International Journal of Software Engineering
and Knowledge Engineering
Vol. 22, No. 5 (2012) 1–24

DOI: 10.1142/S0218194012005937

LEARNING BETTER INSPECTION OPTIMIZATION POLICIES

MARKUS LUMPE* and RAJESH VASA†

Faculty of ICT, Swinburne University of Technology
Hawthorn, Australia
**mlumpe@swin_edu_au*
†*rvasa@swin_edu_au*

TIM MENZIES‡ and REBECCA RUSH§

CSEE, West Virginia University
Morgantown, West Virginia
‡*tim@menzies_us*
§*rrush4@mix_wvu_edu*

BURAK TURHAN

Info Processing Science, University of Oulu
Oulu, Finland
turhanb@computer_org

Received 15 July 2011
Revised 9 September 2011
Accepted 19 December 2011

Recent research has shown the value of social metrics for defect prediction. Yet many repositories lack the information required for a social analysis. So, what other means exist to infer how developers interact around their code? One option is *static code metrics* that have already demonstrated their usefulness in analyzing change in evolving software systems. But do they also help in defect prediction? To address this question we selected a set of static code metrics to determine what classes are most "active" (i.e., the classes where the developers spend much time interacting with each other's design and implementation decisions) in 33 open-source Java systems that lack details about individual developers. In particular, we assessed the merit of these activity-centric measures in the context of "inspection optimization" — a technique that allows for reading the fewest lines of code in order to find the most defects. For the task of inspection optimization these activity measures perform as well as (usually, within 4%) a theoretical upper bound on the performance of any set of measures. As a result, we argue that activity-centric static code metrics are an excellent predictor for defects.

Keywords: Data mining; defect prediction; static measures.

1. Introduction

A remarkable recent discovery is that *social metrics*, which model the sociology of the programmers working on the code, can be an effective predictor for defect injection and removal [1−4]. For example, Guo *et al.* [4] demonstrate that the reputation of the developer who reported a defect naturally relates to the odds that this defect will get fixed eventually.

Models that offer predictions on likely location of defects have traditionally relied on static code metrics [16, 17, 20]. Yet, the premise of social metrics research is that code repositories contain more than just static code measures and that these measures provide a valuable dimension worth investigating. But, not all code repositories contain detailed knowledge about how developers interact around a code base. Consider, for example, the Helix project [5] which has studied 40+ multi-year large open-source Java systems under active development. Many developers contributed to those systems but their code repositories are very weak sources for information regarding a developer's social context. This occurs because the systems in use to support software development do not always capture the social dimension consistently. Additionally, aspects like "reputation" [4] are fuzzy and there is no widely accepted standard to measure these social dimensions.

Nevertheless, social aspects do add a valuable and useful dimension that we should aim to measure objectively. In this paper, we show that it is possible to use static code measures to capture how programmers interact with their code by taking into consideration software evolution, that is, we add the dimension of time. Specifically, it is feasible to find what parts of the code are most "active," that is, are the focus of much of the shared attention of all developers working to organize behavior and functionality at suitable system-specific levels [6−8]. This opens intriguing options for guiding *quality assurance* (QA) processes. In particular, we demonstrate that a small set of *activity-centric* static code metrics [7, 8] can serve as a good predictor for defects in object-oriented software.

Now, defect prediction techniques, in general, rely heavily on the available input [64, 65] and, depending on the amount of processing required, can be characterized as either lightweight or complex quality assurance methods. Early approaches were based on univariate logistic regression [43, 66]. Later models for defect prediction incorporated multiple explanatory variables in the analysis in recognition of the fact that the actual probability of defects is a function of several factors [36−39]. Recently, machine learning [15, 40, 41] has become a formidable contender in the area of defect prediction that offers an promising alternative to standard regression-based methods. However, the more complex these approaches become the more difficult they are to master, especially, when the reasons as to why the underlying model characterizes some modules more defect-prone than others are hard to grasp. This can hamper adoption of these techniques in industry.

An ideal approach for defect prediction, we advocate, would be relatively straightforward, based on simple measures, easy to understand, and directly associated with the developer's mental model for effective software development [6]. This is the domain of activity-centric static code metrics [7, 8]. In particular, we present evidence in this paper that, based on experiments with 33 open-source Java software systems, shows that activity-centric metrics perform very close to the theoretical upper bound on defect prediction performance [67]. To compute that upper bound, we adopt the *defect density inspection bias* proposed by Arisholm & Briand [20] which aims at an optimal inspection policy in order to locate defects in the code base. Such a policy seeks to identify the *most* faults while reading the *least* amount of code and fits within the developer's workflow as it yields an inspection strategy that orders classes based on their defect probability.

The rest of this paper is structured as follows. The next section presents the economic case for defect detection (find more bugs, earlier) then introduces the concepts of static code defect predictors and inspection optimization. We then turn to the experiments showing the value of activity measures. We demonstrate that in our selected systems, activity-based defect predictors work within 4% of a theoretical upper bound on predictor performance (this is the basis for our claim that a small set of static metrics can generate an excellent performance within the context of inspection optimization). The validity of our conclusions is then discussed, which will lead into a review of possible future directions for this work.

2. Background

This section reviews the core motivation of this work: *the reduction of software construction costs by an earlier detection of defects.* We start with a discussion of some of the practical considerations governing defect detection in the software life cycle. Then, we shift our focus on *lightweight sampling policies.* In particular, we explore one special kind: *static code defect predictors.* Finally, we explore the use of data miners for the task of *inspection optimization.*

2.1. *Defect detection economics*

Boehm & Papaccio advise that reworking software is far cheaper earlier in the life cycle than later "*by factors of 50 to 200*" [9]. This effect has been widely documented by other researchers. A panel at IEEE Metrics 2002 concluded that finding and fixing severe software problems after delivery is often 100 times more expensive than finding and fixing them during the requirements and design phase [10]. Also, Arthur *et al.* [11] conducted a small controlled experiment where a dozen engineers at NASA's Langley Research Center were split into development and specialized verification teams. The same application was written with and without specialized verification teams. Table 1 shows the results: (a) more issues were found using specialized verification than without; (b) the issues were found much earlier. That is, if the verification team found the *same* bugs as the development team, but found them *earlier*, the cost-to-fix would be reduced by a significant factor. For example,

Table 1. Defects found with and without specialized verification teams. From [11].

Phase	With verification team	No verification team
Requirements	16	0
High-level design	20	2
Low-level design	31	8
Coding & user testing	24	34
Integration & testing	6	14
Totals	97	58

Table 2. Cost-to-fix escalation factors. From [12].

		Phase issue found					
		f = 1	f = 2	f = 3	f = 4	f = 5	f = 6
i	Phase issue introduced	Requirements	Design	Code	Test	Int	Operations
1	Requirements	1	5	10	50	130	368
2	Design		1	2	10	26	74
3	Code			1	5	13	37
4	Test				1	3	7
5	Integration					1	3
	$\Delta = \text{mean}(\frac{C[f,i]}{C[f,i-1]})$		5	2	5	2.7	2.8

Note: $C[f,i]$ denotes the cost-to-fix escalation factor relative to fixing an issue in the phase where it was found (f) versus the phase where it was introduced (i). The last row shows the cost-to-fix delta if the issue introduced in phase i is fixed immediately afterwards in phase $f = i + 1$.

consider Table 2 that shows the cost of quickly fixing an issue relative to leaving it for a later phase (data from four NASA projects [12]). The last line of that table reveals that delaying issue resolution even by one phase increases the cost-to-fix to $\Delta = 2 \ldots 5$. Using this data, Dabney *et al.* [12] calculate that a dollar spent on verification returns to NASA, on those four projects, \$1.21, \$1.59, \$5.53, and \$10.10, respectively.

The above notes leads to one very strong conclusion: *find bugs earlier*. But how? Software assessment budgets are finite while assessment effectiveness increases exponentially with assessment effort. However, the *state space explosion problem* imposes strict limits on how much a system can be explored via automatic formal methods [68, 69]. As to other testing methods, a *linear* increase in the confidence C that we have found all defects can take *exponentially* more effort. For example, for one-in-a-thousand detects, moving C from 90% to 94% to 98% takes 2301, 2812, and 3910 black box probes, respectively.[a] Exponential costs quickly exhaust finite resources. Standard practice is to apply the best available assessment methods on the

[a] A randomly selected input to a program will find a fault with probability p. After N random black-box tests, the chances of the inputs not revealing any fault is $(1-p)^N$. Hence, the chances C of seeing the fault is $1-(1-p)^N$ which can be rearranged to $N(C,p) = \frac{\log(1-C)}{\log(1-p)}$. For example, $N(0.90, 10^{-3}) = 2301$.

sections of the program that the best available domain knowledge declares is most critical. We endorse this approach. Clearly, the most critical sections require the best known assessment methods. However, this focus on certain sections can blind us to defects in other areas. Therefore, standard practice should be augmented with a *lightweight sampling policy* to explore the rest of the system. This sampling policy will always be incomplete. Nevertheless, it is the only option when resources do not permit a complete assessment of the whole system.

2.2. *Static code defect prediction*

A typical, object-oriented, software project can contain hundreds to thousands of classes. In order to guarantee general and project-related fitness attributes for those classes, it is commonplace to apply some quality assurance (QA) techniques to assess the classes's inherent quality. These techniques include inspections, unit tests, static source code analyzers, etc. A record of the results of this QA is a *defect log*. We can use these logs to learn *defect predictors*, if the information contained in the data provides not only a precise account of the encountered faults (i.e., the "bugs"), but also a thorough description of static code features such as *Lines of Code* (LOC), complexity measures (e.g., McCabe's cyclomatic complexity [31]), and other suitable object-oriented design metrics [6−8, 14].

For this, data miners can learn a predictor for the number of *defective* classes from past projects so that it can be applied for QA assessment in future projects. Such a predictor allows focusing the QA budgets on where it might be most cost effective. This is an important task as, during development, developers have to *skew* their quality assurance activities towards artifacts they believe require most effort due to limited project resources.

Now, static code defect predictors yield a *lightweight sampling policy* that, based on suitable static code measures, can effectively guide the exploration of a system and raises an alert on sections that appear problematic. One reason to favor static code measures is that they can be automatically extracted from the code base, with very little effort even for very large software systems [16]. The industrial experience is that defect prediction scales well to a commercial context. Defect predicting technology has been commercialized in *Predictive* [17] a product suite to analyze and predict defects in software projects. One company used it to manage the safety critical software for a fighter aircraft (the software controlled a lithium ion battery, which can over-charge and possibly explode). After applying a more expensive tool for structural code coverage, the company ran Predictive on the same code base. Predictive produced results consistent with the more expensive tool. But, Predictive was able to faster process a larger code base than the more expensive tool [17].

In addition, defect predictors developed at NASA [15] have also been used in software development companies outside the US (in Turkey). When the inspection teams focused on the modules that trigger the defect predictors, they found up to 70% of the defects using just 40% of their QA effort (measured in staff hours) [18].

Finally, a subsequent study on the Turkish software compared how much code needs to be inspected using random selection versus selection via defect predictors. Using random testing, 87% of the files would have to be inspected in order to detect 87% of the defects. However, if the inspection process was restricted to the 25% of the files that trigger the defect predictors, then 88% of the defects could be found. That is, the same level of defect detection (after inspection) can be achieved using $\frac{87-25}{87} = 71\%$ less effort [19].

2.3. *Inspection optimization*

Inspection optimization is a term proposed by Arisholm & Briand [20]. It is a technique for assessing the value of, say, a static code defect predictor. They define it as follows:

> *If X% of the classes are predicted to be defective, then the actual faults identified in those classes must account for more than X% of all defects in the system being analyzed. Otherwise, the costs of generating the defect predictor is not worth the effort.*

In essence, this is *inspection optimization* — find some ordering to project artifacts such that humans have to read the *least* code in order to discover the *most* faults, which we model as outlined below:

- *After* a data miner predicts a class is defective, *then* a secondary human team examines the code.
- This team correctly recognizes $\Delta\%$ of the truly defective classes (and $\Delta = 100\%$ means that the inspection team is perfect at its task and finds every defect present).
- A *good learner* is one that finds the *most* defective classes (measured in terms of probability of detection, *pd*) in the *smallest* classes (measured in terms of lines of code, LOC).

Inspection optimization can be visualized using Fig. 1 that illustrates three plausible *inspection ordering policies*:

- The blue *optimal* policy combines knowledge of class size and the location of the actual defects.
- The green *activity* policy guesses defect locations using a defect predictor learned from the activity measures.
- The red *baseline* policy ignores defect counts and just sorts the classes in ascending order of size.

Each of these ordering policies sorts the code base along the x-axis. The code is then inspected, left to right, across that order, so that, by the end of the x-axis, we have read 100% of the code. Along the way, we encounter classes containing $y\%$ of

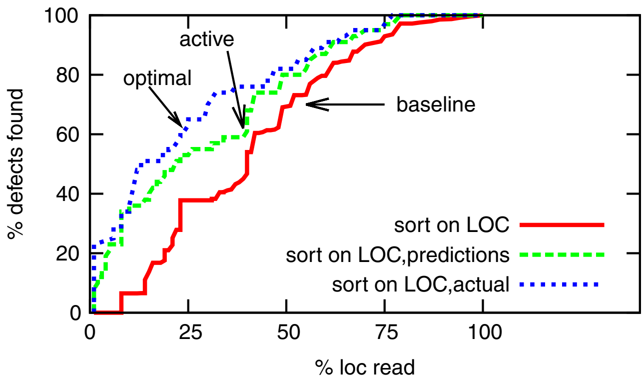

Fig. 1. Percentage of defects found after sorting the code using different inspection ordering policies. Note that, in this case, developers were continually modifying a small number of very active classes handling complex interfacing tasks. Hence for the blue curve, reading just this 1% of the code found nearly a quarter of the defects.

defects (a.k.a. *recall*). A *better* policy finds more defects sooner, that is, it yields a larger area under the curve of %LOC-vs-recall. In Fig. 1, we note that that the green *activity* policy does better than the red *baseline* (and comes close, within 95%, of the blue *optimal*).

These three policies are defined by an equation modeling the distance to some *utopia* point of most defects and smallest LOC:

$$0 \leq (score(D_c, L_c, \alpha)) = \frac{\sqrt{\alpha D_c^2 + (1 - L_c)^2}}{\sqrt{\alpha + 1}} \leq 1$$

Here, D_c and L_c are the number of defects and lines of code in class C (normalized to range between 0 and 1), whereas α is a constant controlling the sorting. At $\alpha = 0$, we ignore defects and sort only on LOC. This implements the *baseline* policy. This baseline policy is the *Koru ordering* advocated by researchers who argue that smaller classes have a relatively higher density of errors [21−23]. Note that if the *activity* policy cannot out-perform *baseline*, then our notion of activity is superfluous.

The other policies use $\alpha = 1$. For the *activity* policy, we have to:

- Train a learner using the measures of Table 3 without LOC,
- Set D_c via the learned model,
- Sort using score, D_c, LOC, and $\alpha = 1$,
- Calculate Fig. 1 and determine the area under the %LOC-vs-recall curve.

The *optimal* policy does the same, but sets D_c using the historical defect logs. Note that *optimal* is different to *activity* since the former knows exactly where the defects are, whereas the latter must guess the defect locations using the learned model.

In practice, the *optimal* policy is impossible to apply since it implies that we were to know the number of defects *before* the classes would be inspected. However, it is the theoretical upper-bound on the performance of inspection optimization. Hence, we report *activity* and *baseline* performances as a ratio of the area under the curve of *optimal*.

This ratio calculation has another advantage. Note that the Δ effectiveness of the secondary human inspection team is the same, regardless of the oracle that sorts the code. Hence, in the ratio calculation, Δ cancels out and we can ignore it from our analysis.

3. Activity

The novel feature of this paper is augmenting the usual static code measures with the concept of *activity*. As discussed below, we find that activity can be a very useful concept for inspection optimization.

When do we call a software artifact, say a class, "active"? We contend that activity arises when code is being modified, typically via enhancement or correction. This is change and we can detect and measure it through the evolution of the associated volumetric and structural properties of a class [6].

However, one surprising observation from the Helix studies [6, 7] has been that (a) only a small set of highly active classes undergoes change frequently and (b) predictable patterns of modification emerge very early in the lifetime of a software system. Therefore, we ask whether the same metrics used to analyze the Helix data set can also guide defect discovery, since change and defects are closely related concepts. In particular, we argue that change can lead to defects via:

- *Defect discovery*: Since active classes are used more frequently by developers, then developers are most likely to discover their defects earlier.
- *Defect injection*: When developers work with active classes, they make occasional mistakes, some of which lead to defects. Since developers work on active classes more than other classes, then most developer defects accumulate in the active classes.

(The second point was first proposed by Nagappan & Ball who say "*code that changes many times prerelease will likely have more post-release defects than code that changes less over the same period of time*" [13].)

Table 3 summarizes our choices of measures of activity, each tagged with a rationale motivating its selection. These measures capture volumetric and the structural properties of a class and provide us with an empirical component for detecting and measuring change. Furthermore, these measures are sufficiently broad to encompass, from a design perspective, the amount of functionality as well as how the developers have structurally organized the solution, and how they chose to decompose the functionality.

Table 3. Measures used in this study (collected separately for each class).

Measure	Description	Rationale for selection
Bugs	annotations in the source control logs	Used to check our predictions
LOC	lines of code in the class	Used to estimate inspection effort
Getters	get methods	Read responsibility allocation
Setters	set methods	Write responsibility allocation
NoM	all methods	Breadth of functional decomposition
InDegree	other classes depending on this class	Coupling within design
OutDegree	other classes this class depends upon	Breadth of delegation
Clustering coefficient	degree to which classes cluster together	Density of design

The measures NoM, Getters, and Setters define simple class-based counts (cf. Table 4). For the complexity measures InDegree, OutDegree, and Clustering Coefficient, however, we need to construct a complete class dependency graph first. The class dependency graph captures the dependencies between these classes. That is, when a class uses either data or functionality from another class, there is a dependency between these classes. In the context of Java software, a dependency is created if a class inherits from a class, implements an interface, invokes a method on another class (including constructors), declares a field or local variable, uses an exception, or refers to class types within a method declaration. Thus, a class dependency graph is an ordered pair (N, L), where N is a finite, nonempty set of types (i.e., classes and interfaces) and L is a finite, possibly empty, set of *directed links* between types (i.e., $L \subseteq N \times N$) expressing the dependencies between classes. For the purpose of the metrics extraction, we analyze each node $n \in N$ in the graph to compute the structural complexity metrics of class C type node n represents as shown in Table 4.

Table 4. Activity-centric metrics definitions.

Metrics	Definition
NoM	Counts all member functions defined by class C.
Getters	Counts all non-overloaded member functions in class C with arity zero, whose name starts with "get."
Setters	Counts all non-overladed member functions in class C with arity one, whose name starts with "set."
InDegree	Let n be the type node for class C. Then $\lvert\{(n', n) \in L \mid n \neq n'\}\rvert$ is the in-degree of class C.
OutDegree	Let n be the type node for class C. Then $\lvert\{(n, n') \in L \mid n \neq n'\}\rvert$ is the out-degree of class C.
Clustering coefficient	Let n be the type node for class C. Then $$\frac{2\lvert\{(n_i, n_j) \in L \lvert n_i, n_j \in N_n\}\rvert}{\lvert N_n\rvert(\lvert N_n\rvert - 1)}$$ is the clustering coefficient of class C, where N_n is the neighborhood of n with $$N_n = \{n' \lvert (n', n) \in L \vee (n, n') \in L\}$$

An important feature of these measures is that they are relatively easy to collect. For example, one measure we rely on for defect prediction is the *Number of Getter Methods* (Getters) that developers have added to a class. Parsers for such simple measures are easy to obtain from early design representation (e.g., UML models) and can, with little effort, be adapted to new languages. Moreover, all measures are pairwise independent [7, 8] (measured using Spearman's rank correlation). In particular, Getters and Setters do not occur in pairs and are not being used as a means to expose simply the private fields of a class [8]. In general, the odds are only 1:3 that if a class defines a getter, then this class will also provide a matching setter method.

4. Activity and Inspection Optimization

To assess the value of the selected activity-centric metrics (cf. Table 3), we distilled them for 33 open-source Java projects from the Helix project and used to resulting information to build defect predictors. As shown below, the median value for the $\frac{learning}{oracle}$ ratio is 96%, that is, very close to the theoretical upper bound possible for any defect predictor for the task of inspection optimization.

4.1. *Data selection*

The data used in this study was built as a join between two complementary data sets:

- The PROMISE repository [24] contains defect information for various open-source object-oriented systems. The defect data for this study was collected by Jureczko [25].
- The Helix repository [5, 6] provides static source code metrics for a compilation of release histories of non-trivial Java open-source software systems.

The joined data sets represent 33 releases of the projects listed in Table 5. All projects are "long term" (at least 15 releases span over a development period of 36 months or more) and comprise more than 100 classes each. In addition, every project can be characterized as either *application*, *framework*, or *library*, a broad "binning"

Table 5. Java systems used in this study.

System	Description
ant	Build management system
ivy	Dependency manager
jedit	Text editor
lucene	Text search engine
poi	API for Office Open XML standards
synapse	Enterprise service bus
velocity	Template language engine
xalan	XSLT processor
xerces	XML processor

strategy that reflects the inherent, yet recurring, differences in software design and composition. For a detailed description of these data sets, see Vasa's Ph.D. thesis [6].

For LOC (i.e., the *Lines of Codes*) we use an estimator based on the size of the compiled byte code rather than the actual source code. The byte code provides us with a noise-free image of the class's defined functionality. LOC of a class C is given as the sum of the following components extracted from the binaries:

- out-degree of C line(s) for `import` statements
- 1 line for the class declaration
- 1 line for super class declaration if not `java.lang.Object`
- 1 line for each interface implemented by C
- 1 line for each field defined in class C
- 1 line for each method m defined in class C, plus
  - # parameters of m
  - # throws defined by m
  - `MaxLocals` attribute (i.e., local variables) of m
  - # byte code instructions in m

We selected these components as they provide a very consistent approximation of the size of source code independent of the actual coding style used. The LOC estimator correlates very well with the lines of source code (cf. Fig. 2). Furthermore, for the purpose of inspection optimization, an added benefit of processing byte code rather than source is that the data miner will only report those classes that actually appear in the released version. That is, the secondary human inspection team is given further guidance to focus its QA effort. Previous research [26−29] found that, in

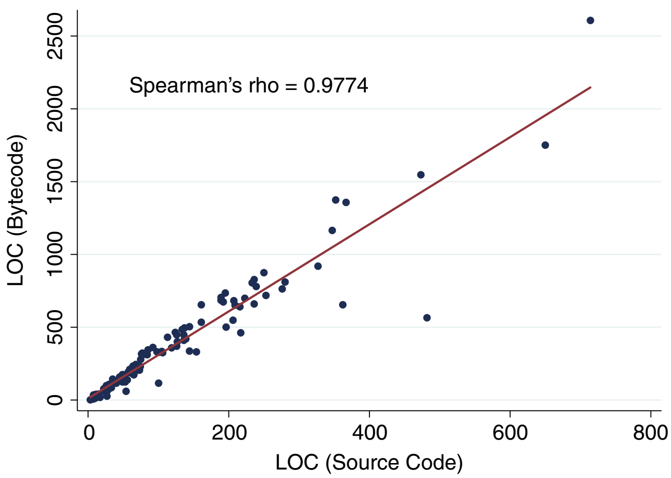

Fig. 2. *Lines of Code* (LOC) extracted from byte code is a very strong approximation of the LOC extracted directly from source code.

general, not all parts of the code base are included in the final release build. This is due to the release build configuration settings used. Hence, processing 10% of the classes as per byte code, is equivalent to analyzing 10% of the active source code classes (i.e., the classes that must be inspected in the QA process).

The joined, activity-based, data sets are constructed as follows:

(1) From the PROMISE repository we fetch the bug information for release N per class.
(2) We extract from the Helix repository the static code metrics, including LOC, for release N per class.
(3) Using the fully qualified class name as key, both information is merged into the activity data set for release N per class.

Table 6 shows the distribution of defects seen in our classes. Usually, most classes have no defects, but in 10% of cases, each class has more than 1 to 5 recorded defects.

4.2. *Experimental setup*

For the purpose of finding a predictor for inspection optimization we employed a technique, called *N-way cross-evaluation* [30]. The data set is divided into $N = 10$ buckets. For each bucket in the N-way, a predictor is learned on the nine of the buckets, then tested on the remaining bucket. These N studies implement N *hold out* studies where a model is tested on data *not* used in training.

To appreciate cross-validation, consider another approach called *self-test* where the learned model is assessed on the *same* data that was used to create it. Self-tests are deprecated by the research community [30]. If the goal is to understand how well a defect predictor will work on future projects, it is best to assess the predictor via hold-out modules not used in the generation of that predictor.

In the WEKA 3.7.3 implementation of the cross-val procedure used in this study, results are reported once for each test-instance as that instance appears in one of the N hold-outs.[b] So a data set containing C examples will generate C predictions, regardless of the value of N used for the number of hold-outs.

4.3. *Selection of learners*

As mentioned above, there are many methods for converting static code measures into defect predictors [15, 31−41]. We adopted Holte's *simplicity-first* heuristic [42] and applied a simple linear regression (LSR) algorithm available in WEKA [30], with no pre-processing.

[b] Note that prior to WEKA 3.7.2, the cross-val procedure `java -cp weka.jar $learner -t file.arff` incorrectly returns *self-test* results.

Table 6. Percentile distributions, defects per class.

System	# Classes	10%	30%	50%	70%	90%
ant-1.3	125	0	0	0	0	1
ant-1.4	178	0	0	0	0	1
ant-1.5	293	0	0	0	0	1
ant-1.6	351	0	0	0	0	2
ant-1.7	493	0	0	0	0	2
ivy-1.4	241	0	0	0	0	0
ivy-2	352	0	0	0	0	1
jedit-3.2	272	0	0	0	1	4
jedit-4	306	0	0	0	0	2
jedit-4.1	312	0	0	0	0	2
jedit-4.2	367	0	0	0	0	1
jedit-4.3	492	0	0	0	0	0
lucene-2	195	0	0	0	1	4
poi-2	314	0	0	0	0	1
synapse-1	157	0	0	0	0	0
synapse-1.1	222	0	0	0	0	1
synapse-1.2	256	0	0	0	1	2
velocity-1.6	229	0	0	0	1	2
xalan-2.4	428	0	0	0	0	1
xalan-2.5	763	0	0	0	1	2
xalan-2.6	875	0	0	0	1	2
xerces-1	162	0	0	0	2	2
xerces-1.2	438	0	0	0	0	1
xerces-1.3	452	0	0	0	0	1
lucene-2.2	247	0	0	1	2	4
lucene-2.4	428	0	0	1	2	5
poi-1.5	237	0	0	1	1	4
poi-2.5	348	0	0	1	2	2
poi-3	442	0	0	1	1	2
velocity-1.4	196	0	1	1	1	2
velocity-1.5	214	0	0	1	2	4
xalan-2.7	908	1	1	1	1	2
xerces-1.4	329	0	0	1	2	4

Note: The table is sorted by the median defects (see the 50% percentile column). For example, in xalan-2.7 the median (50th percentile) defects per class is 1, whereas in lucene-2.4, 10% of classes have 5 defects or more.

Note that WEKA's LSR tool uses a simple greedy *back-select*, which is applied after the linear model has been generated. In that back- select, WEKA steps through all the attributes removing the one with the smallest standardized coefficient until no improvement is observed in the estimate of the model error given by the Akaike information criterion. As a consequence, some attributes may be absent from the final learned model.

Initially, we planned to test various learners, feature extractors, instance selectors, and discretization methods (as we have done in the past [15, 40, 41]). But our results were so encouraging that there was little room for further improvement over simple LSR.

5. Results

5.1. *Sanity check*

Table 7 shows the distribution of *actual-predicted* defects for our classes where *actual* comes from historical logs and *predicted* comes from the C predictions seen in our 10-way. This result is our *sanity check*: if the *actual-predicted* values were large, then we would doubt the value of activity-based defect prediction. Note that, in the median case (shown in the middle 50% column), the predictions are very close to actuals (−0 to −0.3). Since our estimates are close to actuals, we may continue.

Table 7. Percentile distributions of actual-predicted number of defects per class.

System	10%	30%	50%	70%	90%
lucene-2.4	−2.0	−0.9	−0.3	0.4	2.6
velocity-1.6	−1.4	−0.6	−0.3	0.0	1.4
xerces-1.0	−1.1	−0.6	−0.3	0.7	1.0
ant-1.4	−0.4	−0.3	−0.2	−0.1	0.8
lucene-2.0	−2.0	−0.8	−0.2	0.1	1.7
poi-1.5	−1.6	−0.9	−0.2	0.0	2.3
synapse-1.2	−0.8	−0.4	−0.2	0.0	1.0
xalan-2.5	−0.9	−0.5	−0.2	0.5	0.8
xalan-2.6	−0.9	−0.5	−0.2	0.4	1.2
xalan-2.7	−0.5	−0.3	−0.2	0.1	0.7
xerces-1.2	−0.5	−0.2	−0.2	0.0	0.9
xerces-1.4	−2.2	−0.4	−0.2	0.4	0.8
ant-1.3	−0.5	−0.3	−0.1	0.0	0.6
ant-1.7	−0.8	−0.3	−0.1	0.0	0.8
ivy-2.0	−0.3	−0.1	−0.1	0.0	0.3
lucene-2.2	−2.5	−0.8	−0.1	0.4	2.0
poi-2.0	−0.2	−0.1	−0.1	−0.1	0.7
poi-3.0	−1.1	−0.4	−0.1	0.0	1.0
synapse-1.0	−0.3	−0.2	−0.1	0.0	0.0
synapse-1.1	−0.7	−0.4	−0.1	0.0	0.9
velocity-1.5	−1.5	−0.7	−0.1	0.3	1.5
xalan-2.4	−0.5	−0.2	−0.1	0.0	0.7
ant-1.5	−0.3	−0.1	0.0	0.0	0.3
ant-1.6	−0.8	−0.3	0.0	0.0	0.9
ivy-1.4	−0.2	−0.1	0.0	0.0	0.0
jedit-3.2	−2.3	−0.8	0.0	0.0	1.8
jedit-4.0	−1.3	−0.5	0.0	0.0	1.0
jedit-4.1	−1.1	−0.4	0.0	0.0	1.0
jedit-4.2	−0.6	−0.2	0.0	0.0	0.3
jedit-4.3	−0.1	0.0	0.0	0.0	0.0
poi-2.5	−1.3	−0.6	0.0	0.7	0.9
velocity-1.4	−1.2	−0.2	0.0	0.1	1.0
xerces-1.3	−1.4	−0.2	0.0	0.0	0.7

Note: For example, the median (50th percentile) value of *actual-predicted* is −0.3 to 0.

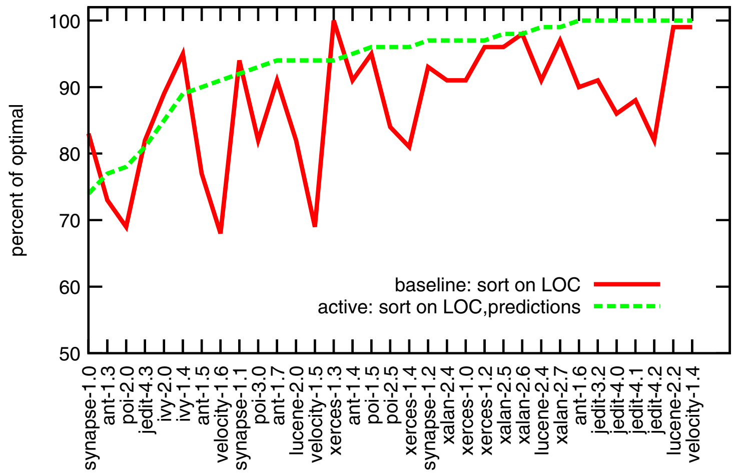

Fig. 3. Performance results expressed as a ratio of the *optimal* policy. Data sets are sorted according to the *activity* results. Median values for *baseline*, *activity* are 91% and 96% of *optimal*, respectively.

5.2. *Baseline and activity versus optimal*

Figure 3 shows the ratio of the *optimal* policy achieved with the *activity* policy (the green curve) and the *baseline* policy (the red curve). These curves are statistically significantly different (Wilcoxon, 95% confidence). For both curves, the result are expressed in as a ratio of the *optimal* policy that uses historical knowledge to determine the number of defects in each class.

We observe that the results of the *baseline* policy are far more erratic than for the *activity* policy. The *spread* of a distribution is the difference between the 75% and 25%th percentile range. The spread of the values in Fig. 3 are:

- *Activity*: $98 - 91 = 7$
- *Baseline*: $95 - 82 = 13$

That is, the results of the *activity* policy are more predictable (fall into a narrower range), whereas the results from *baseline* can spread nearly twice as far. Moreover, the *activity* results not only are more predictable, but also out-perform the *baseline* policy. The median value of the red *baseline* policy results (i.e., inspecting the code based on increasing class size) is 91% of *optimal*. Note that *baseline* is rarely any little better than *activity*, and often, it is much worse:

- When *baseline* out-performs *activity* (in only $\frac{3}{33}$ of our comparisons), it does so only by a small margin.
- In the $\frac{30}{33}$ data sets where *baseline* does worse than *activity*, sometimes it does much worse (see the velocity-1.5 and velocity-1.6 results which fall to 70% of *optimal*).

The median value of the *activity* policies results are 96%, which is within 4% of *optimal*. Further, the top ten results of *activity* all score 100% of *optimal* (see the right-hand side of the green curve in Fig. 3). That is, for the purpose of optimizing inspection, there is little to no room for improvement on top of the activity-centric measures. Hence, we strongly recommend the *activity* policy.

5.3. *Summary*

Our key observations in this study are as follows:

- According to Table 7, activity-centric measures combined with linear regression lead to defect predictors with low error rates in open-source object-oriented systems.
- According to Fig. 3, for the task of inspection optimization, activity-centric defect prediction works significantly better than the baseline and very close to the optimum.

6. Discussion

The results here are quite unequivocal — activity is a strong predictor for software defects, and this effect can be detected with a simple model such as linear regression. Hence, we need to explain why this effect has not been reported before. We conjecture that the use of a small set of activity-centric static metrics is *too simple* and *too novel* a concept to be reported previously.

6.1. *Too simple?*

We can broadly classify object-oriented software quality research as (a) studies with more focus on prediction models than the metrics, and (b) studies with more focus on metrics validation than the models (as in this study). It is no surprise that the former kind of studies did not explicitly investigate the concept of activity, as they usually operate within existing sets of common metrics in order to choose the best model among many. The literature offers many complex methods for data mining such as support-vector machines, random forests, and tabu search to tune the parameters of a genetic algorithm (i.e., [32–35]). In this era of increasing learner complexity, something as easy as linear regression on a small set of static code measures aiming especially on activity may have been discounted before being explored rigorously. Therefore, our first explanation is that the use of activity as a concept *is so simple* that it escaped the attention of this type of research.

Nevertheless, we cannot ignore the latter type of studies, in which the focus has usually been validating object-oriented metrics as predictors of defects through correlational methods (i.e., [43–47]). Briand & Wüst provide an extensive survey of empirical studies of quality in object-oriented systems, and observe that the majority of studies falls in this category [48]. However, they also state that only half of the studies employ multivariate prediction models, and the other half just reports univariate relations between object-oriented metrics and defects. Further, only half of the studies with a prediction model, conduct a proper performance analysis through cross-validation. After this filtering, remaining work contains hard-to-compare empirical studies, where the size and the number of data sets are so small that the combined results are conflicting and do not reveal a common trend possibly due to varying contexts of the studies.

Another aspect of related studies is that they consider certain subgroups of object-oriented metrics relating to concepts such as coupling, cohesion, inheritance and polymorphism, and size [48]. Briand & Wüst report that the significance of the relation between different subgroups of metrics and defects are mostly inconclusive, and only a number of size and coupling measures are consistent. We have further run a smaller-scale review of major studies conducted with the guidelines of the original survey [20, 38, 43−45, 47, 49−52]. Similar to Briand & Wüst, we observed that the table of metrics versus different systems used to assess those metrics were sparsely populated.

6.2. *Too novel?*

The starting point for this research was the observation in the Helix data sets that most classes stabilize very early in their life cycle while a very small number of active classes garner the most attention by developers [6, 7]. As discussed above in Sec. 3, this is not the standard picture of the life cycle of a class. To us, this observation was so unique that it prompted the question "*does the amount a class is used by developers predict for system defects*" (i.e., this study). However, without that initial surprising observation, we would not have conducted the study reported in this paper.

Compared to other studies (e.g., studies surveyed in [48]), the size and number of data sets used in our study is extensive and reveals a clear benefit of using activity-centric metrics in the context of open-source object-oriented systems. In contrast to our concept of activity, Turhan *et al.* [53] investigate popularity. Their approach is to augment standard static code metrics within a call graph-based ranking framework, which is inspired by the PageRank algorithm [54]. Rather than constructing learners with a standard set of metrics that value each module equally, Turhan *et al.* first rank the modules using the dependency graph information and weigh the information learned from "popular" modules more. Their approach reduced the false alarm rates significantly. However, this technique is an indirect way of utilizing activity, and does not include explicit activity-centric metrics that are used in this study. Similarly, Zimmermann *et al.* include *eigenvector centrality*, a measure of *closeness centrality* of network nodes similar to PageRank, in their analysis of complexity versus network metrics for predicting defects from software dependencies [55]. Though, eigenvector centrality is found to be correlated with defects for the

Windows system they have evaluated, this metric did not stand out among other network or complexity based metrics to allow a discussion on "activity" (see the next section below for a possible cause). Finally, Kpodjodo *et al.* monitored their proposed, again PageRank inspired, *Class Rank* metric among several versions of a single system and found moderate evidence in favor [56]. In this paper, we handle activity as a concept rather than relying on a single measure, and we achieve near optimal results compared to moderate improvements of similar work.

6.3. *Hidden?*

It is possible that *activity was buried* under other effects. When we look at the measures that we have used in previous studies (e.g., [15]), we can see some overlap between those measures and ones used here (cf. Table 3). Miller [57], Witten & Frank [30], and Wagner [58] offer a theoretical analysis discussing how an excess of attributes containing multiple strong predictors for the target class can confuse learning. For example, both Wagner and Miller note that in a model comprising N variables, any noise in variable N_i adds to the noise in the output variables.

We have also observed supporting evidence for this explanation in our small scale quality-in-object-oriented-systems review. In all cases, where both an univariate and a multivariate analysis is being utilized, it is common for metrics that have been verified by the univariate model to not be included in the multivariate model for the same data [43, 44, 47, 49, 51, 52]. El Emam *et al.* use this phenomenon to control for the confounding effects of size on metrics believed to serve as suitable predictors for defects [22]. Similarly, the multivariate model metrics may include those that are not verified by the univariate model [20, 38, 49], for which Guyon *et al.* provide simple examples showing that the prediction power can be significantly increased when features are used together rather than individually [59]. Hence, even though some measures exist in a data set, noise from the other variables may have drowned out their effect.

7. Validity and Future Work

Internal validity: Apart from joining the PROMISE data sets (for defect counts) with the Helix data sets (for the activity-centric measures), we did not pre-process the datasets in any way. This was done to enable replication of our results.

Construct validity: We have made the case above that the measures listed in Table 3 reflect the "activity" of different classes, that is, how often a developer will modify or extend the services of a class as an expression of the attractiveness of this class for the developer's design choices. This case has not been tested here. Hence:

Future work 1: Analyze participant observation of developers to determine what classes they inspect as part of their workflow.

External validity: Our use of cross-validation means that all the results reported above come from the application of our models to data not seen during training. This gives us some confidence that these results will hold for future data sets.

As to our selection of data sets, the material used in this study represents real-world use, collected from real-world projects. Measured in terms of number of data sets, this paper is one of the largest defect prediction studies that we are aware of. Nevertheless, there is a clear bias in our sample: Open-source Java systems. Hence:

Future work 2: Test the validity of our conclusions to close-sourced, non-object-oriented, and non-Java projects.

Conclusion validity: We take great care to *only* state our conclusions in terms of areas under a %LOC-vs-recall curve. For the purpose of finding the most defects after inspecting the fewest lines of code (i.e., the inspection optimization criterion proposed by Arisholm & Briand [20]), the activity-centric metrics exhibit an excellent performance (median results within 96% of the optimum).

While the area under a %LOC-vs-recall is an interesting measure, it is not the only one seen in the literature. Hence:

Future work 3: Explore the value of activity for other evaluation criteria. Those other criteria may include:

- Counting the number of files inspected, rather than the total LOC, as done, for example, by Weyuker, Ostrand, and Bell [60, 61],
- Precision, as advocated, for example, by Zhang & Zhang [62] (but depreciated by Menzies *et al.* [63]),
- Area under the curve of the *pd-vs-pf* curves, as used by Lessmann *et al.* [32].

8. Conclusion

We have shown above that a repository containing just static code measures can still be used to infer interaction patterns amongst developers. Specifically, we studied the "active" classes, that is, the classes where the developers spend much time interacting with each other's design and implementation decisions. In 33 open-source Java systems, we found that defect predictors based on static code measures that model "activity" perform within 96% of a theoretical upper bound. This upper bound was derived assuming that the goal of the detectors was "inspection optimization," that is, read the fewest lines of code to find the most defects.

Though, we have focused on inspection optimization and limited our discussions around it, application of our techniques is not limited within the scope of this particular QA method. For example, our techniques can be directly applied to address regression test case selection (or regression test prioritization) problem, especially in very large systems. The important challenges for such systems are (a) to identify specific parts of the system against which regression tests should be developed and (b) to determine which tests should have priority over others within the existing (possibly huge) regression test library. In practice, it usually takes from a few hours to weeks for developers to get feedback from regression test results (without considering the cost of mental context switch overheads for developers). Our techniques can be used to address both problems: (a) they point to most problematic parts, so regression tests should cover those parts, (b) they provide a prioritization of problematic parts, so a small portion of all tests consisting of high priority ones could be run more frequently to provide faster feedback to developers. While the scope of our hypothetical example is the whole system, it is straightforward to scale it down to the operational level where developers can also benefit from our techniques directly: developers can be guided to develop and run local regressions tests on the critical parts in their local machines as pointed out by our techniques. In summary, applications of our techniques in different QA activities allow cost reductions through efficient management of resources and faster (early) feedback cycles to stakeholders.

There is another aspect of activity-centric measures that recommends their use. In this paper, we show that simple linear regression over these measures works very well indeed. That is, the machinery required to convert these measures into defect predictors is far less complex than alternative approaches, such as:

- Lessmann's random forests and support-vector machines [32],
- The many methods explored by Khoshgoftaar [33–35],
- Defect prediction via multiple explanatory variables [38, 39],
- Our own defect predictors via feature selection [15], instance selection [40], or novel learners built for particular tasks [41].

The comparative simplicity of activity-centric prediction, suggests that previous work [31−39], including our own research [15, 40, 41] may have needlessly complicated a very simple concept, that is, defects are introduced and discovered due to all the activity around a small number of most active classes.

Acknowledgments

This work was conducted at Swinburne University of Technology, West Virginia University, and University of Oulu with partial funding from (1) the New Zealand Foundation for Research, Science and Technology, (2) the United States National Science Foundation, CISE grant 71608561, (3) a research subcontract with the Qatar University NPRP 09-1205-2-470, and (4) TEKES under the Cloud-SW project in Finland.

References

1. C. Bird, N. Nagappan, H. Gall, B. Murphy and P. Devanbu, Putting it all together: Using socio-technical networks to predict failures, in *Proceedings of 20th International Symposium on Software Reliability Engineering*, Nov. 2009, pp. 109−119.
2. C. Bird, N. Nagappan, P. Devanbu, H. Gall and B. Murphy, Does distributed development affect software quality? An empirical case study of Windows Vista, in *Proceedings of 31st International Conference on Software Engineering*, May 2009, pp. 518−528.

3. N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proceedings of the 30th International Conference on Software Engineering*, May 2008, pp. 521−530.
4. P. J. Guo, T. Zimmermann, N. Nagappan and M. Brendan, Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows, in *Proceedings of the 32nd International Conference on Software Engineering*, May 2010, pp. 495−504.
5. R. Vasa, M. Lumpe and A. Jones, Helix — Software Evolution Data Set, http://www.ict.swin.edu.au/research/projects/helix, December 2010.
6. R. Vasa, Growth and Change Dynamics in Open Source Software Systems, Ph.D. dissertation, Swinburne University of Technology, Faculty of Information and Communication Technologies, October 2010.
7. R. Vasa, M. Lumpe, P. Branch and O. Nierstrasz, Comparative analysis of evolving software systems using the gini coefficient, in *Proceedings of 25th IEEE International Conference on Software Maintenance*, Edmonton, Alberta, IEEE Computer Society, September 2009, pp. 179−188.
8. M. Lumpe, S. Mahmud and R. Vasa, On the use of properties in java applications, in *Proceedings of the 21st Australian Software Engineering Conference*, Auckland, New Zealand, April 2010, pp. 235−244.
9. B. Boehm and P. Papaccio, Understanding and controlling software costs, *IEEE Trans. Software Engineering* **14**(10) (1988) 1462−1477.
10. F. Shull, V. R. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero and M. V. Zelkowitz, What we have learned about fighting defects, in *Proceedings of 8th International Software Metrics Symposium*, Ottawa, Canada, 2002.
11. J. D. Arthur, M. K. Groner, K. J. Hayhurst and C. M. Holloway, Evaluating the effectiveness of independent verification and validation, *IEEE Computer*, October 1999.
12. J. B. Dabney, G. Barber and D. Ohi, Predicting software defect function point ratios using a bayesian belief network, in *Proceedings of the PROMISE Workshop*, 2006.
13. N. Nagappan and T. Ball, Use of relative code churn measures to predict system defect density, in *ICSE'05*, 2005, pp. 284−292.
14. S. R. Chidamber and C. F. Kemerer, A metrics suite for object oriented design, *IEEE Trans. Software Engineering* **20**(6) (1994) 476−493.
15. T. Menzies, J. Greenwald and A. Frank, Data mining static code attributes to learn defect predictors, *IEEE Trans. Software Engineering* **33**(1) (2007) 2−13.
16. N. Nagappan and T. Ball, Static analysis tools as early indicators of pre-release defect density, in *Proceedings of the 27th International Conference on Software Engineering*, May 2005, pp. 580−586.
17. J. Turner, A predictive approach to eliminating errors in software code, 2006, available from http://www.sti.nasa.gov/tto/Spinoff2006/ct1.html.
18. A. Tosun, B. Turhan and A. Bener, Practical considerations of deploying AI in defect prediction: A case study within the Turkish telecommunication industry, in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, ACM, May 2009, pp. 1−9.
19. A. Tosun, A. Bener and R. Kale, AI-based software defect predictors: Applications and benefits in a case study, in *Twenty-Second IAAI Conference on Artificial Intelligence*, July 2010, pp. 1748−1755.
20. E. Arisholm and L. Briand, Predicting fault-prone components in a Java legacy system, in *Proceedings of the 5th ACM/IEEE International Symposium on Empirical Software Engineering*, September 2006, pp. 8−17.

21. A. Koru, D. Zhang, K. El Emam and H. Liu, An investigation into the functional form of the size-defect relationship for software modules, *IEEE Trans. Software Engineering* **35**(2) (2009) 293–304.
22. A. Koru, K. E. Emam, D. Zhang, H. Liu and D. Mathew, Theory of relative defect proneness: Replicated studies on the functional form of the size-defect relationship, *Empirical Software Engineering*, October 2008 473–498.
23. A. Koru, D. Zhang and H. Liu, Modeling the effect of size on defect proneness for open-source software, in *International Workshop on Predictor Models in Software Engineering* (*PROMISE'07*), May 2007, Article No. 10.
24. G. Boetticher, T. Menzies and T. Ostrand, The PROMISE Repository of Empirical Software Engineering Data, 2007, http://promisedata.org/.
25. M. Jureczko and D. Spinellis, Using object-oriented design metrics to predict software defects, *Models and Methods of System Dependability, Oficyna Wydawnicza Politechniki Wroc awskiej*, 2010, pp. 69–81.
26. J. Krinke, Identifying similar code with program dependence graphs, in *Proceedings of the Eighth Working Conference on Reverse Engineering* (*WCRE'01*). IEEE Computer Society, October 2001, pp. 301–309.
27. G. Antoniol, G. Canfora, G. Casazza and A. De Lucia, Information retrieval models for recovering traceability links between code and documentation, in *Proceedings of the International Conference on Software Maintenance* (*ICSM'00*), 2000, pp. 40–49.
28. K. Kontogiannis, R. DeMori, E. Merlo, M. Galler and M. Bernstein, Pattern matching for clone and concept detection, *Journal of Automated Software Engineering* **3** (1996) 77–108.
29. J. H. Johnson, Substring matching for clone detection and change tracking, in *Proceedings of the International Conference on Software Maintenance* (*ICSM 94*), 1994, pp. 120–126.
30. I. H. Witten and E. Frank, *Data Mining*, 2nd edn. (Morgan Kaufmann, Los Altos, 2005).
31. T. McCabe, A complexity measure, *IEEE Trans. Software Engineering* **2**(4), (1976), 308–320.
32. S. Lessmann, B. Baesens, C. Mues and S. Pietsch, Bench-marking classification models for software defect prediction: A proposed framework and novel findings, *IEEE Trans. Software Engineering* **34**(4) (2008) 485–496.
33. T. M. Khoshgoftaar, N. Seliya and K. Gao, Assessment of a new three-group software quality classification technique: An empirical case study, *Empirical Software Engineering* **10** (2005) 183–218.
34. T. M. Khoshgoftaar, S. Zhong and V. Joshi, Enhancing software quality estimation using ensemble-classifier based noise filtering, *Intell. Data Anal.* **9** (2005) 3–27.
35. T. M. Khoshgoftaar, X. Yuan and E. B. Allen, Balancing misclassification rates in classification-tree models of software quality, *Empirical Software Engineering* **5** (2000) 313–330.
36. R. Harrison, S. Counsell and R. Nithi, An investigation into the applicability and validity of object-oriented design metrics, *Empirical Software Engineering* **3**(3) (1998) 255–273.
37. L. C. Briand, S. Morasca and V. R. Basili, Defining and validating measures for object-based high-level design, *IEEE Transactions on Software Engineering*, **25**(5), (1999) 722–743.
38. L. Briand, J. Wüst, J. Daly and D. Victor Porter, Exploring the relationships between design measures and software quality in object-oriented systems, *Journal of Systems and Software* **51**(3) (2000) 245–273.

39. N. Nagappan, T. Ball and A. Zeller, Mining metrics to predict component failures, in *Proceedings of the 28th International Conference on Software Engineering*, ACM, May 2006, pp. 452–461.
40. B. Turhan, T. Menzies, A. Bener and J. Di Stefano, On the relative value of cross-company and within-company data for defect prediction, *Empirical Software Engineering* **14**(5) (2009) 540–578.
41. T. Menzies, O. Jalali, J. Hihn, D. Baker and K. Lum, Stable rankings for different effort models, *Automated Software Engineering* **17**(4) (2010) 409–437.
42. R. Holte, Very simple classification rules perform well on most commonly used datasets, *Machine Learning* **11** (1993) 63–90.
43. V. R. Basili, L. C. Briand and W. L. Melo, A validation of object-oriented design metrics as quality indicators, *IEEE Trans. Software Engineering* **22**(10) (1996) 751–761.
44. L. C. Briand, J. Wüst, S. V. Ikonomovski and H. Lounis, Investigating quality factors in object-oriented designs: An industrial case study, in *Proceedings of 21st International Conference of Software Engineering*, May 1999, pp. 345–354.
45. M. Cartwright and M. Shepperd, An empirical investigation fan object-oriented software system, *IEEE Trans. Software Engineering* **26** (2000) 786–796.
46. 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. Software Engineering* **27**(7) (2001) 630–650.
47. K. K. Aggarwal, Y. Singh, A. Kaur and R. Malhotra, Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: A replicated case study, *Software Process: Improvement and Practice* **14**(1) (2009) 39–62.
48. L. C. Briand and J. Wüst, Empirical studies of quality models in object-oriented systems, *Advances in Computers* **56** (2002) 98–167.
49. K. E. Emam, W. Melo and J. C. Machado, The prediction of faulty classes using object-oriented design metrics, *System and Software* **56** (2001) 63–75.
50. L. C. Briand, W. L. Melo and J. Wüst, Assessing the applicability of fault-proneness models across object-oriented software projects, *IEEE Transactions on Software Engineering* **28**(7) (2002) 706–720.
51. K. K. Aggarwal, Y. Singh, A. Kaur and R. Malhotra, Investigating the effect of coupling metrics on fault proneness in object-oriented systems, *Software Quality Professional* **8**(4) (2006) 4–16.
52. ———, Investigating effect of design metrics on fault proneness in object-oriented systems, *Journal of Object Technology* **6**(10) (2007) 127–141.
53. B. Turhan, G. Kocak and A. Bener, Software defect prediction using call graph based ranking (CGBR) framework, in *Proceedings of 34th Euromicro Conference on Software Engineering and Advanced Applications*, September 2008, pp. 191–198.
54. S. Brin and L. Page, The anatomy of a large-scale hypertextual Web search engine, in *Proceedings of the 7th International Conference on World Wide Web*, April 1998, pp. 107–117.
55. T. Zimmermann and N. Nagappan, Predicting defects using network analysis on dependency graphs, in *Proceedings of the 30th International Conference on Software Engineering, (ICSE'08)*, New York, NY, USA, 2008, pp. 531–540.
56. S. Kpodjedo, F. Ricca, G. Antoniol and P. Galinier, Evolution and search based metrics to improve defects prediction, *International Symposium on Search Based Software Engineering* 2009, pp. 23–32.
57. A. Miller, *Subset Selection in Regression*, 2nd edn. (Chapman & Hall, 2002).

58. S. Wagner, Global sensitivity analysis of predictor models in software engineering, in *International Workshop on Predictor Models in Software Engineering* (*PROMISE'07*), May 2007, Article No. 3.
59. I. Guyon, A. Elisseefi and L. Kaelbling, An introduction to variable and feature selection, *Journal of Machine Learning Research* **3**(7−8) (2003) 1157−1182.
60. T. J. Ostrand, E. J. Weyuker and R. M. Bell, Where the bugs are, in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, July 2004, pp. 86−96.
61. E. Weyuker, T. Ostrand and R. Bell, Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models, *Empirical Software Engineering* **13**(5) (2008) 539−559.
62. H. Zhang and X. Zhang, Comments on data mining static code attributes to learn defect predictors, *IEEE Trans. Software Engineering* **33**(9) (2007) 635−637.
63. T. Menzies, A. Dekhtyar, J. Distefano and J. Greenwald, Problems with precision: A response to comments on data mining static code attributes to learn defect predictors, *IEEE Trans. Software Engineering* **33**(9) (2007) 637−640.
64. K. Sunghun, T. Zimmermann, E. J. Whitehead Jr. and A. Zeller, Predicting faults from cached history, in *Proceedings of the 29th International Conference on Software Engineering* (*ICSE'07*), Washington, DC, USA, 2007, pp. 489−498.
65. F. Rahman, D. Posnett, A. Hindle, E. Barr and P. Devanbu, BugCache for inspections: Hit or miss? in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (*ESEC/FSE'11*), Szeged, Hungary, September 2011, pp. 322−331.
66. W. Li and S. Henry, Object-oriented metrics that predict maintainability, *Journal of Systems and Software* **23**(2) (1993) 111−122.
67. P. Cohen, *Empirical Methods for Artificial Intelligence* (MIT Press, 1995).
68. M. Lumpe, L. Grunske and J.-G. Schneider, State space reduction techniques for component interfaces, in *Proceedings of 11th International Symposium on Component-Based Software Engineering* (*CBSE 2008*), LNCS 5282, October 2008, pp. 130−145.
69. M. Lumpe and R. Vasa, Partition refinement of component interaction automata: Why structure matters more than size, in *Electronic Proceedings in Theoretical Computer Science* **37** (2010) 12−26.