RESPONSE TO REVIEWERS

Comments from the editor

The reviewers have finished to evaluate your work. As you can observe, they raised some concerns that deserve your attention ...

Thank you very much indeed for your comments.

... It is important that you take into account their comments and evolve the paper accordingly. Please, evolve the paper to make clear the defect model, hypothesis, limitations and readability

We offer clarifications to all the issues highlighted by the reviewers. In addition to these, we have made several key modifications to our paper. We have changed the title of our paper to reflect the changes we have made. In order to assist reviewers in tracking our changes, we have ensured all new additions are highlighted in blue. A brief summary of our changes are listed below:

1. At the suggestion of Reviewers 1 and 2, we have included a section on the choice of datasets. Additionally, we have greatly expanded the section on defect prediction to highlight its importance and its relevance to our work.
2. To address comments by Reviewer #1 on how this work fits in with the body of knowledge, we have compiled a new section (see §2.1) on the previous attempts to address similar issues.
3. Further, at the suggestion of Reviewer #1, we have included a section remarking on the key limitations of our work. It now appears as a new section (see §7).
4. Also, as a response to Reviewer #1's comments, we have modified the future work section to reflect the implications of our findings on *technical debt*.
5. We acknowledge Reviewer #2's comment that our initial submission lacks a formal introduction of software "defects" and its relevance to code reorganization. In order to address this, we have attempted to clarify the interpretation of defects in our work. As for other issues related to defects, we have added a new section on the choice of datasets (see §3).
6. Also, keeping in mind the suggestions of Reviewer #2, we have rewritten the research questions (especially, RQ1 and RQ5).
7. Additionally, at the suggestion of Reviewer #2, we clarify our descriptions of several figures included in this paper. Specifically, we expand on the importance of tuning which relates to Figure 5 (previously Figure 4). We also expand on the significance of Figure 9 (previously Figure 8).
8. Finally, we have evolved our paper in the most appropriate areas in response to all the major and minor issues raised by both Reviewer #1 and Reviewer #2.

Response to Reviewer #1

Thank you for your detailed comments. Our new draft contains many corrections, clarifications, and extensions as directed by your feedback. All our changes are marked in the body and appears in blue.

I missed a section discussing possible implications for researchers and practitioners. How the results of this work contribute to future research activities in the area? How practitioners can benefit from the results achieved in the paper?

You're quite correct, there was no future work section in the paper. The following text has been included in the revised submission and appears in §7 (see Reviewer 1a).

> "Our research shows that it is potentially naïve to explore thresholds in static code attributes in isolation to each other. This work clearly demonstrates how changing one necessitates changing other associated metrics. So, for future work, we recommend that researchers and practitioners look for tools that recommend changes to sets of code changes. When exploring candidate technologies, apart from XTREE, researchers may potentially use: (1) Association rule learning; (2) Thresholds generated across synthetic dimensions (eg. PCA); and (3) Techniques that cluster data and look for deltas between them. (Note: we offer XTREE as a possible example of this point)."
>
> "Additionally, we plan on extending our work by exploring scalable solutions to achieve similar results in much larger datasets. After this, we shall look at applications beyond that of measuring static code attributes. For example, as an initial attempt, we have been looking at sentiment analysis in stack overflow exchanges to learn dialog pattern that most select for relevant entries."

I also missed a section presenting the main limitations of the work. Although some words have been said about it, I believe it is important to have a more structured discussion on a specific section.

You are quite correct that we have missed a section highlighting the key limitations of our work. In the revised paper, we have dedicated a section to discussing the limitations and future work (see Reviewer1b in §7).

It would be interesting to have further discussion at the end indicating how this work complements the body of knowledge about code smells. Do reached results indicate a similar direction? contradictory? Does This work somehow confirms the results of other studies or points to new directions?

Thank you for your comment, we admit the last version of the paper has an inadequate discussion on this matter. This revised version now includes an expanded prior work section

which places greater emphasis on these issues (see Reviewer1c in §2.1).

As to your specific comment that this be included in the end, we feel our response to this recommendation is an important note on the state-of-the-art in this area. We have therefore added it towards the beginning of the paper instead, so it may premise the rest of our work.

... this work evaluated XTREE, a framework to evaluate whether a code reorganization is likely to have a perceivable benefit in terms of defect-proneness. It is fine to have this scope (defect-proneness) in terms of evaluation. However, it would be nice to have a discussion on why this characteristic (defect-proneness) is good to support the decision on code reorganization and/or how it could complement other characteristics.

You are quite right in pointing out that we need a better description of defect proneness. We have now added a section on defect-proneness and its relevance to our paper (please see content marked with Reviewer1d in §3).

... nowadays we are seeing a growing discussion on technical debt, that brings financial aspects (interest, principal, debt) to the decision-making process on development activities. Could we use XTREE as a strategy to prioritize the payment of technical debt items?

Thank you for that suggestion. We do think XTREE can be used as a recommendation system to prioritize technical debt. In our revised paper, we expanded the introduction to highlight this possibility (see content prefixed with Reviewer1e in §7).

Finally, I have one last question: is there any reason for do not use the GQM template to state the goal of the study and also to define null and alternative hypotheses for the research questions?

We can certainly do that if you wish. As to why we *didn't* do it, this paper is clearly MSR-conference-style and not ESEM-conference-style (GQM is less uses at MSR than ESEM).

To whether or not that is a "good thing" or not, we cannot say. What we would say is that we think our use of stand-out boxes for research findings makes our results easy to access within this paper.

That said, if the reviewer thinks this paper needs further structuring to clarify our message then, of course, we would follow that direction.

Response to Reviewer #2

Thank you for your detailed review. We have made several corrections, clarifications, and revisions as directed by your comments, this new draft is significantly improved thanks to your feedback.
All our changes are marked in the body and appears in blue.

The introduction talks about Fowlers bad smells and refers to literature on the (often contradictory) effects of such smells. And Figure 1 shows a list of bad smells. However, without making it clear, the paper continues to talk about smells but the real topic is not the smells but the metrics that are used to identify various smells. Whats the connection between whether a smell is bad or not and the threshold values of the various code metrics?

Thank you for this comment. At your suggestion, we have significantly expanded §3 in order to discuss the use of static code metrics with regards to code smells.

I also find it very difficult to understand the conceptual model of defects in this paper ...

We do apologize for the inconvenience. Your remark clearly indicates that we need to further clarify our use of the term. So at your suggestion, we have dedicated a section to discussing the choice of dataset in §3 (please see content marked with Reviewer2a).

... The concept of number of defects (defect counts) is fully mixed with the concept of percent of classes with defects/defective modules/ presence or absence of defects. What does it then mean to reduce defects in our data sets? Is it the total number of defects in the data set (system?) or the number of infected modules in the data set?

We have clarified this in §3. Briefly, with respect to the datasets used in this study, when we say, "reduce defects in our data sets," we refer to reorganizing code such that our secondary oracle reports that the number of infected modules (i.e., classes with `nDefects` > 0) is reduced to having *no defects*. For notes on our *primary* and *secondary* oracle, see page2, column1, of this paper.

In response to your comment, we have expanded §3, to better explain this in much greater detail (please see content marked with Reviewer2b in §3).

Section 2 discusses the idea of why not just ask developers about the effect of code smells when the research literature is contradictory. The authors argue that practitioners should not be expected resolve contradictions. My point is that it is so obvious that most of Section 2 should be deleted. Figure 1 as part of a related work section may remain.

This is an interesting comment since it is the *opposite* of the feedback we got on a earlier draft of this paper. In fact, we had a certain senior SE person (who the reviewer might actually know....) demanding "why not just use best practices from the literature?". That comment lead to the Fig1 survey.

That said, we quite take your point that our current section §2.3 is far too long. For this draft, prompted by your comment, we have cut its length in half.

... How does a log of defects look like? What kind of information is recorded? ... In case four new modules are developed, is there then also a log history that shows how the total number of modules in the system correlates with the total number of defects? Are we talking about total number of defects or only the number of infected modules?

To answer these issues, we have provided a sample defect data set in Figure 3. The example shows a few rows of data from Ant 1.3 containing: (1) Module name; (2) 20 CK-Metrics; and (3) Defects (as a count of raw defects and a boolean variable). Note that Figure 5 lists all the remaining datasets and they all have been formated in the same way.

From RQ1: "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 new." What is the idea here?

Thank you for the question. Your observation regarding this — *Is the point that a log history of defects has shown that modules with more than 100 loc have more defects (per lines of code?) than smaller modules, and then the action is to reduce the size of that module?* — is accurate. We have now modified the content in corresponding section to further explain this in **RQ1** (please see content marked with Reviewer2c in §5).

RQ1 is "which of the methods is most accurate?" without presenting the methods. They are just referred to in three other papers that probably describe them. What are your selection criteria for comparing the tool (or framework or system as you also call it) XTREE with exactly these methods?

Thank you for your comment. We have fixed the description of **RQ1** to be more specific (please see content marked with Reviewer2d in §1.1). As to selecting primary change oracles, we undertook a survey of literature to identify commonly used methods to do this. We found three other techniques: CD from a previous work, VARL based thresholds [1], and Statistical thresholding [2].

It is stated that this paper reports a case study. I know that within the SW community is its common to use the term case study to denote just a demonstration of an example performed by the researchers. But in more mature disciplines, which we should aim to become a member of, case study has a certain meaning (e.g. Yin 2003). It would mean an evaluation in a real software development context. This is note the case here. See also the work by Runeson and Host 2009 on case studies in software engineering.

Thank you for that comment. We can show that our definition of "case study" is aligned with how Per Runeson uses the term. In 2013, we were working on revising a data mining paper for TSE paper where Runeson was the associate editor. He pushed us hard for better terminilogy and definitions, including our term "case study". To address his concerns, we wrote the following notes about "case studies" and data mining papers. Note that he accepted the point offered below and accepted the paper.

> As we read standard texts on empirical software engineering (e.g. Runeson & Host[1]) the term "case study" often refers to some act which includes observing or changing the conditions under which humans perform some software engineering action.
>
> In the age of data mining and modelbased optimization this definition should be further extended to include case studies exploring how data collected from different projects can be generalized across multiple projects. As Cruzes et al. comment:
>
> > "Choosing a suitable method for synthesizing evidence across a set of case studies is not straightforward... limited access to raw data may limit the ability to fully understand and synthesize studies.[2]"

Note that the above quote from Cruzes et al. is almost a summary of the specific goals of this paper: given limited amounts of software project data, how can we rank bad smells such as to avoid making unecessary, or bad, changes.

Regarding case study, what kind of work remains before the proposed tool could be used by practitioners?

We need to address a scale-up problem (which is the current focus of Mr. Krishna's Ph.D. work) and once that is done, we will make this a Python package freely available to the world on Windows, Linux, and Mac machines via the simple command '`pip install xtree`'. After that is installed, the code would search a directory of project data to automatically generate the figures of this paper.

Whats the difference between tool, framework, and method in this paper?

Thank you for pointing that out. We offer the following definitions: (1) Framework: A basic structure underlying recommending reorganization. It comprises of a primary change oracle and a secondary verification oracle; (2) Method: Refers to how the primary change oracle offers these recommendation. In this work it may be based on Cluster Deltas (XTREE) or threshold based (Shatnawi and Alves); and (3) Tool: Comprises of any one primary change oracle (XTREE) and a secondary verification oracle (Tuned Random Forest).

[1] P. Runeson and M. Host, "Guidelines for conducting and reporting case study research in software engineering," Empirical Software Engineering, vol. 14, pp. 131164, 2009

[2] P. R. M. H. Daniela Cruzes, Tore Dyba, "Case studies synthesis: Brief experience and challenges for the future," in ESEM11: the International Symposium on Empirical Software Engineering and Measurement, 2011.

We may have used these interchangeably in the first draft of our paper. This has been fixed in this revision.

How does this work related to the work by Arcelli Fontana, in particular: Arcelli Fontana, F., Mantyla, M.V., Zanoni, M. et al. Comparing and experimenting machine learning techniques for code smell detection, Empir Software Eng (2016) 21: 1143. doi:10.1007/s10664-015-9378-4?

Thank you for the reference. Our work differs from this specific paper in one key aspect — we attempt to assist reorganization of code and prevent needless reorganization while they focus on detecting code smells using machine learning techniques. Our work is much closer to other work by Arcelli Fontana [3]. We discuss this in §2.1.

Improving on one metric may cause degradation on another one. This issue could have been much more spelt out with good examples in the paper. It's stated that XTREE recommends (or suggests) increasing coupling while reducing LOC of a module. Sure, if you split a module into several smaller modules, or move code from one large module to other, smaller modules, usually the overall coupling will increase. But isn't that an avoidable consequence that will occur implicitly in the process of module splitting or moving code between modules? The way its formulated now gives the impression that the programmers should follow the recommendation of increasing the coupling, that is, as if its a conscious action. Or am I missing the point?

You are quite right, we have now rewritten associated text with **RQ5** in §6.5 to better explain the underlying connectedness of the metrics (Reviewer2e in §6.5).

To summarize, we were trying to convey the following information with this specific example:

1. Attempting to reduce certain code metrics (like LOC) necessitates increasing other metrics. Any recommendation that fails to respect these "co-changes" may not the practical to follow.

2. XTREE is aware of the relationship between metrics and always recommends changes to conjunctions of metrics. These changes are pragmatic in that XTREE sometimes suggests the same metric be either increased or decreased based on the dataset. As an example, consider Figure 8 in Ant coupling between objects (CBO) has to be reduced while in Lucene CBO has to be increased.

Developers are now aware of these interconnected changes. They can proceed to make these changes because of the historical evidence of its effectiveness.

In Section 4: "It can be difficult to judge the effects of removing bad smells. Code that is reorganized cannot be assessed just by a rerun of the test suite since such reorganizations may not change the system behavior (e.g., refactorings)." Isn't the point about removing bad smells to improve the code without changing the behavior (refactorings)? Why cannot a test suite be run before and after if the behavior is not changed?

It is true that refactoring improves the code by performing behavior preserving modifications. Evaluating this code with test suites that are designed to identify faults would not work for three reasons: (1) Test suites may not be designed to identify code smells, they only report pass/fail based on whether or not a specific module runs. It is entirely possible that a test case may pass for code that contains one or more code smells; (2) While smells are certainly symptomatic of design flaws, not all smells cause the system to fail in manner suitable for identification by test cases; and (3) It make take a significant amount of effort to write new test cases that identify bad smells, especially for stable software.

Finally, in our work we tackle reorganization which subsumes refactoring. Since not all reorganization efforts comprise of refactoring, that makes it difficult to assert when test case will pass or fail. Thus, using test suites is not a reliable option.

At your suggestion, we have now added this in §5.2 (please see text marked Reviewer2f).

*Figure 4 shows the effect of tuning, and the authors write: "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%." This is a good example of researcher bias. Your example is the most favorable example from your point of view. It is one of only two of the eight data sets that improved on both pd and pf.*

You are right ... there are only a few cases where we see an improvement in both recall and false alarm at the same time (poi, synapse and log4j). But, we don't think is due to a researcher bias. Instead, this due to the specifics of tuning and the goals used for this. We would like to draw your attention to two other kinds of improvements as a result of tuning:

1. Cases where there is a significant improvement in one measure (recall or false alarm) with no deterioration in the other measure (Ivy, Jedit, Lucene, and Xalan); and

2. Cases where there is a significantly large improvement in one measure with acceptable deterioration in the other metric (Poi and Ant).

We attribute these to the nature of multiobjective search performed by differential evolution. As highlighted by Fu et al. [4] (page 2, §2.2), defining objectives for tuning is a notoriously difficult task. They find that tuning for recall or false alarms independently can lead to some undesirable results. Tuning on recall yields oracles that have near 100% recall but with almost 100% false alarm. Similarly, tuning for only false alarm results in oracles that have 0% false alarm with 0% recall. We take their advice and tune on metrics that combine both precision and recall, thereby turning this into a multiobjective search. Results in Figure 5 (previously Figure 4) show recall and false alarms have reached a "trade-off" with maximum improvement, such is the nature of multiobjective differential evolution.

Thank you for your comments, we ought to have had a formal discussion of these in the paper. At your suggestion, these observations and reasons for the same have now been dealt with in much greater detail in the current version on the paper (see text marked with Reviewer2g in §5.2.1)

The authors state that if there are no historical records of defects, the results of this paper can be used as a guide (which results?). It is referred to Table 8 in the abstract but there is no Table 8. Is it meant to be Figure 8? In case, its very hard to understand how that figure could be used.

A good catch ... there seems to be a typo, Table 8 has now been replaced by Figure 9 in the abstract. Regarding the understandability of Figure 9, you are quite right in that we haven't provided sufficient information explaining this. As per your suggestion, we have now expanded the section that refers to this figure (see text prefixed with Reviewer2h in §6.5) to better explain how the conclusions from this figure may be used when historical records are absent.

References

[1] 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.

[2] 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.

[3] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda. Towards a prioritization of code debt: A code smell intensity index. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 16–24, Oct 2015.

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