The *METRICS NEWS* can be ordered directly from the Editorial Office (for address see below).

Editors:

Reiner Dumke
Professor on Software Engineering,
University of Magdeburg, FIN/IVS,
Postfach 4120, D-39016 Magdeburg, Germany
Tel.: +49-391-67-18664, Fax: +49-67-12810
email: dumke@ivs.cs.uni-magdeburg.de

Christof Ebert
Dr.-Ing. in Computer Science
Alcatel Telecom, Switching Systems Division,
Fr. Wellensplein 1, B-2018 Antwerpen, Belgium
Tel.: +32-3-240-4081, Fax: 32-3-240-9935
email: christof.ebert@alcatel.be

Eberhard Rudolph
Professor on Software Engineering
Hochschule Bremerhaven, FB2 System Analysis,
Mozartstr. 37, D-27570 Bremerhaven, Germany
Tel.: +49-471-26142, Fax: +49-471-207389
email: rudolph@oscar-e.hs-bremerhaven.de

Horst Zuse
Dr.-Ing. in Computer Science
Technical University of Berlin, FR 5-3,
Franklinstr. 28/29, D-10587 Berlin, Germany
Tel.: +49-30-314-73439, Fax: +49-30-314-21103
email: zuse@tubvm.cs.tu-berlin.de

Editorial Office: Otto-von-Guericke-University of Magdeburg, FIN/IVS, Postfach 4120, 39016 Magdeburg, Germany
Technical Editor: DI Erik Foltin
The journal is published in one volume per year consisting of two numbers.

EDITORIAL

This is the third issue of a new scientific journal in the field of software metrics and related quantitative aspects, the

METRICS NEWS.

The title was chosen to reflect the Journals attempt to summarize recent software metrics trends as position papers, chosen papers from our metrics workhops, and *news* (as information about the software metrics research area in the world, new books and conferences). The editors are working many years in the software metrics field and are specialized in measurement frameworks, function point analysis, measurement theoretical view, and practical applications.

The background of the METRICS NEWS contributors is the GI-interest group on software metrics founded in 1991. All members from the industry or academia are invited to present their experience or research results in the area of software quality assurance, software metrics, process management, software measurement frameworks etc.

The English language was chosen to reflect the international character of our research contacts and results embedded in European initiatives.

The editors are grateful to the Otto-von-Guericke University of Magdeburg for publishing this journal.

We hope that the new journal will be helpful to increase the awareness of the importance of software metrics issues in the improvement of software development processes and products.

The Editors

8th International Workshop on Software Measurement

of the German Interest Group on Software Metrics and the
Canadian Interest Group on Metrics (C.I.M.)
***September 17 and 18, 1998**, Magdeburg, Germany*

Scope

Software measurement is one of the key technologies to control or to manage the software development process. The applicability of metrics, the efficiency of metrics programs in industry and the theoretical foundations have been subject of recent research to evaluate and improve modern software development areas such as object-orientation, component-based development, multimedia system design, reliable telecom-munication systems etc. Our recent workshops have been attentive to these concerns. Research initiatives were directed initially to the validation of software metrics and their practical use based on critical analysis of the benefits and weaknesses of software measurement programs. But up to now the application of metrics does still involve the risk of failure. Therefore, it is necessary to exchange the experience of successful metrics applications between practitioners and researchers to stimulate further theoretical investigations to improve the engineering foundations in software development and measurement. We are looking for papers in the area of software metrics and software measurement from (but not limited to) the following areas:

- Lessons learned from establishing a measurement program in industry (could be successes or failures with post mortems)
- Experience reports
- Theoretical background and further practical applications of the function point method
- Metrication of new OO languages or methods such as Java and UML
- Metrics data bases and repositories
- Development and use of measurement tools
- Theory of measurement and its practical implications

Program Committee

Alain Abran, University of Quebec, Montreal, Canada
Katrin Baumann, SAP, Walldorf, Germany
Reiner Dumke, University of Magdeburg, Germany
Christof Ebert, Alcatel, Antwerp, Belgium
Horst Zuse, TU Berlin, Germany

Submissions

4 Position Papers

Authors should send full papers (max. 12 pages) or abstracts (2-3 pages) by mail, fax or e-mail by **July 15, 1998** to

Alain Abran
University of Quebec
Dept. of Computer Science
C.P. 8888, Succ. Centre-Ville
Montreal (Quebec), Canada H3C 3P8
Tel.: +1-514-987-3000
Fax: +1-514-987-8477
e-mail: abran.alain@uqam.ca

or to

Reiner Dumke
Otto-von-Guericke-Universität Magdeburg
Institut für Verteilte Systeme
Postfach 4120
D-39016 Magdeburg, Germany
Tel.: +49-391-67-18664
Fax: +49-391-67-12810
e-mail: dumke@ivs.cs.uni-magdeburg.de

Workshop Timetable

Submission deadline: July 15, 1998
Notification of acceptance: August 15, 1998
Full paper for publishing in a metrics book: workshop date
Workshop date: September 17-18, 1998

News

For the latest news about the Workshop please see the following Web site:

http://ivs.cs.uni-magdeburg.de/sw-eng/us/

It is planned to broadcast the conference world-wide by videoconferencing based on Mbone in the Internet.

MEASUREMENT IN PHYSICS AND SOFTWARE ENGINEERING

Part II

Horst Zuse
University of Southwestern Louisiana[1]
2 Rex Street
P.O. Box 44330
Lafayette, LA 70504-4330, USA

[1] Horst Zuse is from February 20, 1998 till May 20, 1998 with the University of Southwestern Louisiana, Lafayette, Louisiana, as a Visiting Professor.

4 Additive Measures

In Part I we finished with the question whether measures in the software measurement area assume additivity. Let P1, P2 are flowgraphs with P1, P2 ∈ **P**, where **P** is a set of flowgraphs, then for many measures holds

$$u(P1 \circ P2) = u(P1) + u(P2).$$

We denote with u a measure, for example a complexity measure. The statement P1 o P2 is the sequential concatenation of two Flowgraphs P1 and P2 to the sequence P1 o P2. u(P1 o P2) means the application of the Measure u to the sequence of the Flowgraphs P1 o P2. For example, the Measure LOC assumes such a property. It holds for LOC

$$LOC(P1 \circ P2) = LOC(P1) + LOC(P2).$$

This is similar to the additive behavior of masses, lengths or resistors, etc.. We illustrate this with the next picture.

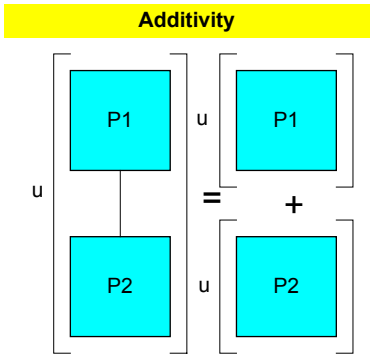


Figure 4.1: Additive Measure LOC. If P1 and P2 are arbitrary flowgraphs then it holds LOC(P1 o P2) = LOC(P1) + LOC(P2).

In physics such an additive behavior can be found in many cases. For example, masses can be combined and the formula of the weights of the two masses is additive. The next picture illustrates this.

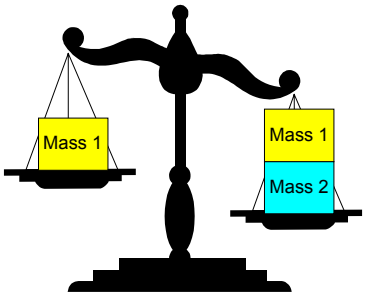


Figure 4.2: The weights of masses are additive.

However, in physics we also can find additive and non-additive measures. An example is lengths measurement. The next picture illustrates this.

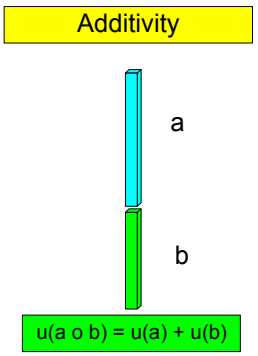


Figure 4.3: Lengths measurement is additive if we add two boards to a sequence.

If u is a ruler then it holds:

$$u(a \circ b) = u(a) + u(b).$$

This is a well known property of lengths measurement, but it also holds non-additivity for lengths measurement. Let us consider the law of Pythagoras.

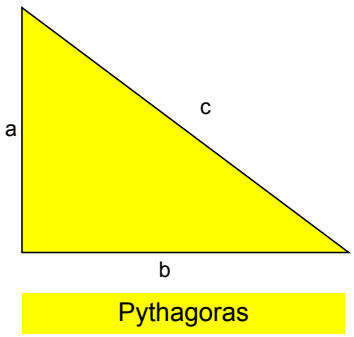


Figure 4.4: The Law of Pythagoras.

Considering the law of Pythagoras we do not have additivity. The length of c can be determined by the lengths of a and b. It holds

$$c^2 = a^2 + b^2.$$

or

$$c = (a^2 + b^2)^{½}.$$

In the software measurement area we also have non-additive measures, like the Measure

$$OV = |E|,$$

where E is the set of edges in a flowgraph. Let us consider again the sequence of two Flowgraphs P1 and P2 written as P1 o P2, then it holds

$$OV(P1 \circ P2) = OV(P1) + OV(P2) + 1.$$

However, there is a big difference of the combination rule of Pythagoras and the Measure OV. Let us assume for the lengths of the Boards a and b hold a=5 and b=7, then we get

$$c^2 = (a \circ b)^2 = 5^2 + 7^2 = 74.$$

Assume, that a and b were measured in kilometers. Then we can multiply the numbers with 1000 to apply the formula for meters. It holds then

$$1000^2\, c^2 = 1000^2\, (a \circ b)^2 = 1000^2\, 75 = 1000^2\, 5^2 + 1000^2\, 7^2.$$

Doing this, the formula of Pythagoras always is true. We are getting the right results in meters. We even can delete the 1000^2 on the left and right side of the equation above and the result is still true. We say, the Law of Pythagoras is invariant related to the transformation of the lengths a, b and c by any Constant $\propto > 0$. More formally we can write:

$$\propto^2 c^2 = \propto^2 (a \circ b)^2 = \propto^2 5^2 + \propto^2 7^2,$$

where $\propto > 0$. The term $\propto^2$ can be deleted without any problems. Of course, we expect such a behavior of lengths measurement. Let us consider the combination rule

$$OV(P1 \circ P2) = OV(P1) + OV(P2) + 1.$$

Again, we multiply the measurement values of OV() with $\propto > 0$. It holds

$$\propto OV(P1 \circ P2) = \propto OV(P1) + \propto OV(P2) + 1.$$

Here, we see a difference. We cannot delete $\propto$ because there is an additional constant + 1. Taking an example, with OV(P1) = 5 and OV(P2) = 7 it holds:

$$13 = 5 + 7 + 1.$$

Multiplying the numbers of OV() – not the Constant + 1 – with, for example, $\propto = 5$ it follows:

$$5\ 13 = 5\ \ 5 + 5\ 7 + 1,$$

=>

$$65 = 25 + 35 + 1.$$

The second equation is not true although the first equation was true (The left and right side are not equally). This shows that there are existing combination rules where we cannot apply the multiplication of a Constant $\propto > 0$. Coming back to geometry and physics, there we will analyze what is going on. The additive property of length and the Law of Pythagoras show us, that we have different numerical representations of lengths. We introduced an additive and non-additive one. However, everyone agrees, that the length of a o b or the length c of the hypothenuse is length measurement. We can walk along the length of a, b or c and nobody would say that it is not lengths in the usual sense. For this reason we need a common base for lengths measurement, which holds for additivity and non additivity of lengths. This common base is the extensive structure for lengths measurement.

Length measurement is based on the following conditions. Firstly, we can compare the lengths of two boards. We illustrate this with the next picture.

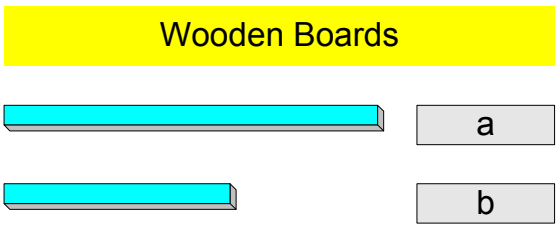


Figure 4.5: Lengths of boards can be compared without using a measure.

Lengths of boards here is considered from a qualitative view. Boards a and b can be compared without a measure. Everyone agrees that Board a is longer than Board b. Of course, we expect that a measure realizes that. The size of programs also can be considered from a qualitative view without using a measure.

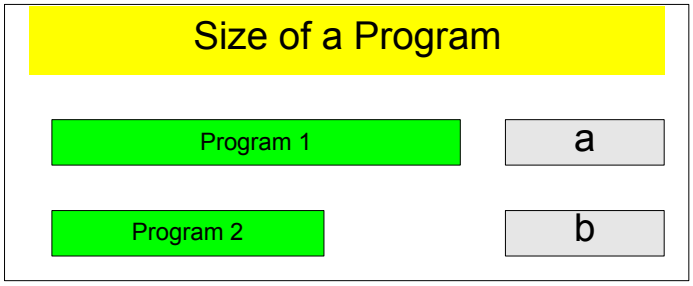


Figure 4.6: The size of programs also can be compared without using a measure.

Everyone agrees that Program 1 is large than Program 2. This is a qualitative view. However, behind the lengths of boards or length, generally seen, more is hidden than only the comparison. As we showed above, boards can be concatenated. For example, Boards a and b are concatenated to Boards a o b. Based on this concatenations the qualitative idea behind length is the following:

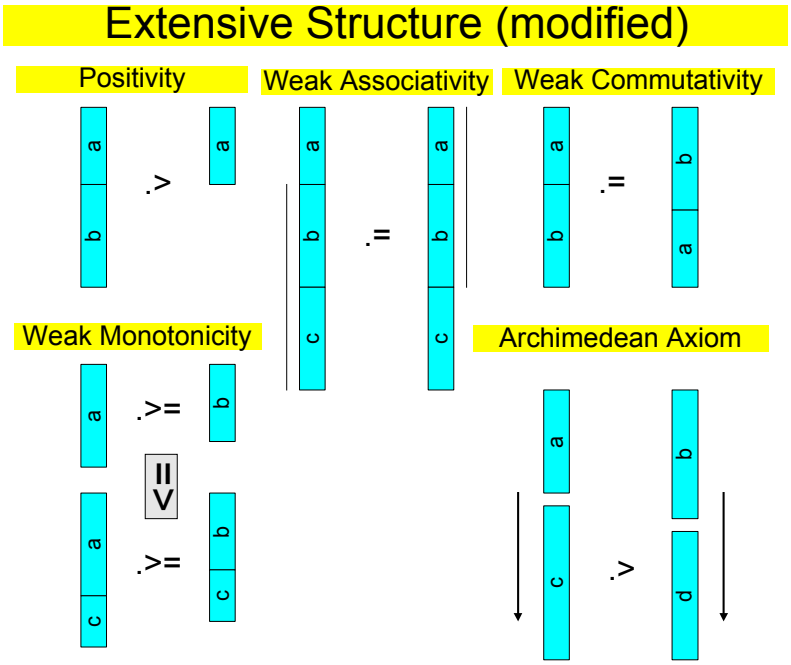


Figure 4.7: Qualitative explanation of lengths. The extensive structure is the name for the five qualitative conditions.

Lengths measurement can be explained by the five qualitative conditions above. The conditions are qualitative because no measure is involved. The set of the five conditions is called an extensive structure. For example, positivity means that having a Board a and Board b concatenated to Board a o b is always longer than Board a alone. Weak commutativity is clearly lengths behavior. The length of Board a o b is the same as for Board b o a. Or weak monotonicity says, that having Board a longer than Board b and adding Board c to both, then the length of Board a o c is greater than for Board b o c. The Archimedean condition also is easily to understand. Having two arbitrary Boards a and b and adding n-times Board c to Board a and Board d to Board b (Board c is longer than Board d), then it will never happen that Board a will be less than Board, even it held at the beginning Board a is less than Board b.

We reduced lengths measurement to qualitative conditions. It is a common base for lengths measurement. And, the interesting thing is, that agreeing to the qualitative properties above, the theorem of the extensive structure says, that there exists an additive measure with exactly these properties [1] (We started with this property at the beginning). However, the theorem only says, that such a measure exists. This implies, that non-additive measures for lengths also can exist. The reader sees at this end, that measurement is more than using measures. Measurement is based on qualitative conditions. The qualitative conditions above are valid for all kinds of length measurement, independently if we have an additive, non-additive behavior of a measure or we measure in km, miles, etc.
In Part III we will consider software measures from a qualitative view. We will see, that this it makes easier to understand the different numerical behavior of software measures.

References

[1] Zuse, Horst: *A Framework for Software Measurement*. DeGruyter Publisher, Berlin, Hawthorne, USA, 1997, 755 pages.

RULES OF THUMB FOR YEAR 2000 AND EURO-CURRENCY SOFTWARE REPAIRS

Capers Jones, Chairman
Software Productivity Research, Inc.
1 New England Executive Park
Burlington, MA 01803-5005

Abstract

Cost and schedule estimates for the year 2000 repair problem are extremely complex because this problem affects not only software but also data bases, test libraries, and hardware devices. Cost estimates for Euro-currency updates are more conventional, but are made more difficult by the fact that the work on the unified European currency and the work on year 2000 repairs are being carried out on parallel schedules, and hence the two tasks are contending for scarce resources.

Because many corporations lack estimating tools and estimating expertise even for normal software projects, estimating the work of these two massive simultaneous updates is far beyond their capabilities. Indeed, almost half of the U.S. Fortune 500 companies as well as a majority of government agencies have discovered that their initial estimates for year 2000 repairs were low by as much as 50%. These informal rules of thumb are not a substitute for a formal cost estimate for either problem, but can hopefully awaken software project managers and executives to the fact that the costs of these two massive updates should be derived from formal estimates using state-of-the art tools and methods.

Introduction

Cost estimating for the year 2000 problem is one of the most complex estimating tasks in the history of any industry. The reason why estimates are so difficult is because the year 2000 problem has some unique aspects which have not been encountered by the software industry in the past, and hence there is no historical data available. The unique aspects of the year 2000 problem include:

1. Thousands of software applications are affected simultaneously
2. Not only software but physical devices and data bases must be repaired
3. Unusual activities such as “triage” must be estimated
4. Possible damages and litigation expenses must be estimated
5. Hundreds of programming languages are involved in the problem
6. Both in-house and outsource work can take place simultaneously
7. Year 2000 search and repair technologies are new and uncertain
8. Multiple forms of year 2000 repairs are occurring simultaneously
9. For some applications, the source code is missing or uncompilable

Overall, the costs for making year 2000 repairs will comprise the largest software expense in history. Indeed, a strong case can be made that the year 2000 costs are the largest single business expense of any kind in history.

Cost estimating for the European currency updates is a more conventional exercise, but still unusually difficult and prone to error. The reasons why the Euro-currency estimates are complicated include:

1. The Euro work and the year 2000 work are occurring simultaneously
2. Hundreds or thousands of applications are on parallel update schedules
3. Many applications being updated are poorly structured and hard to repair
4. Both in-house and outsource work can take place simultaneously

The costs of updating software applications for the unified European currency comprise the second largest set of software costs in history, and probably the second largest business expense in history.

(The fact that the politicians of the European Union would schedule the introduction of the unified European currency so that it is competing with the year 2000 problem for scarce software resources is one of the worst public policy decisions in human history.)

The probable outcome of what will occur when the Euro meets the Year 2000 may resemble what happened when the Titanic met the iceberg. The year 2000 problem will slice through all of the Euro applications that are not year 2000 compliant, and may well sink the economies of the countries within the European Union.)

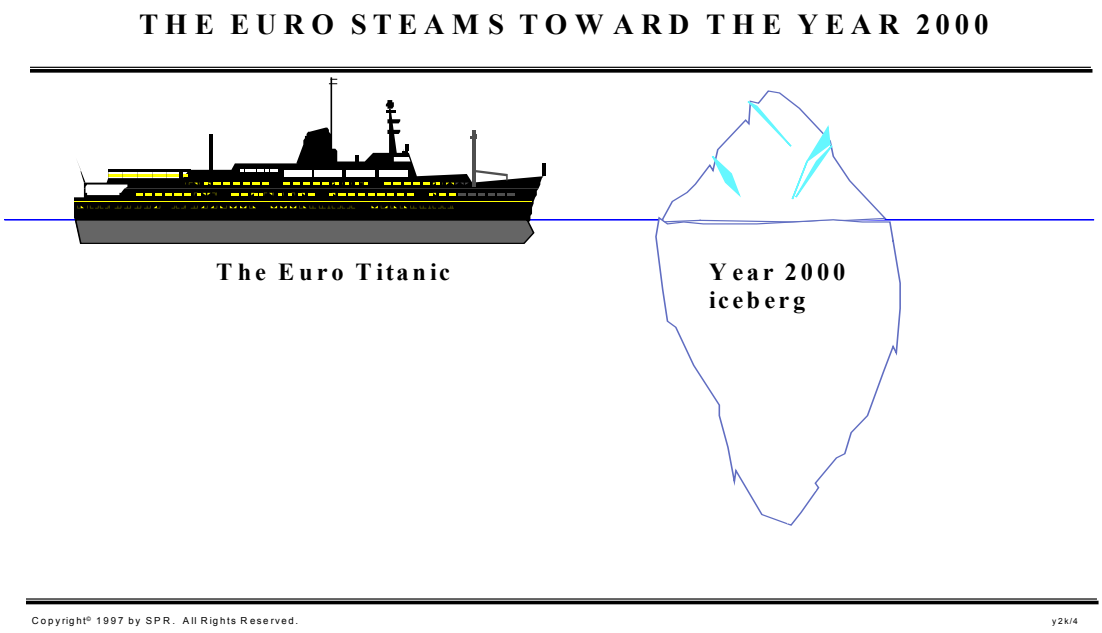


Figure 1: The Probable Result of the Euro Meeting the Year 2000

The rules of thumb in this article are inadequate for contracts or serious business purposes. Their purpose is simply to give executives and project managers a way of judging the approximate magnitude of these two massive software problems. The rules are taken from my book, *The Year 2000 Software Problem - Quantifying the Costs and Assessing the Consequences* (Addison Wesley Longman, 1997).

The rules of thumb in this article are based on the function point metric, and assume version 4 of the function point counting rules as published by the International Function Point Users Group (IFPUG).

Function points have become the dominant metric for software productivity and quality research, and the IFPUG organization and its affiliates in some 20 countries comprise the largest software measurement group in the world.

A few rules of thumb are based on the older "lines of code" (LOC) metric and some rules are given using both LOC and function points. With some 500 programming languages involved in the year 2000 problem, and many other kinds of non-coding work as well, the LOC rules are not safe for anything but yielding very approximate ranges. In particular, LOC metrics are not safe for:

- Hardware repairs
- Data base repairs
- Litigation preparations
- Repairs in languages such as Visual Basic

The LOC rules in this article are derived from languages such as COBOL and C, and should not be used for object-oriented languages, program and application generators, visual languages, or other non-procedural languages.

Rules of Thumb For Year 2000 Repair Estimation

Accurate cost estimating for year 2000 repairs is extremely complicated, since it involves software repairs, data base repairs, test library repairs, tools purchases, hardware upgrades, contracts, and a host of other factors.

For the past 50 years, software cost estimates have almost always erred in one direction: they are too low. The year 2000 problem is following this same classic pattern. A majority of large corporations have found that their initial year 2000 repair estimates are too low, often by more than 50%.

- If you depend upon date field expansion, you can approximate the overall calendar period from the start of your "triage" to achieving a fairly good level of year 2000 compliance by raising the overall size of your portfolio in function points to the 0.3 power. This rule of thumb will give the approximate time period in calendar months. For example a portfolio of 100,000 function points raised to the 0.3 power indicates a calendar period of 31.6 calendar months to move from triage to compliance.

- If you depend exclusively upon windowing, bridging, and other forms of "masking" for your year 2000 repairs, raise the size of your portfolio to the 0.25 power instead of the 0.3 power. For example a portfolio of 100,000 function points raised to the 0.25 power indicates a calendar period of 17.8 calendar months to move from triage to compliance via making rather than true date field expansions.

- If you operate in an IBM mainframe environment and can utilize object-code interception for year 2000 repairs, then raise the size of that portion of your portfolio to the 0.2 power instead of the 0.3 power which would be used for date field expansion. For example a portfolio of 100,000 function points utilizing primarily object-code interception would need about 10 calendar months to move from triage to compliance (assuming that object-code interception works well enough to be termed compliance).

- If you started year 2000 repairs in or before 1996 with date field expansions as your main strategy, expect your total expenses for achieving year 2000 compliance to approximate 30% to 50% of your typical annual budget for your company's software organization.

- If you started year 2000 repairs in 1997 with a mix of date field expansions and windowing or bridging or some other masking method, expect your total expenses for achieving year 2000 compliance to approximate 40% to 60% of your typical annual budget for your company's software organization.

- If you start year 2000 repairs in 1998, expect your total expenses to absorb more than 50% of your annual software budget for 1998 and about 75% for 1999 unless you are fortunate enough to be able to use object-code interception or some other temporary masking method. (You may not finish on time however.).

- In the absence of any other data, assume that year 2000 repairs will take at least 4 to 6 staff months of effort on the part every software employee in your company for a mix of date field expansion and masking.

- Assume that 50% of the applications in your portfolio are dormant and do not need repairs, and 50% are active applications which do need repairs.

- Assume that the source code is missing or uncompilable for about 15% of the aging legacy applications in your active portfolio, so either replacement or object-code repairs will be necessary.

- Assume that "dead code," comments, and blank lines in your applications, for which you should not have to pay repair costs, totals 30% of the total volume of physical lines in every application undergoing year 2000 repairs.

- Assume that the amount of code in your active applications which needs repair is about 5% to 10% of the total code measured in physical lines.

- Assume that your initial software year 2000 repairs using date field expansion and based on the number of lines of code needing repair (5% of the total) will proceed at a rate of 1500 lines of code per staff month, or 15 function points per staff month.

- Assume that "top gun" experts can achieve double the nominal rate of 1500 lines of code or 15 function points per month, but that novices will achieve only half of the nominal rate.

- Assume that you will need at least one full-time year 2000 repair specialist for every 2,500 function points (roughly 250,000 source code statements) in your active portfolio if you

have 24 months in which to accomplish year 2000 repairs. As the elapsed time passes, your staffing needs will rise accordingly. Using January of 1998 as the starting point, your staffing needs will change by 250 function points per calendar month.

- In the absence of "data point" metrics assume that your data base repairs and your source code repairs will take roughly the same amount of effort; i.e. one month of data base repair work for every month of software repair work.

- Assume that true date field expansion for data bases is so complex and time consuming that alternatives such as data duplexing, windowing, or bridging are the possible strategies for year 2000 data base repairs commencing in January of 1998.

- Assume that testing and regression testing your year 2000 repairs will each take about 60% as much effort as the repairs themselves, and more than half of the calendar time. Year 2000 testing is unusually complex, and final testing requires cooperative testing with your clients, suppliers, government agencies, and perhaps even competitors if you are in an industry such as telecommunications or public utilities.

- Assume that "bad fixes" will approximate 10% of your initial defect repairs, and hence the repairs will have to be done over. The bad-fix injection rate for year 2000 repairs has received little attention in the press, but will prove to be a major problem.

- Assume that 10% of your year 2000 test cases are incorrect and will need to be repaired themselves. If you are doing Euro-currency updates and year 2000 repairs on the same application at the same time, using different teams, assume that the bad fix injection rate will be 15%.

- Assume that your software year 2000 repairs will be about 95% efficient, but that 5% of the total possible number of year 2000 "hits" will be missed before the end of the century, and will not be discovered until after the year 2000 itself.

- If your organization uses embedded software in manufacturing or process control devices, assume that only about 80% of these can be repaired in time, and 20% will fail when the year 2000 occurs.

- In the absence of any metrics for repairing embedded devices, assume that the costs will roughly 50% of the costs of your software repairs. This is an unsafe rule of thumb, but since many manufacturing and process control companies have ignored embedded devices and omitted them from their year 2000 estimates, it is better than a zero value.

- Assume that year 2000 repairs will degrade application performance and throughput by 10% to 20% unless performance monitoring, careful tuning, and extensive testing are used to regain performance.

- If your enterprise average for software compensation levels is about $60,000 per year, expect your year 2000 repair costs to run from $0.25 to $1.00 per physical line of code if you do the work yourself, and you use date field expansion as your primary repair method. This is equivalent to $25 to $100 per function point. A mid point would be about $0.50 per physical line of code or roughly $50 per function point.

- If you outsource your year 2000 repairs to a contractor, expect your year 2000 repair costs to run from $1.00 to $2.50 per physical line of code. This is equivalent to $100 to $250 per function point. This assumes date field expansion as the primary repair method. A mid point would be about $1.75 per physical LOC or $175 per function point.

- If you utilize windowing, bridging, or some other masking approach as your primary year 2000 repair strategy, and you do the work with your own staff, your costs per LOC will be lower than with date field expansions. Assume your costs can run from about $0.10 to $0.50 per physical line of code, which is roughly equivalent to $10 to $50 per function point. This assumes about $60,000 per year as an average compensation level for your staff.

- If you outsource your year 2000 repairs and masking is the method utilized (i.e. bridging, windowing, etc.) assume that the costs will run from about $0.20 to $1.00 per physical lines of code, or roughly $20 to $100 per function point.

- Assume that the damages, litigation, and recovery costs from unrepaired year 2000 problems in software and data bases will cost about $3.00 to more than $20.00 per physical line of code. This is equivalent to between $30 and $2000 per function point.

- Do not assume that any cost data about year 2000 repair costs will be accurate for your enterprise unless the data uses the same compensation rates, burden rates, and work pattern assumptions that your enterprise uses. This is an obvious fact and does not require a specific source of information.

Once again, these rules of thumb are not safe for serious year 2000 estimating purposes, but are included for the benefit of year 2000 project managers and enterprise executives who are seeking some kind of general guidance about potential resources for dealing with the year 2000 issues.

To illustrate some of the many complexities of year 2000 repairs, table 1 illustrates some of the variations in schedules noted for various alternative repair strategies. Table 1 shows the approximate number of calendar months required to implement a selection of year 2000 repair strategies. Table 1 has a significant margin of error, but one point cannot be overemphasized: Time is running out for the year 2000 problem, and a significant number of companies and applications will not be repaired in time.

Table 1 is based on the results of fairly large corporations which own at least 100,000 function points in their portfolios and employ several hundred software personnel. In other words, table 1 is based on the time needed to fix at least 100 software applications and about the same number of data bases.

Table 1: Approximate Calendar Months to Implement Year 2000 Strategies

Year 2000 strategy	**Calendar Months to Implement**

Data base date field expansions	40
Software application date field expansions	36
Data duplexing	36
Redevelopment of applications > 2500 function points	30
Redevelopment of applications < 2500 function points	22
Compression	21
Bridging	20
Windowing	18
Encapsulation	14
Replacement via commercial packages	13
Substitution of manual methods	12
Object-code interception (if possible)	6
Average	22

Table 1 assumes that a company will be doing year 2000 repairs with their own personnel. It is possible to bring in contractors or outsource vendors and these specialists may be able to do the work somewhat faster. Indeed, a capable set of year 2000 experts such as the specialists employed by the major year 2000 outsource vendors can probably accomplish year 2000 repairs in about 15% less time than inexperienced groups just beginning their year 2000 repairs.

There are also a number of year 2000 tools available that can find the year 2000 problem and assist in repairing it for common languages such as COBOL. These year 2000 "search engines" can speed up the process of finding year 2000 hits, and some can assist in repairing straight-forward instances of date problems.

The best case scenario, which would be the combination of "top gun" personnel and a full suite of year 2000 search and repair engines can do better than table 1 by about 30%. However, doing worse than table 1 can occur too, and does occur with distressing frequency.

Rules of Thumb for Euro-Currency Conversion

The European Union has scheduled the beginning of a unified currency for Western Europe to start on January 1, 1999 and then expand until 2002. Of course this puts the Euro-conversion work in conflict with the much more serious year 2000 repair work. It is hard to imagine a more serious public policy mistake than scheduling the second largest software project in history so that it conflicts with the largest software project in history.

This is such a serious international error that it lowers the probability that the European Union will be able to finish either their year 2000 repairs or their Euro currency work on schedule. In fact, the timing of the Euro currency work is bad enough to lead to a recession in Western Europe. However, politicians seem to know and care little about technical problems so Europe is going to have to live with their errors.

The following rules of thumb are only approximate and not a replacement for a full-featured cost estimating tool. Note that for Euro-currency estimates, under-estimating has been a very common problem.

- To estimate the schedule for any software application that is being updated for the European-currency raise the size of the application in function points to the 0.25 power. This will give the approximate number of calendar months from the start of analysis to final testing. For example if you are updating an application of 1000 function points raising 1000 to the 0.25 power yields a schedule of 5.6 calendar months.

- For applications that are being simultaneously updated for both the Euro and for year 2000 repairs, use 0.3 rather than 0.25. For example, updating a 1000 function point application for both Euro and year 2000 repairs would take 7.9 calendar months if 1000 is raised to the 0.3 power.

- If you lack all other data points, you can assume that Euro-currency updates will take between 2.5 and 4 staff months on the part of all of your software employees.

- Assume that about 25% of the applications in your total software portfolio have currency calculations and will need updates. This rule is for generalized companies such as manufacturing. For key industries such as banks, retail chains, hotels, etc. the percentage can top 70% of all software applications.

- Assume that the amount of code in your active applications which needs Euro-currency modification repair is about 7% of the total code of financial applications measured in physical lines.

- Assume that your initial Euro-conversion work based on the number of lines of code needing repair (7% of the total) will proceed at a rate of 1200 lines of code per staff month, or 12 function points per staff month.

- Assume that you will need at least one full-time Euro-conversion specialist for every 2,500 function points (roughly 250,000 source code statements) in your active portfolio if you had 24 months in which to accomplish Euro repairs. As the elapsed time passes, your staffing needs will rise accordingly. If you start Euro conversion in the summer of 1998 you will need one full-time for about every 1,500 function points (roughly 150,000 source code statements) in your active portfolio.

- In the absence of "data point" metrics assume that your data base updates and your source code updates will take roughly the same amount of effort; i.e. one month of data base repair work for every month of software repair work.

- Assume that testing and regression testing your Euro-conversion work will each take about 45% as much effort as the repairs themselves, and just under half of the calendar time.
- Assume that "bad fixes" will approximate 10% of your Euro updates, and hence the repairs will have to be done over. If you are doing year 2000 repairs and Euro repairs on the same application, using different teams, assume a 15% bad fix injection rate.

- Assume that 10% of your Euro-conversion test cases are incorrect and will need to be repaired themselves.

- Assume that your Euro-currency conversion work will be about 90% efficient, but that 10% of the total possible number of Euro "hits" will be missed and will not be discovered until the software fails or produces erroneous results.

- If your enterprise average for software compensation levels is about $60,000 per year, expect your Euro-conversion costs to run from $0.50 to $1.25 per physical line of code. This is equivalent to $50 to $125 per function point.

- If you outsource your Euro updates to a contractor, expect your costs to run from $1.15 to $2.75 per physical line of code. This is equivalent to $115 to $275 per function point.

- Do not assume that any cost data about Euro-currency costs will be accurate for your enterprise unless the data uses the same compensation rates, burden rates, and work pattern assumptions that your enterprise uses. This is an obvious fact and does not require a specific source of information.

- If yours is one of the unfortunate companies that is tasked with trying to do both year 2000 repairs and Euro-currency work at the same time, be cautious and conservative. These two massive tasks are not synergistic but rather antagonistic, and the combination will tend to raise bad fix injection rates, more than double costs, and stretch out schedules so that neither task has a high probability of finishing on time.

Summary And Conclusions On Manual Estimating

Simple rules of thumb are never very accurate, but continue to be very popular. The main sizing and estimating rules of thumb and the corollary rules presented here are all derived from the use of the function point metric. Although function points metrics are more versatile than the former "lines of code" metric the fact remains that simple rules of thumbs are not a substitute for formal estimating methods.

The year 2000 problem is the largest and most complicated software estimating problem in history. The European currency updates comprise the second largest software estimating problem in history and is also complex, although not as complex as the year 2000 work.

Both of these massive problems need formal cost estimates supported by state-of-the-art cost estimation tools and methods. The simple rules of thumb shown here are not suited for contracts or serious business purposes, but are being published merely to raise awareness of the magnitudes of these two critical software issues.

References

[1] Abran, A. and Robillard, P.N.: *Function Point Analysis, An Empirical Study of its Measurement Processes.* IEEE Transactions on Software Engineering, Vol 22, No. 12; Dec. 1996; pp. 895-909.

[2] DeJager, Peter and Richard Bergeon: *Managing 00 - Surviving the Year 2000 Computing Crisis*. John Wiley & Sons, 1997.

[3] Dreger, Brian: *Function Point Analysis*. Prentice Hall, Englewood Cliffs, NJ; 1989; ISBN 0-13-332321-8; 185 pages.

[4] IFPUG *Counting Practices Manual*, Release 4, International Function Point Users Group, Westerville, OH; April 1995; 83 pages.

[5] Jones, Capers: *The Year 2000 Software Problem - Quantifying the Costs and Assessing the Consequences*. Addison Wesley Longman, 1998.

[6] Jones, Capers: *Principles of Software Cost Estimating*. (in production at McGraw Hill and scheduled for summer of 1998).

[7] Jones, Keith: *Year 2000 Software Crisis Solutions*. International Thomson Computer Press, 1997.

[8] Kappelman, Leon (editor): *Solving the Year 2000 Problem*. International Thomson Computer Press, 1997.

[9] Lefkon, Dr. Dick (editor): *Year 2000 Best Practices for Y2K Millennium Computing: Panic in Year Zero*. Mainframe Special Interest Group (SIG) of the Association of Information Technologies (AITP); New York, NY.

[10] Petzinger, Thomas Jr.: The Front Lines column; *Bob Bemer Aims 'Silver-Plated Bullet' at Year 2000 Problem*. Wall Street Journal; June 20, 1997; page B1.

[11] Kendall, Robert C: Year 2000: *How to Make the Problem Smaller, How to Make it Bigger*. American Programmer magazine; Vol. 10, No. 6; June 1997; pp. 25-28.

[12] Ragland, Bryce: *The Year 2000 Problem Solver*. McGraw Hill, 1997.

[13] Robbins, Brian and Rubin, Dr. Howard: *The Year 2000 Planning Guide*. Rubin Systems, Inc.; Pound Ridge, NY; 1997.

[14] Ulrich, William and Ian S. Hayes: *The Year 2000 Software Crisis - Challenge of the Century*. Prentice Hall, Yourdon Press; 1997.

THE SOFTWARE ENGINEERING CHARM - NAVIGATING BETWEEN CRAFT AND SCIENCE

Christof Ebert, Alcatel Telecom, Antwerp

Abstract

Society in the information age is heavily dependent on reliable computing and communication technology. However, due to the lack of an engineering base this technology is not accessible and the development of necessary software can not be guaranteed. The dilemma is new in that a discipline without solid foundations is taken so seriously by other disciplines and by the whole society that three decades after the discipline's emergence this very society and millions of people are heavily dependent on its correct functioning. The goal of this paper is to make aware the necessity that we have to treat software engineering as an immature engineering discipline in order to learn from other disciplines and hence to progress.

Keywords

engineering, maturity, science theory, software engineering

1 Introduction

Our modern society is increasingly dependent on software in all forms, sizes and complexities to drive computers that are embedded in nearly all aspects of daily life. We are on the other hand consistently surprised by the inability of software engineers to deliver software with the required functionality on time and within budget. While the society should not wait for a catastrophe before it tries to improve this critical resource, the software engineering community - which is ourselves - has to think about positioning itself correctly and finally start to learn from other disciplines and from within.

How can this be achieved? After all the so-called "software crisis" is as old as software engineering as a discipline. Many talented individuals tried to overcome the crisis over these 30 years. What should not happen is to blame the first two (or were it already three ?) generations of serious software engineers for the named problems. Civil engineering needed centuries of trial and error, and mechanical engineering also took decades during certain stages of craftsmanship before visible advances had been achieved. However, the dilemma is new in that a discipline without solid foundations is taken so seriously by other disciplines and by the whole society that three decades after the start of the discipline (which again is completely new to be dated so exactly) this very society with millions of people is heavily dependent on its correct functioning.

The goal of this paper is to make aware of the necessity to treat software engineering (in the course of this paper abbreviated as SE) as an immature engineering discipline. We selected the words "immature engineering discipline" carefully. Although behaving often more like a craft, we think that any engineering discipline emerges on a trajectory in the craft vs. science spectrum (fig. 1). Engineering in our point of view includes both craftsmanship and scientific working. Maturity means that both aspects of the same engineering discipline have evolved towards what is considered mature (e.g. repeatability, reflected and defined methods, processes, and artifacts, etc.).

To stimulate discussion we decided to use only black and white for coloring our critical self assessment with the implicit danger of being blamed of being too negative. A panelist of a virtual roundtable recently even questioned the necessity of SE as a discipline on its own [8]. Indicating that most parts of computer science are today influenced by other disciplines he pointed out that SE could become part of the different application domains' related disciplines. The viewpoint of this paper is less radical, questioning instead how to build upon available experience and how to learn from other disciplines in order to improve qualitatively.

The paper is organized as follows. The following section acts as a methodological introduction and provides some restrictions and underlying assumptions. The third section positions SE somewhere in between a craft and a science. Problems that arise with its current position are briefly summarized and the underlying assumptions of this article are given. Section 4 notes resulting scientific questions that are not answered completely in this study. The key to providing answers is given in sections 5 and 6 that look on insights from within science theory and on answers from within SE.

2 Some restrictions and assumptions

In order to narrow the focus of this paper we did not include systems engineering aspects although we clearly see software and systems engineering foundations and current positions being closer than to any other engineering discipline. The basic difference between these two disciplines is that systems engineering positions itself entirely in the application domain, that is complex systems. Systems engineers are from origin engineers from other areas that have to work interdisciplinary and hence to think and act as generalists that are experts in the application domain (e.g. space projects). A parallel - which we should take as one starting point of seeing similarities and learn from each other - is that in the case of systems engineering it was also industry that saw huge deficits in knowledge and abilities of their workforce that could not be provided by universities [3]. Companies such as GD, TRW, or NASA thus started their own systems engineering programs.

This paper does not intend to treat topics that have been discussed elsewhere comprehensively, though they should nevertheless be considered while talking about SE as a science. This includes educational aspects, curriculum aspects [13], the legitimization of SE as a discipline (i.e. separated from computer science) [16,17], research goals and programs, or research credits and evaluation of research results. Such aspects had been named before by Berry in his outstanding paper on the academic legitimacy of the SE discipline [1]. Research aspects and goals had been discussed broadly in the Computer Science and Technology Board (CSTB) available from the US National Academy [2]. Substantial critique of current research in SE including helpful directions towards improving the situation had been provided recently by Glass [15].

Although we feel the strong need of a common language for any science, we do not strive to re-write a glossary. Most of our terminology is according to the established wording of the current SE scientific community and the related engineering standards (e.g. [4]). Especially the use of "discipline" and "science" might sometimes look strange. We distinguish between an engineering discipline (e.g. civil engineering) and a science (e.g. natural sciences), to underline that engineering has aspects of a science and of a craft.

Due to the broad topic and the many well-known open questions that are often only restated, we tried to emphasize those inputs that can directly be used for finding answers and for getting further. DeMarco and Lister once remarked that "somewhere today a project is failing" [12] which exactly shows the tragedy. We know about it (everybody does), however since we do not know where to find remedies and treats (besides a lot of really helpful project- or people-specific cookbooks), we close our eyes and are sure that this time, it must work. It almost never works and the few success stories provide the stuff for the next cookbook [16]. There are so many of these cookbooks around that practitioners often question their applicability and helpfulness and typically start writing their own project-specific handbooks. And here we are, questioning from within industry what could be done together with academia to shape SE as a solid engineering discipline.

We will finish this section with our assumptions towards a path of SE to an engineering discipline that is able to mature:

- SE is an engineering discipline and should hence learn from other engineering disciplines to struggle its way to maturity;
- there is no engineering discipline without scientific foundations (e.g. common language, defined methods, hermeneutics);
- computer science is but one scientific disciplines that SE is based upon.

Obviously SE has to consider inputs from other disciplines, from science theory, and - in order to accelerate quickly - from within its already available and broad set of experiences. This is to be done in the following sections.

3 From a Craft to a Science

The still shortest definition of SE is given in [4] and is based on a similar definition of "engineering": SE is [5] the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. SE is [6] the study of approaches as in [5]. The maturity of SE as an engineering discipline is determined by the flow of scientific results (e.g. from computer science) to daily work and of problems resulting from daily business as questions to other sciences. Being a science implies knowing about and reflecting the scientific foundations. The second part of the definition makes clear that we reflect our methods and hence we are really talking about a science.

Upon trying to find SE's position in a spectrum of craft vs. science (fig. 1) several interesting problems arose that are typical for immature sciences:

There is no common language within the community (take for example the use of words like "process" or "method"). Of course this makes scientific discourses impossible.
Hermeneutic questions are typically not addressed by the community, and thus (empirical) validation of what we call "knowledge" is more than questionable. We could even go further and state that there is no theory in SE that has ever been treated seriously in terms of applicability and validation or falsification.

Available solutions that we earlier called cookbook recipes lack broad applicability. Limit and risk assessments are typically conducted superficially or postmortem, and so are cost benefit analyses. Too often software engineers and their companies rely on some method advocate who suggests a new method that is told to have worked somewhere else without even questioning environments or comparing processes. Nobody ever provides insight about applicability to classes of problems because there are no such classes. Lack of industrial acceptance is the result.

Controlled experiments are almost never conducted and upon reporting results from such experiments (e.g. [6]) the conclusion is drawn that they are typically impossible in SE for cost reasons. A survey of 400 recent research articles in SE showed that of those that would require experimental validation 40% have none at all compared to 15% in other disciplines [5]. Even postmortem analyses are neglected in industry due to an assumed lack of interest to really learn from other projects, and of course fear to unveil management deficiencies.

We will contrast these observations with a traditional engineering discipline, namely electrical engineering (EE). For centuries there was no discipline called EE and even associated crafts were more or less magic. Reducing simple observations to what was essence (as opposite to accident) during the last century provided laws and a completely new (scientific) branch of physics (e.g. by scientists Kirchhoff, Faraday, or Maxwell). Almost immediately the practical applicability was obvious and mechanical engineering expanded to what was later known as EE (e.g. by engineers Siemens or Bell). Ever since new theoretical insights that usually swapped over from physics were instantaneously used by engineers to provide technical artifacts for the use of mankind. A good example of such chains from theory to practice is the generalization of Kirchhoff's laws to the theory of circuits and then to CAD tools that allow for easy design, test and simulation of new circuits. Resulting problems from daily applications of circuit theory are continuously played back and forth to the theoretical branch of EE to obtain applicable solutions. The tools in turn adapt and provide these theoretical results almost immediately in a way that is useful for practitioners.

The path of EE to an engineering science was paved with small and controlled experiments. Complexity was reduced by considering regular geometric shapes. Related questions were reduced to "limitation problems" and considered much later. All theories emerged from such small examples and even today circuit design of VLSI chips starts with elementary cells and then reduces them in size, plays a little with geometry and materials, and later places other cells nearby.

4 Resulting Scientific Questions

Contemporary science theory considers everything scientific that is treated by reasonable people as such - independent of methods and approaches. Therefore, our introductory assumption on SE being an engineering discipline and thus being also a science still holds from this viewpoint. Trying to position the current state of SE in the already mentioned spectrum resulted in many singular areas, each with its own spot, however not connected by something called scientific or engineering framework (fig. 2). Nevertheless we investigated SE in terms of scientific approaches of other engineering disciplines. Several questions came up while trying to sort out aspects that help in building a solid engineering discipline.

Dependent on the reader's background the list may seem irrelevant (for a craft) or superficial (for science theory).

What is the meaning of SE? This is probably the most difficult question, because the easy answer based on the SE definition (i.e. the application of engineering to software) does not progress towards the pragmatic component of "meaning".

What are the underlying assumptions of SE as a scientific discipline? We know that especially SE is shaped by conceptual confusion in dealing with phenomena pertaining to human beings and computers (e.g. "intelligence", "communication") or in separating processes from artifacts (e.g. "design", "quality").

What is theory in SE? We might generalize Naur's view towards the assumption that SE is theory building [9]. Theory building in this broad context is to find ways in which software can meaningfully be applied to meet customer demands. Theory as it is included in any software system finally deeply influences maintenance activities, since theory building is tied up with people who might not be available in the future.

What makes the SE discipline mature? Knowing that a mature engineering discipline needs to be mature as a craft and as a science leaves this question open. Due to a completely inductive approach the maturity of an engineering discipline is associated with validity and broad applicability of basic hypotheses.

What is scientific quality in contrast to the quality of resulting artifacts? Intuitively we know that in order to provide improved products, SE has to evolve as a discipline. However, we must clearly separate one from the other. The problem with design is that we are trying to invent a form to fit in a context that we do not fully understand [10].

How is it feasible to separate SE specific problems from domain specific problems? This question is important as SE usually provides artifacts or products that are to be used in other domains. The question might be enhanced from an ontological point of view by asking what is "real world" in SE.

What are choices or decisions and what is determination during any design process? Often the software engineer is portrayed as making design decisions in an open situation, where there are several possibilities to choose from. In paying attention to design decisions, there is obviously an emphasis on our responsibilities, and hence an ethical dimension of our activities is included in this discussion. This coincides with the search for values that allow for repeatable conflict resolution.

How should SE experiments be designed and conducted in order to validate or reject underlying theoretical assumptions? This includes project size, repeatability, appropriate measurements, dealing with all influencing factors, keeping these factors stable but one, proper and suitable statistical techniques, consideration of time and learning curves, stochastic systems design, simulation, reliability and quality analysis of results, and the reporting of the results.
How can interdisciplinary techniques effectively being integrated in an emerging science? SE is often considered a conglomerate of many disciplines, while the adaptation of techniques from these disciplines is neglected. Aspects to be integrated include holistic thinking

capability and conceptual analysis, basic principles and theory of several other engineering disciplines, interpersonal skills, organization, and administrative skills.

How is complexity to be managed in an interdisciplinary area? Complexity in this context has to be seen from different viewpoints, namely interacting disciplines that build the foundation of SE (e.g. technical constraints vs. management restrictions vs. ethical guidelines), system and environmental aspects of any specific application domain interacting in a given project (e.g. real-time issues, interfaces to other systems), software complexity (e.g. interacting components), or psychological complexity (e.g. human interface, maintenance aspects).

Is there a difference between theoretical understanding and methodical support? Of course this question provokes discussion because everybody agrees - from a scientific point of view - that there must be a difference. There are many hints on the other hand that in SE descriptive and prescriptive approaches are typically not clearly separated and if so, only to bridge the gap between modeling and real world (take any introduction to SA or OOA as an example). Since there are no such things as methods a priori (those a posteriori are published anyway), what should result is situative method selection and adaptation.

Is there anything such as discourse theory in SE? We don't think so, however we like to emphasize questioning who is our audience; what do they know and which problems do they have; and how is our research helpful for them. Studies showed that typically 40 % of the total elapsed effort goes almost evenly distributed into specifying, coding, and testing, while another 40 % of the effort are eaten by communication and information exchange. It is amazing that even in teams that work together for a long time lots of problems cannot be discussed openly due to fixed positions, ownership of ideas (NIH syndromes) and unwillingness to retract.

Most of these questions have already been answered for other engineering disciplines. However, there can be no single theoretical framework providing answers to the set of questions raised above. We will try to provide some inputs in chapters 5 and 6. We realize that while collecting problems and trying to provide some inputs from other areas we always reflect our own perspectives and experiences.

5 Inputs from Science Theory

The immature state of SE is reflected by the lack of even a general framework of paradigms, categories, or conceptual schemes that is subject to so many scientific revolutions in other sciences. The first step towards more scientific thinking in this context is the current transition away from treating all software - which includes both problem descriptions and design approaches - as essentially being the same. Techniques for approaching distinct domain-specific problems provide improvements in quality and productivity that seemed impossible for general-purpose SE techniques (e.g. component-ware, frameworks).
Talking about science and SE as a science of course raises the question whether we need something like a SE science similar to other engineering sciences. Perhaps a craft would fit for all purposes beyond what could be provided by associated sciences. From a science theory point of view there are several answers that are however self-contained and lack applicability to the original question. The primary goal of science might be the acquisition of knowledge. Another more behaviouralistic view is that scientific results are prescriptions for actions in

practical decision-making situations. Both viewpoints are combined once the function of knowledge is questioned. While mathematics, humanistic sciences, or philosophy strive for explanation and understanding, natural or social sciences strive for prediction and control. Possible scientific contents of SE include:

- the subject-object relations between the researcher, the object of research, and the scientific community in a situation where the objects of research are changing rapidly;
- the relationships between descriptive and prescriptive knowledge (i.e. the combination of theory and practice);
- the applicability and the limits of methods from other sciences.

Of course the search for knowledge or truth alone are not sufficient as the main objectives of science. However, the search for applicable knowledge justifies a more scientific approach. Knowledge acquisition in natural sciences and engineering disciplines is based on an inductive approach. This proceeding from the special observation to the general theory is pictured in fig. 3 as a spiral. The increasing radius is a measure of the discipline's maturity. Observations that are based on inputs from application domains, such as project experiences, eventually result in hypotheses that combine several observations. These hypotheses might be fed with inputs from other related disciplines or sciences, such as EE or CS.

The critical boundary between a craft and a science is crossed upon formulating generalizing laws from the given hypotheses. Laws that are not proven false for a while are considered know-how because they allow for engineering work, for example by providing methods and tools for improved software production. The inductive cycle is completed when a general theory is formulated that allows for deductive approaches, such as coining new hypotheses for other areas. We consider all accepted theories as the knowledge of SE that in turn establishes links to other sciences and disciplines by realizing that distinct questions had been formulated elsewhere before.

6 Answers that we already know

It would be bewildering in this context to neglect those broad experiences made in the 30 year history of SE. Hence we try to provide insights from within our community that can help in building a sound engineering discipline. Considering the following suggestions shows that there is both a framework for scientific advance and enough space to overcome the software crisis in a controlled manner.

Shape the SE community. Something we can easily provide from within is a set of applicable standards and a defined language. To make such things common thinking, training and education need to be adjusted. Although SE educational needs at universities have been addressed before, they need a flavor of real life which could be provided by business school like case studies. It must be ensured that research and practice talk the same language. Taxonomies that bridge SE language and that of application domains can help in organizing our knowledge in a way that not only improves communication but also organizes and structures the way we think. We have to learn from our projects, and it should not be too difficult to get sufficient archival material or even material from ongoing projects for that purpose. Licensing of staff based on at least minimal competence for general practice similar

to other disciplines also influences professionalism. Methods for monitoring and dealing with professional malpractice combined with formal decertification might set additional targets for a true engineering discipline. An instrument that could be used to steer the direction of SE as a scientific discipline is tenure. If tenure committees and funding agencies focused on quality instead of quantity of research (publications) results would be more valuable.

Apply the same restrictions as in other engineering disciplines. Take civil engineering and think about the degree of parallelism that is possible due to clearly defined processes, interfaces between them and resulting products. In other disciplines separated processes became almost independent domains of systematic scientific inquiry. The need obviously is to reduce the complexity of processes. Processes have their importance since they allow local control and reproducible results that can be planned and tracked.

Reduce complexity. It had been pointed out that software in general is so complex that it lacks the degree of repetitiveness which is so characteristic of artifacts in other engineering disciplines [11]. However, from a distant point of view all engineering artifacts seem complex and still can be controlled [7]. The key is to keep things as simple as feasible. Once distinct techniques are applicable to controlled complexity, complexity can slowly be increased in limited areas and where experience is available. Reducing complexity means separating problem-specific complexity (e.g. functionality, interconnections between domains) and solution-specific complexity (i.e. the complexity created by constructing an artifact). Typical SE problems arise when both complexities are tangled, for instance when performance (as a problem-specific aspect) is improved at the sake of modularity (which is solution-specific). Instead of focusing on maintaining legacy software on the one hand and creating sophisticated object-oriented frameworks on the other, SE could instead show how to migrate and replace legacy systems by better architectures.

Allow for routine work. Current software engineering practice is based on uniqueness of projects [11]. Knowing that something similar has been done before is considered as having no practical impact because some interfaces might differ and several environmental flavors could have changed. Other engineering disciplines clearly separate between routine design (including reuse and adjusted copies) and innovative design. SE too should support routine tasks and investigate how to improve exactly what is never taught at universities: how to copy what is good and how to make use of what already exists.

Consider all influencing factors according to their individual impacts. A typical example is the people factor [12]. Knowing about it does not mean considering it. Give another boss to a successful team and it may separate immediately. It is by no means easy to consider such factors, however we have to deal with past experiences. The sorry state of SE is underlined by knowing that practitioners don't even care for "scientific" results because they have no idea how to transfer them to their own environments, while researchers disregard studies done by other researchers because too often it is impossible to get further information from the authors.

Consider different viewpoints. Viewpoints' impacts are often neglected. Consider a survey of assumed capability maturity in a big organization. Independent of any "objective" assessment (if this is no oxymoron) management would surely rank the organization much higher than practitioners that know about the daily deficiencies. Closely related to viewpoints and to the maturity of processes are attitudes. As long as firefighting is rewarded in many organizations

instead of smoothly steering a project, sound engineering and planning techniques will not evolve.

Conduct small controlled experiments. Almost all natural sciences, and hence all engineering sciences, gained their knowledge by proceeding from the particular to the general. Scaleability should be an underlying goal of experiment design. Completeness in terms of projects and solutions must be ensured which supports the classification of projects that can be used for benchmarking. Experiments and measured pilots with readily available results should substitute "common SE sense" that too often judges over applying or replacing existing practices.

Keep quantitative approaches sound and simple. Lord Kelvin is quoted with the insight that science starts with accurate measuring. Measurement seems to be the key and still Fenton et al recently showed that many reported projects and experiments are worthless for the SE community due to invalid statistical procedures and neglecting influencing factors [6]. Software measures and models should be mathematically and statistically sound. A quality standard checklist used by reviewers could serve not only as entry criteria upon submitting an article but also could improve experiment design.

Learn from your faults. The willingness to reflect on our own practice in terms of faults is psychologically difficult to achieve. Failures may be costly, however not learning from failures is even more expensive. The same holds for management faults. The typical top-management reaction on a missed schedule is to "improve productivity", which is all too often achieved by reducing effort for "non-productive processes" (such as requirements engineering, training or reviews). If schedules are missed, we have to improve our planning capabilities and if estimations are wrong we have to adjust estimation techniques and not the development process.

Consider the application domains. Models are used in various SE areas without a theory for building and validating models. Fundamental assumptions about the nature of the application area are typically made without even realizing what reality is and what our perception of reality is [14]. Limits of our models are not questioned due to lack of knowledge about real limits. Upon finishing a model it is used for abstractions to reduce the solution space and again nobody is able to validate these abstractions in relation to reality.

Take whatever you can learn from other disciplines. Technology transfer is the key to linkages in many engineering areas, cross-country and cross-disciplinary. The adaptation of configuration and change management techniques from the aircraft industry to SE was but one example in the past. Today it might be the confession towards clear and stable interfaces between reusable software components. Requirements engineering might learn from hermeneutics which is concerned with the interpretation of text in a very broad sense. Software design might learn from architecture [10]. In order to learn from other disciplines we must be aware that this touches education and skills. Kipling is quoted saying that "those who know only England do not know England". As long as our workforce is recruited mainly from CS, they cannot be expected to be open and sensible towards engineering issues.

7 And how do we proceed from here?

Several steps must be accomplished, to progress to what we call maturity. Applying the named principles or inputs from within SE is the first step. It is now the time to rate research results according to their impact to making SE mature. This could help in reducing one effect of advocacy research, namely that practitioners simply ignore research results. If there is a crisis, then it is not an average 20 % schedule or 30 % budget overrun (which by the way is quite common in other engineering projects), but this acceptance gap between theory and practice.

SE practitioners should change their approach towards complex software systems and the underlying development process. Complexity is not completely essential, and thus approaches to reduce and to manage complexity must be applied. Both sides, researchers and practitioners, should cooperate more closely which includes openness towards exchange of problems and solutions. Proprietary interests of companies are inefficient and expensive in the long run. Classroom examples and "experiments" that lack scaleability are worthless because they hinder discussion. Research support and funding from governments or industry should be more oriented towards systematic problem-solving and broad dissemination of knowledge. The probably best example for implementation of this approach is the NASA's Software Engineering Laboratory.

While this paper only reflected on few observations dealing with SE in the craft vs. science spectrum, it intents to stimulate discussions on the topic. Many approaches especially several that are mentioned in the SE related references below indicate not only awareness but also improving maturity of the discipline. After all we still happen to deliver software - independent of schedule and budget constraints - that is used by millions of people worldwide. The good news is that despite all mentioned scientific pitfalls and with an "initial maturity level" some key players are highly profitable. It seems as if we have to learn from another discipline that never became clear about being craft or science: medicine. It is not so much the position in the craft vs. science spectrum that distinguishes between a doctor and a quack - it is the degree he serves mankind.

References

[1] Berry, D.M.: *Academic Legitimacy of the Software Engineering Discipline*. Techn. Rep. CMU/SEI-92-TR-34, Software Engineering Institute, Pittsburgh, USA, 1992.

[2] Scaling Up: *A Research Agenda for Software Engineering. Computer Science and Technology Board*. Available through Nat. Academy Press, 2102 Constitution Ave., Washington, DC 20418, USA. Excerpts in ACM Comm., Vol. 33, No. 3, pp. 281 - 293, Mrc. 1990.

[3] Shenhar, A.: *Systems Engineering Management: A Framework for the Development of a Multidisciplinary Discipline*. IEEE Transact. SMC, Vol. 24, No. 2, pp. 327 - 332, Feb. 1994.

[4] IEEE *Standard Glossary of Software Engineering Terminology*. IEEE, New York, USA, 1991.

[5] Tichy, W.F., et al: *Experimental Evaluation in Computer Science: A Quantitative Study*. J. Systems and Software, Vol. 28, pp. 9 - 18, 1995.

[6] Fenton, N., S.L. Pfleeger and R.L. *Glass: Science and Substance: A Challenge to Software Engineers*. IEEE Software, pp. 86 - 95, Jul. 1994.

[7] Waldrop, M.M.: *Complexity - The Emerging Science at the Edge of Order and Chaos*. Penguin Books, London, 1994.

[8] Lewis, T. (Ed.): *Where is Software headed? A Virtual Roundtable*. IEEE Computer, pp. 20 - 32, Aug. 1995.

[9] Naur, P.: *Computing: A Human Activity*. ACM Press. Addison-Wesley, Reading, USA, 1991.

[10] Alexander, C.: *Notes on the Synthesis of Form*. Harvard Univ. Press, Cambridge, USA, 1964.

[11] Parnas, D.L.: *Software Aspects of Strategic Defense Systems*. American Scientist, Vol. 73, No. 5, pp. 432 - 440, Sep. - Oct. 1985.

[12] DeMarco, T. and T.Lister: *Peopleware*. Dorset House, New York, USA, 1987.

[13] Freeman, P. and A.I. Wasserman: A *Proposed Curriculum for Software Engineering Education*. Proc. 3. Int. Conf. Software Engineering, IEEE Comp. Soc. Press, Los Alamitos, CA, pp. 56-62, 1978.

[14] Floyd, C. et al (Ed.): *Software Development and Reality Construction*. Springer, Berlin, Germany, 1991.

[15] Glass, R.L.: *The Software-Research Crisis*. IEEE Software, Vol. 11, No. 6, pp. 42 - 47, Nov. 1994.

[16] Wasserman, A.I.: *Toward a Discipline of Software Engineering*. IEEE Software, Vol. 13, No. 6, pp. 23 - 31, 1996.

[17] Shaw, M.: *Prospects for an Engineering Discipline of Software*. IEEE Software, Vol. 7, No. 6, pp. 15 - 24, Nov. 1990.

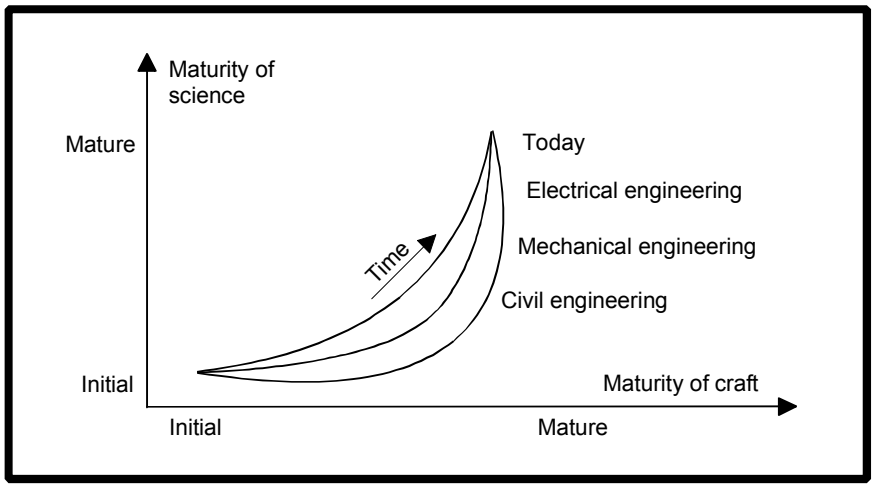


Fig. 1: The evolution of engineering disciplines as trajectories in the craft vs. science spectrum

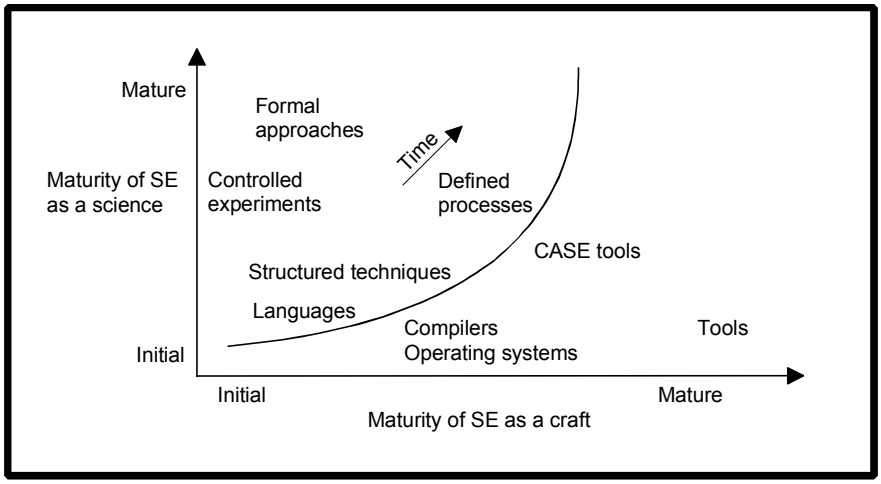


Fig. 2: The SE evolution and some remarkable milestones in the craft vs. science spectrum

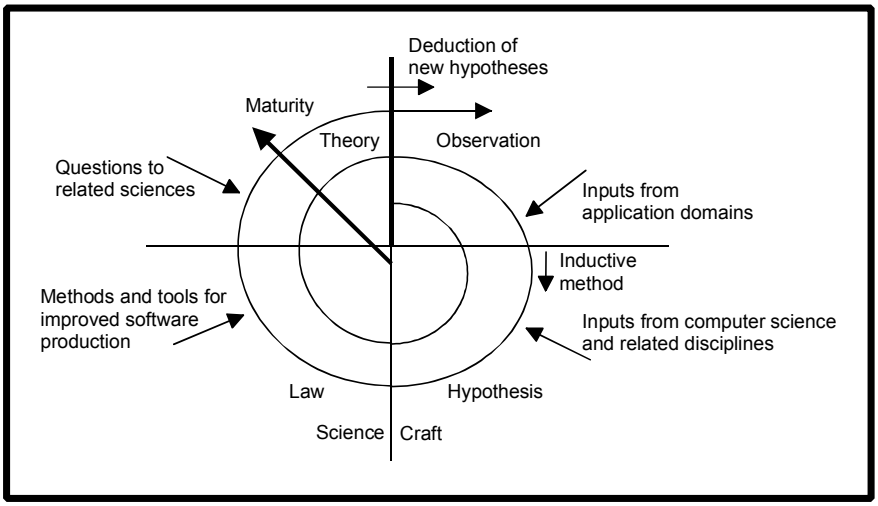


Fig. 3: Evolution of the software engineering discipline: from observations to theory

STRENGTHS AND WEAKNESSES OF SOFTWARE METRICS

Capers Jones, SPR, Inc.

Abstract

The software industry lacks standard metric and measurement practices. Almost every major software metric has multiple definitions and ambiguous counting rules. Further, there are no standards for dealing with basic topics such as the activities to include in software measurement studies. There are also key topics with no metrics at all, such as quantifying the volume or quality levels of data bases and data warehouses. The result is a lack of solid empirical data on software costs, effort, schedules, quality, and other tangible matters. This report analyzes some of the key software size metrics and the underlying technical problems associated with software measurement.

Introduction

Measurement, metrics, and statistical analysis of data are the basic tools of science and engineering. Unfortunately, the software industry has existed for more than 50 years with a dismaying paucity of measured data, with metrics that have never been formally validated, and with statistical techniques that are at best questionable.

One of the fundamental measurement issues for software that need to be put on a firm and rational basis is how to measure the size of various software deliverables. A full treatment of the size topic is too large for a journal, so this article only summarizes the current state of software size metrics research.

What Software Artifacts Require Size Data?

Before turning to the available metrics for dealing with sizes, it is useful to step back and examine the kinds of items associated with software projects where size information is needed. There are 20 software artifacts where size information is important because the costs of creating or dealing with these deliverables is significant:

Table 1: Twenty Software Artifacts Requiring Size Metrics

1) The functionality of the application
2) The volume of information in data bases
3) The quantity of new source code to be produced
4) The quantity of changed source code, if any
5) The quantity of deleted source code, if any
6) The quantity of base code in any existing application being updated
7) The quantity of "dead code" no longer utilized but still present
8) The quantity of reusable code from certified or uncertified sources
9) The number of paper deliverables (plans, specifications, documents, etc.)
10) The sizes of paper deliverables (pages, words)
11) The number of national languages (English, French, Japanese, etc.)

12) The number of on-line screens
13) The number of graphs, and illustrations
14) The sizes of non-standard deliverables (i.e. music, animation)
15) The number of test cases that must be produced
16) The number of bugs or errors in requirements
17) The number of bugs or errors in specifications
18) The number of bugs or errors in source code
19) The number of bugs or errors in user manuals
20) The number of secondary "bad fix" bugs or errors

The available metrics for quantifying the sizes of software deliverables include the following:

- Natural metrics such as counts of pages and words in paper documents
- Source code metrics using physical lines
- Source code metrics using logical statements
- Function point metrics as defined by the IFPUG organization
- Function point variants as defined by a dozen other organizations
- Object-oriented metrics of various kinds
- Counts of bugs or defects

A major omission from this list of metrics is the absence of any known size metric for data bases, data warehouses, or repositories.

Strengths and Weaknesses of Source Code Metrics

When the software industry began in the early 1950's the first metric developed for quantifying the output of a software project was the metric termed "lines of code" or LOC. Almost at once some ambiguity occurred, because a "line of code" could be defined either:

- A physical line of code.
- A logical statement.

Physical lines of code are simply sets of coded instructions terminated by pressing the enter key of a computer keyboard. For some languages physical lines of code and logical statements are almost identical, but for other languages there can be major differences in apparent size based on whether physical lines or logical statements are used.

Table 2 illustrates some of the code counting ambiguity for a simple COBOL application, using both logical statements and physical lines:

Table 2: Sample COBOL Application Showing Sizes of Code Divisions Using Logical Statements and Physical Lines of Code

Division	Logical Statements	Physical Lines
Identification Division	25	25
Environment Division	75	75
Data Division	300	350
Procedure Division	700	950
Dead code	100	300
Comments	200	700
Blank lines	100	100
Total Lines of Code	1,500	2,500

As can be seen from this simple example, the concept of what actually comprises a "line of code" is surprisingly ambiguous. The size range can run from a low of 700 lines of code if you select only logical statements in the procedure division to a high of 2,500 lines of code if you select a count of total physical lines. Almost any intervening size is possible, and most variations are in use for productivity studies, articles, etc.

Bear in mind that table 1 is a simple example using only one programming language for a new application. The SPR catalog of programming languages [5] contains almost 500 programming languages and more are being added on a daily basis. Furthermore, a significant number of software applications utilize two or more programming languages at the same time. For example combinations such as COBOL and SQL or Ada and Jovial are very common. SPR has observed one system that actually contained twelve different programming languages.

There are other complicating factors too, such as the use of macro instructions, inclusion of copybooks, inheritance, class libraries, and other forms of reusable code. There is also ambiguity when dealing with enhancements and maintenance, such as whether or not to count the base code when enhancing existing applications.

Obviously with so many variations in how lines of code might be counted, it would be useful to have a standard for defining what should be included and excluded. Here we encounter another problem. There is no true international standard for defining code counting rules. Instead, there are a number of published local standards which unfortunately are in conflict with one another.

Citing just two of the more widely used local standards, the Software Productivity Research (SPR) code counting rules published in 1991 are based on logical statements [6] while the Software Engineering Institute (SEI) code counting standards published in 1992 are based on physical lines of code [10]. Both of these conflicting standards are widely used and widely cited, but they differ in many key assumptions.

As an experiment, the author carried out an informal survey of code counting practices in software journals such as American Programmer, Byte, Application Development Trends, Communications of the ACM, IBM Systems Journal, IEEE Computer, IEEE Software, Software Development, and Software Magazine [7].

About a third of the published articles using LOC data used physical lines, another third used logical statements, while the remaining third did not define which method was used and hence were ambiguous in results by several hundred percent. While there may be justifications for selecting physical lines or logical statements for a particular research study, there is no justification at all for publishing data without stating which method was utilized!

The main strengths of physical lines of code (LOC) are:

1. The physical LOC metric is easy to count.
2. The physical LOC metric has been extensively automated for counting.
3. The physical LOC metric is used in a number of software estimating tools.

The main weaknesses of physical lines of code are:

1. The physical LOC metric may include substantial "dead code."
2. The physical LOC metric may include blanks and comments.
3. The physical LOC metric is ambiguous for mixed-language projects.
4. The physical LOC metric is ambiguous for software reuse.
5. The physical LOC metric is a poor choice for full life-cycle studies.
6. The physical LOC metric does not work for some "visual" languages.
7. The physical LOC metric is erratic for direct conversion to function points.
8. The physical LOC metric is erratic for direct conversion to logical statements.

The main strengths of the logical statements are:

1. Logical statements exclude dead code, blanks, and comments.
2. Logical statements can be mathematically converted into function point metrics.
3. Logical statements are used in a number of software estimating tools.

The main weaknesses of logical statements are:

1. Logical statements can be difficult to count.
2. Logical statements are not extensively automated.
3. Logical statements are a poor choice for full life-cycle studies.
4. Logical statements are ambiguous for some "visual" languages.
5. Logical statements may be ambiguous for software reuse.
6. Logical statements may be erratic for direct conversion to the physical LOC metric.

Strengths and Weaknesses of Function Point Metrics

The IBM Corporation, which in the 1970's was the world's largest developer of software applications, commissioned a team to develop a software metric which could be used for software economic studies regardless of what programming language, or combination of languages, were utilized for the code itself.
The IBM metrics team was headed by Allan Albrecht, and the result of their research was termed the "function point" metric. Function points were used internally by IBM in the mid 1970's, and then placed into the public domain at a presentation by Albrecht in October of 1979 at a conference in Monterey, California [2].

Function point metrics are developed from the requirements and specifications of a software application, and consist of the weighted and adjusted totals of five key elements:

1. Inputs (screens, signals, etc.)
2. Outputs (screens, reports, checks, etc.)
3. Inquiries
4. Logical files
5. Interfaces

The actual rules for counting and adjusting function points are fairly complex so training is needed to count function points accurately. The counting rules have passed from IBM, and are now controlled in the United States by the non-profit International Function Point Users Group (IFPUG). IFPUG publishes function point counting rules [11] and administers the certification exams which are a prerequisite for those who wish to become function point analysts.

Many software researchers recognized the advantages of function points for software economic studies, but were not completely satisfied with the form and structure of the function point metric as originally defined by the IBM team, and more recently defined by the IFPUG counting practices committee.

In 1983, Charles Symons gave a presentation in London on an alternative function point variant which he termed the Mark II Function Point [4], which is now widely used in the United Kingdom and to a lesser degree in Hong Kong and Canada.

The Mark II function point alternative was only the first in a growing set of function point variants, which now include at least these 25 alternative functional metrics. It is fairly obvious that 25 variations in how the function point metric might be counted is too many. The International Standards Organization (ISO) is attempting to develop a standard for function point sizing, but the complicated international politics of functional metrics are likely to interfere with the pure technical issues.

The main reason cited for not working directly with IFPUG is that the counting practices committee is viewed as slow-moving and not always receptive to external inputs.

Another reason for developing alternative function point metrics is based on a misunderstanding of function point principles. Function points were developed as a unit of measure for expressing the size of software applications.

Other factors in addition to size determine the effort required to build the application. Unfortunately, some of the alternative function point metrics were developed for dealing with software that is costly to construct, such as real-time and embedded software.

The reason for developing alternative function points for these difficult applications is that when productivity is measured using standard IFPUG function points, the productivity is rather low. Instead of exploring the reasons why real-time software productivity is often lower than information systems, some researchers prefer to develop alternative function points

which gave real-time software larger apparent sizes than standard IFPUG function points and hence artificially elevate the productivity of real-time development.

The overall range of results across the many function point variants has not been studied in detail, but is probably in excess of plus or minus 50%.

The main strengths of function point metrics are:

1. Function points stay constant regardless of programming languages used.
2. Function points are a good choice for full-life cycle analysis.
3. Function points are a good choice for software reuse analysis.
4. Function points are a good choice for object-oriented economic studies.
5. Function points are supported by many software cost estimating tools.
6. Function points can be mathematically converted into logical code statements for many languages.

The main weaknesses of function point metrics are:

1. Accurate counting requires certified function point specialists.
2. Function point counting can be time-consuming and expensive.
3. Function point counting automation is of unknown accuracy.
4. Function point counts are erratic for projects below 15 function points in size.
5. Function point variations have no conversion rules to IFPUG function points.
6. Many function point variations have no "backfiring" conversion rules.

Correlating Lines of Code and Function Points Metrics Via "Backfiring"

In the 1970's Allan Albrecht and his colleagues at IBM measured a number of projects using both logical source code statements and function point metrics. These pioneering studies found some interesting but not perfect correlations between source code size and function points for many programming languages.

Table 3 illustrates some typical ratios of logical source code statements and equivalent volumes of function points. This is a small excerpt from the main SPR table of languages with almost 500 entries [5]:

Table 3: Ratios of Logical Source Code Statements to Function Points for Selected Programming Languages Using Version 4 of the IFPUG Rules

Language	Nominal Level	Source Statements Per Function Point		
		Low	Mean	High
Basic assembly	1.00	200	320	450
Macro assembly	1.50	130	213	300
C	2.50	60	128	170
FORTRAN	3.00	75	107	160
COBOL	3.00	65	107	150
PASCAL	3.50	50	91	125
PL/I	4.00	65	80	95
ADA 83	4.50	60	71	80
C++	6.00	30	53	125
Ada 95	6.50	28	49	110
Visual Basic	10.00	20	32	37
SMALLTALK	15.00	15	21	40
SQL	27.00	7	12	15

Backfiring has become the most popular method for ascertaining function point sizes of aging legacy applications. In fact, for many legacy applications backfiring is the only convenient method for developing function point totals because the specifications are often missing and the original developers have departed.

Backfiring was quickly adopted by commercial software estimating vendors, and is now a common feature among most of the well-known software estimating products such as CHECKPOINT®, COCOMO II, GECOMO, KnowledgePlan™, and SLIM®.

Backfiring was also adopted for benchmark studies by a number of well-known consulting companies such as Compass Group, Gartner Group's Real Decision subsidiary, Meta Group, Quantitative Software Management (QSM), Rubin Systems, Inc.; and the author's own company, Software Productivity Research, Inc.

Because backfiring is so often used in commercial software estimating tools and also by management consultants and commercial benchmark companies, it is arguably the most widely used software metric in the world.

The accuracy of backfiring is not claimed to be as high as normal function point counts. The accuracy of backfiring from logical statements is about plus or minus 20%, while normal counting by certified function point counters has been measured to range by about plus or minus 10% in a study commissioned by IFPUG and performed by Dr. Chris Kemerer [9] when he was at MIT.

It is a curious situation that neither the IFPUG organization nor any of the other major function point associations have evaluated backfiring, and the published literature on this

method comes primarily from estimating tool vendors and software management consultants rather than the function point user community itself.

The main strengths of backfiring function points are:

1. Backfiring is extremely quick and easy to perform.
2. Backfiring automation is commercially available.
3. Backfiring is supported by many software cost estimating tools.
4. Backfiring is used in many software benchmark studies.

The main weaknesses of backfiring function point metrics are:

1. Backfiring is of lower accuracy than normal function point counting.
2. Backfiring is ambiguous if the starting point is physical lines of code (LOC).
3. Backfiring may be ambiguous for mixed-language applications.
4. Backfiring results may vary based on individual programming styles.
5. Backfiring is not endorsed by any of the major function point associations.

Strengths and Weaknesses of Object-Oriented Metrics

As the object-oriented (OO) paradigm began to spread throughout the software community, it was quickly apparent that OO projects needed to be estimated and measured just as had procedural projects. Both lines of code metrics and function point metrics have been utilized with OO projects with varying degrees of success [8].

The fundamental differences in the way OO projects are constructed compared to procedural projects quickly led to new kinds of software metrics aimed exclusively at OO projects. Some of the specialized OO metrics include those of Kemerer and Chidamber [3] in the United States originally termed “metrics for object oriented systems environments” or MOOSE; and Abrieu and colleagues in Portugal with their “metrics for object oriented design” or MOOD metrics [1].

Some of the specialized OO metrics constructs include weighted methods per class, depth of inheritance tree, number of children, coupling between object classes, and a number of others.

In the past, both lines of code metrics and function point metrics have splintered into a number of competing and semi-incompatible metric variants. There is some reason to believe that the OO metrics community will also splinter into competing variants, possibly following national boundaries.

The main strengths of OO metrics are:

1. The OO metrics are psychologically attractive within the OO community.
2. The OO metrics appear to be able to distinguish simple from complex OO projects.

The main weaknesses of OO metrics are:

1. The OO metrics do not support studies outside of the OO paradigm.
2. The OO metrics do not deal with full life-cycle issues.
3. The OO metrics have not yet been applied to testing.
4. The OO metrics have not yet been applied to maintenance.
5. The OO metrics have no conversion rules to lines of code metrics.
6. The OO metrics have no conversion rules to function point metrics.
7. The OO metrics lack automation.
8. The OO metrics are difficult to enumerate.
9. The OO metrics are not supported by software estimating tools.

Summary and Conclusions

Software metrics research is an important topic, but not yet a well-formed or mature topic. Each of the major software metrics candidates has splintered into a number of competing alternatives, often following national boundaries. There is no true international standard for any of the more widely used software metrics. Further, the adherents of each metric variant claim remarkable virtues for their choice, and often criticize rival metrics.

From a distance, the software metrics domain is fragmented, incomplete, and gives the appearance of being more influenced by “metrics politics” than by technical considerations.

References

[1] Abreu, Fernando Britoe; *An email information on MOOD*. Metrics News, Otto-von-Guericke-Univeersitaat; Magdegurg; Vol. 7, No. 2; p.11; June 1997.

[2] Albrecht, A.J.: *Measuring Application Development Productivity*. Proceedings of the Joint IBM/SHARE/GUIDE Application Development Conference; October 1979; reprinted in Jones, Capers; Programming Productivity - Issues for the Eighties; IEEE Computer Society Press; 1986.

[3] Chidamber S.R. and Kemerer, Chris F.: *Toward a Metrics Suite for Object Oriented Design*. OOPSLA 1991; pp 197-211.

[4] Garmus, David (Editor): *IFPUG Counting Practices Manual*. Release 4.0; International Function Point Users Group (IFPUG); Westerville, OH; April 1994.

[5] Jones, Capers: *Table of Programming Languages and Levels*. Version 8.2; Software Productivity Research, Burlington, MA; URL http://www.SPR.com.

[6] Jones, Capers: *Applied Software Measurement*. McGraw Hill; 1991; (Revised 2nd edition; 1996)

[7] Jones, Capers: *Critical Problems in Software Measurement*. IS Management Group, Carlsbad, CA; 1993.

[8] Jones, Capers: *Economics of Object-Oriented Software*. Software Productivity Research; Burlington, MA; April 1997.

[9] Kemerer, Chris F.: *Reliability of Function Point Measurement: A Field Experiment*. MIT Sloan School Working Paper 3192-90-MSA; January 1991.

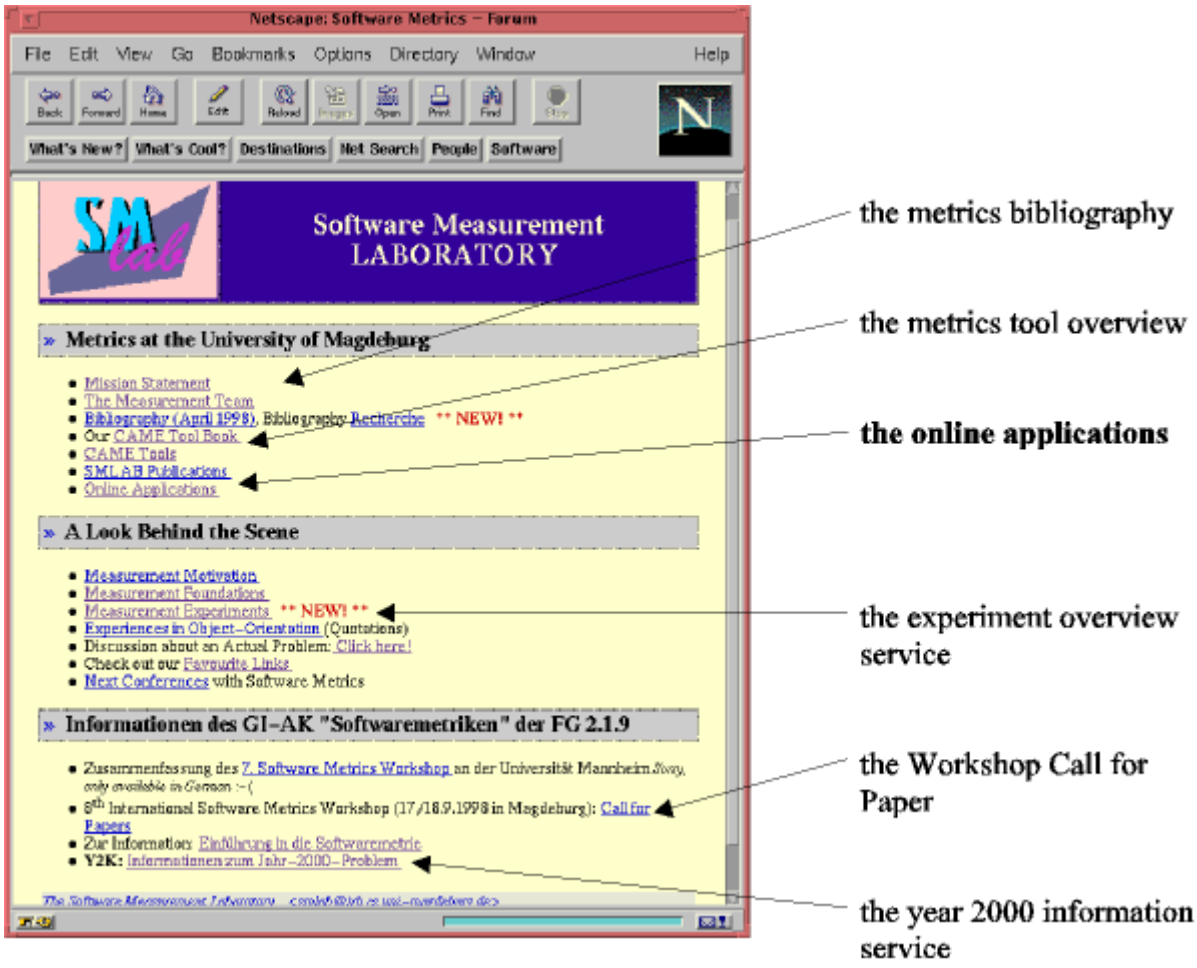
[10] Park, Robert E.: *SEI-92-TR-20; Software Size Measurement: A Framework for Counting Software Source Statements*. Software Engineering Institute, Pittsburgh, PA; 1992; 220 pages.

[11] Symons, Charles: *Software Sizing and Estimating - Mk II FPA*. John Wiley & Sons; Chichester, UK; 1991.

EDUCATION IN SOFTWARE MEASUREMENT IN THE WWW

Reiner R. Dumke, University of Magdeburg

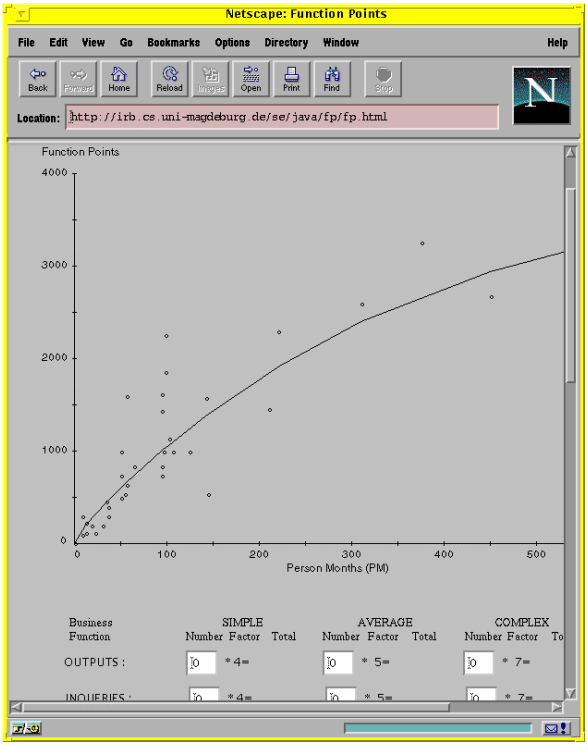
In the last month, we have extended the possibilities to use examples of process evaluation based on Capability Maturity Model, ISO 9000, and product evaluation based on the Function Point Method and the COCOMO. You can use it for education as online (Java-based) application located in our SMLAB (***http://ivs.cs.uni-magdeburg.de/sw-eng/us/***).



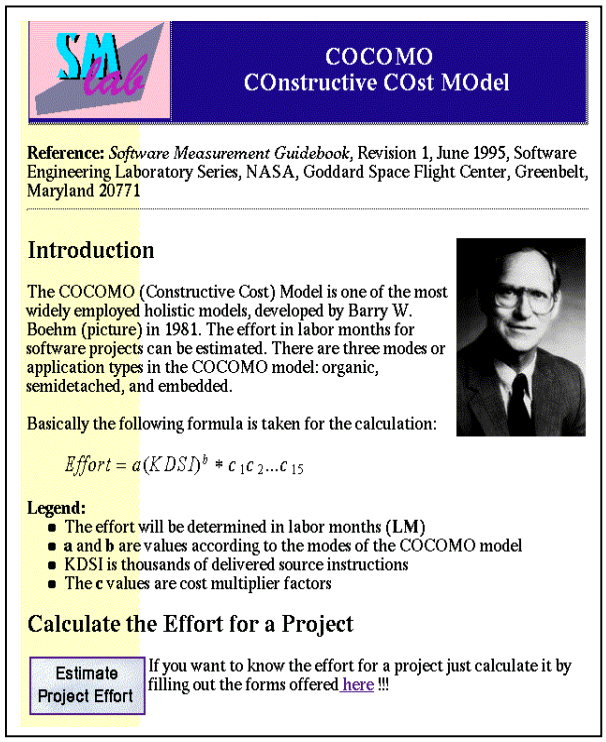
The next pictures present the modified versions of the current evaluation methods and give an impression of the first pages of the online applications for process and product evaluations.

Position Papers

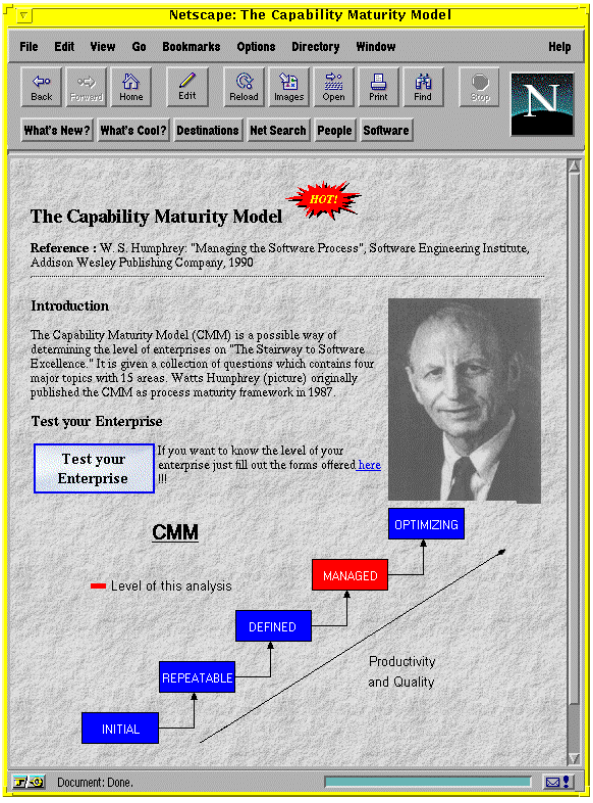
The Function Point Evaluation



The COCOMO-based Evaluation



The CMM-based Evaluation



The ISO 9000 Evaluation

THE ROLE OF FUNCTION POINT METRICS IN THE 21ST CENTURY

Capers Jones, SPR, Inc.

Abstract

The function point metric began in the mid 1970's as a basic size metric for software projects. By the 1990's Function points had expanded far beyond that modest beginning. Function points are now being used for software quality studies, software contract management, business process reengineering (BPR), and software portfolio control. The Internal Revenue Service is also exploring the usage of function point metrics for software taxation purposes.

This article covers the recent expansion of function point metrics, and points out some areas that are currently outside the scope of function points. If these areas are brought under the function point umbrella, the software community may start the 21st century with one of the most powerful suites of metrics of any technical domain.

Introduction

Function Point metrics originated in IBM in the mid 1970's as a methodology for sizing, estimating, and measuring software projects. The function point metric was first publicly discussed by the inventor, Allan Albrecht, at a joint IBM/SHARE/GUIDE conference in Monterey, California in October of 1979.

IBM put into the function point metric into the public domain at that time. Albrecht's conference paper on function points had only a limited circulation in the conference proceedings but IBM began to teach function point metrics to interested clients.

The first international publication of the function point metric occurred in 1981 in Capers Jones book, Programming Productivity -- Issues for the Eighties which was published by the IEEE Computer Society press. This book included Albrecht's full paper on function points, with the permission of IBM and the Monterey conference organization.

When the power and utility of the function point metric became known, usage of the metric expanded very rapidly and quite spontaneously. By 1984 usage of function points among IBM's client base had grown enough to form the nucleus of the present International Function Point Users Group (IFPUG), which started in Toronto, Canada. Also in 1984, Albrecht and IBM issued the first major revision to the function point counting rules.

From the time of the founding of the IFPUG organization, usage of the function point metric expanded very rapidly throughout the world. By the early 1990's, the function point metric had become a major tool for software project managers. Function points could be quantified directly from software requirements, they could be understood by both clients and software development personnel, and they were far less erratic and ambiguous than the former "lines of code" metric for sizing, estimating, and measurement purposes.

New uses for the Function Point Metric

Not only did the function point metric expand in terms of numbers of users, but the applications of the function point metric turned out to be far broader than originally envisioned. Following are brief discussions of the newer uses for the function point metric and the approximate year when the usage started.

1980 -- Function Points and Programming Language Evaluation

Allan Albrecht and his colleagues at IBM used both function points and the older lines of code (LOC) metrics concurrently. This concurrent usage of both metrics led to the discovery of "backfiring" or direct conversion from LOC to function points as early as 1979. By 1980, it was also realized that the function point metric could be used to evaluate the "level" of programming languages, as well as the resulting productivity rates.

Prior to function point analysis, definitions of programming language levels were highly ambiguous. It can now be stated with some justification that the phrase "high level" languages should be applied to languages that take less than 50 statements to encode one function point. By contrast, "low level" languages take more than 100 statements to encode one function point.

Information on the ratio of function points to source code is now available for more than 500 programming languages and dialects (Table of Programming Languages and Levels -- Version 8.2; Software Productivity Research, September 1996).

1985 -- Function Points and Software Cost Estimating

The first commercial software estimating tool that was designed and built around the function point metrics was the SPQR/20 tool released in March of 1985. From that point on, usage of function point metrics rapidly expanded through the software estimating business. In 1985, there were only four commercial software estimating tools on sale in the United States, and only one was based on function points.

By 1995, more than 50 commercial software estimating tools are being marketed in the United States, and at least 30 of the major ones support function point metrics. Some of the many commercial software cost estimating tools supporting function point metrics include, in alphabetical order: Before Your Leap (BYL), Bridge Modeler, CHECKPOINT, COCOMO II, Estimacs, GECOMO, KnowledgePlan, Price-S, ProQMS, and SLIM.

1986 -- Function Points and Software Quality

The function point metric is now rapidly becoming the preferred metric for software quality analysis. Current U.S. norms indicate a total of about 5.0 bugs or defects per function point, coupled with a defect removal efficiency that averages about 85%. This means that about 0.75 bugs per function point are delivered with the first releases of new software packages.

Best-in-class organizations can drop below two defects per function point in total potentials, and eliminate more than 95% of all defects prior to delivery to clients.

The former "lines of code" metric made many defect classes difficult to study. Function points are now being used to explore defects levels found in:

Requirements (about 15% of all defects)
Specifications (about 35% of all defects)
Source code (about 40% of all defects)
User documents (about 5% of all defects)
Bad fixes or secondary defects (about 5% of all defects)

1988 -- Function Points and Software Benchmarks

One of the major uses of the function point metric is to do benchmark studies, or comparisons of software quality and productivity between companies, between industries, and even between countries. The function point metric has become a basic tool for benchmark comparisons, and is used by a large and growing number of benchmark consulting groups. For example Compass Group, Gartner Group, IBM, and Software Productivity Research (SPR) all utilize function point metrics as the basis of their quantitative benchmark studies. As of 1997, at least 15 international consulting companies are using function points for software benchmark studies.

1989 -- Function Points and Software Portfolio Analysis

It was soon noted that function point metrics could be used for analyzing complete portfolios as well as individual projects. Function point based portfolio analysis has now been applied to a number of companies and even to several industries, such as banks, insurance companies, computer manufacturers, software companies, and the like. Smaller companies may own less than 250,000 function points in their corporate portfolios, but large corporations in the Fortune 500 category often exceed 1,000,000 function points in their full portfolios.

Note: This ability to utilize function point metrics for large collections of software written in multiple languages is leading to more powerful year 2000 estimating methods than the simplistic "lines of code" metric.

1990 -- Function Points and Tool or Environment Analysis

One of the most powerful uses of function point metrics is to perform multiple-regression studies to explore the impact of tools such as those of Sybase, DataEase, Lotus Notes, Texas Instruments, COGNOS, Microsoft, and the like. When productivity studies include the tools utilized as well as the function point productivity data, it is possible to derive the impact of an ever-growing number of software development and maintenance tools. For example, design tools, testing tools, complexity analysis tools, CASE tools, reverse engineering tools, cost estimating tools, project management tools, and a host of others are now being evaluated via function point benchmark studies.

It is now known that it takes more than 50,000 function points to fully equip a software engineering team. More recent studies have demonstrated that software quality assurance teams with more than 8,000 function points of tools have higher quality levels than QA groups that are sparsely equipped. Software project managers with more than 10,000 function

points of management tools on tap can do a better job of estimation and planning than managers with sparse tool suites.

1991 - Function Points and Make Versus Buy Analysis

Basic software application packages such as spreadsheets can sometimes be purchased for as little as $0.25 per function point. More specialized niche packages in the domains of finance or stock market analysis can cost in the range of $10.00 to more than $300.00 per function point. Information of this kind serves as a new way of evaluating the economics of make versus buy analysis. Software development costs can run from less than $200.00 per function point for small end-user applications to more than $2,000.00 for large military and defense applications. By enumerating acquisition costs versus development costs per function point, a new way of exploring packages is emerging.

1993 -- Function Points and Outsource Analysis

The function point metric provides a new and useful way for evaluating outsource contracts. In fact several outsourcers are already including "cost per function point" as part of their contracts for diverse topics including but not limited to: 1) Cost per function point for basic development projects; 2) Cost per function point on a sliding scale for creeping requirements or late improvements; 3) Cost per function point for maintenance and enhancement work.

The usage of cost per function points as a software contract methodology is starting to become more widespread. Not only that, but the method is also being used internationally. Several international outsourcers in India such as Tata are using the differential between their cost per function point and U.S. averages to leverage their business.

Function point cost differentials are also starting to be used by Eastern European outsource groups in the Russia, the Ukraine, and the Czech Republic. The main reason for the rapid adoption of function points by global outsource vendors is because of the significant marketing advantages which these off-shore outsourcers perceive. The average cost for building a function point in Western Europe is close to $1500 but in Eastern Europe the average cost is less than $300.

1994 -- Function Points and Business Process Reenginering (BPR)

Once a company quantifies the volume of its software portfolio using the function point metric, it is obvious that segments of that portfolio service the operating components of the business. For example, in a typical manufacturing company, out of a portfolio of 1,000,000 function points engineering might use 200,000 function points, manufacturing might use 350,000; marketing and sales might use 50,000; finance and administration might use 100,000; and so forth.

When BPR studies occur, the use of function point metrics provides a new and powerful tool for aligning the software capabilities of the company with the new or revised operating units.

1995 - Function Points and Taxation

The internal revenue service in the United States, and several equivalent groups abroad, are exploring function points for determining the taxable value of software assets. Function points are also being used for determining the assets value of software companies when they merge or are acquired. Since function point metrics are far more appropriate for economic analysis than the former "lines of code" metric this topic can be expected to grow in importance.

Between 1995 and 1997 function points and lines of code metrics were in direct conflict in at least half a dozen major U.S. tax cases, with the lines of code metric being on the losing side of the decision in every case.

Having learned from experience that function points are often on the prevailing side in litigation, the IRS has begun to utilize certified function point counters as expert witnesses. In the most recent tax litigation in 1996 and 1997 the IRS is now starting to use sophisticated cost models supporting function points, such as CHECKPOINT® and KnowledgePlan™ rather than their former crude lines of code model.

In one major tax case in 1996, function points and the CHECKPOINT® estimating tool were utilized by both the IRS and by the defendant, which may be the first tax case in which function points were utilized by both sides of the dispute without the older lines of code metric being part of the case at all.

1995 - Function Points and Outsource Litigation

A somewhat ominous sign that indicates function point metrics are now entering the main stream of business topics is the increasing number of lawsuits involving outsource contracts where function point metrics are part of the pleadings.

The author and his colleagues at SPR have served as expert witnesses in several lawsuits where function points were part of the cases involving breach of contract disputes between contractors and clients. In one major case, the outsource contract itself had included claims of increasing productivity over five years measured using function points. The suit alleged that the productivity gains had not been achieved, although the most troublesome issue was the original client productivity prior to starting the contract.

Other recent cases involve damage claims due to alleged poor quality by a contractor, and claims by a vendor to recover the costs of lost effort due to excessive requirements changes by a client. Function point metrics have even been part of a suit for wrongful termination of an employee.

Quite a few function point consultants are now serving as expert witnesses in litigation where function points are a significant factor. Since the software industry is notoriously prone to litigation, it can be assumed that this trend will probably escalate as we near the end of the century.

Indeed, now that function point metrics are being applied to various kinds of year 2000 repairs, it can be expected that they will play a major role in forthcoming year 2000 litigation.

1996 - Function Points and the Year 2000 Problem

There are more than 500 programming languages affected by the on-rushing year 2000 problem. This means that simplistic year 2000 cost measures such as multiplying the size of a portfolio by a fixed rate of $1.00 is not an adequate method.

One of the problems with outsource contracts based on the lines of code metric is the fact that many aging legacy applications contain significant quantities of "dead code" which may comprise 30% by volume of code in some applications. Obviously the more astute CIO's object to paying outsource vendors fees of between $1.00 and $2.00 per line of code for dead code, blank lines, comments, and other artifacts that have no year 2000 impact whatsoever.

Among SPR's clients, many of the more sophisticated companies are choosing to keep year 2000 repairs in house, primarily because some of the year 2000 vendors don't really know how to construct a year 2000 contract that is not one-sided and damaging to the client.

The first major book on year 2000 cost estimating using function points is Capers Jones' Economic Impact of the Year 2000 Software Problem which is now in production with Addison Wesley and will be published in the Autumn of 1997.

Potential Expansion of Function Point Metrics

In spite of the considerable success of function point metrics in improving software quality and economic research, there are a number of important topics that still cannot be measured well or even measured at all in some cases. Here are some areas where there is a need for either an expansion of the current function point metric, or possibly the development of related metrics within a broad family of functional metrics:

Improved Guidelines for Embedded and Real-time Software

Allan Albrecht is an electrical engineer by training, and always envisioned the function point metric as being appropriate for any kind of software. However, since the first use of the function point metric was for information systems, there has been a shortage of counting guidelines for the real-time and embedded software domain.

What is needed is an expanded set of counting rules, which include topics such as how to deal with sensor-based inputs, with situations where "inputs" may be changes in voltage potentials, and with "outputs" that may be electronic signals.

The Need for Data Point Metrics

In addition to software, companies own huge and growing volumes of data and information. As topics such as repositories, data warehouses, data quality, data mining, and on-line analytical processing (OLAP) become more common, it is obvious that there are no good metrics for sizing the volumes of information that companies own. Neither are there good

metrics for exploring data quality, the costs of creating data, migrating data, or eventually retiring aging legacy data.
A metric similar to function points in structure but aimed at data and information rather than software would be a valuable addition to the software domain. Aspects of the British Mark II function point metric which include entities and relationships are moving in the direction of data metrics.

Surprisingly, data base and data warehouse vendors have performed no research on data metrics, and are about to be "blind-sided" by the on-rushing year 2000 problem. Because the costs of data base repairs are much higher than the costs of software repairs for year 2000 purposes, many companies are lagging in performing this critical task.

There is a strong possibility that the year 2000 problem will put at least a temporary end to data warehousing and OLAP and will have very negative impacts on the entire chain of decision support tools and methods.

In a year 2000 context, the absence of an effective "data point" metric means that a major cost element cannot easily be estimated or measured using either function point metrics or the older lines of code metric.

The Need for Service Point Metrics

The utility of function points for software studies has raised the question as to whether or not something similar can be done for service groups such as customer support, human resources, sales personnel, and even health and legal professionals.

What would be useful would be a metric similar in structure to function points, only aimed at service functions within large corporations. Right now, there is no easy way to explore the lifetime costs of systems that include extensive human service components as well as software components.

Experiments with variations on the function point metric have been carried out for customer support groups, insurance claims handling, and medical office practices. The results have been encouraging, but are not yet at a point for formal publication.

The Need for Hardware Point Metrics

The U.S. Navy and the other military services have a significant number of complex projects that involve hardware, software, and microcode. Several years ago the Navy posed an interesting question: "Is it possible to develop a metric like function points for hardware projects, so that we can do integrated cost analysis across the hardware/software barrier?"

The ability to perform integrated sizing, cost, and quality studies that could deal with software, hardware, data bases, and human service and support activities would be a notable advance indeed.

Summary and Conclusions

Function point metrics are rapidly placing software quality, productivity, and economic studies on a firm economic base. It is fair to assert that function point metrics are rapidly becoming the dominant metric of the software world.

However, in spite of the great success of the function point metric, there are still important business topics that are difficult to measure. It might be possible to expand the function point metric into other domains, or more probably, to construct a family of metrics that can support integrated cost and quality studies across the domains of software, data bases, hardware, and services.

Suggested Readings on Function Point Analysis

Dreger, Brian: *Function Point Analysis*. Prentice Hall, Englewood Cliffs, NJ; 1989; 225 pages.

This was the first college primer intended to teach function points to those without any prior knowledge of the metric. This book is a very readable introduction to an important topic. It even manages to add a few items of humor to lighten up what might otherwise be a very dry topic. The only caveat about this book is that it was published in 1989, and the rules for counting function points underwent a minor revision in 1993 and a major revision in 1994. To learn the rudiments of function point counting the book is still useful, but to learn the most current rules and practices, the book is unfortunately out of date. As of 1996, this book will be available in a CD ROM edition under license to Miller Freeman publishers.

Garmus, David & Herron, David: *Measuring the Software Process: A Practical Guide to Functional Measurement*. Prentice Hall, Englewood Cliffs, NJ; Due out in November of 1995.

Function point metrics are expanding rapidly throughout the world. David Garmus is a member of the counting practices committee of the International Function Point Users Group (IFPUG). David Herron is a former member of the same committee, and the two authors have formed a consulting company specializing in function point metrics. This is a new primer aimed at introducing function point metrics to a wide audience. The book covers the new Version 4.0 counting practices revision and hence is current through 1995 in terms of IFPUG counting rules.

The book attempts to discuss other topics such as benchmarking and estimating, but the authors are not recognized experts in these topics and the discussions are slight and sometimes at odds with best current practices. Surprisingly, given that the title of the book is "Measuring the Software Process" the authors do not even discuss process or activity-level measurements. Their view seems to stop at entire projects, which is not granular enough for serious process analysis.

Howard, Phil: *Guide to Software Productivity Aids*. Applied Computer Research, Scottsdale, AZ; ISSN 0740-8374; published quarterly.

This is not a true book in the sense that most other citations are books. This is a quarterly catalog of software tools marketed in the United States. The catalog is larger than many book, and typically runs to perhaps 500 pages. The text of the tool descriptions are provided by the tool vendors, and hence may exaggerate features and functions. However, this book is useful because it has contact information for hundreds of vendors and descriptions of thousands of tools.

The ACR catalog lists all of the major cost estimating tools, function point counting tools, and project management tools which support function point metrics.

Jones, Capers: *Applied Software Measurement (2nd Edition)*. McGraw Hill, 1996; ISBN 0-07-032826-9; 618 pages.

This book has become a standard reference volume in many companies, and in some university software management curricula as well. The new second edition of this book includes U.S. national averages for software productivity and quality derived from more than 6700 projects using the Function Point metric for normalizing the data. It also included comparative data on productivity from 50 industries, including banking, insurance, telecommunications, computer manufacturers, government, etc.

A major revision was published in August of 1996 that included substantial new data on about 2000 new software projects, and discussed the emerging results of various new technologies such as object-oriented approaches, client-server applications, ISO 9000 certification, and the Software Engineering Institute (SEI) approach. The new edition shows data for six subindustries (system software, military software, commercial software, information systems software, contract and outsourced software, and end-user software). National averages are then derived from the overall results of these six domains.

Jones, Capers: *Assessment and Control of Software Risks*. Prentice Hall, 1994; ISBN 0-13-741406-4; 711 pages.

This book discusses some 65 technical and sociological risk factors associated with software development and maintenance operations. The data has been collected during the course of SPR's software process assessment activities. Among the technical risks are those of inadequate tools, inadequate methodologies, and inadequate support for quality assurance.

Among the social risks are those of excessive schedule pressure, the low status of the software community within many corporations, and the high risks of litigation. Other risks include the tendency of vendors to make false claims about quality and productivity,

and the tendency of software managers and staff to believe those claims without requiring proof. Several large companies are using this book as a guide for improving their software processes, and upgrading their software curricula for both management and staff. A somewhat controversial section of this book discusses the deficiencies and gaps in the SEI assessment program. The book includes quantitative data on "best in class" quality and productivity results derived from the top 10% of SPR's clients. A Japanese translation was published in August of 1995.

This book was published in CD ROM form in 1996 and is available through both Prentice Hall and Miller Freeman.

Jones, Capers: *Patterns of Software System Failure and Success*. International Thomson, Boston, MA; December 1995; 250 pages; ISBN 1-850-32804-8; 292 pages.

This book was published in December of 1995. The contents are based on large-scale studies of failed projects (i.e. projects that were either terminated prior to completion or had severe cost and schedule overruns or massive quality problems) and successful projects (i.e. projects that achieved new records for high quality, low costs, short schedules, and high customer satisfaction).

On the whole, management problems appear to outweigh technical problems in both successes and failures. Other factors discussed include the use of planning and estimating tools, quality control approaches, experience levels of managers, staff, and clients, and stability of requirements.

Also discussed are intermittent and extrinsic factors such as bankruptcy, layoffs, downsizings, litigation and other business problems that can affect software projects. It is possible to minimize the probability of failure and maximize the probability of success, but both technical and sociological changes must occur to achieve significant improvements.

Jones, Capers: *Software Quality – Analysis and Guidelines For Success*. International Thomson, Boston, MA; ISBN 1-85032-867-6; 1997; 492 pages.

This new book covers 70 topics that influence software quality: Inspections, ISO 9000-9004 standards, the SEI CMM, all forms of testing, total quality management (TQM), and many more. This book differs from many other books on software quality in that is shows the actual measured impact of the various methods in terms of defect potentials, defect removal efficiency levels, costs, and other relevant quantified information. Function points are used as the normalizing metric, and there are cautions that both “lines of code” and “cost per defect” exert severe bias when performing large-scale quality studies. The contents are based on large-scale studies of many software projects in the U.S., Europe, and the Pacific Rim. A useful feature of the book is the summarization of quality “best practices” based on empirical results rather than hypothetical considerations.

Software project management problems appear to outweigh technical problems in both projects with poor quality levels. Other factors discussed include the use of quality estimating tools, quality measurements, quality control approaches, experience levels of

managers, staff, and clients, and stability of requirements. Also discussed are sociological factors such as layoffs, downsizings, litigation and other business problems that can affect the quality of software projects. It is possible to optimize software quality, costs, and schedules concurrently but both technical and sociological changes must occur to achieve significant improvements.

Jones, Capers: *The Year 2000 Software Problem*. Addison Wesley Longman, Reading, MA; ISBN 0-201-30964-5; Due in the Autumn of 1997; 300 pages.

This new book utilizes function point metrics as the primary tool for exploring the costs of the most expensive software problem in history. The book covers year 2000 costs in 30 countries and more than a dozen industries. The book also discusses "masking" as well as date field expansion, and hence covers the costs of windowing, bridging, encapsulation, etc. Because it is obvious that many companies will not achieve year 2000 compliance, the book explores the possible damages and recovery costs for the anticipated 1,700,000 U.S. software applications that will not be ready for the next century and hence will fail on January 1, 2000. The book explores the year 2000 impacts of information systems, systems software, military software, commercial software, and end-user applications. Data base repairs, litigation, hardware upgrades, and all other known cost elements are included. Paul Strassmann, former CIO of the Department of Defense, has written a foreword.

Kan, Stephen H.: *Metrics and Models in Software Quality Engineering*. Addison Wesley, Reading, MA; ISBN 0-201-63339-6; 1995; 344 pages.

This is a thoughtful overview of a number of metrics in the software quality domain. The book also covers topics of great importance to the software management community, such as the Baldrige Award, the ISO 9000-9004 standards, the Software Engineering Institute (SEI) capability maturity model (CMM), and total quality management (TQM). The chapters and sections are fairly complete, and give context and background information. A full reading of this book will transfer a considerable amount of useful information to the readers, and it covers many topics well enough to give fresh insight.

Some of the Quality Initiatives in Germany:

- Arbeitskreis Software-Qualität Franken e.V.

see: http://www.asqf.de

- Gesellschaft für Softwarequalitätssicherung GmbH

see: http://www.sqs.de

- Fraunhofer Institut Experimentelles Software Engineering (IESE)

  see: http://www.iese.fhg.de/

- Deutsche Informatik Akademie Seminare:

  - Strategien zur Verbesserung des Softwareentwicklungsprozesses (Dr. C. Ebert)
  - Methodisches Testen und Analysieren von Software (Dr. P. Liggesmeyer)
  - Software-Projektsteuerung für die Praxis (Metriken für Projektmanagement, Qualitäts- und Prozeßverbesserung) (Dr. C. Ebert)

  see: http://www.gi-ev.de/dia/

- Deutschsprachige Anwendergruppe für Softwaremetriken und Aufwandsschätzung (DASMA)

  see: http://home.t-online.de/erwin.loch/dasma_index.htm

- European TeleCASE Center (ETC) in Braunschweig

  see: http://www.dlr.de/bs_d.html

- Das Software Meßlabor (SMLAB) der Universität Magdeburg

  see: http://ivs.cs.uni-magdeburg.de/sw-eng/us/

- etc.

Emam, K.E.; Drouin, J.N.; Melo, W.: SPICE - The Theory and Practice of Software Process Improvement and Capability Determination

IEEE Computer Society Press, 1998 (486 p.)

This book is intended for current and potential users of the emerging International Standard on Software Process Assessment (SPA). Users may include organizations wanting to conduct process assessment internally or as a service to their customers, providers of models, methods, and tools compliant with the requirements of the emerging stand, educators wanting to teach

the principles of SPA as part of a university course, and researchers developing technology to support process assessments and then empirically evaluating them.

This book provides the first comprehensive coverage of the theory and practice of SPA as embodied in the current version of the ISO/IEC documents. The official title of these documents is ISO/IEC ODTR 15504, although they are also commonly known as the SPICE Version 2.00 documents. The main focus of the book is twofold: to help the reader understand the evolution of the SPA ideas that were developed between 1993 and 1997 and to provide guidance for interpreting and using the documents for assessment and subsequent improvement and/or capability determination.

The name SPICE—Software Process Improvement and Capability dEtermination—was given to this project as the development vehicle for the initial version of the SPA documents and as the management of the trials.

Whitmire, S.A.: Object-Oriented Design Measurement

John Wiley & Sons Inc., 1997 (452 p.)

"Once in a while a potentially classic book comes along—one that is built not only to last but also to have a lasting impact on its field. In this case, Scott Whitmire's text is likely to offer significant influences on the very foundational ideas underpinning object technology.

. . .

Scott also dares to attempt to create a formal object model based on category theory. Whilst we have all known that this would be a fruitful path to follow, no one until Scott has dared to accept this challenge in what I know is a difficult area for many of us not trained as pure mathematicians and logicians.

Not until he has undertaken this tour de force does Scott turn to his original aim: Of formally describing many of the object-oriented measures that have been proposed in the literature and are in use in industry today. He summarizes and transcends the empirical, theoretical, and mathematical work done before, integrating it into a whole that is both satisfying and valuable. At the same time, this is not a dry, theoretical treatise with no obvious utility. Scott is an industry user driven to investigate these areas from his industry needs."

cited from the foreword by Brian Henderson-Sellers

Lehner, F.; Dumke, R.; Abran, A.: Software Metrics - Research and Pracitce in Software Measurement

Gabler-Verlag, Wiesbaden, 1997 (232 p.) *ISBN: 3-8244-6518-3*

This book contains all presentations of the 1996 workshop of the GI-interest group on software metrics and of the Canadian Group (CIM) in September in Regensburg. It is a collection of theoretical studies in the field of software measurement as well as experience reports on the application of software metrics in Canadian, Austrian, Belgian and German companies and universities. Some of these papers and reports describe new software

measurement applications and paradigms for knowledge-based techniques, maintenance service evaluation, factor analysis discussions and neural-fuzzy applications. Others address the object-oriented paradigm and discuss the application of the Function Point approach to an object-oriented design method, the evaluation of the Java development environment, the analysis of quality and productivity improvements of object-oriented systems, as well as the definition of the metrics of class libraries. Other papers offer a different perspective, presenting a software measurement education system designed to help improve the lack of training in this field, for example, or they include experience reports about the implementation of measurement programs in industrial environment.

Poulin, J.S.: Measuring Software Reuse

Addison-Wesley, 1997 (195 p.)

With the techniques in this book, you will have the tools you need to design a far more effective reuse program, prove its bottom-line profitability, and promote software reuse within your organization. Measuring Software Reuse brings together all of the latest concepts, tools, and methods for software reuse metrics, presenting concrete quantitative techniques for accurately measuring the level of reuse in a software project and objectively evaluating its financial benefits.

Dumke, R.; Foltin, E.; Koeppe, R.; Winkler, A.: Softwarequalität durch Meßtools - Assessment, Messung und instrumentierte ISO 9000

Vieweg Publ., 1996 (223 p.) *ISBN 3-528-005527-8*

This book gives an overview about the software metrics tools for all phases of the software development process. The metrics tools are defined as CAME tools (Computer Assisted Software Measurement and Evaluation). The introduction describes the essential aspects of the software measurement. The description oft the CAME tools includes the cost estimation tools, Capability Maturity Model evaluation tools, metrics tool for software specification, design and code, and tools forthe software testing and maintenance (including network performance). Some tables helps to decision of choosing the useful tool integration for the software quality assurance process.

Zuse, H.: A Framework of Software Measurement

de Gruyter Publ., Berlin New York, 1997 (755 p.) *ISBN 3-11-015587-7*

This book describes a framework for software measurement from a theoretical, practical and educational view. The main idea is the application of the measurement theory on the area of software measurement.

The book is written in nine chapters and includes exercises for a teaching in software measurement. The chapters describe the software measurement aspect, the history of software measurement, the theoretical foundations from theoretical and practical view, especially the

object-oriented software measures, the discussion about the properties and validation, and helpful remarks for a successful application of software measures.

The book includes a CD ROM that include a demo tool for software measurement education based on more than thousand references and metrics.

EASE'98:

Empirical Assessment & Evaluation in Software Engineering,
30th March - 1st April, Staffordshire, U.K.
see: **http://keele.ac.uk/depts/cs/Announcements/conferences/ease98.html**

AQuIS'98:

Fourth International Conference on Achieving Quality in Software: Software Quality in the Communication Society,
30th March - 2nd April, Venice, Italy,
see: **http://www.iei.pi.cnr.it/AQUIS98**

SQM'98:

International Conference on Software Quality Management,
April 6-8, 1998, Amsterdam, Netherlands
contact: **kstonestct@cix.compulink.co.uk**

IFPUG'98, Spring:

International Function Point User Group Spring Conference,
April 5-9, 1998, Washington D.C., USA
see: **http://www.ifpug/org/**

QW'98:

Quality Week 1998,
May 26-29, San Fancisco, USA,
see: **http://www.soft.com/QualWeek/QW98**

FESMA'98:

First European Conference on Software Measurement,
May 6-8, 1998, Antwerp, Belgium
see: **http://www.kviv.be/ti/fesma.htm**

Linz'98:

Evaluation and Evaluation Research in Informations Systems,

[60] *Position Papers*

June 5, 1998, Linz, Austria,
contact: **haentschel@winie.uni-linz.ac.at**

IWSF'98:

International Workshop on Software Functional Size Measurement,
August 3-5, 1998, Mont-Tremblant, Canada
see: **http://saturne.info.uqam.ca/Labo_Recherche/lrgl.html**

IWSM'98:

8th International Workshop on Software Measurement,
September 17-18, Magdeburg, Germany,
see: **http://ivs.cs.uni-magdeburg.de/sw-eng/us**

IFPUG'98, Fall:

International Function Point User Group Fall Conference,
September 21-25, 1998, Orlando, USA
see: **http://www.ifpug/org/**

IASTED'98:

International Conference on Software Engineering,
October 28-31, Las Vegas, USA, see: **http://www.iasted.com**

ISSRE'98:

9th International Symposium on Software Reliability Engineering,
November 4-7, 1998, Paderborn, Germany,
contact: **belli@adt.uni-paderborn.de**

EuroSTAR'98:

6th European International Conference on Software Testing,
30th November - 4nd December, 1998, Munich, Germany
see: **email: EuroSTAR@aimware.com**

SEPG'99:

11th Software Engineering Process Group Conference,
March 8-11, 1999, Atlanta, USA
see: **http://www.sei.cmu.edu/products/sepg99/**

metrics themes are also discussed in the yearly **OOIS, ECOOP** *and* **ESEC** *conferences*

Other Information Sources and Related Topics

- **http://rbse.jsc.nasa.gov/virt-lib/soft-eng.html**
  Software Engineering Virtual Library in Houston

- **http://www.mccabe.com/**
  McCabe & Associates. Commercial site offering products and services for software developers (i. e. Y2K, Testing or Quality Assurance)

- **http://www.sei.cmu.edu/**
  Software Engineering Institute of the U. S. Department of Defence at Carnegie Mellon University. Main objective of the Institute is to identify and promote successful software development practices.
  Exhaustive list of publications available for download.

- **http://dxsting.cern.ch/sting/sting.html**
  Software Technology INterest Group at CERN: their WEB-service is currently limited (due to "various reconfigurations") to a list of links to other information sources.

- **http://www.spr.com/index.htm**
  Software Productivity Research, Capers Jones. A commercial site offering products and services mainly for software estimation and planning.

- **http://fdd.gsfc.nasa.gov/seltext.html**
  The Software Engineering Laboratory at NASA/Goddard Space Flight Center. Some documents on software product and process improvements and findings from studies are available for download.

- **http://www.qucis.queensu.ca/Software-Engineering/**
  This site hosts the World-Wide Web archives for the USENET usegroup comp.software-eng. Some links to other information sources are also provided.

- **http://www.esi.es/**
  The European Software Institute,Spain

- **http://saturne.info.uqam.ca/Labo_Recherche/lrgl.html**
  Software Engineering Management Research Laboratory at the University of Quebec, Montreal. Site offers research reports for download. One key focus area is the analysis and extension of the Function Point method.

- **http://www.SoftwareMetrics.com/**

Homepage of Longstreet Consulting. Offers products and services and some general information on Function Point Analysis.

- **http://www.utexas.edu/coe/sqi/**
  Software Quality Institute at the University of Texas at Austin. Offers comprehensive general information sources on software quality issues.

- **http://wwwtrese.cs.utwente.nl/~vdberg/thesis.htm**
  Klaas van den Berg: Software Measurement and Functional Programming (PhD thesis)

- **http://divcom.otago.ac.nz:800/com/infosci/smrl/home.htm**
  The Software Metrics Research Laboratory at the University of Otago (New Zealand).

- **http://ivs.cs.uni-magdeburg.de/sw-eng/us/**
  Homepage of the Software Measurement Laboratory at the University of Magdeburg.

- **http://www.cs.tu-berlin.de/~zuse/**
  Homepage of Dr. Horst Zuse

- **http://dec.bournemouth.ac.uk/ESERG/bibliography.html**
  Annotaded Bibliography on Object-Oriented Metrics

- **http://www.iso.ch/9000e/forum.html**
  The ISO 9000 Forum aims to facilitate communication between newcomers to Quality Management and those who, having already made the journey have experience to draw on and advice to share.

- **http://www.qa-inc.com/**
  Quality America, Inc's Home Page offers tools and services for quality improvement. Some articles for download are available.

- **http://www.quality.org/qc/**
  Exhaustive set of online quality resources, not limited to software quality issues

- **http://freedom.larc.nasa.gov/spqr/spqr.html**
  Software Productivity, Quality, and Reliability N-Team

News Groups

- **news:comp.software-eng**

- **news:comp.software.testing**

- **news:comp.software.measurement**

METRICS NEWS

VOLUME 3 **1998** **NUMBER 1**

CONTENTS

Editorial **3**

Call for Papers **5**

Position Papers **7**

Zuse, H.: *Measurement in Physics and Software Engineering - Part II* **7**

Jones, C.: *Rules of Thumb for Year 2000 and Euro-Currency Software Repairs* **13**

Ebert, C.: *The Software Engineering Charm - Navigating Between Craft and Science* **23**

Jones, C.: *Strengths and Weaknesses of Software Metrics* **35**

Dumke, R.: *Education in Software Measurement in the WWW* **45**

Jones, C.: *The Role of Function Point Metrics in the 21st Century* **47**

Initiatives **59**

New Books on Software Metrics **61**

Conferences addressing Metrics Issues **65**

Software Metrics in the World-Wide Web **67**

ISSN 1431-8008