Significance of Different Software Metrics in Defect Prediction

Marian Jureczko

Institute of Computer Engineering, Control and Robotics
Wrocław University of Technology
Wybrzeże Wyspiańskiego 27, 50-370, Wrocław - Poland
marian.jureczko@pwr.wroc.pl

Abstract - **This paper presents an empirical analysis of significance of different process and product metrics in defect prediction models. 48 releases of 15 open-source and 38 releases of 7 proprietary projects were investigated. Pearson correlation coefficients with the number of defects were calculated for each of the metrics respectively. Subsequently defect prediction models were built using linear stepwise regression and a discriminant analysis was conducted. Since the stepwise regression was used, the obtained models always consisted of a subset of the investigated metrics. Therefore, it was possible to check whether the selection of metrics corresponds with the correlations with number of defects and the discriminant power of the metrics. Moreover, according to the obtained results some of the metrics were recommended with regard to defect prediction.**

Keywords - **defect prediction, software metrics, empirical study**

1. Introduction

Many organizations are interested in predicting the number or location of defects in their software systems. However, in order to construct an effective defect prediction model a complex measurement program must be launched. Despite of the fact that the goal of defect prediction is desired by many companies, according to Kaner and Bond [12] only few of them actually establish the measurement programs and many of the companies establish the programs only to conform with the criteria laid down in the Capability Maturity Model.

One of the major issues in launching measurement program is the cost. According to Fenton [6] the cost of a software measurement program can be estimated to 4% of the project development budget. Fortunately, the aforementioned cost may be lowered by reducing the number of different software metrics. Nevertheless, it is hard to decide, which metrics should be collected and which could be ignored. Before the measurement program starts, there is usually no clue, which metrics will be good defect indicators in the project under examination. Furthermore, cross-project defect prediction studies [10, 23] showed major issues in defining measurement guidelines, which are applicable to defect prediction in every project.

This study investigates the significance of different software metrics with regard to defect prediction. A correlation analysis was performed in order to investigate the relation between the software metrics and the number of defects. Subsequently results of constructing defect prediction models were analyzed. The constructing procedure utilized stepwise regression. The analysis investigated which metrics were selected to the model and which were filtered out. Finally, discriminant analysis were conducted in order to determine the discriminant power of the metrics in classifying Java classes as defective or defect free. The results of those three analyses were compared and the most promising metrics were selected and recommended with regard to defect prediction.

The rest of this paper is organized as follows. Subsequent sections describe related work. The 3rd section regards methods of data acquisition. The 4th, 5th and 6th sections present correlation analysis, metrics occurrences in defect prediction models and discriminant analysis respectively. Threats to validity are discussed in the 6th section. Conclusions are given in the 7th section.

2. Related Work

Considerable research has been performed on software metrics and defect prediction models. However, only few of the studies considered identifying small number of metrics that would be effective with regard to defect prediction.

Knab, Pinzger and Bernstein [13] conducted an experiment, where different defect prediction models were constructed using decision trees over seven versions of the Mozilla project. Product metrics as well as process metrics were used. Different models used different sets of metrics. And therefore, it was possible to compare the efficiency

of different metrics set. The best model was built using only the metrics, which described the historical defects (NDPV related) and it gave good predictions in 62% of the investigated cases.

Shihab et al. [19] investigated three different versions of the Eclipse project. The goal of the study was to found a small number of software metrics that would be very effective in defect prediction models, namely identify the independent variables that produce the impact and quantify by how much they impact the defects. The authors started with 34 different software metrics. They iteratively applied the stepwise logistic regression and eventually identified a smaller set of metrics: Anonymous Type Declarations (ACD), Number of Static Methods (NSM), Max Number of Parameters (PAR max), Total Prior Changes (TPC), Total Lines of Code (TLOC) and Pre-release Defects (PRE). The authors concluded that using a much smaller set of statistically significant and minimally collinear metrics does not significantly affect the prediction results of the logistic regression model.

Catal et al. [3] examined Chidamber-Kemerer metrics suite and some method-level metrics (the McCabe's and Halstead's ones) for a defect prediction model which is based on Artificial Immune Recognition System (AIRS) algorithm. The authors investigated together 84 metrics from the method-level metrics transformation and 10 metrics from the class-level metrics. The dataset under study was a storage management project for receiving and processing ground data in the NASA. The project had been implemented in C++. According to obtained results the authors concluded that the best fault prediction is achieved when CK metrics are used with the lines of code (LOC) metric.

Olague et al. [18] investigated three metrics suites (CK, MOOD and QMOOD). The metrics suites were validated for their ability in predicting fault-proneness. The authors used defect data from six versions of Mozilla Rhino, an open-source implementation of JavaScript written entirely in Java. The authors obtained the best results for CK metrics. However, they concluded that both CK and QMOOD suites contain metrics, which are effective with regard to detecting error-prone classes.

D'Ambros et al. [5] described and analyzed following approaches to defect prediction (it included the kind of data they require):

- Change Log Approaches – they use information extracted from the versioning system, assuming that recently or frequently changed files are the most probable source of future bugs.
- Single-version approaches – they assume that the current design and behavior of the program influences the presence of future defects. These approaches do not require the history of the system, but analyze its current state in more detail, using a variety of metrics, especially the Chidamber Kemerer metrics suite.

The authors investigated the aforementioned approaches by constructing defect prediction models based on employed on those approaches metrics and compared the prediction efficiency. They concluded that the greater predictive power have those metrics, which describe the size of modification, namely WCHU (weighted churn) and LDHH (linear decay of the entropy over time for the churn metrics).

Graves et al. [7] showed that the number of prior changes to a file is as a good fault predictor. Specifically, they found that the number of lines of code of a module is not helpful in predicting faults when the number of times a module was changed is taken into account. Authors concluded that the change history contains more useful information than could be obtained from the size and structure metrics.

Moser et al. [17] conducted a comparative analysis of the efficiency of the process and the product metrics for defect prediction and showed that process metrics perform definitely better. They also reported that pre-release defects can be successfully used in predicting post-release defects.

3. Data Acquisition

48 releases of 15 open-source and 38 releases of 7 proprietary projects were investigated in order to collect the product metrics. Unfortunately, in the case of process metrics some data was missing and therefore, all four process metrics were collected only for 36 releases of 6 proprietary projects. The defect prediction models and discriminant analysis were employed only for the second set. The majority of the collected data was used in other studies [10,15]. However, some data is new. One new proprietary project, that consists of eight versions and three new versions of one of the preexisting proprietary projects are analyzed in this study.

The following open-source projects were investigated (descriptions after [10]):

- Apache Ant (http://ant.apache.org/). Ant is a well-known Java-based, shell independent build tool.
- Apache Camel (http://camel.apache.org/). Apache Camel is a powerful open source

integration framework based on known Enterprise Integration Patterns with powerful Bean Integration.
- Apache Ivy (http://ant.apache.org/ivy/). Ivy is a dependency manager that is focused on flexibility and simplicity.
- Apache Log4j (http://logging.apache.org/log4j/). Logging package for printing log output to different local and remote destinations. Log4j supports selection and filtering of log events at runtime.
- Apache Lucene (http://lucene.apache.org/). Lucene provides Java-based indexing and search technology, as well as spellchecking, hit highlighting and advanced analysis / tokenization capabilities.
- Apache POI (http://poi.apache.org/). The POI project consists of APIs for manipulating various file formats based upon Microsoft's OLE 2 Compound Document format, and Office OpenXML format, using pure Java.
- Apache Synapse (http://synapse.apache.org/). Synapse is a simple, lightweight and high performance Enterprise Service Bus (ESB) made by Apache. Synapse has support for HTTP, SOAP, SMTP, JMS, FTP and file system transports, Financial Information eXchange (FIX) and Hessian protocols for message exchange as well as first class support for standards such as WS-Addressing, Web Services Security (WSS), Web Services Reliable Messaging (WSRM), efficient binary attachments (MTOM/XOP).
- Apache Tomcat (http://tomcat.apache.org/). Tomcat is an open source servlet container that implements the Java Servlet and the JavaServer Pages (JSP), and provides a "pure Java" HTTP web server environment for Java code to run.
- Apache Velocity (http://velocity.apache.org/). Velocity is a Java-based template engine. It permits anyone to use a simple yet powerful template language to reference objects defined in Java code. Velocity separates Java code from the web pages, making the web site more maintainable over its lifespan and providing a viable alternative to Java Server Pages (JSPs) or PHP.
- Apache Xalan-Java (http://xml.apache.org/xalan-j/). Xalan is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. It implements XSL Transformations (XSLT) Version 1.0 and XML Path Language (XPath) Version 1.0.
- Apache Xerces (http://xerces.apache.org/xerces-j/). Xerces is a parser that supports the XML 1.0 recommendation and contains advanced parser functionality, such as support for XML Schema 1.0, DOM level 2 and SAX version 2.
- Ckjm (http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/). Ckjm calculates 19 different size and complexity metrics by processing the bytecode of compiled Java files.
- JEdit (http://www.jedit.org/). JEdit is a cross platform programmer's text editor written in Java.
- PBeans (http://pbeans.sourceforge.net/). PBeans is a Java persistence layer and an object/relational database mapping (ORM) framework.

Six of the investigated proprietary projects are a custom built, enterprise solution that has been already successfully installed in the customer environment. They belong to the insurance domain and were built on top of Java-based frameworks. The seventh project is a tool, which supports quality assurance in software development.

The product metrics were calculated using the ckjm[1] tool. Process metrics as well as the data regarding number of defects were collected using the BugInfo[2] tool. The collected metrics were published in the Metrics Repository[3] as well as in the PROMISE Repository[4]. Further description of the collected data is skipped since the data set is available on internet and, therefore, everyone can easily access and look through it.

Following product metrics were analyzed (definitions after [11]:
- Chidamber & Kemerer metrics suite [4]:
  - Weighted Methods per Class (WMC). The value of the WMC is equal to the number of methods in the class (assuming unity weights for all methods).
  - Depth of Inheritance Tree (DIT). The DIT metric provides for each class a measure of the inheritance levels from the object hierarchy top.
  - Number of Children (NOC). The NOC metric simply measures the number of immediate descendants of the class.
  - Coupling between object classes (CBO). The CBO metric represents the number of classes coupled to a given

1 http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/
2 http://kenai.com/projects/buginfo
3 http://purl.org/MarianJureczko/MetricsRepo
4 http://promisedata.org/

class (efferent couplings and afferent couplings). These couplings can occur through method calls, field accesses, inheritance, method arguments, return types, and exceptions.
    - Response for a Class (RFC). The RFC metric measures the number of different methods that can be executed when an object of that class receives a message. The response set is calculated by inspecting method calls within the class method bodies. The value of RFC is the sum of number of methods called within the class method bodies and the number of class methods.
    - Lack of Cohesion in methods (LCOM). The LCOM metric counts the sets of methods in a class that are not related through the sharing of some of the class fields. It considers all pairs of class methods. In some of these pairs both methods access at least one common field of the class, while in other pairs the two methods do not share any common field accesses. The lack of cohesion in methods is then calculated by subtracting from the number of method pairs that do not share a field access the number of method pairs that do.
- A metric suggested by Henderson-Sellers [8]: Lack of Cohesion in Methods (LCOM3). LCOM3 is a normalized version of the Chidamber and Kemerer's LCOM metric and can be calculated using the following equation:

$$LCOM3 = \frac{(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)) - m}{1 - m} \quad (1)$$

where m is the number of methods in a class; a is the number of attributes in a class and $\mu(A)$ is the number of methods that access the attribute A.
- Martin's metrics [14]:
    - Afferent Couplings (Ca). The Ca metric represents the number of classes that depend upon the measured class.
    - Efferent Couplings (Ce). The Ce metric represents the number of classes that the measured class is depended upon.
- QMOOD metrics suite [2]:
    - Number of Public Methods (NPM). The NPM metric simply counts all the methods in a class that are declared as public. The metric is known also as Class Interface Size (CIS).
    - Data Access Metric (DAM). This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class.
    - Measure Of Aggregation (MOA). The metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of class fields whose types are user defined classes.
    - Measure of Functional Abstraction (MFA). This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by the member methods of the class. The constructors and the java. lang.Object (as parent) are ignored.
    - Cohesion Among Methods (CAM). This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods.
- quality oriented extension of Chidamber & Kemerer metrics suite [21]:
    - Inheritance Coupling (IC). This metric provides the number of parent classes to which a given class is coupled. A class is coupled to its parent class if one of its inherited methods functionally dependent on the new or redefined methods in the class. A class is coupled to its parent class if one of the following conditions is satisfied:
        - One of its inherited methods uses an attribute that is defined in a new/redefined method.
        - One of its inherited methods calls a redefined method.
        - One of its inherited methods is called by a redefined method and uses a parameter that is defined in the redefined method.

  - Coupling Between Methods (CBM). The metric measures the total number of new/redefined methods to which all the inherited methods are coupled. There is a coupling when at least one of the given in the IC metric definition conditions is held.
  - Average Method Complexity (AMC). This metric measures the average method size for each class. Size of a method is equal to the number of Java binary codes in the method.
- McCabe's cyclomatic complexity measure [16]. McCabe's cyclomatic complexity (CC) is equal to number of different paths in a method (function) plus one. The cyclomatic complexity is defined as:

$$CC = E - N + P \tag{2}$$

  where E - the number of edges of the graph, N - the number of nodes of the graph, P - the number of connected components. CC is the only method size metric. For the sake of prediction models class level metrics Max(CC) and Avg(CC) have been derived.
- Lines Of Code (LOC).

Following process metrics were analyzed (definitions after [15]):

- Number of Revisions (NR). The NR metric represents the number of revisions of a given Java class during development of the investigated release of a software system.
- Number of Distinct Commiters (NDC). The NDC metric counts the number of distinct authors, usually developers, who committed their changes in a given Java class during the development of the investigated release of a software system.
- Number of Modified Lines (NML). The value of the NML metric is equal to the sum of all lines of source code which were added or removed in a given Java class. Each of the revisions which were committed during the development of the investigated release of a software system is considered. According to the CVS version-control system, a modification in a given line of source code is equivalent to removing the old version and subsequently adding a new version of the line.
- Number of Defects in Previous Version (NDPV). The NDPV metric counts the number of defects which were repaired in a given class during the development of the previous release of a software system.

4. Correlation Analysis

Pearson correlation coefficients calculated for all investigated projects, for each version separately are presented on Figure 1. According to the figure, all process metrics (NR, NDC, NML and NDPV) are highly correlated with the number of defects. The finding is also supported by Table 1 – the obtained correlation coefficients are equal 0.17 to 0.33. According to Figure 1 the highest correlation coefficient was obtained in the case of the NR metric and according to Table 1 it is in the case of the NDPV metric. Figure 1 presents the distribution of correlation coefficients among different versions of software projects, while Table 1. presents results of calculations where the information about the project version was ignored.

The correlations between product metrics and number of defects are not as uniform as in the case of process metrics. There are metrics with negative correlation coefficients. The lowest correlation rank was obtained in the case of the CAM metric. The DIT, NOC, IC and CBM metrics have correlation coefficients close to 0. However, there are also product metrics that are strongly correlated with the number of defects: WMC, CBO, RFC, Ce and LOC.

FIGURE 1
PEARSON CORRELATION COEFFCIENTS FOR ALL INVESTIGATED PROJECTS, CALCULATED FOR EACH PROJECT VERSION SEPARATELY

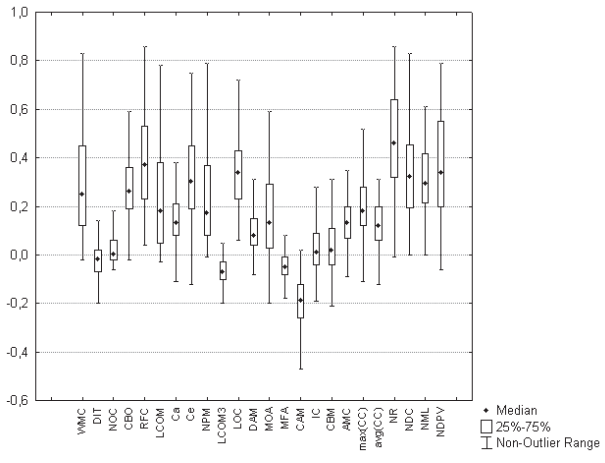
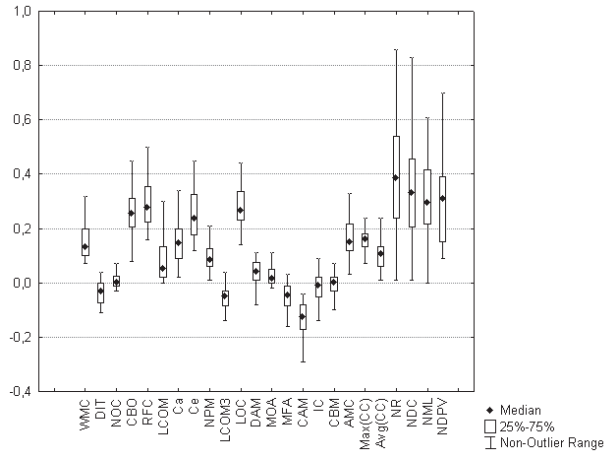


FIGURE 2
PEARSON CORRELATION COEFFCIENTS FOR PROJECTS WHICH WERE USED TO BUILD DEFECT PREDICTION MODELS, CALCULATED FOR EACH PROJECT VERSION SEPERATLEY



Correlation coefficients with number of defects that were calculated only for versions of projects used to build models are presented on Figure 2. The median values are very similar to median values from Figure 2. However, the variance of obtained values was significantly decreased. Nevertheless, the same metrics are strongly correlated with the number of defects and the same metrics are not correlated with them.

TABLE 1
PEARSON CORRELATION COEFFCIENTS CALCULATED FOR ALL INVESTIGATED PROJECTS COLLECTIVLY COEFFICIENTS DENOTED WITH '*' ARE STATISTICALLY SIGNIFICANT ON $\alpha = 0.05$

Metric	Correlation
WMC	0.24*
DIT	-0.07*
NOC	0.00
CBO	0.19*
RFC	0.30*
LCOM	0.10*
Ca	0.15*
Ce	0.15*
NPM	0.20*
LCOM3	-0.09*
LOC	0.28*
DAM	0.10*
MOA	0.12*
MFA	-0.08*
CAM	-0.17*
IC	-0.04*
CBM	0.03*
AMC	0.10*
Max(CC)	0.17*
Avg(CC)	0.10*
NR	0.31*
NDC	0.27*
NML	0.17*
NDPV	0.33*

5. METRICS IN DEFECT PREDICTION MODELS

Together 23 different defect prediction models were constructed. Each of the models was trained on different version of project. Unfortunately, this analysis had to be limited to the industrial projects, since there were issues regarding collecting process metrics in the open-source projects. The training set consisted of the all aforementioned product and process metrics. The models were built using stepwise linear regression. In linear regression, the model specification is that the dependent variable is a linear combination of the parameters (independent variables). Stepwise linear regression is a model build technique that finds subsets of predictor variables that most adequately predict responses on a dependent variable by linear regression, given the specified criteria for adequacy of model fit. In this case the F-test was used as the criterion.

Due to the applied regression method, each of the constructed models used only several metrics (usually five to ten), not all of them. The numbers of occurrences of different metrics in the obtained models are presented on Figure 3. The most frequently used metric was the NR metric – it was used in 14 different defect prediction models. Following metrics also were relevant: RFC, NOC, LOC, NML, NDPV and NDC – each of them were used in 9 or more different defect prediction models. The least frequently used metrics were: MOA, IC and WMC; each of them were used only in one defect prediction model.

Average prediction efficiency for the investigated models was equal 50%. The models were assessed by investigating the percentage of Java classes that must be covered in order to find 80% of the defects in a given software project. Let assume that: $E(M_i)$ is the evaluation of the efficiency of the model M_i in predicting defects in release $i+1$. The model built on release i forms the basis for making predictions

in release $i+1$ of the same project. Let n be the number of classes in release i. Let $c_1, c_2, ..., c_n$ denote the classes from release i in descending order of numbers of predicted defects according to the model M_{i-1}, and $d_1, d_2, ..., d_n$ be the number of defects in each class.

$$D_j = \sum_{l=0}^{j} d_l \tag{3}$$

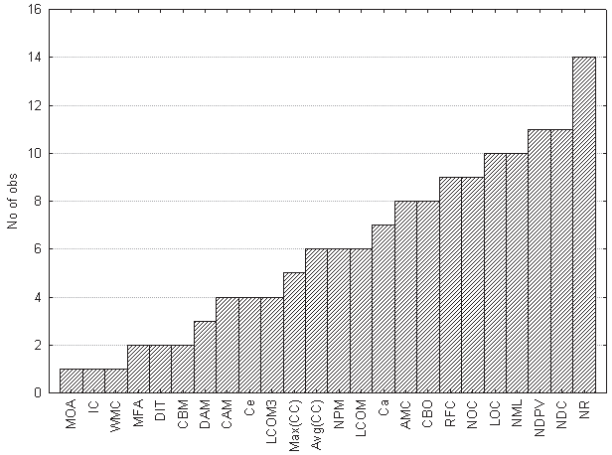
D_j is the total defects in the first j classes. Let k be the smallest index so that:

$$D_k > 0.8 * D_n \tag{4}$$

then the assessment can be calculated using following formula:

$$E(M_{i-1}) = \frac{k}{n} * 100\% \tag{5}$$

FIGURE 3
METRICS OCCURENCES IN DEFECT PREDICTION MODELS



6. Discriminant Analysis

The discriminant analysis is a method used in statistics to find a linear combination of features which characterize or separate two or more classes of objects. The resulting combination may be used as a linear classifier or, more commonly, for dimensionality reduction before later classification. In this study, the discriminant analysis was used to determine which metrics play key role in predicting whether a class is defective or defect free. The analysis was conducted on the same data set as in previous section, namely on the industrial projects. All projects were investigated collectively in one analysis.

The investigated classes were splitted into two set. The first set contained classes with at least on defect (defective), the second set contained only defect free classes. Subsequently, the discriminant analysis was conducted and all aforementioned product and process metrics were employed in building the discriminant functions. The results were as follows – Wilks's Lambda equal 0.88511. It does not denote high discriminant power, since the optimal value is 0. Nevertheless, the statistic F (23,74470) was equal 420.27 and the p value was less than 0,0001. Hence, the obtained results are significant.

Detailed results are presented in Table 2. The most important value with regard to significance in defect prediction in this table is the partial Wilks' Lambda. This value denotes the discriminant power of a variable. Partial Wilks' Lambda equal 0 denotes perfect discriminant power. Unfortunately, the obtained values are high and therefore, the results are not conclusive. Nevertheless, the best results were obtained in the case of the NDC, NDPV and CAM metrics, what only partly corresponds with results obtained in the two other analyses (section 4 and 5).

Table 2 shows also several others interesting statistics. Wilks' Lambda represents the Wilks' Lambda for the overall model that will result after removing the respective variable. F to remove is the statistic F associated with the respective partial Wilks'Lambda. P is associated with the respective F. The tolerance value is computed as 1 minus R-square of the respective variable with all other variables included in the current model. Thus, it is the proportion of variance that is unique to the respective variable.

TABLE 2
DISCRIMINANT ANALYSIS SUMMARY

	Wilks' Lambda	Partial Wilks' Lambda	F to remove	p	Tolerance
NDC	0,891200	0,993169	512,1971	0,000000	0,210497
NDPV	0,890865	0,993543	483,9489	0,000000	0,744417
CAM	0,889251	0,995347	348,1407	0,000000	0,765632
CBO	0,886510	0,998424	117,5827	0,000000	0,083411
DIT	0,886027	0,998968	76,9059	0,000000	0,237317
Ca	0,885949	0,999056	70,3454	0,000000	0,096925
RFC	0,885697	0,999340	49,1548	0,000000	0,139449
MFA	0,885537	0,999521	35,6699	0,000000	0,311232
DAM	0,885448	0,999622	28,1772	0,000000	0,612836
AMC	0,885322	0,999764	17,6041	0,000027	0,386163
NOC	0,885181	0,999923	5,7704	0,016304	0,711307
Avg(CC)	0,885181	0,999924	5,6994	0,016977	0,290469
WMC	0,885132	0,999979	1,5980	0,206203	0,044243
IC	0,885132	0,999979	1,5802	0,208741	0,167265
NML	0,885128	0,999983	1,2961	0,254932	0,594554
NPM	0,885127	0,999984	1,1896	0,275421	0,068383
MOA	0,885126	0,999986	1,0831	0,298023	0,981911
LOC	0,885123	0,999989	0,8167	0,366144	0,250063
LCOM	0,885118	0,999994	0,4616	0,496868	0,361736
NR	0,885117	0,999995	0,3906	0,531983	0,165525
CBM	0,885116	0,999996	0,2841	0,594043	0,257962
LCOM3	0,885114	0,999998	0,1243	0,724435	0,543870
Max(cc)	0,885114	0,999999	0,1065	0,744132	0,261123

The correctness of classification obtained from the discriminant functions is presented in Table 3. According to the table 87.8% Java classes were classified correctly. Unfortunately, 7706 defect free classes were classified as defective. In fact, only 2813 classes were defective. Therefore, the prediction 'defective' is not very reliable.

TABLE 3
CLASSIFICATION MATRIX

	Percent correct	Defect free (observed)	Defective (observed)
Defect free (predicted)	97.939	63975	1346
Defective (predicted)	15.993	7706	1467
Total	87.849	71681	2813

7. THREATS TO VALIDITY

7.1 Construct validity

It is not possible to guarantee that all the links between defects and versioning system files (SVN, CVS) are retrieved, since the defects are identified according to the comments in the source version control system. In fact, this is a widely known problem and the adopted method represents the current state of the art [23]. Especially it was not possible to link defect with anonymous inner classes due to file-based nature of the source code version control systems. Therefore, not taking inner classes into consideration has become a common practice [1,5]. In fact, the aforementioned issue is the only data filtering method employed in this study.

It was not possible to track operations like changing class name or moving class between packages. Therefore, after such a change, a class is always interpreted as a new one.

The defects are assigned to file version (and class) according to bugfix date. Therefore, it was not possible to track down unsolved defects and to link defects to the version where it was actually made.

7.2 Statistical conclusion validity

Threats to statistical conclusion validity relate to the issues that affect the validity of inferences. In this study robust statistical tool was used: Statistica.

7.3 Internal validity

The threats to internal validity concern the external factors that might affect the outcomes observed in the study. Probably the only one external factor in defect prediction studies is the human factor, namely the decision about defect severity. According Ostrand et al. [19] the severity rating is highly subjective and inaccurate and therefore was not taken into consideration in this study.

7.4 External validity

A wide range of different kinds of projects were considered in this study. The projects represent two different source code ownership models (industrial and open-source, however, the open-source projects were used only in the correlation analysis), different application domains and different development processes. Nevertheless, all investigated projects were written in Java. The investigated projects may also share other features about which the author is not aware of.

8. Conclusions

All analysis, namely correlations, metrics occurrences in the defect prediction models and discriminant analysis showed high value of the process metrics (NR, NDC, NML or NDPV). All process metrics are correlated with number of defects on 0.3 or higher level (median values, Figure 1 and Figure 2). Furthermore, all the process metrics were frequently selected to the defect prediction model by the stepwise linear regression and two of the metrics (NDC and NDPV) have the greatest discriminant power.

Among the investigated product metrics, several may be pointed out as very useful in defect prediction. Those are: LOC, RFC, CBO, AMC and CAM. However, ignoring all the others metrics will not be very reasonable, since there are metrics that were frequently selected to the models despite of low correlations. Therefore, it leads to the conclusion that correlation analysis is not sufficient to asset the usefulness of software metric. Unfortunately, sometimes in research, the analysis goes no further [9]. The correlation is crucial only for univariate models, which are barely used for defect prediction. Usually the collinearity of all selected metrics is relevant. Moreover, the metric NOC, according to Figure 1, Figure 2 and Table 1 is not correlated with the number of defects. Nevertheless, it was selected to nine different defect prediction models (Figure 3). On the other hand there is the WMC metric, which is correlated with the number of defects. However, the WMC metric was used only in one model.

The obtained results are not unambiguous. However, some of the investigated metrics might be strongly recommended with regard to defect prediction. Several metrics were pointed out in the conducted experiments: NR, NDC, NML, NDPV, LOC, RFC, CBO and AMC. The metrics are correlated with the number of defects and all of them are frequently used in models that were built using stepwise linear regression. Some of them (NDC, NDPV, RFC, CBO) have also good discriminant power. Therefore, those eight metric might be recommended as the first selection with regard to defect prediction. Nevertheless, an interesting thing to note is the fact that it does not indicate low usefulness of the other investigated metrics. Hence, adding other software metrics to the aforementioned ones may significantly improve efficiency of defect prediction. The selected metrics constitute a solid basis. Nevertheless, other should be considered (when possible) since there are major differences between software projects and defects indicators that work well in one project may be useless in other [10,23].

Acknowledgment

Fellowship co-financed by European Union within European Social Fund.

References

[1] A. Bacchelli, M.D'Ambros and M.Lanza. "Are popular classes more defect prone?," *Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science*, 6013, 59–73, 2010.

[2] J.Bansiya and C.G Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, 4–17, 2002.

[3] C.Catal, B.Diri and B.Ozumut, "An Artificial Immune System Approach for Fault Prediction in Object-Oriented Software," in *2nd International Conference on Dependability of Computer Systems DepCoS-RELCOMEX,* 2007.

[4] S.Chidamber and C.F.Kemerer, "A metrics Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering*, vol. SE-20, no.6, 476-493, 1994.

[5] M. D'Ambros, M.Lanza and R.Robbes. "An extensive comparison of bug prediction approaches," in the *7th IEEE Working Conference on Mining Software Repositories*, MSR'2010, 31–41, 2010.

[6] N.E.Fenton, "Software Metrics: Successes, Failures and New Directions," *Journal of Systems and Software*, 47, 2-3, 149-157, 1999.

[7] T.L.Graves, A.F.Karr, J.S.Marron and H.Siy. "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol.26, no.7, 653–661, 2000.

[8] B.Henderson-Sellers, "Object-oriented metrics: measures of complexity," *UpperSaddle River*, NJ, USA: Prentice-Hall, Inc 1996.

[9] T.Illes-Seifert and B.Peach, "Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs," *Information and Software Technology,* vol. 52, 539-558, 2010.

[10] M.Jureczko and L.Madeyski, "Towards identifying software project clusters with regard to defect prediction," in the *6th International Conference on Predictor Models in Software Engineering,* PROMISE'2010.

[11] M.Jureczko and D.Spinellis "Using object-oriented design metrics to predict software defects," *Monographs of System Dependability* 1, 69-81, 2010.

[12] C.Kaner and W.Bond, "Software engineering metrics: What do they measure and how do we know?," in the 10th *International Software Metrics Symposium*, 2004.

[13] P.Knab, M.Pinzger and A.Bernstein, "Predicting defect densities in source code files with decision tree learners," in *the 2006 International Workshop on Mining Software Repositories*, 2006.

[14] R.Martin, "Oo design quality metrics - an analysis of dependencies," in *Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*, OOPSLA'94.

[15] L.Madeyski and M.Jureczko "Which process metrics improve software defect prediction models? An empirical study," submitted to *Information and Software Technology,* 2011.

[16] T.J.McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, 308–320, 1976.

[17] R.Moser,W.Pedrycz and G.Succi,. "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in the *30th International Conference on Software Engineering*, ICSE '08, 181–190, 2008.

[18] H.Olague, L.Etzkorn, S.Gholston and S.Quattlebaum, "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes," *IEEE Transactions on Software Engineering,* vol. 33, no. 6, 402-419, 2007.

[19] T.Ostrand, E.Weyuker and R.Bell, "Where the bugs are," in the *2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'04, 86–96, 2004.

[20] E.Shihab, A.M.Jiang, W.M.Ibrahim, B.Adams and A.E.Hassan, "Understanding the impact of code and process metrics on post-release defects: a case study on the Eclipse project," in *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement,* 2010.

[21] M.-H.Tang, M.-H.Kao and M.-H.Chen, "An empirical study on object-oriented metrics," in the *6th International Symposium on Software Metrics*, METRICS '99, 242-250, 1999.

[22] T.Zimmermann, R.Premraj and A.Zeller, "Predicting defects for eclipse," in the *Third International Workshop on Predictor Models in Software Engineering*, PROMISE 2007, 1-9, 2007.

[23] T.Zimmermann, N.Nagappan, H.Gall, E.Giger and B.Murphy, "Cross-project Defect Prediction," in *the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2009*, 91-100, 2009.

ABOUT THE AUTHOR

Marian Jureczko recived the MSc degree in computer science from Wroclaw University of Technology in 2006. Since October 2006 he is PhD student. His main research interests are software quality, software metrics, defect prediction and software testing.