Testing the value of checklists in code inspections

Les Hatton
CISM, University of Kingston*

July 4, 2007

Abstract

Checklists form an important part of the fabric of code and design inspections. Ideally, they are intended to improve the efficiency in the sense of number of faults found per inspection hour by highlighting known areas of previous failure. In practice, although the benefit of using them has been previously quantified by some sources, the statistical robustness of the conclusions has not been so commonly represented. In this paper, the effectiveness of checklists is subjected to formal statistical testing using data collected from 308 inspections in workshops for industrial engineers in the last three years. No evidence was found to support the view that checklists made a significant difference in these inspections whether the inspector was experienced or not.

Further analysis revealed that individual inspection performance varied by a factor of 10 in terms of faults found per unit time and individuals found on average about 53% of the faults. Two-person teams found on average 76% of the faults.

1 Overview

Software quality remains an outstanding challenge for software practitioners in the 21st century. Systems still fail all too often and there is considerable public disquiet at the cost of failed projects, [14]. Unfortunately, software development is a very rapidly moving discipline and is often not well-constrained by experiment although much has been learned experientially, particularly with regard to testing software systems.

In the sense that testing is an activity whose primary purpose is to find problems, there are essentially two forms of software testing:-

- Static testing. This covers all test techniques which attempt to discover potential problems by simply examining a design or piece of code. No attempt is made to run the system. The archetypal static test technique is the design or code inspection, [5]. Static test techniques find *faults*.

*L.Hatton@kingston.ac.uk, lesh@leshatton.org

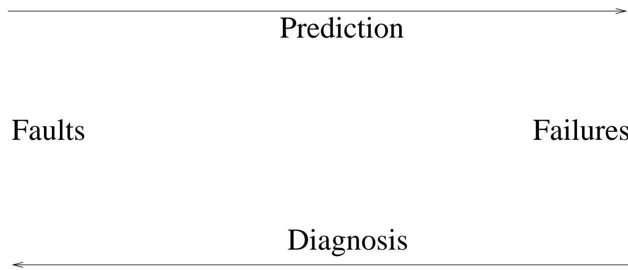

Figure 1: The relationship between diagnosis and prediction in a mature engineering discipline. The clockwise feedback nature of this process involves the study of failures in older systems to discover the underlying faults responsible (diagnosis), eventually followed by the anticipation of failures in new systems from the presence of similar faults (prediction). This process is generally ongoing.

- Dynamic testing. This covers all those test techniques which observe a running system and attempt to discover inputs which lead to unexpected outputs. Dynamic test techniques find *failures*. All failures are caused by at least one fault but not all faults fail. The nomenclature used here is the same as used in [12].

In an ideal world, testing would simply stop when all the faults that have failed (defined to be *defects* here) had been found and corrected. This is generally impractical given commercial exigencies so, in all engineering systems, there is a risk of failure in the delivered product and engineers traditionally deal with this using the concept of "good enough" or with critical systems, ALARP, (As Low As is Reasonably Practical). With conventional systems such as those in civil, mechanical and aeronautical engineering, historical experience is often an excellent guide to the future behaviour of an engineering system based on its past behaviour. This experience is accumulated by a continual feedback process based on the analysis of failure as shown in Figure 1. Over a period of perhaps many years as the engineering discipline matures, failing systems are diagnosed to determine why they failed in order to prevent future failures of the same nature.

Checklists of faults which could lead to failure are a natural output of the diagnostic process and can take many forms as is apparent for example on the excellent NASA Goddard Space Flight Centre website for Software Quality, [4].

The primary object of this paper is to assess how useful checklists are in practice in code inspections using formal statistical inference.

1.1 Code inspections

The benefit of code inspections has been re-iterated many times since the seminal work of [5] and a detailed review can be found at [3]. It is almost certainly true to say that inspections are one of the most successful technologies for the removal of defect ever discovered although it is unclear if all fault modes can be detected in this way. The reader is referred to [6] and [13] for more details and to [15] for a review of some of the inspection approaches which have been proposed.

Inspections have another important property. Because there is generally no knowledge of run-time behaviour when an inspection is carried out, inspections attack the entire fault space irrespective of any temporal properties. In contrast, dynamic testing only attacks that subset which can be provoked to fail in a given run-time. Figure 2 illustrates this point. As time goes by, the subset of faults which have actually failed gradually grows within the set of faults which could fail. It may never fill the entire fault space for various reasons. As was demonstrated so emphatically by [1], a significant percentage of defects, around a third in his case take at least 5,000 execution years to fail for the first time so such defects would be effectively impossible to reveal with dynamic testing. For such defects, static techniques such as code inspections remain the only option. Of course, some systems may never see a total of 5,000 execution years but for a reasonably ubiquitous embedded control system for example, such a total can be exceeded collectively within a few weeks or even days.

Another source of faults which may never fail during the life-cycle of a program is the set of faults which cannot be executed because there is no possible execution path which can provoke their failure. In some complex systems, significant parts can be effectively unreachable so this may not be small.

Finally, inspections also have the benefit of taking place before run-time testing starts. It has been very widely known since the work of [2] that defects found no later than this phase are very substantially cheaper than finding the same defects during run-time testing, (unit, system, acceptance ...).

1.2 Checklists

The essence of the checklist, a prediction mechanism in the sense of Figure 1, is to formalise the process of common fault mode detection and avoidance. Their central role in the practice of inspections is comprehensively discussed by [6] but they naturally involve an overhead. First, data on previous defects (faults which are known to have failed) needs to be acquired and then analysed to identify repetitions before these are codified in checklist form for future developments. It is also possible to construct checklists by interview with experienced engineers in the absence of any such data but the result and any resulting benefit is often harder to quantify. Second, inspectors must continually cross-refer between checklists and the code or design being inspected. Unfortunately, in software-controlled systems, historical experience is usually not available, not documented, or is of questionable relevance because the build techniques have changed too much and prediction mechanisms are correspondingly undermined,

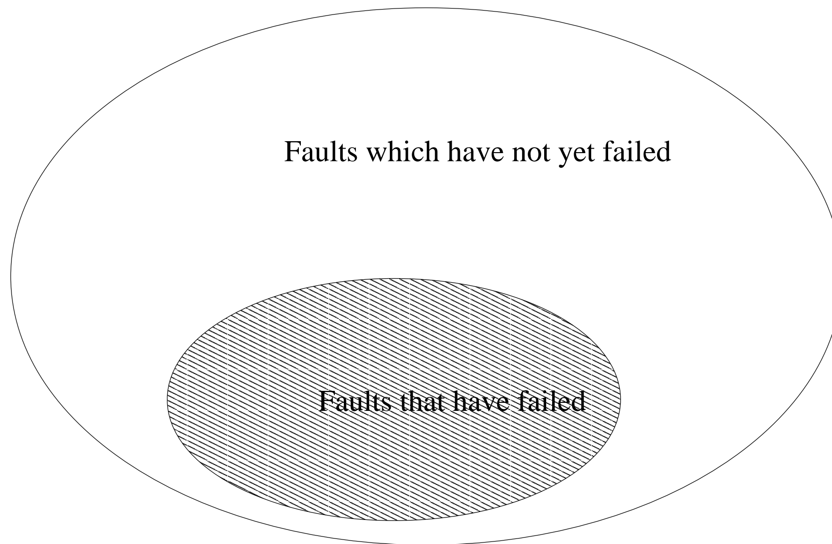

Figure 2: The relationship between faults and defects. Dynamic testing by definition can only attack the faults that can be provoked to fail in the available time shown by the shaded area. In contrast, inspections operate on the entire fault space represented by the larger unshaded area.

(primarily because diagnosis is so poor, [9]). This has many negative implications but this paper will focus on one of them and attempt to assess the benefits of using checklists in code inspections.

2 The experiments

This paper had its roots in an experiment to assess how well the total number of faults in a component could be predicted by using *capture-recapture techniques*, [17]. This was implemented by using a two-phase inspection whereby engineers would inspect a program individually (phase 1, individual preparation) and then, together with another engineer, they would determine which faults were found by both of them (phase 2, simulating a logging meeting), [15]. Together these phases will be known here as a *two-person* team inspection.

Such information can be used as follows. Let P(A) be the probability of a fault being detected by person A in some product. Similarly let P(B) be the probability of a fault being detected by person B in the same product. Then *with the single assumption that A and B are independent*, (which will be tested shortly),

$$P(A \cap B) = P(A)P(B) \tag{1}$$

Further suppose that there are 'n' faults altogether, that person A finds 'a' faults, person B finds 'b' faults and that they find 'q' faults in common. Then, using the above equation,

$$\frac{q}{n} = \frac{a}{n}\frac{b}{n} \tag{2}$$

From this, re-arranging yields:

$$n = \frac{ab}{q} \tag{3}$$

Provided, q $\neq 0$, this provides a simple way of calculating the total number of faults using data on the number of faults found in common between two inspection groups under the assumption of independence described above.

2.1 Experimental design

Some time was taken to select the right program to inspect for the experiment. The principle constraints were engineering time and also the ability to calibrate the experiment in the sense of knowing actually how many faults were in fact present. This latter problem was solved by restricting the scope of the inspection to a known class of faults in the programming language C which can be easily categorised by checklists, [8], [11], often known as code inconsistencies and also easily verified. In addition, it would need enough faults in it to give reasonable results and it would be preferable to have originated from a real system. The chosen program was a C program of 62 lines, originally part of a high-integrity system. It was then slightly modified and reduced to remove all identification, (although all faults described below were unaltered by this process). Although small, this program fitted all the criteria well although.

The program was then analysed both automatically and manually for the chosen set of faults. This allowed the total number of faults to be specified quite precisely as 26 ± 2. The error bound although subjective, was added to mirror the fact that some statically detectable failure prone fragments in this program may have noise associated with them in the manner described by [11]. The presence of so many faults in a real system is somewhat unusual although not unknown. Almost all programs analysed in [8] showed the presence of such faults but at a much lower rate of around 1 every 120 lines or so. This program is atypically but not uniquely very poor but suited the experiment well giving a finer granularity to the results. Engineers were not told how many faults to expect.

Examples of some of the fault modes and occurrence rates are shown below. The remainder had some degree of noise associated with them, for example the recommendation that 'if' and 'else' branches be brace-enclosed.

Fault mode	occurrences
Returning the address of a local from a function	1
Test of unsigned variable for negativity	1
Implicit conversion between int and unsigned int or long	3
Unary negative applied to unsigned object	1
Pointer cast to potentially stricter alignment	1
malloc() called in absence of a prototype	1
Non re-entrancy created by unnecessary internal use of static	1
Unconditionally uninitialised variable	1
enum value missing from switch on enum	1
Non-empty case falls directly through to next case	1
Re-initialisation expression of for loop of floating type	1
Comparison of floating types for equality	1
Memory leak	1
Conditionally uninitialised variable	6

As noted above, these faults fall under the general umbrella of inconsistencies in the use of language. They are easy to categorise in checklists and are often the subject of such lists in real inspections, [10]. Such faults can in principle be detected automatically by tools although tool deployment is often rather variable between development groups. In this case, whether or not the faults could be detected automatically does not detract from the essence of an inspection and it allowed independent verification of the faults. It should be noted that in practice, it is generally considered wise to automate such fault detection as is possible in an organisation, leaving the remainder for human eyes.

These faults were gathered together into the following checklist categories:-

- Dataflow, (i.e. initialisation and use of objects)
- Static faults (Fundamental misunderstandings in the use of C.)
- Interface disorders
- Undefined behaviour in the language
- Function use unprotected by prototypes
- Potentially dangerous behaviour
- Masked declarations

In the case of code inconsistencies, it is generally not feasible to present a full list of all fault modes as there are several hundred of them in the case of the programming language C and presentation as categories is common in checklists presented in programming standards, [10]. The checklist was therefore presented to the engineers in this form with a pre-amble. Those engineers using the checklist were told that all the faults in the program fell into these categories with the intention of promoting a structured and profitable way of inspecting the program for faults:- dataflow, interfaces, and so on. All the engineers were familiar with code inconsistencies in the programming language

C but those engineers not using the checklist were given no other information other than that the code to be inspected contained an unspecified number of such inconsistencies. The overall distribution of industrial engineering experience in years was not measured in detail but was in the range 2-20 years.

2.2 Experimental procedure

The original idea was to test capture-recapture techniques and their potential ability to predict the total number of faults in a program. Such techniques depend for their success on independence as described above so the experiment was set up in two parts as follows.

- Part 1 was set up to test independence. Checklists were introduced on a random basis, (the developer could choose whether to use them or not), to mirror common practice in industry as their use is not always mandated, [10].
- Part 2 was then intended to test the ability to predict the total number of faults assuming part 1 supported independence as a reasonable assumption. (It was expected that the assumption of independence might well be prejudiced by the use of checklists by some developers.)

2.2.1 Part 1: 2003-2005

Initially, groups of industrial embedded system engineers in three separate countries, from Germany (4x2 person teams in one location and 4x2 person teams in another), Austria (3x2 person teams) and from India (11x2 person teams) over a 2 year period, were allowed to inspect the described program individually for a total of 30 minutes, corresponding to an inspection speed of approximately 120 lines of code per hour. This lies within the most efficient range quoted by [13], although is a little fast according to [6]. Following this, individual engineers were then combined in teams of two to compare notes for a further 15 minutes to determine which faults they had independently found in common. *The choice as to whether to use a checklist or not was left up to the individual engineer and was not recorded.* The results were then collected and are shown below.

Faults found by Inspector 1	Faults found by Inspector 2	Faults in common
7	9	3
12	13	3
14	12	4
10	7	4
10	11	6
14	10	5
8	9	3
6	6	3
9	6	3
15	10	4
10	13	3
15	17	11
25	20	17
19	15	9
14	13	8
11	18	9
5	17	5
17	9	6
13	8	2
15	10	6
17	8	5
17	10	6

Table 1: Inspection results for part 1

Testing the assumption of independence It was originally expected that the use of checklists by some of the participants would compromise the notion of independence. If two engineers are using the same checklist individually, it would be expected that this would increase the number of faults found in common. However the assumption of independence was essential to the capture-recapture part of the project originally planned, consequently it was tested statistically.

Two events A and B, are independent if and only if $P(A \cap B) = P(A)P(B)$ and so the statistic $P(A \cap B) - P(A)P(B)$ was tested using the Kolmogorov-Smirnov test for normality. The result is that this statistic is consistent with a normal distribution $N(\mu = 0.0183, \sigma = 0.07433)$ with a power p = 0.82. The equivalent 95% confidence interval for the mean is (-0.0084,0.0484) which includes 0.0 so *the hypothesis that $P(A \cap B) - P(A)P(B)$ is normally distributed with a mean of zero cannot be rejected, and therefore neither can the hypothesis of independence be rejected.*

This inconclusive result was surprising. The use of checklists by at least some of the engineers *even anonymously* would be expected to guarantee that the independence assumption would be questionable at best because a percentage of the population would be individually driven preferentially towards well-known

fault modes. Furthermore, some of these fault modes were present in the inspected program. As a result, part 2 of the experiment was re-designed to explore this observation further and to test the effectiveness of checklists in these inspections by deliberately selecting specific engineers to use checklists.

2.2.2 Part 2: 2006-7

In this part engineers in Germany and Sweden (119x2 person teams altogether) over an eight month period, were allowed to inspect the described program under the same conditions as before except that *specified engineers were selected to use checklists*. The selection was done so that at the comparison stage where pairs of engineers compared what they had found in common, *either both or neither* had used checklists to maximise the possibility of seeing a positive relationship between the use of checklists and the number of faults found. Selected engineers used the checklist side by side with the code under inspection. The other engineers did not see the checklist. Finally the selection was made such that no engineer using a checklist sat next to an engineer not using a checklist.

The results were again collected but because of space limitations are only summarised below. In the interests of both pedagogy and repeatable science, the full anonymised raw data for this experiment are freely available for download and analysis [1] in the form of a zipped Excel spreadsheet.

Category	Mean	Std. dev.	Number
Entire population	13.66	5.12	238
Individual faults found using checklists	13.97	5.27	106
Individual faults found not using checklists	13.40	5.00	132
Common faults found using checklists	7.87	3.69	53
Common faults found not using checklists	7.41	4.06	66

Table 2: Inspection results for part 2. Recall that when teams measured common faults, either both were using checklists or neither.

These data can now be tested directly. The following hypotheses will therefore be made:-

- H_0 The null hypothesis, the number of faults found does *not* depend on the use of checklists.
- H_1 The alternative hypothesis, the number of faults found does depend on the use of checklists.

To infer that the number of faults does indeed depend on the use of checklists, the data must reject the null hypothesis at some standard level of significance. The data will be analysed using the z-test for the difference of means in a population, [18]. This states that the following statistic is approximately distributed

[1] http://www.leshatton.org/Data_Inspections_05-06-2007.html

as N(0,1) provided the numbers in the sample are considered large as is the case here,

$$z = \frac{\overline{X_1} - \overline{X_2}}{(\frac{(s_1)^2}{N_1} + \frac{(s_2)^2}{N_2})^{\frac{1}{2}}} \tag{4}$$

where $\overline{X_i}, s_i$ and N_i are respectively the sample means, standard distributions and number of samples for the data using checklists and the data not using checklists respectively. Any significant difference in the means can be taken to imply the presence of a real effect at some level of confidence. Substituting the numbers from the table above yields,

$$z = \frac{13.97 - 13.40}{(\frac{(5.27)^2}{106} + \frac{(5.00)^2}{132})^{\frac{1}{2}}} \simeq 0.055 \tag{5}$$

In order to reject the null hypothesis and infer a significant difference between the checklist and non-checklist populations at the commonly used 5% level, $|z|$ must be > 1.9. This result is not even significant at the 10% level so there is no basis for rejecting H_0 and it must be concluded that *there is no basis to reject that the number of faults found does not depend on the use of checklists.* It is useful to repeat the same calculation for the number of faults found in common. As discussed above, it might be expected that any effect would be amplified if both inspectors were using checklists. Repeating the calculation for the faults found in common gives:-

$$z = \frac{7.87 - 7.41}{(\frac{(3.69)^2}{53} + \frac{(4.06)^2}{66})^{\frac{1}{2}}} \simeq 0.063 \tag{6}$$

Again this is not significant at any standard level, so there is still no evidence to reject the hypothesis that the number of faults does not depend on the use of checklists.

Discussion This negative result is again troubling. In the first part of this experiment when the choice of whether to use checklists or not was left up to the engineer and not recorded, no significant pattern challenging the notion of independence emerged from the data even though it might be expected that the presence of at least some engineers using checklists would prejudice independence. In the second part with a different set of engineers in which the choice to use a checklist was assigned to the engineers in a way designed to emphasise the presence of any biasing effect, there is still no significant pattern.

If there is any benefit to be derived from doing checklists, it does not present itself at any statistically significant level in either of the two parts of the experiment described here. In spite of this, it is known from the previous references that inspections are very successful at finding defect so the hypothesis will be advanced here that *when a human inspects a program, a checklist is too simplistic to describe the mental processes involved.* It is also possible that engineers simply ignore them even when requested to use them but this seems unlikely given the numbers involved in the experiment.

There is of course an underlying assumption made here and that is that checklists for code consistency are representative of checklists as a whole. Given the nature of checklists, this does not seem an unreasonable assumption to make and the experiment described here does mirror common industrial practice as described in [10] but further experiment would be necessary to explore this in detail.

2.2.3 Effects of experience

The size of the dataset allowed statistical tests to be made to see if there was any relationship with experience of the developer. For the purposes of this analysis, an experienced or inexperienced developer will be defined as being outside one standard deviation of the entire population mean in Table 2 on the appropriate side of the mean. In other words, experience will be defined as finding at least 16 faults and inexperience will be defined as finding at most 11. Although somewhat arbitrary, this has the advantage of being related in conventional statistical terms to the population as a whole. Analysing these two populations separately gives the following results:

Category	Mean	Std. dev.	Number
Experienced population using checklists	19.60	4.02	35
Experienced population not using checklists	19.56	3.47	35
Inexperienced population using checklists	8.22	2.24	32
Inexperienced population not using checklists	8.81	1.92	52

Table 3: Inspection results for experienced and inexperienced populations. Here *experience* is defined by those who found at least 16 faults and *inexperience* is defined by those who found at most 11 faults.

For the experienced population:-

$$z = \frac{19.60 - 19.56}{(\frac{(4.02)^2}{35} + \frac{(3.47)^2}{35})^{\frac{1}{2}}} \simeq 0.006 \quad (7)$$

For the inexperienced population:-

$$z = \frac{8.81 - 8.22}{(\frac{(1.92)^2}{52} + \frac{(2.24)^2}{32})^{\frac{1}{2}}} \simeq 0.104 \quad (8)$$

Again, neither result is significant at any standard level, so the experiment cannot distinguish any significant difference due to the use of checklists based on whether the population is experienced or not in the sense defined above. The individual experience in terms of years was not recorded for the engineers so no relationship could be drawn between temporal experience and the number of faults found in this experiment.

2.2.4 Variation between engineers

Finally, it has often been noted, [7] and [16] amongst others, that there are wide variations (i.e. a factor of 10 or more) in the performance of software developers in various skills. This dataset affords the possibility of quantifying this for the particular case of fault detection in inspections. The mean and standard deviation for the dataset as a whole were:-

- Mean = 13.66
- Standard deviation = 5.12

This gives a rather wide 95% confidence interval of $13.66 \pm 1.96 \ . \ 5.12 = [3.62, 23.70]$ with the best member of the population about a factor of 10 better than the worst, a very similar result to those of [7] and [16] but in a rather different category.

2.2.5 2 person teams versus individuals

Finally, the data also allows an estimate of how much better a two person team performs than an individual. Here a two-person team means two individual inspections followed by a meeting to discover faults found in common. The average number of faults found was 13.66 (53%) by individuals and 19.71 (76%) by a two-person team.

3 Conclusions

A formal statistical analysis of 308 individual inspections for code inconsistencies some of which were controlled by checklists and others not reveals in the slightly awkward parlance of statistical inference, that there was no evidence to reject the hypothesis that checklists have no effect on the number of such faults found when inspections are carried out at recommended rates.

Furthermore, when the population was split into inexperienced and experienced engineers in terms of number of faults found, there was still no statistically significant relationship between the use of checklists and the number of faults found in either population. The null hypothesis again could not be rejected even though it might have been expected that inexperienced programmers would benefit more from checklists than experienced programmers. In the language of statistical inference, this does not of course prove that checklists have no effect but the result is sufficiently inconclusive that the role of checklists in inspections should be investigated further.

It may well be that checklists are critically dependent on the way they are worded and/or on the way they are enforced and that the checklist used here is deficient in some regard. In its defence, this checklist followed a model often used in industry and such sensitivity alone would be cause for some concern. However, the simplest explanation of the inconclusive result reported here seems to be that reasonably experienced engineers appear to use personalised 'internal' methods whether given checklists explicitly or not. It may even be that these internal methods take the form of implicit checklists but this is a matter for further

investigation. It should also be re-iterated that the checklists considered here refer only to one particular kind of fault, viz. code inconsistency and the piece of code under inspection was relatively short. Whether these results extend to the many other kinds of inspection reported in the software engineering literature is also a matter for further formal investigation.

Finally this dataset also revealed that there is quite a high variation in individual inspector's performances with the worst a factor of 10 or so less effective than the best. Such a factor has emerged in other studies in areas as disparate as productivity [7], and performance [16]. In the study described by [7], the group under study was restricted to experienced programmers only and the broad variation still existed. This continuing wide disparity between the best performing and the worst performing in any programming group when measured in very different ways remains a challenging obstacle to progress in the consistent production of reliable systems within a fixed time and a fixed budget.

References

[1] E. Adams. Optimising preventive service of software products. *IBM Journal of Research and Development*, 1(28):2–14, 1984.

[2] B. Boehm. *Software Engineering Economics*. Prentice-Hall, NJ, 1981.

[3] Bill Brykczynski. A survey of software inspection checklists. *ACM Sigsoft, Software Engineering Notes*, 24(1):82–89, 1999.

[4] NASA Goddard Space Flight Center. Software quality practices website. *http://sw-assurance.gsfc.nasa.gov/disciplines/quality/*, 2007-.

[5] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 2:182–211, 1976.

[6] T. Gilb and D. Graham. *Software Inspections*. Addison-Wesley, 1993. ISBN 0-201-63181-4.

[7] E.E. Grant H. Sackman, W.J. Erikson. Exploratory experimental studies comparing online and offline programming performance. *CACM*, 11(1):3–11, 1968.

[8] L. Hatton. *Safer C: Developing software in high-integrity and safety-critical systems*. McGraw-Hill, 1995. ISBN 0-07-707640-0.

[9] L. Hatton. Characterising the diagnosis of software failure. *IEEE Software*, 18(4):34–39, July 2001.

[10] L. Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology*, 46:465–472, 2004.

[11] L. Hatton. EC– a measurement based safer subset of ISO C suitable for embedded system development. *Information and Software Technology*, 47:181–187, 2005.

[12] L. Hatton. Some empirical test observations in client/server systems. *IEEE Computer*, 40(5):24–29, May 2007.

[13] W. Humphrey. *A discipline of software engineering*. Addison-Wesley, 1995. ISBN 0-201-54610-8.

[14] Royal Academy of Engineering. The challenge of complex it projects, 2004. Royal Academy of Engineering report, London, ISBN 1-903496-15-2.

[15] S.L. Pfleeger, L. Hatton, and C. Howell. *Solid Software*. Prentice-Hall, 2002. ISBN 0-13-091298-0.

[16] L. Prechelt. Comparing java vs. c/c++, efficiency differences to interpersonal differences. *CACM*, pages 109–112, October 1999.

[17] Z.E. Schnabel. The estimation of the total fish population in a lake. *Amer. Math. Mon.*, 45:348–352, 1938.

[18] M.R. Spiegel and L.J. Stephens. *Statistics*. Schaum. McGraw-Hill, 3rd edition, 1999.