Volume IV/ Issue 3(9)/ Fall 2009

A STUDY OF QUALITY IN THE PROCESS OF SOFTWARE PRODUCT DEVELOPMENT ACCORDING TO MAINTAINABILITY AND REUSABILITY

Ion **BULIGIU**, Liviu **CIORA**, Andy **ŞTEFĂNESCU**
University of Craiova, **Romania**
Facultaty of Economic and Business Administration
buligiu_ion@yahoo.com

Abstract

Our research indicates the modality by which algorithm modifications imply intervention in modules where expressions are evaluated or selections of elements are performed, the conclusion being the fact that in order to design robust software, a clear definition of the modules is necessary. Thus, the weak module which can be easily modified must be defined and placed so as not to affect other modules through modifications applied to them.

The reusability issue is even more important as the main software producing companies have developed class libraries which reduce programming efforts. It is thus possible to start the realisation of software with personnel no larger than 15 people, but with high qualification and logistical resources. The problem of reusability occurs especially in the interference area. In designing interfaces, graphic elements are dominant, as well as those of information search-find. All this implies the definition of text placement and designing parts of the text which determine actions or operation selection.

The problem of reusability occurs when in new software products conversions, compressions, sorting and optimisations as operations with extremely low proportion in the computing volume must be introduced, but which represent significant consume from the point of view of the programming effort.

Keywords: software products, quality, software maintainability and reusability.

JEL Classification: L86

1. Introduction

In the IT domain, there are fundamental difficulties regarding product measurement, especially in the case of software products. On a closer look to the concept of quality, especially project quality, we discover that human originality and creativity are closely connected to it. These aspects of quality are hard to measure, especially since computer programmers see their work as a work of art rather than a commercial product.

According to software quality, the necessity to supply a solution appropriate to the user's needs is often considered to be *design quality,* while conformity to specifications is *production quality*. The classical cycle of a software product development resolves this ambiguity addressing the design and production in different phases of the process. The quantitative side is dominant, being the reason why class definition to table defining and data computing is extremely important.

Quality is an integral part of achieving an IT product. While some aspects of quality can be improved after implementation (for example supplementary hardware can be added to improve performance), others, such as software reliability and maintainability are based on including direct quality into the product, so that if the improvement of its quality is desired, it can be achieved only by re-design and re-development, to a higher price than if the improvement had been done during its initial development.

Starting from the definition of quality as it is formulated by the ISO 9000:2000 standard [Ince, (1996)], a quality IT product that which fits the purpose and which satisfies the client's demands, both expressed and implicit.

The requests expressed for an IT product are those established in the users' specifications, which must include a complete list of specifications, necessary to satisfy the client's needs. These specifications may include, but are not limited to, availability, reliability, usability, efficiency, maintainability and constraints in using the IT products and they have to be clearly defined by characteristics which can be observed and evaluated by the user.

Users' specifications for an IT product can be done by: specification regarding the product, specification regarding selected technical configuration and specification of design objectives for a product's application in the case of specific usage. Satisfying implicit requirements is more difficult, especially since they must be specified.

In the case of developing an It product, as a result of a contract or internal understanding, according to theory, all specifications should be mentioned. The ISO 9001 standard assigns it to the responsibility of the producer to make sure that the specifications are clearly defined and appropriately documented.

Within this context, implicit specifications are specifications so obvious that nobody considers they should be included in the specification. These specifications may be those referring to areas outside the competence of those defining specification (which is often related to delivering the IT product), determined by the nature of the product or service, by the client's manner of organising or the environment in which it operates.

We consider that defining IT products quality is much more difficult, compared to other products judging by the quantification difficulties of the qualitative attributes.

The main and derivative characteristics of software product quality according to the quality model defined by the ISO / IEC 9126 are described in the following table.

Table 1. Main and derivative characteristics of software product quality

No.	Characteristics	Derivate characteristics and meaning value
1	**Functionability**	▪ *Adequacy*: the presence and adequacy of the functions set according to specifications. ▪ *Accuracy*: the attributes of the software product related to obtaining correct or agreed results (for example the necessary precision degree of the values). ▪ *Interoperability*: the possibility of interaction between the software product and other specified products. ▪ *Conformity*: conformity with standards, conventions, regulation and other similar prescriptions connected to the application domain. ▪ *Security*: the software product's possibility to prevent unauthorised access, accidental or deliberate, to programs or data.
2	**Reliability**	▪ *Maturity*: the software product's degree of maturity, respectively the failure frequency on account of the software product errors. ▪ *Fault tolerance*: the software product's capacity to maintain a certain specified level of performance in the case of error. ▪ *Recoverability*: the possibility of re-establishing the level of performance and data recovery in case of error, the time and effort necessary for it.
3	**Usability**	▪ *Understanding facility*: the quick understanding, the user's effort to recognise the logic concept and its applicability. ▪ *Operability*: easy operating, respectively adapting the interaction style and the user interface type. ▪ *Learning facility*: the quick learning of the product's applicability.
4	**Efficiency**	▪ *Durability*: the efficiency as time response on processing operations, transfer rates in various conditions and configurations. ▪ *Resource behaviour*: the memory consumption in various conditions.
5	**Maintainability**	▪ *Analysability*: the rapidity and precision of identifying an error in execution between the software product messages and its causes. ▪ *Changeability*: the necessary effort for changing, repairing the error or for changing the environment. ▪ *Stability*: it refers to the risk of unexpected defect of modifications. ▪ *Testability*: it refers to the effort necessary for validating the modified software.
6	**Portability**	▪ *Adaptability*: the adaptability to other specific environments without using other facilities than those specific to the software product. ▪ *Easy installing*: the possibility of easily installing the software product in a specified environment. ▪ *Conformity*: the degree of the software product's adherence to standards and reglementations related to portability. ▪ *Interchangeability*: the possibility and the effort of using it instead of another specified software, in that software's environment.

2. Software maintainability

Maintainability is a process specific to software products destined to function for a long time period that is more than three years. In time, due to the evolution of technologic processes, legislation modifications and structural modifications of collectivities, adopting software products so as to answer the users' real demands is necessary.

Modifications of algorithms imply intervention in modules where expressions are evaluated or elements selections are performed. In order to elaborate maintainable software it is necessary to define modules clearly enough. Thus, modules possibly subjected to changes must be defined and placed so as not to affect other modules through modifications operated on them.

Modifications of input data imply increasing the data volume subjected to processing, introducing new variables to describe collectivity's elements or changing the representation model. In all these cases, modifications within the modules are necessary to maintain performance at a transaction level and to work on the new data. The program is maintainable if it accepts data modifications through similar processes. That is, when adding fields, processing modules are also added, when adding articles, modules of realisation rapid access are also added. When eliminating fields, processing modules are also eliminated or modules are deactivated.

Modifications at hardware level imply rethinking the product so as to accept hardware modifications. The depth of these modifications is in most cases so great that it is preferable to acquire a new product. Since there is an abundance of free circulating software, the problem of maintainability has a different connotation. The existence of extremely cheap products found on disks or CDs makes the users change their position towards acquiring maintainable software, eliminating it from the beginning. What is more, the user is after the products' structural dynamics for which it develops IT software applications and develops conditions to pass in very short periods of time from one product to another, each developed according to a new concept, with other hardware structures.

The problem of maintainability is shifted in this case towards the data that is a large variability of storage volume which may be processed by any software product.

Maintenance at result level is seen as the necessity of obtaining results in the exact shape and quality demanded by the user. Software producers have the obligation to take into consideration structures of results necessary to the users. Software products will be thus designed so as, by specific operations, to offer users the possibility to change result structures. In order to ensure maintainability, a series of measures will be taken into consideration, the most important of them being:

▪ defining reserves / back-up on support for each article, to allocate new fields while the development of the information data base to describe processes or elements of a colectivity;

▪ building modified expressions which, through values of coefficients to allow including or excluding some factors; for example, if the initial program evaluates the expression:

$$expr = a + b + c + d; \tag{1.1}$$

and if this expression is susceptible to be modified by decreasing or by eliminating some terms, the following form is implemented:

$$\exp r = \sum_{i=1}^{4} \alpha_i \cdot x_i \tag{1.2}$$

where: x_1 – is set in correspondence to a; x_2 – is set in correspondence to b; x_3 – is set in correspondence to c; x_4 – is set in correspondence to d.

For:

$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 1$ results e = a + b + c + d.

$\alpha_1 = 1,\ \alpha_2 = -1,\ \alpha_3 = 1,\ \alpha_4 = -1$ results e = a - b + c - d.

$\alpha_1 = 1,\ \alpha_2 = 0,\ \alpha_3 = 0,\ \alpha_4 = -1$ results e = a - d. (1.3.)

Introducing elements which ensure variability in performing selections; thus the function:

$$f(x) = \begin{cases} g_1(x),\ if\ \ x \in [a_1, b_1] \\ g_2(x),\ if\ \ x \in [a_2, b_2] \\ \dots\dots\dots\dots\dots\dots \\ g_n(x),\ if\ \ x \in [a_n, b_n] \end{cases} \tag{1.4}$$

implies the use of a massive one-dimensional with cu *n+1* components, a_1, a_2 , …, a_{n+1} is implemented by defining the number of intervals *n*, by allotting memory for the *n+1* components of the massive one-dimensional *a[]* and by defining analytical expressions of the n functions $g_i(x)$, $i = 1, 2, ..., n$; all as input data; the maintainable program is endowed with an interpreter which uses analytic expressions of the functions $g_i(x)$ and evaluates them.

Maintainability compared to hardware evolution is possible in design phase by including elements which would accept the following modifications.

As a rule, the new computer generations accept products created for previous generations. The disadvantages stem from the impossibility to use the facilities available on the new computers.

For example, a product developed for a computer in which floating precision calculus were emulated by using procedures will not use the co-processor's facilities.

Moreover, developing in the multimedia field implies increasing the software products' capacity to operate with images and sounds. Attaching components which allow input/output multimedia operations compatible is a strong point in maintainability manifestation area.

Maintainability is measured using the metric I_{ment}:

$$I_{ment} = \frac{T_{modif}}{T_{dezv}} \tag{1.5}$$

where: *Tmodif* – the time needed for operating modifications in the software product to maintain it in current use; *Tdezv* – the time needed for the development of the product (analysis, design, encripting testing, and implementation).

Practice shows that if $0{,}6 > I_{ment} > 0{,}4$ the decision to replace the product in the near future is necessary since the future maintenance demands will imply very high costs.

If constantly $I_{ment} > 1$ this means that the product was not designed to fulfil new demands. In present circumstances, evaluating maintenance on a source text does not prove to be eloquent due to various ways of ensuring maintenance, including by building translators which have the role to modify source texts, bringing them to new demands imposed by the user.

Software products developed according to component-based techniques have real maintenance processes, with a minimum realization effort.

3. Software reusability

Object-oriented programming is the direct result of the need to design re-usable software. Embedding, an essential characteristic of the objects refers to isolating in a single entity of the operands and operators (methods). When the object is defined, all the elements are taken into consideration so as to ensure a complete processing. The operands and operators cover a sufficiently large area and through guaranteeing the correctness of the calculus and generality they are processed as such.

If a programs library implies the proceedings existence (methods) the operands being left to users both for defining and initialising, the objects exclude the users' contribution especially in producing errors in defining operands and their initialising. Defining and using objects is possible only when in programming languages specific mechanisms to dynamic allocation and differentiated treatment of operands and operators are implemented by adding properties regarding access, reference and domains.

Legacy is the most obvious way of re-using software at the level of applications development. Legacy creates the possibility that what exists to be able to add new properties by building derivate classes.

Polymorphism ensures working independence of the programmers without further supplementary restrictions on the way of defining functions different as structure which create the same processing procedure (writing, calculus, sorting, and drawing). Reusing software is possible if in module design processing correctness and generality is ensured.

Reusing ensures work consumption reduction and leads to shorting the time necessary for creating a software product. First of all, those who develop software must know exactly what exists, what module is available, the using way and how much they are processed or are available.

Conditions for software reusability are:

- The components to fully develop the required processing function;

▪ The qualitative level of the reusable component must be superior or at least equal to that of the product to be realised;

▪ Concordance between data structures which the product under construction operates on and the data structures of the reused component both regarding its input and output;

▪ The availability of the component by taking it from another software product of the firm or by purchasing it at convenient price;

▪ Homogeneity from the point of view of hardware and software demands compared to the developing product demanding reusing.

Software reusing becomes operational when the available components fulfil all conditions and convince the programmer about their usefulness in his activity.

The problem with reusing is the more important the more basic software producing firms have developed class libraries which reduce the programming effort by 60 to 80% by taking over. It is thus possible to start high complexity software even in software producing firm with staff no larger than 15 people, but with enough equipment and high level qualification. The problem of reusing mainly occurs in the area of interfaces interferences.

In achieving interferences, graphic and search-find information elements are predominant. All these imply defining text placing and constituting parts of the text which determine actions or operation selection. The quantitative side is dominant, reason why it is important to define classes oriented towards developing interferences. Moreover, interferences are developed to define tables and process the data in the tables. The problem of reusing occurs mainly when in the new software products conversions, compressions, sorting, optimisations must be included as operations with extremely low percentage within processing volume, but from the point of view of the programming effort significant consumptions are involved.

The reusability degree RD is seen in the relation:

$$RD = \frac{LR}{LT} \tag{1.6}$$

where: LR – length as number of instructions or Kbytes of the reused components included in the considered software product; LT – total length as number of instructions or Kbytes of the software product in which the components have been reused.

For example, in the case of a program which implements operations on the matrix necessary to comparing levels of characteristics of a definite number of users of software applications series of 15 procedures summing up LR = 362 source code lines, were reused from a matrix functions and procedures library. Since the source code of the program contains *LT* = 1150 de instructions, the re-usage degree is: $GR = \frac{362}{1150} = 0{,}31$

When a program is analysed, it is necessary to identify: *LT* – the total length of the program; *LR* – the length of the components effectively reused; LR_{max} – the maximum length of the components that might have been reused.

In this case, the degree of work waste included in the software product is calculated, DS, in the relation: $DS = \frac{LR_{\max} - LR}{LT}$ (1.7)

As a rule, when a program is evaluated, the indicator DS is evaluated to bring its cost at the real level. Wasting work is a result of not knowing about the existence of software components, an aspect which is not charged on the user, but on the software producer.

4. Conclusions

Structural matrixes for object oriented design require a different approach from matrixes used for procedural architectures or data oriented since the programs are differently structured. In the present situation, the programs are structured around objects, which embed states, properties and possible actions for a certain object. Functional design is based on procedures, functions and models and on their decomposition in procedures or smaller and simpler modules. To conclude with, measuring structural complexity for the two types of codes is different. In order to exemplify the differentiation, we can use the example of the number of code lines for procedural programs – a useful element in studying complexity, but it is quite easy to realise that this parameter for object oriented programming does not have too much significance. Therefore, matrixes are required to show the complexity of the object oriented code and quantifying the implementation effort, in this case measuring is oriented towards classes, modularity, embedding, legacy and abstractisation.

The set of matrixes appropriate to this purpose is made up of six matrixes which measure the dimension of a class of objects and its complexity, using legacy, coupling classes, class cohesion and communication between object classes as described below.

- Model density in a class – this matrix studies the number and complexity of the methods in the object class. Complexity at class level are based on McCabe method, cyclomatic complexity and the number of code lines in the class studied, adding up complexities for all the methods in the class. This software matrix is an indicator of effort necessary for implementing and testing the class, higher values suggesting that the class is too big dimensionally, and therefore must be fractioned.
- The depth of the legacy tree – is the matrix analysing how many levels of legacy make up a class hierarchy. If the value of the matrix is high, it indicates a high project complexity, but a better re-usability of it.
- The number of instantiers – measures the number of immediate successors of the class. If this number is high, the weakening of the parent class abstractisation is observed, and then a more laborious testing is necessary, together with a better re-usability. An eventual re-designing might be considered.
- Coupling objects classes. It is a measurement of the way in which other classes are based on a certain call and vice versa. It is actually a measurement of the classes which are coupled. Two classes are considered coupled when the declared methods in a class use methods or variables instantiated by other classes. The high level of this matrix implies a higher complexity, reduces maintainability and reduces re-usability.
- Class response – represents the dimension of a set of methods which can be potentially launched in execution to answer a message received by the object containing them. It is calculated by adding up the number of response methods to the number of local methods subordinated to those previously counted and which supplies in turn response to the immediate superior level. Complexity increases proportionally to increasing responses for a class and implies a necessary supplement for testing.
- Lack of method cohesion. This matrix counts how many different methods in a class referentiate an instantial given variable. Design complexity and difficulty increase with the matrix increase.

For all the matrixes presented above, increasing value levels is correlated to the following effects: low productivity, higher effort in reusing classes, difficulties in class design and class implementation, high number of maintenance operations, a higher number of classes with malfunction/defect. Problems reported by users.

Reference levels through which each class comes to fulfil at least two of the established criteria and consequently they must be identified and investigated for an eventual redesign are: response for a class > 100, coupling between objects > 5, response for a class - 5 times higher than the number of methods in the class, density of methods in a class > 100 and number of methods > 40.

5. References

[1] Ince, D., (1996), *ISO 9001 and Software Quality Assurance*, McGraw-Hill, London.

[2] Jenner, M., (1995), *Software Quality Management and ISO 9001: How to Make Them Work for You*, Wiley & Sons, New York, NY.

[3] Jung, Ho-Won; Seung-Gweon, Kim; Chang-Shin, Chung, (2004), *Measuring software product quality: a survey of ISO/IEC 9126 Software*, IEEE Volume 21, Issue 5, Sept.-Oct. 2004.

[4] Pecht, M., (2009), *Product Reliability, Maintainability, and Supportability Handbook,* CRC Press, USA.

[5] Rosenberg, L., E.Stapko, A.Gallo, (1995), *Applying object oriented metrics*, ISSRE.

[6] Smith, David John, (2005), *Reliability, maintainability and risk*, Ed.Elsevier, Amsterdam.