Empirical Software Engineering – An International Journal manuscript No.
(will be inserted by the editor)

Design Evolution Metrics for Defect Prediction in Object Oriented Systems

Segla Kpodjedo · Filippo Ricca · Philippe Galinier · Yann-Gaël Guéhéneuc · Giuliano Antoniol

Received: date / Accepted: date

Abstract Testing is the most widely adopted practice to ensure software quality. However, this activity is often a compromise between the available resources and software quality. In object-oriented development, testing effort should be focused on defective classes. Unfortunately, identifying those classes is a challenging and difficult activity on which many metrics, techniques, and models have been tried. In this paper, we investigate the usefulness of elementary design evolution metrics to identify defective classes. The metrics include the numbers of added, deleted, and modified attributes, methods, and relations. The metrics are used to recommend a ranked list of classes likely to contain defects for a system. They are compared to Chidamber and Kemerer's metrics on several versions of Rhino and of ArgoUML. Further comparison is conducted with the complexity metrics computed by Zimmermann *et al.* on several releases of Eclipse. The comparisons are made according to three criteria: presence of defects, number of defects, and defect density in the top-ranked classes. They show that the design evolution metrics, when used in conjunction with known metrics, improve the identification of defective classes. In addition, they show that the design evolution metrics make significantly better predictions of defect density than other metrics and, thus, can help in reducing the testing effort by focusing test activity on a reduced volume of code.

Keywords Defect Prediction, Design Evolution Metrics, Error Tolerant Graph Matching

1 Introduction

In the software market, companies often face the dilemma to either deliver a software system with poor quality or miss the window of marketing opportunity. Both choices

S. Kpodjedo · P. Galinier · Y. G. Guéhéneuc · G. Antoniol
SOCCER Lab.
DGIGL, École Polytechnique de Montréal, Québec, Canada
E-mail: {segla.kpodjedo, philippe.galinier, yann-gael.gueheneuc.}@polymtl.ca,
E-mail: antoniol@ieee.org

F. Ricca
DISI, University of Genoa, Italy
E-mail: filippo.ricca@disi.unige.it

may have potentially serious consequences on the future of a company. Defects slipping from one release to the next release may harm the image and trust of the users into the companies; delaying a release may give competitors a commercial advantage.

However, software development is labor intensive and software testing can cost up to 65% of available resources [MGM06]. Testing activities (*e.g.*, unit, integration, or system testing) are often performed as "sanity checks" to minimize the risk of shipping a defective system.

A large body of work on object-oriented (OO) unit and integration testing focuses on the important problem of minimizing the cost of test activities while fulfilling a clear test coverage criteria (*e.g.*, [BLW03]). We believe that previous work does not fully address the problem of assessing the cost of testing activities that must be devoted to a class: it leaves managers alone in the strategic decision of allocating resources to focus the testing activities.

For example, let us consider a manager who wants to substantially improve the quality of a large OO system in its next release. She needs to know what are the key classes on which to focus testing activities, *i.e.*, allocate her resources. We believe that key classes can be defect-prone classes, *i.e.*, classes which have the highest risk of producing defects and classes from which a defect could propagate extensively across the system. Although reliability or dependability are the ultimate goals, locating defects is crucial. Provided with a ranked list of classes likely to contain defects, the manager can decide to prioritize testing activities based on her knowledge of the project (frequency of execution or relevance to the project of the classes). Consequently, the manager would benefit from an approach to identify defective classes.

Many approaches to identify defective classes have been proposed in the literature. They mainly use metrics and machine learning techniques to build predictive models. However, as of today, researchers agree that more work is needed to obtain predictive models usable in the industry[1]. This paper contributes to the field by investigating the identification of defective classes using design evolution metrics based on changes observable in the designs of OO systems.

In a previous work [KRAG09], we showed that metrics based on class evolution and class connectivity help in identifying defective classes. We computed these metrics using an Error Tolerant Graph Matching (ETGM) algorithm [KRG$^+$10] applied to subsequent releases of a system. Our algorithm computes, for each class, its *Class Rank* (CR) [PBMW98] and its *Evolution Cost* (EC) [KRAG09], where CR measures the relative importance of the class in the system while EC, given the class history, quantifies the amount of changes in signature and relations (*i.e.*, associations, aggregations, and generalizations) that the class underwent.

CR and EC are computed quantities summarizing the evolution of a class; as aggregated measures, they sometimes mask the finer-grain data available. Indeed, it seems reasonable to assume that details about design changes may help to locate classes that underwent risky changes and are likely to contain defects. For example, if a high number of method signatures have been modified in a new version of a class, then this class is more defect-prone than other classes that underwent less or no changes, because of possible unwanted side effects.

Thus, we introduce a new set of metrics, the design evolution metrics (DEM), computed on the basic design changes used previously to compute CR and EC. The

[1] Researchers discussed the limits of current predictive models at the 6th edition of Working Conference on Mining Software Repositories (MSR'09).

DEM include metrics counting the additions, modifications, or deletions of attributes, methods, or relations in the classes between releases of a system. We build models using the DEM and other metrics to study their explanatory and predictive power to identify defective classes. We compare the DEM with traditional object-oriented and complexity metrics when included in models for (1) explaining defects in a system, (2) identifying defective classes, (3) predicting the number of defects in a class, and (4) predicting the defect density in a class. We perform our comparisons on 7 releases of Rhino, 9 of ArgoUML, and 3 of Eclipse.

Our comparisons show that the DEM improve, with statistical significance, the identification of defective classes. The DEM have, in particular, a very good predictive power when predicting defect density, *i.e.*, identifying classes providing a high number of defects in a small amount of code volume. Therefore, they are able to support a manager in the difficult task of choosing the classes on which to concentrate her resources.

This paper is organized as follows. Section 2 presents related work. Section 3 recalls our ETGM algorithm [KRAG09, KRG+10, KRGA08] for the sake of completeness and presents the design evolution metrics. Section 4 describes our case study and Section 5 presents and discusses its results. Section 6 highlights threats to validity and Section 7 concludes and outlines future work.

2 Related Work

Predicting location, number or density of defects in systems is a difficult task that has been studied in several previous works. We focus on the works most closely related to ours: the use of metrics extracted from design or code, project or software historical data, and code churns to predict defects.

2.1 Static OO Metrics

Several researchers identified correlations between static OO metrics, such as the Chidamber and Kemerer (C&K) metrics [CK94], and location of defects. The intuition supporting the use of complexity metrics for the prediction of defective classes is that complex code is more defect-prone than simple code.

Basili *et al.* [BBM96] were among the first to use the OO metrics proposed by C&K [CK94] to predict defective classes. To validate their work, these authors collected data on the development of eight similar small-sized information management systems (180 classes in total). All eight systems were developed using an OO analysis/design method and the C++ programming language. Results showed that five out of the six considered metrics, defined in Section 4.2, appear to be useful to predict defective classes: WMC, DIT, RFC, NOC, CBO. LCOM was not found useful.

Another empirical study [CS00] conducted on an industrial C++ system (over 133 KLOC) supported the hypothesis that classes participating in inheritance structures have a higher defect density than others. It followed that C&K's DIT and NOC metrics could be used to identify classes that are likely to be more defective, thus confirming the previous work by Basili *et al.*

More recently, Gyimothy *et al.* [GFS05] compared the accuracy of a large metric suite, including C&K metrics, to predict defective classes in the open-source system

Mozilla. They concluded that CBO is the best predictive metric. They also found LOC to be useful in the prediction.

Zimmermann *et al.* [ZPZ07] related bug reports for Eclipse (releases 2.0, 2.1, and 3.0) to fixes, *i.e.*, they computed the mapping of packages and files to the number of defects in each considered release. They conducted an empirical study using common complexity metrics (*e.g.*, fan-in and fan-out) to define prediction models. Their models showed that a combination of complexity metrics can predict defects, suggesting that the more complex a class, the more defective. We use in this paper their work as a comparison basis to evaluate our proposed metrics.

El Eman [EBGR01] showed that, after controlling for the confounding effect of "size", the correlation between OO metrics and defect-proneness disappeared: many OO metrics are correlated with size and, therefore, "add nothing" to the models that predict defects. This latter work can be regarded as an incentive to develop new metrics, possibly based on software evolution, to avoid strong correlation with size.

2.2 Historical Data

Some researchers used historical data to predict defects, following the intuition that systems with defects in the past will also have defects in the near future.

Ostrand *et al.* [OWB05] proposed a negative binomial regression model based on various metrics (*e.g.*, number of defects in previous releases, code size) to predict the number of defects per file in the next release. The model was applied with success on two large industrial systems, one with a history of 17 consecutive releases over four years, the other with 9 releases over two years. For each release of the two systems, the top 20% of the files with the highest predicted number of defects contained between 71% and 92% of the defects actually detected, with an overall average of 83%. We adapt in the following their approach to evaluate the results of the predictive models by focusing on the top recommended classes.

Graves *et al.* [GKMS00] presented a study on a large telecommunication system of approximately 1.5 million lines of code. They used the system defect history in a two-year period to build several predictive models. These models were based on combinations of the ages of modules, the number of changes done to the modules, and the ages of the changes. They showed that size and other standard complexity metrics were generally poor predictors of defects compared to metrics based on the system history.

2.3 Code Churn

Code churn is the amount of change taking place within the code source of a software unit over time [NB05]. It has been used by some researchers to identify defective artifacts: changes often introduce new defects in the code. We share with these researchers the intuition that frequently-changed classes are most likely to contain defects.

Nagappan *et al.* [NB05] used a set of relative code churn measures to predict defects. Predictive models were built using statistical regression models using several measures related to code churn (*e.g.*, Churned LOC, the sum of added and changed lines of code between two releases of a file). They showed that the absolute measures of code churn generate a poor predictor while the proposed set of relative measures form a good

predictor. A case study performed on Windows Server 2003, with about 40 million lines of code, illustrated the effectiveness of the predictor by discriminating between defective and non-defective files with an accuracy of 89%.

Munson *et al.* [ME98] predicted defects in an embedded real-time system using the notion of code churn over 19 successive versions. The analyzed system is composed of 3,700 C modules for about 300 KLOC. Code churn metrics were found to be among the most highly correlated metrics with bug reports (Pearson correlation of 0.65).

Hassan [Has09] proposed a measure of entropy for code changes based on the idea that a change affecting only one file is less complex than a change affecting many different files. The proposed metrics, based on Shannon's entropy, were proven superior to metrics based on prior defects and–or changes for six different systems.

Moser *et al.* [RM08] proposed 17 change metrics at the file level, ranging from the number of refactorings, authors, bug fixes, age to various measures of code churn. Using three different binary classifiers, they found that their metrics were significantly better than the ones of Zimmermann *et al.* on the Eclipse data set. Replication value of this work was unfortunately impaired as the new metric data set was not made publicly available.

We share with the presented previous work the general problem of identifying defective classes. As those previous works, we use metrics, *i.e.*, change metrics, and exploit the evolution of designs. While some previous works investigated mostly changes in the sizes of files, we focus on metrics reporting changes visible in the design of OO systems and prove the efficiency of such metrics in explaining and predicting defects.

3 Design Evolution Metrics

The *DEM* aim at capturing elementary design evolution changes. In our study, we represent systems by their class diagrams, because such diagrams are simple to reverse engineer from source code and are often used or altered during development and maintenance. They capture design changes, such as additions or deletions of methods and attributes.

Identifying and counting design changes between two releases of the class diagram of an evolving system of realistic size is tedious and error-prone. Therefore, to automate the computing of the DEM, we first compute an optimal or sub-optimal matching of subsequent class diagrams to retrieve any class evolution through time. Second, once this data is obtained, we compute the proposed design evolution metrics.

In the following subsections, we first define the DEM and then present the problem of retrieving the evolution of the class diagram of an OO system. Our solution to this latter problem is based on an Error-Tolerant Graph Matching (ETGM) algorithm.

3.1 Definitions

In our approach, we consider simple design evolution metrics pertaining to basic changes that affect the design of an OO system. We show in Sections 4 and 5 that these metrics can identify classes with high defect-density and complement previously-used metrics.

We assume that the evolution of classes is available, extracted by hand or computed by an algorithm, *e.g.*, the ETGM algorithm presented in the following subsection. Once

the evolution of classes is available, we count the numbers of simple design changes. At this stage of the research, we consider as relations: associations, aggregations, and generalizations. Also, we do not consider modified attributes, changes to the visibility, and changes of relations, which will all be studied in future work. Thus, we use the following counts:

- Number of added methods: *nbAddMet*
- Number of added attributes: *nbAddAtt*
- Number of added outgoing relations: *nbAddRelOut*
- Number of added incoming relations: *nbAddRelIn*
- Number of deleted methods: *nbDelMet*
- Number of deleted attributes: *nbDelAtt*
- Number of deleted outgoing relations: *nbDelRelOut*
- Number of deleted incoming relations: *nbDelRelIn*
- Number of modified methods: *nbModMet*
- Number of modified outgoing relations: *nbModRelOut*
- Number of modified incoming relations: *nbModRelIn*

Once the class diagram evolution is available, the above metrics can be easily computed as follows. Let a class C be represented by the quadruple (A, M, R_{in}, R_{out}) with A representing the set of attributes, M the set of methods, R_{in} the set of incoming relations, and R_{out} the set of outgoing relations.

If a class $C_1(A_1, M_1, R_{in1}, R_{out1})$ is matched with another $C_2(A_2, M_2, R_{in2}, R_{out2})$, then $A = A_1 \cap A_2$ (respectively, $M = M_1 \cap M_2$) represents the set of matched attributes (respectively, methods)[2].

Each element in $A_1 - A$ counts as a deleted attribute while each element in $A_2 - A$ counts as an added attribute. Modified methods are methods sharing the same name and either the same return type or input type(s). New relations count as additions while relations present in previous release and absent from the new one count as deletions. An added or deleted relation is also counted as a modified relation when the two classes involved were present in a previous release.

3.2 Class Diagram Evolution Problem

We compute the evolution of classes by assuming that OO systems evolve over time and, thus, that their class diagrams are subject to modifications and restructuring: classes, methods, and attributes are created, deleted, and modified.

Let us consider two versions D_1 and D_2 of the class diagram of a system, observed at two different instants in time and let $Cl(D_1)$ and $Cl(D_2)$ denote the set of classes in D_1 and D_2. The Class Diagram Evolution (CDE) problem consists in finding the true matching between the two class diagrams, *i.e.*, the one-to-one partial function m that indicates, for each class c in $Cl(D_1)$, if c was deleted or not and, if not, which class $m(c)$ in $Cl(D_2)$ is the new version of c. The function $m : Cl(D_1) \rightarrow Cl(D_2)$ defines a mapping between the two class diagrams in which a *class match* is any couple $(c, c') \in Cl(D_1) \times Cl(D_2)$ such that $c' = m(c)$ (we write $(c, c') \in m$). If $(c, c') \in m$, then, according to m, c' represents the new version of c.

[2] An attribute is matched to another if they share the same name and type while a method is matched to another if they share the same signature.

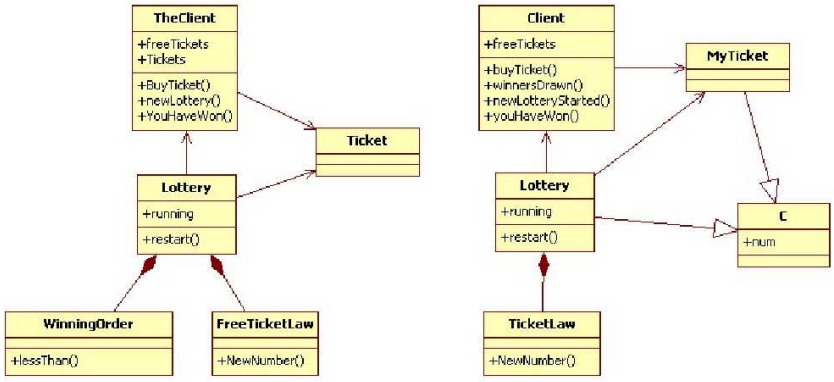

Fig. 1 Class diagram evolution example

For example, in Figure 1, we can assume that: class $WinningOrder$ was deleted; class C was created; classes $TheClient$, $Ticket$, $FreeTicketLaw$ were renamed to $Client$, $MyTicket$, $TicketLaw$, respectively. We can therefore assume that the following matching is true:

($TheClient \rightarrow Client$,
$Lottery \rightarrow Lottery$,
$Ticket \rightarrow MyTicket$,
$FreeTicketLaw \rightarrow TicketLaw$)

3.3 Modeling the CDE Problem as an ETGM Problem

We model the class diagrams of two releases as labeled graphs, with nodes being classes and edges representing the relations among classes. Labels on edges specify the type of the relation: association, aggregation, or inheritance; labels on nodes encode class properties: class name, methods, and attributes data.

We then translate the CDE problem into an optimization problem: the differences between the two class diagrams stem from edit operations with assigned costs, accounting for both internal and structural differences. Given some cost parameter values, we solve the optimization problem using an ETGM algorithm able to find a matching with minimal overall cost. Costs account for four different types of discrepancies between the two class diagrams: class insertions or deletions, internal class discrepancies, structural errors, and relation type discrepancies.

- We say that a class $c \in Cl(D_1)$ is *deleted* if class c is not involved in m: there is no c' such that $(c, c') \in m$. Similarly, we say that a class $c' \in Cl(D_2)$ is *inserted* if class c' is not involved in m. A penalty cost is paid for every deleted class and inserted class.
- We call *internal information* of a class the name of the class, its set of attributes, and its set of method signatures. Given two classes c and c' with their names represented by l and l', their attribute sets by A and A', their method-signature

sets by M and M', we compute [KRGA08] the internal similarity $intSim(c, c')$ between c and c' as:

$$intSim(c, c') = l_w \times \left(1 - \frac{Levenshtein(l, l')}{max(length(l), length(l'))}\right) + m_w \times Jaccard(SetAs, SetAs') + a_w \times Jaccard(SetMs, SetMs') \quad (1)$$

where $Levenshtein()$ is the Levenshtein distance between two strings; $Jaccard()$ the Jaccard index between two sets of strings; l_w, m_w and a_w the weights of the class name, methods, and attributes[3]. The similarity function is normalized between 0 and 1. For each class match (c, c'), there is a price to be paid proportional to $1 - intSim(c, c')$.

- A *structural error* occurs for two class matches (c, c') and (d, d') in m, if there is a relation between c and d but not between c' and d'. A unitary cost is paid for every structural error.
- A *relation type discrepancy* occurs for two class matches (c, c') and (d, d') in m, if there is relation between c and d and also between c' and d', but these two relations do not have the same type.

The cost function is governed by a set of parameters (*e.g.*, l_w, m_w, a_w). To be a true matching, an optimal matching (the matching with the lowest cost) depends on the choice of the parameter values. We discovered an appropriate set of parameter values by performing an extensive study [KRG+10] and used these values to perform all the computations presented in the following sections.

3.4 ETGM Algorithm

Solving the ETGM problem is known to be NP-hard [TF79]. In general, given a pair of graphs containing n and m nodes, the search space cardinality is of the order of n^m. Exact algorithms that can guarantee optimal solutions have an exponential worst-case complexity: they require prohibitive computation times even for medium-size graphs. Therefore, we resort to heuristics to obtain near optimal solutions with good performance.

We use a taboo search algorithm [Glo86] to implement an ETGM algorithm. Starting from an initial matching, our algorithm builds and transforms iteratively the current matching by inserting or removing a class match. At each iteration, the algorithm performs the best local transformation, *i.e.*, the transformation that produces the matching with the lowest cost. The taboo mechanism is used to avoid cycling.

The initial matching used by the algorithm contains all class matches (c, c'), such that classes c and c' share the same name. The local search that follows eventually match classes that do not share a same name. More details about the algorithm are available in our previous work [KRG+10].

[3] In the study, we fix $l_w = 0.6$, $m_w = 0.2$, and $a_w = 0.2$.

Systems	Releases (Number Thereof)	Number of Classes	Number of LOCs	Number of Defects
Rhino	1.5R1–1.6R1 (7)	89–270	30,748–79,406	12–114
ArgoUML	0.12–0.26.2 (9)	792–1,841	128,585–316,971	25–187
Eclipse	2.0–3.0 (3)	4,647–17,167	781,480–3,756,164	1044–2502

Table 1 Summary of the object systems

4 Case Study

The description of the study follows the Goal-Question-Metrics paradigm [BCR94]. The *goal* of this empirical study is to compare the efficiency of the DEM in explaining and predicting defects in classes with regard to other previously-used metrics. The *quality focus* is to achieve a prediction better than that of the predictors based on the C&K metrics and on the complexity metrics computed by Zimmermann *et al.* [ZPZ07]. The *perspective* is that of both researchers, developers, and managers, who want to identify defective classes. The *context* of this study are three open-source systems: the Rhino JavaScript/ECMAScript interpreter, the ArgoUML CASE tool, and the Eclipse Integrated Development Environment (IDE).

This study extends our previous study [KRAG09] by answering the same general research question with evolution metrics: *do evolution metrics improve prediction accuracy in identifying defective classes with respect to other previously-used metrics, such as the C&K metrics?* We also consider several releases of three different systems, while in our previous work we used only Rhino v1.6R5 to which all past defects were assigned. Also, we emphasize the relevance of our metrics and limit threats to validity by comparing predictors built with our metrics against predictors built with metrics detailed in another previous work[4] [ZPZ07].

4.1 Objects

We selected Rhino, ArgoUML, and Eclipse as systems for our case study because: (i) several releases of these systems are available, (ii) these systems were previously used in other case studies [EZS+08, ZPZ07] and, (iii) defect data are available from previous authors [EZS+08, ZPZ07] for Rhino and Eclipse or from a customized Bugzilla repository for ArgoUML. Table 1 provides summary data about releases and defects for the three systems.

Rhino[5], the smallest system, is a JavaScript/ECMAScript interpreter and compiler that implements the ECMAScript international standard, ECMA-262 v3 [ECM07]. We downloaded Rhino releases between 1.4R3 to 1.6R5 from the Rhino Web site. We used only 7 releases, those for which the total number of defects is greater than ten, from 1.5R1 to 1.6R1[6]. Thus, in comparison to our previous work, we exclude all the releases greater than 1.6R1 because they either do not include enough defects or have already been studied in our previous work [KRAG09]. We could have aggregated and assigned the defects of several releases to a single release, such as in previous work [OWB05,

[4] The metric values are available on-line at `http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/`.

[5] `http://www.mozilla.org/rhino/`

[6] Rhino1.4R3 is excluded since it is the initial release

KRAG09]. However, this assignment is possible only *post mortem*, *i.e.*, once defects are known, and is thus not representative of the day-to-day use of predictors. Therefore, we decided not to aggregate defects and to skip releases where defects impacted very few classes because, in these releases, a constant predictor would be difficult to outperform.

ArgoUML is a UML CASE tool to design and reverse-engineer various kinds of UML diagrams. It is also able to generate source code from diagrams to ease the development of systems. ArgoUML is written in Java. We use all pre-built releases available on ArgoUML Web site[7] except ArgoUML0.10.1, the initial release. We extract defect data from the ArgoUML customized Bugzilla repository, *i.e.*, we use the bug-tracking issues identified by the special tag "DEFECT". We then match the bug IDs of the bug tracking issues with the SVN commit messages, as retrieved from the ArgoUML SVN server. Once the file release matching the bug ID is retrieved, we perform a context diff with the previous file release to assign the defect to the appropriate class.

Eclipse[8] is a large, open-source, IDE. It is a platform used both in the open-source community and in industry, for example as a base for the WebSphere family of development environments. Eclipse is mostly written in Java, with C/C++ code used mainly for the widget toolkit. C++ code was not considered in this study. We used releases 1.0, 2.0.0, 2.0.1, 2.0.2, 2.1, 2.1.1, 2.1.2, 2.1.3 and 3.0. to compute the DEM. Defect and metrics data made available by previous authors [ZPZ07] pertain to releases 2.0, 2.1, and 3.0. We retained in our study only the sub-set of classes whose name and path perfectly match those of the files in the Z&Z dataset, which include more than 95% of the original files and defects.

We recovered the class diagrams of the releases of the systems using the Ptidej tool suite and its PADL meta-model. PADL is a language-independent meta-model to describe the static part and part of the behavior of object-oriented systems similarly to UML class diagrams [GA07]. It includes a Java parser and a dedicated graph exporter.

4.2 Treatments

The treatments of our study are predictors for defects in a system. We build these predictors using logistic and Poisson regressions built with different sets of metrics:

1. **C&K** are the metrics defined by Chidamber and Kemerer [CK94]. The C&K metrics are Response For a Class (RFC), Lack of COhesion on Methods (LCOM), Coupling Between Objects (CBO), Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC) and Line Of Code (LOC). WMC is defined as the sum of methods complexity. We define LCOM following C&K, thus it cannot be negative [BDW98]. We also define LCOM2 and LCOM5 following Briand *et al.* [BDW98] and complete the set with the number of attributes (nBAtt) and number of methods (nbMet). Thus, this metric set has a cardinality of 11 and is a super-set of the set of metrics used in previous work (*e.g.*, [BMW02, GFS05]).
2. **Z&Z** includes the complexity metrics computed by Zimmermann *et al.* in their study of Eclipse [ZPZ07]. We use this set when studying Eclipse by reusing metrics and defect data provided on-line[5] by the authors. We chose this metric set to prevent bias in the computation and analysis of the metric values.

[7] `http://argouml-downloads.tigris.org/`

[8] `http://www.eclipse.org/`

3. **DEM** is the set of basic design changes and account for the number of added, modified, and deleted attributes, methods, and relations in a class between its introduction in the system to the release under study. It comprises the following metrics $nbAddAtt$, $nbAddMet$, $nbAddRelOut$, $nbAddRelIn$, $nbDelAtt$, $nbDelMet$, $nbDelRelOut$, $nbDelRelIn$, $nbModMet$, $nbModRelOut$ and $nbModRelIn$.

Finally, we define two unions of the previous sets: **Z&Z+DEM** and **C&K+DEM** to study the benefits of our novel metrics when combined with traditional metrics.

4.3 Research Questions

We aim at answering the following four research questions:

- **RQ1 – Metrics Relevance:** To answer the general research question presented above, a preliminary study must be performed to give us confidence that the design evolution metrics indeed are useful to predict defective classes. RQ1 aims at providing evidence that a relation between the design evolution metrics and number of defects exists. Therefore, we sought to reject the following null-hypothesis: *A linear regression model built with **DEM**, **Z&Z+DEM**, or **C&K+DEM** does not better explain the number of defects discovered in classes with respect to the **Z&Z** or **C&K** sets.*
- **RQ2 – Defect-proneness Accuracy:** Often, developers and managers are interested to know whether a given class contains defects or not. Thus, a classification of a class into "defective" or "not-defective" may be enough to save the developers' and managers' efforts. Therefore, we sought to reject the following null-hypothesis: *A binary predictor built to identify defective classes with the **DEM**, **Z&Z+DEM**, or **C&K+DEM** sets does not perform better than a predictor built only with the **Z&Z** or **C&K** metric sets.*
- **RQ3 – Defect Count Prediction Accuracy:** An adequate testing of defect-prone classes would lead to more defects being removed from the system and, thus, it is interesting to know the possible number of defects in a class. We want to reject the following null-hypothesis: *A predictor of the number of defects in classes built with the **DEM**, **Z&Z+DEM**, or **C&K+DEM** sets, does not perform better than a predictor built only with the **Z&Z** or **C&K** metric sets.*
- **RQ4 – Defect Density Prediction Accuracy:** Finally, we establish the general usefulness of the DEM by comparing their ability to reduce effort needed to test code with defects; effort in terms of LOCs to analyze. Therefore, we sought to reject the following null-hypothesis: *A predictor of defect density in classes built with the **DEM**, **Z&Z+DEM**, or **C&K+DEM** sets, does not perform better than a predictor built only with the **Z&Z** or **C&K** metric sets.*

4.4 Analysis Method

We perform the following analyses to answer the research questions:

- **RQ1 – Metrics Relevance:** We build multi-dimensional linear regression models for each release of the systems, using the number of defects reported for a class as dependent variable and the different sets of metrics as independent variables.

For each set of metrics, we apply backward elimination to select a first set of relevant metrics. If the DEM are important to explain defects in classes, then they should be kept as explanatory variables – even when mixed with other metrics – and increase the proportion of variability accounted for than if one uses only the C&K and Z&Z metrics.

We consider that a metric significantly contributes to explain the dependent variable if it is included in at least 75% of the built models with a p-value of 0.05 or smaller, *i.e.*, the metric must contribute to the modeling of defective classes in at least 75% of the releases. This choice was inspired by model built for disease prediction [HL00].

The DEM should also improve the models and their adjusted R^2. An Adjusted R^2 expresses the proportion of variability in a data set that is accounted for by a statistical model and adjusted for the number of terms in a model. A Wilcoxon test was applied to assess statistical significance of adjusted R^2 improvement.

At standard significance levels (*i.e.*, 5% and 10%), intercepts were never significantly different from zero; thus we force regression models built to answer **RQ1** to pass through the origin.

– **RQ2 – Defect-Proneness Prediction Accuracy:** To answer **RQ2**, we apply logistic regression. Logistic regression models were previously used to predict if a class is defective or not, in our previous work and by other researchers, for example [GFS05].

In a logistic regression-based predictor, the dependent variable is commonly a dichotomous variable and, thus, it assumes only two values $\{0, 1\}$, *i.e.*, defect-free and defective. The multivariate logistic regression predictor is based on the formula:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{C_0+C_1\cdot X_1+\cdots+C_n\cdot X_n}}{1+e^{C_0+C_1\cdot X_1+\cdots+C_n\cdot X_n}} \tag{2}$$

where X_i are the characteristics describing the modeled phenomenon, C_0 is the *intercept*, C_i $(i = 1..n)$ is the *regression coefficient* of X_i [9], and $0 \leq \pi \leq 1$ is a value on the logistic regression curve. In our study, variable X_i will be metrics quantifying structural or evolution properties. The closer $\pi(X_1, X_2, \ldots, X_n)$ is to 1, the higher is the probability that the class contains defects.

– **RQ3 – Defect Count Prediction Accuracy:** We apply Poisson regression to predict the location and numbers of defects in the classes of a system. Poisson regression is a well-known technique for modeling counts. It has already been used in the context of defect prediction by [Eva97].

In a Poisson regression-based predictor, the dependent variable is commonly a count with no upper bound; the probability of observing a specific count, y, is given by the formula:

$$P(Y = y) = \frac{\lambda^y e^{-y}}{y!} \tag{3}$$

where λ is known as the population rate parameter and represents the expected value of Y. In the general case, λ is expressed in log-linear form as:

$$log(\lambda(X_1, X_2, ..., X_p)) = a + b_1X_1 + b_2X_2 + \cdots + b_pX_p \tag{4}$$

[9] The bigger $|C_i|$, the more X_i impacts the outcome. In particular, if $C_i > 0$, the probability of the outcome increases with the value of X_i.

where X_i are the characteristics describing the modeled phenomenon. In our study, variable X_i will be metrics quantifying structural or evolution properties.

– **RQ4 – Defect Density Prediction Accuracy:** We investigate the usefulness of the DEM to predict defect density rather than numbers of defects. To that end, Poisson regression models are trained and tested for defect density, *i.e.*, the number of defects divided by the number of LOCs.

To answer **RQ2**, **RQ3**, and **RQ4**, and consistently with sound industrial practices, as reported in [OWB05], results are organized as ranked lists of classes recommended for testing.

All statistic computations were performed with the R[10] programming environment.

4.5 Building and Assessing Predictors

We focus on inter-release prediction because such prediction is the most interesting with respect to practitioners and researchers: using data from a release to identify defective classes in a subsequent release.

Models are trained with the sets of metrics on a release i and used to predict a defect measure (probability, number, and density) for classes in the subsequent release $i+1$. A step-wise backward elimination is applied using the whole set of metrics on a release i and the best[11] model returned is tested on the subsequent release $i+1$. Backward elimination starts with a model including all independent variables and creates new models with fewer variables by removing one variable at the time and by penalizing models with a low likelihood and containing many parameters.

For each system and research question (**RQ2**, **RQ3**, and **RQ4**), we report for each set of metrics, the metrics present in at least 75% of the best models, *i.e.*, those effectively used for the predictions.

Our logistic regression model (for **RQ2**) assigns a probability of being defective to each class in a system while our Poisson regression-models assign a predicted number of defects (for **RQ3**) or defect density (for **RQ4**). Rather than trying to devise an optimal threshold above which the classes should be recommended, we rank classes [OWB05] according to their predicted probability of being defective (for **RQ2**), their predicted number of defects (for **RQ3**), and their predicted defect density (for **RQ4**).

Predictors are built with the different sets of metrics and we use results obtained with different cut points to compare different models. For **RQ2** and **RQ3**, we consider the classes in the top 10%, 20% and 30% defect-prone classes. For defect density (**RQ4**), the number of LOCs is the relevant measure. In a nutshell, we study the numbers of defects per LOCs, and we cumulatively partition the results to obtain the top-ranked classes containing 10% 20% and 30% of the LOCs of the system.

For **RQ2**, we use the F-measure that is the geometric mean between precision and recall to assess the performance of the models. The F-measure is defined as:

$$F = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5}$$

where precision is defined as the ratio between retrieved defective classes over retrieved classes and recall as the ratio between retrieved defective classes over all defective

[10] `http://cran.r-project.org/`

[11] We use Akaike's information criterion to elect the "best" model.

classes. An ideal model would obtain an F-measure value of 1 while real models usually trade precision for recall or vice versa. For **RQ3** and **RQ4**, we use the percentage of defects present in the top recommended classes as performance indices.

Note that the above performance indices are used in Section 5 to further specify the Research Questions. A special focus is also made there on the top 10%, top 20%, top 30% classes (or LOCs) as we believe a tester or manager will not likely go beyond those top sets of classes.

For each system, predictions are made for every release and we consider the average values of the performance indices. We also perform a *Wilcoxon signed rank test* to perform a comparison of different predictors and assess whether or not our metrics induce statistically significant improvement over a random predictor[12] or a predictor built without the DEM. We then compute the *Cohen-d statistics*[13] to obtain a statistically-reliable effect size of our metrics. The Cohen standardized difference between two groups [Coh88] is defined as the difference between the means (M_1 and M_2) divided by the pooled standard deviation (σ_p) of both groups: $d = (M_1 - M_2)/\sigma_p$. A Cohen-d inferior to 0.2 is perceived as a very small or trivial effect; a value between 0.2 and 0.5 is considered to represent a small effect; a value between 0.5 and 0.8 is deemed a medium effect, and a value of more than 0.8 provides evidence of a large effect [Coh88].

Given the small sample size (2 inter-release predictions) of Eclipse, we could not apply to the results from this system either the Wilcoxon tests or the Cohen-d statistics. Therefore statistical tests could not be conducted for the **Z&Z** set.

5 Study Results and Discussions

We now present and discuss the results of our case study.

5.1 RQ1 – Metrics Relevance

We answer **RQ1** by testing the following null-hypothesis: *DEM do not contribute to better explain the number of defects discovered in classes with respect to **Z&Z** or **C&K** metric sets.* We use this preliminary analysis to verify that the **DEM** correlate with the number of defects in classes, *i.e.*, that these metrics bring are relevant wrt. defects.

5.1.1 Most Used Metrics

Following the elimination procedure, different independent variables (metrics) were retained depending on the system and its releases. Those variations were expected and are due to several factors, including the system size in a release, its evolution history, the class diagram structure, and design stability.

Table 2 shows the metrics kept in the models built with the different sets of metrics. For each system, metrics from the **DEM** are kept as relevant to explain the number of defects per classes, even when they are added to the **C&K** and **Z&Z** sets.

[12] We consider that a random prediction model would give in average X% of the defective classes or the defects in any X% partition of the system

[13] We compute the Cohen-d statistics using pooled standard deviation.

	TM	DEM	TM+DEM
Rhino	RFC, CBO, LOC, LCOM2, LCOM1	nbDelAtt, nbDelRelOut, nbAddMet, nbAddRelOut, nbModMet, nbModRelOut,	RFC, CBO, LOC, LCOM1, nbAtt, LCOM2, nbMet, nbModMet, nbDelRelOut, nbAddRelOut, nbDelAtt, nbDelMet, nbAddMet, nbAddRelIn, nbModRelOut
Argo	RFC, LCOM2, LOC, WMC, DIT	nbAddMet, nbAddAtt, nbAddRelOut, nbModRelIn, nbDelRelOut, nbModMet, nbAddRelIn	RFC, LOC, LCOM1, LCOM2, DIT, WMC, nbDelRelOut, nbAddMet, nbModRelIn, nbAddRelOut, nbModRelOut
Eclipse	FOUT(max,sum), MLOC(max,sum,avg), NBD(sum), NOF(avg,max), NOM(avg,max,sum), NOT, TLOC, NSF(avg,sum), NSM(avg,max), PAR(sum), VG(avg,max,sum)	nbAddMet, nbAddRelOut, nbModRelOut, nbAddAtt, nbDelMet, nbModMet, nbAddRelIn, nbModRelIn, nbDelRelOut	FOUT(max,sum), TLOC, MLOC(avg,max), NBD(sum), NOF(avg,max), NOI, NOM(avg,max,sum), NSF(avg,sum), NSM(avg,max), PAR(avg,max,sum), VG(avg,max,sum), nbAddAtt, nbAddRelOut, nbAddMet, nbAddRelIn, nbDelAtt, nbDelRelIn, nbDelMet, nbDelRelOut, nbModMet, nbModRelOut

Table 2 RQ1: Metrics kept 75% (or more) times when building linear regression models to explain the number of defects—TM = C&K for Rhino and ArgoUML, TM = Z&Z for Eclipse

Rhino	C&K	DEM	C&K+DEM
1.5R1	0.3723	0.5169	0.6058
1.5R2	0.2925	0.5271	0.6063
1.5R3	0.6314	0.4468	0.711
1.5R4	0.6569	0.6437	0.7362
1.5R4.1	0.5632	0.6063	0.6619
1.5R5	0.6511	0.634	0.767
1.6R1	0.5246	0.6326	0.6608
Mean	0.5274	0.5725	0.6784
Std	0.1434	0.0759	0.0623
Median	0.5632	0.6063	0.6619

Table 3 Adjusted R^2 from linear regressions on Rhino

Some metrics are always kept: for the **C&K** set, RFC, LOC, and LCOM2 are present as significant metrics for both Rhino and ArgoUML. The metrics in **DEM** consistently kept are the number of added or modified methods and number of additions, deletions, and modifications of outgoing relations. The **Z&Z** set contain many relevant metrics, such as TLOC (the total LOCs) and FOUT (fan-out).

5.1.2 Proportion of variability explained

Tables 3, 4, and 5 show the values of adjusted R^2 for the regression models built using the various sets.

For Rhino, see Table 3, all sets of metrics mostly give an adjusted R^2 superior to 0.5. The most effective model uses the **C&K+DEM** set and has an average of 0.6784, contrasting with the adjusted R^2 of 0.5274 of the **C&K** model: adding **DEM** to **C&K** provides a gain of 0.1510. The **DEM** model, with an adjusted R^2 of 0.5725, outperforms the **C&K** model by 0.0451. A Wilcoxon test rejects with a p-value of

Argo	C&K	DEM	C&K+DEM
0.12	0.1292	0.0794	0.1479
0.14	0.4454	0.2206	0.4875
0.16	0.2873	0.2654	0.3248
0.18.1	0.3028	0.2608	0.342
0.20	0.2529	0.1572	0.2597
0.22	0.1627	0.2221	0.2794
0.24	0.2379	0.1529	0.2924
0.26	0.0433	0.0562	0.0705
0.26.2	0.1623	0.1279	0.1852
Mean	0.2249	0.1714	0.2655
Std	0.117	0.0758	0.1214
Median	0.2379	0.1572	0.2794

Table 4 Adjusted R^2 from linear regressions on ArgoUML

Eclipse	Z&Z	DEM	Z&Z+DEM
2.0	0.2962	0.1378	0.3136
2.1	0.2236	0.1642	0.2545
3.0	0.3141	0.195	0.3416
Mean	0.2766	0.1657	0.3032

Table 5 Adjusted R^2 from linear regressions on Eclipse

0.007813 the following null hypothesis *The **C&K+DEM** set does not provide a better adjusted R^2 with respect to the **C&K** set.*

For ArgoUML, see Table 4, the values of R^2 are substantially lower than for Rhino. Differently from Rhino, the **DEM** model is now, in average, 0.0535 lower than the **C&K** model. However, the best model remains the **C&K+DEM** model with an average of 0.2655, improving the **C&K** model by 0.0406. The low means are due to some releases, such as ArgoUML 0.26, for which the maximal adjusted R^2 obtained was only 0.0705 because there are only 25 bugs in 1,628 classes. Similarly to Rhino, a Wilcoxon test rejects the following null-hypothesis: *The **C&K+DEM** set does not provide a better adjusted R^2 with respect to the **C&K** set.* with a p-value of 0.001953.

Linear regression models built on Eclipse, see Table 5, provide adjusted R^2 of at most 0.3416 (for Eclipse 3.0 and with the **C&K+DEM** model). Except for the values being higher than those for ArgoUML, the model using the mixed set **Z&Z+DEM** outperforms the models built with the **Z&Z** and **DEM** sets. The size of the **Z&Z** set, with 31 metrics, could explain in part the clear advantage it has over **DEM** set, which includes only 11 metrics.

As a conclusion, the **DEM** set improves the adjusted R^2 of any model built with the **C&K** set or **Z&Z**. For Rhino, it even outperforms the **C&K** set.

We can thus answer **RQ1** affirmatively and conclude that on Rhino, ArgoUML, and Eclipse, the design evolution metrics actually correlate with the numbers of defects and would help in explaining the number of defects in a class.

5.2 RQ2 – Defect-proneness Accuracy

To answer **RQ2**, we rank the classes of a system according to their predicted probability of being defective, given by a logistic regression model. Then, we select the top-ranked

	TM	DEM	TM+DEM
Rhino	LCOM1, LCOM2, CBO	nbAddMet, nbModMet, nbAddRelIn, nbDelRelOut, nbModRelOut	LCOM1, LCOM2, NOC, nbModRelOut, nbAddMet, nbAddAtt, nbDelRelOut
Argo	RFC, DIT, LCOM5, LOC	nbAddMet, nbDelRelOut, nbAddAtt, nbDelMet	RFC, WMC, CBO, DIT, nbAddAtt, nbAddRelOut
Eclipse	TLOC, FOUT(avg,max), NBD(max,sum), NSF(max,sum), PAR(avg,max), VG(max,sum)	nbAddAtt, nbAddMet, nbDelRelOut, nbModRelOut	TLOC, FOUT(avg), NBD(max,sum), NOF(max), NOI, NOT, NSF(max,sum), PAR(avg,max), nbAddAtt, nbModMet, nbModRelOut

Table 6 RQ2: Metrics kept 75% (or more) times when building logistic regression models to predict defective classes—TM = C&K for Rhino and ArgoUML, TM = Z&Z for Eclipse

classes and tag those classes as likely to be defective. We report in the following the most used metrics in the models and the results obtained.

5.2.1 Most Used Metrics

Table 6 reports the most frequently retained metrics in predictors of **RQ2**, after the backward elimination procedure used in the training phase. We can observe that metrics such as the numbers of added attributes and methods (nbAddAtt, nbAddMet) and that of modified outgoing relations (nbModRelOut) were almost always used in all systems and for both the **DEM** set and mixed set (**C&K+DEM** or **Z&Z+DEM**).

5.2.2 Analysis of the Obtained Means

Figures 2, 3, and 4 report the average F-measure in the top ranked classes. As shown in the figures, the **C&K+DEM** model is consistently better than the **C&K** model. On Rhino, the improvement is roughly of 4 points on average for the top 10% and 20% top ranked classes and 8 points for the top 30% classes. The improvement is on average less important for ArgoUML (about 2%) and Eclipse (about 1%). The same remark applies when considering the medians: the improvement is about 5 points for Rhino and 2 for ArgoUML.

5.2.3 Wilcoxon Tests

We performed a Wilcoxon paired test to check whether predictors built with our metrics are indeed improving the F-measure when compared to predictors built only with the **C&K** set. The null hypothesis tested is *the F-measure of a predictor built with C&K+DEM is not greater than a predictor built with C&K metrics when the top 10%, 20%, 30% classes are selected.*

For both Rhino and ArgoUML, we were able to reject the null-hypothesis, as shown by the p-values reported in Table 7. Considering that a random ranking should have an average of X% of defective classes within the top X% classes, we also perform a Wilcoxon test and confirm that the **C&K+DEM** model is substantially better than the **random** model (see Table 8).

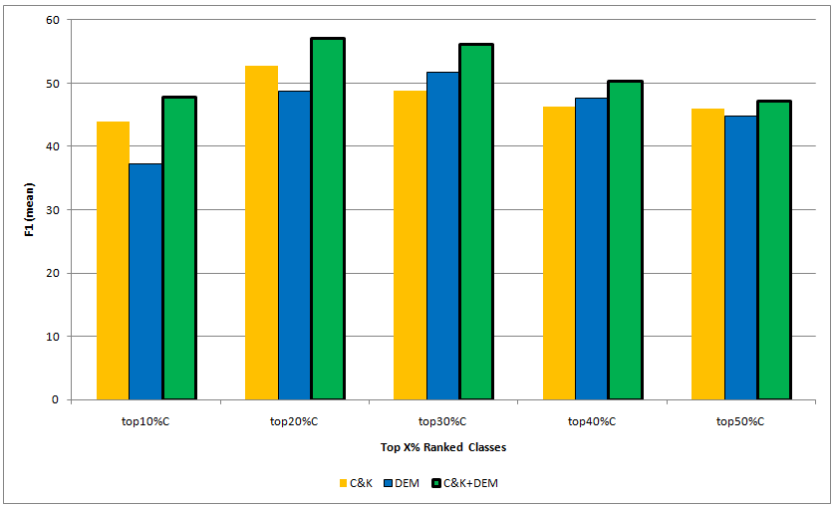

Fig. 2 Average F-measure for defective classes on Rhino per top classes

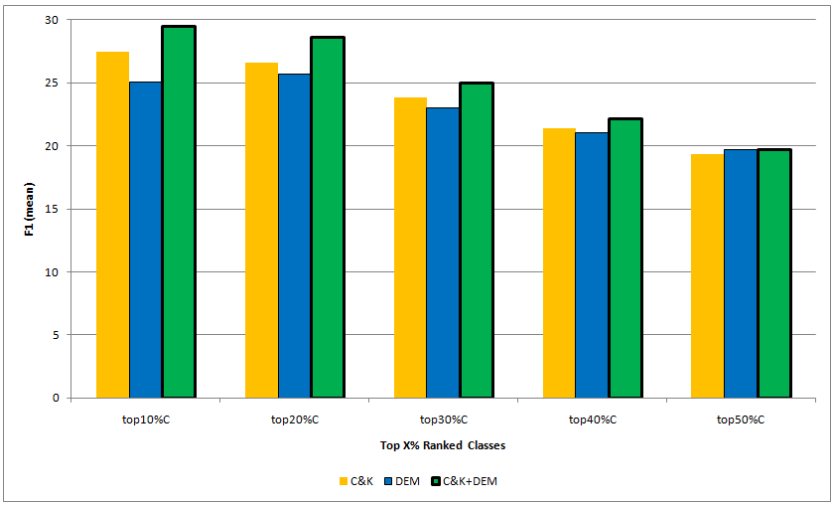

Fig. 3 Average F-measure for defective classes on ArgoUML per top classes

5.2.4 Cohen-d Statistics

To further assess the improvement of F-measure brought by the **DEM**, we also compute the Cohen-d statistics to quantify the effect size of using **DEM** in building predictors with respect to **C&K** metrics or a random ranking. Results are reported in Tables 9 and 10.

In summary, when comparing **C&K+DEM** to **C&K**, for Rhino, we have a large effect on the top 20% classes, a medium effect on the top 30% classes and a small effect on the top 10%; for ArgoUML, there is only a small effect (on the top 10% and top 20% classes) and a very small effect on the top 30%.

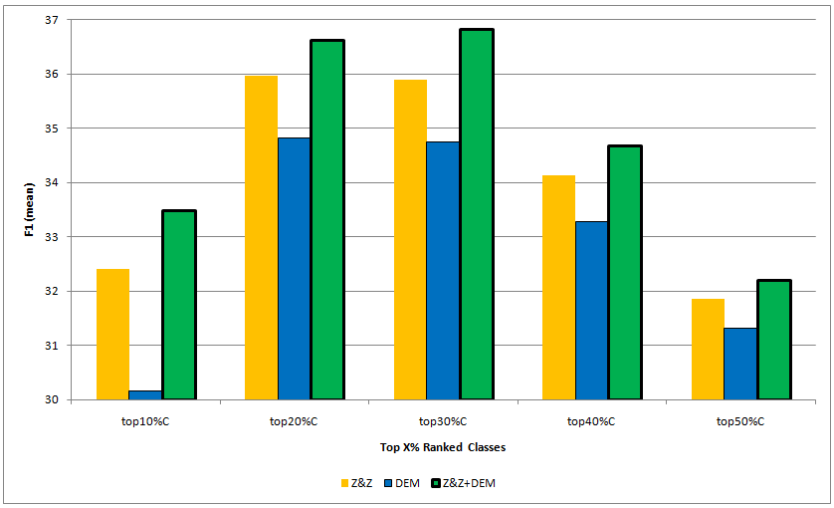

Fig. 4 Average F-measure for defective classes on Eclipse per top classes

	Top 10%	Top 20%	Top 30%
Rhino	0.05017	0.05017	0.05017
ArgoUML	0.01125	0.003906	0.03796

Table 7 C&K+DEM $\leq$ C&K? p-value of Wilcoxon signed rank test for the F-measure of defective classes (confidence level: light grey 90%, dark grey 95%)

	Top 10%	Top 20%	Top 30%
Rhino	0.01563	0.01563	0.01563
ArgoUML	0.003906	0.003906	0.003906

Table 8 C&K+DEM $\leq$ random? p-value of Wilcoxon signed rank test for the F-measure of defective classes (confidence level: light grey 90%, dark grey 95%)

	Top 10%	Top 20%	Top 30%
Rhino	0.44	0.80	0.59
ArgoUML	0.22	0.22	0.13

Table 9 Assessing C&K+DEM improvement over C&K: Cohen-d statistics

The comparison with a random predictor displayed in Table 10 clearly demonstrates the superiority of a model using the **DEM** set.

	Top 10%	Top 20%	Top 30%
Rhino	5.07	6.78	3.12
ArgoUML	3.20	2.62	1.94

Table 10 Assessing C&K+DEM improvement over random: Cohen-d statistics

Overall, the reported means and statistical tests support that our design evolution metrics are useful for predicting defective classes and we can claim statistical significance of the observed improvement yet with a small effect size.

	TM	DEM	TM+DEM
Rhino	LCOM1, LCOM2, nbMet, CBO	nbDelAtt, nbModMet, nbAddMet, nbModRelOut	RFC, LOC, nbAtt, nbMet, LCOM1, LCOM2, nbModMet, nbDelRelOut, nbAddRelOut
ArgoUML	LOC, RFC, LCOM1, nbMet	nbModRelOut, nbAddAtt, nbAddMet, nbDelRelOut	RFC, LCOM1, CBO, LCOM2, WMC, DIT, nbModRelOut, nbDelRelIn
Eclipse	FOUT(avg,sum), MLOC(sum), NBD(max,sum), NOM (avg), NSF(sum), NSM(avg), PAR(avg,max), TLOC, VG(max,sum)	nbDelMet, nbAddMet, nbAddRelIn, nbDelRelOut, nbAddRelOut	FOUT (avg,sum), MLOC(avg,sum), NBD(max,sum), NOF(sum), NOI, NOT, NSF(sum), NSM(avg), PAR(avg,max), TLOC, VG(max,sum), nbModMet, nbDelRelIn, nbAddRelIn, nbAddRelOut, nbModRelOut

Table 11 RQ3: Metrics kept 75% (or more) times when building Poisson regression models to predict the number of defects—TM = C&K for Rhino and ArgoUML, TM = Z&Z for Eclipse

5.3 RQ3 – Defect count prediction

To answer **RQ3**, we first rank the classes of a system according to their predicted number of defects, given by a Poisson regression model. Then, we select the top X% classes and assess the percentage of defects contained within the selection. We report in the following the most used metrics (kept after the elimination procedure) in the models and the results obtained.

5.3.1 Most Used Metrics

The metrics kept most of the time are reported in Table 11. The number of modified outgoing relations (nbModRelOut) is the single most used metric for the **DEM** and **C&K+DEM** sets.

5.3.2 Analysis of the Obtained Means

Figures 5, 6, and 7 report the mean of the percentages of defects contained in the top X% ranked classes. The **C&K+DEM** model is consistently better than the **C&K** model. On Rhino, the improvement is roughly of 6% on average from the top 10% to 30% ranked classes. The improvement is less important for ArgoUML (2% to 3%) and Eclipse (2%). Looking at the medians, the improvement due to the *DEM* metrics seem to increase with the cardinality of the set of classes considered. The improvement brought by the mixed model is quite important for the top 30 % classes (in particular more than 5 % for Rhino) but mostly small for the top 10 % and top 20 % classes (in particular less than 1 % for the top 20 % classes of Rhino).

5.3.3 Wilcoxon Tests

We perform a Wilcoxon paired test to check whether our metrics are indeed improving over **C&K** set. The null hypothesis tested is *the percentage of defects of a predictor built with C&K+DEM is not greater than that of a predictor built with C&K metrics when the top 10%, 20%, 30% classes are selected.*

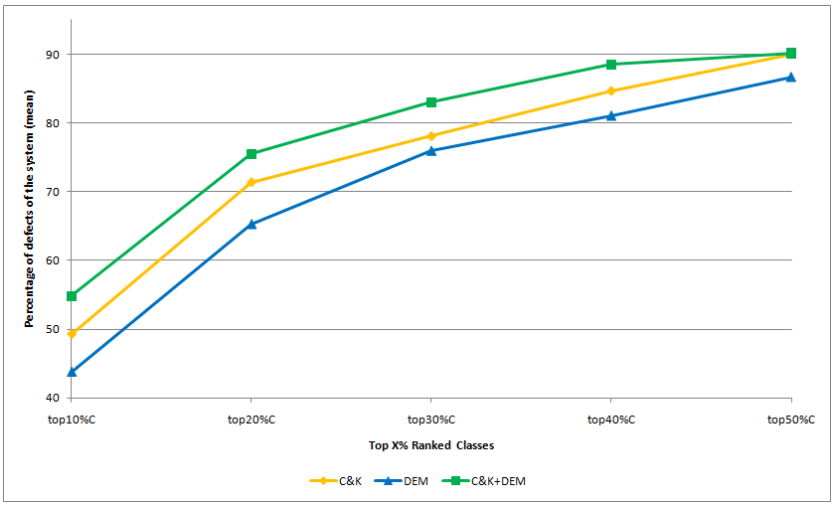

Fig. 5 Average Percentage of defects on Rhino per top classes

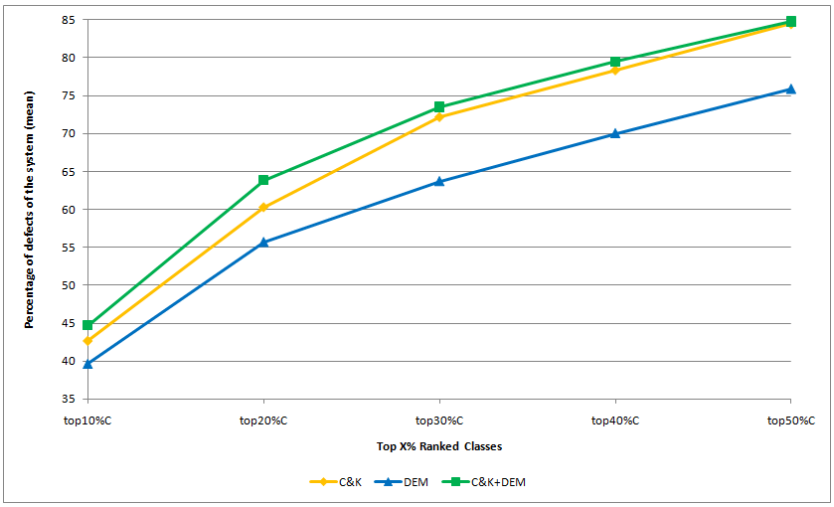

Fig. 6 Average Percentage of defects on ArgoUML per top classes

For both Rhino and ArgoUML, considering the best model, *i.e.*, **C&K+DEM** and as shown by the p-values reported in Table 12, we were able to reject the null-hypothesis - though at a 90% confidence level for some partitions. Considering that a random ranking should have an average of X% of defects within the top X% classes, we also performed a Wilcoxon test to verify that the **C&K+DEM** model is substantially better than the **C&K** model (see Table 13).

5.3.4 Cohen-d Statistics

To assess the size in the improvement of percentage of defects in the top ranked classes, we also computed the Cohen-d statistics to quantify the effect size of using **DEM** in

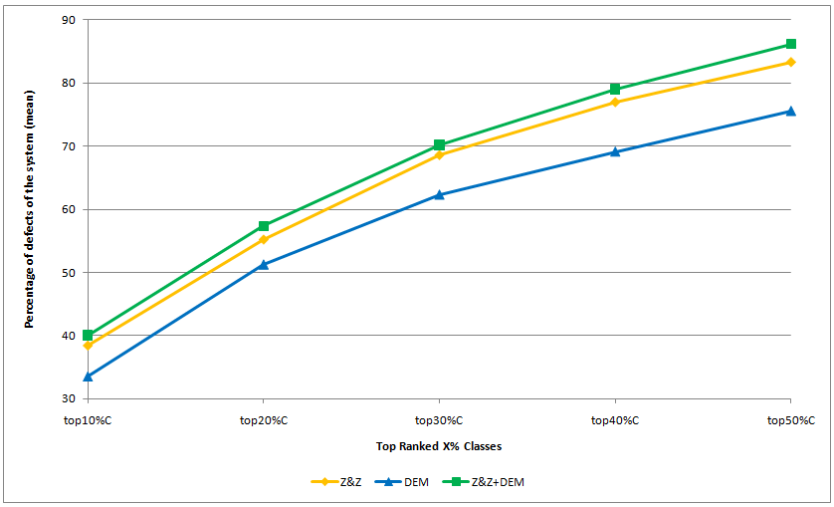

Fig. 7 Average Percentage of defects on Eclipse per top classes

	Top 10%	Top 20%	Top 30%
Rhino	0.03125	0.05017	0.08876
ArgoUML	0.01802	0.02596	0.09766

Table 12 C&K+DEM $\leq$ C&K? p-value of Wilcoxon signed rank test for the percentage of defects per top classes (confidence level: light grey 90%, dark grey 95%)

	Top 10%	Top 20%	Top 30%
Rhino	0.01563	0.01563	0.01563
ArgoUML	0.003906	0.003906	0.003906

Table 13 C&K+DEM $\leq$ random? p-value of Wilcoxon signed rank test for the percentage of defects per top classes (confidence level: light grey 90%, dark grey 95%)

building predictors with respect to **C&K** metrics or a random ranking. Results are reported in Tables 14 and 15.

In summary, when comparing **C&K+DEM** to **C&K**, for Rhino, we have a large effect on the top 10% and top 30% classes and medium effect on the top 20%; for ArgoUML, there is only a small effect on the top 20% classes and a very small effect on the rest. The comparison with a random predictor shows the clear superiority of a model built using our evolution metrics.

	Top 10%	Top 20%	Top 30%
Rhino	0.88	0.58	0.91
ArgoUML	0.18	0.29	0.12

Table 14 Assessing C&K+DEM improvement over C&K: Cohen-d statistics

Overall, the reported means and statistical tests support our conjecture that our evolution metrics are useful for predicting the number of defects. In addition, we can

	Top 10%	Top 20%	Top 30%
Rhino	10.63	8.44	12.05
ArgoUML	4.21	5.54	5.51

Table 15 Assessing C&K+DEM improvement over random: Cohen-d statistics

	TM	DEM	TM+DEM
Rhino	LCOM1, LCOM2, CBO, DIT, WMC	nbAddMet, nbModRelOut, nbAddAtt, nbDelRelIn, nbAddRelIn, nbModRelIn, nbAddRelOut	LOC, nbMet, nbAddMet, nbModMet, nbDelRelIn, nbModRelOut
ArgoUML	LCOM1, LCOM2, DIT, RFC, WMC	nbAddAtt, nbDelMet, nbAddMet, nbModMet, nbDelRelOut, nbModRelOut	LOC, LCOM1, nbMet, DIT, nbAddAtt, nbDelRelOut, nbDelMet, nbAddRelIn, nbModRelOut
Eclipse	NBD(avg,max), NOM (avg), PAR(max)	nbAddAtt, nbAddMet, nbModRelIn	TLOC, NOI, NOF(avg), NBD(avg,max), nbAddAtt, nbAddMet, nbModRelOut

Table 16 RQ4: Metrics kept 75% (or more) times when building Poisson regression models with different metric sets—TM = C&K for Rhino and ArgoUML, TM = Z&Z for Eclipse

claim statistical significance of the observed improvement on all systems and a large effect on Rhino.

5.4 RQ4 – Defect Density Prediction

To answer **RQ4**, we test whether, given the same volume of recommended code, our metrics provide a higher percentage of defects than traditional metrics. We use Poisson regression to assess the predictive accuracy for defect density of models built with the **DEM** and other metrics sets. We first rank the classes of a system according to their predicted defect density; then, we cut this list by selecting the classes containing the top X% LOCs and assess the percentage of defects contained within the selection. We report in the following the most used metrics (kept after the elimination procedure) in the models and the results obtained.

5.4.1 Most Used Metrics

Table 16 reports the metrics that were the most used in the prediction, *i.e.*, those kept after the elimination procedure. We observe that the number of added attributes and methods and the number of modified outgoing relations (nbAddAtt, nbAddMet, nbModRelOut) are again the most frequently kept by the elimination procedure.

5.4.2 Analysis of the Obtained Means

Figures 8, 9, and 10 report the average percentage of defects contained in the top LOCs. They show that, for all systems, the models built with **DEM** are clearly superior to the ones built with only **C&K** or **Z&Z**.

For Rhino, we have on average roughly 7% more defects with the top 10% (from 10% to 17%), 6% more defects with the top 20% LOCs (from 27% to 33%) and 10% more defects for the top 30% LOCs (from 40% to 50%). For ArgoUML, the difference is, in average, roughly 3% more defects with the top 10% LOCs (from 14% to 17%),

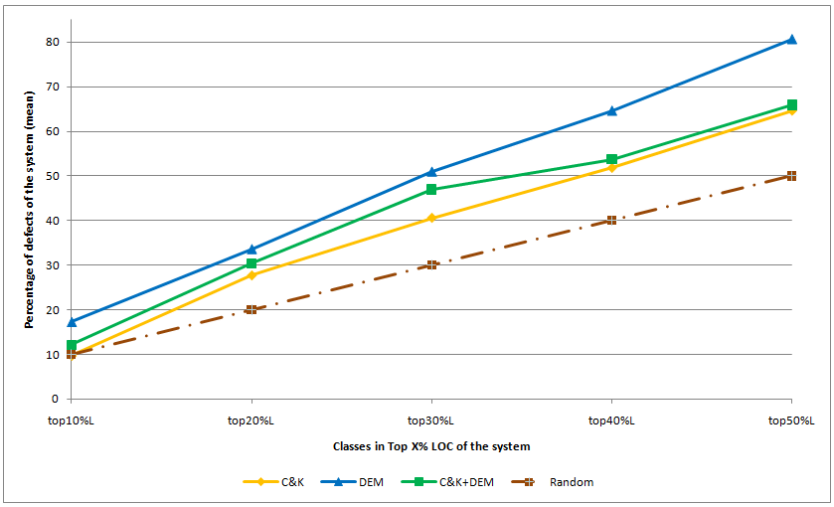

Fig. 8 Average Percentage of defects on Rhino per top LOCs

10% more defects with the top 20% LOCs (from 20% to 30%), and 13% more defects for the top 30% LOCs (from 32% to 45%). With the two predictions for Eclipse, we have on average 6% more defects (from 9% to 15%) with the top 10% LOCs, 9% more defects (from 18% to 27%) with the top 20% LOCs, and 8% more defects (from 29% to 37%) with the top 30% LOCs. On the medians, the **DEM** model improves over the **C&K** model by 8 to 11 % for Rhino and by 4 to 8 % for ArgoUML. In summary, for all systems, there is a substantial gain when models are built with only the **DEM** set.

Note that on average, the mixed set performs worse than the **DEM** set for all the systems but better than the **C&K** or **Z&Z** set. It appears that adding the traditional metrics degrades the predictive power of the **DEM** set. This is not rare in a prediction context as overfitting can cause occurrences of a set performing much worse than one of its subsets.

5.4.3 Wilcoxon Tests

We perform a Wilcoxon paired test to check whether our metrics are indeed improving over **C&K**. The null hypothesis tested is *the percentage of defects of a predictor built with C&K+DEM is not greater than that of a predictor built with C&K metrics when the top classes containing from 10% to 30% LOCs of the system are selected.*

For both Rhino and ArgoUML, considering the best model, *i.e.*, **DEM**, we were able to reject the null-hypothesis, as shown by the p-values reported in Table 17. Considering that a random ranking should have an average of X% of defects within the top X% LOCs, we also performed a Wilcoxon test and confirmed that the **DEM** model is substantially better than a random predictor (see Table 18).

5.4.4 Cohen-d Statistics

To assess the size in the improvement of percentage of defects in the top LOCs, we also computed the Cohen-d statistics to quantify the effect size of using the **DEM** model

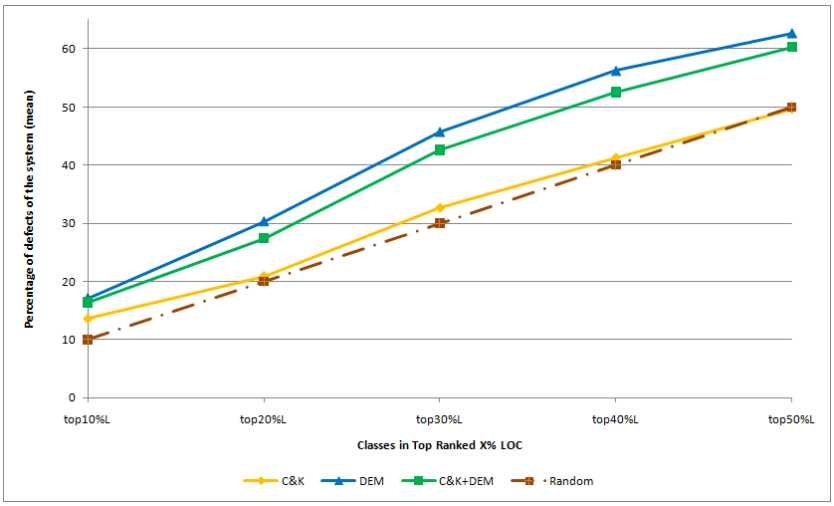

Fig. 9 Average Percentage of defects on ArgoUML per top LOCs

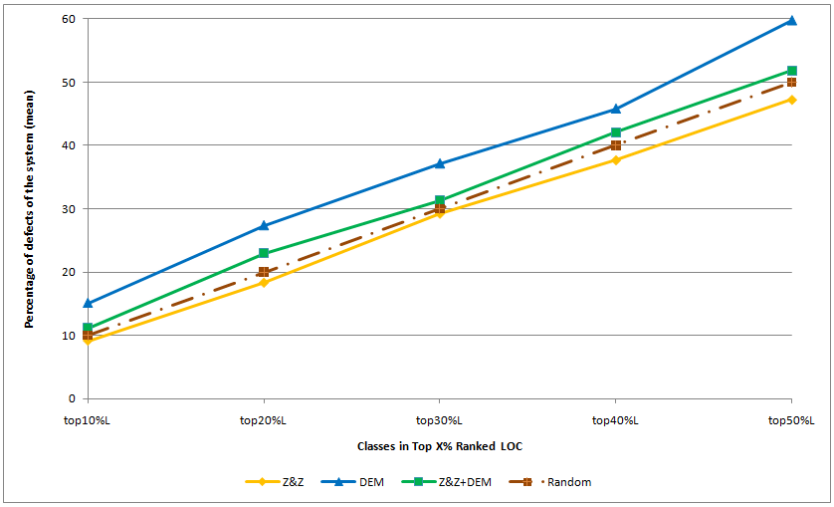

Fig. 10 Average Percentage of defects on Eclipse per top LOCs

with respect to **C&K** metrics or a random ranking. Results are reported in Tables 19 and 20.

In summary, when comparing **DEM** to **C&K** models, except for a medium effect for ArgoUML on the top 10% LOCs, we always observe a large effect for Rhino and ArgoUML. The comparison with a random predictor again demonstrates the clear superiority of our model.

Overall, the reported means as well as the Wilcoxon tests and Cohen-d statistics provide evidence that our metrics increase the percentages of detected defects for a given size of code. Hence, they help managers save their developers' efforts by returning less LOCs to be analyzed to locate and correct a defect.

	Top 10%	Top 20%	Top 30%
Rhino	0.01563	0.07813	0.04688
ArgoUML	0.07422	0.003906	0.003906

Table 17 DEM $\leq$ C&K? p-value of Wilcoxon signed rank test for the percentage of defects per top LOCs (confidence level: light grey 90%, dark grey 95%)

	Top 10%	Top 20%	Top 30%
Rhino	0.01563	0.01563	0.01563
ArgoUML	0.003906	0.003906	0.007813

Table 18 DEM $\leq$ random? p-value of Wilcoxon signed rank test for the percentage of defects per top LOCs (confidence level: light grey 90%, dark grey 95%)

	Top 10%	Top 20%	Top 30%
Rhino	2.17	0.85	1.76
ArgoUML	0.53	1.05	1.08

Table 19 Assessing DEM improvement over C&K: Cohen-d statistics

	Top 10%	Top 20%	Top 30%
Rhino	3.60	4.05	6.78
ArgoUML	3.06	1.9	1.82

Table 20 Assessing DEM improvement over random: Cohen-d statistics

6 Threats to Validity

This work extends our previous work [KRAG09] in two ways: (i) the aggregate metrics first proposed were replaced by their simple basic constituents, which are simpler and more intuitive design evolution metrics, (ii) the case study now involves a comparison on several releases of three different software systems showing that the design evolution metrics are useful in identifying defective classes. Our purpose is not to investigate the formal properties of the DEM following the guidelines of measurement theory [FP97]. We believe that before any formal study of the properties of a metric, the metric itself must be shown useful. Thus, this work is a preliminary study which provides evidence that the DEM can help developers in saving effort by focusing quality assurance on defective classes.

Threats to *construct validity* concern the relation between the theory and the observation. This threat is mainly due to the use of incorrect defect classification or incorrect collected metrics values. In our study, we used material and defects manually classified and used by others [EZS+08, ZPZ07] and the independent issues stored in ArgoUML bug-tracker. We inspected several randomly-chosen ArgoUML issues and manually verified that they represented corrective maintenance requests in most of the cases. Releases of ArgoUML were found to contain relatively few defects but it's possible that defects are more than those we had access to. This may be due to the known fact that, in general, developers do not adequately document all the bugs they fix. Manual classification of defects for large Bugzilla repository is not feasible and thus a clear insight about how many defects were possibly missed cannot be proposed. We conjecture that more defect data should result in better performances of the built models.

Extraction of C&K metrics for Rhino and ArgoUML is performed with PADL, a tool already used in other experiments. Metrics values were manually assessed for a subset of the classes and compared with the previous used tool [KRAG09]. We chose to use PADL because it offers a richer set of metrics than the C&K metrics suite used in [KRAG09] and thus makes it more challenging to show the usefulness of the DEM. The Eclipse case study was performed using the metrics suite, values, and defect classification provided by Zimmermann *et al.* [ZPZ07]. Consequently, we believe that it is highly unlikely that the relation found between the theory and the observation is due to a statistical fluctuation.

Threats to *internal validity* concern any confounding factor that could influence our results. In particular, these threats can be due to subjectiveness during the manual building of oracles and to the bias introduced by manually classifying defects.

As reported by Antoniol *et al.* in [AMAP07], most of bug tracking entries are not related to corrective maintenance. We attempted to avoid any bias in the building of the oracle by adopting a classification made available by other researchers [EZS+08, ZPZ07] or documented in the independent ArgoUML bug-tracking system. For Rhino, the defect data results from a manual classification provided by Eaddy in [EZS+08]. As for ArgoUML, its bug tracking system has a field used to explicitly specify when an issue is a "defect". In our study, we selected only the entries ArgoUML developers tagged as "defect"; thus minimizing the risk that non defect issues are part of our dataset. Finally, as we replicated Zimmerman *et al.* [ZPZ07] study for comparison purposes, we reused their publicly available defect datasets. However their data, though about post-release defects, may contain some non defect entries.

Furthermore, Bird *et al.* [BBA+09] argue that defects documented by the developers are only a subset of all defects and are hardly representative of the whole set of defects in terms of important defect features, such as severity. They specifically claimed that the Eclipse data set by [ZPZ07] was only a sample of the actual defects but fortunately representative in terms of severity. Unfortunately, their own data sets were not made publicly available.

Another factor influencing results is the choice of the costs used in our ETGM algorithm. A complete study of the impact of costs is beyond the scope of this work and is documented in an earlier publication [KRG+10]. We used costs learned from that study and though we cannot claim that changing ETGM costs would not affect our results, we are confident that the chosen costs are appropriate for this study. The same costs were used on the various releases of the three systems. In addition, we manually inspected matched and non-matched classes and found an excellent agreement with the expected results.

Threats to *conclusion validity* concern the relationship between the treatment and the results. Proper tests were performed to statistically reject the null-hypotheses. In particular, non-parametric tests were used in place of parametric tests where the conditions necessary to use parametric tests do not hold. As an example, we selected the Wilcoxon test because it is very robust and sensitive [CPM+00].

Threats to *external validity* concern the possibility of generalizing our results. The study is limited to three systems: Rhino, ArgoUML, and Eclipse and a total of 19 releases on which we have defect data. Yet, our approach is applicable to any other OO system. Results are encouraging on the studied systems but more work is needed to verify if our approach is in general better than previously known fault location approaches. We cannot claim that similar results will be obtained with other systems. We have built different predictive models and cannot be sure that their relative perfor-

mances will remain the same on different systems or releases. On different systems or releases, the procedure of variable selection can lead to different models with different sets of variables. Nevertheless, the three systems correspond to different domains and applications, have different sizes, are developed by different teams, and have a different history. We believe this choice confirms the external validity of our case study.

7 Conclusion

Testing activities play a central role in quality assurance. Testing effort should be focused on defective classes to avoid wasting valuable resources. Unfortunately, identifying defective classes is a challenging and difficult task. In a previous paper, we used a search-based technique to define software metrics accounting for the role that a class plays in the evolution of the design of a system, described with class diagrams.

This work extends our previous work by comparing, on the one hand, the Chidamber and Kemerer's metrics suite and traditional complexity metrics (*e.g.*, fan-in, fan-out) with, on the other hand, a novel set of design evolution metrics, **DEM**, measuring basic design changes between releases of a system. To reinforce the empirical evidence of a relation between our evolution metrics and defects in classes, we extended our previous case study with several releases of Rhino, a Java ECMA script interpreter, ArgoUML, a Java UML CASE tool, and Eclipse, a Java development environment, to predict defective classes. We thus were able to address four research questions: **RQ1** on metrics relevance, **RQ2** on prediction of defective classes, **RQ3** on prediction of numbers of defects, and **RQ3** on prediction of defect density.

By means of multivariate linear models, we positively answered **RQ1**: the new metrics contribute to better explain the numbers of defects in the classes in Rhino, ArgoUML, and Eclipse. On the extended set of systems, we found that integrating the new metrics led to a significant improvement but with small effect size regarding the location of the defects, thus answering positively **RQ2**. Combining the **DEM** with traditional metrics led to a significant improvement with mostly medium to large effect size thus answering positively **RQ3**. Finally, the prediction for which the **DEM** were far better was about defect density, *i.e.*, when it comes to maximize the number of defects contained in a small share of the volume code in a system. We positively answered **RQ4** as the **DEM** consistently outperformed traditional OO and complexity metrics with a large effect size.

Our future work will be devoted to replicate this case study on different systems and deeply analyze the correlation and impact of each DEM on defect-proneness of classes of a system. We will also study finer-grain changes to attributes, visibility, and relations as well as the impact of using more than one previous release, possibly the entire system history, in building models. Finer-grain study includes also further investigating whether inheritance introduces defects, *e.g.*, when classes participating in hierarchies are defective.

Acknowledgment

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (Research Chairs in Software Evolution and in Software Patterns and Patterns of Software) and by G. Antoniol Individual Discovery Grant.

Data

All artifacts (releases, class diagrams, graph representations) used in this work can be downloaded from the SOCCER laboratory Web server, under the Software Evolution Repository (SER) page, accessible at `http://web.soccerlab.polymtl.ca/SER/`.

References

[AMAP07] Kamel Ayari, Peyman Meshkinfam, Giulio Antoniol, and Massimiliano Di Penta. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *IBM Centers for Advanced Studies Conference*, pages 215–228, Toronto CA, Oct 23-25 2007. ACM.

[BBA+09] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced? bias in bug-fix datasets. In *ESEC/SIGSOFT FSE*, pages 121–130, 2009.

[BBM96] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. on Software Engineering*, 22:751–761, 1996.

[BCR94] V. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[BDW98] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Softw. Eng.*, 3(1):65–117, 1998.

[BLW03] Lionel C. Briand, Yvan Labiche, and Yihong Wang. An investigation of graph-based class integration test order strategies. *IEEE Trans. on Software Engineering*, 29(7):594–607, 2003.

[BMW02] Lionel C. Briand, Walcelio L. Melo, and Jürgen Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. on Software Engineering*, 28(7):706–720, 2002.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. on Software Engineering*, 20(6):476–493, June 1994.

[Coh88] J. Cohen. *Statistical power analysis for the behavioral sciences (2nd ed.)*. Lawrence Earlbaum Associates, Hillsdale, NJ, 1988.

[CPM+00] Wohlin C., Runeson P., Host M., Ohlsson M.C., Regnell B., and Wesslen A. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.

[CS00] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Trans. on Software Engineering*, 26(8):786–796, August 2000.

[EBGR01] K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. on Software Engineering*, 27(7):630–650, July 2001.

[ECM07] ECMA. *ECMAScript Standard - ECMA-262 v3*. ISO/IEC 16262, 2007.

[Eva97] William M. Evanco. Poisson analyses of defects for small software components. *Journal of Systems and Software*, 38(1):27–35, 1997.

[EZS+08] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transaction on Software Engineering*, 34(4):497–515, 2008.

[FP97] N.E. Fenton and S.L Pfleeger. *Software Metrics: A Rigorous and Practical Approach (2nd Edition)*. Thomson Computer Press, Boston, 1997.

[GA07] Yann-Gaël Guéhéneuc and Giuliano Antoniol. Demima: A multi-layered framework for design pattern identification. *Trans. on Software Engineering*, December 2007.

[GFS05] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transaction on Software Engineering*, 31(10):897–910, 2005.

[GKMS00] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. on Software Engineering*, 26(7):653–661, 2000.

[Glo86] F. Glover. Future paths for integer programming and links to artificial intelligence. In *Computers and Operations Research*, pages 533–549, 1986.

[Has09] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *ICSE*, pages 78–88, 2009.

[HL00] David Hosmer and Stanley Lemeshow. *Applied Logistic Regression (2nd Edition)*. Wiley, 2000.

[KRAG09] Segla Kpodjedo, Filippo Ricca, Giuliano Antoniol, and Philippe Galinier. Evolution and search based metrics to improve defects prediction. *Search Based Software Engineering, International Symposium on*, 0:23–32, 2009.

[KRG+10] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Giuliano Antoniol, and Yann-Gael Gueheneuc. Studying software evolution of large object-oriented software systems using an etgm algorithm. *Journal of Software Maintenance and Evolution, http://dx.doi.org/10.1002/smr.519*, 2010.

[KRGA08] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol. Error correcting graph matching application to software evolution. In *Proc. of the Working Conference on Reverse Engineering*, pages 289–293, 2008.

[ME98] J. Munson and S. Elbaum. Code churn: a measure for estimating the impact of code change. In *Proceedings of the International Conference on Software Maintenance*, pages 24–31, 1998.

[MGM06] Jeff Offutt Mats Grindal and Jonas Mellin. On the testing maturity of software producing organizations. In *Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, pages 171–180, 2006.

[NB05] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 284–292, 2005.

[OWB05] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. on Software Engineering*, 31:340–355, 2005.

[PBMW98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[RM08] Giancarlo Succi Raimund Moser, Witold Pedrycz. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE*, pages 181–190, 2008.

[TF79] W. Tsai and K.-S. Fu. Error-correcting isomorphism of attributed relational graphs for pattern analysis. *IEEE Trans. on Systems, Man, and Cybernetics*, 9:757768, 1979.

[ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, May 2007.