Recommendations For Intelligent Code Reorganization

Rahul Krishna[a,*], Tim Menzies[a,*], Lucas Layman[b]

[a] Department of Computer Science, North Carolina State University, Raleigh, NC, USA
[b] Fraunhofer CESE, College Park, USA

Abstract

Context: Developers use bad code smells to guide code reorganization. Yet developers, text books, tools, and researchers disagree on which bad smells are important.
Objective: To evaluate the likelihood that a code reorganization to address bad code smells will yield improvement in the defect-proneness of the code.
Method: We introduce XTREE, a tool that analyzes a historical log of defects seen previously in the code and generates a set of useful[1] code changes. Any bad smell that requires changes outside of that set can be deprioritized (since there is no historical evidence that the bad smell causes any problems).
Evaluation: We evaluate XTREE's recommendations for bad smell improvement against recommendations from previous work (Shatnawi, Alves, and Borges) using multiple data sets of code metrics and defect counts.
Results: Code modules that are changed in response to XTREE's recommendations contain significantly fewer defects than recommendations in previous studies. Further, XTREE endorses changes to very few code metrics, and those bad smell recommendations (learned from previous studies) are not universal to all software projects.
Conclusion: Before undertaking a code reorganization based on a bad smell report, use a tool like XTREE to check and ignore any code reorganizations that are useless; i.e. which lacks evidence in the historical record that it is useful to make that change. Note that this use case applies to both manual code reorganizations proposed by developers as well as refactoring conducted by automatic methods. This recommendation assumes that there is an historical record. If none exists, then the results of this paper could be use used as a guide (see our Table 8).

Keywords: Bad smells, performance prediction, decision trees

1. Introduction

According to Fowler [1], bad smells (a.k.a. code smells) are "a surface indication that usually corresponds to a deeper problem". Fowler strongly recommends removing code smells by

> "...applying a series of small behavior-preserving transformations, each of which seem "too small to be worth doing". The effect of these refactoring transformations is quite significant. By doing them in small steps you reduce the risk of introducing errors".

Bad smells are a widely used technique for evaluating code. While the original bad small concept was largely subjective [1], researchers including Marinescu [2] and others has provided formal definitions of "bad smells" in terms of static code attributes such size, complexity, coupling, and other metrics. Consequently, code smells are captured by popular static analysis tools such as PMD[2], CheckStyle[3], FindBugs[4], and SonarQube[5], although not always explicitly labeled as such.

Refactoring bad smells from code is generally accepted as a recommended practice, but code reorganization by definition does not change the external functionality of the system and thus may have no perceived value. For example, this paper began when a Washington-based software company shared one of their management challenges with us: their releases are delayed by developers spending much time removing bad smells within their code. Kim et al.[3] surveyed developers at Microsoft and found that refactoring incurs significant cost and risks. A Google search on the terms "refactoring cost" yields numerous threads on StackOverlfow from developers asking how to justify refactoring and when the costs of refactoring outweight its benefits.

The challenge is to encourage code reorganizations that result in perceivable benefits (such as reduced defect proneness or lower maintenance costs) and avoid code reorganizations with no demonstrable benefit and thus waste effort.

This paper describes one approach to evaluate whether code

[*] Corresponding author: Tel:+1-919-396-4143(Rahul)
Email addresses: rkrish11@ncsu.edu (Rahul Krishna[a,]), tim.menzies@gmail.com (Tim Menzies[a,]), llayman@cese.fraunhofer.org (Lucas Layman[b])

[2] https://github.com/pmd/pmd
[3] http://checkstyle.sourceforge.net/
[4] http://findbugs.sourceforge.net/
[5] http://www.sonarqube.org/

reorganizations that change the structure of code are likely to have a perceivable benefit by evaluating:

- If there is no evidence in the historical record that changing "X" is useful;

- Then discourage developers from doing "X".

We use data mining to generate two oracles: a *primary change oracle* and a *secondary verification oracle*. By combining these two oracles, we can generate and validate useful changes to code modules. We focus on "bad smells" indicated by code metrics such as size and complexity as captured in popular tools such as SonarQube and Klockwork[6].

In this paper, we use the XTREE cluster delta algorithm as the *primary change oracle*. XTREE explores the historical record of a project to find clusters of module. It then proposes a "minimal" set of changes Δ that can move a software module M from a defective cluster C_0 to another with fewer defects C_1 (so Δ is some subset of $C_1 - C_0$).

As to the *secondary verification oracle*, this checks if the primary oracle is proposing sensible changes. For this purpose we use Random Forest [4], augmented with SMOTE (synthetic minority over-sampling technique [5]). In our framework, learning the secondary oracle is a *separate* task from that of learning the primary oracle. This ensures that the verification oracle offers an independent opinion on the value of the proposed changes.

When these oracles were applied to defect data from open source object-oriented software systems, we discovered a *conjunctive fallacy* in the bad smell literature [6, 7, 8, 9, 10]. A common heuristic is to recommend code reorganization of "bad" code to make it "better" as follows: for all static code measures that exceed some threshold, make changes such that the thresholds are no longer exceeded. That is:

$$\begin{aligned} bad &= (a_1 > t_1) \bigvee (a_2 > t_2) \bigvee ... \\ better &= \neg\, bad = (a_1 \leq t_1) \bigwedge (a_2 \leq t_2) \bigwedge ... \end{aligned} \tag{1}$$

We say that the above definition of "better" is a conjunctive fallacy since it assumes that the best way to improve code is to decreases multiple code attribute measures below t_i in order to remove the "bad" smells. In reality, there exists an intricate set of associations between static code measures such that *decreasing* a_i necessitates *increasing* a_j. It is easy to see why this is so. A requirement for code reorganization is that the new code supports the same functionality as before. If we pull code out of a function (since that function has grown too large), *that functionality has to go somewhere else*. What we see in XTREE's change recommendation is that if we decrease a module's lines of code then in the refactored code, the functionality is achieved via a communication between several different parts of the system. That is, *decreasing* the lines of code metric within a module *increases* that module's coupling metrics.

1.1. Structure of this Paper

This paper claims that (a) XTREE is a better way than Equation 1 to define useful changes to software and (b) those changes are useful for recognizing superfluous code reorganizations since they usually refer to a very small subset of the static code measures (which allows us to rule out code reorganizations based on any of the omitted static code measures). In support of these claims, the rest of this paper is structured as follows. In Section 2, we offer some background on bad smells. In Section 3, different methods are presented for determining when code does/does not have bad smell. One of those methods will be XTREE while the others will come from other researchers. Section 4 defines an assessment method that checks which of these methods is most useful. The results from that assessment are presented in Section 5. Those results will answer five research questions.

RQ1: Effectiveness[7]: According to the verification oracle, which of the methods defined in Section 3 is the best change oracle for identifying what and how code modules should be changed? To answer this question, we used data from five OO Java projects (Ivy, Lucene, Ant, Poi, Jedit). It was found that:

Result 1
XTREE is the most accurate oracle on how to change code modules in order to reduce defects.

RQ2: Succinctness: Our goal is to critique and, possibly, ignore irrelevant bad smell detectors. If those changes are minimal (i.e. affect fewest attributes) then those changes will be easiest to apply and monitor. Hence, we ask, which of the Section 3 methods recommended changes to the fewest code attributes?

Result 2
Of all the code change oracles studied here, XTREE recommends the fewest number of changes to static code measures.

RQ3: Stopping: Our advice to managers is to discourage code reorganization based on changes that lack historical evidence of being effective. How effective is XTREE at offering such "stopping points" (i.e. clear guidance on what *not* to do)??

Result 3
In any project, XTREE's recommended changes only mentions one to four of the static code attributes. Any bad smell defined in terms of the remaining 19 to 16 code attributes (i.e. most of them) would hence be deprecated.

RQ4: Stability: Across different projects, how variable are the changes recommended by our best change oracle?

[6] http://www.klocwork.com/

[7] test

Result 4
The direction of change recommended by XTREE is very stable across repeated runs of the program

RQ5: Conjunctive Fallacy: Is it always useful to aplly Equation 1; i.e. make code better by reducing the values of multiple code attributes? We find that:

Result 5
XTREE usually recommends reducing lines of code (size of the modules). That said, XTREE often recommends increasing *the values of other static code attributes.*

Note that **RQ3, RQ4, RQ5** confirms the intuitions of the project managers that prompted this investigation:

- Yes, indeed, their programmers were wasting time on performing code reorganizations that decreased multiple measures when, in fact, they should be trying to *decrease* some measures while *increasing* others.
- Across all studied projects, XTREE found that around one-fifth of the measures were usually found within change recommendations. That is, it is 80% likely that that code reorganization based on reducing any one measurement picked at random will be useful.

Consequently, we recommend the following use case for XTREE:

- Before doing code reorganization based on a bad smell report...
- ...check and discourage any code reorganization for which there is no proof in the historical record that the change improves the code.

This use case described applies to both manual code reorganizations proposed by developers as well as refactoring conducted by automatic methods [11]. That is, XTREE could optimize automatic code refactoring by discouraging refactorings for useless goals.

1.2. Relationship to Prior Work

A four page preliminary report on the XTREE system [12] has been presented previously[8]. That short report offered case studies on only two of the five data sets studied here. Also, that prior work had:

- no comparison to any baseline method from other researchers, whereas this paper compares XTREE's recommendations with those found by researchers exploring bad smells.
- no *secondary verification oracle*, i.e. no way to check if XTREE's recommended changes were sensible.

Further, of this entire paper, the only sections containing material found in prior papers is 3.2.1, 3.2.2 as well as two-fifths of the results in Figure 5.

2. Why Not Just Ask Developers to Rank Bad Smells?

Why build tools like XTREE to critique proposed developer actions? Our answer, documented in this section, is that (1) SE literature is not clear which bad smells are ignorable or high priority and must be resolved. Also, (2) developers themselves are unclear on what bad smells are most important to fix.

Much research endorses them to guide code improvement (e.g., code reorganization or preventative maintenance). A recent literature review by Tufano et al. [13] lists dozens of papers on smell detection and repair tools. Yet other papers cast doubt on the value of bad smells as triggers for code improvement [14, 15, 16]. As we show in this paper, tools, text books, and developers disagree on what bad smells are important enough to hunt down and eliminate. Our findings are consistent with other research suggesting that universal bad smells that ignore project context are not useful to guide refactoring [14, 15, 16].

If the SE literature is contradictory, why not ignore it and use domain experts (software engineers) to decide what bad smells to fix? We do not recommend this since developer *cognitive biases* can mislead them to assert that some things are important and relevant when they are not. A widespread malaise in software engineering is that beliefs about software are rarely revised and hence may be inaccurate and misleading [17, 18, 19, 20, 21]. Software developers may have strong views on many issues, including bad smells, but those views may be wrong for the current project. Software engineering is not the only field with this problem. For example, the medical profession applies many practices based on studies that have been disproved (a recent article in the Mayo Clinic Proceedings [22] found 146 medical practices based on studies in year i, but which were reversed by subsequent trials within years i + 10). Even when the evidence for or against a treatment or intervention is clear, medical providers and patients may not accept it [23]. Aschwanden warns that *cognitive biases* such as confirmation bias (the tendency to look for evidence that supports what you already know and to ignore the rest) influence how we process information [24].

As in medicine, so too in software engineering. According to Passos et al. [17], developers often assume that the lessons they learn from a few past projects are general to all their future projects. They comment "past experiences were taken into account without much consideration for their context" [17]. Jørgensen & Gruschke [18] offer a similar warning. They report that the supposed software engineering "gurus" rarely use lessons from past projects to improve their future reasoning and that such poor past advice can be detrimental to new projects. [18]. Other studies have shown some widely-held views are now questionable given new evidence:

- In other work [20] with the Software Engineering Institute (SEI), we have revisited the "phase delay" truism that *the cost of fixing an error dramatically increases the longer it is in the system.* We found no evidence for such large phase delay effect in modern software (in 171 software projects shepherded by the SEI, 2006-2014) possibly since it has been

[8] https://goo.gl/2In3Lr

Fowler'99 [1] and [25]	Lanza'06 [26]	SonarQube [27]	Yamashita'13[15]	Developer Survey 2015
Alt. Classes with Diff. Interfaces				
Combinatorial Explosion [25]				
Comments			11	VL
Conditional Complexity [25]			14	?
Data Class	✓			
Data Clumps				
Divergent Change				
Duplicated Code	✓	✓	1	VH
Feature Envy	✓		8	
Inappropriate Intimacy		✓		L
Indecent Exposure [25]				?
Incomplete Library Class				
Large Class	✓	✓	4	VH
Lazy Class/Freeloader		✓	7	
Long Method	✓	✓	2	VH
Long Parameter List		✓	9	L
Message Chains				H
Middle Man				
Oddball Solution [25]				
Parallel Inheritance Hierarchies				
Primitive Obsession				
Refused Bequest	✓	✓		
Shotgun Surgery	✓			
Solution Sprawl [25]				
Speculative Generality				L
Switch Statements				L
Temporary Field		✓		?

Figure 1: Bad smells from different sources. Check marks (✓) denote a bad smell was mentioned. Numbers or symbolic labels (e.g. "VH") denote a priorization comment (and "?" indicates lack of consensus). Empty cells denote some bad smell listed in column one that was not found relevant in other studies. Note: there are many blank cells.

heavily mitigated by 21^{st} century software development languages, tools and development practices. Whatever the reason, the main point is that phase delay is a widely held belief, despite little recent evidence to support it.

- Devanbu et al. examined responses from 564 Microsoft software developers from around the world, they found that "(a) programmers do indeed have very strong beliefs on certain topics; (b) their beliefs are primarily formed based on personal experience, rather than on findings in empirical research; (c) beliefs can vary with each project, but do not necessarily correspond with actual evidence in that project" [21].

Devanbu et al. further comment that "programmers give personal experience as the strongest influence in forming their opinions". This is a troubling result, especially given the above comments from Passos and Jørgensen et al. [17, 18] about how quickly practitioners form, freeze, and rarely revist those opinions.

If the above remarks hold true for bad smells, then we would expect to see much disagreement on which bad smells are important and relevant to a particular project. This is indeed the case. The first column of Figure 1 lists commonly mentioned bad smells and comes from Fowler's 1999 text [1] and a subsequent 2005 text by Kerievsky that is widely cited [25]. The other columns show data from other studies on which bad smells matter most. The columns marked as Lanza'06 and Yamashita'13 are from peer reviewed literature. The column maked SonarQube is a popular open source code assessment tool that includes detectors for six of the bad smells in column one. The *developer survey* (in the right-hand-side column) shows the results of an hour-long whiteboard sessions with a group of 12 developers from a Washington D.C. web tools development company. Participants worked in a round robin manner to rank the bad smells they thought were important (and any disagreements were discussed with the whole group). Amongst the group, there was some consensus on the priority of which bad smells to fix (see the annotations VH=very high, H=high, L=low, VL=very low, and "?"= no consensus).

A blank cell in Figure 1 indicates where other work has chosen to ignore one of the bad smells in column one. Note that most of the cells are blank, and that the studies ignore the majority of the Fowler bad smells. SonarQube has no detectors for many of the column one bad smells. Also, nearly half the Yamashita list of bad smells does not appear in the Fowler and Kerievsky list. The eight numbers in the Yamashita'13 column show the rankings for the bad smells that overlap with Fowler and Kerievsky; Yamashita also discussed other smells not covered in Fowler'99.

Two of the studies in Figure 1 offers some comments on the relative importance of the different bad smells in the Yamashita'13 study and the developer study. Three of the bad smells listed in the top half of the Yamashita'13 rankings also score very high in the developer survey. Those three were *duplicated code, large class*, and *long method*. Note that this agreement also means that the Yamashita'13 study and the developer survey believe that very few code smells are high priority issues requiring code reorganization.

In summary, just because one developer strongly believes in the importance of a bad smells does not mean that belief transfers to other developers or projects. Developers can be clever, but their thinking can also be distorted by cognitive biases. Hence, as shown in Figure 1, developers, text books, and tools can disagree on which bad smells are important. Special tools are needed to assess their beliefs, for example, their beliefs in bad smells.

3. Learning Bad Smell Thresholds

Having made the case for automatic support for assessing bad smells, this section reviews different ways for building those tools (one of those tools, XTREE, will be our recommended *primary change oracle*). Later in this paper, we will offer a *secondary verification oracle* that checks the effectiveness of the changes proposed by XTREE.

The SE literature offers two ways of learning bad smell thresholds. One approach relies on *outlier statistics* [6, 7]. This approach has been used by Shatnawi [8], Alvenes et al. [9] and Hermans et al. [10]. Another approach is based on *cluster deltas* that we developed for Centroid Deltas [28] and use here for XTREE. These two approaches are discussed below.

amc	average method complexity	e.g. number of JAVA byte codes
avg_cc	average McCabe	average McCabe's cyclomatic complexity seen in class
ca	afferent couplings	how many other classes use the specific class.
class.		
cam	cohesion amongst classes	summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods.
cbm	coupling between methods	total number of new/redefined methods to which all the inherited methods are coupled
cbo	coupling between objects	increased when the methods of one class access services of another.
ce	efferent couplings	how many other classes is used by the specific class.
dam	data access	ratio of the number of private (protected) attributes to the total number of attributes
dit	depth of inheritance tree	
ic	inheritance coupling	number of parent classes to which a given class is coupled (includes counts of methods and variables inherited)
lcom	lack of cohesion in methods	number of pairs of methods that do not share a reference to an case variable.
locm3	another lack of cohesion measure	if m,a are the number of $methods, attributes$ in a class number and $\mu(a)$ is the number of methods accessing an attribute, then $lcom3 = ((\frac{1}{a}\sum, j^a \mu(a,j)) - m)/(1-m)$.
loc	lines of code	
max_cc	maximum McCabe	maximum McCabe's cyclomatic complexity seen in class
mfa	functional abstraction	number of methods inherited by a class plus number of methods accessible by member methods of the class
moa	aggregation	count of the number of data declarations (class fields) whose types are user defined classes
noc	number of children	
npm	number of public methods	
rfc	response for a class	number of methods invoked in response to a message to the object.
wmc	weighted methods per class	
nDefects	raw defect counts	Numeric: number of defects found in post-release bug-tracking systems.
defect	defects present?	Boolean: if *nDefects* > 0 then *true* else *false*

Figure 2: OO code metrics used for all studies in this paper. Last lines, shown in denote the dependent variables.

3.1. Outlier Statistics

The outlier approach assume that unusually large measurements indicate risk-prone code. Hence, they generate one bad smell threshold for any metric with such an "unusually large" measurement. The literature lists several ways to define "unusually large".

3.1.1. Enri & Lewerentz

Given classes described with the code metrics of Figure 2, Enri and Lewerentz [6] found the mean μ and the standard deviation σ of each code metrics. Their definition of problematic outlier was any code metric with a measurement greater than $\mu + \sigma$. Shatnawi and Alves et al. [8, 9] depreciate using $\mu + \sigma$ since it does not consider the fault-proneness of classes when the thresholds are computed. Also, the method lacks empirical verification.

3.1.2. Shatnawi

Shatnawi [8]'s preferred alternative to $\mu + \sigma$ is to use the VARL method (Value of Acceptable Risk Level) proposed by Bender [7] in his epidemiology studies. This approach uses two constants to compute the thresholds which, following Shatnawi's guidance, we set to $p_0 = p_1 = 0.05$.

VARL encodes the defect count for each class as 0 (no defects known in class) or 1 (defects known in class). Univariate binary logistic regression is applied to learn three coefficients: α is the intercept constant; β is the coefficient for maximizing log-likelihood; and p_0 measures how well this model predicts for defects. A univariate logistic regression was conducted comparing metrics to defect counts. Any code metric with $p > 0.05$ is ignored as being a poor defect predictor. Thresholds are then learned from the surviving metrics M_c using an equation proposed by Bender:

$$bad\ smell\ if\ M_c > VARL$$

Where,

$$VARL = p^{-1}(p_0) = \frac{1}{\beta}\left(\log\left(\frac{p_1}{1-p_1}\right) - \alpha\right) \tag{2}$$

3.1.3. Alves et al.

Alves et al. [9] propose another approach to finding thresholds that utilizes the underlying statistical distribution and scale of the metrics. Metric values for each class are weighted according to the source lines of code (SLOC) of the class. All the weighted metrics are then normalized by the sum of all weights of for the system. The normalized metric values are ordered in an ascending fashion (this is equivalent to computing a density function, in which the x-axis represents the weight ratio (0-100%), and the y-axis the metric scale). Alves et al. then select a percentage value (they suggest 90%) which represents the "normal" values for metrics. The metric threshold, then, is the metric value for which 90% of the classes fall below. The intuition is that the worst code has outliers beyond 90% of the normal code measurements. Hermans et al. [10] used the Alves et al. method in their 2015 paper on exploring bad smells. For consistency with Shatnawi, we explore the correlation between the code metrics and the defect counts and reject code metrics that are poor predictors of defects (i.e. those with $p > 0.05$).

3.1.4. Discussion of Outlier Methods

The advantage of the outlier-based approaches is that they are simple to implement, but the approaches have two major disadvantages. First, they are *verbose*. A threshold can be calculated for every metric – so, which one should the developers focus on changing? Without a means for prioritizing the thresholds and metrics against one another, developers may have numerous or conflicting recommendations on what to improve. Second, the outlier approaches suffers from *conjunctive fallacy* discussed in the introduction. That is, while they propose thresholds for many code metrics individually, they make no comment on what minimal metrics need to be changed at the same time (or whether or not those changes lead to minimization or maximization of static code measures).

3.2. Cluster Deltas

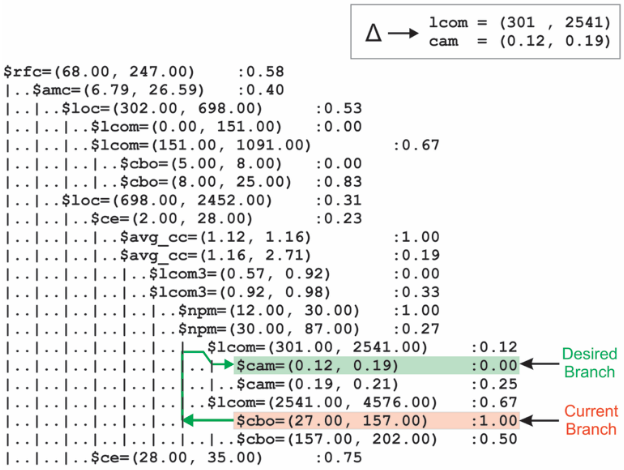
Cluster deltas are a general method for learning *conjunctions* of changes that need to be applied at the same time. This ap-

On the right-hand-side is a tree generated by iterative dichomization. This tree can be read like a nested if-then-else statement; e.g.

- Lines 3 and 8 show two branches for lines of code (denoted here as '*$loc*) below 698 and above 698.
- Any line with a colon ":" character shows a leaf of this nesting. For example, if some new code module is passed down this tree and falls to the line marked in **orange**, the colon on that line indicates a prediction that this module has a 100% chance of being defective.

Using this tree, XTREE looks for a nearby branch that has a lower chance of being defective. Finding the green desired branch, XTREE reports a bad smell threshold for that module that is the delta between the **orange** current branch and green designed branch.[a] In this case, that threshold relates to:

- Lines of code and comments (*lcom*)
- The cohesion between classes (*cam*) which measures similarity of parameter lists to assess the relatedness amongst class methods.



```
$rfc=(68.00, 247.00)      :0.58
|..$amc=(6.79, 26.59)     :0.40
|..|..$loc=(302.00, 698.00)       :0.53
|..|..|..$lcom=(0.00, 151.00)     :0.00
|..|..|..$lcom=(151.00, 1091.00)           :0.67
|..|..|..|..$cbo=(5.00, 8.00)     :0.00
|..|..|..|..$cbo=(8.00, 25.00)    :0.83
|..|..$loc=(698.00, 2452.00)      :0.31
|..|..|..$ce=(2.00, 28.00)        :0.23
|..|..|..|..$avg_cc=(1.12, 1.16)           :1.00
|..|..|..|..$avg_cc=(1.16, 2.71)           :0.19
|..|..|..|..|..$lcom3=(0.57, 0.92)         :0.00
|..|..|..|..|..$lcom3=(0.92, 0.98)         :0.33
|..|..|..|..|..|..$npm=(12.00, 30.00)      :1.00
|..|..|..|..|..|..$npm=(30.00, 87.00)      :0.27
|..|..|..|..|..|..|..$lcom=(301.00, 2541.00)       :0.12
|..|..|..|..|..|..|..|..$cam=(0.12, 0.19)          :0.00
|..|..|..|..|..|..|..|..$cam=(0.19, 0.21)          :0.25
|..|..|..|..|..|..|..$lcom=(2541.00, 4576.00)      :0.67
|..|..|..|..|..|..|..$cbo=(27.00, 157.00)          :1.00
|..|..|..|..|..|..|..|..$cbo=(157.00, 202.00)      :0.50
|..|..|..$ce=(28.00, 35.00)       :0.75
```

[a] But XTREE makes no recommendations for CBO even though it is in the organge branch. Is this because there is nothing about CBO in the desired branch? Should there be?

Figure 3: A brief tutorial on XTREE.

proach works as follows:

- Cluster the data.
- Find neighboring clusters C_+, C_- where C_+ has more examples of defective modules than C_-;
- Compute the delta in code metrics between the clusters using $\Delta = C_- - C_+ = \{\delta | \delta \in C_-, \delta \notin C_+\}$, i.e. *towards* the cluster with lower defects;
- The set Δ are changes needed in defective modules of C_+ to make them more like the less-defective modules of C_-

Note that Δ is a conjunction of recommendations. Since it is computed from neighboring clusters, the examples contain similar distributions and Δ respects the naturally occurring constraints in the data. For example, given a bad smell pertaining to large methods, Δ will not suggest lowering lines of code without also increasing a coupling measure. Cluster deltas are used in CD [28] and XTREE.

3.2.1. CD

Borges and Menzies first proposed the CD centroid delta approach to generate *conjunctions* of code metrics that need to be changes at the same time in order to reduce defects [28]. CD used the WHERE clustering algorithm developed by the authors for a prior application [29]. Each cluster was then replaced by its centroid and Δ was calculated directly from the difference between code metrics values between one centroid and its nearest neighbor.

As shown below, one drawback with with CD is that it is *verbose* since CD recommended changes to all code metrics with different values in those two centroids. This makes it hard to use CD to critique and prune away bad smells. Further, CD will be shown to be not as effective in proposing changes to reduce defects as XTREE.

3.2.2. XTREE: Overview

XTREE is a cluster delta algorithm that avoids the problem of verbose Δs. XTREE is our *primary change oracle* that makes recommendations of what changes should be made to code modules. Instead of reasoning over cluster centroids, XTREE utilizes a decision tree learning approach to find the fewest differences between clusters of examples.

XTREE uses a multi-interval discretizer based on an iterative dichotomization scheme, first proposed by Fayyad and Irani [30]. This method converts the values for each code metric into a small number of nominal ranges. It works as follows:

- A code metric is split into r ($r = 2$) ranges, each range is of size n_r and is associated with a set of defect counts x with standard deviation σ_r.
- The best split for that range is the one that minimizes the expected value of the defect variance, after the split; i.e. $\sum_r \frac{n_r}{n} \sigma_x$ (where $n = \sum_r n_r$).
- This discretizer then recurses on each part of the split to find other splits in a recursive fashion. As suggested by Fayyad and Irani, minimum descriptor length (MDL) is used as a termination criterion for the recursive partitioning.

When discretization finishes, each code metric M has a final expected value M_v for the defect standard deviation across all the discretized ranges of that metric. Iterative dichomization sorts the metrics by M_v to find the code metric that best splits the data i.e., the code metric with smallest M_v.

A decision tree is then constructed on the discretized metrics. The metric that generated the best split forms the root of the tree with its discrete ranges acting as the nodes.

When all the metrics are arranged this way, the process is very similar to a hierarchical clustering algorithm that groups together code modules with similar defect counts and some shared ranges of code metrics. For our purposes, we score each cluster found in this way according to the percent of classes with known defects. For example, the last line of Figure 3 shows a tree leaf with 75% defective modules.

Figure 3 offers a small example of how XTREE builds Δ by comparing branches that lead to leaf clusters with different defect percentages. In this example, assume a project with a table of code metrics data describing its classes in the form of Figure 2. After code inspections and running test cases or operational tests, each such class is augmented with a defect count. Iterative dichomization takes that table of data and, generates the tree of Figure 3.

Once the tree is built, a class with code metric data is passed into the tree and evaluated down the tree to a leaf node (see the **orange** line in Figure 3). XTREE then looks for a nearby leaf node with a lower defect count (see the green line in Figure 3). For that evaluated class, XTREE proposes a bad smell thresholds that is the difference between green and **orange**.

3.2.3. XTREE: Details

Using the training data construct a decision tree as suggested above.

Next, for each test code module, find C_+ as follows: take each test, run it down the decision tree to find a leaf in the decision tree that mostly matches the test case. After that, find C_- as follows:

- Starting at the C_+ leaf, ascend $lvl \in \{0,1,2...\}$ tree levels;
- Identify *sibling* leaves; i.e. leaf clusters that can be reached from level lvl that are not same as *current* C_+;
- Find the *better* siblings; i.e. those 50% (or less) fewer defects than C_+. If none found, then repeat for $lvl+=1$. Also, return `nil` if the new lvl is above the root.
- Set C_- to the *closest* better sibling where distance is measured between the mean centroids of that sibling and *current*

Now find $\Delta = C_- - C_+$ by reflecting on the set difference between conditions in the decision tree branching from C_+ to C_-. To find that delta, for discrete attributes, delta is the value of the *desired*; for numerics expressed as ranges, the delta could be any value that lies between $[LOW, HIGH]$ in that range (random number selected from the low and high boundaries of the that range.)

4. Evaluation

The previous section proposed numerous methods for detecting bad smells that need to be resolved. This section offers an experiment of the following form:

- Using each method as a *primary change oracle* to recommend how code should be changed.
- Apply those changes.
- Run a *secondary verification oracle* to assess the defect-proness of the changed code.
- Sort the change oracles on how well they reduce defects (as judged by the verification oracle).

Using this experiment, we can address the research questions dicsussed in the introduction.

RQ1: Effectiveness: which of the methods defined in Section 3 is the best change oracle for identifying what and how code modules should be changed? To answer to this question, we will assume that developers refactor their code until the bad smell thresholds are not violated. This code reorganization will start with some *initial* code base that is changed to a *new* code base. For example, if the bad smell is $loc > 100$ and a code module has 500 lines of code, we reason optimistically that we can change that code metric to 100. Using the secondary verification oracle, we then predict the number of defects in d_+, d_- in *initial* and *new*. We evaluate the performance of XTREE, CD, Shatnawi, and Alves methods in setting bad smell thresholds. The best bad smell threshold method is the one that maximizes

$$improvement = 100 * \left(1 - \frac{d_-}{d_+}\right) \tag{3}$$

RQ2: Succinctness: which of the Section 3 methods recommended changes to the fewest code attributes? To answer this question, we will report the frequency at which different attributes are selected in repeated runs of our oracles.

RQ3: Stopping: How effective is XTREE at offering "stopping points" (i.e. clear guidance on what *not* to do)? To answer this point, we will report how often XTREE's recommendations *omit* a code attribute. Note that the *more often* XTREE omits an attribute, the more likely it is *not* to endorse a bad smell based on the omitted attribites.

RQ4: Stability: Across different projects, how variable are the changes recommended by XTREE? To answer this question, we conduct a large scale "what if" study that reports all the possible recommendations XTREE might make. We then count how often attributes are *not* found in the recommendations arising from this "what if" study.

RQ5: Conjunctive Fallacy: Is it always useful to apply Equation 1; i.e. make code better by reducing the values of multiple code attributes? To answer this question, we will look at the *direction of change* seen in the **RQ4** study; i.e. how often does XTREE recommend decreasing or increasing a static code attribute.

4.1. Test Data

To explore these research questions, we used data from Jureczko et al.'s collection of object-oriented Java systems [31]. To access that data, go to git.io/vGYxc. The Jureczko data records the number of known defects for each class using a post-release bug tracking system. The classes are described in terms of nearly two dozen metrics included in the Chidamber and Kemerer metric suite, such as number of children (noc), lines of code (loc), etc. For details on the Jureczko code metrics, see Figure 2. For details on the rows and versions of that data, see the left-hand-side columns of Figure 5.

	Data set properties					Results from learning								
	training		testing			untuned			tuned			change		
data set	versions	cases	versions	cases	% defective	pd	pf	good?	pd	pf	good?	pd	pf	
jedit	3.2, 4.0, 4.1, 4.2	1257	4.3	492	2	55	29		64	29	y	9	0	⋆
ivy	1.1, 1.4	352	2.0	352	11	65	35	y	65	28	y	0	-7	⋆
camel	1.0, 1.2, 1.4	1819	1.6	965	19	49	31		56	37		5	6	
ant	1.3, 1.4, 1.5, 1.6	947	1.7	745	22	49	13	y	63	16	y	14	3	⋆
synapse	1.0, 1.1	379	1.2	256	34	45	19		47	15		2	-4	
velocity	1.4, 1.5	410	1.6	229	34	78	60		76	60		-2	0	
lucene	2.0, 2.2	442	2.4	340	59	56	25		60	25	y	4	0	
poi	1.5, 2, 2.5	936	3.0	442	64	56	31		60	10	y	4	-21	⋆
xerces	1.0, 1.2, 1.3	1055	1.4	588	74	30	31		40	29		10	-2	×
log4j	1.0, 1.1	244	1.2	205	92	32	6		30	6		-2	0	×
xalan	2.4, 2.5, 2.6	2411	2.7	909	99	38	9		47	9		9	0	×

Figure 4: Training and test *data set properties* for Jureczko data , sorted by % defective examples. On the right-hand-side, we show the *results from learning*. Data is usable if it has a recall of 60% or more and false alarm of 30% or less (and note that, after tuning, there are more usable data sets than before). Results marked with "⋆" show large improvements in performance, after tuning (lower *pf* or higher *pd*). Data in the three bottom rows , marked with "×", are performing poorly– that data so many defective examples that it is hard for our learners to distinguish between classes.

4.2. Secondary verification Oracles

As mentioned in the introduction, our proposed framework has two oracles: a primary change oracle (XTREE) and the secondary verification oracle described in this section.

It can be difficult to judge the effects of removing bad smells. Refactored code cannot be assessed just by a rerun of the the test suite (since refactorings are supposed to preserve behavior). Also, it many not be practical to assessing code reorganizations by making the code reorganizations, then seeing what happens to that code for the rest of the project. In a test-driven agile project, as SCRUM meetings change what stories are implemented next, the operational profile that lead to defects in the old code may not be repeated in future executions.

To resolve this problem, SE researchers such as Cheng et al. [32], O'Keefe et al. [33, 34], Moghadam [35] and Mkaouer et al. [36] use a *secondary verification oracle* (which is learned separately to the primary oracle) that assesses how defective is the code before and after some code reorganization. For their second oracle, Cheng, O'Keefe, Moghadam and Mkaouer et al. use the QMOOD hierarchical quality model [37]. A shortcoming of QMOOD is that quality models learned from other projects may perform poorly when applied to new projects [29]. Hence, for this study, we eschew older quality models like QMOOD. Instead, we use Random Forests [38] to learn defect predictors from OO code metrics. Note that, unlilke QMOOD, such measurements are specific to the project, and the measurements can be reconstructed and the predictors rebuilt for other projects.

Random Forests are a decision tree learning method but instead of building one tree, hundreds are built using randomly selected subsets of the data. The final predictions come from averaging the predictions over all the trees. Recent studies endorsed the use of Random Forests for defect prediction [39].

Figure 4 shows studies with Random Forests and the Jureczko data. Given V released versions, we test on version V and train on the available data from $V-1$ earlier releases (as shown in Figure 4. Note the three bottom rows marked with $\times$: these contain predominately defective classes (two-thirds, or more). In such data, it is hard to distinguish good examples (due to all the bad examples).

In order to identify the presence (or absence) of defects, we can use Boolean classes in the Jureczko data (`True` if defects >0; `False` if defects = 0). For such data, the quality of the predictor can be measured using (a) the probability of detection (a.k.a. "pd" or recall): the percent of faulty classes in the test data detected by the *predictor*; and (b) the probability of false alarm (a.k.a. "pf"): the percent of non-fault classes that are *predicted* to be defective.

The "untuned" columns of Figure 4 show a preliminary study. This study used Random Forest with its "off-the-shelf" tunings (i.e. 100 trees per forest). The forests were built from training data and applied to test data not seen during training. In this study, we called a data set "usable" if Random Forest was able to classify the instances with a performance threshold of $pd \geq 60 \wedge pf \leq 30\%$ (determined from standard results in other publications [40]). Note that no data set meet that criteria.

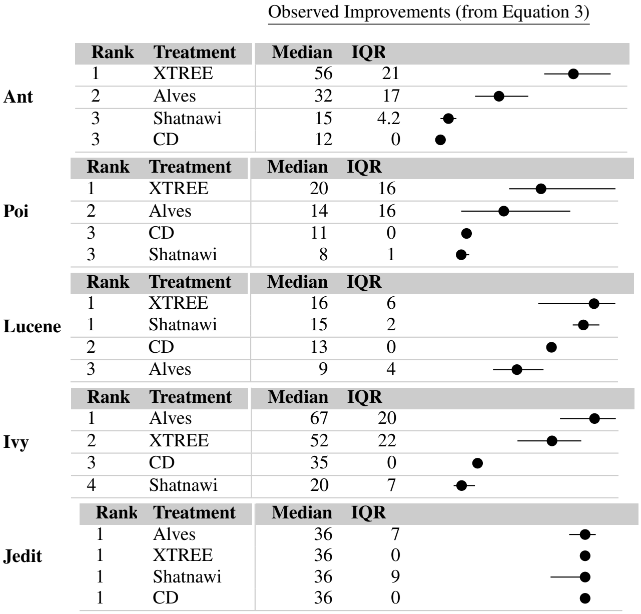
The "tuned" columns of Figure 4 show that we can salvage some of the data sets. We applied both the SMOTE algorithm and differential evolution to improve the performance of the classifier. Pelayo and Dick [41] report that defect prediction is improved by SMOTE [42]; i.e. an over-sampling of minority-class examples and an under-sampling of majority-class examples. Fu et al. [43] report that parameter tuning with differential evolution [44] can quickly explore the tuning options of Random Forest to find better settings for the (e.g.) size of the forest, the termination criteria for tree generation, etc. We note that SMOTE-ing and parameter tunings were applied to the training data only and not to the test data.

The rows marked with a ⋆ in Figure 4 show data sets whose performance was improved remarkably by these techniques. For example, in *poi*, the recall increased by 4% while the false alarm rate dropped by 21%. However, as might have been expected, we could not salvage the data sets in the three bottom rows.

We eliminate the data sets for which we could not build an adequately performing Random Forest classifier with $pd \geq 60 \wedge pf \leq 30\%$. Thus, our analysis uses the *jedit, ivy, ant, lucene* and *poi* for evaluating recommended changes.

4.3. Statistics

We use 40 repeated runs, each with different random number seeds (we use 40 since that is more than the 30 samples needed to satisfy the central limit theorem). Each run collects the Equation 3 values. We use multiple runs since two of our methods use some random choices: CD uses the stochastic WHERE clustering algorithm [29] while XTREE non-deterministically picks thresholds randomly from the high and low boundary of a range. Hence, to compare all four methods, we must run the analysis many times.



Observed Improvements (from Equation 3)

	Rank	Treatment	Median	IQR
Ant	1	XTREE	56	21
	2	Alves	32	17
	3	Shatnawi	15	4.2
	3	CD	12	0
Poi	1	XTREE	20	16
	2	Alves	14	16
	3	CD	11	0
	3	Shatnawi	8	1
Lucene	1	XTREE	16	6
	1	Shatnawi	15	2
	2	CD	13	0
	3	Alves	9	4
Ivy	1	Alves	67	20
	2	XTREE	52	22
	3	CD	35	0
	4	Shatnawi	20	7
Jedit	1	Alves	36	7
	1	XTREE	36	0
	1	Shatnawi	36	9
	1	CD	36	0

Figure 5: Results for **RQ1** from the Jureczko data sets. Results from 40 repeats. Values come from Equation 3. Values near 0 imply no improvement, *Larger* median values are *better*. Note that XTREE and Alves are usually best and CD and Shatnami are usually worse.

To rank these 40 numbers collected from CD, XTREE, Shatnawi, and Alves et al., we use the Scott-Knott test recommended by Mittas and Angelis [45]. Scott-Knott is a top-down clustering approach used to rank different treatments. If that clustering finds an interesting division of the data, then some statistical test is applied to the two divisions to check if they are statistically significant different. If so, Scott-Knott recurses into both halves.

To apply Scott-Knott, we sorted a list of $l = 40$ values of Equation 3 values found in $ls = 4$ different methods. Then, we split l into sub-lists m, n in order to maximize the expected value of differences in the observed performances before and after divisions. E.g. for lists l, m, n of size ls, ms, ns where $l = m \cup n$:

$$E(\Delta) = \frac{ms}{ls} abs(m.\mu - l.\mu)^2 + \frac{ns}{ls} abs(n.\mu - l.\mu)^2$$

We then apply a apply a statistical hypothesis test H to check if m, n are significantly different (in our case, the conjunction of A12 and bootstrapping). If so, Scott-Knott recurses on the splits. In other words, we divide the data if *both* bootstrap sampling and effect size test agree that a division is statistically significant (with a confidence of 99%) and not a small effect ($A12 \geq 0.6$). For a justification of the use of non-parametric bootstrapping, see Efron & Tibshirani [46, p220-223]. For a justification of the use of effect size tests see Shepperd&MacDonell [47]; Kampenes [48]; and Kocaguenli et al. [49]. These researchers warn that even if a hypothesis test declares two populations to be “significantly” different, then that result is misleading if the “effect size” is very small. Hence, to assess the performance differences we first must rule out small effects using A12, a test recently endorsed by Arcuri and Briand at ICSE’11 [50].

The Scott-Knott results are presented in the form of line diagrams like those shown on the right-hand-side of Figure 5. The black dot shows the median Equation 3 values and the horizontal lines stretch from the 25th percentile to the 75th percentile (the inter-quartile range, IQR). As an example of how to read this table, consider the *Ant* results. Those rows are sorted on the median values of each method. Note that all the methods have Equation 3 >0%; i.e. all these methods reduced the expected value of the performance score in that experiment while XTREE achieved the greatest reduction (of 56% from the original value). These results table has a left-hand-side **Rank** column, computed using the Scott-Knott test described above. In the *Ant* results, XTREE is ranked the best, while CD is ranked worst.

5. Results

5.1. RQ1: Effectiveness

Which of the methods defined in Section 3 is the best change oracle for identifying what and howcode modules should be changed?

Figure 5 shows the comparison results. The Alves and Shatnawi entries show the net effects of applying Equation 1 where the t_i values were found by Alves and Shatnawi. In this part of the analysis, we applied Scott-Knott to all the Equation 3 values seen when code metrics were changed according to the Alves and Shatnawi

Two data sets are very responsive to defect reduction suggestions: Ant, and Ivy (both of which show best case improvements over 50%). The expected value of defects is changed less in Jedit. This data sets’ results surprisingly uniform; i.e. all methods find the same ways to reduce the expected number of defects. For an explanation of the Jedit uniformity, see §**??**.

Two data sets are not very responsive to defect reduction: Poi and Lucene. The reason for this can be see in Figure 4: both these data sets contain more than 50% defective modules. In that space, all our methods lack a large sample of defect-free examples.

Also consider the relative rank of the different approaches, CD and Shatnawi usually perform comparatively worse while XTREE gets top ranked position the most number of times. That said, Alves sometimes beats XTREE (see Ivy) while sometimes it ties (see Jedit).

Features	Ant				Ivy				Lucene				Jedit				Poi			
	XTREE	CD	Alves	Shatn	XTREE	CD	Alves	Shatn	XTREE	CD	Alves	Shatn	XTREE	CD	Alves	Shatn	XTREE	CD	Alves	Shatn
wmc	.	92	100	100	18	95	100	100	89	95	100	.	.	63	.	.	.	100	100	.
dit	.	77	100	.	.	87	100	.	.	80	100	.	.	72	100	100	.	46	100	.
noc	.	20	100	.	.	.	100	.	.	26	.	.	.	.	.	.	.	.	.	.
cbo	88	99	100	100	91	100	100	100	60	94	100	100	.	100	100	.	1	74	100	.
rfc	100	100	100	.	8	95	100	.	10	83	100	.	100	100	100	100	100	95	100	.
lcom	.	98	100	100	15	100	100	100	.	94	100	.	.	100	100	.	.	100	100	100
ca	.	93	100	.	7	95	100	.	40	89	100	.	.	63	100	100	.	74	100	.
ce	5	100	100	.	.	97	100	.	.	90	100	.	.	100	100	100	.	64	100	.
npm	.	88	100	.	8	97	100	.	.	93	100	100	.	100	100	.	.	100	100	.
lcom3	.	90	100	.	7	95	100	.	13	79	100	100	.	63	100	100	.	92	100	100
loc	100	99	100	100	97	97	100	100	60	100	100	100	.	100	.	100	100	100	100	100
dam	.	21	100	.	.	22	100	.	.	55	100	.	.	45	100	100	.	73	100	.
moa	.	67	100	.	.	82	100	.	.	60	100	100	.	54	100	100	.	58	100	.
mfa	5	93	100	.	.	90	100	.	5	80	100	.	.	72	100	.	.	72	100	.
cam	.	99	100	100	84	100	100	100	10	94	100	.	.	100	100	.	.	98	100	100
ic	.	52	100	100	.	70	.	.	.	68	100	.	.	36	100	.	.	43	100	100
cbm	.	59	100	.	.	85	.	.	.	71	100	.	.	36	100	100	.	67	100	.
amc	.	99	.	.	.	95	100	.	30	100	.	.	.	100	100	100	.	97	.	.
max cc	.	87	100	100	.	85	100	100	.	71	.	.	.	45	100	.	.	63	100	.
avg cc	12	99	100	100	.	95	100	100	13	98	.	.	.	100	100	100	.	92	100	.

Figure 6: Results for **RQ2**. Percentage counts of how often an approach recommends changing a code metric (in 40 runs). "100" means that this code metrics was always recommended. Cells marked with "." indicate 0%. For the Shatnami and Alves et al. columns, metrics score 0% if they always fail the $p \leq 0.05$ test of §**??**. For CD, cells are blank when two centroids have the same value for the same code metrics. For XTREE, cells are blanks when they do not appear in the delta between branches. Note that XTREE mentions specific code metrics far fewer times than other methods.

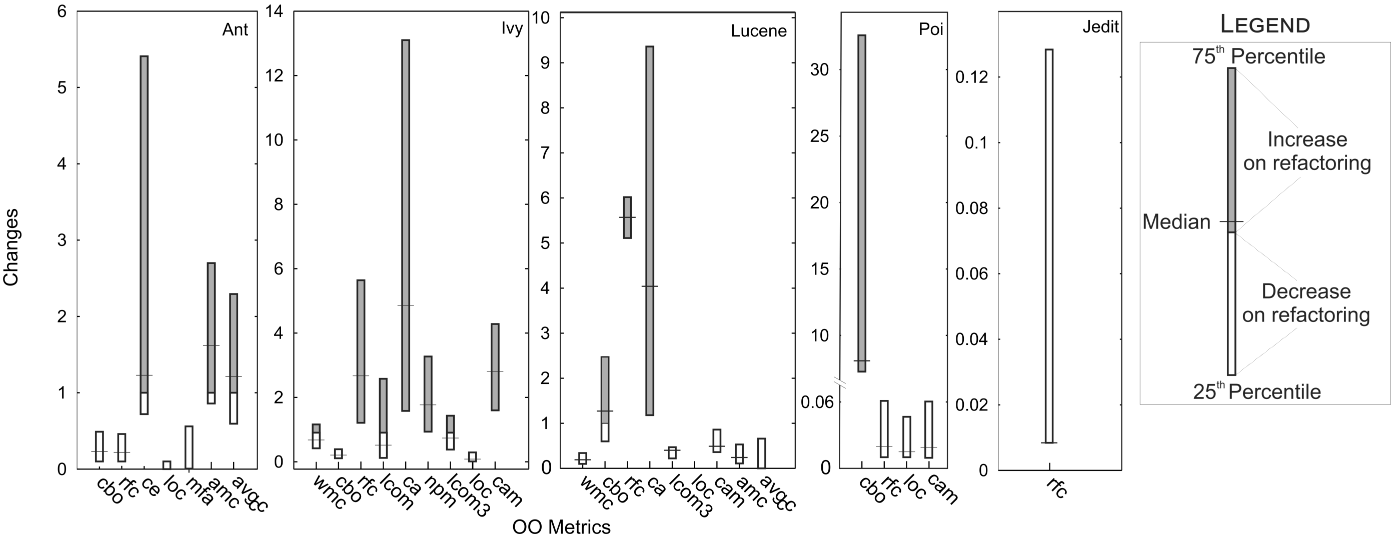


Figure 7: Results from XTREE. While Figure 6 are the *number* of times a code metric was changed, this figure shows *how far* each code metric was changed. Each vertical bar marks the 27,50,75th percentile change seen in 40 repeats. All numbers are ratios of initial to final values. All bar regions marked in gray show *increases*. The interesting feature of these results are that many of the changes proposed by XTREE require *increases* (this puzzling observation is explained in the text).

In summary, our **Result1** is that, of the change orcales studies here, XTREE is the best oracle on how to change code modules (in order to reduce defects).

5.2. RQ2: Verbosity

Which of the Section 3 methods recommended changes to the fewest code attributes?

Figure 6 shows the frequency with which the methods recommend changes to specific code metrics. Note that XTREE proposes thresholds to few code metrics compared to the other approaches.

Hence, our **Result2** is that, of all the code change oracles studied here, XTREE recommended far fewest changes to static code measures. Note only that, combining Figure 5 with Figure 6, we see that even though XTREE proposes changes to far fewer code metrics, those few code metrics are usually just as effective (or more effective) than the multiple thresholds proposed by CD, Shatnawi or Alves. That is, XTREE proposes *fewer* and better thresholds than the other approaches studied here.

5.3. RQ3: Stopping

How effective is XTREE at offering "stopping points" (i.e. clear guidance on what not to do)?

The **RQ2** results showed that XTREE's recommendations are small in a *relative sense*; i.e. they are relatively smaller than the other methods studied here. Note only that, but XTREE's recommendations are small in an *absolute sense*. Consider all frequency at which Figure 6 values for XTREE that are over 33%; i.e. in at a third of our repeated runs, XTREE mentioned a code attribute. For Ant, Ivy, Lucene, Jedit, and Poi, those frequencies are 3, 3, 3, 4, 1, 2 respectively (out of twenty). This means that, usually, XTREE omits references to 17,17,17,16,19,18 static code attributes (out of 20). Any code reorganization based on a bad smell detector that uses these omitted code attributes could hence be stopped.

	wmc	dit	noc	cbo	rfc	lcom	ca	ce	npm	lcom3	loc	dam	moa	mfa	cam	ic	cbm	amc	max_cc	avg_cc
Ant				−	−			+			−			−				+		+
Ivy	−			−	+	−	+		−	−	−				+					
Poi				+	−						−				−					
Lucene	−			+	+		+			−	−				−			−		−
Jedit					−															

Figure 8: Direct of changes seen in a comparison of statistically significantly different static code attributes measures seen in the clusters found by XTREE. Each dataset contains 20 Static Code Metrics (for a description of each of these metrics, please refer to [51]). The rows contain the datasets, and the columns denote the metrics. A "+" symbol represents a recommendation that requires a significant statistical increase (with a p-value$\leq$0.05), and likewise, a "−" represents a significant statistical decrease.

Hence our **Result3** is that, in a any project, XTREE's recommended changes only mentions one to four of the static code attributes. Any bad smell defined in terms of the remaining 19 to 16 code attributes (i.e. most of them) would hence be deprecated.

5.4. RQ4: Stability

Across different projects, how variable are the changes recommended by our best change oracle?

Figure 6 counted how *often* XTREE's recommendations mentioned a static code attribute. Figure 7, on the other hand, shows the *direction* of the recommended change:

- Gray bars show an *increase* to a static code measure;
- White bars shows a *drecrease* to a static code measure;
- Bars that are all white or all gray indicate that in our 40 repeated runs, XTREE recommended changing an attribute the same way, all the time.
- Bars that are mixtures of white and gray mean that, sometimes, XTREE makes different recommendations about how to change a static code attribute.

Based on Figure 7, we say that **Result4** is that the direction of change recommended XTREE is very stable repeated runs of the program (evidence: the bars are mostly the same color).

Figure 7 also comments on the inter-relationships between static code attributes. Note that while some measures in Figure 7 are decreased, many are increased. For example, consider the *Poi* results from Figure 6 that recommends decreasing *loc* but making large increases to *cbo* (coupling between objects). Here, XTREE is advising us to break up large classes class by into services in other classes. Note that such a code reorganization will, by definition, increase the coupling between objects. Note also that such increases to reduce defects would never be proposed by the outlier methods of Shatnawi or Alves since their core assumption is that bad smells arise from unusually large code metrics.

5.5. RQ5: Conjunctive Fallacy

Is it always useful to apply Equation 1; i.e. make code better by reducing the values of multiple code attributes?

In Figure 7, the bars are colored both white and gray; i.e. XTREE recommends *decreasing* and *increasing* static attribute values. That is, always decreasing static code measures (as suggested by Equation 1) is *not* recommended by XTREE.

To explore this point further, we conducted the following "what-if" study. Once XTREE builds its trees with its leaf clusters then:

1. For all clusters C_0, C_1 where the percent of defects in C_0 is greater than C_1...
2. For all attribute measures that are have a statistically significantly different distribution between C_0 and C_1...
3. Report the direction of the change.

Note that the changes found in this way are somewhat more general than the above results since they were limited to comments on the test set given to the program. The above three points comments on the direction of *all possible changes* that XTREE could ever report

Figure 8 shows the results of this "what if" analysis. As might be expected, the recommendation is to always reduce lines of code (loc). But for the other attributes, there are many recommendations to increase those values. Hence, **Result5** is that while XTREE always recommends reducing loc, it also often recommends increasing the values of other static code attributes.

6. Reliability and Validity of Conclusions

The results of this paper are biased by our choice of code reorganization goal (reducing defects) and our choice of measures collected from software project (OO measures such as depth of inheritance, number of child classes, etc). That said, it should be possible extend the methods of this paper to other kinds of goals (e.g. maintainability, reliability, security, or the knowledge sharing measures favoured by Bosu and Carver [52]) and other kinds of inputs (e.g. the process measures favored by Rahman, Devanbu et al. [53])

6.1. Reliability

Reliability refers to the consistency of the results obtained from the research. It has at least two components: internal and external reliability.

Internal reliability checks if an independent researcher reanalyzing the data would come to the same conclusion. To assist other researchers exploring this point, we offer a full replication package for this study at https://github.com/ai-se/XTREE_IST.

External reliability assesses how well independent researchers could reproduce the study. To increase external reliability, this paper has taken care to clearly define our algorithms. Also, all the data used in this work is available online.

For the research wishing to reproduce our work to other kinds of goals, we offer the following advice:

- Find a data source for the other measures of interest;
- Implement another secondary verification oracle that can assess maintainability, reliability, security, etc;
- Implement a better primary verification oracle that can do "better" than XTREE at finding changes (where "better" is defined in terms of the opinions of the verification oracle).

6.2. *Validity*

Validity refers to the extent to which a piece of research actually investigates what the researcher purports to investigate. Validity has at least three components: internal and external validity.

6.3. *Internal Validity*

The results of this paper are also specific to the data sets explored here. Such sampling bias threatens any data mining experiment; i.e., what matters *there* may not be true *here*. For example, the data sets used here comes from a survey of open source JAVA projects from Jureczko et al. [31]. Any biases in their selection procedures threaten the validity of these results.

That said, the best we can do is define our methods and publicize our data and code so that other researchers can try to repeat our results and, perhaps, point out a previously unknown bias in our analysis. Hopefully, other researchers will emulate our methods in order to repeat, refute, or improve our results.

The recommendation to management, shown on page one of this paper, was to stop developers doing "X" if there was no historical evidence that "X" was useful. This recommendation assumes that such an historical record can be accessed. If not, then the results of this paper could be use used as a guide (see our Table 8). Alternatively, some form of transfer learning [54, 55, 56] could be used to import data from another project (but more research would be required to test the validity of combing tools like XTREEs with transfer learning).

As with any empirical study, several issues can affect the final results. Therefore, any conclusions made from this work must be offered with the following caveats.

6.4. *External Validity*

Based on the experimental results above, as well as the discussion in §2, we believe that bad smell indicators (e.g. $loc > 100$) have limited external validity beyond the projects from which they are derived. While specific models are externally valid, there may still be general methods like XTREE for finding the good local models.

Our definition of bad smells is limited to those represented by OO code metrics (a premise often used in related work). XTREE, Shatnawi, Alves et al. can only comment on bad smells expressed as code metrics in the historical log of a project.

If developers want to justify their refactorings via bad smells expressed in other terminology, then the analysis of this paper must:

- Either wait till data about those new terms has been collected.
- Or, apply cutting edge transfer learning methods [54, 55] to map data from other projects into the current one.

Note that the transfer learning approach would be highly experimental and require more study before it can be safely recommended.

Sampling bias threatens any data mining experiment; i.e., what matters there may not be true here. For example, the data sets used here comes from Jureczko et al. and any biases in their selection procedures threaten the validity of these results. That said, the best we can do is define our methods and publicize our data and code so that other researchers can try to repeat our results and, perhaps, point out a previously unknown bias in our analysis. Hopefully, other researchers will emulate our methods in order to repeat, refute, or improve our results.

7. Conclusions

How to discourage useless code reorganizations? We say: ignore those not supported by the historical log of data from the current project. When that data is not available (e.g. early in the project) then developers could use the general list of bad smells shown in Figure 1. However, our results show that bad smell detectors are most effective when they are based on a small handful of code metrics (as done by XTREE). Hence, using all the bad smells of Figure 1 may not be optimal.

For our better guess at how to reduce defects by changing code attributes, see Figure 8. But given the large variance if the change recommendations, we strongly advice teams to use XTREE on their data to find their own best local changes.

XTREE improves on prior methods for generating bad smells:

- As described in §2, bad smells generated by humans may not be applicable to the current project. On the other hand, XTREE can automatically learn specific thresholds for bad smells for the current project.
- Prior methods used an old quality predictor (QMOOD) which we replace with defect predictors learned via Random Forests from current project data.
- XTREE's conjunctions proved to be arguably as effective as those of Alves (see Figure 5) but far less verbose (see Figure 6). Since XTREE *approves* of fewer changes it hence *disapproves* of most changes. This makes it a better tool for critiquing and rejecting many of the code reorganizations.

Finally, XTREE does not suffer from the conjunctive fallacy. Older methods, such as those proposed by Shatnawi and Alves assumed that the best way to improve code is to remove outlier values. This may not work since when code is refactored, *the functionality has to go somewhere*. Hence, reducing the lines of code in one module necessitates *increasing* the coupling that module to other parts of the code. In future, we recommend software team use bad smell detectors that know what attribute measures need *decreasing* as well as *increasing*.

Acknowledgements

The work is partially funded by NSF awards #1506586 and #1302169.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman, Boston, MA, USA, 1999.

[2] Radu Marinescu and Michelle Lanza. Object-oriented metrics in practice, 2006.

[3] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[4] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[5] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[6] Karin Erni and Claus Lewerentz. Applying design-metrics to object-oriented frameworks. In *Software Metrics Symposium, 1996., Proceedings of the 3rd International*, pages 64–74. IEEE, 1996.

[7] Ralf Bender. Quantitative risk assessment in epidemiological studies investigating threshold effects. *Biometrical Journal*, 41(3):305–319, 1999.

[8] R. Shatnawi. A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on Software Engineering*, 36(2):216–225, March 2010.

[9] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *2010 IEEE Int. Conf. Softw. Maint.*, pages 1–10. IEEE, sep 2010.

[10] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 20(2):549–575, 2015.

[11] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):17, 2015.

[12] Rahul Krishna and Tim Menzies. Actionable= cluster+ contrast? In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 14–17. IEEE, 2015.

[13] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, pages 403–414. IEEE, May 2015.

[14] M.V. Mantyla, J. Vanhanen, and C. Lassenius. Bad smells - humans as code critics. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 399–408, Sept 2004.

[15] A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 242–251, Oct 2013.

[16] D.I.K. Sjoberg, A. Yamashita, B.C.D. Anda, A. Mockus, and T. Dyba. Quantifying the effect of code smells on maintenance effort. *Software Engineering, IEEE Transactions on*, 39(8):1144–1156, Aug 2013.

[17] Carol Passos, Ana Paula Braun, Daniela S. Cruzes, and Manoel Mendonca. Analyzing the impact of beliefs in software project practices. In *ESEM'11*, 2011.

[18] Magne Jørgensen and Tanja M. Gruschke. The impact of lessons-learned sessions on effort estimation and uncertainty assessments. *Software Engineering, IEEE Transactions on*, 35(3):368 –383, May-June 2009.

[19] Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E. Hassan. An empirical study of goto in c code from github repositories. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 404–414, New York, NY, USA, 2015. ACM.

[20] T. Menzies, W. Nichols, F. Shull, and L. Layman. Are delayed issues harder to resolve? *Empirical Software Engineering (submitted)*, 2016. Online at https://goo.gl/MZ0h6H.

[21] Prem Devanbu, Thomas Zimmermann, and Christian Bird. Belief & evidence in empirical software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, pages 108–119. ACM, 2016.

[22] Vinay Prasad, Andrae Vandross, Caitlin Toomey, Michael Cheung, Jason Rho, Steven Quinn, Satish Jacob Chacko, Durga Borkar, Victor Gall, Senthil Selvaraj, Nancy Ho, and Adam Cifu. A decade of reversal: An analysis of 146 contradicted medical practices. *Mayo Clinic Proceedings*, 88(8):790 – 798, 2013.

[23] Christie Aschwanden. Convincing the Public to Accept New Medical Guidelines. `http://goo.gl/RT6SK7`, 2010. FiveThiryEight.com. Accessed: 2015-02-10.

[24] Christie Aschwanden. Your Brain Is Primed To Reach False Conclusionss. `http://goo.gl/OO3B7s`, 2015. FiveThirtyEight.com. Accessed: 2015-02-10.

[25] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesly Professional, 2005.

[26] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Verlag, 2006.

[27] A. Campbell. SonarQube: Open source quality management, 2015. Website: tiny.cc/2q4z9x.

[28] R Borges and T Menzies. Learning to Change Projects. In *Proceedings of PROMISE'12, Lund, Sweden*, 2012.

[29] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. Local versus global lessons for defect prediction and effort estimation. *Software Engineering, IEEE Transactions on*, 39(6):822–834, June 2013.

[30] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuousvalued attributes for classification learning. In *Thirteenth International Joint Conference on Articial Intelligence*, volume 2, pages 1022–1027. Morgan Kaufmann Publishers, 1993.

[31] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 9:1—-9:10. ACM, 2010.

[32] Betty Cheng and Adam Jensen. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 1341–1348, New York, NY, USA, 2010. ACM.

[33] Mark O'Keeffe and Mel Ó Cinnéide. Search-based refactoring: An empirical study. *J. Softw. Maint. Evol.*, 20(5):345–364, September 2008.

[34] Mark Kent O'Keeffe and Mel O. Cinneide. Getting the most from search-based refactoring. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1114–1120, New York, NY, USA, 2007. ACM.

[35] Iman Hemati Moghadam. *Search Based Software Engineering: Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings*, chapter Multi-level Automated Refactoring Using Design Exploration, pages 70–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[36] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 331–336, New York, NY, USA, 2014. ACM.

[37] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, January 2002.

[38] L Breiman. Random forests. *Machine learning*, pages 5–32, 2001.

[39] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485–496, July 2008.

[40] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw Eng.*, 33(1):2–13, Jan 2007. Available from `http://menzies.us/pdf/06learnPredict.pdf`.

[41] L. Pelayo and S. Dick. Applying novel resampling strategies to software defect prediction. In *Fuzzy Information Processing Society, 2007. NAFIPS '07. Annual Meeting of the North American*, pages 69–72, June 2007.

[42] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[43] Wei Fu, Tim Menzies, and Xipeng Shen. Tuning for software analytics: is it really necessary? *Submitted to Information and Software Technology*, 2016.

[44] Rainer Storn and Kenneth Price. Differential Evolution — A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[45] Nikolaos Mittas and Lefteris Angelis. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Trans. Software Eng.*, 39(4):537–551, 2013.

[46] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. Mono. Stat. Appl. Probab. Chapman and Hall, London, 1993.

[47] Martin J. Shepperd and Steven G. MacDonell. Evaluating prediction systems in software project estimation. *Information & Software Technology*, 54(8):820–827, 2012.

[48] Vigdis By Kampenes, Tore Dybå, Jo Erskine Hannay, and Dag I. K. Sjøberg. A systematic review of effect size in software engineering experiments. *Information & Software Technology*, 49(11-12):1073–1086, 2007.

[49] Ekrem Kocaguneli, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, and Tim Menzies. Distributed development considered harmful? In *Proceedings - International Conference on Software Engineering*, pages 882–890, 2013.

[50] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE'11*, pages 1–10, 2011.

[51] Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. Local vs. global models for effort estimation and defect prediction. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 343–351. IEEE, nov 2011.

[52] A. Bosu and J. C. Carver. Impact of peer code review on peer impression formation: A survey. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 133–142, Oct 2013.

[53] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings - International Conference on Software Engineering*, pages 432–441, 2013.

[54] Jaechang Nam and Sunghun Kim. Heterogeneous defect prediction. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 508–519, New York, NY, USA, 2015. ACM.

[55] Xiaoyuan Jing, Fei Wu, Xiwei Dong, Fumin Qi, and Baowen Xu. Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 496–507, New York, NY, USA, 2015. ACM.

[56] L. Layman R. Krishna, T.Menzies. "too much automation? the bellwether effect and its implications for transfer learning. In *ASE'16*, 2016.