Radovan Andrejčič
Univerza v Mariboru
Bojan Peček
Iskra-Delta, Ljubljana

UDK 681.3:519.683

In a software life cycle the most expensive is the phase of maintenance. A lot of costs are caused by different solutions of similar problems. The programmers source code is an exhibitional example of a variety of algorithms solving the similar tasks. In larger programmer groups there are often arranged some kind of rules for programming the source code. They are nearly always called "programming standards".

135 programs were analyzed from four programmers groups (computer centers) each consisting of 2 - 4 programmers. Statistical methods such as statistical testing, sampling, analysis of variance, etc were used as an unbiased judge. Without knowing intermediate appointments about the programming was compared the source code between programmers, programmers within groups and the code between groups. Differences between programmers within groups were surprisingly small.

Vzdrževanje je najdražja faza življenskega cikla programske opreme. Zelo veliko stroškov povzroča raznolikost reševanja podobnih problemov. V večjih programerskih skupinah si oblikujejo neke vrste pravila programiranja, ki jih skoraj vedno imenujejo "programski standardi".

Delo opisuje analizo 135 programov iz štirih programerskih skupin (računalniških centrov) sestavljenih iz 2 - 4 programerjev, ki temelji na nepristranskih statističnih testih, vzorčenjih, analizi variance itd. Brez poznavanja dogovorov o programiranju je bila primerjana programska koda med programerji, programerji v okviru skupine in skupinami. Presenečajo nepričakovano majhne razlike med programerji v okviru skupin.

1 Introduction

Many software life cycles from different authors have been proposed. They differ in unimportant details. It is common to all of them, that the phase of maintenance is the most expensive. This phase is now the major programming activity, and very soon more programmers will be performing maintenance than development [Jones p. 35].

How to reduce costs of maintenance? Few would disagree that the quality software is not less expensive. But what is the quality software anyway?

2 Software Quality

It is as hard to define as defining a "good car driving". It is differently comprehended from a programmer to another programmer, from one manager to another etc. With the most known facets the software quality can be defined [by Arthur] as:

```
software quality = F(correctness, efficiency,
                     flexibility, integrity,
                     maintainability,
                     portability, reliability,
                     reusability, testability,
                     usability)
```

where each facet can be further reviewed through more criterias. For an example the maintainability can be presented as:

```
maintainability = F(concision, consistency,
                    modularity, simplicity,
                    instrumentation,
                    self-documentation)
```

Some of these criterias are easy to measure, others are not. Everyone can explain modularity, but descriptions vary from one person to another - from equalling modularity with the structured programming, over equalling with a "no GOTO programming", to a philosophy of cohesion and coupling.

And concision and simplicity? Specter of answers is nearly unlimited. Different comprehensions cause different solutions. And this is very often a reason which makes programmers spend more time and money to understand the other programmer than to solve the problem.

Achieving an uniform coding through exact standards is not realistic. "Many rules do have legitimate exceptions" [Grauer p. 92]. But on the other hand - nearly every group of programmers or computer center elaborates its own philosophy of programming. That guidelines are usually called "programming standards". So, a kind of uniformity is possible. But how much?

3 Source Code Analysis

3.1 Technics

It is of course impossible, or at least too expensive to extract data from a sample by hand. A tool or tools are needed.

Our research of the source code has based on two programs. The first one has been oriented on the analysis of the WORKING-STORAGE section. Its input has been the cross reference and the map listing produced by the compiler. Results have given information about distributions of variable descriptions, number of references, number of words in variables, paragraphs, USAGE clauses, etc.

The other program has been oriented on a procedure division. It has produced a table of usages of the COBOL reserved verbs. Occurrences of each verb have also been analyzed in the IF statement. Logical operators have been counted detail in either IF and PERFORM UNTIL statements. This program has also given a number of comments, number of paragraphs, sections, library lines of COPY statements, total number of verbs etc.

Both programs as well as the whole research were done under the DELTA/V V2.0 operating system. Because the majority of the sample programs were written for the PDP-11 computer with the DELTA/M operating system, a little recoding was sometimes needed. What does this mean for the transportability of programs? (This interesting question is not the subject of this paper).

3.2 Sample

3.2.1 Criterion for a Sample

Collecting and analyzing a sample is not only a technical problem, but also an operational one. Very important question is immediately arisen: which programs to include in a sample - every program of an application or just the significant ones? In the first case, the analysis gives the exact answer about the application. But this perfection can hide differences between similar programs. It might show greater similarity than it really exists.

In our research the second method has been used.

3.2.2 Sample Size

Four applications (programming groups) from different computer centers were included into the sample. It was common to all of them that they used the same computer language - COBOL and each group had formed some kind of its own programming rules. It is not worth mentioning that they all sweared on the structured programming (which was prescribed in their "standards").

In this paper applications are marked with letters "A" through "D" and programmers within a group with numbers. Data in table 1 have no significant meaning. They are presented just as an illustration of a sample size.

Table 1 - Illustration about the Sample Size

Prog ramr	No p rgms	No of lines	Average lin/progr	Standar deviat.	Exec. verbs
A1	11	5274	479.45	277.0	2544
A2	10	5517	551.70	331.6	3079
A3	5	1344	268.80	180.0	706
A4	9	4550	505.56	209.4	2908
B1	16	13977	873.56	349.6	5116
B2	17	17706	1041.53	555.2	6984
B3	2	2548	1274	393	1029
C1	15	10599	706.60	195.9	3042
C2	10	6437	643.70	323.1	2187
D1	22	45908	2086.73	617.6	19147
D2	8	15606	1950.75	775.2	6834
D3	10	11149	1114.90	389.3	3518
sA	35	16685	476.71	281.9	9237
sB	35	34231	978.03	475.8	13229
sC	25	17036	681.44	256.4	5229
sD	40	72663	1816.58	731.2	29499
SUM	135	140615	1041.59	731.9	57194

3.3 Analyzing Comment Statements

3.3.1 Importance of the Comment Statements

"Although COBOL is often thought of as a self-documenting language, this is only partially true. With a careful choice of words, each statement can indeed be self-documenting, but it cannot explain its own purpose: it merely states its contribution to a technique or algorithm" [Ledin, Kudlik, Ledin p. 97].

Comments are still needed, they become even more and more important. Specially in the last time, when programs are often not maintained by the original author. As Yurdon says "No programmer, no matter how wise, how experienced, how hard pressed for time, no matter how well intentioned, should be forgiven an uncommented program".

3.3.2 Number of Comments per Source Code

Absolute number of comments in a program does not have any meaning. It needs to be compared with the number of source lines, or the number of executable statements, or with the reserved COBOL verbs. Table 2 presents data about the number of source lines per comment where source lines per comment (SLC) is calculated as

$$SLC = \frac{\text{total number of lines}}{\text{number of comments}}$$

Table 2 - Source Lines per Comment

Prog ramr	No p rgms	No of comment	Average SLC	Standar deviat.	Sum of squares
A1	11	257	20.54	5.76	5003.5
A2	10	318	17.35	3.48	3131.2
A3	5	95	14.14	4.43	1097.7
A4	9	223	20.37	5.90	4046.0
B1	16	1820	7.68	2.52	1044.3
B2	17	2720	6.51	1.54	761.1
B3	2	432	5.9	1.50	74.1
C1	15	2611	4.06	0.49	250.8
C2	10	1387	4.64	1.63	241.7
D1	22	8140	5.64	0.42	702.8
D2	8	2922	5.34	1.22	239.9
D3	10	2418	4.61	0.5	215.1

3.3.3 Differences in Percentage of Comments between Applications

Is there any significant stability in commenting programs? This answer was researched with the analysis of variance. In table 3 there is an analysis of not only differences between programs and programmers, but also of differences between applications. [Andrejcic p. 161].

Table 3 - Analysis of Variance between Applications and Programmers

Source of Variation	Deg. fre.	Sum of square	Mean square	Calculated F	F table
Applications	3	4506.4	1502.1	37.33	23.70
Programmers	8	207.3	25.91	2.92	2.663
Programs	123	1091.2	8.87		

The first null hypothesis - that differences between groups (applications) are not significant can be absolutely rejected ($F_{(3;0.001)} = 23.7$). The second null hypothesis, that differences between programmers do not exist can be rejected too, but the risk is this time for a bit greater - over half a per cent ($F_{(8;0.01)} = 2.663$). This has given a reason for a detailed investigation about differences between programmers within groups. (See table 4).

Table 4 - Analysis of Variance between Programmers within Applications

Source of var.	App.	Deg. fre.	sum square	mean squ.	calculated F	F(0.1) table
between prog ramm ers	A	3	184.3	61.4	2.125	2.28
	B	2	13.7	6.9	1.502	2.49
	C	1	2.0	2.0	1.549	2.88
	D	2	3.62	3.6	7.262	2.44
between prog ramms	A	31	896.2	28.9		
	B	32	146.6	4.6		
	C	23	30.0	1.3		
	D	37	18.5	0.5		

These analyses have shown, that the only application in which significant differences exist was the application "D" (for the risk of 10%, but it is not greater for the risk of 0.1% - $F_{(2;0.001)} = 7.29$). The t-test proved that the programmer "D3" had more comments than the other two. Results have shown greater stability in writing comments than it had been expected.

Comparing the average (9.03 source lines per comment) of this sample with the previous investigation gives also unexpected results. Al-Jarrah-Torsun (page 344) have counted an average of 66.6 source cards per comment card. It is such a great difference, that it needs no special statistical prof. It does not also need the result of Smolej-Korelic - 23.82 lines per comment.

3.3.4 Correlation between Program Length and Number of Comments

Naturally, it is expected that longer programs are more complex and for this reason they need to be more commented. But the previous investigation of dr. Smolej and Korelic(*) discovered unexpected negative correlation between comments and characteristics of complex programs.

Table 5 - Correlation between Program Length and Density of Comments

Prog	Coe.cor.	t	t(table)
A1	0.160	0.486	t(0.50; 9) = 0.703
A2	-0.186	0.525	t(0.50; 8) = 0.706
A3	-0.702	1.707	t(0.10; 3) = 2.353
A4	0.771	3.199	t(0.01; 7) = 3.499
B1	-0.186	0.708	t(0.40;14) = 0.868
B2	0.148	0.580	t(0.50;15) = 0.691
B3	1.000		
C1	0.277	1.038	t(0.20;13) = 1.350
C2	-0.535	1.790	t(0.10; 8) = 1.860
D1	0.304	1.428	t(0.10;20) = 1.725
D2	0.705	2.435	t(0.05; 6) = 2.447
D3	0.409	1.269	t(0.20; 8) = 1.397

According to table 5 programmer "A4" is the only one who can be assumed to have larger programs less commented. The risk of rejecting the null hypothesis, that the correlation coefficient is not significant, is about over 1%. The nearest result of the programmer "D2" increases this risk up to over 5%.

Four programmers ("A2", "A3", "B1" and "C2") had even negative coefficient correlation. This means that the larger programs had relatively more comments. Programmer "A3" had this coefficient even -0.702, but his amount of the sample (5) was too small to reject the null hypothesis. Programmer "C2" with the coefficient correlation -0.535 and greater amount of sample (10) was much closer to the rejection of the null hypothesis.

3.3.5 Sampling Contents Of Comments

"The mere presence of comments, however, does not ensure a well-documented program, and poor comments are sometimes worse than no comments at all" [Grauer p. 103].

There are also known the first rules which suggest how to write comments (to explain reason, not to duplicate code, etc). Their present usage can be compared with the first considerations about structured programming in the early '70.

How to establish the quality of comments? At least two problems occur. The first one is to distinguish good comments from bad ones. It is impossible to do it automatically. A man as an observer and arbiter is needed. And this causes the second problem. The amount of comments is too large to examine every comment line.

---

* Dr. Smolej and Korelic have analyzed 238 programs written by 8 programmers from one computer center with the goal to find representative characteristics of an average program.

Sampling was chosen to give an illustration of the quality of comments. Samples of comments were collected from each programmer. Each line was subjectively estimated as good or bad. Criterias for a good comment line were easy to satisfy. Each line that might be of any help in understanding the code was accepted as good. No additional comparing with the nearest lines of source code was done. Also the AQL was very low - 10% with the risk of 5%. MIL. STD. 105D double sampling plan was used.

Table 6 - Sampling the Quality of Comments

Prog.	Numb.	First sample				Second sample				Acc
ramr	comm.	amo.	AC	RE	res	amo.	AC	RE	res	Rej
A1	257	20	3	7	4	20	8	9	7	AC
A2	318	32	5	9	14					RE
A3	95	13	2	5	4	13	6	7	10	RE
A4	223	20	3	7	12					RE
B1	1921	80	11	16	46					RE
B2	2720	80	11	16	39					RE
B3	432	32	5	9	17					RE
C1	2610	80	11	16	5					AC
C2	1378	80	11	16	7					AC
D1	8140	80	11	16	3					AC
D2	2923	80	11	16	7					AC
D2	2418	80	11	16	6					AC

Results have very clearly rejected programmers with the less commented programs and accepted programmers with better results. What coincidence? Obviously, some groups take a great care about this problem, while the others do not!

3.4 Analyzing User-Defined Words

Beside correct comments a mnemonical significant data names are very important for understanding a data flow. Nearly all authors who deal with programming techniques suggest to use as many of the 30 characters as needed to make names in a program easy to understand. Not only to the original author, but for others as well.

Maybe this is a reason for a surprise when Al-Jarrah-Torsun discovered that the average user defined name had "only" 7.81 characters. In the next table distributions of all user-defined names are shown. Results are grouped into classes of three lengths. The hyphen is counted as the other characters.

Only programmers "B1" and "B2" have user-defined names longer than 18 characters.

In applications "C" and "D" similarities are immediately seen. Not only tests of the mean values and F-tests of the intermediate differences of standard deviation, but also much harder the Kolmogorov-Smirnov test of goodness of fit have proved that there were no differences in distributions between programmers within groups. This was specially surprising in the application "D", where distribution of each programmer was bimodal. (Every programmer had more variables with length of 13, 14 or 15 characters than with 10, 11 or 12). The first explanation was, that this was caused by the influence of the "COPY" statements. But further analyses had contradicted this suspicion.

The Kruskal-Wallis procedure [Andrejcic p. 348]

$$H = \frac{12}{12 * (12 + 1)} * \left(\frac{1369}{4} + \frac{100}{3} + \frac{361}{2} + \frac{144}{3} \right) - 3 * (12 + 1) = 7.47$$

compared with chi-square $X_{(0.05;3)} = 7.81$ gave no reason to reject the null hypothesis, that there were no significant differences between applications. However, the result was very near to the border value for the risk of 5%.

4 Conclusion

4.1 Interpretation

The first discovery was that the relative number of comments is increasing (comparing with the oldest analysis by Al-Jarrah-Torsun and a bit younger by Smolej-Korelic). All applications were produced with the interactive editor, while Al-Jarrah-Torsun wrote about cards. So, maybe also the economical effects can have some influence on the density of comments.

Not only the density, but also the constancy was surprising. It was even not effected by the program length, as it had been measured by the previous analysis. Influence of the group agreements on the programmer were reflected immediately. This brings to a conclusion, that commenting is given more and more care. It has now its place also in "the programming standards".

Table 7 - Distributions of Lengths of the User-Defined Names

Length	A1	A2	A3	A4	B1	B2	B3	C1	C2	D1	D2	D3
1-3	100	66	95	248	96	19	15	384	272	180	59	30
4-6	334	548	105	333	596	995	308	1643	1026	2168	813	731
7-9	419	270	82	323	1070	1400	266	341	221	3966	1269	1072
10-12	72	153	4		315	760	58	76	76	385	238	179
13-15	22	50			255	301	9	30	10	1155	376	296
16-18		11			186	126	2	2		21	16	13
19-21					142	38						
22-24					105	1						
25-27					94							
28-30					44							
average	6.6	7.2	4.9	5.2	10.7	8.7	6.9	5.6	5.5	7.9	8.0	8.2
rank	8	6	12	11	1	2	7	9	10	5	4	3

Sampling of comments gave some disappointing results, or at least unexpected. It was very easy to distinguish between the good and the bad comments. Criterias were easy to achieve, but results rejected programmers with the less commented programs.

After the research was finished, each "programming standard" was studied in detail. Results of the analysis were compared with these prescriptions. In applications"C" and "D" detailed programming guidelines about the form of comment were stated, while in others they were omitted.

Al-Jarrah-Torsun found that the average user-defined name had 7.81 characters and their expectation that it "was expected to find them to be on average much longer" [Al-Jarrah-Torsun p. 343] was not in place. It seems that an average of 8 characters is the most common value. Equality of distributions of the user-defined names was greater than expected. Descriptional estimates about the importance of the long user-names were:

B1 - very important
B2 - very important
B3 - very important
C1 - less important
C2 - less important
D1 - very important
D2 - important
D3 - important

An interview with programmers on the application "A" was not possible. Answers were as expected, except the programmer B3's and D1's. Programmer "B3" was a beginner and the worst typist. "D3" was also very bad, the worst in his group, but they both answered under impression of the group agreements. If the programmer "B3" would be separated, the Kruskal-Wallis procedure

$$H = \frac{12}{11 * (11 + 1)} * \left(\frac{1156}{4} + \frac{9}{2} + \frac{289}{2} + \frac{144}{3}\right) - 3 * (11 + 1) = 8.18$$

would reject the null hypothesis (with the risk of 5%), that there were no differences between applications about lengths of the user names. This would prove, that the statistical significant differences exist. For this reason the correlation between the typing speed and the length of user-names was not analyzed. As there were nearly no differences between programmers within groups, results were obviously more depended on agreements than the dexterity. The suspicion, that the uniformity of distribution was caused by the COPY statements in the WORKING-STORAGE section was comprehended. The amount of user-names from the library files was found to be very low.

With the method of comparing the mean value with the constant it was evidenced that each programmer had different average than the sample of Al-Jarrah-Torsun (7.81) with the greatest risk of 3.67 for the programmer "D2". Coincidently, the whole sample together had an average of 7.79 with standard deviation of 3.62, so critical risk (CR) [Andrejcic p. 100]

$$CR = \frac{7.81 - 7.79}{\frac{3.62}{\sqrt{27484}}} = 0.916$$

gave no reason to contradict the hypothesis that the both samples had statistically equal mean values.

4.2 Comment of the Analysis

It needs to be stated clearly, that the goal of this analysis was not to point to the quality of the software. The goal was to find similarities and differences between applications and programs within an application. And this paper is only to give a short illustration of the analysis, so only a part of the research is shown. There are of course more calculations and comparisons.

This analysis neither measures nor estimates the quality of applications. It is impossible to do it just on some facets about the state of the source code. It is well known, that the quality of the software is designed and determinated in the previous phases of the software life cycle.

The quality of the source code is not the most important component of the software quality. So, it cannot be made equally with the software quality which make part of the linear equation [by ROLAND]

$$X = W_1 * X_1 + W_2 * X_2 + \ldots + W_n * X_n + C$$

where W's are weighting factors and X's are software metrics - each of which may be or may not be given in turn by linear equations of the same form, and C is the constant. One of them is also the maintainability as it had been shown at the beginning. Uniformity of code can be of a great help in eliminating difficulties and frustrations in authorship of the program.

Anyhow, the analysis proved that differences within groups were very small, but "programming standards" were different. Even if they all referred to the same philosophy, they are a great reason for different solutions of similar problems. And our opinion is that this is a subjective argument, which needs to be eliminated. Our analysis examines for the realistic possibilities to achieve it.

References

1. dr. Radovan Andrejcic: "STATISTIKA PRI KADROVANJU IN IZOBRAZEVANJU" - VSOD Kranj 1979
2. Lowell Jay Arthur: "MEASURING PROGRAMMER PRODUCTIVITY AND SOFTWARE QUALITY" - John Willy & sons 1985
3. dr. Robert T. Grauer: "STRUCTURED METHODS THROUGH COBOL" - Prentice Hall
4. Capers Jones: "PROGRAMMING PRODUCTIVITY" - McGraw-Hill 1986
5. M. M. Al-Jarrah and I. S. Torsun: "EMPIRICAL ANALYSIS OF COBOL PROGRAMS" - Software Practice and Experience 9/1979
6. George Ledin Jr., Michael Kudlick, Victor Ledin: "THE COBOL PROGRAMMER'S BOOK OF RULES" - Belmont California
7. Dr. Vitomir Smolej and Igor Korelic: "EMPIRICNA ANALIZA COBOLSKIH PROGRAMOV" - Informatica, Ljubljana oct. 1981
8. John Roland: "SOFTWARE METRICS" - Computer Lang. 6/1986
9. dr. Francis J. Wall: "STATISTICAL DATA ANALYSIS HANDBOOK" McGraw-Hill 1986