SOFTWARE DEVELOPMENT:
STATE OF THE ART VS.
STATE OF THE PRACTICE

Tom DeMarco and Tim Lister
The Atlantic Systems Guild
353 W. 12th St.
New York, NY 10014

ABSTRACT: The state of the art of software development has changed considerably from the folkloric approaches of the 1950s and 60s. But has the state of the practice kept up? A commonly held (rather cynical) view is that the great revolutions associated with the names of Dijkstra, Wirth, Mills, Hoare, Parnas, Myers and others might as well not have happened for all the effect they had on the practice of the average developer.

During the years 1984 through 1987, the authors conducted a series of performance benchmarking exercises to allow individuals and organizations to evaluate their relative productivity. The emphasis of the benchmarks was on speed of program construction and low defect rate. A side-effect of the exercise was that nearly 400 programmers wrote the same program (they all wrote to the same specification) and sent in listings of these programs along with their questionnaires, time logs, and test results. This afforded an opportunity to assess design and coding practice of a wide sample of developers.

KEYWORDS: Empirical study, program design, software development methodology, software measurement.

1. BACKGROUND

Both individuals and the organizations they work for are aware of a distressing ignorance of how their effectiveness compares to that of their peers and competitors. Beginning in 1984, we offered both the opportunity to take part in a kind of "implementation bakeoff," called Coding War Games to help them assess their relative productivity. These exercises were run according to the following rules:

- developers took part in pairs; the two individuals of each pair competed with each other as well as with other pairs in the exercise
- all participants wrote the same program to the same specification
- after design, coding, desk-check and compilation, pair members exchanged programs for testing; development was thus conducted using a clean-room approach, as described in [1]; programs were not repaired at any point—defects reported included all the defects present at the end of desk-check
- after their own tests, developers ran our predefined tests on the product
- submissions sent to us were identified by a code number drawn by the individual at the beginning of the exercise; only the individual knew his/her code; results were conveyed back to participants by code number, thus assuring confidentiality

(A summary of findings from the 1984 and 1985 Coding War Games is presented in [2].)

Over the period of the experiment, we had 392 individuals take part in the Coding Wars. Participants came from 79 different organizations. We divided the participants by language used into seven communities. The relative proportions by language are shown in Figure 1:

![Figure 1: Participants by Language Community](image)

There were 118 COBOL programmers, and 50 in the group called "OTHER." Included under OTHER were Ada, Modula-2, Forth, Cybil, Mapper, MUMPS, Jovial, APL and some languages that we couldn't identify at all.
A distinguishing characteristic of our sample is the fact that participants were curious about how good they were. In addition, they were solicited through announcements in various journals, and through seminars and conferences, thus proving that they could read, or worked for organizations with non-zero training and travel budgets. Aside from these admitted biases, the sample seemed normal to us. Individuals came from all kinds of organizations: financial, service, engineering, university and public sector. There were participants from the U.S., the United Kingdom and Denmark. They were all professional programmers, ranging in experience from one to 16 years.

To determine if the improved state of the art of software development is indeed reflected in the practice, we examined the listings submitted by language. We counted executable statements and declarations, modules, subroutine invocations, pathological connections, module size, variable locality, and coupling. In addition, we attempted to assess module binding strength. Not all analyses were performed on all language groups.

2. HYPOTHESES TO TEST

We selected seven indications of good modularization and programming technique, as reflected in the literature. Each of these was formulated into a positive hypothesis, asserting that the technique was reflected in practice. Then we analyzed participants' code to confirm or refute each hypothesis. The seven hypotheses were:

1. Developers divide programs into modules of manageable size.
2. Developers attempt to maximize locality of data.
3. Developers attempt to maximize module binding strength.
4. Developers attempt to practice data hiding.
5. Developers can use clean room methods to develop zero-defect products.
6. Developers practice structured coding.
7. Organizations with strongly enforced design methodologies achieve meaningful convergence of design from individual to individual.

3. THE RESULTS: AN ASSESSMENT OF THE STATE OF THE PRACTICE

It is possible that the task of wading through 300 plus listings of the same program causes severe damage to the human judgmental capacities. If that is not the explanation, then we must conclude that the state of modularization and programming technique is surprisingly respectable. This is exactly the opposite of what we expected to find. Based on an initial scan of a subset of the data, we were convinced we would be writing a paper demonstrating that the state of the practice was utterly uninfluenced by the state of the art. (Such a paper might have been more amusing.) Five of the seven hypotheses were confirmed. The details are presented below:

- **Hypothesis 1 (Use of Small Modules): Confirmed**

  In all of the following, we use the word module to mean a closed subroutine with a single identifier by which the unit can be activated as a whole. By the middle seventies, the case for small modules was already being made persuasively in the literature (see [3 and 4], for example). But hearsay evidence implied that the state of the practice even ten years later was a module of 500 lines or more. This is complicated by an early "modularization" act performed by the manager at the time of job assignment. These allocations are usually of at least 500 lines, and also called modules. But do developers then implement these as undivided monoliths, or subdivide them further, out of respect for the principle of small modules?

  Our sample indicated a strong preference for subdivision into small modules. The average module was 23 statements. In an analysis of 200 COBOL and Pascal programs, we found only 55 that had even a single module that exceeded a page of listing (the limit proposed by Mills).

  Use of small modules paid off handsomely. When we analyzed the records of those 55 developers who allowed large modules (one or more), we found that they had been substantially outdone by those who exhibited a more serious respect for small modules. The no-large-module subset had 38% fewer defects and was nearly twice as likely to pass the aggregate test of functionality (referred to below as the acid test) included in our test set.

  When we divided the sample into quartiles based on speed and accuracy, we found an increasing number of long module users in the poorer quartiles:

<table>
<thead>
<tr>
<th>Performers in:</th>
<th>1st quartile</th>
<th>2nd quartile</th>
<th>3rd quartile</th>
<th>4th quartile</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st quartile</td>
<td>21% had long module(s)</td>
<td>31% had long module(s)</td>
<td>36% had long module(s)</td>
<td>47% had long module(s)</td>
</tr>
</tbody>
</table>

- **Hypothesis 2 (Maximum Locality): Partially Confirmed**

  Locality analysis was performed only on the Pascal subset.

  As a coarse metric of locality, we looked at the percentage of all variables that were defined so as to be visible in only one module. (The metric is coarse because variables that are truly local to the main-line module are nonetheless visible to subordinate modules under the scoping rules of Pascal.) We defined a locality factor as the count of local variables divided by the count of all variables. Locality factors varied from 0 (all variables visible to all modules) to .96 (only 4% of variables visible
HYPOTHESIS 5 (CLEAN ROOM METHODS): WEAKLY CONFIRMED

The term clean-room development, first coined by Harlan Mills, implies a development process that emphasizes correct initial program construction, rather than cycles of coding and repair. In its most common form developers either code or test, but not both; those who write the code have had the last chance to save their honor (avoid defects) when they announce the product is ready for test. The Coding War Games required a total separation of code and test.

After a controlled experiment on the use of clean-room methods, Selby, Basili and Baker reported in [9] that 86% of participants had complained of missing the satisfaction of testing their own code, but 81% nonetheless said they would use the method again. And resultant product quality was excellent.

Our findings were similar. The most frequent comment from participants was objection to clean-room procedures. However, more than one third of all participants did deliver zero-defect products, even without the opportunity to test and repair. The exercise was small, but not trivial. It involved an average length of 163 Pascal statements or 234 COBOL statements. The fact that clean-room novices could perform so well in this mode is an encouraging sign.

HYPOTHESIS 6 (USE OF STRUCTURED CODING): CONFIRMED

A frequently repeated horror story among consultants is that the 1970s never happened at such-and-such corporation, where spaghetti-bowl coding is the invariant rule. Our sample gave little evidence of this (perhaps apocryphal) effect. Programs were mostly well structured and sensibly indented to call attention to control subordination.

There was more goto use than we might have expected. In the Pascal subset, for example, fully 21% of the programs had goto statements. We looked at each incident of goto use to understand the developers rationale for this possible violation of structured programming technique. Of the 105 goto statements analyzed, the vast majority were emergency exits or loop exits, in keeping with the disciplined use of gotos suggested by Knuth and Zahn [10 and 11]. Fewer than 4% of the programs could be judged “unstructured.”

The COBOL subset was slightly less pure in its use of structured programming. The average program had 11 subroutine invocations and 8 gotos. Again, the gotos were mostly used to implement sensible structured programming constructs not directly supported by the language.

HYPOTHESIS 7 (CONVERGENCE OF DESIGN): REFUTED

An argument for strong, centrally enforced Methodology is that any two developers would be inclined to come up with the same design for a given specification.

One company, known for its almost religiously imposed Methodology, had 16 participants in the 1987 Games. All had been trained in “The Methodology.” There was no sign at all of convergence of design concept among the 16. Their programs had as few as 4 and as many as 23 modules with an even distribution between the extremes. They didn’t even code in a very similar
style (from 0 to 17 gotos, a nearly 3-to-1 variation in total program length).

4. ANOMALIES DETECTED EN ROUTE

In looking over the code, we observed three pronounced patterns that, though they do not pertain precisely to the theme of this paper, seemed worth noting.

The first was a surprising variation in program length. For those who hope to treat length as an indication of function size, the variation of nearly ten-to-one across a sample of programs in the same language and written to the same specification is truly daunting. Figure 2, presented below, shows the variation in program length for the COBOL community.

The observed variation is bound to raise questions about our counting standard. For the record, we performed a syntax-directed count of executable and declaration statements, excluding comments and blank lines. We defined total length as the sum of executable statements and declaration statements. To achieve some uniformity between languages, we did not count braces in C or BEGINs and ENDs in other languages. All counts were conducted to a written standard.

Since we had so many programmers who hit upon the same functionally bound module, we used that module as the basis for a detailed analysis of varying length. All the modules analyzed were functionally identical. The count of statements, however varied for this single tiny module by a factor of eight. When we counted lines instead of statements, the variation was greater. This seemed like an ideal opportunity to experiment in the use of some of the Halstead metrics [13]. We computed Halstead's length, vocabulary and volume for a subset of the modules. Each of the Halstead metrics succeeded to some degree in damping out the variation. Consider the maximum divided by the minimum for different ways of assessing the module's size:

<table>
<thead>
<tr>
<th>SIZE MEASUREMENT</th>
<th>MAX/MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATEMENT COUNT</td>
<td>8.00</td>
</tr>
<tr>
<td>HALSETHAD'S LENGTH</td>
<td>1.73</td>
</tr>
<tr>
<td>VOCABULARY</td>
<td>1.59</td>
</tr>
<tr>
<td>VOLUME</td>
<td>1.94</td>
</tr>
</tbody>
</table>

The second pattern observed was the variation in the number of variables used. In the Pascal community, for example, the average number of variables used was 20. But some programmers used as few as 7 and others as many as 42. Those who used more variables had proportionately longer programs.

The third pattern was that those who wrote more verbose programs took longer to do the work and had a higher defect rate than those who wrote spare programs. This would not have been thought an anomaly at all in the 1970s. But current conventional wisdom is that spare programs might be tighter and so harder to deal with. Our data indicates otherwise.

5. USE OF CWG RAW DATA

Such a large sample of programs written to the same specification appears to be unique. We propose to make copies of the raw data available gratis to researchers who request them. The data is in the form of Macintosh-format Excel diskettes.

Figure 2: Distribution by Program Length (COBOL only)
REFERENCES


