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.

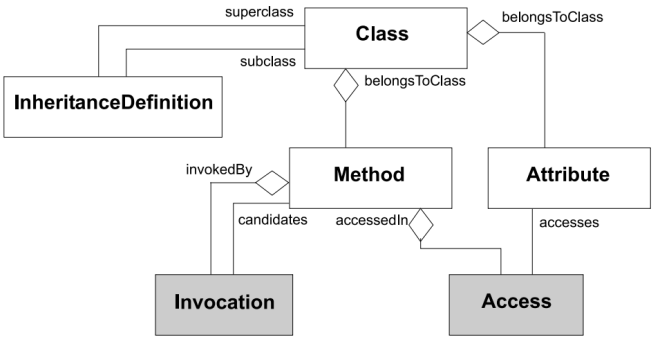

Fig. 1. A model for programming concepts with classes [4].

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 [9]. 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 [6]. 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)
WLOC	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 [6]. 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], [9].

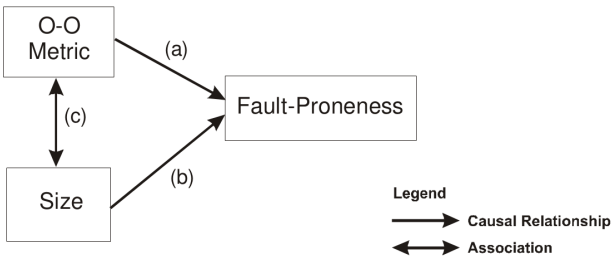

Fig. 2. Path diagram illustrating the confounding effect of class size on the relationship between an object-oriented metric and fault-proneness [6].

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 [6].

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 [6]. 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 [6].

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 [6]. 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 [7]. 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 [8]. 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
METRICS ABOUT FAULT-PRONENESS.

Abbrevation	Definition	Sources
LSCC	low-level design class cohesion metric	[12]
LOC	Lack of Cohesion counts ..	[13]
DOI	Depth of Inheritance ..	[13]
PCCC	path connectivity class cohesion	...

four scenarios, but do not significantly affect the abilities of the metrics to predict faulty classes [8].

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 basic metrics are based simply on counting the number of interactions or the lines of code of a class. Next, we take a closer look at two studies on metrics on fault-proneness. Build on the basic metrics of the Chapter IV, divided in class metrics, method metrics and attribute metrics, more detailed aspects are now examined. This provides a summary of many similar metrics compared to related research. First, we look at a study that deals with class cohesion metrics. hier die tablee m,aybe IV.

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. In this paper, 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 [14]. 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.

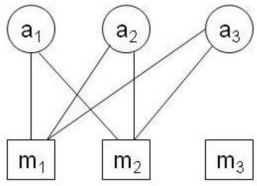

Fig. 4. Sample representative graph for a hypothetical class [12].

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 [15]. 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 [12]. 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 [16]. 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 [14].

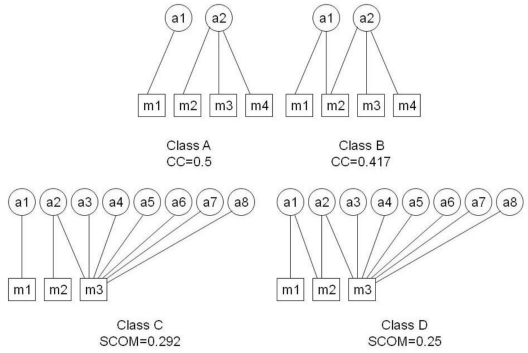

Fig. 5. Classes with different method-method connectivity patterns [12].

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 [14].

B. A Bayesian Network

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 [17]. 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 $N = (G, X)overX$ 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 model parameters that are used are listed in the following [17]:

1. Weighted methods per class (**WMC**): The number of methods implemented in a given class.
2. Depth of inheritance tree (**DOI**)
3. Response for class (**RFC**): Number of methods implemented within a class plus the number of methods accessible to an object class due to inheritance. (Traditionally, it represents the number of methods that an object of a given class can execute in response to a received message [17].)
4. Number of children (**NOC**): The number of classes that directly inherit from a given class.
5. Coupling between object classes (**CBO**): The number of distinct non-inheritance related classes to which a given class is coupled. (i.e., when a given class uses the methods or attributes of the coupled class [17].)
6. Lack of cohesion in methods (**LCOM**): A measure of the degree to which a class represents single or multiple abstractions. There are varying definitions for LCOM. (i.e., by computing the average percentage of methods in a given class using each attribute of that class, and then subtracting that percentage from 100 percent [17].)
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.

The dependent variables, which serve as surrogate metrics of software quality, are

- Fault Content (FC): We define fault content as the number of faults per class. The estimation of ourmodel is a (marginal) conditional probability ofobserving a certain number of faults per class, giventhe metrics for that class.
- Fault proneness (FP): The conditional probability thata class contains a fault, given the metrics for that class.

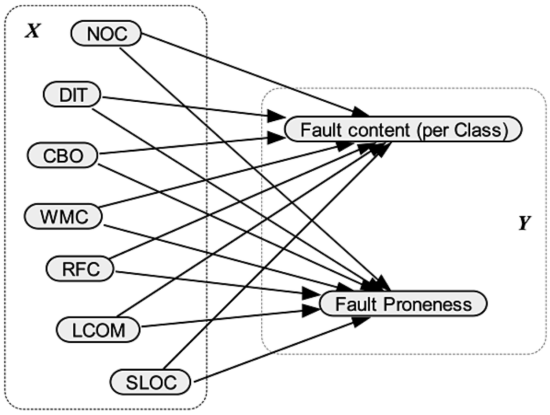

Fig. 6. BN model for fault content and fault-proneness analysis.

The model construction is as follows

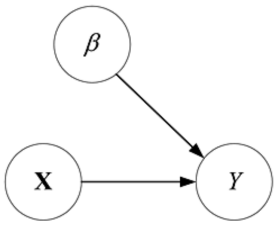

Fig. 7. BN representation of a general linear model.

The results also show that this model producesthese estimations at a statistically significant level.

- RESULTS

- BN: the results of performing multiple regression, the metrics WMC, CBO, RFC, and SLOC are very significant for assessing both fault content and fault proneness

- this specific set of predictors is very significant for assessing fault content and fault proneness in large software systems

- it was observed that import coupling (that count the number of other classes called by a class) metrics are strongly associated with fault pron and predict faulty classes with high accuracy

- Associate refactoring to the last sections

VI. Discussion

- discuss the metrics depending on the benefits/limitations and which metrics scored well in comparison (die genauer betrachteten oder alle)

- discussion with the classification according to relevance in the development process and the relevance for stakeholder

- what you should always pay attention to, no matter how large the project is; what depends on the size of the project; other projectsprops ($\rightarrow$ limitations)

A. Classification of relevance

- in which parts of the development is it important to pay attention to fault proneness

- the aspect of fault-proneness must be considered early on, during planning and development

- in addition always pay attention to all aspects early to avoid costs etc.

- who is most influenced, developers or customers

- for the developer important how to build and construct his object-oriented classes, with a metric as orientation to avoid bugs eg

B. Limitations

- Refactoring classes requires not only accounting for cohesion, but also accounting for other quality attributes, such as coupling (ref. [14])

- all errors that are feasible can never be covered

- many studies have only been tested with small software projects and classes, which is not very meaningful over large ones

C. Benefits

- advantages if fault-proneness is included in early software steps; pay attention to metrics

- note time and money

VII. Conclusion

- summarize discussion and results; name important metrics/properties that should stay in mind after reading this

- what can reduce the fault-pron in software development

- outlook

- This paper discussed the concept of fault proneness using the example of object-oriented programming and design metrics, which means that for the most part the results can only be transferred to the object-oriented context.

- maybe future work

- final catch and the link to the superordinate context (research trends in software technology); link to introduction and motivation

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

In the area of software reengi- neering and reverse engineering [7], metrics are being used for assessing the quality and complexity of software systems, as well as getting a basic understanding and providing clues about sensitive parts of software systems.

wie in related work aufgezeigt die flexible metrics die können als ausblick hilfreich sien wenn sich der kontexxt leicht ändert

future work wäre größere systeme anschauen und testen auf faulty classes und metriken testen an sich

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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[11] L. C. Briand, W. L. Melo, and J. Wust, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE transactions on Software Engineering*, vol. 28, no. 7, pp. 706–720, 2002.

[12] J. Al Dallal, "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.

[13] 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.

[14] 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.

[15] 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.

[16] L. Fernández and R. Peña, "A sensitive metric of class cohesion," 2006.
[17] 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.