C# ENTITIES QUALITY ANALYSIS

Ion IVAN[1]
PhD, University Professor, Department of Economic Informatics
University of Economics, Bucharest, Romania
Author of more than 25 books and over 75 journal articles in the field of software quality management, software metrics and informatics audit. His work focuses on the analysis of quality of software applications.
E-mail: ionivan@ase.ro , **Web page:** http://www.ionivan.ro

Daniel MILODIN[2]
University of Economics, Bucharest, Romania
He graduated "The Informatized Project Management" Master Program

E-mail: daniel.milodin@ase.ro

Sorin-Nicolae DUMITRU[3]
University of Economics, Bucharest, Romania

E-mail: sorin.dumitru@yahoo.com

Dragos PALAGHITA[4]
4th year student, University of Economics, Bucharest, Romania

E-mail: dpalaghita@gmail.com

Abstract: *In this paper we will present the requirements for C# batch program development. Also there is constituted a quality analysis system of indicators for C# programs.*
The software structure used to analyze the C# program batch and computation of regarded indicators is presented. C# program classes are constructed and the belonginess criterion of a C# program to a particular class is established through an aggregation process.

Key words: *C#; quality indicators; C# entities; quality analysis*

1. C# program batch definition

In order to analyze C# programs it is necessary to build a representative program batch.

The size of the batch, expressed as number of programs, is based on probability, which guarantees that the results given by the programs in the batch are correct and represent the type of programs of which they are a part of, based on the result dispersion and on the error margin.

For a 95% probability, a variance of 0.25 and a maximum error of 3% the batch must contain 1067 programs. The result is based on the formula used to determine sample volume, in the case of extraction with return [ISAIC99][5]:

$$n = \frac{z^2 \sigma^2}{\Delta_x^2}$$

where:

z^2 – the Gauss-Laplace repartition quantum for the requested probability level (for P-95%, z=1,96);

σ^2 - the population variance;

Δ_x^2 - the error margin.

The programs must be characterized by:

- homogeneity, the program length must not differ significantly;
- must be developed using C#;
- must be syntactically correct;
- the degree of difficulty must be similar;
- must not contain calls to extern libraries or modules.

To obtain the batch the following procedure was applied:

A collectivity of 50 programmers is considered.

They have significantly similar levels of experience and qualification.

The proposed issues to be solved have restrictions regarding:

- the type of the procedures;
- the nature of obtained results;
- the data structures used;
- the complexity of the software product;
- type of the results returned by the procedures;
- the parameter number for each procedure;
- the types of parameters used by each procedure;
- the complexity of each procedure;
- the number and type of variables that may be used inside the procedures.

The programmers have tested the programs on 100 data sets.

The associated tree like structure was elaborated, and the tests were meant to cover the largest possible scenario number.

The programmers made a series of recordings regarding:

- the time spent coding the program;
- number of errors;
- number of runs;
- running time;
- number of runs with errors;
- number of runs which produced correct results;
- number of runs which produced flawed results,

which they freely declared.

2. Program quality indicator system

There are numerous indicators. The ones for which data is automatically collected and are automatically computed, having input data a number of files which contain in fact programs are called operational, like in figure 1:

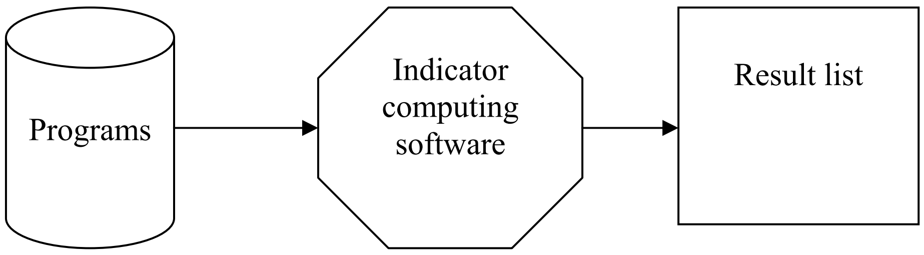

Figure1. Automatic indicator computing.

For program P_1 in which over 80% of the components represent routine activity, having 10000 lines of code, the following activities have been unrolled:

- specification elaboration: 2 days;
- product analysis: 2 days;
- product design: 2 days;
- product implementation: 8 days;
- product testing: 3 days;
- product deployment: 3 days,

resulting in a total of 20 days.

This indicator is influenced by:

- personnel experience, because an experienced personnel will foresee implementation errors, and will decrease the needed time period to implement the software product;
- the complexity, which leads to increasing the time allocated to create the software product; medium complexity programs are preferred, as they do not solicit personnel and computing resources;
- personnel qualification influences the quality and the time needed to complete the development of the software product;
- reused components greatly reduce the time allocated to developing the software product by obtaining them using program libraries;
- the routine of unrolled activities affect personnel attention to details; the reallocation of personnel on different types of programs is preferred such that the state of indifference is avoided.

Source code lines and the length of the file are indicators that apply to programs that have a large number of source code lines.

Complexity is a quality indicator for software products; it establishes the program level of difficulty, expressed as number of source code lines, number of instructions used, and number of repetitive cycles.

The operands n_1 and n_2 are considered. The complexity is determined using the formula:

$$C = n_1 \lg n_1 + n_2 \lg n_2$$

There is software that automatically receives programs sources and identifies the operators, +, -, *, /, %, <, >, !, &&, ||, <<, >>, ==, it counts them and computes their complexity.

Time of completion represents the number of days or hours needed to go through the stages of the software development cycle.

Program length is given as a number of source code lines.

The programmers are instructed to realize the instruction alignment in a convenient way.

The coding of the programs is made depending on the personal way of aligning instructions, specific to each programmer. The C++ language does not enforce any restrictions regarding the arrangement of instructions in the page.

The following instructions are considered:

```
int a,b,c;
a = b + c;
```

There are several ways of writing these instructions:

- all instructions on one line: *int a,b,c; a = b + c;*
- each operand and operator in a new line:

```
int
a,
b,
c;
a
=
b
+
c;
```

- several combinations that imply the methods stated above.

In a structural way, aligning instructions on source code lines must respect several rules:

- a procedure is called with its parameters on one line of code;
- the repetitive cycles are written on several lines of code to clearly identify the running conditions, the operations inside the cycle and the exit conditions from the respective cycle;
- simple instructions are written on one source line;
- the simple conditional instructions are written on several lines to determine the condition to respect, the body that is executed if the condition is true, and the block that is executed if the condition is false;

- the alignment must be done in such a way that the program will be well structured.

To obtain a better view of the program or program block content, each new instruction must be tabbed right compared to the previous instruction if it depends on it.

The procedure to determine the maximum of three integer numbers is considered. It is written in several different readings considering instruction alignment.

Reading R_1

```
procedure maxim(int a, int b, int c)
{
int nr;
if (a>b) nr:= a;
else nr:= b;
if (nr<c) nr:=c;
printf("%d", nr);
}
```

Reading R_2

```
procedure maxim
(int a, int b, int c)
{
int nr;
if (a>b)
 nr:= a;
else
nr:= b;
if (nr<c)
nr:=c;
printf("%d", nr);
}
```

Reading R_3

```
procedure
maxim
(int a, int b, int c)
{
int nr;
if
(a>b)
 nr:= a;
else
 nr:= b;
if
 (nr<c)
nr:=c;
printf("%d", nr);
}
```

In table 1, centralization is made for the number of specific source lines of each reading of the procedure:

Table 1. Procedure sizing

Reading	Size
R_1	8
R_2	12
R_3	15

The designated procedure to compute the maximum has a number of 8 source code lines. The length of the file is influenced by the arrangement of lines in blocks method.

Other traits of the software that is processed to give information regarding the indicators are:

- integrability expresses the ability of the software product to be called and integrated as a component module in a complex software product; Integrability assumes the uniformity of naming used in each module, for naming the variables as well as naming created functions, function parameterization on the basis of new frame formats conveyed upon during design, returning results that can serve as input in another module and processing results obtained from other modules;
- interoperability represents the capacity of the software product to couple with other products; this attribute allows the reuse of some programs to build complex applications; the coupling is assured directly or using interfaces;
- orthogonality is a quality characteristic which establishes the degree of resemblance between two or more software products; orthogonality represents the base of defining the software reuse indicator and determines a program's level of orthogonality.

There are quality characteristics that are highlighted by the behavior of the software product in exploitation.

Viability is perceived as a capacity metric of the software product to function correctly in all given conditions from the beginning. There are quantitative subscriptions of viability, expressed through the probability that a software product to fulfill its functions with certain performances and errors in a time interval and given exploitation conditions, as well as there are qualitative subscriptions which look at viability as a capacity of the software product. The ISO/IEC 9126 standard defines viability as a set of attributes that are based on the capability of the software product to maintain its level of performance in an established time period and conditions. The viability limits of a software product are caused by errors in the requirements definition, design and implementation stages. The problems caused by these errors depend on the conditions in which the software product is used. Corresponding to the software product life cycle, there is a projected or provisioned viability, on experimental viability and an operational viability (at the beneficiary). Viability defines the capacity of a software product to fulfill the functional parameters covering the whole use interval.

Maintenance is a specific process of software products meant to function over a large time interval, meaning longer then three years. In time, because of the technological processes evolution, law changes, structural collectivity modifications, the software products must answer the real requirements of users in order to be chosen.

A software product is maintainable if it allows a quick and easy actualization such that it can be used in the best conditions. A product that respects the maintenance criterions has a sufficiently long use period to amortize the production costs. Exceeding the cost recovery period and ensuring that the software product is viable contributes to the increase of the product's efficiency.

Portability is defined as a set of attributes based on the facility that software products should have, to transfer from one environment to another. The environment is represented by the hardware and/or software context in the organizational framework. Programs are said to be portable, not only if they are implemented on several computers directly, without any modifications, but also if the execution on other types of computer systems needs little modification and a reduced programming effort compared to redeveloping the program. Another aspect of this characteristic is tied to the portability of compilers, to different video and audio facilities offered by the computing systems and used by the application programs. A portable product is used easily in the organization, in organization branches, in departments; this relives the organization of the effort to buy new software programs.

Correctness gives the degree in which a software product satisfies or not the specifications of the problem that needs to be solved. A program is correct if for the input data that satisfies the specification of the problem, the obtained results are correct. The correctness of a software product represents the product's capacity to unroll a group of operations necessary for supplying results used in the analysis and prognosis process through respecting the set of norms at implementation time. The correctness of the program does not refer only to its capacity to respect implemented rules to obtain results, but also to the implementation of correct norms that supply correct results.

There are estimative and effective quality levels. The estimative quality levels are the ones computed and set at application design time. They are based on the formulas that are implemented, repetitive cycles used and the quantity of memory needed to run the software product. The effective levels are the ones obtained after running the software product and making modifications to the source code in order to reach the objectives that the application was developed to fulfill. Between the estimative and effective levels there are some differences caused by the necessity of adapting the software product to the requirements, the improvement of algorithm efficiency, and the appearance of new objectives. The quality indicators that characterize the software product are validated if the differences between the estimated and effective values are acceptable.

3. Software structure

The application developed to analyze the source file quality is available on the internet, being freeware.

The user inputs C# source files, which are tested with the help of the application.

The values returned by the application correspond to applying the quality indicators on the source files inputted by the users.

The software product includes:

- the human – computer interface;
- the computing modules;
- the modules used to validate the data inputted by the user;
- access authentication;
- the problem definition module;
- program storage base for each user.

To ensure user created product testing, a C# program batch is created, which allows the following operations:

- visualization of stored programs
- the selection of a program product from the batch;
- indicator computation for the batch, and individually for each program product ;
- creation of back-up copies and ensures their processing, to avoid the loss of basic information.

The program batch is meant to centralize the existing tendencies in the tested programs, to identify programming and logic errors, and to create code sequences used frequently by programmers.

4. Program classes

The program batch allows the computation of a multitude of indicators.

In [MACES85] the importance coefficients for characteristics are presented. The importance coefficients are aggregated in a characteristic model based on the correspondence between the weight of a characteristic and the value of the indicator of that specific characteristic, as it results from table 2.

Table 2. The correspondence between quality characteristics and their importance

Characteristic / Program	C_1	C_2	…	C_j	…	C_m
P_1						
P_2						
…						
P_i				X_{ij}		
…						
P_n						
Weights	p_1	p_2	…	p_j	…	p_m

Where p_i is the quality characteristic's importance weight C_i and $\sum_{i=1}^{m} p_i = 1$.

The quality characteristic importance coefficients are used to determine the program quality aggregated indicator using the formula:

$$IA(P_i) = \sum_{i=1}^{m} p_j * x_{ij} = IA_i$$

The determination of the aggregated program quality indicator underlines the basis of grouping the programs into quality classes.

The grouping into classes is done by establishing the number of classes and their size. Choosing the number of classes requires the knowledge of the program quality values variation, the elaboration of several class patterns until the optimum solution of the phenomena is reached.

To determine the quality class existence intervals the minim and maximum value of the quality values needs to be established. The difference between the two values is the amplitude, A.

To establish the quality classes, their number must be determined. It is determined using the Sturges rule:

$$N = 1+3,322 * \log_{10} n$$

where:
n – the number of programs for which the quality classes are determined;
N – the number of quality classes.

Based on the number of classes the size of each class, *d*, is determined according to the formula:

$$d = A / N$$

After the number of quality classes is computed, it is delimited by lower and upper bounds.

The bounds are determined in the following manner:

- the upper bound of each interval, will take the value of the lower bound of the following interval, this way repetitive bound intervals are obtained;
- the upper and lower bounds of the intervals are differenced by one unit;

The stages that must be covered to establish the program quality classes are the following:
S1: individual quality characteristic importance weight allocation
S2: determination of the program quality aggregated indicator;
S3: building the program's quality classes;

The covering of the above stages ensures the delimitation of program quality on value intervals, quality classes contributing to the grouping of programs considering the aggregated quality level.

5. Establishing the belonging of a program to a quality class

Program quality estimation imposes the creation of value delimited quality classes of quality indicators. The belonging of a program to a certain quality class supposes the computation of quality indicators applied to the respective program and the identification of the bounds between which the program must fit.

The analyzed programs must respect a series of frame conditions which certify their belonging to one of the quality classes:

- to be written in C#;
- to allow data input;
- to allow data visualization;
- not to contain syntax errors, or interpretation errors;
- to validate processed data.

The quality level for the *m* characteristics must not differ significantly compared to the average level.

The frame conditions have the role of limiting programs that are the subject of analysis. Program analysis is realized based in quality characteristics computed for every program.

Program quality is composed of a set of quality characteristics which are divided in:

- technical and usage characteristics;
- economical characteristics;
- social characteristics.

Quality characteristics are imposed not only by the client, but by the need that the software product runs efficiently, as well. Based on the quality characteristics the quality cost in determined, respectively the implied quality characteristics implementation cost and the cost of the quality characteristics requested by the client.

The quality cost is determined based on the formula:

$$CC(P_i)=\sum_{i=1}^{m} p_j * x_{ij} * c_j = CC_i$$

where:

m – number of the quality characteristics;

c_j – the quality cost of characteristic *j*;

x_{ij} – quality value of characteristic C_j for program $P_{i.}$

The belonging of a program to a quality class is established after the building the quality classes, considering the quality aggregated value of each program.

After the quality classes and their bounds are agreed upon, the programs and their belonging to quality classes are identified.

The quality characteristics dependency structure is presented in [IEEE94]:

- maintainability defines the degree of effort to repair, maintain or improve a product – C_1;
- correctness, represents the degree of effort necessary to correct errors and meet the user formulated requirements – C_2;
- modifiability, describes the effort to improve or modify the functions of the software product – C_3;
- testability, measures the effort needed to test the software product – C_4.

Table 3 presents the results of applying the above quality characteristics to a sample of 40 programs and the weight of the quality characteristics in the quality system aggregation:

Table 3. The level of quality characteristics

Characteristic / Program	C_1	C_2	C_3	C_4	IA	Characteristic / Program	C_1	C_2	C_3	C_4	IA
P_1	0.8	0.5	0.25	0.55	0.5675	P_{21}	0.6	0.68	0.55	0.84	0.6845
P_2	0.5	0.6	0.69	0.78	0.6375	P_{22}	0.7	0.77	0.77	0.82	0.764
P_3	0.6	0.1	0.65	0.76	0.5305	P_{23}	0.9	0.87	0.78	0.87	0.8655
P_4	0.3	0.98	0.89	0.92	0.7445	P_{24}	0.7	0.85	0.91	0.92	0.835
P_5	0.8	0.32	0.82	0.9	0.713	P_{25}	0.97	0.81	0.82	0.9	0.8865
P_6	0.9	0.65	0.34	0.74	0.7055	P_{26}	0.69	0.82	0.86	0.74	0.763
P_7	0.12	0.45	0.78	0.58	0.4395	P_{27}	0.18	0.88	0.8	0.84	0.646
P_8	0.54	0.89	0.79	0.64	0.695	P_{28}	0.76	0.89	0.79	0.82	0.815
P_9	0.69	0.32	0.75	0.15	0.4445	P_{29}	0.39	0.81	0.75	0.88	0.696
P_{10}	0.57	0.91	0.76	0.77	0.7435	P_{30}	0.57	0.91	0.76	0.77	0.7435
P_{11}	0.22	0.52	0.58	0.88	0.547	P_{31}	0.77	0.75	0.82	0.88	0.8055
P_{12}	0.68	0.23	0.32	0.89	0.5765	P_{32}	0.65	0.76	0.88	0.89	0.784
P_{13}	0.21	0.17	0.89	0.77	0.47	P_{33}	0.74	0.73	0.89	0.77	0.769
P_{14}	0.88	0.75	0.77	0.61	0.75	P_{34}	0.88	0.75	0.77	0.91	0.84
P_{15}	0.37	0.85	0.68	0.58	0.5995	P_{35}	0.85	0.85	0.83	0.92	0.868
P_{16}	0.89	0.83	0.69	0.46	0.716	P_{36}	0.89	0.83	0.84	0.93	0.8795
P_{17}	0.49	0.27	0.54	0.91	0.5685	P_{37}	0.85	0.77	0.87	0.91	0.851
P_{18}	0.61	0.84	0.23	0.29	0.5145	P_{38}	0.81	0.84	0.85	0.9	0.8505
P_{19}	0.55	0.55	0.61	0.39	0.511	P_{39}	0.82	0.83	0.77	0.83	0.818
P_{20}	0.92	0.64	0.88	0.94	0.85	P_{40}	0.92	0.88	0.88	0.94	0.91
Characteristic weight	**0.3**	**0.25**	**0.15**	**0.3**	**1**	**Characteristic weight**	**0.3**	**0.25**	**0.15**	**0.3**	**1**

The next stage constitutes the determination of quality classes.

The minimum value is 0.4395, and the maximum value is 0.91, the amplitude is 0.4705. The number of quality classes is 6, and the class size is 0.0784.

The quality classes are:

Class 1 $\in$ [0.4395;0.5179);
Class 2 $\in$ [0.5179;0.596);
Class 3 $\in$ [0.596;0.67);
Class 4 $\in$ [0.67;0.75);
Class 5 $\in$ [0.75;0.83);
Class 6 $\in$ [0.83;0.91].

Determining the quality classes and their lower and upper bounds makes up the starting point for the next stage, software product quality centralization.

After the centralization of data regarding the quality level of each program, the distribution of program in quality classes is obtained, given in table 4:

Table 4. Program distribution in quality classes

Class	Program number
1	5
2	5
3	3
4	9
5	8
6	10
Total	40

Corresponding to this algorithm of computing the aggregated level of software product quality, the quality characteristics costs specific to every program are added to the algorithm. Based on these the software product quality cost is determined.

By building dedicated software the processing of the 1067 programs of the sample is automated, the extension of the number of processed programs is a quantitative adaptation.

6. Conclusions

Program classes are related to costs. Thus, the level of expenses that has to be supported by customers in order to benefit from superior quality software products is identified.

The method of developing classes mixes the quality level of software products with the importance given by clients to each software quality characteristic.

For each client, the characteristics bare a level of importance dependent on the utility that they bring to the bought software product. The quality cost calculation includes the weight given to each quality characteristic, considering the level of homogeneity desired for the software product. The customers that buy the software product strictly for one of its quality characteristics will not pay extra for improvements as long as the clients who buy products with a large range of applicability need a high level of quality and homogeneity.

By implementing quality classes the software products are differentiated and catalogued, thus offering to clients to clients to find the needed software depending on its cost and desired quality.

Bibliography

1. Anghelache, C. and Niculescu, E. **Statistica: Indicatori, formule de calcul si sinteze,** Editura Economica, Bucharest, 2001
2. IEEE Standards Collection **Software Engineering, Std. 1045-1992 IEEE standard for software productivity metrics,** Published by The Institute of Electrical and Electronics Engineers, New York, 1994
3. Isaic–Maniu, Al., Mitrut, C. and Voineagu, V. **Statistica pentru managementul afacerilor,** Editura Economica, Bucharest, 1999
4. Ivan, I. and Boja, C. **Metode statistice in analiza software,** Editura ASE, Bucharest, 2004
5. Ivan, I. and Popa, M. **Entitati text – dezvoltare, evaluare, analiza,** Editura ASE, Bucharest, 2005
6. Ivan, I. and Popescu, M. **Metrici software,** BYTE România, vol.2, no.5, May 1996, pp.73-82
7. Ivan, I. and Teodorescu, L. **Managementul calitatii software,** Editura INFOREC, Bucharest, 2001
8. Ivan, I., Milodin, D. and Dumitru, S. N. **Ortogonalitatea programelor C++ software mining,** Revista Romana de Informatica si Automatica, vol. 17, no. 2, 2007, pp. 39 – 54
9. Macesatu, M., Arhire, R. and Ivan, I. **Clase de complexitate pentru produse program,** Buletinul Roman de Informatica, nr. 1, Bucharest, 1985, pp. 63 – 68

[1] Ion IVAN has graduated the Faculty of Economic Computation and Economic Cybernetics in 1970, he holds a PhD diploma in Economics from 1978 and he had gone through all didactic positions since 1970 when he joined the staff of the Bucharest University of Economics, teaching assistant in 1970, senior lecturer in 1978, assistant professor in 1991 and full professor in 1993. Currently he is full Professor of Economic Informatics within the Department of Economic Informatics at Faculty of Cybernetics, Statistics and Economic Informatics from the University of Economics. He is the author of more than 25 books and over 75 journal articles in the field of software quality management, software metrics and informatics audit. His work focuses on the analysis of quality of software

applications. He is currently studying software quality management and audit, project management of IT&C projects. He received numerous diplomas for his research activity achievements. For his entire activity, the National University Research Council granted him in 2005 with the national diploma, Opera Omnia.
He has received multiple grants for research, documentation and exchange of experience at numerous universities from Greece, Ireland, Germany, France, Italy, Sweden, Norway, United States, Holland and Japan.
He is distinguished member of the scientific board for the magazines and journals like:
- Economic Informatics; - Economic Computation and Economic Cybernetics Studies and Research; - Romanian Journal of Statistics
He has participated in the scientific committee of more than 20 Conferences on Informatics and he has coordinated the appearance of 3 proceedings volumes for International Conferences.
From 1994 he is PhD coordinator in the field of Economic Informatics.
He has coordinated as a director more than 15 research projects that have been financed from national and international research programs. He was member in a TEMPUS project as local coordinator and also as contractor in an EPROM project.

[2] He graduated "The Informatized Project Management" Master Program. He also graduated the Economy Informatics section from Cybernetics, Statistics and Economic Faculty, in 2005.
He published a series of articles in specialized magazines, articles regarding the study of the orthogonality for the alphabets, Arabian digits, Latin alphabet, using experimental methods and techniques, and articles regarding the collectivity components classification, using different criteria.
The disquisition paper in the master program follows the same research line, "The projects orthogonality, conditions for entering in the evaluation program", proposing to define the concept of project's orthogonality, methods for determine the degree of similarity between two projects, and to develop software products used to identify the project's similarity.

[3] Graduated the Economy Informatics section from Cybernetics, Statistics and Economic Faculty, in 2006. In the same year, he follows the "Informatics Security Master" program.
He published a series of articles in specialized magazines, regarding the study of software's source codes, based on text entities theory and software orthogonality using experimental methods and techniques.
The disquisition paper in the master program follows the "Informatics Security" line, "The security of online quality analizing applications of C++ programs" , proposing to define the concept of software quality, methods for determine the degree of similarity between two source codes, and to develop web-based software products used to identify the software's similarity and determine the quality indicators.

[4] Dragos Palaghita is a 4th year student in the University of Economics, Bucharest, Cybernetics Statistics and Economic Informatics faculty, Economic Informatics section. He is programming in C++ and C# and his main areas of interest are Informatics Security and Software Quality Management.

[5] Codification of references:

[ANGH01]	Anghelache, C. and Niculescu, E. **Statistica: Indicatori, formule de calcul si sinteze,** Editura Economica, Bucharest, 2001
[IEEE94]	IEEE Standards Collection **Software Engineering, Std. 1045-1992 IEEE standard for software productivity metrics,** Published by The Institute of Electrical and Electronics Engineers, New York, 1994
[VOIN99]	Isaic–Maniu, Al., Mitrut, C. and Voineagu, V. **Statistica pentru managementul afacerilor,** Editura Economica, Bucharest, 1999
[IVAN04]	Ivan, I. and Boja, C. **Metode statistice in analiza software,** Editura ASE, Bucharest, 2004
[IVPMR05]	Ivan, I. and Popa, M. **Entitati text – dezvoltare, evaluare, analiza,** Editura ASE, Bucharest, 2005
[IVAN96]	Ivan, I. and Popescu, M. **Metrici software,** BYTE România, vol.2, no.5, May 1996, pp.73-82
[IVAN01]	Ivan, I. and Teodorescu, L. **Managementul calitatii software,** Editura INFOREC, Bucharest, 2001
[MILO07]	Ivan, I., Milodin, D. and Dumitru, S. N. **Ortogonalitatea programelor C++ software mining,** Revista Romana de Informatica si Automatica, vol. 17, no. 2, 2007, pp. 39 – 54
[MACES85]	Macesatu, M., Arhire, R. and Ivan, I. **Clase de complexitate pentru produse program,** Buletinul Roman de Informatica, nr. 1, Bucharest, 1985, pp. 63 – 68