

Metrics for Object-Oriented Software Reliability Assessment - Application to a Flight Manager

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 millions 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 structured sets.

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

1 Introduction

1.1 Needs

Object-Oriented Technologies (OOT) present advantages recognized in numerous software engineering domains (Web, mobile phones, games, etc.). 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 [1]. However, the safety requirements inherent to critical avionics domain impose a reliability analysis of these technologies.

Independently 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 [2].

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 misunderstandability increases too. This misappreciation is at the origin of fault introduction that decreases the reliability.

To take advantages of the OOT features, the aircraft manufacturers have to bring out to the certification authorities, guarantees on their technology control, by analyzing and treating their risks. Today, no directive concerning the use of the OOT exists in the applicable standard (DO-178B [3]). 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 control the risks of faults in their applications. Thus, the risks of intrinsic faults have to be identified to propose means to estimate and to treat these risks.

The only global document dealing with OOT fault risk identification was produced by the group OOTiA (Object-Oriented Technology in Aviation) [4]. It contains a list of potential issues induced by the use of the OOT regarding to the avionics certification constraints. The design faults highlighted in the OOTiA document are actual. However, no detailed analysis of their 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 intuitive way. For example, it is disadvised to use more than 6 levels of inheritance. But, the respect of this constraint does not provide assessed guarantees that the faults associated with the inheritance mechanism are controlled (no estimation of the risk reduction).

Our work on the risks of OOT faults is based on the OOTiA document. To identify these risks, we proposed a model formalizing the sources of these risks [5]. Then, we presented the need to estimate these risks. These assessments allow, for instance, to measure the efficiency of proposed guidelines [6].

In this paper, we present a metrics assessing the complexity of the object-oriented models or programs. The proposed metrics estimates the OOT fault risks 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 causes an 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 of their characteristics (identifier, signature, contract, etc.). This misunderstanding is at the origin of specific faults as illustrated in [2]. 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, [2] mentions the risk of SDA (State Defined Anomaly). Due to the redefinition of a method, the designer could violate 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 [7] and their check in a formal way. 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 structure of an object-oriented model or program and the distribution of the elements in this structure are at the origin of these types of faults. By consequence, we investigated the measures of object-oriented programs complexity taking the following 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. 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 [8]. We focus on two main categories: the statistical metrics on the code, and the metrics based on the information theory defined by Shannon [9].

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 consider criteria such as the number of Lines Of Code (LOC), the number of classes, the number of inheritances, the Depth of Inheritance Tree (DIT) [10], the Number Of Local Methods in a class (named NOM [11] or NLM [12]), the program control graph complexity (McCabe [13]), etc.

Several papers apply these metrics on program examples, notably in [14]. The authors deduce predictions from these assessments (by means of Poisson regression tree) in [15]. In [16], a state of the art of the existing object-oriented metrics is proposed, by considering different levels of granularity: system, structure inheritance, classes, methods and their coupling.

In [17], the authors list thresholds on a set of software complexity metrics. They confront them with feedbacks from C++ programs, to study the correlation between the examined metrics and the numbers of actual faults.

In [18], these metrics are criticized: (i) they only take into account one or two aspects of a program (size, data, controls, etc.) leading to incomplete judgments, and (ii) they are only applicable at advanced stages of the software development.

The experiments exposed in [19] show that the total number of classes and the number of methods to be known play a major role in the understanding of an object-oriented program.

All these metrics allow to estimate the complexity associated with certain aspects of the program (the structure, the length of the code, etc.). However, most of them 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 have relationships by inheritance 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, and thus reliability assessments, 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 [9] to estimate the program complexity. Among these metrics, the criteria proposed by Halstead are the most widely used [20]. 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 [21, 22, 23, 24],
- the distribution of the complexity of classes composing a project [25], or
- the various structural relationships proposed by the UML design formalism [26].

A synthesis of the software complexity measurements using entropy is proposed in [27].

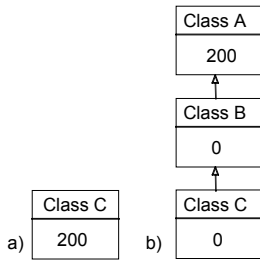


Figure 1. Two different hierarchies

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-levels inheritance chain (see figure 1.b).

The elements understanding depends on their number. it is also influenced by the complexity of the structure organizing these elements.

For this reason, this paper introduces a new complexity metrics, $MESS$ (as Metrics based on Entropy for Structured Sets) coupling structure of inheritances and distribution of elements. This metrics will provide a complementary assessment 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 used on to various elementary object structures to highlight that our metrics takes the elements organization into account. In section 4, we apply the metrics $MESS$ and other complexity metrics to a Java code of a Flight Manager, 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*, 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 .

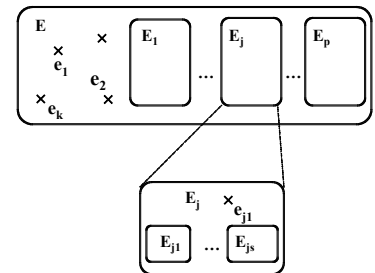


Figure 2. Structured sets

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

$I_{E'}(e)$ defines the information amount required to identify e among the own elements and the subsets of E' , without discrimination between elements and subsets. According to Shannon information theory [9], to distinguish e belonging to E' within the $\|E'\|$ components, it is necessary to have $\log_2(\|E'\|)$ information amount. So,

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

The definition of $I_E(E')$ is one of the contributions of this paper. Indeed, compared with 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 path. For example, on the figure 3, the element e would be prefixed by $EE_iE_{ii}E'$. It insures a direct access to the subset which contains e . We only have to distinguish the subset within the others. Thus, as E_i is a direct subset of E ,

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

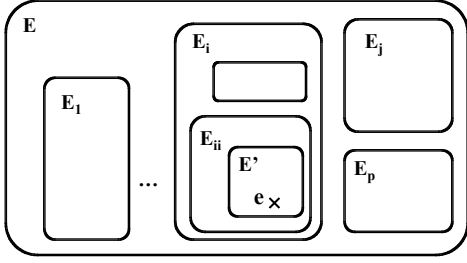


Figure 3. Structured sets involved in $I_E(e)$

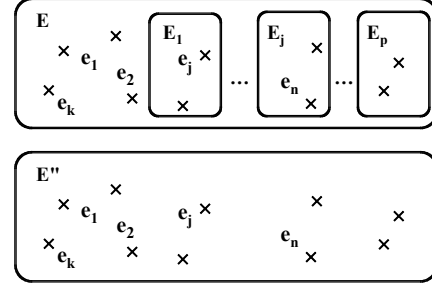


Figure 4. Reference set

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 amount of information associated with the elements of E , noted $MESS(E)$, is defined 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, $\|E''\| = \|E\|$. 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 $MESS(E'') = \log_2(\|E''\|) = \log_2(\|E\|)$.

$MESSN$ assesses the influence of the structure of E . This new metrics can be used to choose a structure favoring the identification of the elements and so the reliability.

2.2 Illustration

In this section, we apply the metrics $MESS$ and $MESSN$ to an example of a 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 left part of figure 5).

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

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 origins of the values used in the formula are illustrated in the figure 5 (right part).

Moreover, $I_E(e_1) = I_E(e_2)$, $I_E(e_3) = I_E(e_4)$ 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, $MESS(E'') = \log_2(|E|) = \log_2(8) = 3$.

Finally, using formula 6, we obtain $MESSN(E) = 3.25/3 \approx 1.08333$.

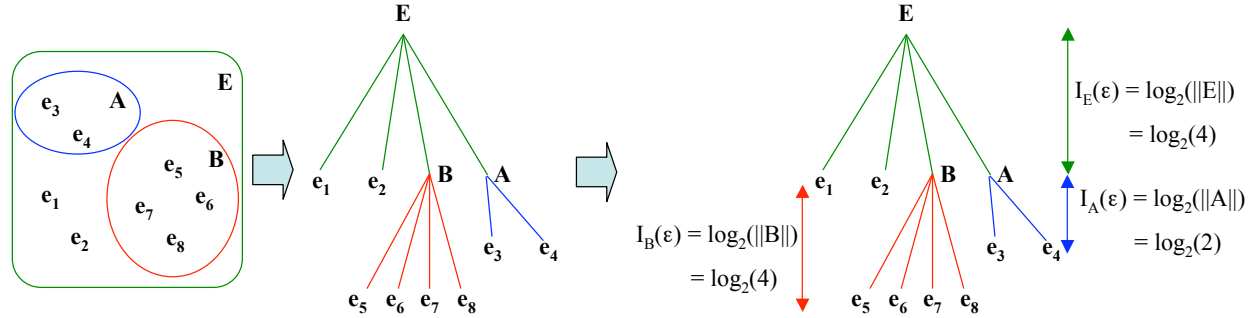


Figure 5. Metrics principle illustration

3 Application to object-oriented languages

3.1 Principle

In this section, we apply the metrics $MESS$ to 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.

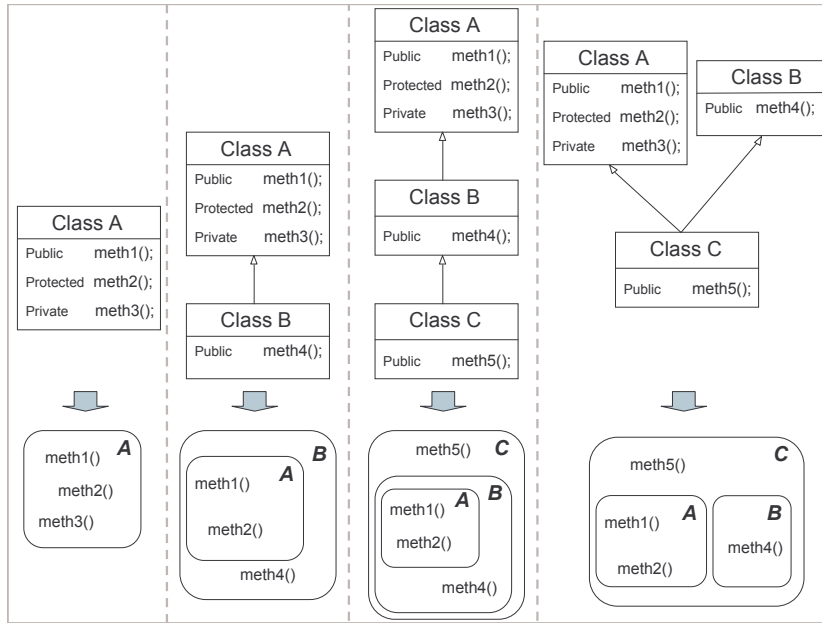


Figure 6. Relationships between structured set and object-oriented design

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, as well as of the distribution of the methods in this structure.

The compared metrics are the following ones:

- The proposed *MESS* and *MESSN*.
- CE (Classical Entropy) assesses the entropy of the distribution of the use of the methods. This calculation is detailed in [28]. We assume that the calls of each method are equiprobable. Indeed, we focus on the declarative aspect of elements.
- NOM (Number Of Methods) [11] also called NLM (Number of Local Methods) [12].
- DIT [29] Depth of Inheritance Tree.
- NAC (Number of Ancestor Classes) [12] also called NOA (Number Of Ancestors) [30].

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 on 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

inherited methods are declared. As identified by the OOTiA [4], 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.

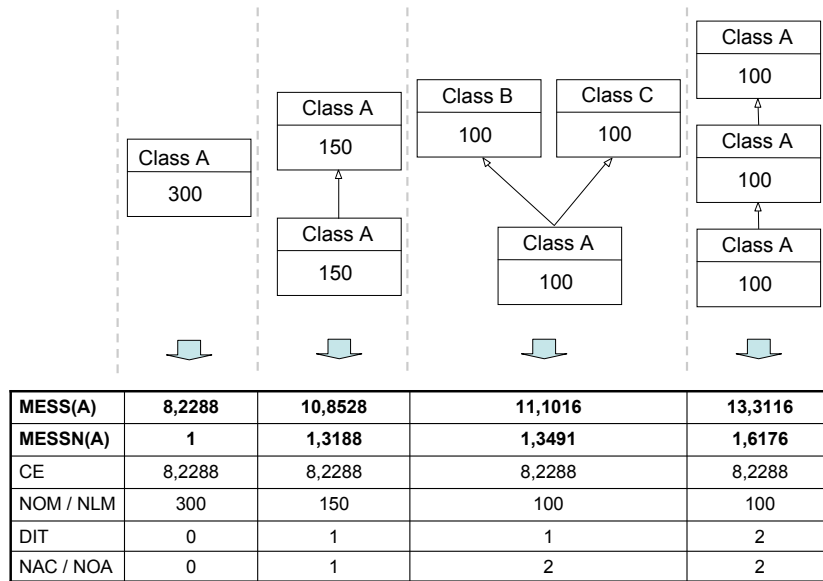


Figure 7. Structure influence

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

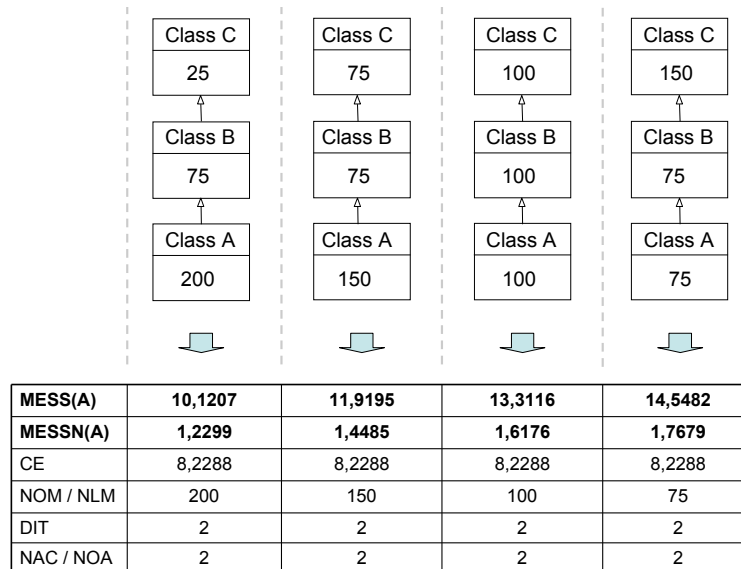


Figure 8. Method distribution influence

On this figure, we observe that the metrics DIT and NAC / NOA do not handle the distribution of the accessible elements in the structure. The *MESS* metrics integrates this factor.

4 Assessment of the Java code of a flight manager

In this section, we propose two bench tests on code modules of a prototype of a flight manager software. First, we compare the *MESS* metrics with other metrics in order to illustrate that the *MESS* metrics integrates several code characteristics (structure and elements distribution). Then, we establish empirically with the second experimentation that the *MESS* metrics is a good predictor of the presence of faults in a program.

4.1 First experimentation : metrics comparison

We study a Java code module of a Flight Manager (FM) prototype developed by Thales Avionics. This module is a package containing 50 classes (with 236 methods) and inheriting from 12 classes of the API Java J2SE. For each class, 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, our metrics is compared to AM. We represent on figure 9, for each class of the examined code, (i) its *MESSN* value vs. (ii) the number of available methods (AM).

Let us mention 3 couples of results to highlight differences in the results provided by *MESSN* and AM. These differences will be justified at the end of this section. Remark: The classes have been renamed to protect industrial confidentiality.

- Consider the 1st couple of classes 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. Thus, a conventional metrics like AM concludes that the complexity of *C1* is twice as important as the complexity of *C2*. 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. Our metrics avoids overestimating the complexity of *C1*.
- Consider the 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) and *C4* which accesses to 21 methods. We notice that the two classes have the same *MESSN* value even if the accessible methods number of *C4* is half as important as *C3*.

- Consider the 3rd couple of classes $C5$ and $C2$, represented by points ' \odot ' 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.

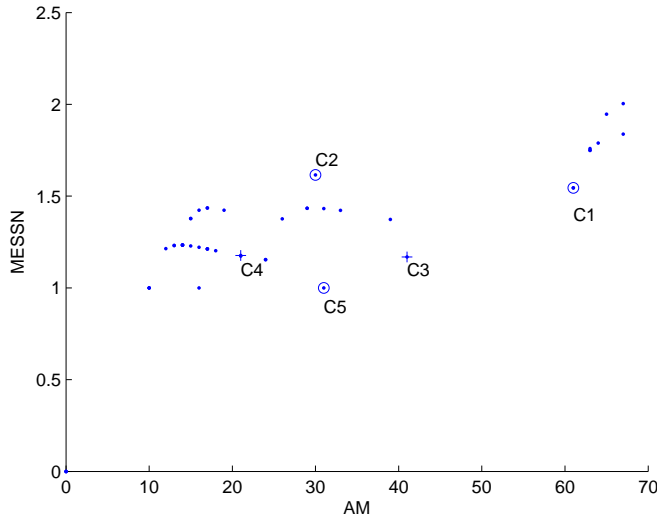


Figure 9. AM vs. MESSN

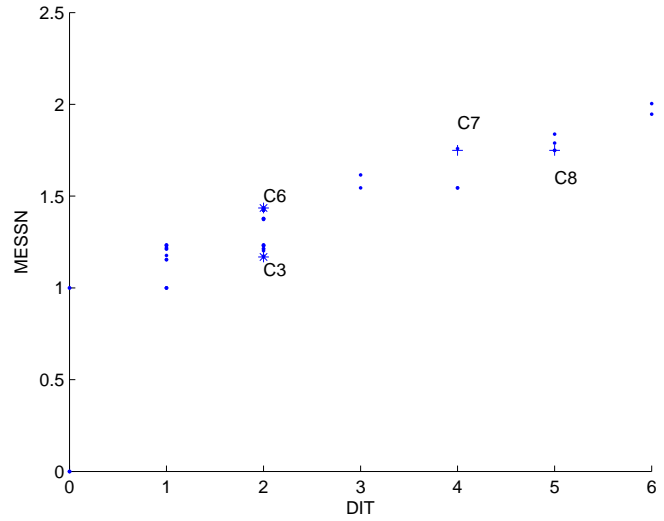


Figure 10. DIT vs. MESSN

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

We study two significant couples of classes.

- Consider the 1st couple of classes $C7$ and $C8$, represented by points '+' on figure 10. This couple indicates that two structures with different DIT can have equivalent $MESSN$ measurements because of other characteristics of the structure than the depth of inheritance.
- Consider the 2nd couple of classes, $C3$ and $C6$ represented by points '*' on figure 10. Contrary 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 structural distribution of accessible methods.

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

We examine the following classes that will explain the reasons of 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 close to 1.5. This small 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 medium measurement is 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 (≤ 1.25) is low because most of the methods (25/41) are locally defined.

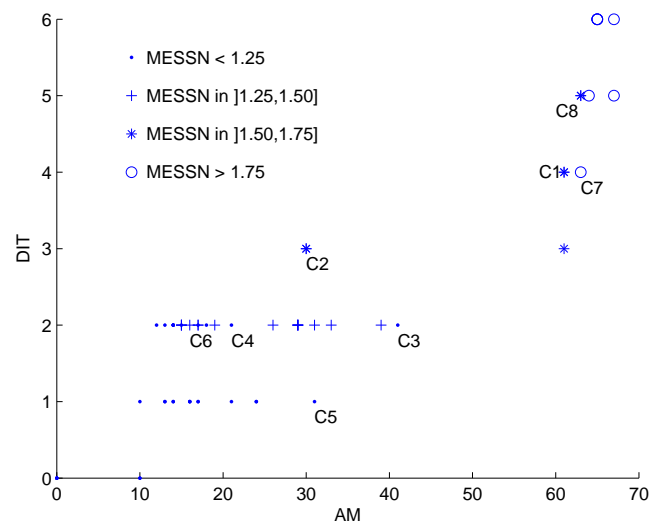


Figure 11. MESSN (AM, DIT)

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 *MESS* 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.

4.2 Second experimentation: *MESS* validation

The objective of this second experimentation is to show on the flight manager, that the *MESS* metrics is a good predictor of fault presence in object-oriented programs.

We collected statistical data on the presence of faults, or dangerous use of object features in the experimented code, with two static analysis tools of Java programs:

- FindBugs [31] and
- the Eclipse platform TPTP (Test & Performance Tools Platform project).

These tools identify by static analysis, fault risks such as erroneous use of feature. For example, the following situations are identified:

- Finalizer does not call superclass finalizer,
- Superclass uses subclass during initialization,
- Confusing method names,
- Class defines field that masks a superclass field,
- Read of unwritten field,
- Method ignores return value,
- etc.

Thanks to these tools, we entered the identified fault risk numbers for each class. As classes have very different sizes (from 197 to 15707 bytes), we calculate the fault risk number by class, normed per 1000 bytes.

Then, we compare the fault frequency obtained per 1000 bytes for the classes of the considered module, with values obtained by conventional object metrics. We first compare values obtained with the NLM metrics (Number of Local Method), then with the AM metrics (Available Methods, local or inherited), and finally with the DIT metrics (Depth of Inheritance Tree).

To conclude, we underline the lack of fault prediction of the conventional metrics and we illustrate the correlation existing between the fault frequencies and the values obtained by applying the *MESS* metrics on the experimented classes.

On the figures proposed in the next sub-sections, each rhombus represents a class, the abscissa represents the fault risk frequency obtained for the class (per 1000 bytes), and the ordinate represents the value of the studied metrics.

4.3 Fault risk vs. NLM

On figure 12, the rhombus represent the values obtained for each class with the NLM metric (Number of Local Methods), relatively to each class fault frequency (per 1000 bytes).

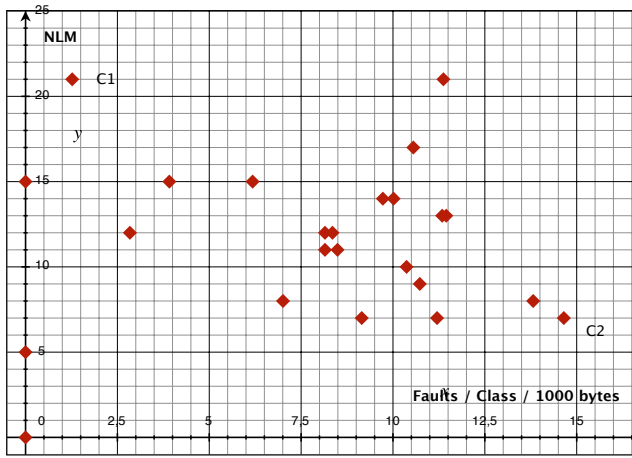


Figure 12. NLM vs. Fault risk / Class / 1000 bytes

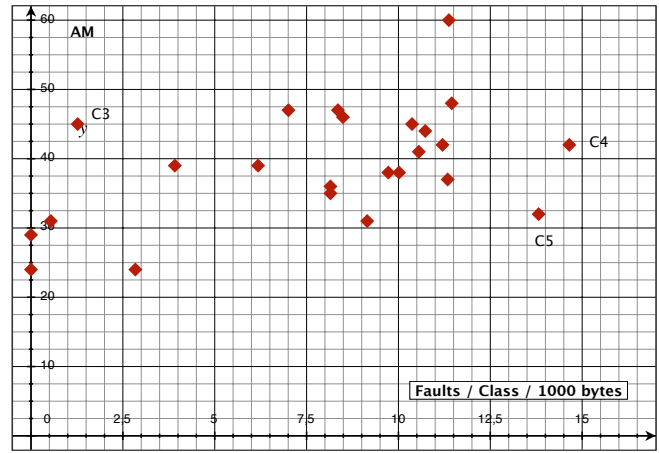


Figure 13. AM vs. Fault risk / Class / 1000 bytes

We observe on this figure that the rhombus are not aligned. For example, a class $C1$ which owns locally more than 20 methods has a fault risk lower than 2 per 1000 bytes, whereas another class $C2$ owning only 7 local methods has a fault risk close to 15. Consequently, the NLM metrics seems not to be a good predictor of fault presence for our case study.

4.4 Fault risk vs. AM

Figure 13 represents the obtained values with the AM metrics (Available methods) on each class studied, relatively to the fault frequency (per 1000 bytes) obtained on the same application.

We observe on figure 13 that AM estimation provides a class $C3$ with 45 available methods, which have a fault risk lower than 2 per 1000 bytes, whereas an estimation of 42 available methods is obtained for another class $C4$ presenting a fault risk close to 15. Another significant example is the class $C5$ owning only 31 available methods, whith a fault risk upper than 13.

These examples and the rhombus scattering obtained show that the AM metrics is not a good predictor of the software fault risk.

4.5 Fault risk vs. DIT

In this section, we estimate of the correlation existing (or not) between the fault presence in classes and the inheritance depth of these classes. The purpose is to establish if the metrics DIT (Depth of Inheritance Tree) is a good predictor of fault risk or not. We propose an illustration of the results obtained on figure 14.

We observe that the group of rhombus (except someones) are around an upper line $(0,1)$, $(12,3)$. It means that globally, the DIT metrics provides a useful criterion to assess the fault risk.

However, we notice that two classes, $C6$ and $C7$, having a DIT value equal to 3, present very different fault risk. $C6$ risk fault is lower than 2 whereas $C7$ risk fault is close to 15. A same observation can be

made on classes *C8* and *C9*.

Consequently, our experimentation leads to conclude that the inheritance depth of a class is an interesting program criterion for the fault risk estimation, but it is not sufficient to predict efficiently the fault presence in classes of an object-oriented program.

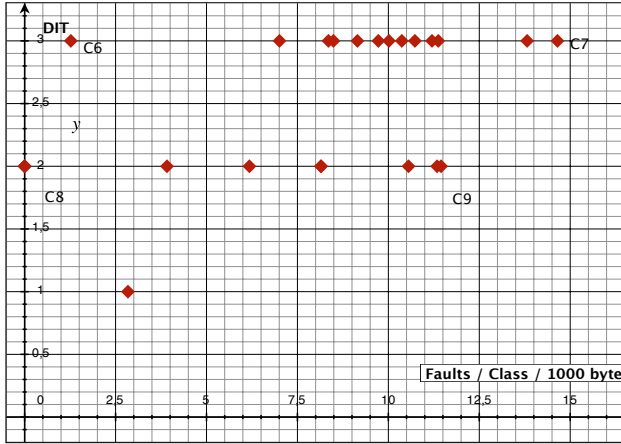


Figure 14. DIT vs. Fault risk / Class / 1000 bytes

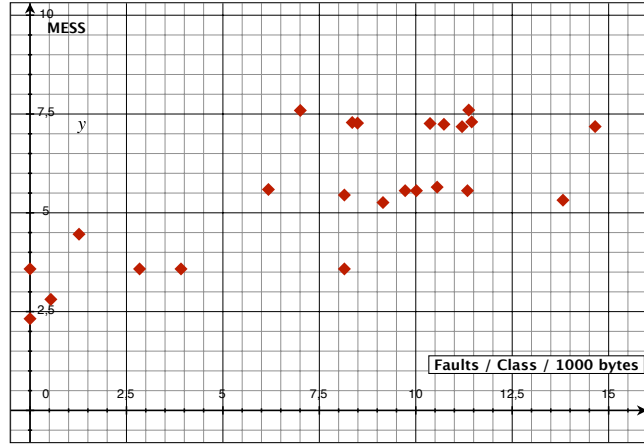


Figure 15. MESS vs. Fault risk / Class / 1000 bytes

4.6 Fault risk vs. *MESS*

Finally, we propose in this sub-section to evaluate our metrics, *MESS*, in order to establish its ability to predict the fault presence. We compare the obtained *MESS* values of the analysed classes with the associated fault risks. The results are provided on figure 15.

We observe on this figure that the rhombus representing the classes are close to the axis (0,3), (15, 8). We notice that globally, the number of potential faults in the classes increases with the *MESS* assessments. This first observation is schematized by the line on figure 16.

Moreover, we distinguish two groups of rhombus :

- a first rhombus group representing classes with a weak fault risk (weaker than 5 per 1000 bytes), and with *MESS* estimation lower than 5,
- a second rhombus group representing classes with a high fault risk (between 6 and 15 faults per 1000 bytes), and with *MESS* estimation upper than 5 (except a class).

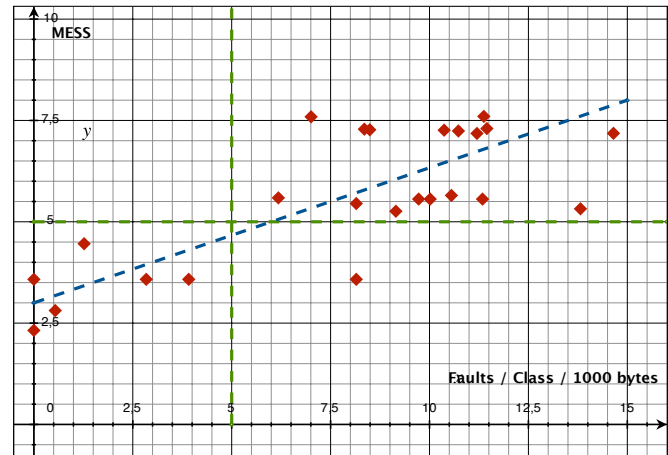


Figure 16. MESS vs. Fault risk / Class / 1000 bytes

This observation suggests the presence of a significant threshold on the estimation of a designer understanding of the program elements. This threshold ($MESS = 5$) should not be exceeded in order to prevent the fault presence. This threshold separating the two groups of rhombus is materialized by two lines on figure 16.

To sum up, this second experimentation provides an empirical validation of the *MESS* metrics aptness to predict fault risk in an object-oriented program.

Moreover, the obtained results corroborate an intuition: the structuration of object-oriented programs affects the designer understanding of the contained information. This understanding directly affects the probability that the designer introduces new design faults in the program.

Thanks to this experimentation, we observe that the metrics *MESS* is a better predictor of fault presence than the other studied metrics : NLM, AM and DIT.

5 Conclusion and future works

The *MESS* metrics application emphasizes the classes whose available elements (methods, attributes) are the most misappreciated. The experimentation puts in evidence that the *MESS* metrics is a good predictor of fault risk.

This metrics could be used from the preliminary design stage (on UML diagrams for example) to choose a structure reducing the fault introduction risks. So it can be used to support a fault prevention approach.

The *MESS* metrics represents an assessment of the software complexity regarding 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 complete 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, they must not be considered for the risk estimation. Therefore, we will define a *MESS*-based metrics, taking into account the frequency of the use of elements (risk-taking estimation). This type of estimation would justify structures considered as dangerous by the *MESS* metrics whose actual risk is reduced by an intense use of understandable elements and a weaker use of the misappreciated ones.

References

- [1] D. Port and M. McArthur, "A study of productivity and efficiency for object-oriented methods and languages," in *APSEC '99: Proceedings of the Sixth Asia Pacific Software Engineering Conference*.

Washington, DC, USA: IEEE Computer Society, 1999, p. 128, ISBN 0-7695-0509-0.

- [2] 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*. IEEE, November 2001, pp. 84–95.
- [3] D.-B. . ED-12B, "Software considerations in airborne systems and equipment certification," RTCA. Inc., EUROCAE, December 1992.
- [4] OOTiA, "Handbook for Object-Oriented Technology in Aviation," Octobre 2004, <http://shemesh.larc.nasa.gov/foot/>.
- [5] 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.
- [6] G. Motet and S. Gaudan, "Assessment of the risks for using object-oriented technologies in critical software," invited communication at the 6th Workshop on Critical Software Tokyo, JAXA publisher, Japan, 2006.
- [7] B. Meyer, *Object-oriented software construction*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1977, iSSN 0-13-629155-4.
- [8] V. S. Alagar, Q. Li, and O. S. Ormandjieva, "Assessment of maintainability in object-oriented software," in *proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS'01)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 194–206.
- [9] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, July and October 1948.
- [10] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in *proceedings of the conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*. New York, NY, USA: ACM Press, 1991, pp. 197–211, ISBN 0-201-55417-8.
- [11] 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, ISBN 0-8186-3740-4.
- [12] W. Li, "Another metric suite for object-oriented programming," *Journal of Systems and Software*, vol. 44, no. 2, pp. 155–162, 1998, iSSN 0164-1212, Elsevier Science Inc., New York, NY, USA.

- [13] McCabe, "A complexity measure," *IEEE Trans. on Software Engineering*, vol. 2, pp. 308–320, 1976.
- [14] L. C. Briand, J. Wüst, S. V. Ikonovski, 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. [Online]. Available: citeseer.ist.psu.edu/briand98investigating.html
- [15] 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, ISSN 0098-5589, Piscataway, NJ, USA, IEEE Press.
- [16] C. B. Archer, "Measuring Object-Oriented Software Products," *Software Engineering Institute*, June 1995, cMU/SEI-CM-28, nlin/0307015, Carnegie Mellon University, Pittsburgh PA.
- [17] S. Benlarbi, K. E. Emam, N. Goel, and S. Rai, "Thresholds for object-oriented measures," in *proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 24–38.
- [18] C. Wohlin, "Revisiting measurement of software complexity," in *proceedings of the Third Asia-Pacific Software Engineering Conference (APSEC'96)*. Washington, DC, USA: IEEE Computer Society, 1996, p. 35, ISBN 0-8186-7638-8.
- [19] B. Unger, L. Prechelt, and M. Philippsen, "The impact of inheritance depth on maintenance tasks: detailed description and evaluation of two experiment replications." Institut für Programmstrukturen und Datenorganisation, Karlsruhe University, Tech. Rep. iratr-1998-18, 1998. [Online]. Available: citeseer.ist.psu.edu/unger98impact.html
- [20] M. H. Halstead, *Elements of Software Science*. New York, NY, USA: Elsevier Science Inc., 1977.
- [21] E. Berlinger, "An information theory based complexity measure," in *proceedings of the 1980 National Computer Conference*. AFIPS Press: Reston VA, 1980, pp. 773–779.
- [22] W. Harrison, "An entropy-based measure of software complexity," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 1025–1029, 1992.
- [23] 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, 1040-550X, John Wiley & Sons, Inc., New York, NY, USA.
- [24] C. Cook, "Information theory metric for assembly language," *Software Engineering Strategies*, pp. 52–60, March/April 1993. [Online]. Available: citeseer.ist.psu.edu/cook93information.html

- [25] 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.
- [26] D. Kang, B. Xu, J. Lu, and W. C. Chu, "A complexity measure for ontology based on uml," in *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.
- [27] S. K. Abd-El-Hafiz, "Entropies as Measures of Software Information," in *proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 110–117, ISBN 0-7695-1189-9.
- [28] D. E. Knuth, *The Art of Computer Programming*. Boston, MA, USA: Addison-Wesley longman Publishing Co., Inc., 1998, vol. 3.
- [29] 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, ISSN 0098-5589, Piscataway, NJ, USA.
- [30] 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, ISSN 0167-9236, Amsterdam, The Netherlands, the Netherlands, Elsevier Science Publishers B. V.
- [31] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *PASTE, Manuvir Das and Dan Grossman*. ACM, 25 June 2007, pp. 1–8, ISBN 978-1-59593-595-3.