

A new structural complexity metrics applied to Object Oriented design reliability assessment

Stéphanie Gaudan
Thales Avionics Toulouse, France
Stephanie.Gaudan@fr.thalesgroup.com

Gilles Motet and Guillaume Auriol
INSA-LESIA Toulouse, France
Firstname.Name@insa-toulouse.fr

Abstract

In avionics domain, the software applications grew to one million of source lines of code representing important development expenditures. To cut the costs, the avionics suppliers are studying the potential use of new software approaches such as Object-Oriented Technologies. These technologies reduce the development and maintenance costs, in particular, thanks to the use of the inheritance mechanism. However, these technologies also create a new structural complexity of the programs which is at the origin of new risks of faults. To be embedded in aircraft systems, the avionics certification authorities require the assurance of the control of these faults.

This paper proposes a new metrics which quantifies the intrinsic risk level of an object-oriented program. This metrics takes into account the influence of the object structure on the difficulty in identifying an element among the numerous others. It is based on the information theory and the entropy principle applied to the structure elements.

To highlight its benefits, the metrics is applied to a Java program of a flight manager module developed by the Thales Avionics compagny.

1 Introduction

1.1 Needs

Object-Oriented Technologies (OOT) present numerous advantages recognized in the world of the software engineering (Web, mobile phones, videos games, etc.). In particular, these technologies reduce development and maintenance costs, by notably favoring the reusability. Avionics software industries have a growing interest in these technologies to benefit from their advantages.

However, the safety requirements inherent to critical avionics domain impose a reliability analysis of these technologies.

Independantly of their application, any new technology creates new intrinsic risks. For modelling or programming languages, these risks concern in particular the design faults. For instance, the inheritance feature offered by the OOT is at the origin of new types of programming faults [1].

The use of the OOT features has an influence on the global complexity of a model or a program. When this complexity increases, the model or program mis-understandability increases too. This misappreciation is at the origin of fault introduction that decreases the reliability.

To retain the benefits provided by the OOT features, the aircraft manufacturers have to bring out to the certification authorities, guarantees on their technology control, by analysing and treating their risks. Today, no directive concerning the use of the OOT exists in the applicable standard (DO-178B [2]) because this last one dating 1992, did not integrate these technologies. The aircraft manufacturers are therefore responsible for making judicious choices on the technology use. They will have to justify these choices in order to convince the authorities. In particular, they must possess means to estimate the residual risks.

To use OOT, it is thus necessary to identify the risks of intrinsic faults, to propose means to estimate and to treat these risks.

The only existing document on this subject is produced by the group OOTiA (Object-Oriented Technology in Aviation) [3]. It contains a list of potential issues induced by the use of the OOT with regard to the avionics certification constraints. The design faults highlighted in the OOTiA document are actual. However, no detailed analysis of the causes is proposed (no identification of the risks). Associated guidelines are recommended, but their use brings no guarantee on fault risk reduction, because they are defined in an in-

tuitive way. For example, it is disadvised to use more than 6 levels of inheritance. But, the respect of this constraint brings no guarantee of the actual control of the faults associated with the inheritance mechanism (no estimation of the risk reduction).

Our work on the risks of OOT faults are based on the OOTiA document. To identify these risks, we proposed a model formalizing the sources of these risks [4]. Then, we exposed the need to estimate these risks. These assessments allow, for instance, to measure the efficiency of proposed guidelines [5].

In this paper, we present a novel metrics assessing the complexity of the object-oriented models or programs. The proposed metrics estimates the OOT fault risks by taking into account the structural aspects (classes, inheritances) as well as the elements (methods, attributes) contained in the object models or programs.

1.2 Structure and elements

The object-oriented design introduces new types of faults, notably due to the use of inherited methods. The presence of inheritance relationships between classes leads to the implicit propagation of elements (methods, attributes) through the structure (inheritance chains). This propagation can lead to the designer misappreciation of the elements accessible in a given class C and their characteristics (identifier, signature, contract, etc.). This misunderstanding is at the origin of specific faults as illustrated in [1]. It increases with the number of:

- elements (methods, attributes) accessible in C (i.e. locally defined or inherited),
- classes inherited by C , including multiple inheritances,
- inheritance levels between these classes and C .

For example, [1] mentions the risk of SDA (State Defined Anomaly): due to the redefinition of a method, the designer could break the contract defined by the overridden method. This type of fault is not automatically detectable. Indeed, the complexity of such analysis would not be tractable on the whole industrial application, because it would require the formalization of all the contracts of methods [6] and their check in a formal manner. This would generate a considerable quantity of proof obligations. Consequently, it is essential to propose an assessment of these risks of faults to focus analysis efforts on the most risky components.

To control these types of faults, it is necessary to identify and then to estimate the various factors of their risks. The structural characteristics of inheritances and

the elements distributed in this structure are at the origin of these types of faults. By consequence, we investigated the measures of object-oriented programs complexity taking these factors into account: elements and structure which contains them.

We present in the following section, works on the measurements of software complexity concerning the structure and the elements of the code.

1.3 Related works

Numerous metrics have been proposed to estimate the complexity of software programs. They aim at assessing the complexity that the designer has to master when he or she develops or reuses these programs. Thus, these metrics lead to the assessment of the reliability because the complexity directly influences the probability of presence of faults and thus of failures at run-time [7].

We put together the previous researches according to two main categories: the statistical metrics on the code, and the metrics based on the information theory defined by Shannon [8].

1.3.1 Statistical metrics

The statistical metrics are the most widely used. They consist in different statistical analysis of various code parameters. These metrics take into consideration various criteria as the number of Lines Of Code (LOC), the number of classes, the number of inheritances, the Depth of Inheritance Tree (DIT) [9], the Number Of Local Methods in a class (named NOM [10] or NLM [11]), etc.

Several articles compare these various metrics on program examples, notably in [12]. The authors deduce predictions from these assessments (by means of Poisson regression tree) in [13]. In [14], we find a state of the art of the existing object-oriented metrics, by considering the different levels of granularity: system, structure inheritance, classes, methods and their coupling.

In [15], the authors list thresholds on a set of software complexity metrics. Then, they confront them with feedbacks from C++ programs, to study the correlation between the examined metrics and the numbers of actual faults.

In [16], these metrics are criticised (i) for taking into account only one or two aspects of a program (size, data, controls, etc.) leading to incomplete judgments, and (ii) for only being applicable at advanced stages of the software development.

The experiments of [17] show that the total number of classes and the number of methods to be known plays a major role in the comprehensibility of a class.

All these metrics allow to estimate the complexity linked to certain aspects of the program (the structure, the length of the code, etc.). However, they do not take into account the coupling between the structure and the elements of the code. For instance, the metrics "average number of methods per class" does not make reference to the structure of the program in term of inheritance. The fact that the classes are in relationships by inheritance or not has no impact on this average value.

Some complexity metrics consider the coupling between components of a program. However, they require an analysis of the method bodies (method calls, attribute modifications, etc.). In this paper, we are only interested in the declarative aspect of the program elements. This will allow complexity values to be obtained at preliminary design stages.

1.3.2 Metrics based on the information theory

Numerous works use the information theory introduced by Shannon [8] to estimate the program complexity. Among these metrics, the criteria proposed by Halstead are the most widely used [18]. Other metrics based on the information theory and the entropy exist. They take the following criteria into account:

- the frequency of use of the operators in a program [19, 20, 21, 22],
- the distribution of the complexity of classes composing a project [23], or
- the various structural relationships proposed by the UML design formalism [24].

A synthesis of the software complexity measurements using entropy is proposed in [25].

A same estimation is generally obtained by applying these measurements to a class *C* locally possessing all its methods (see figure 1.a), and to a class *C* inheriting all its methods by a 2-level inheritance chain (see figure 1.b). In section 3.2, we will illustrate this result.

However, the structure complexity also influence considerably the comprehension of their elements as the number of elements contained in this structure. Indeed, the difficulty to identify an element which is at the origin of faults, depends on the number of elements but also on the structure organizing these elements.

For this reason, this paper introduces a new complexity metrics, *MESS* (as Metrics based on Entropy

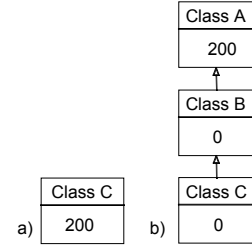


Figure 1. Two different hierarchies

for Structured Sets) coupling structure of inheritances and distribution of elements. This metrics will provide a complementary evaluation of the reliability of the object-oriented models or programs, as shown by its application in section 4.

1.4 Overview

The metrics *MESS* is defined in the section 2, in the theoretical framework of the structured sets. This metrics is instantiated on object-oriented programs in section 3. It is applied to various elementary object structures to highlight their consideration by our complexity metrics.

In section 4, we apply the metrics *MESS* and other complexity metrics to a Java code of a Flight Manager part, developed by Thales Avionics. We analyze the obtained results, and show the contributions of *MESS*.

2 Theory

2.1 Definitions

In this section, we define the theoretical framework of the structured sets. This step is necessary to understand the *MESS* metrics presentation.

A *structured set* of elements, called *E*, is composed of *k* own elements, called *e_i*, and of *p* disjoint subsets, called *E_j*. Formula 1 and figure 2 respectively formalizes and illustrates this structured set definition.

$$E = \bigcup_{i=1}^k e_i \cup \bigcup_{j=1}^p E_j \quad (\text{Formula 1})$$

Each directly accessible subset (called *direct subset*) is also a structured set (see figure 2).

Let *n* be the cardinal of *E*, noted | *E* |, that is, the number of elements belonging to *E*. These *n* elements of *E* are distributed locally (own elements) or among the subsets defined in the structured set *E*.

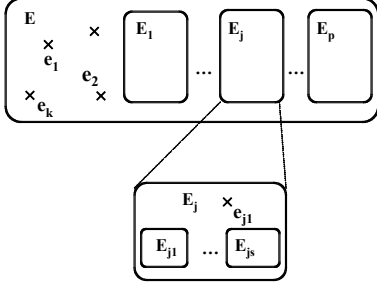


Figure 2. Structured sets

The *local cardinal* of E , noted $\|E\|$, is the total numbers of

- own elements of E , and
- direct subsets of E .

According to the formula 1,

$$\|E\| = k + p.$$

We define a measure of the information quantity required to identify an element within a structured set.

Let e be an element belonging to a structured set E and E' be the subset of E such as e is an own element of E' (see figure 3).

The information quantity required to identify the element e contained in E is called $I_E(e)$. $I_E(e)$ is the total of:

- the information quantity required to identify in E the subset E' , called $I_E(E')$, and
- the information quantity required to identify in E' the element e , called $I_{E'}(e)$.

Finally, we have:

$$I_E(e) = I_{E'}(e) + I_E(E') \quad (\text{Formula 2})$$

Let us explain the two parts involved in the formula 2.

The definition of $I_{E'}(e)$ is based on the information theory principle [8]. It is defined as the information amount required to identify e among the own elements and the subsets of E' , without discrimination between elements and subsets.

Then, according to Shannon [8], to distinguish e belonging to E' within the $\|E'\|$ own elements, it is necessary to have $\log_2(\|E'\|)$ information amount. Then, we obtain:

$$I_{E'}(e) = \log_2(\|E'\|) \quad (\text{Formula 3})$$

The definition of $I_E(E')$ is one of the contributions of this paper. Indeed, in comparison to the previous

metrics using the information theory (see section 1.3.2), this new metrics takes account of the structure and of the distribution of elements in E .

The definition of $I_E(E')$ is given by formula 4:

$$I_E(E') = \begin{cases} \log_2(\|E\|) & (1) \\ I_E(E_i) + I_{E_i}(E') & (2) \end{cases}$$

- (1) If E' is a direct subset of E
 - (2) Otherwise, with E_i direct subset of E such as $E' \subset E_i$
- (Formula 4)

To estimate the $I_E(E_i)$ information amount, we assume a naming of the elements in which they are prefixed relatively to the hierarchical access within a set. For example, on the figure 3, the element e would be prefixed by ' $EE_iE_{ii}E'e$ '.

It insures a direct access to the containing subset. We only have to distinguish the subset within the others. Thus, as E_i is a direct subset of E , we consider that

$$I_E(E_i) = \log_2(\|E\|).$$

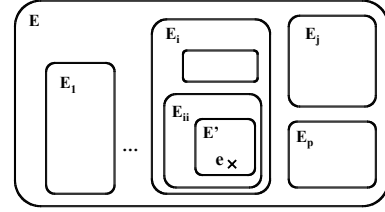


Figure 3. Structured sets involved in $I_E(e)$

According to the formulas 3 and 4, the value of $I_E(e)$ is obtained by the formula 2.

The entropy of E , that is, the average quantity of information associated with the elements of E , noted $MESS(E)$, is given by:

$$MESS(E) = \sum_{e \in E} \frac{1}{n} \times I_E(e) \quad (\text{Formula 5})$$

Let E'' be a structured set containing locally all the n elements of E (see figure 4).

Thus we have, $|E''| = \|E''\| = |E|$.

This set E'' has a minimum $MESS$ value. Thus, this set E'' is our reference to estimate the impact of the structure on our complexity metrics.

Let $MESSN$ be the normed metrics of $MESS$. In $MESSN(E)$, $MESS(E)$ is counterbalanced by $MESS(E'')$. This metrics is defined by:

$$MESSN(E) = \frac{MESS(E)}{MESS(E'')} \quad (\text{Formula 6})$$

with $MESS(E'') = \log_2(\|E''\|) = \log_2(|E|)$.

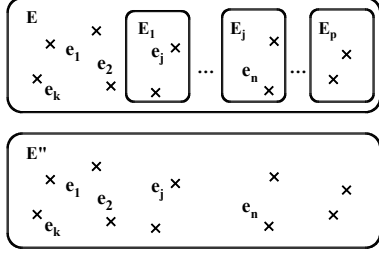


Figure 4. Reference set

$MESSN$ represents the influence of the structure of E on its complexity value. This new metrics can be used to choose a structure favoring the identification of the elements.

2.2 Illustration

In this section, we apply the metrics $MESS$ and $MESSN$ to an example of structured set. E contains two own elements (e_1 and e_2) and two subsets A and B respectively containing two and four own elements (see upper left part of figure 5).

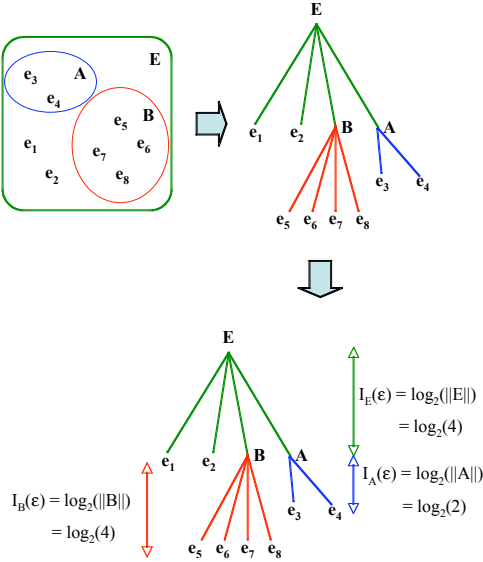


Figure 5. Metric principle illustration

We note $I_E(x)$ the quantity of required information amount to identify x among a set E , with x an own element (noted e_i in the formula 1) or a direct subset of the set E (noted E_j in the formula 1).

The application of the formula 2 provides the following results:

- $I_E(e_1) = I_E(E) + I_E(e_1) = 0 + \log_2(4) = 2$,

- $I_E(e_3) = I_E(A) + I_A(e_3) = \log_2(4) + \log_2(2) = 3$,
- $I_E(e_5) = I_E(B) + I_B(e_5) = \log_2(4) + \log_2(4) = 4$.

The used values are illustrated in the figure 5.

Moreover,

$$I_E(e_1) = I_E(e_2),$$

$$I_E(e_3) = I_E(e_4) \text{ and}$$

$$I_E(e_5) = I_E(e_6) = I_E(e_7) = I_E(e_8).$$

Applying the formula 5, we deduce:

$$MESS(E) = (2 \times 2 + 2 \times 3 + 4 \times 4)/8 = 3.25.$$

Furthermore, we have $MESS(E'') = \log_2(|E|) = \log_2(8) = 3$. Finally, using formula 6, we obtain

$$MESSN(E) = 3.25/3 \approx 1.08333.$$

3 Application to object-oriented languages

3.1 Principle

In this section, we apply the complexity metrics $MESS$ defined in the previous section within the framework of the object-oriented programs.

The object-oriented programs can be seen as structured sets. Each class C is represented by a structured set C . The methods declared in the class C are own elements of the set C . The classes inherited by the class C are subsets of C , which contain only inherited elements (private attributes and private methods are not inherited). Some examples of the relationships between object-oriented models and structured sets are illustrated in the figure 6.

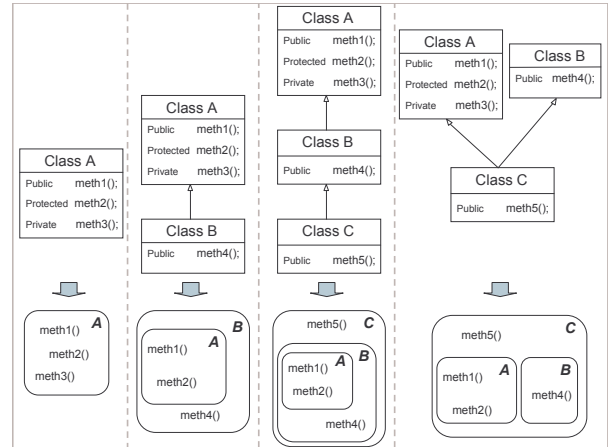


Figure 6. Relationships between structured set and object-oriented design

In this paper, for a given class, only the inherited or locally defined methods are considered. The assessment can be extended taking the attributes into account.

3.2 Structure and method distribution influences

In this subsection, we apply various metrics to various object-oriented models, to highlight the influence on the complexity of:

- the global structure,
- the distribution of the methods in this structure.

The compared metrics are the following ones:

- The proposed *MESS* and *MESSN*.
- CE assesses the entropy on the distribution of the use of the methods. This calculation is detailed in [26]. We make the hypothesis that the calls of each methods are equiprobable. Indeed, we focus on the declarative aspect of elements.
- NOM (Number Of Methods) [10] also called NLM (Number of Local Methods) [11].
- DIT [27] Depth of Inheritance Tree.
- NAC (Number of Ancestor Classes) [11] also called NOA (Number Of Ancestors) [28].

These complexity metrics are used to assess various object-oriented structures in which a class *A* provides 300 (local or inherited) methods. The results are supplied by the figure 7.

This figure shows that the metrics *MESS* (as the metrics DIT and NAC / NOA) takes into account the increasing complexity linked to the distance between the considered class and the classes in which the methods are declared. As identified by the OOTiA [3], this correlation is important, because the depth of the inheritance chains has an impact on the fault risk.

We note that, as introduced in section 1.3.2, the metrics CE does not take the different structures containing the 300 methods into account.

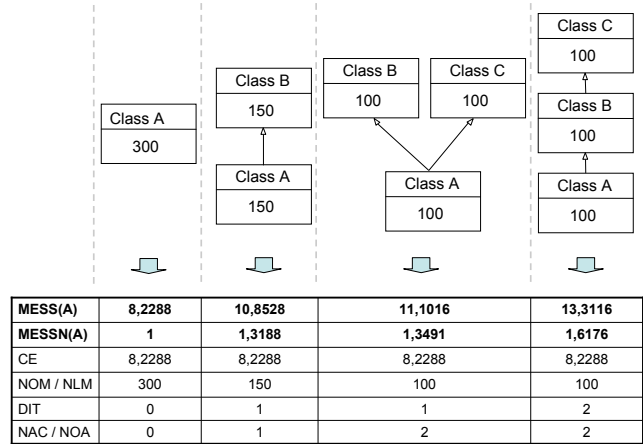


Figure 7. Structure influence

However, our metrics supply more precise estimations by taking the distribution of elements in the structure into account. This point is highlighted by applying the previous metrics of complexity to several distributions of 300 methods in the same inheritance chain. The results are supplied by the figure 8.

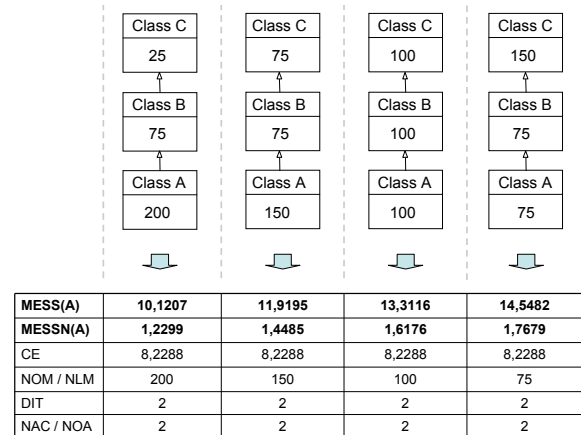


Figure 8. Method distribution influence

In this figure, we observe that the metrics DIT and NAC / NOA do not handle the complexity factor due to the distance between a given class and the class where the accessible elements are declared. The *MESS* metrics integrates this factor.

4 Assessment of the Java code of a flight manager module

We study a Java code module of a Flight Manager (FM) prototype developed by the Thales Avionics Company. This module is a package containing 50 classes (representing a set of 236 methods) and inheriting from 12 classes from the Java Sun API. On each of these classes, we calculate the following complexity metrics:

- AM (Available Methods): number of accessible methods, locally declared or inherited,
- DIT (Depth of Inheritance Tree),
- *MESSN* (our metrics).

First of all, we compare our metrics with AM. We represent in the figure 9, for each class of the examined code, (i) its complexity value according to the *MESSN* metrics vs. (ii) its number of available methods (AM).

We describe more precisely three significant couples of classes.

- *1st couple of classes C1 and C2, represented by points \odot on the figure 9.* The considered classes respectively named *C1* and *C2*, have respectively 61 (AM = 61) and 30 (AM = 30) accessible methods. A classical metrics like AM concludes that the complexity of *C1* one is twice as important as the *C2* one. However, the metrics *MESSN* gives equivalent measurements for the two classes because it also takes into account the distribution of these methods in the hierarchy, as it will be justified at the end of this section. Our metrics avoids overestimating the complexity of *C1*.
- *2nd couple of classes C3 and C4, represented by points '+' on the figure 9.* The Java Sun API supplies the class *C3*, which contains 41 methods (AM = 41). We notice that this class has a *MESSN* value almost equal to the *C4* one, which only has half of the accessible methods. This result will also be justified at the end of the section by the method distribution analysis.
- *3rd couple of classes C5 and C2, represented by points \ominus on the figure 9.* Although these classes have similar number of accessible methods (in *C2*, AM = 30 and in *C5*, AM = 31), their *MESSN* complexity values are quite different, as it will be justified at the end of this section.

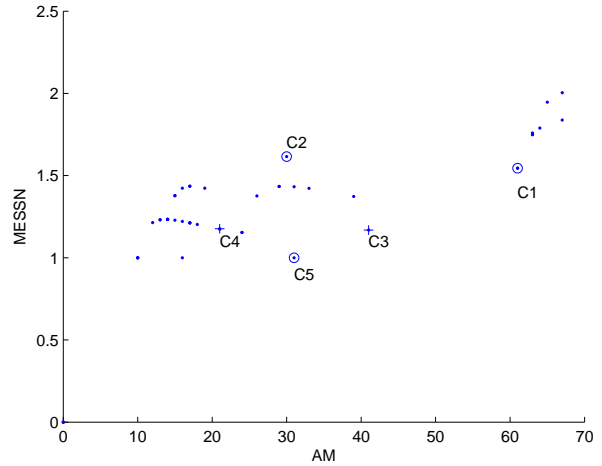


Figure 9. AM vs. MESSN

Then, we compare our metrics with DIT. For each class of the application, we represent in the figure 10 (i) the complexity according to the metrics *MESSN* vs. (ii) its number of inheritances DIT.

We study two significant couples of classes.

- *1st couple of classes C7 and C8, represented by points '+' on the figure 10.* This couple indicates that two structures with different DIT can have equivalent *MESSN* measurements because of other parameters of the structure.
- *2nd couple of classes, C3 and C6 represented by points '*' on the figure 10.* In contrast to the previous one, this couple shows that two classes having a same DIT value, could have quite different *MESSN* measurements.

As it will be explained at the end of this section, these results are justified by the fact that *MESSN* takes also into account the number and the distribution of accessible methods.

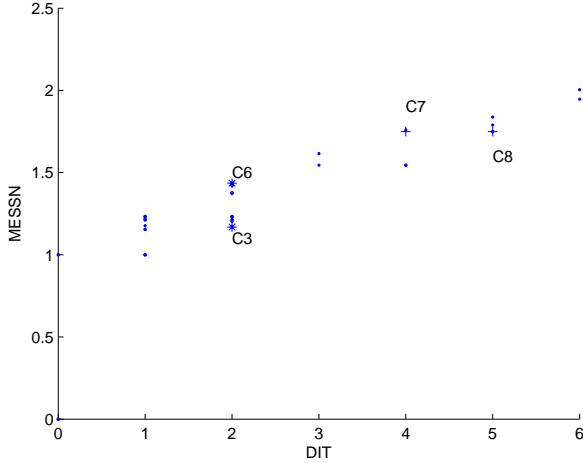


Figure 10. DIT vs. MESSN

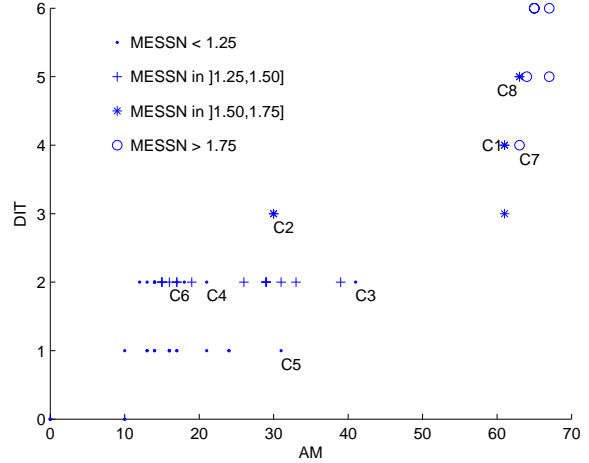


Figure 11. MESSN (AM, DIT)

Figure 11 finally represents on the same sample of classes, the effect of the coupling between the number of methods (AM) and the inheritance height (DIT) on the value of the *MESSN* complexity. We distinguish various complexity levels as indicated by the legend on the figure. The thresholds 1.25, 1.50 and 1.75 are arbitrarily chosen.

We examine the following classes that will explain the reasons for the previous observations.

- *Class C1*. *C1* is located at the 4th level of an inheritance chain (DIT = 4) and has access to 61 methods (AM = 61). This class has a *MESSN* value lower than 1.5. This measurement can be explained by a good distribution of the methods in the various levels of the hierarchy. Indeed, 2/3 of the methods accessible in *C1* are declared between *C1* and two levels of super-classes of *C1*.
- *Class C2*. On the opposite, this class, located at the 3rd level of inheritance (DIT = 3) and containing only 30 methods (AM = 30), has a *MESSN* value between 1.5 and 1.75. This measurement could be justified by an unfavorable distribution of the methods. Indeed, 5/6 of the methods accessible in *C2* are inherited from more than two inheritance levels. This *MESSN* value represents the difficulty to identify the accessible methods in this class.
- *Class C3*. The *MESSN* value is low because most (25/41) of the methods are locally defined.

Moreover, we remind on the figure 11 the previous studied classes.

It is important to keep in mind that, as illustrated in section 3.2 by the figure 8, two classes having identical AM and DIT values could have different *MESSN* values. This difference is due to the various method distributions in the structures.

To sum up, our experimentation reveals that the *MESSN* metrics takes into account the number of methods and the depth of the inheritance chain (Class *C1* and *C2* in the figure 11), but also the distribution of the methods within the structure. We notice that the classes inherited from the Java Sun API present *MESSN* values revealing good structures and good distributions of the methods. It means that they favour a good understanding of their elements for the designers.

5 Conclusion

5.1 Results synthesis

The *MESS* metrics application emphasises the classes from which the accessible elements (methods, attributes) are the most misappreciated. The normed metrics *MESSN* presents the advantage to cut out the intrinsic complexity of the applications. Indeed, the applications like Flight Manager have complex functions. However, during their design, our metrics help to choose a structure reducing the potentiality of faults due to the elements distribution. This metrics can be used from the preliminary design phases on UML diagrams for example. It leads the structure decisions towards the complexity control.

A more specific study has to be done to define *MESS* estimations thresholds beyond which the designer would misunderstand the accessible elements, and would potentially introduce faults. These vari-

ous thresholds will depend on criticality levels of the avionics applications.

To validate this metrics, we are going to confront the estimations obtained on the FM classes with fault identification and bug reports statistics on the various classes. It will put in evidence the correlation between the metrics values and the faults frequencies obtained by feedback.

5.2 From hazard assessment to risk assessment

The *MESS* metrics represents an assessment of the software complexity with regard to the *declarations* of code elements (accessible elements). In risk management terminology, it is a measure of the hazard, which is potentially at the origin of faults.

To make fault risk estimation, we have to consider the risk-taking expressed in a program. It implies to take the code and the body of the methods into account to observe the *use* of the code elements (method calls, reading of attributes, etc.). If some of these elements are not used in a class, they cannot be at the origin of any risk in this class. Thus, we don't have to pay attention to them for the risk estimation. Therefore, we will define a *MESS*-based metrics, taking into account the frequency of the use of elements (risk-taking estimate). This type of estimate would justify structures considered as dangerous by the *MESS* metrics, by an intense use of local elements and a weaker use of inherited elements. If the most used elements are easily accessible and identifiable, then the complexity value will decrease.

References

- [1] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson, "A fault model for subtype inheritance and polymorphism," in *Proceedings of the Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE'01)*, PRC, Ed. IEEE, November 2001, pp. 84–95.
- [2] D.-B. . ED-12B, "Software considerations in airborne systems and equipment certification," RTCA. Inc., EUROCAE, December 1992.
- [3] OOTiA, "Handbook for object-oriented technology in aviation," <http://shemesh.larc.nasa.gov/foot/>, Octobre 2004.
- [4] S. Gaudan, G. Motet, E. Jenn, and S. Leriche, "Identification model of the object-oriented technology's risks for an avionics certification," in *Proceedings of the 3rd European Congress Embedded Real-Time Software (ERTS06)*. Toulouse, France: SEE, 2006.
- [5] G. Motet and S. Gaudan, "Assessment of the risks for using object-oriented technologies in critical software," in *Proceedings of the 6th Workshop on Critical Software Tokyo*. Japan: JAXA publisher, 2006.
- [6] B. Meyer, *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., 1977.
- [7] V. S. Alagar, Q. Li, and O. S. Ormandjieva, "Assessment of maintainability in object-oriented software," in *TOOLS '01: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 194–206.
- [8] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, July and October 1948.
- [9] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in *OOP-SLA '91: Proceedings of the conference on Object-Oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM Press, 1991, pp. 197–211.
- [10] W. Li and S. Henry, "Maintenance metrics for the object oriented paradigm," in *Proceedings of the First International Software Metrics Symposium*. Baltimore, MD, USA: IEEE Computer Society, 21-22 May 1993, pp. 52–60.
- [11] W. Li, "Another metric suite for object-oriented programming," *Journal of Systems and Software*, vol. 44, no. 2, pp. 155–162, 1998.
- [12] L. C. Briand, J. Wüst, S. V. Ikonovskii, and H. Lounis, "Investigating quality factors in object-oriented designs: An industrial case study," in *Proceedings of the International Conference on Software Engineering*, 1999, pp. 345–354.
- [13] L. C. Briand and J. Wüst, "Modeling development effort in object-oriented systems using design properties," *IEEE Transactions on Software Engineering*, vol. 27, no. 11, pp. 963–986, 2001.

- [14] C. B. Archer, "Measuring object-oriented software products," (*CMU/SEI-C-28*). *Software Engineering Institute*, July 1995.
- [15] S. Benlarbi, K. E. Emam, N. Goel, and S. Rai, "Thresholds for object-oriented measures," in *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 24–38.
- [16] C. Wohlin, "Revisiting measurement of software complexity," in *APSEC '96: Proceedings of the Third Asia-Pacific Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 1996, p. 35.
- [17] B. Unger, L. Prechelt, and M. Philippsen, "The impact of inheritance depth on maintenance tasks: detailed description and evaluation of two experiment replications., Tech. Rep. iratr-1998-18, 1998.
- [18] M. H. Halstead, *Elements of Software Science*. New York, NY, USA: Elsevier Science Inc., 1977.
- [19] E. Berlinger, "An information theory based complexity measure," in *Proceedings of the 1980 National Computer Conference*. AFIPS Press: Reston VA, 1980, pp. 773–779.
- [20] W. Harrison, "An entropy-based measure of software complexity," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 1025–1029, 1992.
- [21] L. H. Etzkorn, S. Gholston, and W. E. Hughes, "A semantic entropy metric," *Journal of Software Maintenance*, vol. 14, no. 4, pp. 293–310, 2002.
- [22] C. Cook, "Information theory metric for assembly language," in *Software Engineering Strategies*, March/April 1993, pp. 52–60.
- [23] H. M. Olague, L. H. Etzkorn, and G. W. Cox, "An entropy-based approach to assessing object-oriented software maintainability and degradation - a method and case study," in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP 06)*, vol. 1, June 26-29 2006, pp. 642–652.
- [24] D. Kang, B. Xu, J. Lu, and W. C. Chu, "A complexity measure for ontology based on uml," in *FTDCS '04: Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 222–228.
- [25] S. K. Abd-El-Hafiz, "Entropies as measures of software information," in *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 110–117.
- [26] D. E. Knuth, *The Art of Computer Programming*. Boston, MA, USA: Addison-Wesley longman Publishing Co., Inc., 1998, vol. 3.
- [27] S. R. Chidamber and C. F. Kemerer, "A metric suite for object-oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.
- [28] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi, "A software complexity model of object-oriented systems," *Decision Support Systems*, vol. 13, no. 3-4, pp. 241–262, 1995.