Metrics About Fault-Proneness In Object-Oriented Systems

Sabrina Böhm
Universität Ulm
Ulm, Germany
sabrina.boehm@uni-ulm.de

***Abstract*—Software quality is becoming increasingly important in modern times. Faulty or insufficient software can have severe consequences. For this reason, aspects such as qualitiy, security, reliability and maintainability must be considered at an early stage of the development process. Early detection of bugs or problems in the software prevents enormously high costs at the later times. In safety-critical areas, for example, the property of fault-proneness must be carefully considered. In order to include this aspect early in the development process, there are a number of metrics and methods that can estimate the fault prediction or reduce the fault-proneness of the software in an object-oriented context. By following certain rules and procedures, high costs and time can be saved for both developers and consumers. Some of these metrics are presented in this paper to support the software development process. In addition to an interaction based metric, a bayesian network is also analyzed and explained in more detail. The goal of this work is to illustrate some of the widely used methods and design metrics that consider fault-proneness in object-oriented systems.**

I. Introduction

In the field of software development there are some keywords like security, consistency or reliability that are indispensable today. Everyone wants the best and most intelligent software, but with increasing complexity the possibility of fault-proneness in the software increases. In the following, the aforementioned topic is examined under the programming language model of object orientation and metrics that can be considered, which is an important topic in research trends in software technology nowadays. One might think that testing the software and the resulting errors is one of the last steps in the software development process, but this is a false assumption. The earlier the system is examined and tested for critical points, the more work will be saved in later, more cost-intensive development steps.

A software fault is defined as an anomalous condition or defect at the component, device, or subsystem level that can lead to a failure. Therefore an undesirable companion of developing, where the objective is to appear as little as possible. Fault-proneness is an important external software quality attribute of interest to software developers and practitioners. The fault-proneness of an object-oriented class indicates the extent to which the class, given the metrics for that class, is fault-prone. Since it is difficult to measure the fault-proneness of software that is not yet in use, predictive models are applied to estimate the fault-proneness of software classes.

Several studies like "*Fault-Proneness of Open Source Software: Exploring its Relations to Internal Software Quality and Maintenance Process*" [1] or "*Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study*" [2] have been carried out to determine which metrics are useful in capturing important quality attributes such as fault-proneness and fault prediction, which are summarized in the following sections. Furthermore the main research objective is the consolidation of some metrics and methods related to fault-proneness in the software development process.

This paper is organized as follows: Section II deals with the content background of design metrics in object-oriented systems and the basic technologies about object orientation. In Section III the research methodology is described. Afterwards the Section IV presents some studies and further literature that handle the concept of fault prediction and further analysis in that topic area. After the related work, it comes to the Section V, that contains the analysis and summary of obejct-oriented design metrics and the results of some empirical studies that are concerned with fault-proneness and that used to support the software development process. After the investigation of the effects on these metrics, a discussion follows in Section VI, including the classification of the relevance in the software development process and the parties involved.Furthermore there are limitations and benefits of the considered metrics. Finally, the paper is concluded and gives an outlook in Section VII.

II. Content Background

In the following, we will consider fault-proneness as already mentioned, but from a restricted point of view, in an object-oriented context. Many software systems in use are based on object-oriented design. This means that data and program code are encapsulated in reusable objects. Everything is based on the communication of objects. For this purpose, classes, interfaces and methods, as well as attributes are declared and thus serve to represent states. This structure alone protects against fault-proneness, since the code is reusable and thus the programming effort is reduced [3]. Therefore, object orientation in itself offers advantages for maintainability and reusability [4]. Thus, fewer errors occur and the fault-proneness is reduced, too.

In the object oriented context there are a few constructs like class, coupling, cohesion, inheritance, information hiding and polymorphism, which also influence the fault-proneness. Examples of languages that program object-oriented are C#, C++ or Java. One of the most common aspects incorporated into metrics is that of coupling, which refers to the degree of direct knowledge that one element has of another. For example subclass coupling describes the relationship between a child and its parent. The child is connected to its parent, but the parent is not connected to the child. The degree of interconnection of the whole system is a key element in software development, i.e., it is important how big the effect of changing one attribute of one class has on all others and especially how many. Therefore coupling plays a central role in the effects of software faults. It is defined as the degree of interdependence or the strength of relationship between software modules. To give a short introduction in object orientation and the relation between its properties, it is shown in figure 1, that the general aspects as class, method and attribute depend on each other. Invoking methods and accessing attributes is the basic principle of communication of object-oriented software systems, which means that faults can occur here depending on the frequency of use of the methods or attributes.

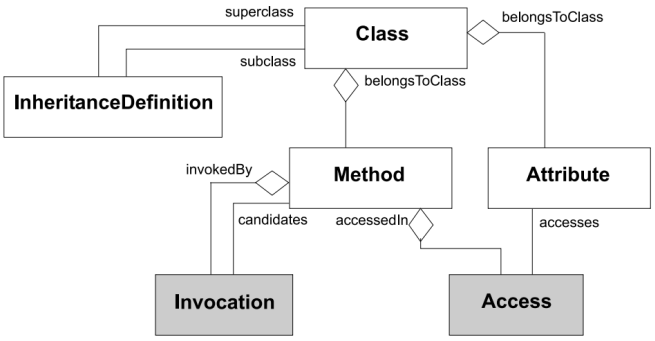

Fig. 1. A model for programming concepts with classes [4].

In how far the interaction of the individual components are connected with the fault-proneness metrics, that is described in the summary of metrics Section V more near.

But why should a programmer actually pay attention to fault-proneness at all? The accurate prediction of where bugs are more likely to occur in the code can help manage efforts in testing, reduce costs, and improve the quality of the software. For different programming paradigms and programming constructs different rules apply which must be considered in relation to fault-proneness. The restriction on object orientation is to facilitate the understanding. In addition many principles are contained, which are taken up in other programming concepts again. Thus some conclusions which are drawn here are also differently realizable and applicable to other software concepts.

In terms of fault-proneness, other quality characteristics that must not be forgotten also play a role like reliability, correctness, completeness, maintainability, as some are dependent on each other in terms of time. For example, software may not be reliable or correct if faults occur frequently that cause the software to become unusable.

When software errors are made, it is often not only tedious to find the programming error, but also expensive. In large systems that are used by many people every day, a small error can cost millions. A common misconception is that fault-proneness should only be considered at the end of the software development process. Especially at the beginning of the development you should build the software architecture in a way that it is less fault-prone. Changes in the architecture are always more expensive later. In addition, even before the programming itself begins, attention should be paid in the planning to various concepts and metrics, which are shown below.

III. Research Methodology

First, this paper deals with research on the keywords ”fault-proneness”, ”fault-proneness in object-oriented systems” and ”fault prediction metrics” in google scholar. To get more literature on this topic, the keywords linked in the papers were used as further search. The linked book ”Empirical Software Engineering” by Springer Verlag appeared frequently with articles like for example ”*Fault prediction modeling for software quality estimation: Comparing commonly used techniques*” [5].

Among them keywords there are ”object-orientation”, ”object-oriented design metrics” and ”faulty classes” to get the content background of object orientation. As first an overview was created in such a way, in order to be able to classify the term ”fault-proneness” into the software development process. The top listed papers that were suggested in google scholar were either sorted out or read more closely by reading the abstract. If the abstract sounded interesting for the topic, then the introduction was read and the conclusion. In some papers, such as the research on metrics, the exact procedure and the analysis section were also read in detail. Two of the metrics were then chosen as a showcase model to illustrate fault-proneness in software systems and how to avoid faults. Further literature was researched for the content background to summarize the basic knowledge about object orientation and a simple understanding of objects, classes and methods.

At the end of research, some conclusions of papers of metrics were compared to summarize the equalities and differences, as well as the benefits and limitations of the object-oriented design metrics just considered.

IV. Related Work

To take a close look at fault-proneness in object oriented systems, basic object orientation knowledge is important, as explained in section II. The paper ”*Beyond Language Independent Object-Oriented Metrics: Model Independent Metrics*” [4] deals not only with the concept of object orientation but also beyond languages and paradigms. The aspects of object orientation such as class, method and attribute can be

found as shown in the following tables, which are subdivided into individual metrics. The model from figure 1 serves as a template to understand the relationships between the individual metrics.

In the following tables I,II and III the abbrevations and the associated metrics are listed. The used model and the metrics of the tables allow a multiple extension into different research directions. Thus, they fit the principle of object orientation and serve as a basic template to get into the basic structure of metrics. The advantages of this approach are the increased flexibility, i.e., new metamodels can be introduced from any context (for example, the financial world or databases), which provides a standard metric without the need to implement new metrics every time a new context is introduced [4]. Now, if you look at this system completely from an object orientation perspective, you can see that the basic programming concepts are translated into metrics. Each new method or variable crates more space to generate faults. The complexity of a class is depends, i.e., on the number of methods (NOM) or the number of attributes (NOA), that can be calculated with $NOA = NIV + NCV$, as shown in table I. An important metric about the methods of a class is the number of input parameters (NOP) or the number of access on attributes (NMAA), as you can see in figure II. For the attribute metrics, the number of direct hits is of great importance, as can be found in figure III.

Many metrics used in various studies are reflected in the tables. The limitations of this approach are that not all object-oriented software metrics can be defined in terms of the language independent model, but these metrics serve as a basic overview. Certain metrics tend to be very specialized and are therefore difficult to define in a generic way. Another limitation of this basic concept is that for some metrics, there is it is not yet known how best to define them in a generic way, so the meta model does not include coupling metrics and cohesion metrics.

In another paper, a systematic literature review was reviewed that looked at 106 papers published between 1991 and 2011. Object-oriented metrics (49%) were used almost twice as often as traditional source code metrics (27%) or process metrics (24%). Object-oriented and process-oriented metrics were reported to be more successful in finding bugs compared to traditional size and complexity metrics [6]. The results of the literature review show that an inheritance and an export coupling metric are strongly associated with fault-proneness. Some evidence also suggests that there may be a small number of metrics that are strongly associated with fault-proneness, and that good predictive accuracy and quality estimation accuracy can be achieved with them.

Today's evidence suggests that most faults in software applications are found in a small percentage of software components [7]. This means that if these faulty software components can be identified early in the lifecycle of the development project, mitigation measures can be taken, such as redesign or refactoring. For object-oriented applications, predictive models using design metrics can be used to identify

TABLE I
CLASS METRICS FROM THE META MODEL.

Abbrevation	Description
HNL	Number of classes in superclass chain of class
NAM	Number of abstract methods
NCV	Number of class variables
NIA	Number of inherited attributes
NIV	Number of instance variables
NME	Number of methods extended, i.e., redefined in subclass by invoking the same method on a superclass
NMI	Number of methods inherited, i.e., defined in superclass and inherited unmodified by subclass
NMO	Number of methods overridden, i.e., redefined compared to superclass
NOA	Number of attributes ($NOA = NIV + NCV$)
NOC	Number of immediate subclasses of a class
NOM	Number of methods
PriA	Number of private attributes (equivalent for protected and public attributes)
PriM	Number of private methods (equivalent for protected and public attributes)
SLOC	Sum of all lines of codes over all methods
WMSG	Sum of message sends in a class
WNMAA	Number of all accesses on attributes
WNOC	Number of all descendant classes
WNOS	Sum of statements in all method bodies of class
WNI	Number of invocations of all methods

TABLE II
METHOD METRICS FROM THE META MODEL.

Abbrevation	Description
LOC	Method lines of code
NMA	Number of methods added, i.e., defined in subclass and not in superclass
MSG	Number of method messages send
NOP	Number of input parameters
NI	Number of invocations of other methods within method body
NMAA	Number of access on attributes
NOS	Number of statements in method body

TABLE III
ATTRIBUTE METRICS FROM THE META MODEL.

Abbrevation	Description
AHNL	Class HNL in which attribute is defined
NAA	Number of times directly accessed

faulty classes early on [7]. The next step is to look at related work based on the metrics just presented and expansions of those.

In the "*Fault-Proneness of Open Source Software: Exploring its Relations to Internal Software Quality and Maintenance Process*" [1] study, it was investigated how the fault-proneness of open source software (OSS) can be explained in terms of internal quality attributes and metrics of the maintenance process. A total of 342 releases of these systems were studied and, as usual, software quality was considered as a set of internal and external quality attributes. A total of 76 internal quality attributes were measured and 23 maintenance process metrics were included in this study. The strengths of this study is the comparison of a few software technology trends like fault-proneness and maintainability, as well as the study itself. First the study considers a wide range of metrics than common studies. Furthermore more OOS systems were involved in order to get a better indication of the results. In addition they focused on the fault-proneness of modern Java-based systems and investigated them as an aggregated sample. The framework for assessing the maintenance process was adopted from their previous studies. The results of the factor analysis performed showed that the metrics studied can be interpreted in terms of two factors, one of which is the system size, as shown in Figure 2. Previous studies in this area are based only on relatively small sets of OSS systems and releases, despite the fact that OSS projects are very diverse and heterogeneous [1]. Conclusively, the results may not be generalizable due to their relatively limited nature. Here, larger systems were chosen and size was found to play an important aspect in fault-proneness. In many other studies only small software systems were considered and it is noticeable that many in their conclusion note that a limitation of their work is that the statements can only be applied to small sized systems [1], [6].

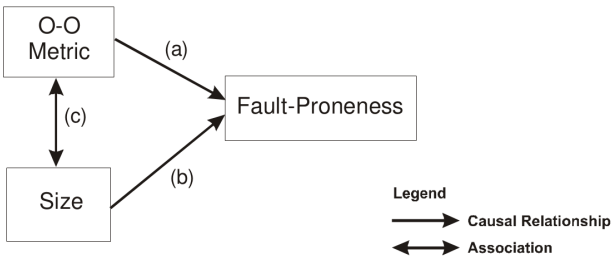

Fig. 2. Path diagram illustrating the confounding effect of class size on the relationship between an object-oriented metric and fault-proneness [7].

Another study, also dealt with object-oriented design metrics for Java applications and construct a prediction model. It became clear that an export coupling metric exhibited the strongest association with fault-proneness, indicating a structural feature that may be symptomatic of a class with a high probability of latent faults [7].

A theoretical basis for developing quantitative models that relate object-oriented metrics and external quality metrics is summarized in Figure 3. As already mentioned, one of the strongest association with fault-proneness was an export coupling metric in a study. This illustrates that there is a presumed relationship between object-oriented metrics and fault-proneness due to the effect on cognitive complexity [7]. Cognitive complexity can be defined as the mental load of the individuals who have to interact with the component, for example, the developers, testers, inspectors, and maintainers.

Fig. 3. Theoretical basis for the development of object oriented product metrics [7].

Some studies also suggest that the depth of inheritance has an impact on the comprehensibility of object-oriented applications, and therefore would be expected to have a detrimental effect on fault-proneness [7]. It was also found that inheritance leads to distributed class descriptions. That is, in this case, the complete description for a class can only be assembled by examining both the class itself and each of the superclasses of it. Since different classes are described in different places in the code of a software program, also often distributed over several files, a programmer must turn to several places, in order to receive a complete description of a class. While this argument applies to software source code, it is not difficult to generalize it to design documents. Faults are therefore very distributed and usually cannot be found and fixed in a single place without making other changes.

Another study has considered a used set of ten software product metrics related to the following software attributes: the size of the software, coupling, cohesion, inheritance, and reuse [8]. Some hypotheses regarding fault-proneness were empirically tested in a case study examining the client side of a large network service management system. The system under consideration is written in Java and consists of 123 classes. Validation was performed using two data analysis techniques: regression analysis and discriminant analysis.

Among other things, Class cohesion is a key attribute used to assess the design quality of a class and refers to the extent to which the methods and attributes of a class are related. Typically, classes contain special types of methods, such as constructors, destructors, and access methods, where each of these special methods has its own properties that can affect the measurement of class cohesion [9]. Here we see again that the metrics can all be derived from the tables I and II below.

Another paper empirically examines this impact of methods on cohesion measures. Twenty existing class cohesion metrics were used and Two types of special methods were considered, constructors and access methods. The empirical study applies the metrics, to five open source systems under four different scenarios, including first considering all special methods, second ignoring only constructors, third ignoring only access methods, and fourth ignoring all special methods. The results of the empirical investigations show that the cohesion values for most of the considered metrics differ significantly in the

TABLE IV
THE CK METRICS.

Abbrevation	Definition	Sources
CBO	Coupling between objects for a class is a count of the number of non-inheritance related couples with other classes	[10]–[12]
LCOM	Lack of Cohesion in Methods counts number of null pairs of methods that do not have common attributes	[11], [12]
NOC	The number of children is the number of sub-classes of a child class in a hierarchy	[11], [12]
DOI	The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes	[11], [12]
WMC	The weighted methods per class is a count of sum of complexities of all methods in a class	[11], [12]
RFC	The response set of a class is defined as set of methods that can be potentially executed in response to a message received by an object of that class	[11], [12]

four scenarios, but do not significantly affect the abilities of the metrics to predict faulty classes [9].

Based on this research, two more specific metrics have now been picked out, that try to support the software development process with their metrics. All of them work in their own way and in certain states of development. In order to provide guidance on how to proceed in the software process, some special metrics will be explained in detail now.

V. SUMMARY OF METRICS

Some fundamental metrics are based simply on counting the number of interactions or the lines of code of a class as shown in Section IV. Next, we take a closer look at the most famous object-oriented metrics, the "CK-metrics". It is a set of metrics proposed by Chidamber and Kemerer in 1991 specifically for object-oriented software [12]. Build on the basic metrics table of the Chapter IV, divided in class metrics, method metrics and attribute metrics, more detailed aspects are now examined.

Many metrics are based on comparable ideas and provide redundant information. By using a subset of metrics, prediction models can be built to identify the classes that are in fault. In the table IV, on can see the CK-metrics summarized, which are named in many other papers too.

Next, some of the CK metrics will be presented in more detail and with some examples.

*1) **CBO**:* : The coupling between objects for a class is a count of the number of non-inheritance related couples with other classes. Two things are coupled if and only at least one of them atcs upon the other. Any evidence of a method of one object using methods or instance variables of another object constitutes coupling. Given is an example for CBO:

```
import java.util.Calendar;

public class AdultIssuePolicy implements
                                    IssuePolicy {
   public Calendar compute(BiblioType type,
    Calendar from) {
      Calendar res = (Calendar)from.clone();
        res.add(Calendar.DATE, 14);
        return res;
   }
}
```

Note, that coupling is not associative, i.e., if A is coupled to B and B is coupled to C, this does not imply that C is coupled to A.

*2) **LCOM**:* : The Lack of Cohesion in Methods counts number of null pairs of methods that do not have common attributes. Consider a Class A with methods m_1, m_2,...m_n. Let $\{I_i\}$ = set of instance variables used by the method m_i. There are n such sets I_1,...,I_n. The LCOM is therefore the number of disjoint sets formed by the intersections of the n sets. Formally let´s say a Class C with

- k fields $f_1, f_2, ..., f_k$ and
- n public methods $m_1, m_2, m_3, ..., m_n$.

So $I_i = \{f_l : f_l \text{ is used by } m_i\}$ and $N = \frac{n(n-1)}{2}$ is the number of different possible pairs of methods. It follows that

$$P = |\{(m_i, m_j) : i < j \text{ and } I_i \cap I_j = \emptyset\}$$
$$Q = |\{(m_i, m_j) : i < j \text{ and } I_i \cap I_j \neq \emptyset\}$$

and $N = P + Q$. The LCOM is P. Given is an example for LCOM:

```
public class A {
  private int f1;
  private int f2;
  private int f3;
  private int f4;
  public void method1() { I1 = { f1, f2 }
     // uses f1
     // uses f2
  }
  public void method2() { I2 = { f2, f3 }
     // uses f2
     // uses f3
  }
  public void method3() { I3 = { f3, f4 }
     // uses f3
     // uses f4
  }
}
```

When looking at the (m_i, m_j) pairs, the $I_i \cap I_j$ of $(method1, method2)$ is $_f2$, $(method1, method3)$ is $\emptyset$ and $(method2, method3)$ is $_f3$. Therefore LCOM is 1.

*3) **NOC**:* The number of immediate subclasses subordinated to a class in the class hierarchy, which is pretty much self explanatory.

*4) **DOI**:* The Depth of inheritance of the class is the depth of inheritance tree metric for the class, it is the length of the longest path to the root. The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex, but it is useful to have a measure of how deep a particular class is in hierarchy so that the class can be designed with reuse of inherited methods.

*5) **WMC**:* The Weighted Methods Per Class is a count of sum of complexities of all methods in a class. Consider a Class C with methods $m_1, m_2, ..., m_n$. Let $c_1, .., c_n$ be the static complexity if the methods. Then

$$WMC = \sum_{i=1}^{n} c_1.$$

*6) **RFC**:* the Response For a Class is defined as set of methods that can be potentially executed in response to a message received by an object of that class. The RFC $= |RS|$ where RS is the response set for the class. The response set if an object $\equiv$ {set of all methods that can be invoked in response to a message to the object}. Given is an example for RFC:

```
public class A {
   private B aB;
   public void methA1() {
      return aB.methB1();
   }
   public void methA2(C aC) {
      return aC.methC1();
   }
}
```

It follows $RS = methA1, methA2, methB1, methC1$.

These metrics are the fundamental rules, that developers follow. People who investigate new studies on metrics in objectorineted systems usually have these rules as a cornerstone. Some issues that comes with this metrics, are what happen with classes with no fields, static member or self calls. Not all questions, provide the metrics answer, they just provide a framework.

Next step, is the look at a study that deals with class cohesion metrics. After that, a bayesian network is explained, which is a bit more difficult.

A. Method-Method Interaction-Based Cohesion Metrics for Object-Oriented Classes

Basic units of design in object-oriented programs are classes. Class cohesion refers to the relatedness of class members, i.e., their attributes and methods. Multiple metrics for class cohesion have been proposed in the literature. These object-oriented metrics are based on information available during the high-level or low-level design phases. A formula that accurately measures the degree of interaction between each pair of methods is proposed and used as the basis for introducing a low-level design class cohesion (LSCC) metric [13]. Low-level design (LLD) cohesion metrics use more finely resolved information than that used by High-level design (HLD) cohesion metrics. HLD cohesion metrics identify potential cohesion issues early in the HLD phase. In figure 4, rectangles represent methods, circles indicate attributes, and links illustrate the use of attributes by methods of a class. Metrics based on counting the number of links, i.e., the use of attributes by a method, can indicate whether a class is strongly or weakly cohesive. This finely granulated information is important to help software developers refactoring their code and detecting which methods to possibly remove, i.e., the methods that exhibit even no links with other methods. When a method-method interaction (MMI) metric is applied to measure the cohesion for the class shown in figure 4, the connectivity between each pair of methods is calculated, and it is clearly seen that method m_3 is weakly interconnected to other methods in this class.

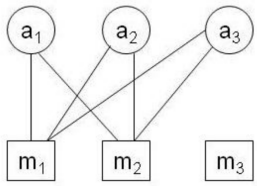

Fig. 4. Sample representative graph for a hypothetical class [10].

As shown in figure 5, there are four other classes with a different method-method interconnection. The class cohesion (CC) of Bonja and Kidanmariam is the ratio of the summation of the similarities between all pairs of methods to the total number of pairs of methods [14]. The similarity between methods i and j is defined as

$$sim(i,j) = \frac{|I_i \cap I_j|}{|I_i \cup I_j|},$$

where I_i and I_j are the sets of attributes linked by methods i and j, respectivly [10]. In contrast with the class cohesion metric of Pena (SCOM), the calculation is defined as follows

$$sim(i,j) = \frac{|I_i \cap I_j|}{min(|I_i|, |I_j|)} \cdot \frac{|I_i \cup I_j|}{n},$$

where n is the number of attributes [15]. Both CC and SCOM neither consider transitive MMI nor account for inheritance or different method types, and they have not been empirically validated against external quality attributes such as fault occurrences. Of course, there are many other metrics of this kind but the results for the metrics that consider the degree of interaction between each pair of methods are very close to each other [13].

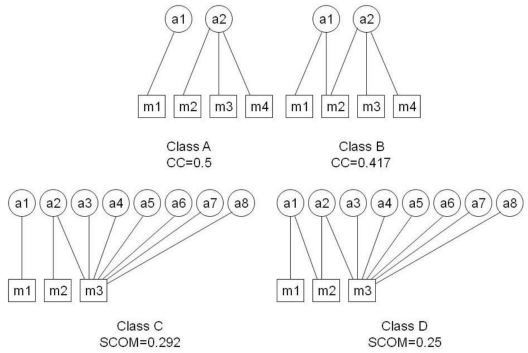

Fig. 5. Classes with different method-method connectivity patterns [10].

The results suggest that class quality, as measured in terms of fault occurrences, can be more accurately explained by

cohesion metrics that account for the degree of interaction between each pair of methods. The fault prediction of interconnection-based object-oriented class cohesion metrics should help the developer to support refactoring during the LLD phase, therefore in an early state of development [13].

B. A Bayesian Network

Some metrics are based on the concept of the possibility that the methods and attributes of a class of the code can be presented as a graph. Here the interaction can be graphically represented by an undirected edge that links a circular node (attribute) to a rectangular node (method).

The next method to take a closer look at the fault-proneness is a concise representation of a joint probability distribution on a set of statistical variables. Bayesian methods can be used for assessing software fault content and fault proneness. A bayesian network (BN) is encoded as an acyclic graph of nodes and directed edges [16]. Assuming that the relationship can be modeled with a general linear model, the structural and numerical specification for the BN is derived. The model can be thought of as a generalization of existing techniques for assessing software quality. The model consists roughly of two parts, first the method produces a probability distribution of the estimated fault content per class in the system and second the conditional probability that a class contains a fault. The structure of the model is a BN model whose underlying representation is the generalized linear model. The definition probabilistic network (acyclic graph $G = (V, E)$; A set S, of (prior) conditional probability distributions). Consider a finite set of random variables $X = \{X_1, X_2, ..., X_n\}$. It can be defined that a probabilistic network $N = (G, X) over X$ consists of

- a directed acyclic graph $G = (V, E)$, V is the set of nodes in the graph and there is a one-one correspondence between V and X. $E \subseteq V \times V$ the set of directed edges, representing conditional independence assumptions, i.e., for each $X_i \in X$, $i(X_i, ND_{x_i}|Pa_{x_i})$ and $ND_{x_i} = X \backslash (X_i \cup Des_{x_i})$
- a set of (prior) conditional probability distributions, that specifies $p(X_i)|p(Pa_{x_i})$ for each $X_i \in X$, where Pa_{x_i} represents the set of immediate parents of X.

Once a network is specified over a set of random variables, their marginal and joint probabilities can be computed. Given a BN structure, the joint probability distribution over X is encoded as

$$p(X) = \prod_{i=1}^{n} p(X_i|P_{ax_i})$$

And given this joint probability, the marginal probability of a random variable X_i is computed as

$$p(X_i) = \sum_{X_i, j \neq i}^{n} p(X)$$

The CK metrics mentioned in the previous section can be found here, too. The model parameters that are used are listed in the following [16]:

1. Weighted methods per class (WMC)
2. Depth of inheritance tree (DOI)
3. Response for class (RFC)
4. Number of children (NOC)
5. Coupling between object classes (CBO)
6. Lack of cohesion in methods (LCOM)
7. Source lines of code (SLOC): This is measured as the total lines of source code in the class and serves as a measure of class size.

As can be seen the parameters 1.-6. are the CK metrics IV. The 7. point is already presented in table I, too. The dependent variables, which serve as surrogate metrics of software quality here, are

- Fault Content (**FC**): We define fault content as the numbe of faults per class. The estimation of our model is a (marginal) conditional probability of observing a certain number of faults per class, given the metrics for that class.
- Fault proneness (**FP**): The conditional probability that a class contains a fault, given the metrics for that class.

In this approach, it is assumed that a general linear model (GLM) relates the dependent and independent random variables and construct a BN structure that represents the GLM. This assumption is motivated by several factors: The GLM is versatile enough to represent existing linear relationships or, through transformations, a variety of nonlinear relationships. Linear models also provide a relatively simple and parameterized way to capture the dependencies between domain variables.

Consider that a responsive variable Y varies as some function of a set $X = \{X_1, X_", ..., X_k\}$ of independent predictor variables. In the GLM, it follows

$$E(X) = \mu = g^{-1}(X\beta),$$

where Y is the set of observations with expected value $E(X) = \mu$. The linear predictor with coefficients β is $X\beta$ and g is the link function determined by the distribution of Y as graphically shown in figure 6.

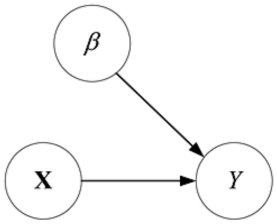

Fig. 6. BN representation of a general linear model.

Figure 7 shows how the individual components of the model parameters influence the FC and FP. Here the connection between X and Y can be clearly seen.

In table 8, the used data is given. After the set up model has gone through the data, one can see, that the SLOC is

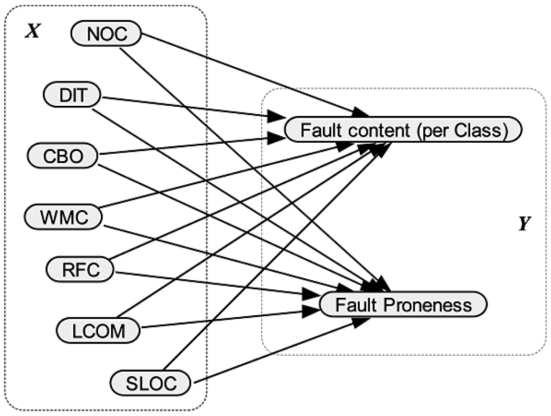

Fig. 7. BN model for fault content and fault-proneness analysis.

about $43K$ and the faulty percentage is nearly 40%, which is enormous. The results also show that this model produces these estimations at a statistically significant level. The results of performing multiple regression, the metrics WMC, CBO, RFC, and SLOC are very significant for assessing both fault content and fault proneness. In general this specific set of predictors is very significant for assessing FC and FP in large software systems.

Item	Value
Number of classes	145
Number of methods	2107
Size	43K SLOC
Total number of (relevant) faults	640
Total % faulty (and fault-prone) classes	39.31
% Design faults	14.375
% Classes (Design faults)	18.62
% Source code faults	85.625
% Classes (Source code faults)	39.31

Fig. 8. Data Description.

VI. Discussion

As Chapter V makes clear, many models and methods are based on the fundamentally same metrics introduced at the beginning. It also became clear that it quickly becomes mathematical and complex when, for example, the BN is examined in more detail. Accordingly, finding fault-proneness classes is not an easy task and requires effort and time from the stakeholders and the developers. In comparison, the individual methods to predict faults have concluded depending on the use case.

When a company or software developer decides to use metrics for their object oriented system to avoid errors, there are many starting points. Some metrics start with an accurate understanding of object orientation, which in itself takes care of fault-proneness. However, just because you think you've found the right metric doesn't mean you won't make any more failures. Another problem, as some papers also note, is that the studies have mostly tested their metrics on small projects and so the validity is not scalable to larger systems. Therefore some aspects such as the size of the software project plays a big role. Some metrics are only designed for smaller systems.

What is certain is that incorporating metrics or certain programming rules is important and can easily be verified via simple metrics to not disregard fault-proneness.

A. Classification of relevance

There are metrics that can or should be used in an early development process and those that are used in later states. Both do not guarantee fault prediction, but it is important to pay attention to fault-proneness at all. The aspect of fault-proneness must also be considered during planning and developing to avoid highly costs.

The most influenced persons are the developers but also the customers. For the developer it is important how to build and construct his object-oriented classes, with a metric as orientation to avoid bugs. For the customers it is of utmost importance that the use of the system runs error free, so the relevance is high.

B. Limitations

A further problem that has emerged is that many studies were initially only tested on small data sets. The results of the studies are therefore only transferable to this type of software and thus the significance is not automatically scalable to larger data sets. For example, when refactoring classes, not only cohesion must be considered, but also other quality attributes, such as coupling. So it's not enough to get into one specific metric, you should include all of them a bit. However, if you don't have a lot of money to spend, corporations don't want to put their money into research on perfect metrics. The research on fault-proneness metrics so far gives a lot of room to develop new methods and approaches.

Sometimes it can also happen that several classes are displayed as faulty and the cause can only be fixed in a completely different class. This class is thus not discoverable by the applied method and the fault must be searched for elsewhere.

Furthermore all errors that are feasible can never be covered. It depends on the way of programming and the communication among the people working on it, i.e. human influences that no metric can cover. In every context, new situations arise and will continue to arise in the future that require new metrics. If project changes occur in any development process states, the fault prediction metrics may also need to be adjusted, which is not done in most developments.

C. Benefits

The widespread assumption is wrong that it is enough to look at the fault context once. A person working on the system should always keep an eye on the metrics to be observed in case of changes in order to avoid a costly awakening later on. Even if you can't be 100% hedged by metrics on fault-proneness, the ability to make fewer mistakes is a good option

for many developers. If fault-proneness is included in early software development steps, a lot of money can be saved. Later changes to the software, for example, if the architecture has to be changed or the database has to be adapted, are enormously expensive and time-consuming.

It may also happen that by considering design metrics, more attention is paid to building the entire system more carefully and cleanly, which additionally improves maintenance and changeability, among other things.

VII. Conclusion

The objective of this work was to illustrate some of the widely used methods and design metrics that consider fault-proneness in object-oriented software systems.

In the future, as well as today, fault-proneness estimation and prediction could play a key role in software product quality control. In this area, much effort has been invested in defining metrics and identifying models for system evaluation, object-oriented models as well as others. Using these metrics to assess which parts of the system are more fault-proneness is of primary importance, as summarized in this paper. Much work has concentrated on how to select the software metrics that are most likely to indicate fault-proneness.

In the field of software evolution, metrics can be used for identifying stable or unstable parts of software systems.

Moreover the question is, what can really prevent faults in the later used system beforehand?

If you take up the other software trends of development mentioned at the beginning, it becomes clear that they are all connected in a certain way. The fault-proneness and the maintenance grow with increasing complexity.Ensuring security and quality always involves a number of features.

In conclusion, the CK metrics such es CBO, LCOM or DOI are always a good start when considering fault-proneness in object-oriented systems.

References

[1] D. Kozlov, J. Koskinen *et al.*, "Fault-proneness of open source software: Exploring its relations to internal software quality and maintenance process," *The Open Software Engineering Journal*, vol. 7, no. 1, 2013.

[2] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study," *Software process: Improvement and practice*, vol. 14, no. 1, pp. 39–62, 2009.

[3] R. G. Fichman and C. F. Kemerer, "Adoption of software engineering process innovations: The case of object orientation," *Sloan management review*, vol. 34, pp. 7–7, 1993.

[4] M. Lanza, S. Ducasse *et al.*, "Beyond language independent object-oriented metrics: Model independent metrics," in *Proceedings of 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering*. Citeseer, 2002.

[5] T. M. Khoshgoftaar and N. Seliya, "Fault prediction modeling for software quality estimation: Comparing commonly used techniques," *Empirical Software Engineering*, vol. 8, no. 3, pp. 255–283, 2003.

[6] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and software technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[7] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of systems and software*, vol. 56, no. 1, pp. 63–75, 2001.

[8] P. Yu, T. Systa, and H. Muller, "Predicting fault-proneness using oo metrics. an industrial case study," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. IEEE, 2002, pp. 99–107.

[9] J. Al Dallal, "The impact of accounting for special methods in the measurement of object-oriented class cohesion on refactoring and fault prediction activities," *Journal of Systems and Software*, vol. 85, no. 5, pp. 1042–1057, 2012.

[10] ——, "Fault prediction and the discriminative powers of connectivity-based object-oriented class cohesion metrics," *Information and Software Technology*, vol. 54, no. 4, pp. 396–416, 2012.

[11] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra, "Investigating effect of design metrics on fault proneness in object-oriented systems." *J. Object Technol.*, vol. 6, no. 10, pp. 127–141, 2007.

[12] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in *Conference proceedings on Object-oriented programming systems, languages, and applications*, 1991, pp. 197–211.

[13] J. Al Dallal and L. C. Briand, "A precise method-method interaction-based cohesion metric for object-oriented classes," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 2, pp. 1–34, 2012.

[14] C. Bonja and E. Kidanmariam, "Metrics for class cohesion and similarity between methods," in *Proceedings of the 44th annual Southeast regional conference*, 2006, pp. 91–95.

[15] L. Fernández and R. Peña, "A sensitive metric of class cohesion," 2006.

[16] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *IEEE Transactions on software Engineering*, vol. 33, no. 10, pp. 675–686, 2007.