THE INFLUENCE OF SOFTWARE COMPLEXITY ON THE MAINTENANCE EFFORT
- CASE STUDY ON SOFTWARE DEVELOPED WITHIN EDUCATIONAL PROCESS –

Iulian Ionut RADULESCU[10]
PhD candidate, Economic Informatics Department
Academy of Economic Studies, Bucharest, Romania
Published articles:
Comparative analysis of the complexity of text entities generated using programming technique (co-author), ASE, Bucharest, 2006
Variations of software complexity for products developed using different programming techniques, Seventh International Conference on Informatics in Economy, Department of Informatics in Economy, ASE, May 2005.
E-mail: iulian.radulescu@gmail.com

Abstract: *Software complexity is the most important software quality attribute and a very useful instrument in the study of software quality. Is one of the factors that affect most of the software quality characteristics, including maintainability. It is very important to quantity this influence and identify the means to keep it under control; by using quantitative methods for evaluating and analyzing it, this is possible. On the other hand, it can help in evaluating the students during education process. The complexity of the projects developed during the specialized courses, which have similar requirements, or even the same requirements, reveals students programming abilities, his know ledges about programming technique and help identifies the ones that try to cheat, by copying.*

Key words: software quality, software characteristics, software complexity, software maintainability, software measurement

The Complexity of Software Products

The major problem of software industry today is represented by the consequences of the extraordinary expansion of information technology, in all society areas, which now has become an information society. The attempt to model new domains of human activity has generated very complex software systems. Business domain complexity has generated complexity within the software product. New technologies have been developed to answer the new business requirements.

Software complexity is an extremely important element in software quality analysis. It influences the majority of software quality characteristics and, on the way it is controlled and monitored, depends the success of a software project.

Software complexity has many aspects. Most of the times are present in the same time inside a software project, which makes it more difficult to have a pertinent analysis of the phenomenon.

The complexity related to the modelled business domain is called *functional complexity* or *problem complexity*. It is an inherited complexity from the business domain which cannot be decreased, but only controlled, in the sense of including or excluding complex functionalities from the final product. The problem complexity cannot be measured using quantitative measures.

Another type of complexity is the *structural* one. It is the easiest to understand and analyze, because it refers to the structure of the software product, to technical elements which makes it: modules, libraries, classes, functions. Structural complexity has the advantage to be measurable. There are numerous sets of metrics which analyze the design and the source code of a software product and offer useful information regarding their complexity. The disadvantage consists in the fact that this type of complexity is evaluated relatively later in the development cycle of a software product, within design and implementation phases.

The most difficult type to assess is *cognitive complexity*. It refers to the effort necessary for a programmer to understand the software product. It is highly related to the technical know ledges of the developer, to its personal abilities like wit, analytical thinking, and of course, to the structural complexity of the analyzed component. A quantitative analysis of this type of complexity is impossible to make, its nature is more psychological than technical.

Software Maintainability

A software product is not completed when all the requirements are implemented. After it is installed in real, production environments, and is used by the final users, the following situations appear:

- defects are discovered during execution in the production environments, more complex than the development and testing environments;
- the customer discovers, once is using the product that he also needs other functionalities to be implemented which become implementation requirements.

These two major categories of possible situations appear during the *maintenance* phase of the product. The costs associated with fixing these problems are distributed as follows:

- any defect is attributable to the software manufacturer so the costs are covered by him;
- any cost related to extensions of the functionality are covered by the customer;

To minimize the costs, especially those related to defect correction, the developed product should be easy to update, meaning:

- should allow the isolation and easy correction of the defects, without major risks of introducing new defects in the code;
- should allow the addition of new functionalities, without affecting the existing ones.

The analysis of software maintainability should be done starting from the development phase, in order to minimize the future costs. Using specific metrics, it could be easily identified the components – classes, functions, modules – which can be, theoretically,

hard to maintain and corrective actions can be taken in order to improve this. Is logic that a class, function or module, which is more complex, is also harder to maintain, so focus will be put on the components with high complexity. Although the relation between maintainability and complexity is obvious, is necessary to demonstrate these using quantitative methods and also to identify the type of correlation between the two.

Applied Software Metrics

In order to study software complexity, McCabe metric[11], which describes the *cyclomatic complexity*, was chosen, for the following reasons:

- is independent from the programming language and is equally applicable, using different variations and extensions of it, to all important programming techniques: structured programming, modular programming, object-oriented programming or component-based programming;
- is offering an image of the structural complexity, of the source code, but also an image of the complexity of implemented algorithms, algorithms which are strongly connected to the functional complexity; in a way, this metric can also be used to describe, within some limits, the functional/algorithmic complexity of a software component.

The indicator is based on the existence of a graph associated to any software program, which is also called *control flow graph*. In such a graph, every node corresponds to a block of code from the source where the execution is sequential, and the arcs correspond to branches created because of the *decision points* or *decision blocks*. The graph has only one entry node and one exit node, and the exit node is accessible from any other node within the graph. In these conditions, the *cyclomatic complexity* or the *cyclomatic number v(G)* is calculated using the following formula:

$$V(G) = e - n + 2p$$

where *e* is the number of arcs, *n* is the number of nodes, and *p* is the number of connected components. For a monolithic program or for a single function, the value of *p* is always 1, because there is only one component involved. When a function or a module contains calls to other functions, then all involved functions are considered connected components and the complexity of such a module is calculated using the relation:

$$v(G) = \sum_{i=1}^{k} v(C_i)$$

where C_i represents the connected component identified inside the module, including the module itself. So, if we have a module M which calls two functions A and B, then the cyclomatic complexity is given by the relation: v(M) + v(A) + v(B). The formula is applicable recursively, in case there is more than one level in function calling stack.

To simplify the things, in case we are dealing with monolithic programs or functions that do not call other functions, the cyclomatic complexity is calculates as follows:

V(G) = number of decisions inside the function or module or program + 1

The number of decisions inside a function/program includes both the conditional constructions, like **if...else**.., **switch**..., and the repetitive ones: **while**, **for...**. It is also important to mention the fact that, in case the decision is compound (for example *A AND B*), it is actually counted as two decisions because, if the operator AND was missing, the sequence would transform in two decision blocks, respectively:

```
If ( A )
    If ( B )
```

Although is relatively easy to determine the complexity, especially applying the last formula which does not require the actual construction of the graph, the results are still obtained in the development phase. To minimize the risks in development process and to identify earlier the possible problems, is useful to obtain information about complexity as earlier as possible within the software development cycle, which means even starting with the analysis and design phases. For these, other metrics should be used, which are not in the scope of this article

Experimental Results

In order to apply the metrics and to analyze the results, a set of C programs were selected, with variable sizes, either monolithic or based on libraries of functions. The programs are developed by students of **Faculty of Cybernetics, Statistics and Informatics Economics** from **Academy of Economic Studies** Bucharest, for the **Data Structures** course.

The following elements were considered, during data collection process, which define the rules of selecting and recording the information:

- **break** statements were counted as executable statements, so they are part of *NLOC* indicator, measured at function/program level;
- if conditional or loop statements contains also assignment statements, like below:

```
if  ( ( f = fopen(„fisier.txt”,”r”) ) != null ) ...
```

then both the conditional statement (**if** in this case) and the assignment statement are counted as executable statements;

- all declaration statements grouped on a single line, like in the example: *int a,b,c* were counted together, as a single line of code;
- if conditions are multiple, and contains the logical operator AND, then every condition is counted separately as decision point, when the cyclomatic complexity is measured. For example:

```
if (a && b )
```

is equivalent with

```
if ( a )
    if ( b )
```

which means two decision points;

- **switch** statement was counted only one time as executable statement, no matter how many **case** statements includes, but, every **case** was considered as separate decision point, and respectively counted for cyclomatic complexity.

In the first phase, the relation between the number of executable statements and cyclomatic complexity will be studied. Although the number of lines of code, as metric, is among the most controversial ones because it is strongly linked to the programming language, it still offers an indication on the level of maintainability for a software program. It is important for the following situations to be studied:

- when the number of lines of code, is small, for a module or program, but the complexity is high; this might be an indication of a very poor design of the module/program which influence in a negative way the maintainability of it;
- when the number of lines of code is big, and also the complexity is high; in this case, if the results are at function level, it indicates that actions like re-factoring are necessary, in order to avoid huge, very complex functions in the source code;
- when the number of lines of code is big, but the complexity is low, which indicates a more normal situation.

Following data collection and based on the evaluation of the indicators *NLOC* (number of executable lines of code) and *V(G)* (cyclomatic complexity) at program level, the following values are obtained for the 20 analyzed programs:

Table 1. The values for NLOC and V(G) based on collected data

Program	Number of functions	NLOC	Cyclomatic complexity	NLOC/(V(G) – 1)
P1	11	354	85	4.21
P2	9	179	75	2.42
P3	8	89	34	2.70
P4	7	92	49	1.92
P5	1	26	5	6.50
P6	8	162	40	4.15
P7	10	284	160	1.79
P8	1	76	14	5.85
P9	1	123	20	6.47
P10	1	37	5	9.25
P11	4	68	24	2.96
P12	4	55	17	3.44
P13	15	262	90	2.94
P14	13	304	75	4.11
P15	5	96	42	2.34
P16	6	112	23	5.09
P17	1	39	5	9.75
P18	17	271	62	4.44
P19	14	379	97	3.95
P20	32	629	426	1.48

During the analysis, we will consider *NLOC* as an independent variable and *V(G)* as dependent variable. The distribution graphic for the values in the above table is the following:

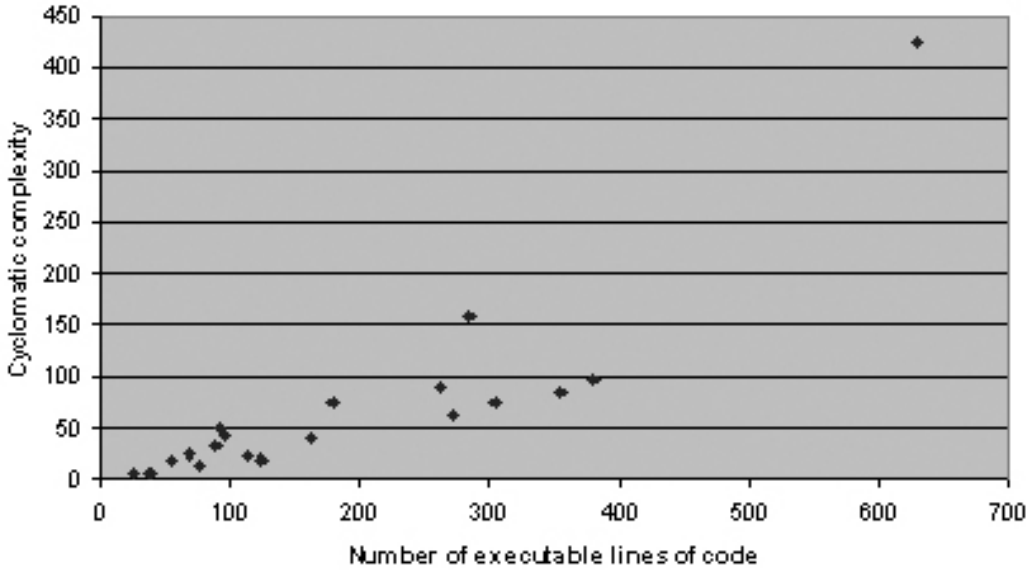

Figure 1. Distribution of cyclomatic complexity in relation with the number of lines of code

It can be noticed that, except for program P20, for which values very different from the others were recorded, the distribution indicates a linear relation between the two variables. Obviously, for high values of LOC indicator, we get high values for cyclomatic complexity, which shows a direct relation between the two. The value for the linear correlation coefficient is:

$$r = 0.8798$$

This indicates a very strong relation, the value being very close to 1.

The conclusion of this experiment is that, although the cyclomatic complexity is independent on the programming language, and implicitly, on the language constructions, there is still a connection between the size of the program, measured by the number of lines of code and its cyclomatic complexity. Still, because of the criteria previously defined for data collection process, the number of executable statements has provided not a result strongly dependent on the language but on the algorithm implementation and developer's skills in writing the code.

As the size of the program is an indication on its maintainability level, we can conclude that the maintainability is a function of complexity, and the relation between them is an inverse one: more the size of the program is bigger, more is hard to maintain, and the complexity is higher.

On the other hand, if the number of lines of code is divided to *V(G) – 1*, which is the number of decisions, it will give an indication to how many lines on code are between two decision points in code.

It can be noticed that, in average, at every **3.5** lines of executable code there is a decision block. For a function with **30** to **35** lines of code, we identify between **8.5** and **10**

decision blocks. This makes the number of possible execution paths significantly bigger and shows that the testing coverage for such a code cannot be 100%.

Student's Evaluation based on the Quality Analysis of the Source Code

The analysis of the programs built by the students during the faculty courses, besides the fact that it serves to a better understanding of software quality and how software characteristics influence each other, it also serves in the actual evaluation of the students. So, having in mind that the project requirements have a similar functional complexity, the following situations should be tracked:

- significant variation of cyclomatic complexity between various projects;
- significant variation of the size of source code between various projects;
- modular design of the project: some projects are monolithic, others are based on libraries of functions.

In the program set chosen above, the followings have functional requirements with close complexity: P1, P2, P6, P7, P9, P13, P14, P18, P19, P20. The others were chosen to be able to show how the relation between size of the sources and cyclomatic complexity evolves on a larger scale of values.

Analyzing the program subset mentioned above, it can be noticed that the values of the two metrics are quite different between projects because of:

- Attention paid to the graphical user interface; some students have preferred to go with a minimum interface for reading values and printing results, for the user. Others they went in more details, creating more professional user interfaces, though this was not a requirement in the project;
- The students ability to work more structured, creating reusable modules;
- There is a certain homogeneity, regarding the indicator *NLOC/(V(G) – 1)*, which shows that, from the source code point of view, the average complexity is pretty much the same.

On the other hand, if the requirements would have been the same for everybody, the analysis above is useful to reveal the following aspects:

- the uniqueness of the chosen solution; if the complexities are equal, the solutions might be the identical or at least, there is a theoretical chance to be like that;
- in case the solutions are completely different, from the complexity point of view, then either some of the projects contains more elements than required, or the best solution was identified, or some of the requirements were not implemented.

Conclusions

Software complexity analysis and the way it influences the rest of software quality characteristics is very important to have a better control of the development process. Although it has the disadvantage that it cannot be used until later in the development cycle, when the code is written, the proposed metrics have still the advantage to capture several aspects and risks elements, which might affect product quality and can generate future supplementary costs. A product which is built to be easily maintained it produces minimum future costs.

Also, the analysis of the complexity can serve as evaluation procedures for students or any participant to specialization courses. It can reveal information about student's technical abilities, design and programming skills, even about personal characteristics.

Bibliography

1. Thomas J. McCabe A**. Complexity Measure**, IEEE Transactions on Software Engineering, VOL. SE-2, No. 4, December 1976
2. Mary Shaw, Paul Luo Li, Jim Herbsleb **Finding Predictors for Open Source Software Systems in Commonly Available Data Sources: a Case Study for OpenBSD**, Institute for Software Research International, School of Computer Science, Carnegie Mellon University, June 2005
3. V. Basili, D. Hutchens **A Study of A Family of Structural Complexity Metrics**, Proceedings of the ACM/NBS Nineteenth Annual Technical Symposium; **"Pathways to System Integrity"**, PP 13-16, June 1980
4. V. Basili **Quantitative Software Complexity Models: A Panel Summary**, Proceedings of the Workshop on **"Quantitative Software Models for Reliability, Complexity, and Cost"**, IEEE publication October 1979
5. Ion Ivan, Mihai Popescu, Panagiotis Sinioros, Felix Simion **Metrici Software**, Editura INFOREC, Bucureşti, 1997
6. Goutam Kuma Saha **Beyond the Conventional Techniques of Software Fault Tolerance Ubiquity**, Volume 4, Issue 47, Jan 28 – Feb 3 2004

---

[10] Iulian Ionut Radulescu
PhD candidate, Economics Informatics Department
Academy of Economic Studies Bucharest, Romania
Currently working at Intrasoft International S.A., leading company in providing software solutions for European Commission, as Senior Software Engineer/Technical Team Leader
Main capabilities and skills: advanced programming in Java/J2EE (SUN Certified), advanced UML and object-oriented design, project management and team leading experience.
Main published books and articles:
- Comparative analysis of the complexity of text entities generated using programming technique - Iulian Radulescu, Ion Ivan, ASE, Bucharest, 2006.
- Variations of software complexity for products developed using different programming techniques - Seventh International Conference on Informatics in Economy, Department of Informatics in Economy, ASE, May 2005.
- Analysis of the correlation between software quality characteristics – SIMPEC Symphosium, Brasov, May 2006.

[11] Thomas J. McCabe A**. Complexity Measure**, IEEE Transactions on Software Engineering, VOL. SE-2, No. 4, December 1976