How long have we all heard that the productivity of the average software engineer in the United States is 7 lines of code per day? That’s it – that’s all we know. That is the extent of publicly available productivity data. Furthermore, many software professionals and managers interpreted this productivity as meaning 7 lines of code per day during the coding or programming phase only. To any self-respecting programmer, that was just absurd and lacked any credibility.

Well, the fact of the matter is there is little other information available that everyone can access for purposes of benchmarking individual performance and estimating. Data does exist however, particularly as collected by software project management consultants from projects they have supported and observed. However, in general, because of the proprietary nature of the data, the data can not be publicly released.

We at the DACS are asked frequently to provide facts about various aspects of life-cycle performance within the software industry, with very few sources to go to to find the information. So when Donald Reifer’s original article “Let the Numbers Do the Talking” appeared in the March 2002 issue of CrossTalk, I felt we finally had a reliable source which we all could use to benchmark and estimate. For that I was very thankful.

Donald Reifer has updated that excellent 2002 article and has agreed to publish “Industry Software Cost, Quality and Productivity Benchmarks” in this issue of the DACS Software Tech News. This article first examines software productivity information, and characterizes productivity by application domain as well as compares productivity between the US and foreign regions. It next reports on cost data by application domain and language levels. Based on the most popular life-cycle models, it then reports on the distribution of effort and schedule by phase. Next, Mr. Reifer discusses typical software support costs, software quality information, and concludes by examining the trends in software improvements. So are we still at 7 source lines of code per day?

I have always found it very strange when others have referred to software testing as an art. As far as I know, I did not take any art classes at engineering school, and I feel I know a little about testing. James Whittaker’s article “Software Testing as an Art, a Craft and a Discipline” sets the record straight. Software testers are some of the most skilled people I know. I do not think of them as artists.

The next article titled “Independent Verification and Validation of Neural Networks – Developing Practitioner Assistance,” by Dr. Laura Pullum, Dr. Marjorie Darrah and Mr. Brian Taylor examines NASA’s IV&V Facility’s initiative to develop a software assurance methodology for neural networks. The “black box” and highly mathematical nature of neural nets requires a different V&V approach versus traditional software V&V.

Software rejuvenation was not a discipline I was aware of before I read Larry Bernstein’s and Chandra Kintala’s fine article “Software Rejuvenation and Self-Healing”. Rejuvenation is a strategy for improving the reliability and trustworthiness of a running software system. A related article (“System, Heal Thyself” by Matt Hamblin) on self-aware computing can be found on the Computerworld website (http://www.computerworld.com/Quicklink#43636) in which the author identifies self-healing systems as an emerging technology.

As usual, we encourage your feedback on this and every issue of Software Tech News.
Industry Software Cost, Quality and Productivity Benchmarks

Donald J. Reifer, Reifer Consultants, Inc.

Abstract: This article provides software cost, quality and productivity benchmarks in twelve application-oriented domains that readers can use to determine how well their organizations are performing relative to industry averages. In addition to answering common questions raised relative to these benchmarks, the article summarizes the return on investments firms are realizing as they try to harness new technology for a variety of reasons.

Introduction

For years, I have heard my friends complain about the lack of software cost and productivity benchmarks in the open literature. They wanted to use the benchmarks to identify how they stacked up with others within their industry in terms of their performance. In March 2002, I tried to do something about this lack of data by publishing an article which put cost and productivity numbers that I had been compiling for more than two decades into the public domain along with a call for others to join me in publishing numbers.1 Unfortunately, nobody has answered the call. As a consequence, most cost and productivity numbers continue to remain confidential.

This paper tries to rectify the current state of affairs by providing readers with a revision of my original paper. I have updated my cost and productivity numbers and added quality benchmarks. The new data is important because I believe that cost and productivity should be measured from a quality point of view. It seems counterproductive to me to increase productivity by cutting quality. Yet, there are firms that will do anything to improve their numbers.

Needless to say, numbers are important. Besides giving you insight into what the costs and benefits are for a given alternative, numbers can be used to establish industry benchmarks that firms can use to compare their performance with others. When push comes to shove, such numbers are what really matters when you are seeking management approval for funds. “If there isn’t some benefit, why spend money?” management asks. Questions like these seem to permeate the atmosphere when management is asked to spend money on software improvements;

● What are the costs and what are the benefits?
● What are the returns on this investment?
● Why should I spend money on this investment rather than on alternatives?
● What would happen if I did nothing (actually my boss’s favorite question)?

Getting management to spend money isn’t easy. These same managers will use rules of thumb that they have developed over the years to determine if the numbers you are pitching are reasonable. For example, they might surprise you by asking the following question if your numbers don’t make sense: “How can you suggest that we will more than triple our productivity if your proposal is accepted?” You need to be prepared to respond to such a question. Having benchmark data that you can use to answer their question could make or break your proposal.

While helpful, the use of such benchmarks is often fraught with danger. This is primarily because people take numbers and use them out of context to justify what they are after. Often, management sees right through such tactics and turns down their request. To avoid failing the “numbers must make sense test”, they must be thoroughly defined and be used properly.

Looking at Software Productivity Numbers

Software productivity refers to the ability of an organization to generate outputs (software systems, documentation, etc.) using the inputs or resources it has at its disposal (people, money, equipment, tools, etc.). Using this definition, an organization can increase its productivity by either focusing on reducing the input or increasing the output.

To improve productivity using my definition, organizations could focus on either the input or output strategy. For an input-based productivity improvement strategy, they would focus on increasing workforce productivity through efficiencies gained by inserting better methods, tools, processes, facilities and equipment into the production mix. In contrast, an output-based productivity improvement strategy would place emphasis on increasing the amount of output generated under equivalent circumstances by using components, architecture-centric reuse and product line tactics. Both strategies strive to produce more output using less input.

Within many industries, productivity is commonly expressed as either equivalent source lines of code (ESLOC)/staff month (SM) or as function points (FP)/SM. While other measures may be used, these two tend to predominate, especially for medium to large scale projects. Of course, the measures ESLOC, FP and SM must

be carefully scoped and consistently defined for these metrics to convey a common meaning (see Notes under Tables for applicable definitions). In addition, the many factors that cause these measures to vary must also be identified. These measures, called cost drivers, must be normalized when defining the terms. Because my firm’s databases are primarily ESLOC-based, I use this metric as the basis for my analysis. For those interested in how we handle other measures, we backfire FP data to convert them to ESLOC using the language conversion factors supplied from the International Function Point Users Group (e.g., one FP is expressed as so many lines of C or Java using such backfiring tables).

Table 1 summarizes the results of our analysis of twelve application domains for which my firm has collected data that seem to be of interest to readers. The numbers in Table 1 were derived by taking a 600 project subset of our 2,000+ project experience database and performing statistical analysis. In addition, there are no foreign projects in our database to distort conclusions. The average productivity in our foreign databases for Europe (Belgium, England, Finland, France, Germany, Ireland, Italy, Spain and Sweden) and Asia (India, Korea, Japan and Singapore) are summarized in Table 2. This Table shows how the average productivity for these regions compares to the United States and tries to explain why they are different. Summaries for Australia, Canada and South American numbers are not included in our analysis because the number of completed projects in our databases is too small to be statistically significant. The box under each of the Tables provides notes about the contents. Data were validated using standard statistical means. When anomalies in the data were observed, site visits were made to address concerns. An acronym list is also provided below to define terms used in the Tables.

What about Cost?

While cost and productivity are related, they should be considered separately. The reason for this is simple to understand, cost and productivity concerns focus on different factors. For example, cost is very sensitive to labor and burden rate factors. The numbers can be easily manipulated by varying rates. In contrast, organizations trying to improve productivity focus on efficiencies that arise from automation, teamwork and improvement of their process infrastructure. For productivity, the focus is on getting more output with a standard input.

When analyzing our database, assuming rates are normalized, we find that software cost tends to be related to both labor costs and language level. I used $15,000 as the standard cost for a staff-month (SM) of effort to reflect the average cost of labor across industries exclusive of profit and general and administrative charges, as applicable. Of course, labor rates vary greatly by industry and $15,000 may not be enough in some (aerospace, telecommunications, etc.) and is too much in others (data processing, web developments, etc.). Language levels are a function of the methods, tools, and language technology used to develop the software product. For simplicity, we define the following three language levels for the purpose of expressing relative cost:4

- Third Generation Languages (3GL) – A high level procedural programming language like Basic, C, C++, Java and/or Pascal designed to facilitate the development of software products by programming professionals.
- Fourth Generation Languages (4GL) – A programming language close to human languages that makes computing power available to non-programmers and is employed typically by users to access databases, generate scripts, support financial modeling and/or generate reports (Access, PERL, Sequel, etc.).
- Fifth Generation Languages (5GL) – A programming language that incorporates the concepts of knowledge-based systems, visualization, animation and/or natural language processing to facilitate the generation of software by a variety of professionals, some of which are outside of the programming arena.

Table 3 shows the dollar cost per ESLOC by application domain that we have developed to serve as benchmarks. The cost numbers vary widely and are sensitive to size, team experience and many factors. Although the Table shows that the cost/ESLOC goes down as language level goes up, this is not true. Firms using fourth and fifth level languages typically use them as part of a mix. The effects illustrated in the Table are therefore the result of using languages from more than one language level, each selected to perform the job it was designed to accomplish.

How is this Effort Distributed?

Distribution of effort and schedule is a function of the life-cycle paradigm or model selected for the project. For

**Table 1: Software Productivity (ESLOC/SM) by Application Domains**

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>Number Projects</th>
<th>Size Range (KESLOC)</th>
<th>Avg. Productivity (ESLOC/SM)</th>
<th>Range (ESLOC/SM)</th>
<th>Example Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automation</td>
<td>55</td>
<td>25 to 650</td>
<td>245</td>
<td>120 to 445</td>
<td>Factory automation</td>
</tr>
<tr>
<td>Banking</td>
<td>30</td>
<td>55 to 1,000</td>
<td>270</td>
<td>155 to 550</td>
<td>Loan processing, ATM</td>
</tr>
<tr>
<td>Command &amp; Control</td>
<td>45</td>
<td>35 to 4,500</td>
<td>225</td>
<td>95 to 350</td>
<td>Command centers</td>
</tr>
<tr>
<td>Data Processing</td>
<td>35</td>
<td>20 to 780</td>
<td>330</td>
<td>165 to 500</td>
<td>DB-intensive systems</td>
</tr>
<tr>
<td>Environment/Tools</td>
<td>75</td>
<td>15 to 1,200</td>
<td>260</td>
<td>143 to 630</td>
<td>CASE, compilers, etc.</td>
</tr>
<tr>
<td>Military — All</td>
<td>125</td>
<td>15 to 2,125</td>
<td>145</td>
<td>45 to 300</td>
<td>See subcategories</td>
</tr>
<tr>
<td>— Airborne</td>
<td>40</td>
<td>20 to 1,350</td>
<td>105</td>
<td>65 to 250</td>
<td>Embedded sensors</td>
</tr>
<tr>
<td>— Ground</td>
<td>52</td>
<td>25 to 2,125</td>
<td>195</td>
<td>80 to 300</td>
<td>Combat center</td>
</tr>
<tr>
<td>— Missile</td>
<td>15</td>
<td>22 to 125</td>
<td>85</td>
<td>52 to 175</td>
<td>GNC system</td>
</tr>
<tr>
<td>— Space</td>
<td>18</td>
<td>15 to 465</td>
<td>90</td>
<td>45 to 175</td>
<td>Attitude control system</td>
</tr>
<tr>
<td>Scientific</td>
<td>35</td>
<td>28 to 790</td>
<td>195</td>
<td>130 to 360</td>
<td>Seismic processing</td>
</tr>
<tr>
<td>Telecommunications</td>
<td>50</td>
<td>15 to 1,800</td>
<td>250</td>
<td>175 to 440</td>
<td>Digital switches</td>
</tr>
<tr>
<td>Test</td>
<td>35</td>
<td>20 to 800</td>
<td>210</td>
<td>100 to 440</td>
<td>Test equipment, devices</td>
</tr>
<tr>
<td>Trainers/Simulations</td>
<td>25</td>
<td>200 to 900</td>
<td>225</td>
<td>143 to 780</td>
<td>Virtual reality simulator</td>
</tr>
<tr>
<td>Web Business</td>
<td>65</td>
<td>10 to 270</td>
<td>275</td>
<td>190 to 985</td>
<td>Client/server sites</td>
</tr>
<tr>
<td>Other</td>
<td>25</td>
<td>5 to 1,000</td>
<td>182</td>
<td>65 to 481</td>
<td>All others</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td><strong>600</strong></td>
<td><strong>10 to 4,500</strong></td>
<td><strong>45 to 985</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes for Table 1**

- The 600 projects are the most recent projects taken from my database of more than 1,800 projects. These projects were completed within the last seven years by any of 40 organizations (each organization is kept anonymous due to the confidentiality of the data). A project is defined as the delivery of software to system integration. Projects include builds and products that are delivered externally, not internally. Both delivery of a product to market and a build to integration fit this definition.
- The scope of all projects starts with software requirements analysis and finishes with completion of software testing.
  - For military systems, the scope extends from software requirements review until handoff to the system test bed for hardware and software integration and testing.
  - For banking and ADP systems, the scope extends from approval of project startup until sell-off.
  - For web systems, the scope extends from product conception to customer sell-off.
- Projects employ a variety of methods and practices ranging from simple to sophisticated.
- Analysis includes all chargeable engineering, management and support labor in the numbers.
  - It includes programming, task management and support personnel who normally charge to project.
  - It does not include quality assurance, system or operational test, and beta test personnel.
- The average number of hours per staff month was 152 (takes holidays, vacation, etc. into account).
- SLOC is defined by Florac and Carleton\(^2\) to be logical source line of code using the conventions published by the Software Engineering Institute in 1993. ESLOC are defined by Boehm to take into account reworked and reused code.\(^3\) All SLOC counts adhere to the SEI counting conventions.
- Function point sizes are defined using current International Function Point Users Group (IFPUG) standards, http://www.ifpug.org/.
- Function point sizes were converted to SLOC using backfiring factors published by IFPUG in 2000, as available on their web site.
- Projects used many different life cycle models and methodologies. For example, web projects typically followed a Rapid Application Development process and used lightweight methods, while military projects used more classical processes and methods. Commercial projects used object-oriented methodology predominantly, while military projects used a broader a mix of conventional and object-oriented approaches.
- Projects used a variety of different languages. For example, web projects employed Java, Perl and Visual C while military projects used predominantly C/C++.


Industry Software Cost
Continued from page 4.

the following three popular life-cycle models, the allocations of effort and duration are shown in Tables 4, 5 and 6:

- Waterfall,
- Model-Based Software Engineering (MBASE),
- Rational Unified Process.

Formats used for the Tables vary widely because the numbers are based on slightly modified public references. In some cases, the allocations of effort and duration are shown as ranges. In other cases, they are normalized so that they sum to 100 percent. In yet other cases, the sums are not normalized and therefore are equal to more than 100 percent.

Tables 4 and 6 clearly show the effort and duration to perform a particular activity excluding requirements synthesis (inception) and system test tasks (transition). Table 6 reflects the effort and duration to perform tasks that are part of the Rational Unified Process (RUP). It is interesting to note that both the MBASE and RUP paradigms can be tailored to support agile methods and extreme programming practices. Do not infer that it takes MBASE 18 percent longer than RUP to do equivalent tasks from these Tables.

That is not the case because different life cycles embody different activities that make comparisons between them difficult and misleading. If you are interested in more detailed comparisons between life cycles, see the approaches outlined in [6], which provides the most complete coverage that I have seen to date.

These allocation tables are interesting because they tell us that software people do much more work than what is normally considered by most to be software development. This workload typically starts with requirements analysis and ends with software integration and test. They get involved in requirements analysis (averages 7 percent additional effort and 16 to 24 percent more time) and system integration and test support (12 percent more effort and continues on page 8

---

Table 2: Productivity Comparisons between U.S., Europe and Asia by Application Domain

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>Average Productivity (ESLOC/SM)</th>
<th>Key Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>United States</td>
<td>Europe</td>
</tr>
<tr>
<td>Automation</td>
<td>245</td>
<td>200</td>
</tr>
<tr>
<td>Banking</td>
<td>270</td>
<td>260</td>
</tr>
<tr>
<td>Command &amp; Control</td>
<td>225</td>
<td>-</td>
</tr>
<tr>
<td>Data Processing</td>
<td>330</td>
<td>310</td>
</tr>
<tr>
<td>Military — All</td>
<td>145</td>
<td>-</td>
</tr>
<tr>
<td>— Airborne</td>
<td>105</td>
<td>-</td>
</tr>
<tr>
<td>— Ground</td>
<td>195</td>
<td>174</td>
</tr>
<tr>
<td>— Missile</td>
<td>85</td>
<td>-</td>
</tr>
<tr>
<td>— Space</td>
<td>90</td>
<td>85</td>
</tr>
<tr>
<td>Scientific</td>
<td>195</td>
<td>-</td>
</tr>
<tr>
<td>Telecommunications</td>
<td>250</td>
<td>240</td>
</tr>
<tr>
<td>Test</td>
<td>210</td>
<td>210</td>
</tr>
<tr>
<td>Trainers/Simulations</td>
<td>225</td>
<td>-</td>
</tr>
<tr>
<td>Web Business</td>
<td>275</td>
<td>225</td>
</tr>
<tr>
<td>Other</td>
<td>182</td>
<td>200</td>
</tr>
</tbody>
</table>

Note: The datasets involved in Table 2 for Asia and Europe are much smaller than those compiled in the United States.
### Table 3: Software Cost ($/ESLOC) by Predominate Language level by Application Domain

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>3GL</th>
<th>4GL</th>
<th>5GL</th>
<th>Norm</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automation</td>
<td>50</td>
<td>*</td>
<td>*</td>
<td>50</td>
<td>Must implement ladder nets.</td>
</tr>
<tr>
<td>Banking</td>
<td>40</td>
<td>30</td>
<td>*</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Command &amp; Control</td>
<td>85</td>
<td>*</td>
<td>*</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>Data Processing</td>
<td>40</td>
<td>30</td>
<td>*</td>
<td>35</td>
<td>Many have moved from COBOL to Java and visual languages</td>
</tr>
<tr>
<td>Environment/Tools</td>
<td>40</td>
<td>25</td>
<td>*</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Military — All</td>
<td>145</td>
<td>*</td>
<td>*</td>
<td>145</td>
<td>While most still use 3G, many have moved to object-oriented languages, Ada on the decline.</td>
</tr>
<tr>
<td>— Airborne</td>
<td>200</td>
<td>*</td>
<td>*</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>— Ground</td>
<td>90</td>
<td>*</td>
<td>*</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>— Missile</td>
<td>225</td>
<td>*</td>
<td>*</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>— Space</td>
<td>210</td>
<td>*</td>
<td>*</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>Scientific</td>
<td>90</td>
<td>*</td>
<td>*</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>Telecommunications</td>
<td>75</td>
<td>50</td>
<td>*</td>
<td>75</td>
<td>Most continue to use C/C++ and Unix</td>
</tr>
<tr>
<td>Test</td>
<td>50</td>
<td>*</td>
<td>*</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Trainers/Simulations</td>
<td>65</td>
<td>*</td>
<td>*</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>Web Business</td>
<td>50</td>
<td>35</td>
<td>25</td>
<td>45</td>
<td>Most use Java, HTML, PERL, etc.</td>
</tr>
</tbody>
</table>

### Notes for Table 3

- Dollars used to determine cost are assumed to be constant year 2003 dollars.
- The cost assumed per staff month (SM) of effort of $15,000 assumes an average labor mix and includes all direct labor costs plus applicable overhead. The mix assumed that the average staff experience across the team in the application domain was three to five years. It assumed that the staff was moderately skilled and experienced with the application and languages, methods and tools used to develop the products.
- The scope of the estimate extended from software requirements analysis through completion of software integration and test.
- The scope of the labor charges included all of the professionals directly involved in software development. The people involved included software engineers, programmers, task managers and support personnel.
- Many of the organizations reporting were trying to exploit commercial off-the-shelf packages and legacy systems to reduce the volume of work involved.
- The military organizations reporting were all CMM Level 3 or greater, while most commercial firms were not. However, the processes that these organizations used were mostly ISO certified.
- Finally, most projects used more than one language on their projects. The Table shows the cost per SLOC for the predominate language. However, the reader is cautioned that this metric may be misleading when used out of context.

---

### Table 4: Waterfall Paradigm Effort and Duration Allocations

<table>
<thead>
<tr>
<th>Phase (end points)</th>
<th>Effort %</th>
<th>Duration %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plans and Requirements (SDR to SRR)</td>
<td>7 (2 to 15)</td>
<td>16 to 24 (2 to 30)</td>
</tr>
<tr>
<td>Product Design (SRR to PDR)</td>
<td>17</td>
<td>24 to 28</td>
</tr>
<tr>
<td>Software Development (PDR to UTC)</td>
<td>52 to 64</td>
<td>40 to 56</td>
</tr>
<tr>
<td>Software Integration and Test (UTC to STR)</td>
<td>19 to 31</td>
<td>20 to 32</td>
</tr>
<tr>
<td>Transition (STR to SAR)</td>
<td>12 (0 to 30)</td>
<td>12.5 (0 to 20)</td>
</tr>
<tr>
<td>Total</td>
<td>107 to 131</td>
<td>116 to 155</td>
</tr>
</tbody>
</table>

*continues on page 8*
To use the percentages in this Table, you would decrease the productivity or increase the cost appropriately. For example, my life cycle with requirements analysis, not synthesis. This means that the software contribution to the IPT in a military project allocating requirements to software would not be within the current scope. This is pretty standard practice in large military software organizations when systems engineering is tasked to allocate requirements to hardware, software and procedures using a systems specification and operational concept document as their basis. To take the nominal 10% effort into account, you could divide the productivity figure or inflate the cost figure in my Tables accordingly.

It should be noted than many of these additional efforts are application domain specific. For example, Independent Verification and Validation (IV&V) and System Engineering Technical Direction (SE/TD) represent activities conducted in support of military contracts. Because independent teams get involved in oversight and testing, additional work is required on the part of the development contractor. Many times this is true because these third-party contractors feel that they must find something wrong, else their worth can not be justified. Yet, such teams add value when properly staffed and directed towards contributing to release of the software product. The trick in keeping IV&V and SE/TD costs down is to make sure that efforts are directed at making the product better, not finding problems just for the sake of justifying one’s existence.

For commercial jobs, alpha and beta testing requires analogous additional support. Development staff must be allocated to address problems identified by testers and an elaborate test management scheme must be employed by the developers to avoid reworking fixes that have already been continued on page 16

### Table 5: MBase Paradigm Effort and Duration Allocations

<table>
<thead>
<tr>
<th>Phase (end points)</th>
<th>Effort %</th>
<th>Duration %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception (IRR to LCO)</td>
<td>6 (2 to 15)</td>
<td>12.5 (2 to 30)</td>
</tr>
<tr>
<td>Elaboration (LCO to LCA)</td>
<td>24 (20 to 28)</td>
<td>37.5 (33 to 42)</td>
</tr>
<tr>
<td>Construction (LCA to IOC)</td>
<td>76 (72 to 80)</td>
<td>62.5 (58 to 67)</td>
</tr>
<tr>
<td>Transition (IOC to PRR)</td>
<td>12 (0 to 20)</td>
<td>12.5 (0 to 20)</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td><strong>118</strong></td>
<td><strong>125</strong></td>
</tr>
</tbody>
</table>

### Table 6: Rational Unified Process (RUP) Effort and Duration Allocations

<table>
<thead>
<tr>
<th>Phase (end points)</th>
<th>Effort %</th>
<th>Duration %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception (IRR to LCO)</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Elaboration (LCO to LCA)</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>Construction (LCA to IOC)</td>
<td>65</td>
<td>50</td>
</tr>
<tr>
<td>Transition (IOC to PRR)</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td><strong>100</strong></td>
<td><strong>100</strong></td>
</tr>
</tbody>
</table>
Software Testing as an Art, a Craft and a Discipline

James A. Whittaker, Professor of Computer Science, Florida Tech

The first book on software testing set the tone for software testers and software testing careers. The title of that book The Art of Software Testing identified our discipline as a collection of artists applying their creativity to software quality. Practitioners of software testing and quality assurance have been sold short by such a label.

Artists indeed! Software testing is a far cry from those endeavors that most people accept as art: painting, sculpture, music, literature, drama and dance. In my mind, this is an unsatisfying comparison given that my training as a tester has been more engineering than art. My success as a tester has everything to do with my engineering abilities and little to do with my artistic penchant.

Certainly, I’ll agree that, like artists, software testers need to be creative, but art implies skill without training. Most virtuoso artists were born to the task and those of us unlucky enough to have no artistic talent are unlikely to develop such skill despite a lifetime of practice.

I also understand that two authors attempted to copyright the title The Craft of Software Testing, acknowledging Myers’ title and also implying a growth of the discipline from art to craft. This too sells testers far short of the difficulty of their calling. Indeed, the idea of software testing as a craft is equally unsettling as calling it an art.

Craftsmen are carpenters, plumbers, masons and landscape designers. Crafts are exemplified by a lack of a real knowledge base. Most craftsmen learn on the job and mastery of their craft is a given as long as they have the drive to practice. Crafts are two parts dexterity and only one part skill. Indeed, carpenters have no need to understand the biology of trees, only to skillfully mold wood into beautiful and useful things.

Testing as arts or crafts doesn’t begin to describe what we do; and I’ll start a fight with anyone who attempts to call it arts and crafts!

I suggest the most fitting title for a book on software testing would be The Discipline of Software Testing. I would argue that discipline better defines what we do as testers and provides us with a useful model on which to pattern our training and our careers. Indeed, this is the best reason to call it a discipline: by studying other disciplines, we gain more insight into testing than using the analogies of arts or crafts.

A discipline is a branch of knowledge or learning. Mastery of a discipline is achieved through training, not practice. Training is different than practice. Practice requires doing the same thing over and over again, the key being repetition. One can practice throwing a ball for example and even though “practice makes perfect”, simply throwing a ball will not make you a major league pitcher, becoming that good requires training.

Training is much more than just practice. Training means understanding every nuance of your discipline. A pitcher trains by building his muscles so that maximum force can be released when throwing a ball. A pitcher trains by studying the dynamics of the mound, where to land his foot for maximum effect on any given pitch and how to make use of his much stronger leg muscles to propel the ball faster.

A pitcher trains by learning how to effectively use body language to intimidate batters and runners. A pitcher trains by learning to juggle, to dance and to do yoga. A pitcher who trains to be at the top of his game does many things that have nothing to do with throwing a ball and everything to do with making himself a better ball thrower. This is why Hollywood’s “karate kid” waxed cars and balanced on fence posts; he wasn’t practicing to fight, he was training to be a better fighter.

Treating software testing as a discipline is a more useful analog than treating it as an art or a craft. We are not artists whose brains are wired at birth to excel in quality assurance. We are not craftsmen who perfect their skill with on-the-job practice. If we are, then it is likely that full mastery of the discipline of software testing will elude us. We may become good, indeed quite good, but still fall short of achieving black belt—dare I say Jedi?—status. Mastery of software testing requires discipline and training.

A software testing training regime should promote understanding of fundamentals. I suggest three specific areas of pursuit to guide anyone’s training:

First and foremost, master software testers should understand software. What can software do? What external resources does it use to do it? What are its major behaviors? How does it interact with its environment? The answers to these questions have nothing to do with practice and everything to do with training. One could practice for years and not gain such understanding.

Software works in an environment best exemplified by the diagram of Figure 1 [2, 3]. This diagram shows four major categories of software users, i.e., entities within an application’s environment that are capable of sending the application input or consuming its output. It is interesting to note that of the four major categories of users, continues on page 10
only one is visible to the human tester’s eye: the user interface. The interfaces to the kernel, the files system and other software components happen without scrutiny. Without understanding these interfaces, testers are taking into account only a very small percentage of the total inputs to their software. By paying attention only to the visible user interface, we are limiting what bugs we can find and what behaviors we can force.

Take as an example the scenario of a full hard drive. How do we test this situation? Inputs through the user interface will never force the code to handle the case of a full hard drive. This scenario can only be tested by controlling the file system interface. Specifically we need to force the file system to indicate to the application that the disk is full. Controlling the UI is only one part of the solution.

Understanding the environment in which your application works is a non-trivial endeavor that all the practice in the world will not help you accomplish. Understanding the interfaces that your application possesses and establishing the ability to test them requires discipline and training. This is not a task for artists and craftspeople.

Second, master software testers should understand software faults. How do developers create faults? Are some coding practices or programming languages especially prone to certain types of faults? Are certain faults more likely for certain types of software behavior? How do specific faults manifest themselves as failures?

There are many different types of faults that testers must study and this forum is too limited to describe them all. For a good start see [2]. However, consider default values for data variables as an example. For every variable used in a program, the variable must be first declared and then given an initial value. If either of these steps is skipped then a fault exists for testers to look for. Failure to declare a variable (as is the case with languages that allow for implicit variable declaration) can cause a single value to be stored in multiple variables. Failure to initialize a variable means that when a variable is used its value is unpredictable. In either case, the software will fail eventually. The trick for the tester is to be able to force the application to fail and then be able to recognize that it has failed.

Testers must understand what software faults are commonly made and be able to identify the potential for a fault to exist in any given feature. This is not an art; this is not a craft. The ability to do this is based on understanding the very nature of that which we test: software and the faults it may contain. Just as construction engineers understand the potential faults in a large building project, so must we understand the potential faults that may be introduced in a large software project.

Third, master software testers should understand software failure. How and why does software fail? Are there symptoms of software failure that give us clues to the health of an application? Are some features systemically problematic? How does one drive certain features to failure?

Recognizing a failure is arguably the most important skill that a tester can possess. After all, if the fault manifests but we fail to notice the failure symptoms then it is unlikely that the bug will get fixed. The problem here relates back to understanding Figure 1. Only some of the symptoms of failure manifest at the user interface level (where the failure is easily seen by a human tester). The others are buried in the file system, kernel calls and API calls to other components. These interfaces are not visible to the human eye and require specialized tools to properly analyze.

Understanding software, faults and failures is the first step to treating software testing as a discipline. Treating software as a discipline is the first step toward mastering software quality. And there is more, always more to learn. Discipline is a lifelong pursuit. If you trick yourself into thinking you have all the answers, then mastery will elude you. But training builds knowledge so the pursuit itself is worthwhile whether or not you ever reach the summit.

References

Although several collections of testing papers were published as books before Myers’ 1979 work [1], his was the first book to be written from scratch as a software testing text.

About the Author
James A. Whittaker is a professor of computer science at the Florida Insti-

continues on page 11
Independent Verification and Validation of Neural Networks — Developing Practitioner Assistance

Dr. Laura L. Pullum, Dr. Marjorie A. Darrah, and Mr. Brian J. Taylor,
Institute for Scientific Research, Inc.

Introduction

Neural networks represent a class of systems that do not fit into the current paradigms of software development and certification. Instead of being programmed, a learning algorithm “teaches” a neural network using a set of data. Often, because of the complex mathematical routines and nature of the training data, the neural network is considered a “black box” and its response may not be predictable.

In most instances, traditional testing techniques prove adequate for the acceptance of a neural network system. However, in more complex, safety- and mission-critical systems, the standard neural network training-testing approach is not able to provide a reliable method for certification. The use of artificial neural networks within NASA applications is expected to increase over the next few decades. Currently, there are over 20 NASA funded activities that use neural network technology, some of which are mission- and safety-critical. Verifying correct operation of neural networks within NASA projects, such as autonomous mission control agents and adaptive flight controllers, or within nuclear engineering applications, such as safety assessors and reactor controllers, requires a rigorous verification and validation (V&V) approach.

This V&V challenge is further compounded by adaptive neural network systems; ones that modify themselves, or “learn,” during operation. These systems continue to evolve during operation, for better or for worse. Traditional software assurance methods fail to account for systems that change after deployment. Furthermore, no overall standard exists that addresses a comprehensive V&V process specifically for neural networks. Although techniques do exist that apply to the V&V of neural networks, many are still underdeveloped or not sufficiently tested.

As the facility responsible for improving software safety, reliability, and quality of programs and missions, the NASA Independent Verification & Validation (IV&V) Facility will be increasingly challenged to certify and evaluate software systems that contain neural network technologies. To prepare for this imminent need, the NASA IV&V Facility has funded an initiative for the IV&V of neural networks, the goal of which is to develop a new software assurance methodology specifically for neural networks.

The resulting methodology will incorporate state-of-the-art practices in the V&V of neural networks, along with the experiences and knowledge from work with intelligent flight control systems.

The methodology addresses each life cycle process and is designed so that IV&V practitioners may apply this methodology at various stages within the project lifecycle.

continues on page 12

Software Testing as an Art

Continued from page 10.

Dr. Andrew W. Sementa, Ph.D., is a professor of computer science at the Institute of Technology. He earned his Ph.D. in computer science from the University of Tennessee in 1992. His research interests are software testing, software security, software vulnerability testing and anti cyber warfare technology. He is the author of How to Break Software, How to Break Software Security (with Hugh Thompson) and over 50 peer-reviewed papers on software development and computer security. He holds patents on various inventions in software testing and defensive security applications and has attracted millions in funding, sponsorship and license agreements while a professor at Florida Tech. He also has served as a testing and security consultant for Microsoft, IBM, Rational and many more US companies. In 2001 he was appointed to Microsoft’s Trustworthy Computing Academic Advisory Board and was named a “Top Scholar” by the editors of the Journal of Systems and Software based on his research publications in software engineering. His research team at Florida Tech is known for its testing technologies and tools, which include the highly acclaimed runtime fault injection tool Holodeck. His research group is also well known for their development of exploits against software security, including cracking encryption, passwords and infiltrating protected networks via novel attacks against software defenses.
Approach

The methodology will be written using the IEEE Standard for Software Verification and Validation, IEEE Std. 1012-1998 (IEEE 1012 1998) as a base document, with the intent of incorporating the methodology as a supplemental procedure. The methodology will provide guidance to the practitioner performing IV&V on neural network systems.

One of the first activities in this effort was the collection and evaluation of existing neural network V&V techniques. This resulted in a compilation and description of the state-of-the-art and -practice in V&V of neural networks, reported in (ISR 2002). A subsequent task involved conducting extensive research on V&V of neural network techniques that are more applicable to the independent V&V of neural networks, such as formal methods, stability analysis, run-time monitoring, testing, visualization, failures modes and effects analysis (FMEA), risk analysis, automated neural network selection, and neural network design verification. Some of this research is continuing, but the completed research is detailed in (ISR 2003a, Taylor 2003, Smith 2003, Darrah 2004). The IEEE 1012 standard for software V&V was concurrently reviewed, noting those tasks in which additional guidance would be required for application to neural network systems (ISR 2003b).

The next steps in the research for this effort will integrate new techniques with past experiences to develop practitioner guidance and associated training materials. To develop a methodology for the IV&V of neural networks we will incorporate the research findings on techniques to complement an accepted industry standard, IEEE 1012. To test the methodology, we will apply it to the Intelligent Flight Control\(^2\) (IFC) system’s first generation flight control concept designed to identify aircraft stability and control characteristics using neural networks, and use this information to optimize aircraft performance in both normal and simulated failure conditions. Ultimately we will develop training materials to assist the practitioner in the use of the methodology.

Related Efforts in IV&V for Neural Networks

Significant applied research has been conducted that addresses the V&V of neural networks. The results of NASA Dryden Flight Research Center (DFRC) and the NASA Ames Research Center (ARC) efforts, documented in “Verification and Validation of Neural Networks for Aerospace Systems” (Mackall 2002), focus on the V&V of pre-trained neural networks. The “V&V of Advanced Systems at NASA” (Nelson 2002) provides insight into neural network verification problems. The “Software Verification and Validation Plan for the Airborne Research Test System II Intelligent Flight Control Program” (ISR 2000) outlines some of the V&V processes that were performed on the IFC system (IFCS) neural networks to support that project.

\(^2\) The IFC project is developing a real-time adaptable flight control system utilizing neural networks. This project is...
The work introduced here combines current research and application results into a methodology that can be applied by an IV&V practitioner faced with the task of verifying and validating a system containing neural networks. This research effort includes the partnership of the Institute for Scientific Research, Inc. (ISR) scientists and engineers with researchers at West Virginia University and NASA IV&V, as well as association with researchers at NASA ARC and NASA DFRC through the IFCS program. To date, this is the largest working group dedicated to the V&V of neural network systems.

Techniques Developed
The table on page 12 presents information on the techniques developed to date and planned for investigation in 2004.

**Application to Practitioner**
As stated earlier, the intention of this effort is to provide guidance in the evaluation of neural networks for the IV&V practitioner, whether at NASA, other government agencies, or in industry. The methodology will take into consideration each process, activity, and task that the IEEE 1012 uses for traditional software and provide additional guidance as required for neural network V&V. This guidance will be provided in the style in which (ISO/IEC 12207 1998) provides guidance.

For an example, refer to the IEEE 1012 (IEEE 1012 1998) excerpts provided below.

For a neural network system, the guidance might include information such as that described below.

Guidance will be developed from the results of the research and will likely be more directed than the above example (from our early results). The guidance will be provided for each task, activity, and process within each life cycle phase.

**Conclusions**
The result of this effort will be a complete methodology for the IV&V of neural network systems that addresses the entire software life-cycle, is compatible with existing software standards, incorporates new technologies and methods research, and includes materials to train IV&V practitioners. This methodology will be made available to government and industry users of neural network technology with the intention that with the methodology, neural networks can be used more widely, verified and validated more completely, and used in more trusted and dependable systems.

**About the Authors**
Laura L. Pullum is a Principal Scientist at the Institute for Scientific Research, Inc., Fairmont, WV. For over 20 years, she has conducted research in software and system dependability, and holds a patent in this area. Dr. Pullum is the author of *Software Fault Tolerance – Techniques and Implementation* (2001) and has written over 350 additional papers and reports. She has served as Principal Investigator on efforts for the National Science Foundation, NASA, the U.S. Air Force, continues on page 14
Navy, and Army, industry and universities. Dr. Pullum holds a Doctorate of Science in Systems Engineering and Operations Research, a M.S in Operations Research, an MBA, and a B.S. in Mathematics.

Marjorie A. Darrah is a Senior Scientist for the Institute for Scientific Research, Inc., Fairmont, WV. Her responsibilities at ISR include research and development in the areas of Neural Networks, Data Mining, and Virtual Reality. Dr. Darrah holds the position of Co-PI on “Development of Methodologies for IV&V of Neural Networks” and PI on “A Formal Method for Verification and Validation of Neural Network High Assurance Systems”, both NASA funded projects. Before joining ISR, she was the chairperson of the Division of Natural Sciences and a mathematics professor at Alderson-Broaddus College, Philippi, WV. Dr. Darrah holds a Doctorate, M.S, and B.S. in Mathematics.

Brian J. Taylor is a Senior Member of Research Staff for the Institute for Scientific Research, Inc. His work includes the development, analysis, and V&V of neural network components for F-15 adaptive flight control systems within the NASA DFRC Intelligent Flight Control project. He is also the Co-PI on a NASA IV&V Facility-funded effort for the “Development of Methodologies for IV&V of Neural Networks.” Taylor was involved in ISR’s first neural network research project where neural networks were used for analytical redundancy of sensors. This project, undertaken with West Virginia University, looked at Sensor Failure Detection, Identification, and Accommodation to improve fault-tolerant flight control and sensor reliability through use of neural networks. Taylor holds a M.S. in Electrical Engineering and B.S. in both Electrical Engineering and Computer Engineering.

**Author Contact Information**

Laura L. Pullum, Marjorie A. Darrah, Brian J. Taylor  
Institute for Scientific Research, Inc.  
320 Adams Street, P.O. Box 2720  
Fairmont, WV 26555-2720  
lpullum@isr.us, mdarrah@isr.us, btaylor@isr.us  
http://www.isr.us

**References**


Institute of Electrical and Electronics Engineering, Inc. (IEEE).  


Software Rejuvenation and Self-healing

Lawrence Bernstein, Industry Research Professor, Computer Science Department
Chandra M. R. Kintala, Distinguished Professor, Electrical and Computer Engineering Department
Stevens Institute of Technology, Hoboken, NJ 07030

Software rejuvenation is a periodic, pre-emptive restart of a running system to prevent future failures. It is one aspect of a self-healing system. It was first introduced, described, implemented, modeled and analyzed in.[1] It is used in systems ranging from a data collector used by most of the US Telephone companies to collect billing information to NASA’s long-duration space mission to Pluto.[2] It is also implemented in IBM’s Netfinity resource manager.[3] Billing system failures and the use of software rejuvenation to prevent those failures, as described in [1], are quite similar to the failures and the fix that Nick van der Zweep described in the Computer World article (QuickLink Ref# 43636) dated January 12, 2004.

Software rejuvenation incurs overhead. Modeling to find optimal times is crucial. A simple and useful model based on Continuous-time Markov chains was introduced in [1] to analyze the reliability improvements due to software rejuvenation; the model is also useful to find optimal trigger rates/frequencies for rejuvenation. This model was then extended using Stochastic Petri Nets to study rejuvenation using the fail-over mechanisms in IBM’s cluster-based systems.[4] X2000 for NASA’s 12-year long Pluto-Kuiper Express mission to do simultaneous on-board preventive maintenance of software and hardware components during cruise and exploration phases used software rejuvenation. Analysis of reliability due to software rejuvenation showed 2 orders of magnitude improvement;[2] optimal interval was found to be 31.2 weeks in the 12-year long cruise phase. A recent paper[5] described software rejuvenation in web servers and how it can be analyzed to determine optimal interval for rejuvenation.

Recent experiments at Stevens Institute of Technology showed that data link protocols suffering memory leak failures could be made reliable using Rejuvenation libraries without having to fix the memory leak bug.[6] In essence Rejuvenation bounds the execution space for the working software so that latent failure modes are not executed. Had this technology been used in the Patriot Missile system during the first Iraq war the counter overflow problem causing the anti-scud system to fail would not have occurred.

The need for this technology was first identified during field tests of the earlier Safeguard anti-missile system. It then was applied to avoid hash table problems in a data switch.

Since the 1960s data communication designers knew to have software modules restart a line when it hung. The rejuvenation technology restarts a line before the hang to avoid potential secondary problems. It is a low cost, easy to implement technology that makes systems more trustworthy.

Software rejuvenation is one aspect of self-healing. Interesting new problems to study rejuvenation of large scale systems are:

- What is a state in a large-scale system for rejuvenation analysis when “state” is across several products and systems?
- Failure symptoms are at a system/network (macro) level but rejuvenation actions are at a component (micro) level; how does one correlate the two?
- What are the models and analytical methods for rejuvenation in large-scale systems?
- How does one do rejuvenation in a large system? Through gradual load shedding?
- What is a safe (clean internal) state to back up to? How does one backup to that state?
- How does the technology become common practice?

Table 7: Typical Software Support Costs

<table>
<thead>
<tr>
<th>Support Cost Category</th>
<th>Effort (% of software development costs)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements synthesis and IPT participation</td>
<td>10 (6 to 18)</td>
<td>Participation in system definition and specification activities</td>
</tr>
<tr>
<td>System integration and test</td>
<td>30 (0 to 100)</td>
<td>Supporting problem resolution activities as the system is integrated and tested</td>
</tr>
<tr>
<td>Repackaging documentation per customer requirements</td>
<td>10 (0 to 20)</td>
<td>Repackaging required to meet some customer preference instead of following normal best practice</td>
</tr>
<tr>
<td>Formal configuration management</td>
<td>5 (4 to 6)</td>
<td>Support for system level CM boards and activities</td>
</tr>
<tr>
<td>Independent software quality assurance</td>
<td>5 (4 to 6)</td>
<td>Quality audits done by an independent organization</td>
</tr>
<tr>
<td>Beta test management support</td>
<td>6 (4 to 10)</td>
<td>Development organization manages bug fixes as they are discovered by outside organizations during beta testing activities</td>
</tr>
<tr>
<td>Independent Verification &amp; Validation (IV&amp;V) or SETD support contractor</td>
<td>6 (4 to 10)</td>
<td>Development organization support to an independent contractor hired to provide technical oversight and direction</td>
</tr>
<tr>
<td>Subcontract management</td>
<td>10 (8 to 16)</td>
<td>Providing technical oversight to those subcontractors tasked with developing all or part of the software deliverables</td>
</tr>
<tr>
<td>Total</td>
<td>82 (30 to 186)</td>
<td>Depending on what tasks are applicable</td>
</tr>
</tbody>
</table>

Notes for Table 7

- Percentage allocations for effort are shown as both an average and a range in parentheses.
- Version control of software deliverables is included in our normal numbers as is normal quality assurance efforts. The additional effort shown in Table 7 refers to formal activities undertaken at the project-level that the software organization must support (e.g., project-level change control boards).
- If large amounts of commercial off-the-shelf software will be used, additional effort must be provided by the software group to support evaluation, glue code development, tailoring and other life-cycle tasks.
- Percentage expressed uses software development costs as their basis.

For example, the chart states that the cost on average quoted for a software job should be increased as much as 186 percent of the base (the base plus 86 percent) to cover the additional costs associated with the activities identified in Table 7 should all of these costs be applicable.

- These support costs can vary greatly based on the manner in which the organization is organized and the work allocated. For example, some firms combine their CM and SQA support in a single Product Assurance organization to take advantage of economies of scale.
- System integration and test support does not include operational test and evaluation for military projects and beta testing for commercial projects. These activities can require even more support than identified depending on the application domain. For example, I have seen aircraft projects that have gone through extensive flight testing burn thousands of hours of software support during these time periods.

- Many firms have adopted defined processes at the organizational level. As such, they are rated as at least a CMM Level 3 organization. When true, these firms generate documentation as part of their normal way of doing business. However, customers can ask for different documentation for their users. When this occurs, repackaging costs for the documentation must be added to the costs because these costs represent divergence to the way they conduct their business.
Industry Software Cost

Continued from page 16.

made. Teams must be coordinated to minimize communications problems. In other words, a lot of extra effort is involved. Many ask me why military projects appear so costly and unproductive. The reason for this is simple. When warfighter lives are at stake, commercial quality is not good enough. As we will show when we look at software quality, the added cost proves worthwhile because the latent defect rates in military systems are much lower than those in their commercial counterparts.

Looking at Software Quality

Achieving high productivity at the expense of quality is the wrong approach. Yet, many organizations have tried to do this in the commercial world at the customer’s expense. To prevent this from occurring, we must view productivity from a quality point-of-view. To accomplish this feat, we must view productivity by normalizing the quality of the products generated upon delivery to the field using metrics like so many software errors per thousand ESLOC.

While there have been many papers published on quality metrics and measurement, few of them have put such error baselines into the public domain. There are many reasons for this omission. First, there is a great deal of confusion over just what an error is. For example, do documentation discrepancies count the same as code anomalies? Second, what is the scope of an error? For instance, do you count defects throughout the life cycle or just start counting once you enter software integration and test? Do you mix development errors and operations errors within the same count? Third and finally, what can you do with the error data once you’ve collected it? Can you feed the data into models and use the predictions about error rates to make decisions on whether or not you have tested enough?

Table 8 portrays error rates by application domain. Errors in this Table are software anomalies discovered and fixed during software integration and testing. The scope of the effort involved starts when integration begins and terminates with delivery of the software. Using this definition, requirements, design, coding and repair problems all constitute an error. However, errors discovered during testing and were not repaired when the product was delivered are not counted. They are fixes pending that will be treated as latent defects in the code that will be repaired at a later time during either system test or maintenance.

The term normative error rate is used to communicate that we are using KESLOC rather then KSLOC as our basis. This is an important consideration because it allows us to take legacy and reused software into account as we compute our averages. Such definitions are consistent with the pioneering work in error analysis that IBM and Motorola has completed in the area of orthogonal defect classification11 (see current updates on-line on the IBM web site located at http://www.research.com/sof/eng/ODC).

Because I do not have comparative data for Asia and Europe, I have not done a comparison. The little data that I do have indicates that the error trends in the United States and abroad are in the same ballpark for similar applications domains.

I felt it was important to include the quality data. The reason for this revolves around the assumption that you don’t want to improve productivity at the cost of quality. To use this data, I suggest that you compare productivity when similar defect rates are present. This will allow you to compare apples with apples because you would be assuming a quality norm for comparison purposes.

Have We Made Progress During the Past Decade?

It is also interesting to look at productivity, cost and quality trends during the past decade. Based on the data I have analyzed, the normative improvements we are realizing across all industries ranges from 8 to 12 percent a year. When compared to hardware gains through new chipsets, these results are dismal. However, when compared against productivity trends in service industries, software shines. When productivity gains of 4 to 6 percent are reported for service industries nationally, stock markets shoot up and the media goes wild. It can be argued that comparing software improvements to hardware gains is like comparing apples with oranges.

It is also interesting to compare my findings to those generated for the DOD by their Cost Analysis Improvement Group (CAIG). My figures for military systems, according to DOD reviewers of this article are about 50 percent better. I really don’t know why. I suspect the reasons deal with scope of the data and definitions. How they collect and validate their data also could be the root cause. It would be interesting to find out why.

Table 9 illustrates the net gain you would realize in a 500 person shop when you accelerated productivity gains from a nominal 4 percent to 12 percent a year. It assumes that you begin with a nominal benchmark productivity of 200 ESLOC/SM as your starting point. The net result is that you would save about $3.8 million if your improvement strategy was capable of achieving this gain. Looking at results of process improvement initiatives continues on page 18
Table 8: Error Rates upon Delivery by Application Domain

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>Number Projects</th>
<th>Error Range (Errors/KESLOC)</th>
<th>Normative Error Rate (Errors/KESLOC)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automation</td>
<td>55</td>
<td>2 to 8</td>
<td>5</td>
<td>Factory automation</td>
</tr>
<tr>
<td>Banking</td>
<td>30</td>
<td>3 to 10</td>
<td>6</td>
<td>Loan processing, ATM</td>
</tr>
<tr>
<td>Command &amp; Control</td>
<td>45</td>
<td>0.5 to 5</td>
<td>1</td>
<td>Command centers</td>
</tr>
<tr>
<td>Data Processing</td>
<td>35</td>
<td>2 to 14</td>
<td>8</td>
<td>DB-intensive systems</td>
</tr>
<tr>
<td>Environment/Tools</td>
<td>75</td>
<td>5 to 12</td>
<td>8</td>
<td>CASE, compilers, etc.</td>
</tr>
<tr>
<td>Military — All</td>
<td>125</td>
<td>0.2 to 3</td>
<td>&lt; 1.0</td>
<td>See subcategories</td>
</tr>
<tr>
<td>— Airborne</td>
<td>40</td>
<td>0.2 to 1.3</td>
<td>0.5</td>
<td>Embedded sensors</td>
</tr>
<tr>
<td>— Ground</td>
<td>52</td>
<td>0.5 to 4</td>
<td>0.8</td>
<td>Combat center</td>
</tr>
<tr>
<td>— Missile</td>
<td>15</td>
<td>0.3 to 1.5</td>
<td>0.5</td>
<td>GNC system</td>
</tr>
<tr>
<td>— Space</td>
<td>18</td>
<td>0.2 to 0.8</td>
<td>0.4</td>
<td>Attitude control system</td>
</tr>
<tr>
<td>Scientific</td>
<td>35</td>
<td>0.9 to 5</td>
<td>2</td>
<td>Seismic processing</td>
</tr>
<tr>
<td>Telecommunications</td>
<td>50</td>
<td>3 to 12</td>
<td>6</td>
<td>Digital switches</td>
</tr>
<tr>
<td>Test</td>
<td>35</td>
<td>3 to 15</td>
<td>7</td>
<td>Test equipment, devices</td>
</tr>
<tr>
<td>Trainers/Simulations</td>
<td>25</td>
<td>2 to 11</td>
<td>6</td>
<td>Virtual reality simulator</td>
</tr>
<tr>
<td>Web Business</td>
<td>65</td>
<td>4 to 18</td>
<td>11</td>
<td>Client/server sites</td>
</tr>
<tr>
<td>Other</td>
<td>25</td>
<td>2 to 15</td>
<td>7</td>
<td>All others</td>
</tr>
</tbody>
</table>

Notes for Table 8

- Errors are anomalies discovered during software integration and testing by comparing actual functionality and behavior to that specified. Requirements, design, code and repair errors are all applicable to the counts in the Table per this definition.

- Error rates in military systems are much smaller than in their commercial counterparts because lives are affected and there can be severe financial consequences when errors occur (a jet crashing into a neighborhood damaging life and property).

- While error rates in commercial products are improving, they are still viewed as too high by consumers. This is especially true when licenses limit the potential legal recourse from damages.

- High error rates typically occur when systems are fielded prematurely. Considerable rework is required to correct anomalies.

- Errors discovered, but not fixed prior to delivery are not included within the counts. They are treated as latent defects that are shipped to the field.

- Errors rates post-delivery tend to be cyclical. As versions are released, error rates soar. As versions mature, error rates stabilize around 1 to 2 errors per KSLOC in systems whose useful life exceeds 5 years. Web business does not fall in this category because such systems have a life which extends at most 2 years before they are replaced.

continues on page 19
Summary and Conclusions

This article was to provide software cost, quality and productivity benchmarks in twelve application-oriented domains that readers can use to determine how well their organizations are doing relative to industry averages. Besides publishing these numbers, one of my goals was to stimulate others to publish their benchmark data as well. I therefore encourage those who disagree with my numbers to publish theirs. But don’t stop there. If you publish, tell the community what your assumptions are, how the numbers were derived, what their source is, and how they were normalized. Just putting numbers out without explaining them would be counterproductive.

I really encourage those of you who do not have numbers to develop them. Throughout my career, I have used numbers to win the battles of the budget, acquire investment dollars, improve my decision-making abilities, and, most importantly, to win management trust. I gained credibility with management at all levels of the enterprise by discussing both the technical and business issues associated with my projects and proposals. I was successful when I emphasized business goals and showed management my ideas made both good business and technical sense. It is not surprising that I rely on the numbers daily. I encourage you to follow suit.

About the Author

Donald J. Reifer is a teacher, change agent, consultant, contributor to the fields of software engineering and management and author of the popular IEEE Computer Society Tutorial on Software Management, 6th Edition. He is President of Reifer Consultants, Inc. and serves as a visiting associate at the Center for Software Engineering at the University of Southern California. Contact him at d.reifer@ieee.org.

Notes for Table 9

- A 4% rate was selected to represent the nominal cost of money borrowed from a bank at the current prime rate.
- The analysis was done using constant year dollars. The returns were not discounted to take the cost of money into account.
- No gains were assumed during the first year of the initiative. This is typical because workforces have to be hired and there are lots of startup costs and issues.
- The compound effect (costs and benefits) of multiple improvement programs being undertaken in parallel was not taken into account. Often, firms embark on initiatives that address many needs in parallel (train the workforce, improve the process, automate the methods, etc.).
- A similar, but more detailed example illustrating many of these points is in [15].
- Another take on benchmarking for ROI for process improvement is in [16].

15 D. J. Reifer, Making the Software Business Case: Improvement by the Numbers, Addison-Wesley, 2002.
The DACS Gold Practice Initiative:

- Promotes effective selection/use of software acquisition & development practices
- Defines essential activities/benefits of each practice
- Considers the environment in which each practice is used
- Addresses the timeliness of practice benefits
- Recognizes interrelationships between practices that influence success or failure
- Contains quantitative and qualitative information
- Includes discussion forums, case studies and survey opportunities
- A continually evolving resource for the DoD, Government, Industry and Academia
- Free to use/free to join

Current Gold Practices:

- Architecture-First Approach
- Binary Quality Gates at the Inch-Pebble Level
- Commercial Specifications and Standards/Open Systems
- Ensure Interoperability
- Formal Inspections
- Integrated Product and Process Development
- Model-Based Testing
- Plan for Technology Insertion
- Statistical Process Control
- Track Earned Value

Learn More About the DACS Gold Practice Initiative:
http://www.goldpractices.com

Become a Registered Member!
http://www.goldpractices.com/registration/register_step1.php

Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400
http://iac.dtic.mil/dacs
The first 50 people to send in a completed survey will receive a FREE DoD/IT Acronym CD from the DACS.

This valuable CD-ROM contains over 9,000 Department of Defense and Information Technology acronyms. There are hundreds of acronym lists available but none are as well done as this CD AND specifically targeted towards DoD and Information Technology. This unique-shaped CD-ROM plays in your computer’s regular, hub-mounted, CD drive. You’ll use this great resource over and over again. It’s FREE, just for filling out our brief survey on the next page!

http://iac.dtic.mil/dacs/
Software Tech News Subscriber Survey

1. Which volume of the Software Tech News did you receive?  
   STN 7:2 Software Cost

2. When did you receive the newsletter? (month/year)

3. How satisfied were you with the CONTENT of the newsletter? (Article Quality)
   ☐ Very Satisfied  ☐ Satisfied  ☐ Neither Satisfied nor Dissatisfied  ☐ Dissatisfied  ☐ Very Dissatisfied

4. How satisfied were you with the APPEARANCE of the newsletter?
   ☐ Very Satisfied  ☐ Satisfied  ☐ Neither Satisfied nor Dissatisfied  ☐ Dissatisfied  ☐ Very Dissatisfied

5. How satisfied were you with the OVERALL QUALITY of the newsletter?
   ☐ Very Satisfied  ☐ Satisfied  ☐ Neither Satisfied nor Dissatisfied  ☐ Dissatisfied  ☐ Very Dissatisfied

6. How satisfied were you with the ACCURACY of the address on the newsletter?
   ☐ Very Satisfied  ☐ Satisfied  ☐ Neither Satisfied nor Dissatisfied  ☐ Dissatisfied  ☐ Very Dissatisfied

7. Approximately how much of the newsletter do you read?
   ☐ The entire issue  ☐ Most of the content  ☐ About half the content  ☐ Briefly Skimmed  ☐ Didn’t Read

8. Would you read this newsletter in an E-mail newsletter format?
   ☐ Definitely  ☐ Probably  ☐ Not Sure  ☐ Probably Not  ☐ Definitely Not

9. How did you request the product or service?
   ☐ Phone Call  ☐ E-mail  ☐ DACS Website  ☐ Subscription Form  ☐ Other: ________________________________

10. Would you recommend the DoD Software Tech News to a colleague?
    ☐ Definitely  ☐ Probably  ☐ Not Sure  ☐ Probably Not  ☐ Definitely Not

11. What topics would you like to see this newsletter devoted to?

    Comments (Optional)

    Contact Information (Optional*)

    Name:  
    Position/Title:  
    Organization:  
    Office Symbol:  
    Address:  
    City:  
    State:  
    Zip Code:  
    Country:  
    E-mail:  
    Telephone:  
    Fax:  

STN 7-2: Software Cost, Quality and Productivity Benchmarks
About the DoD Software Tech News

STN Editorial Board

Philip King  
Editor  
ITT Industries, DACS

Paul Engelhart  
DACS COTR  
Air Force Research Lab (IFEA)

Morton A. Hirschberg  
Editorial Board Chairman  
Army Research Lab (retired)

Ellen Walker  
ITT Industries, DACS

Thomas McGibbon  
DACS Director  
ITT Industries, DACS

David Nicholls  
DACS Deputy Director  
ITT Industries, DACS

Marshall Potter  
Federal Aviation Administration

Richard Turner  
OUSD (AT&L)

Article Reproduction

Images and information presented in these articles may be reproduced as long as the following message is noted:

“This article was originally printed in the DoD Software Tech News, Vol. 7, No. 2. Requests for copies of the referenced newsletter may be submitted to the following address:

Philip King, Editor  
Data & Analysis Center for Software  
P.O. Box 1400  
Rome, NY 13442-1400

Phone: 800-214-7921  
Fax: 315-334-4964  
E-mail: news-editor@dacs.dtic.mil


In addition to this print message, we ask that you send us three copies of any document that references any article appearing in the DoD Software Tech News.

Cover Design by Joe Barbaccia,  
ITT Industries

About This Publication:

The DoD Software Tech News is published quarterly by the Data & Analysis Center for Software (DACS). The DACS is a DoD sponsored Information Analysis Center (IAC), administratively managed by the Defense Technical Information Center (DTIC). The DACS is technically managed by Air Force Research Laboratory, Rome, NY and operated by ITT Industries, Advanced Engineering and Sciences Division.

To Subscribe to this Publication Contact:

Phone: 800-214-7921  
Fax: 315-334-4964  
E-mail: news-editor@dacs.dtic.mil  
Web: www.dacs.dtic.mil/forms/regform.shtml

Distribution Statement:  
Unclassified and Unlimited
In This Issue

Tech View: .................................................... 2

Industry Software Cost, Quality and Productivity Benchmarks ........... 3

Software Testing as an Art, a Craft and a Discipline ....................... 9

Independent Verification and Validation of Neural Networks — Developing Practitioner Assistance ..................... 11

Software Rejuvenation and Self-healing .... 15

Software Tech News Subscriber Survey .... 22

About the DoD Software Tech News......... 23

Advertisement

The DoD Software Tech News is now accepting advertisements for future newsletters. In addition to being seen by the thousands of people who subscribe to the DoD Software Tech News in paper copy, the Tech News will also be placed on the Data & Analysis Center for Software’s website (http://iac.dtic.mil/dacs/), exposing your product, organization, or service to hundreds of thousands of additional eyes.

Interested in learning more? For rates, layout information, and requirements contact:

Philip King, STN Editor
Data & Analysis Center for Software
P.O. Box 1400
Rome, NY 13442-1400

Phone: (800) 214-7921
Fax: (315) 334-4964
E-mail: news-editor@dacs.dtic.mil