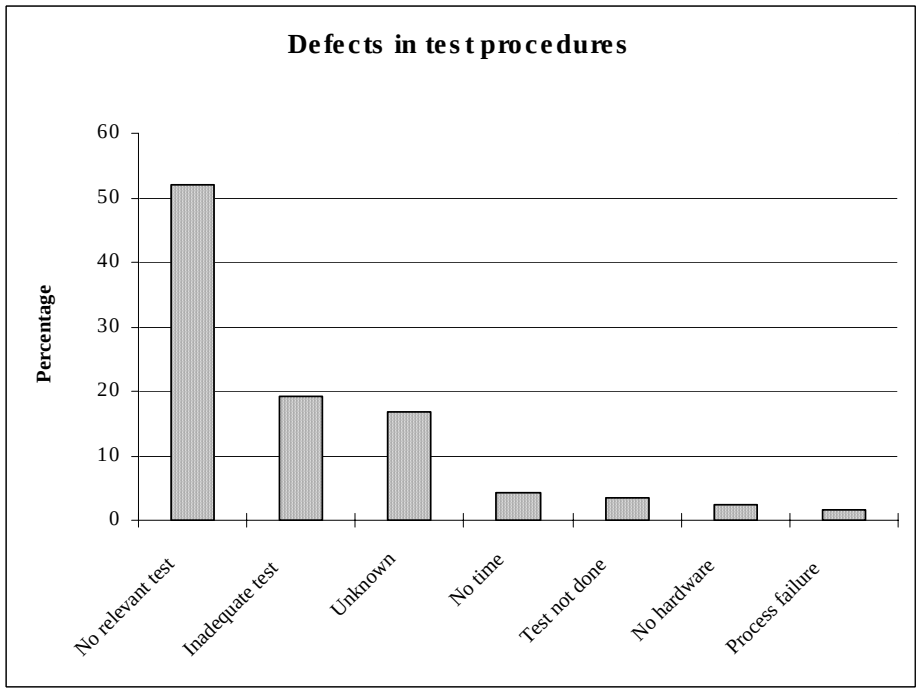
Development is an Aristotlean activity …

… Testing is a Babylonian activity

Typical defect profiles



Vinter and Poulson (1996)



OS supplier (2001)

Measure your own !

Extracting patterns

- **Where do we start ?**
- **Complicating factors**
  - We should test “systems” as well as software
  - Spreadsheets
  - The underlying architecture may not be very reliable
- **Measurements and how to assess them**
- **Searching in unstructured texts**

We test "systems" as well as software

- **Systems inconveniently include human users**
- **Testing involves testing the system as a whole**
  - This may include a mixture of computerisation and additional, poorly documented, time varying and somewhat erratic human behaviour which we attempt to assess with usability testing.

The system you test is often not the system which is used !

British Gas automated gas meter reading, August 2006

- **Note the following**
  - Gas Bill based on two estimates:
    - Human sub-system – They now require me to enter my own reading on their telephone entry system
    - Software sub-system – a telephone entry system attached to their billing database
  - My attempts …
    - Enter my account number.
      - Accepted *without* repeating to check
    - Enter my gas bill reading
      - Repeats to verify and then says "does not agree with our records"
    - Enter same gas bill reading
      - Repeats to verify and accepts

Attempt to check-in online, London Heathrow, November 26th 2007

- **An attempt to check-in online, starring SAS and the worst national airport in the world, London Heathrow**
  - Procedures
    - Human sub-systems – Various rapidly changing and often arbitrary security procedures
    - Software sub-system – Check-in system with web interface and local printing

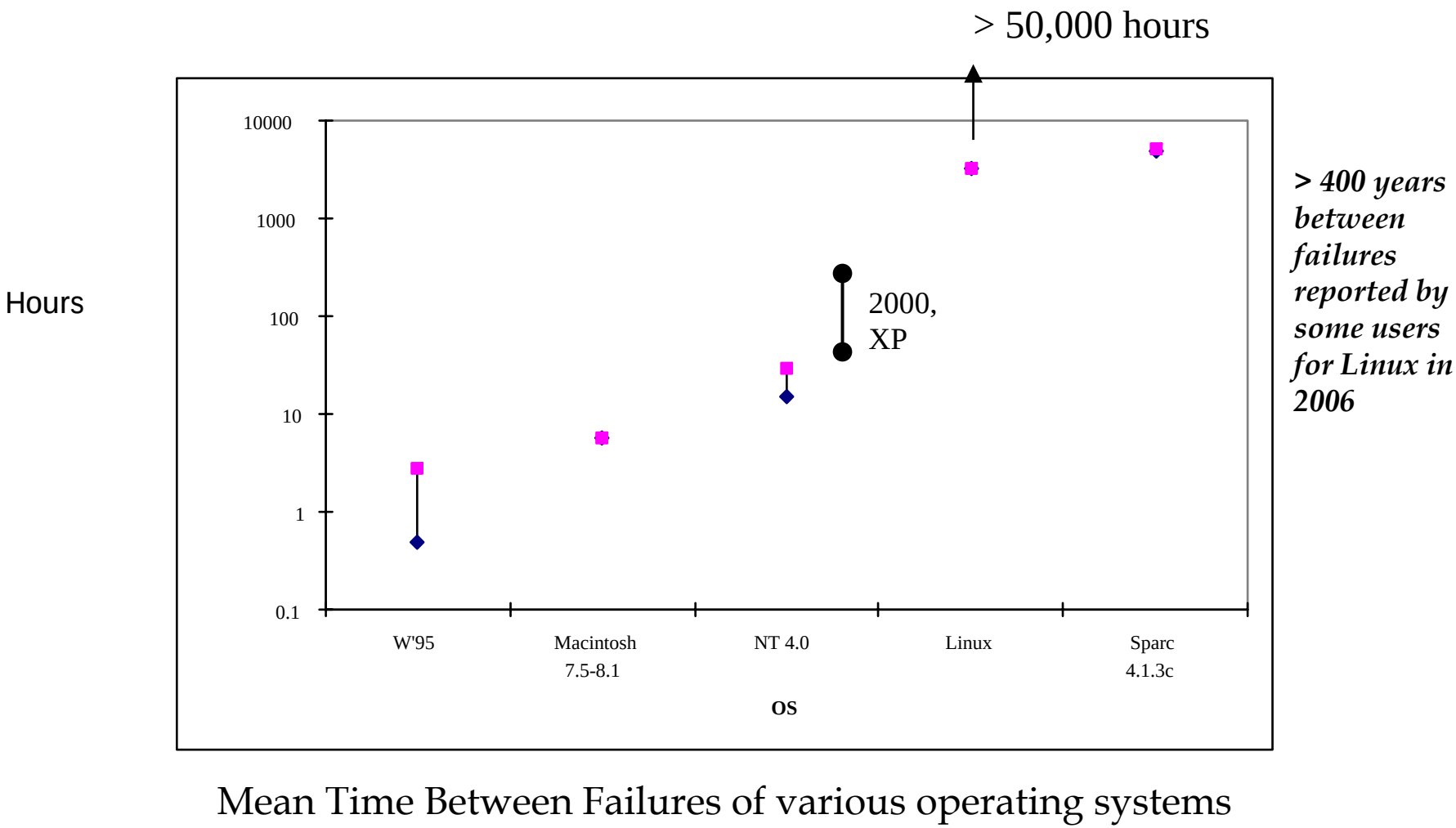
Attempt to check-in online, London Heathrow, November 26th 2007

- **The general idea of online check in is to speed it up …**
  - Step 1:
    - Check in online with SAS and print boarding pass at home.
  - Step 2:
    - Enter flight departures. They call supervisor to validate boarding pass. Unfortunately, their system can't scan it.
  - Step 3:
    - Back to SAS, but I'm already checked in so can't do it again.
  - Step 4:
    - SAS call supervisor and then super-supervisor. Give me second boarding card and *make me promise its me.*
  - Step 5:
    - Back up to departures. They won't let me in as I have two boarding cards. Need to find original person to verify me.

Spreadsheets

- **One of the great liberating applications of the 80s and 90s**
- **One of the major headaches of the 21st century**
  - Weird arithmetic
    - -x^2+1 != 1-x^2 (Allan Stevens 2005)
    - (4/3-1)*3-1 != ((4/3-1)*3-1) If you don't believe me
  - People keep data in them instead of in databases
    - This a major fly in the ointment in most companies because people cannot exchange data.
  - They are hard to test and consequently full of defects
    - 90% of all spreadsheets had errors which led to more than 5% error in the results. Ray Panko (University of Hawaii)
  - They are even harder to search for failure patterns

OS Reliability



Mean Time Between Failures of various operating systems

OS Reliability

24.5 million XP crashes per day

http://www.pcmag.com/article2/0,4149,1210067,00.asp

5% of Windows Computers crash more than twice a day

http://www.nytimes.com/2003/07/25/technology/25SOFT.html

Extracting patterns

- **Where do we start ?**
- **Complicating factors**
- **Measurements and how to assess them**
  - How not to present a result
  - How to present a result
- **Searching in unstructured texts**

Case History 1 – How NOT to present numerical results

The proportion of defects found by external users in this case history of a client server architecture is as follows, (Hatton (2007), IEEE Computer, July):-

Component	Proportion of defects found externally	Proportion of defects found internally
GUI Client	***57.2%***	42.8%
Computation Server	***39.1%***	61.9%

Tentative conclusion:- *External users are more sensitive to defects in GUI clients than in computational servers.*

No, we cannot say this reliably !

Case History 1 – doing it properly

Use the z-test for proportions and assume as a null hypothesis that the distribution of defects found externally by users in the GUI client (p_c)and the computation server (p_s) are in the same proportion.

Then …

$$z = \frac{p_s - p_c}{\sqrt{\hat{p}\hat{q}\left\{\frac{1}{n_1} + \frac{1}{n_2}\right\}}} \sim N(0,1)$$

This gives z = -0.84 which is NOT significant.

We cannot reject the null hypothesis and *we cannot infer any such pattern*

Case History 1 – What can we infer ?

Be very careful to test results for significance before using them !

Sometimes, very useful and highly significant patterns emerge …

Continued testing after delivery reduces the defect density the user sees by about half, providing we can update them regularly.

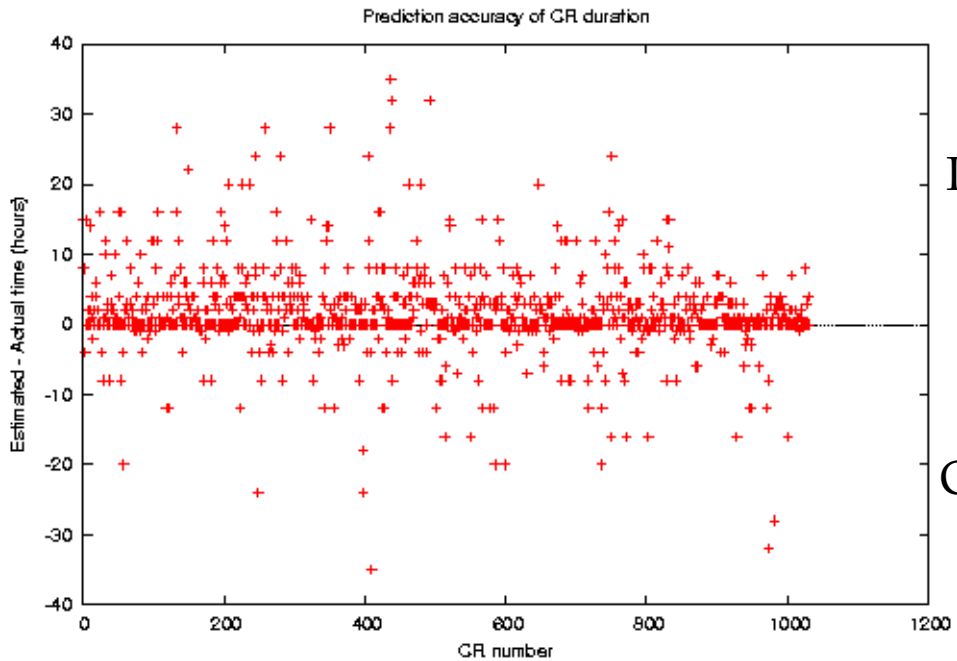
This result is statistically highly significant.

Case History 2 – How good are we at estimating tasks ?

Time →

The difference between Estimated and Actual times of maintenance tasks in a software development project.

(Hatton (2007)
IEEE Computer, May)



Pessimistic

Optimistic

There is very highly significant systematic pessimistic bias but does it change with time ?

Case History 2 – Treading more carefully

The average amount by which engineers over-estimated maintenance tasks in this case history was:-

Data set	Average over-estimate in hours
First half	***2.45***
Second half	***1.2***

Before springing to conclusions, we test it this time …

Case History 2 – How to present numerical results

This time use the z-test for the difference of means in populations, split the population in half and assume as a null hypothesis that the systematic bias does not change.

Then …

$$z = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\left\{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right\}}} \sim N(0,1)$$

This gives z = 3.16 which is VERY HIGHLY significant.

We reject the null hypothesis and *infer the bias is getting less, since X1 = 2.45 hours and X2 = 1.2 hours*

Case History 2 – What can we infer ?

Referring to the same paper, averaging across all the data the following are statistically highly significant …

On average, engineers over-estimate how long maintenance tasks take (corrective, adaptive or perfective) by about 35%

Engineers systematically over-estimate how long a short task will take and under-estimate how long a long task will take.

Engineers improve in estimation skills significantly as projects develop.

Case History 3 – Another example where intuition fails

Code inspections are very widely believed to rely on checklists for their effectiveness. In a study of 107 teams, Hatton IEEE Software (2008) showed:-

Experiment phase	Mean defects found using checklists	Mean defects found without checklists
Phase 1 (2005) – 70 teams	***13.00***	11.09
Phase 2 (2006) – 37 teams	***7.18***	5.72

Conclusion:- *The differences are **not** statistically significant individually or collectively*

Extracting patterns

- **Where do we start ?**
- **Complicating factors**
- **Measurements and how to test them**
- **Searching in unstructured texts**
  - Problem: Defect data is usually disorganised
  - Searching for relationships in disorganised data

The unstructured nature of typical defect data

- **An example from the Common Vulnerabilities Database, CVE-2006-4304, (454,000 lines, 17Mb of unstructured English, http://cve.mitre.org/):-**

  "Buffer overflow in the ppp driver in FreeBSD 4.11 to 6.1 and NetBSD 2.0 through 4.0 beta allows remote attackers to cause a denial of service (panic), obtain sensitive information, and possibly execute arbitrary code via crafted Link Control Protocol (LCP) packets with an option length that exceeds the overall length, which triggers the overflow in (1) pppoe and (2) ippp."

Chance discovery

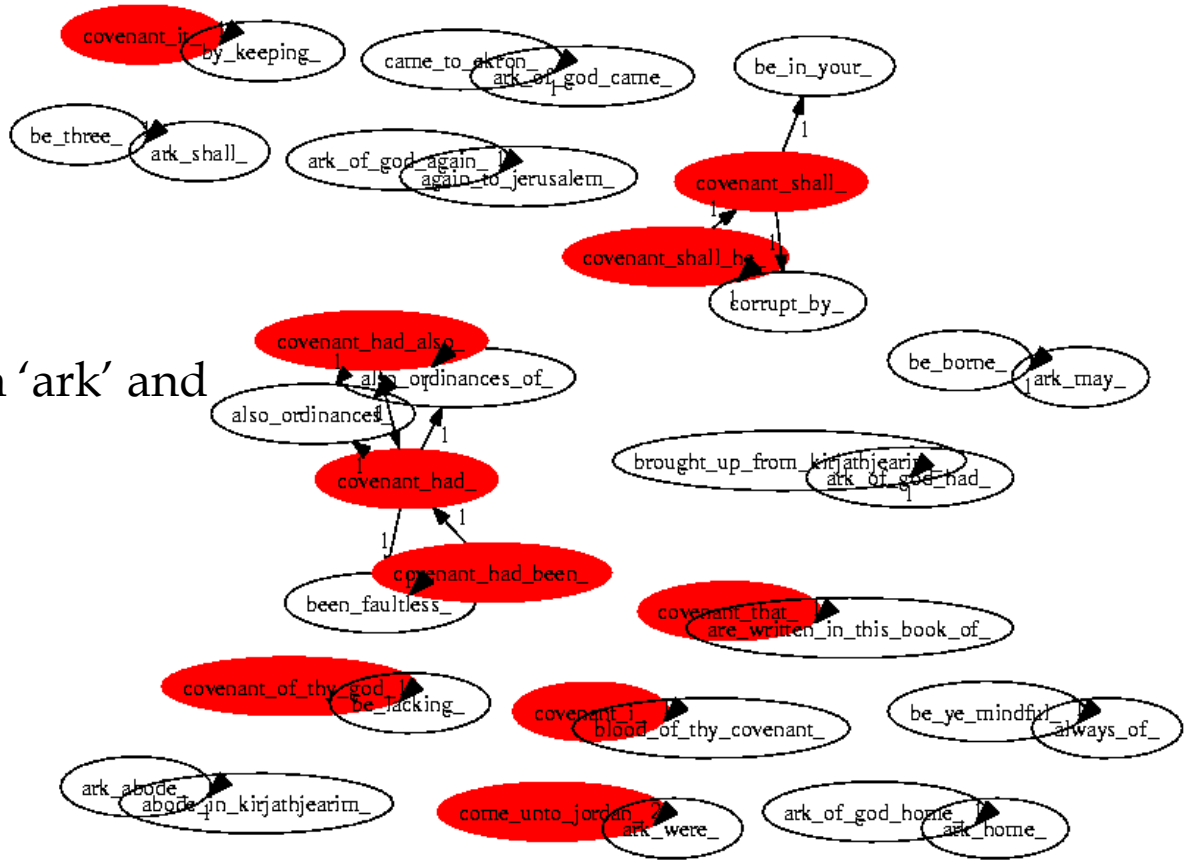
Chance discovery is a class of algorithms to discover relationships in unstructured data. This implementation uses Persistence Correlation Analysis and Entropy measures.

http://www.leshatton.org/chance_exe.html
http://www.leshatton.org/chance_20070301.html

Chance discovery – King James Bible

- Document is 4mb.
- Relationships between ‘ark’ and ‘covenant’.
- Search took 7 secs

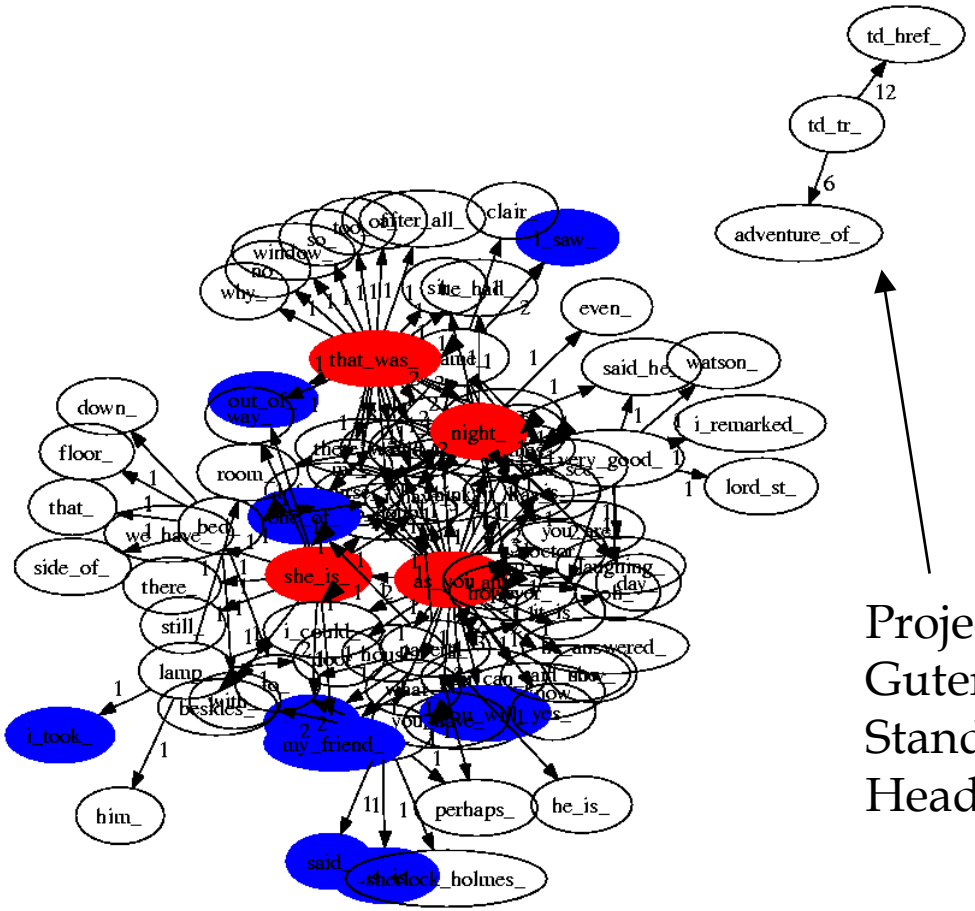
www.gutenberg.org



Chance discovery – The adventures of Sherlock Holmes

- Document is 640K.
- Generic search.
- Search took 12 secs

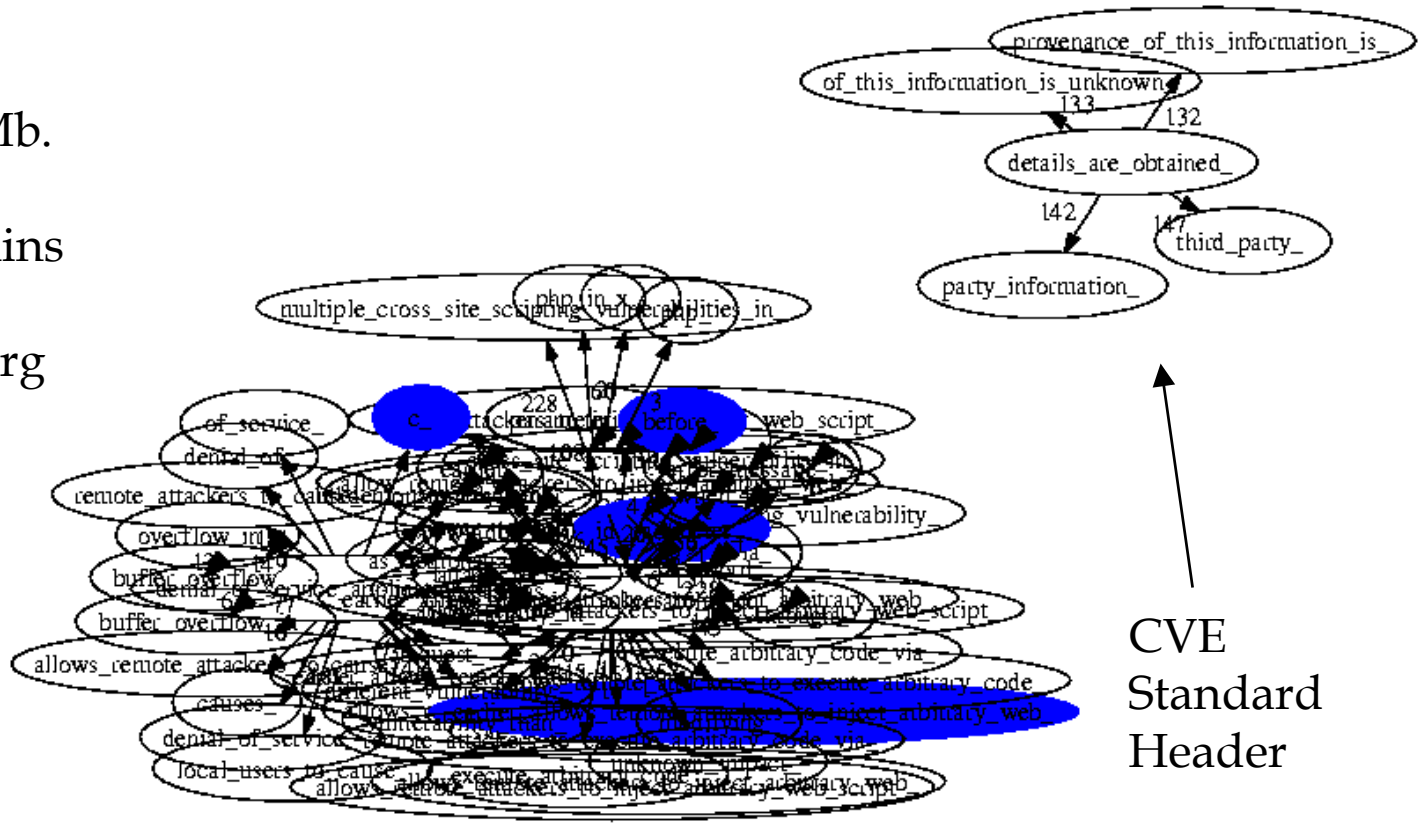
www.gutenberg.org



Chance discovery – the Common Vulnerabilities Database

- Document is 17Mb.
- Generic search.
- Search took 12 mins

www.gutenberg.org



Some failure patterns from the Common Vulnerabilities Database

Test for vulnerability to direct requests.

Design tests to overflow input buffers – this is the most common failure recorded.

Interestingly, a highly related failure involves format string vulnerability, so design tests for these.

Provoking error messages often reveals file paths. Design tests to provoke each error message. This is particularly true for PHP.

Stack-based buffer overflow are often associated with denial of service. Design interface tests to provoke these.

Lessons for Testers

Keep careful defect data but be prepared to mine unstructured data

Analyse it for failure patterns which have statistical significance

Use ONLY statistically significant results to improve your tests and your resource estimation. Be careful ! Differences between individuals are much larger than differences in technologies

Failure patterns contain vital information because they reflect the user's experience

Don't bother testing poorly laid out and non-causal interfaces – report your experiences back to the developer.

For more information and freely downloadable papers see:-

http://www.leshatton.org/, thanks.