Presentation at CERN, 14 Dec, 2011.

“Software Defects, Scientific Computation and the Scientific Method"

Les Hatton

Professor of Forensic Software Engineering
CISM, Kingston University
L.Hatton@kingston.ac.uk

Version 1.1: 14/Dec/2011

Overview

- <u>Popperian deniability and software defects</u>
- Some early thoughts
- A tentative model for defect
- Conclusions

Popperian deniability

- Truth cannot be verified by scientific testing, it can only be falsified.
- Falsification requires quantification of experimental error.
- This has been at the heart of scientific progress.
- This process is NOT generally followed in scientific (or indeed any other kind of) computation.

… defect

- On quantification
  - Computer scientists have researched the average *density* of defect in code extensively
  - Where we have been much less successful is in quantifying the *effects* of such defect on numerical results.

… defect

- On quantification of density
  - A “low defect” piece of software will exhibit less than 1 defect per thousand executable lines of source code in its entire lifetime.
  - Average software is in the range 1-10.

… defect.
A software quality scale based on defect density

Defects/KXLOC

0.1 — **NASA Shuttle software HAL (0.1)**

Linux kernel (0.14)

Several commercial C systems (0.15-0.4)

Commercial Tcl-Tk (0.9)

The best 5% of systems approximately

1.0 — NAG Fortran (2.1)

Medical app C++ (5.1)

Ada comms (7)

NASA Fortran (8)

10.0

Sources Fiedler (1989), Compton (1990), Basili (1996), Hatton (2005,2007,2008)

Overview

- Popperian deniability and software defect
- Some early thoughts
- A tentative model for defect
- Conclusions

Some early thoughts

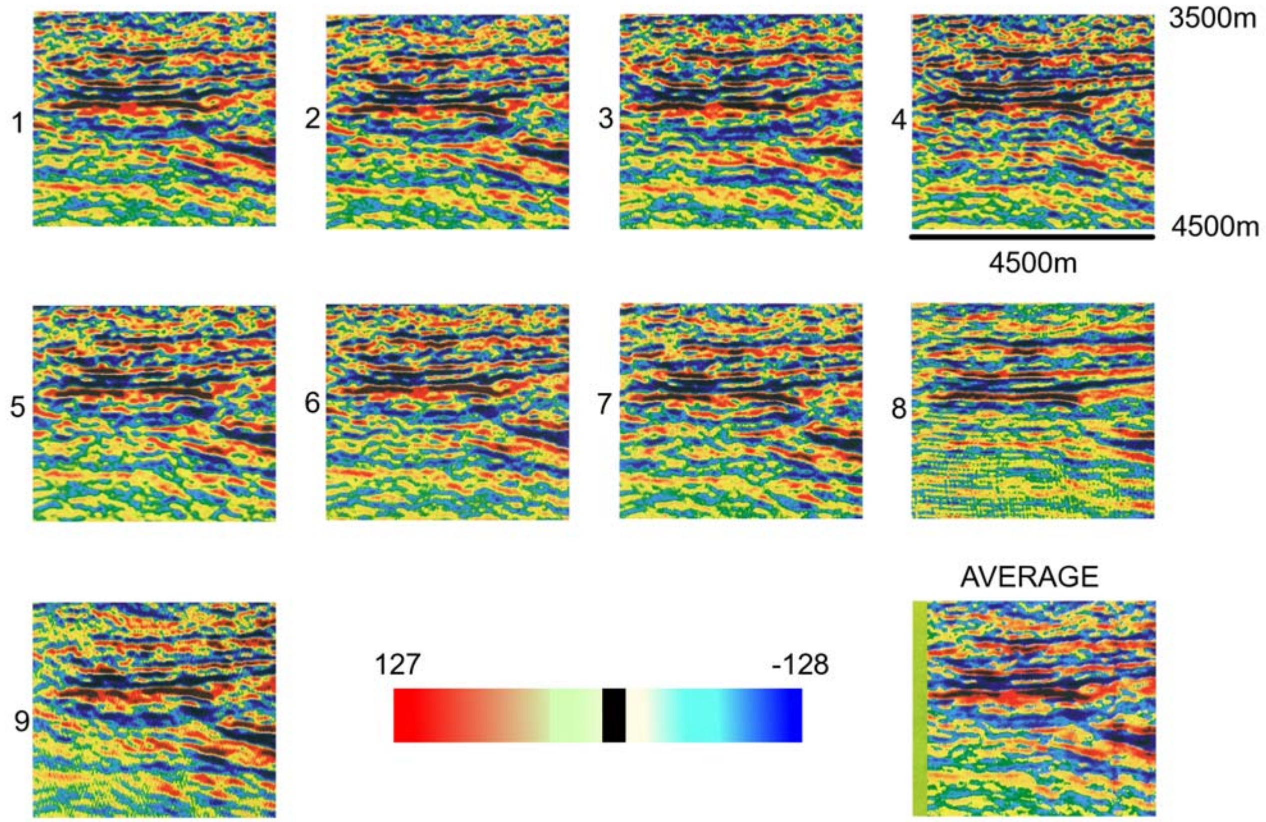
By 2010 I was reasonably convinced that:

- N-version experiments, although not fully independent are exceedingly valuable at highlighting differences, (for whatever reason), and effective at reducing those differences. (1994)
- Scientific software is littered with statically detectable faults which fail with a certain frequency (1997-1998)
- The language does not seem to make much difference. (1999-)
- Defects appear to be fundamentally statistical rather than predictive, (2005-8)
- Software systems exhibit implementation INdependent behaviour (2007-10).

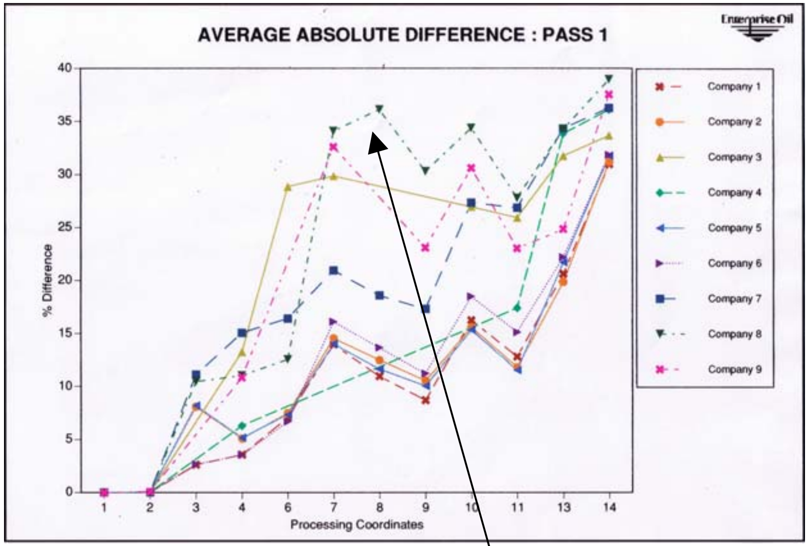
Quantification of differences by N-version (1994)

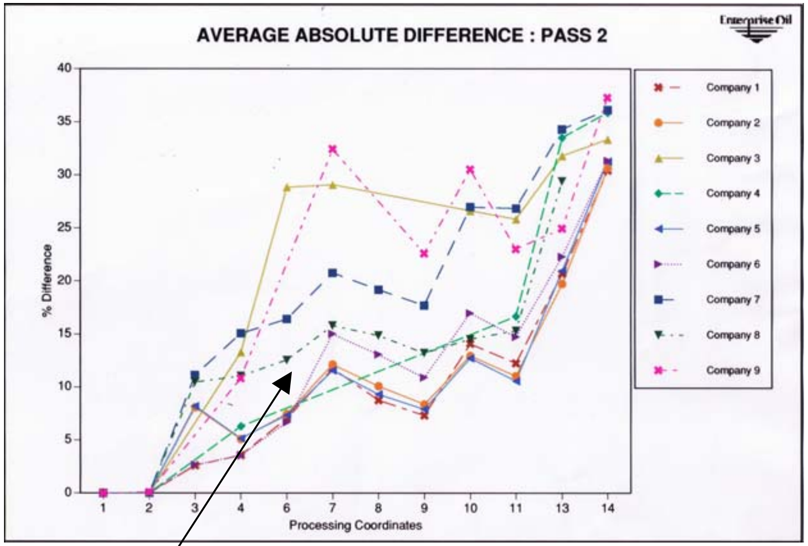
- A comparison of 9 different commercial seismic data processing packages
  - Written to the same semi-formal specifications
  - Written in the same programming language (Fortran 77)
  - Using the same input data tapes
  - Using the same values of disposable parameters
  - Exercised around 200,000 lines of code in each package.

Quantification of differences by N-version (1994)



Convergence using N-version (1994) – but to what ?





Before After

Quantification of differences by N-version (1994)

- Each feedback experiment confirmed …
  - … the existence of a long-standing previously undiscovered defect
  - … its correction led to convergence of the 9 packages. The offending package typically reduced its variance from 40% to 20% whilst the group variance reduced by 16% to 8% over 3 iterations.
- Is this still relevant today ?
  - Language is still in use in various dialects
  - Programmers still use the same test and development processes

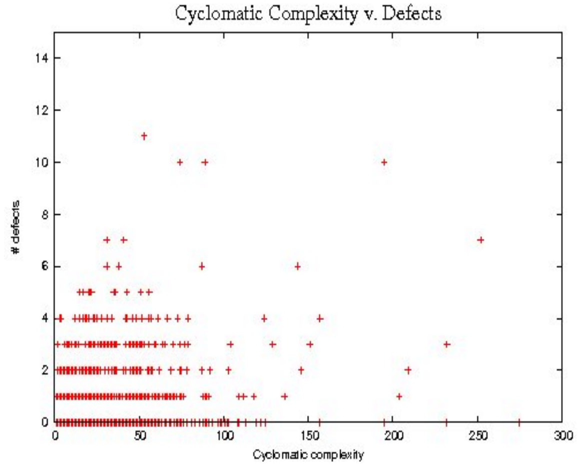
Quantification of differences by N-version

- Note the strong analogue between
  - N independent version run-time comparison
  - Open source facilitating M independent code reviews. (Open source / open data models are crucial in restoring reproducibility to scientific computation).

Are defects related to static complexity ?

- There is little evidence that complexity measures such as the cyclomatic complexity v(G) are of any use at all in predicting defects

Defects

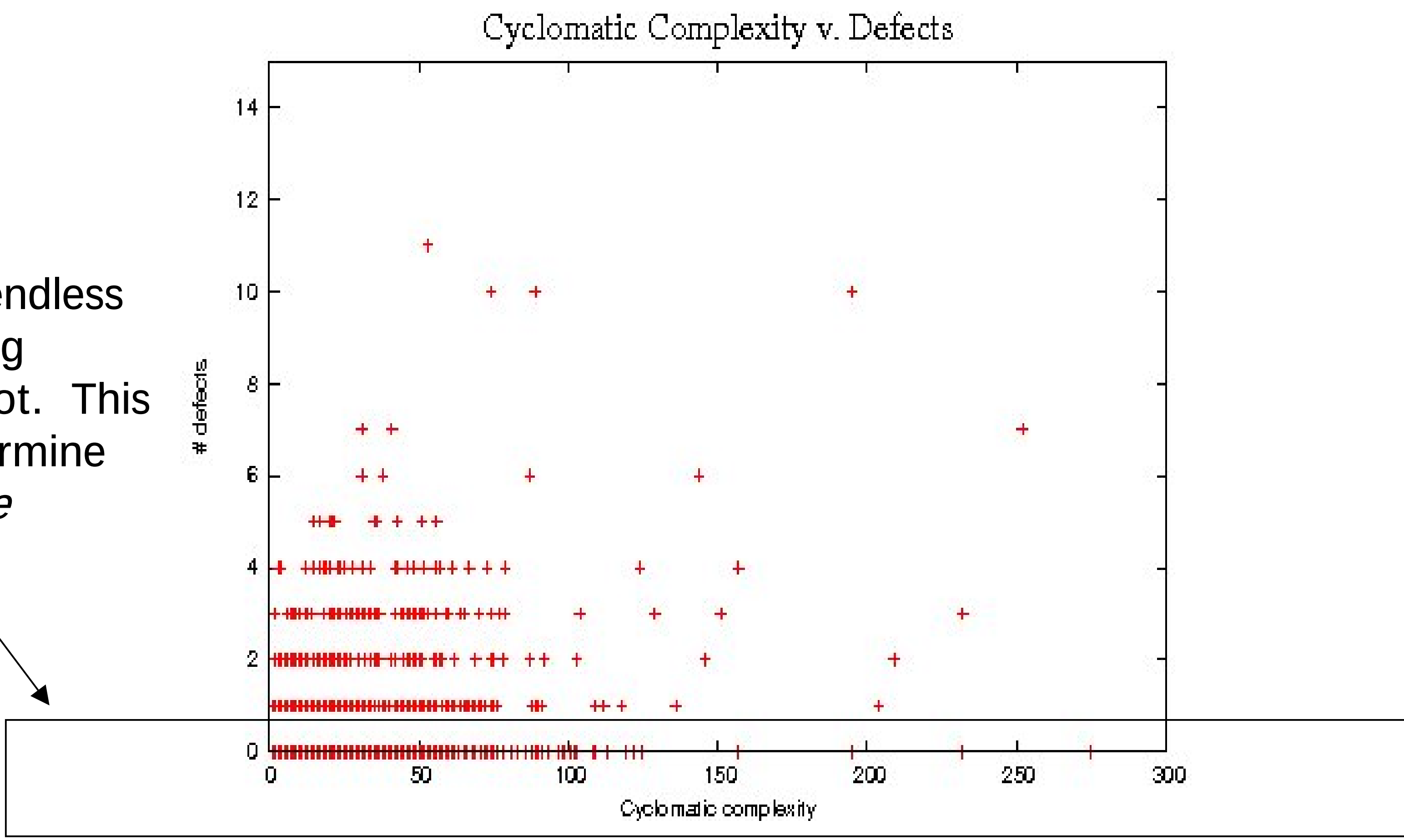


Cyclomatic number v(G)

NAG Fortran library over 25 years
(Hopkins and Hatton (2008))

Is there anything unusual about ‘zero’ defect ?

PCA and endless rummaging suggest not. This may undermine *root-cause analysis*.



… programming language and defect

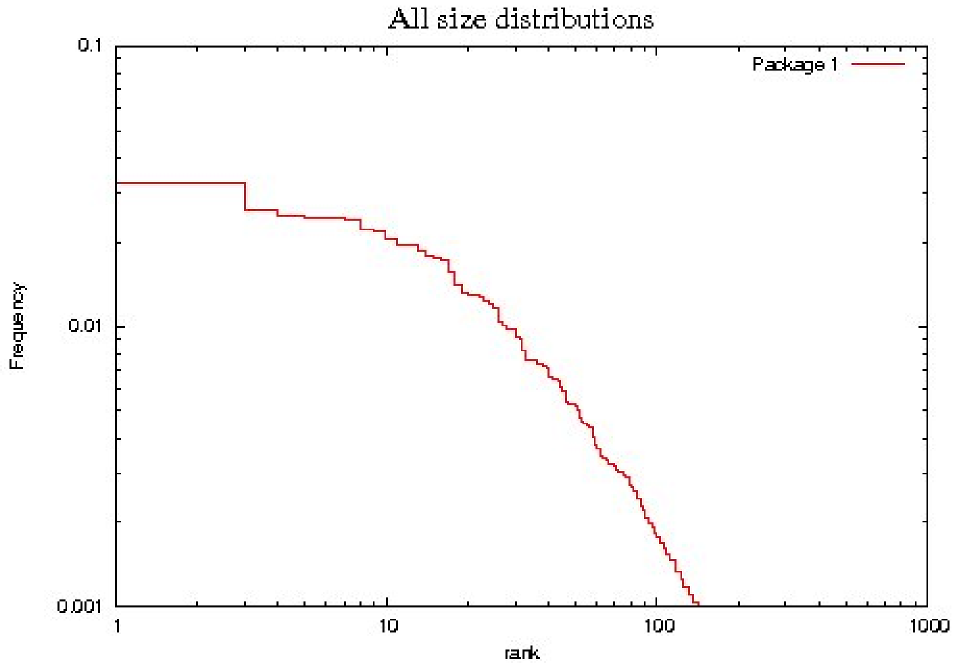
- On static defects
  - Modern programming languages are littered with many types of statically detectable defect, (for example reliance on evaluation order).
  - These typically occur around 5-10 per 1000 lines of executable code and fail at an unacceptably high rate. They must be removed by tools plus inspections.
- Given the undisciplined growth of programming languages, its hardly surprising…

… programming language and bloat

Language	Size 1	Size 2	Increase factor
Ada	270Kb (1983)	1093Kb (1995)	x4
C	191 pp. (1990)	401 pp. (1999)	x2
C++	808 pp. (1999)	1370 pp (2010 draft)	x1.7
Fortran	134 pp. (1978)	354 pp. (1990)	x2.5

Software size distributions appear power-law in LOC

In spite of this, languages are astonishingly similar in their information properties …



Smoothed (cdf) data for 21 systems, C, Tcl/Tk and Fortran, combining 603,559 lines of code distributed across 6,803 components, (Hatton 2009, IEEE TSE)

Overview

- Popperian deniability
- Some early thoughts
- <u>A tentative model for defect</u>
- Conclusions

A tentative model

We are looking for:-

- Language independent behaviour
- Application independent behaviour
- Predicts power-law behaviour in component sizes
- Predicts simple and apparently *power-law* behaviour in defect, (observed frequently)
- Makes other testable predictions.

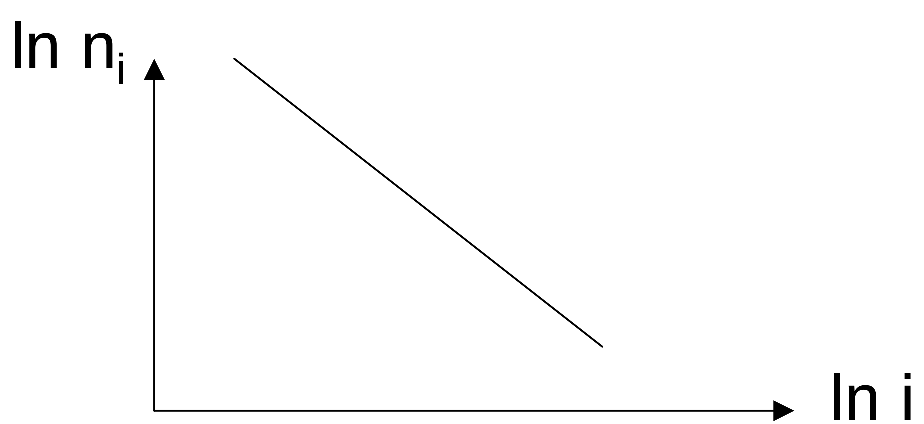
What is power-law behaviour ?

Frequency of occurrence n_i given by $$n_i = \frac{nc}{i^p}$$

This is usually shown as

$$\ln n_i = \ln(nc) - p \ln i$$

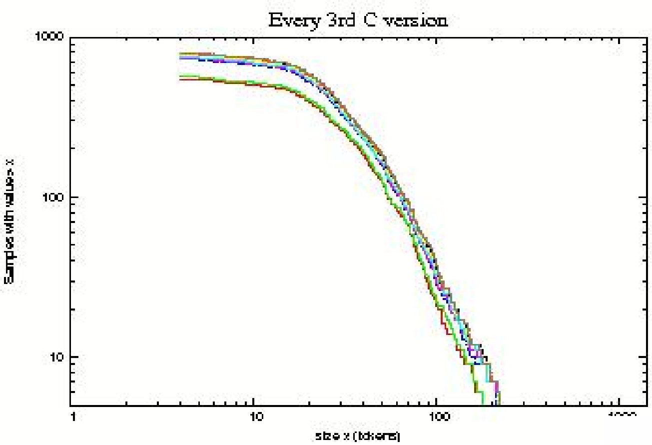
which looks like



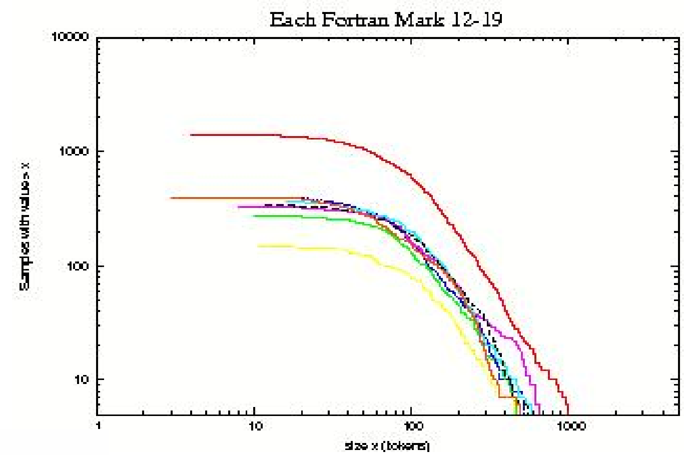
Is power-law behaviour persistent ?

- Question: Does power-law behaviour in component size establish itself over time as a software system matures or is it present at the beginning ?

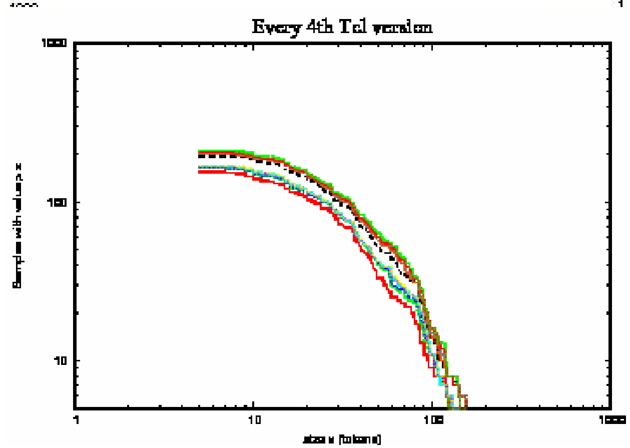
Is power-law behaviour persistent ?



C



Fortran



Tcl

Is power-law behaviour persistent ?

- Answer: *Power-law behaviour in component size appears to be present at the beginning of the software life-cycle.*

- Given that this appears independent of programming language and application area, can we explain why ?

Building systems

- When we build a system we are making choices
  - Choices on functionality
  - Choices on architecture
  - Choices on programming language(s)
- There is a general theory of choice – Shannon information theory.

Building systems

- Software component size - approximate
  - *Number of lines of code*. This is quite dependent on the programming language, (consider the influence of the pre-processor in C and C++ for example).
- Software component size - better
  - Based on *tokens of a programming language*.

Building systems from tiny pieces

- Tokens of language
  - *Fixed tokens*. You have no choice in these. There are 49 operators and 32 keywords in ISO C90. Examples include the following in C, (but also in C++, PHP, Java, Perl …):

    { } [ ] ( ) if while * + *= == // / , ; :
  - *Variable tokens*. You can choose these. Examples include:-

    identifier names, constants, strings
- Every computer program is made up of combinations of these, (note also the Boehm-Jacopini theorem (1966)).

A model for emergent power-law size behaviour using Shannon entropy

Suppose component i in a software system has t_i tokens in all constructed from an alphabet of a_i unique tokens.

First we note that

$$a_i = a_f + a_v(i)$$

Fixed tokens of a language, **{ } [ ] ; while …**

Variable tokens, (id names and constants)

A model for emergent power-law size behaviour using Shannon entropy

An example from C:

Fixed (18)

void int ( ) [ ] { , ;
for = >= -- <=
++ if > -

+

Variable (8)

bubble a N i j t 1 2

```
void bubble( int a[], int N)
{
  int i, j, t;
  for( i = N; i >= 1; i--)
  {
    for( j = 2; j <= i; j++)
    {
      if ( a[j-1] > a[j] )
      {
        t = a[j-1]; a[j-1] = a[j]; a[j] = t;
      }
    }
  }
}
```

Total (94)

A model for emergent power-law size behaviour using Shannon entropy

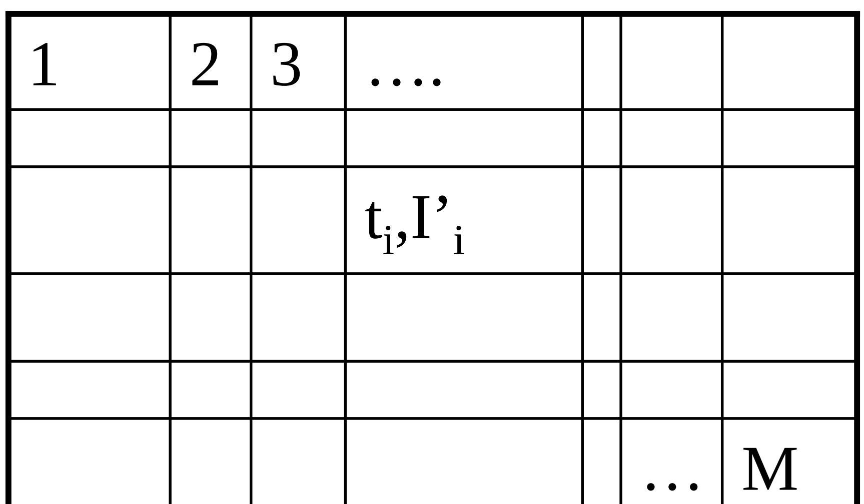
For an alphabet a_i the Hartley-Shannon information content density I'_i per token of component i is defined by

$$t_i I'_i \equiv I_i = \log(a_i a_i \ldots a_i) = \log(a_i^{t_i}) = t_i \log(a_i)$$

We think of I'_i as fixed by the nature of the algorithm we are implementing.

Consider now building a system as follows

Consider a general software system of T tokens divided into M pieces each with t_i tokens, each piece having an *externally imposed information content* density property I'_i associated with it. *Note: no nesting.*



$$T = \sum_{i=1}^{M} t_i$$

$$I = \sum_{i=1}^{M} t_i I'_i$$

General mathematical treatment

The most likely distribution of the I'_i ($= I_i/t_i$)subject to the constraints of T and I held constant

$$T = \sum_{i=1}^{M} t_i \qquad \text{and} \qquad I = \sum_{i=1}^{M} t_i I'_i$$

is

$$p_i \equiv \frac{t_i}{T} = \frac{e^{-\beta I'_i}}{\sum_{i=1}^{M} e^{-\beta I'_i}}$$

where p_i can be considered *the probability of piece i occurring with a share I_i of I*. β is a constant.

General mathematical treatment

However

$$I'_i = \left(\frac{I_i}{t_i} \right) = \left(\frac{t_i}{t_i} \log(a_i) \right) = \log(a_i)$$

Giving the general theorem

$$p_i \sim (a_i)^{-\beta}$$

This states that in any software system, conservation of size and information (i.e. choice) is overwhelmingly likely to produce a power-law alphabet distribution. (Think ergodic here).

One last little bit of maths

- Note that for small components, the fixed token overhead is a much bigger proportion of all tokens, $a_f >> a_v(i)$, so

$$p_i = \frac{1}{Q(\beta)}\left(a_f + a_v(i)\right)^{-\beta} \approx \left(a_f\right)^{-\beta}\left(1 + \frac{a_v(i)}{a_f}\right)^{-\beta} \approx \left(a_f\right)^{-\beta}$$

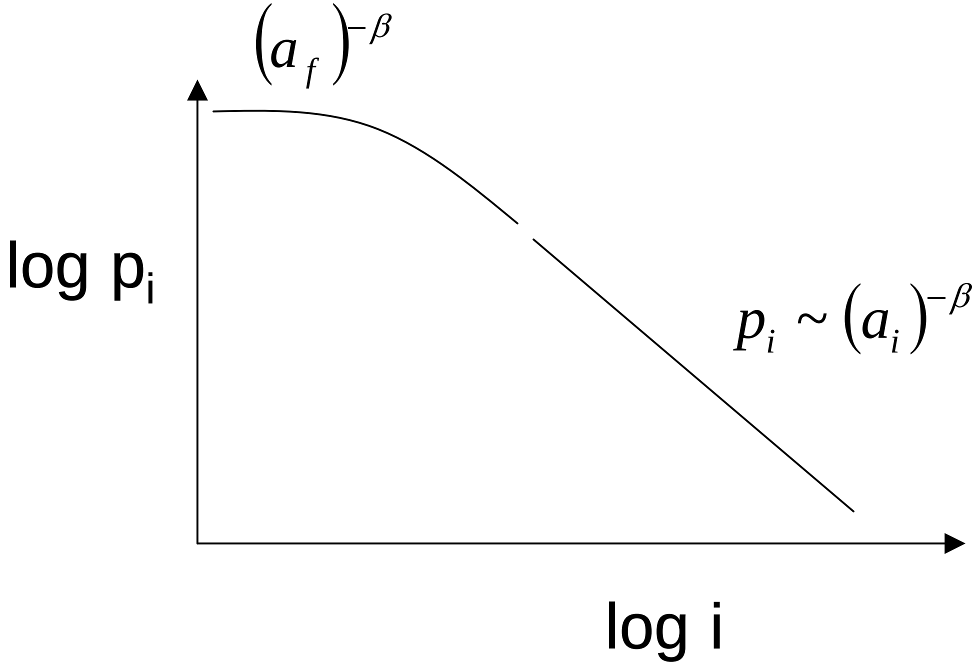
Constant

- For large components, the general rule takes over

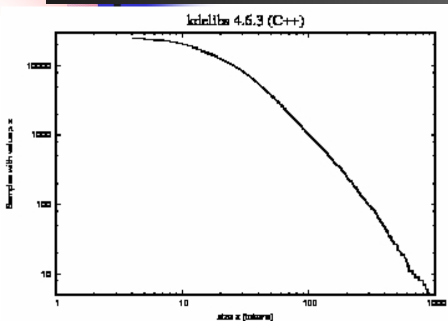
$$p_i \sim \left(a_i\right)^{-\beta}$$

Application to software systems

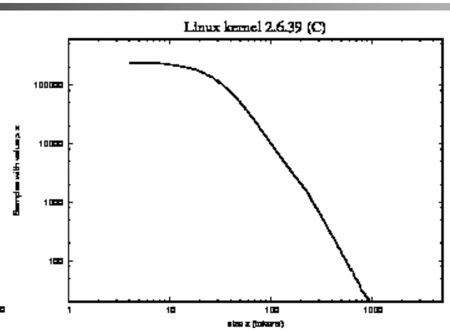
So we are looking for the following signature



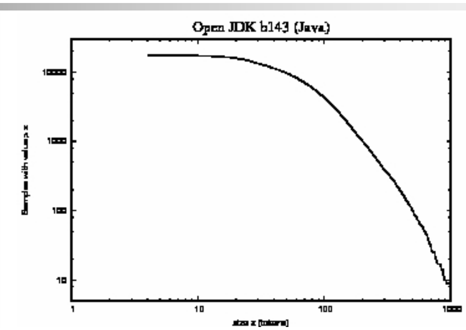
Some results



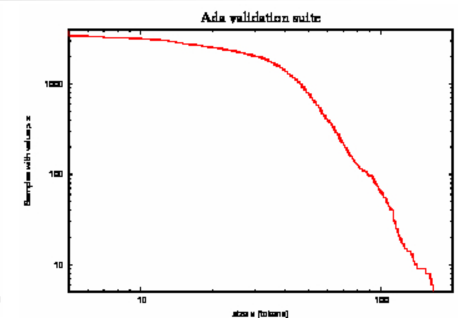
C++



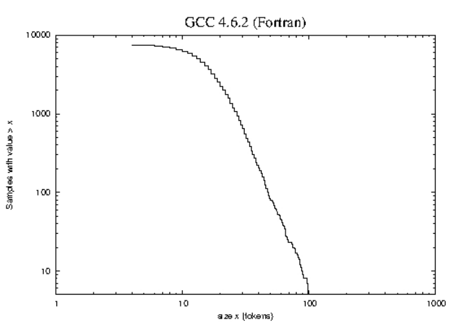
C



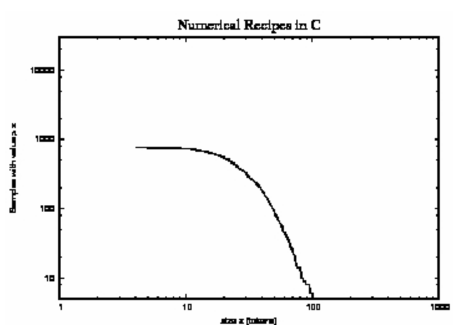
Java



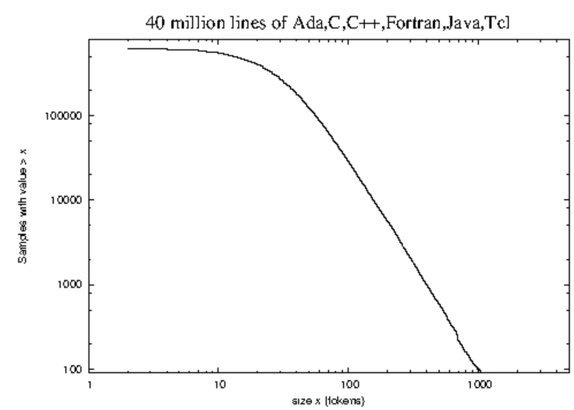
Ada



Fortran



C Numerical



40 million lines of Ada, C, C++, Fortran, Java, Tcl in 78 systems.

Some model predictions

- Suppose there is a constant probability P of making a mistake on any token. The total number of defects is then given by $d_i = P.t_i$ Then

$$p_i = \frac{1}{Q(\beta)}\left(a_i\right)^{-\beta} \approx \left(t_i\right)^{-\beta} \approx \left(d_i\right)^{-\beta}$$

This step uses Zipf's law, Hatton (2009)

- So defects will also be distributed according to a power-law – *i.e they will cluster*.

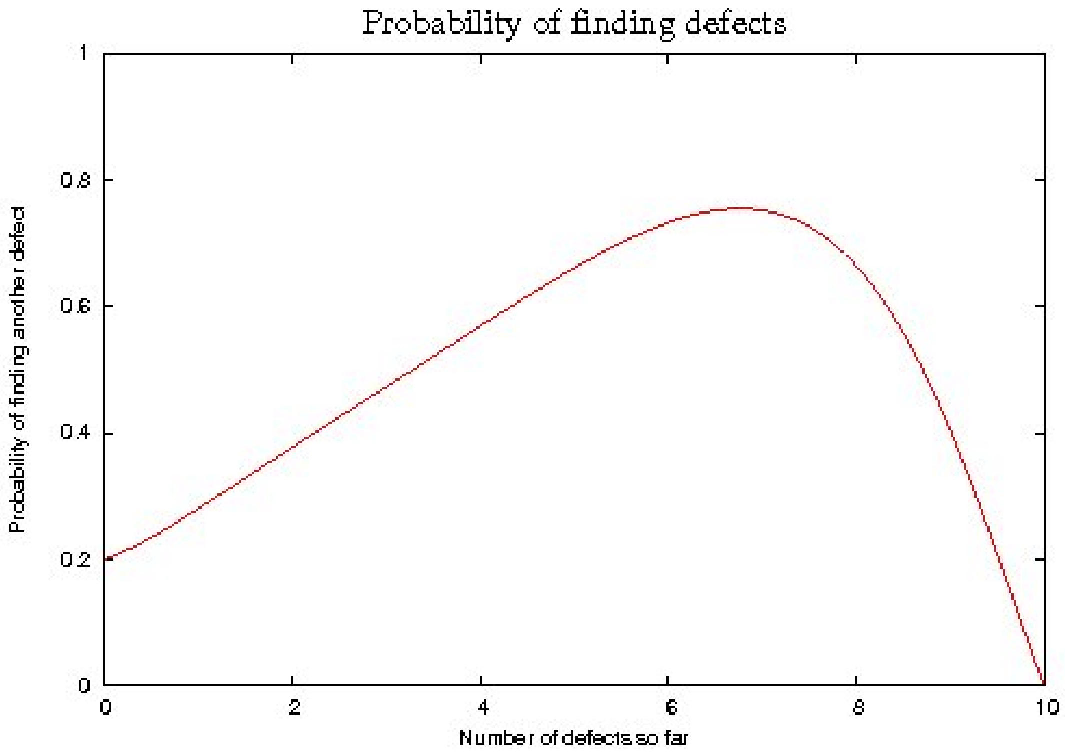
… programming languages Defect clustering in the NAG Fortran library (over 25 years)

A simple model of defects leads to the prediction that defects will cluster

Zero-defect is like winning the lottery. There is no systematic way of achieving it.

Defects	components	XLOC
0	2865	179947
1	530	47669
2	129	14963
3	82	13220
4	31	5084
5	10	1195
6	4	1153
7	3	1025
> 7	5	1867

Clustering can be exploited: Conditional probability of finding defects*



* See, Hopkins and Hatton (2008), http://www.leshatton.org/NAG01_01-08.html

Overview

- Popperian deniability
- Some early thoughts
- A tentative model for defect
- <u>Conclusions</u>

Conclusions

- N versions (or open source) both seem to offer ways of improving software agreement but by an as yet unknown amount.
- Static structural relationships with defect appear to be a blind alley, (cyclomatic complexity …,).
- Defects cluster and this can be exploited.
- Software systems exhibit macroscopic behaviour independent of implementation or language

$$p_i \sim (a_i)^{-\beta}$$

References

My writing site:-

http://www.leshatton.org/

Specifically,

http://www.leshatton.org/variations_2010.html

Thanks for your attention.