2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement

An Empirical Examination of the Relationship Between Code Smells and Merge Conflicts

Iftekhar Ahmed*, Caius Brindescu*, Umme Ayda Mannan, Carlos Jensen, Anita Sarma
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR, USA
{ahmedi, brindesc, mannanu, carlos.jensen, anita.sarma }@oregonstate.edu

***Abstract*—Background:** Merge conflicts are a common occurrence in software development. Researchers have shown the negative impact of conflicts on the resulting code quality and the development workflow. Thus far, no one has investigated the effect of bad design (code smells) on merge conflicts. **Aims:** We posit that entities that exhibit certain types of code smells are more likely to be involved in a merge conflict. We also postulate that code elements that are both "smelly" and involved in a merge conflict are associated with other undesirable effects (more likely to be buggy). **Method:** We mined 143 repositories from GitHub and recreated 6,979 merge conflicts to obtain metrics about code changes and conflicts. We categorized conflicts into *semantic* or *non-semantic*, based on whether changes affected the Abstract Syntax Tree. For each conflicting change, we calculate the number of code smells and the number of future bug-fixes associated with the affected lines of code. **Results:** We found that entities that are smelly are *three times more likely* to be involved in merge conflicts. Method-level code smells (*Blob Operation* and *Internal Duplication*) are highly correlated with *semantic* conflicts. We also found that code that is smelly and experiences merge conflicts is more likely to be buggy. **Conclusion:** Bad code design not only impacts maintainability, it also impacts the day to day operations of a project, such as merging contributions, and negatively impacts the quality of the resulting code. Our findings indicate that research is needed to identify better ways to support merge conflict resolution to minimize its effect on code quality.

Keywords—Code Smell; Merge Conflict; Empirical Analysis; Machine Learning

I. Introduction

Modern software systems are becoming more and more complex and requires a large development team to develop and maintain. Modern Version Control Systems (VCS) have made parallel development easier by streamlining and coordinating code management, branching, and merging. This enables large teams to work together efficiently. But it has been shown that this process is sometimes halted when isolated private development lines are synchronized and the developer runs into merge conflicts. Conflicts distract the developers as they have to interrupt their workflow to resolve them. Developers have to reason about the conflicting changes and find an acceptable merging solution. This process of conflict resolution can itself introduce bugs. Prior work has found that in complex merges, developers may not have the expertise or knowledge to make the right decisions [14, 53] which might degrade the quality of the merged code.

Researchers have looked at many ways of preventing merge conflicts, and make developer's lives easier when they do occur. Researchers have proposed workspace awareness tools [6, 19, 34, 58, 60] that help prevent merge conflicts by making the developers aware of each other's changes. Also, new merge techniques [3, 4, 40] have been proposed that would reduce the number of merge conflicts. However, little research has been devoted to the causes of merge conflicts. Are there any endemic issues that arise from the design itself? We are interested in knowing whether the design of the codebase has an effect on the merge conflicts and what is its impact on the overall quality.

Just like merge conflicts, bad design can inflict pain on developers. Bad design makes maintenance and future changes difficult and error prone. Code smells, an indication of bad design, imply that the structure of the code is badly organized. This can lead to developers stepping on each other's toes as they make their changes. This, in turn, can lead to merge conflicts.

If there are "fundamental flaws" in the design itself, as the project grows, and the codebase grows in size and complexity, understanding and working around these "rough spots" becomes more challenging. Thus, the chances of creating a conflict increases because of the need to generate workarounds. This means that as projects grow, merge conflicts should be more likely to occur, especially around the smelly parts of the code. We aim to examine whether there is a correlation between the two, to examine whether such a link is credible.

In order to evaluate the design we look at the code smells [45]. We investigate if there is a connection between entities that contain code smells, the code smells they contain, and the merge conflicts that surround the smelly entities.

It is important to note that not all smells are created equal. Some might be more associated with a merge conflict than others. For example, a class is considered a *God Class* if it contains an oversized part of the entire functionality of the final product. Therefore, any changes have a high likelihood of involving changes in the *God Class.* When multiple developers are working, they all have a high likelihood of touching the *God class*. This can easily lead to merge conflicts down the road. If the changes involved are not trivial then the task of merging them will be not trivial as well.

In this paper, we investigate the following questions:

RQ1: Do program elements that are involved in merge conflicts contain more code smells?

RQ2: Which code smells are more associated with merge conflicts?

RQ3: Do code smells associated with merge conflicts affect the quality of the resulting code?

To answer these questions, we investigated 143 projects. Across them, we had 36,122 merge commits, out of which 6,979 were conflicting. We identified 7,467 code smells instances

***Both of the first two authors contributed equally**

978-1-5090-4039-1/17 $31.00 © 2017 IEEE
DOI 10.1109/ESEM.2017.12

across our whole corpus. We found that merge conflicts involved more "smelly" program elements than merges that did not conflict. Our results also show that not all code smells are created equal. Some are more likely to cause problems than others. When we looked at the difficulty of merge conflicts, we found that some of the smells are more likely to be involved in *semantic* merge conflicts than others. Finally, we found that code smells have a negative impact on code quality.

II. Related Work

A. Code smells and their impact

Various measures of software quality have been proposed. Boehm et al. [8], and Gorton et al. [31], to mention a few, have explored measures including completeness, usability, testability, maintainability, reliability, efficiency etc. Some of these metrics are difficult to measure, especially in the absence of requirement documents or other supporting information. Researchers have also used code smells as a measurement of software quality [48, 49], though smells are often focused on future maintainability issues. The concept of code smells was first introduced by Fowler [29]. Code smells are symptoms of poor design and implementation choices [29] in code base which eventually affect the maintainability of a software system [44]. Studies also showed that there is an association between code smells and bugs [46, 54] and code maintainability problems [29]. Code smells also leads to design debt. Zazworka et al. [67] found that the *God Class* smell is related to technical debt. Ahmed et al. [1] found how software gets worse over time in terms of design degradation. They analyzed 220 open source projects in their study and confirmed that ignoring the smells leads to "software decay".

Researchers have proposed many different approaches for detecting code smell, such as metric based [21, 22, 45, 46, 48] and meta-model based [52]. Researchers used different techniques for identifying code smells. Fontana et al. [27] used machine learning techniques for detecting code smells. Researchers also used both static analysis [21, 22, 46] and techniques that rely on the evaluation of successive versions of a software system [39, 45, 55].

B. Work related to code smells and bugs

Researchers have also considered the relationship between the presence of code smells and bug appearance in the code base. Khomh et al. [41] showed that classes affected by design problems ("code smells") are more likely to contain bugs in the future. Hall et al. [33] also found relationships between code smells and fault-proneness. According to their study some code smells indicate fault-proneness in the code base but the effect size is small (under 10%). Zazworka et al. [67] found that *God Classes* are fault-prone in some cases. Li et al. [48] also studied the relationship between code smells and the probability of faults in industrial systems, and found that the *Shotgun Surgery* smell was correlated with a higher probability of faults. To the best of our knowledge no work has tried to research on the relationship between code smells and how it impacts collaborative work flow, specifically merging individual works.

C. Merge conflicts

Several studies have been done on identification of conflicts and developers' awareness about potential conflicts. Awareness is frequently defined as an understanding of the activities of others to give a context for one's activities [24], which is a very important issue in Global Software Engineering (GSE) [57]. Researchers have looked at different techniques of avoiding merge conflicts by increasing the developer's awareness of the changes others made to the source code. Biehl et al. [6] proposed FastDash, which sends notifications about potential conflicts when two or more developers are modifying the same file. Another awareness tool called Syde by Hatori et al. [34] consider the source code changes at Abstract Syntax Tree (AST) level operations to detect conflicts by comparing tree operations. Da Silva et al. [19] introduced Lighthouse, which is another tool for increasing awareness among developers about the conflict. Palantír by Sarma et al. [58] detects the changes made by other developers and show them in a graphical, non-intrusive manner. Servant et al. [60] also presented a tool and visualization that can be used to understand the impact of developers' changes to prevent indirect conflicts.

Guimaraes et al. [32] introduce WeCode which continuously merges uncommitted and committed changes in the IDE to detect merge conflicts as soon as possible. Brun et al. [9] used the similar approach in Crystal, to detect both direct and indirect conflicts. A software development model presented by Dewan et al. [23] aims to reduce conflicts by notifying developers who are working on the same file.

D. Work related to merge conflict resolution

Researchers have also studied different ways of managing the merge of developers' changes to efficiently resolve conflicts. This resolution could be either in an automated way or by preserving and presenting a useful context for the developer trying to resolve the conflict. A comprehensive survey of merge approaches was done by Mens [51]. Apel et al. [3, 4] presented a merging technique called semistructured merge. This considers the structure of the code which is being merged. Operation based merging by Lippe et al. [47] considers all the changes performed during development, in addition to the result, when merging.

Kasi and Sarma [40] present a technique of avoiding merge conflicts by scheduling tasks in a way that the probability of a conflict is minimized. SafeCommit by Wloka et al. [65] uses a static analysis approach to identify changes in a commit with no test failure. They proposed to use this approach when detecting indirect conflicts.

E. Conflict categorization

Researchers have come up with different ways of categorizing conflicts. Sarma et al. [58] grouped conflicts into two categories. One is direct conflicts, where the changes conflict directly. The other is indirect conflicts, where the files don't conflict directly, but integrating the changes cause build or test failures. Similarly, Brun et al. [9], categorized conflicts as first level (textual) conflicts and second level (build and test failure) conflicts. Buckley et al. [10] proposed a taxonomy of changes based on properties like time of change, change history, artifact granularity etc. Their taxonomy deals with software changes in general or conflicts at a coarser level.

F. Tracking code changes and conflicts

Researchers have proposed various algorithms for tracking individual lines of code across versions of software. Canfora et al. [12] proposed an algorithm that uses Levenstein edit distance

to compute similarity of lines, matching "chunks" of changed code. Zimmerman et al. [68] proposed annotation graphs which works at the region level for tracking lines. Godfrey et al. [30] described "origin analysis", a technique for tracking entities across multiple revisions of a code base by storing inexpensively computed and easily comparable "fingerprints" of interesting software entities in each revision of a file. These fingerprints can then be used to identify areas of the code that are likely to match before applying more expensive techniques to track code entities. Finally, Kim et al. [42] propose an algorithm, SZZ, for tracking the origin of lines across changes.

III. Methodology

Our goal was to identify the effect of design issues on merge conflicts and the quality of the resulting code (whether these changes are associated with bug fixes or other improvements.)

Here we discuss the various steps of collecting data: (1) selecting the sample of projects for the study, (2) identifying which merge commits lead to merge conflicts, (3) tracking the lines of code through different versions and merges to investigate how the code evolved and which lines were associated with conflicts, (4) identifying code smells at the time of the conflicting merge commit. Next, we determine the nature of the code updates (e.g. was the commit a result of a bug fix or a new feature etc.) taking place on those lines. In order to do this, we manually classify a subset of the commits as bug-fix related or other. We train a machine-learning classifier to classify the rest. Finally, we build a model to predict the total number of bug fixes that would occur on a conflicting line that also contained a code smell. The following subsections describe each of these steps in detail.

A. Project Selection Criteria

We wanted to make sure that our findings would be representative of the code developed in real world, thus we selected active, open source projects hosted in GitHub. We decided to use Java as the language of focus. This decision was influenced by 2 factors: First, Java is one of the most popular languages (according to the number of projects hosted on Github and the Tiobe index [62]). The second was the availability of code smell detection tools for Java, as compared to other programming languages. Further, for ease of building and analyzing the code, we select projects using the Maven [2] build system.

We started by randomly selecting 900 projects, the first to show up when using the GitHub search mechanism. From these, we eliminated aggregate projects (which could skew our results), leaving 500 projects. After eliminating projects that did not compile (for reasons such as unavailable dependencies, or compilation errors due to syntax or bad configurations), 312 projects remained. Finally, we eliminated projects our AST walker, implemented using the GumTree algorithm [26], could not handle. This left us with a total of 200 projects.

Next, we removed projects that were too small, that is, having fewer than 10 files, or fewer than 500 lines of code. We also removed projects that had no merge conflicts. These selection criteria were used, since we are interested in the effect of design issues and merge conflicts in moderately large, collaborative projects. Our final data set contained 143 projects. Table I provides a summary of features and other descriptive information of the projects in our study.

We also manually categorized the domain of the projects by looking at the project description and using the categories used by Souza et al. [20]. Table II has the summary of the domains of the projects.

TABLE I. Project statistics

Dimension	Max	Min	Average	Std. dev.
Line count	542,571	751	75,795	105,280.1
Duration (Days)	6,386	42	1,674.54	1,112.11
# Developers	105	4	72.76	83.19
Total Commits	30,519	16	3,894.48	5,070.73
Total Merges	4,916	1	252.60	522.73
Total Conflicts	227	1	25.86	39.49

TABLE II. Distribution of Projects by domain

Domain	Percentage
Development	61.98%
System Administration	12.66%
Communications	6.42%
Business & Enterprise	8.10%
Home & Education	3.11%
Security & Utilities	2.61%
Games	3.08%
Audio & Video	2.04%

B. Code smell detection tool selection

We chose to use InFusion [36] to identify code smells because it has been found to identify the broadest set of smells [28]. Researchers have found that the metric-based approach identified by Marinescu [49] has the highest recall and precision (precision: 0.71, recall: 1.00) for finding most code smells [59]. InFusion uses this same principle and set of thresholds for identifying code smell, which was another reason for using InFusion. Researchers [1] have evaluated the smell detection performance of InFusion where they found it to have precision of 0.84, recall of 1.00 and an F-measure of 0.91.

C. Conflict Identification

Since Git does not record information about merge conflicts, we had to recreate each merge in the corpus in order to determine if a conflict had occurred. We used Git's default algorithm, the recursive merge strategy, as this is the most likely to used by the average Git project. From our sample of 143 projects we extracted 556,911 commits. This included 36,122 merge commits. The average number of merge commits was 253. Out of all the merges, 6,979 (19.32%) were identified as leading to a conflict. The distribution of merge conflicts is shown in Figure 1. We see that projects experience an average of 25 merge conflicts, or 19.32% of all merges. Merge conflicts, therefore, are a common part of the developer experience.

We then collected statistics regarding each file involved in a conflict. We tracked the size of the changes being merged, the difference between the two branches (in terms of LOC, AST difference, and the number of methods and classes involved). To determine the AST difference, we used the Gumtree algorithm [26]. We also tracked the number of authors involved in the merge.

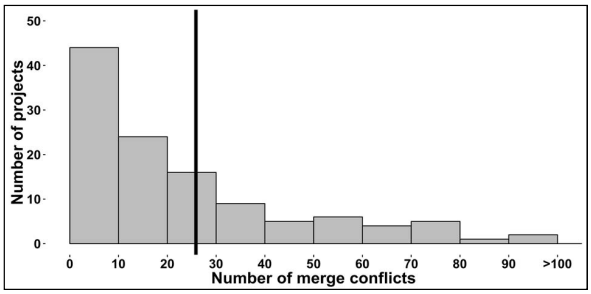

Fig. 1. Distribution of merge conflicts. The vertical line represents the mean (25.86)

D. Conflict Type Classification

To answer our second research question, we needed to categorize the conflicts based on the type of changes (e.g., whitespace or comment added vs. variable name changed).

We identified two categories of conflicts. The first one being *semantic* conflicts which requires understanding the program logic of the changes in order to successfully resolve the conflict. The other type of conflict is *non-semantic* which easier and less risky to resolve since they do not affect the programs' functionality. We manually classify 606 randomly sampled commits. We classify each conflict based on the type of changes causing the merge conflict (e.g., whitespace or comment added vs. variable name changed). Two of the authors coded 300 of these commits using qualitative thematic coding [26]. They achieved an inter-rater agreement of over 80% on 20% of the data: we obtained a Cohen's Kappa of 0.84. Having reached an agreement, one of the authors classified the remaining 306 commits. The codes and their definitions are given in Table III.

TABLE III. CONFLICT CATEGORIES

Category	Definition	Example
Semantic	Conflicts involving semantic changes	A refactoring and a bug fix involving the same lines.
Non-Semantic	Conflicting changes in formatting/comments	One of the branches contains only formatting changes (whitespace).

To train the classifier (to differentiate between *semantic* and *non-semantic* commits) we use a set of 24 features, including: the total size of the versions (LOC) involved in a conflict, the number of statements, methods and classes involved in the conflict. Details of the features are in the accompanying website [16]. We use the set of 606 (10%) commits as training data for a machine learning classifier. We used Adaptive Boost (AdaBoost) ensemble classifier that can only be used for binary classes. We categorized the 6,979 conflicting commits. We use 10 fold cross-validation to test the performance of our classier. The precision of predicting the *semantic* conflicts is high at 0.75.

E. Measuring Code Smells and Tracking Lines

For each of the 6,979 conflicting commits we collected the code smells that were associated with a conflict. We needed to track them to measure the effect of having the smell and being involved in a conflict on the quality of the resulting code.

We use GumTree [26] for our analysis, as it allows us to track elements at an AST level. This way we can track only the elements that we are interested in (statements), and ignore other changes that do not actually change the code. The GumTree algorithm works by determining if any AST node was changed, or had any children added, deleted or modified. The algorithm maps the correspondence between nodes in two different trees, which allows it to accurately track the history of the program elements. This algorithm has unique advantages over other line tracking algorithms, such as SZZ [42]. These advantages include: ignoring whitespace changes, tracking a node even if its position in the file changes (e.g. because lines have been added or deleted before the node of interest), and tracking nodes across refactorings, as long as the node stays within the same file. Using this technique, we can track a node even when it has been moved, for example, because of an extract method refactoring.

For each node (in the AST) involved in a conflict and having a smell, we identify all future commits that touched the file containing said node and tracked the AST node forward in time. For Java, it is possible for multiple statements to be expressed in the same line (e.g., a local variable declaration inside an if statement). In this case, we considered the innermost statement, as this gives the most precise results.

F. Commit Classification

In order to answer our third research question, we needed to categorize the type of change for a code commit. For our purpose, code commits can be broadly grouped into one of two categories: (1) bug-fixes and improvements (modifying existing code), and (2) Other — commits that introduced new features or functionality (adding new code) or commits that were related to documentation, test code, or other concerns. Two key problems with this classification are: (1) it is not always trivial to determine which category a commit falls under, and (2) larger projects see a huge amount of activity. Manual classification of all commits was not an option, and we decided to use machine learning techniques for this purpose, rather than limiting the statistical power of our study (especially as arbitrarily dropping the most active subjects would clearly potentially introduce a large bias into our results.)

In order to build a classifier, we randomly selected and manually labeled a set of 1,500 commits. The first two authors worked independently to classify the commits. Their datasets had a 33% overlap, which we used to calculate the inter-rater reliability. This gave us a Cohen's Kappa of 0.90. In our training dataset, the portion of bug-fixes was 46.30%, with 53.70% of the commits assigned to the Other category. Some keywords indicating bug-fixes or improvements were Fix, Bug, Resolves, Cleanup, Optimize, and Simplify, and their derivatives. Anything that did not fit into this pattern was marked as Other.

Not all bug-fixing commits include these keywords or direct reference to the issue-id; commit messages are written by the initial contributor, and there are few guidelines. A similar observation was made by Bird et al. [7], who performed an empirical study showing that bias could be introduced due to missing linkages between commits and bugs. This means that we are conservative in identifying commits as bug-fixes.

We trained a Naive-Bayes (NB) classifier and a Support Vector Machine (SVM) by using the SciKit toolset [56]. We used 10% of the data to train the classifier. We applied the classifiers to the training data using a 10-fold cross-validation. As before, we used the F1-score to measure and compare the performance of the models. The NB classifier outperformed the

SVM. Therefore, we used the NB classifier to classify our full corpus.

Table IV has the quality indicator characteristics of the NB classifier. Tian et al. [61], suggest that for keyword-based classification the F1 score is usually around 0.55, which also occurs in our case. While our classifier is far from perfect, it is comparable to "good" classifiers in the literature, and we believe it is unlikely for the biases to have a confounding effect on our analysis. Since our analysis only relies on relative counts of bug-fixes for statements, so long as we do not systematically undercount bug-fixes for only some statements, our results should be valid.

TABLE IV. NAIVE BAYES CLASSIFIER DETAILS

	Precision	Recall	F1-measure
Bug-fix	0.63	0.43	0.51
Other	0.74	0.86	0.80

For each line of code resulting from a merge conflict, we count the number of (future) commits in which it appears, as long as those commits are identified as bug-fixes. We stop the tracking when we encounter a commit that is classified as Other. Our reasoning is that once an element has seen a change that is not a bug-fix, it is no longer fair to assume that subsequent bug fixes are associated with the original merge conflict.

G. Regression analysis

In order to answer our third research question that is related to the effect of code smells on quality of the resulting code, we needed to build a regression model to identify the impact of code smell on the number of bug-fixes that occur on lines of code that are associated with a code smell *and* a merge conflict. We use Generalized Linear Regression [15]. The dependent variable (count of bug fixes occurring on smelly and conflicting lines) follows a Poisson distribution. Therefore, we use a Poisson regression model with a log linking function.

In order to build our model, we collect information about the smells and the conflicts. We use Understand [63] to count the number of references to, and from other files to the files that are involved in a conflict. We collect this information as a proxy for the importance of the file. We assume that the more a file is referenced by other files, the more central that file is, and hence more important. Any change in these central files can increase the chance of a change being required in other files, and therefore lead to multiple developers making changes to these files, which can in turn lead to conflicting changes.

We also collect the following factors for each commit such as the difference between the two merged branches in terms of LOC, AST difference, and the number of methods and classes being affected. Our intuition is that larger "chunks" of changes should have a higher chance of causing a conflict. We also calculate the number of authors who made commits to the branches that were merged, since there is a higher likelihood of conflicts if multiple developers are involved.

We also determine the experience level of each developer by splitting them into two categories: *core* and *non-core*. To calculate the category for each developer, we split the development history into quarters. For each commit a developer is classified as *core* if he is in the top 20% of the developers in that quarter (calculated by the number of commits). Otherwise he is *noncore.* We use this process because, in open source projects, authors come and go. Also, an author can be classified as *core* and *non-core* in different quarters, depending on his contribution to the project.

After collecting these metrics, we checked for multi-collinearity using the Variance Inflation Factor (VIF) of each predictor in our model [15]. VIF describes the level of multicollinearity (correlation between predictors). A VIF score between 1 and 5 indicates moderate correlation with other factors, so we selected the predictors with VIF score threshold of 5. This step was necessary since the presence of highly correlated factors forces the estimated regression coefficient of one variable to depend on other predictor variables that are included in the model.

IV. RESULTS

A. RQ1: Do program elements that are invovled in merge conflicts contain more code smells?

As a first step, we collect the total number of code smells for each of the 6,979 conflicting commits in our dataset. Table V contains the percentage of each smell and the percentage of projects that have a particular smell. We find that *external* and *internal duplication* have a much higher instance than others when considering the percentage of smells in the dataset. However, about 50% of projects have *Data Class* and *SAP Breakers* smells.

TABLE V. PERCENTAGE OF CODE SMELLS

Smell	% of smells in the full dataset	% of projects w/ smell
External Duplication	42.79	22.53
Internal Duplication	**34.05**	**23.80**
Feature Envy	4.04	28.42
Data Clumps	3.71	20.36
Intensive Coupling	3.50	14.30
Data Class	3.18	48.05
Blob Operation	2.58	30.05
Sibling Duplication	2.35	10.86
SAP Breakers	1.52	52.76
God Class	**0.89**	**19.10**
Schizophrenic Class	0.58	20.00
Message Chains	0.33	5.34
Tradition Breaker	0.17	6.33
Refused Parent Bequest	**0.19**	**5.25**
Shotgun Surgery	0.01	1.72
Distorted Hierarchy	**0.003**	**0.36**

We next compare the mean number of code smells associated with each merge commit, for cases when they conflict and for cases when they do not conflict. Note that a commit can involve multiple files, and a file can contain multiple smells. We calculate the total number of smells for each file. For example, a conflicting merge commit in the `commandhelper` project (with the SHA1 of `a91faa`) contains one conflicting file, and that conflicting files contains a total of 8 smells.

The mean number of smells in conflicting program elements is **6.54,** whereas the mean for non-conflicting program elements is **1.92**. The results are statistically significant (Mann-Whitney test, U=6.24e6, p<4.77e-10.); we use the non-parametric Mann-Whitney test since our population is not normally distributed. Therefore, we find that *program elements that are involved in merge conflicts are, on average, more smelly than entities that are not involved in a merge conflict.*

B. *RQ2: Which code smells are more associated with merge conflicts?*

Next, we compare the occurrence of each individual smell across conflicting and non-conflicting commits. Since we are performing multiple tests, we have to adjust the significance value accordingly to account for multiple hypothesis correction. We use the Bonferroni correction, which gives us an adjusted p-value of 0.0031.

For 12 out of 16 total smells, we find significant differences (Mann-Whitney test, $\alpha<0.0031$) between the means of conflicting and non-conflicting commits. The conflicting commits have a higher incidence of smells. Table VI presents the results for code smells where the difference was significant along with the p-values of individual comparisons..

TABLE VI. MEAN NUMBER OF SMELLS IN CONFLICTS VS. NON-CONFLICT COMMITS CALCULATED PER COMMIT

Smell	Smells in conflicts	Smells in non conflict	p-value
God Class	1.23	0.25	0.0001
Data Clump	0.65	0.27	0.0001
Sibling Duplication	0.58	0.10	0.000001
Data Class	0.47	0.12	0.000001
Distorted Hierarchy	0.45	0.05	0.000001
Unnecessary Coupling	0.33	0.10	0.0001
Internal Duplication	0.24	0.08	0.000001
SAP Breaker	0.12	0.07	0.000001
Tradition Breaker	0.10	0.05	0.00007
Blob Operation	0.07	0.06	0.0001
Message Chain	0.04	0.03	0.00062
Shotgun Surgery	0.01	0.00769	0.00021

The following are the top 5 smells in terms of their (mean) numbers per conflict: *God Class, Data Clump, Sibling Duplication, Data Class* and *Distorted Hierarchy*. It is worth noting that the distribution of smells per conflict (Table VI) is different from Table V. This is because in Table VI we are looking only at the smells that affect the entities involved in merge conflicts, whereas Table V shows all the smells in the project. This discrepancy is an effect of the fact that merge conflicts exhibit a different smell pattern compared to the overall project.

Next, we perform two steps. First, we investigate the correlation between each smell and the merge conflicts to identify which of the above smells are more strongly associated with conflicts. Then, we categorize merge conflicts into *semantic* and *non-semantic* conflicts to further explore the associations of smells to these types of conflicts.

Code smells and conflicts. We perform a correlation analysis between the count of smells and merge conflicts to distill which of the smells from Table VI are more closely associated with conflicts, and should be attended to. We use the Kendall correlation test because it is a non-parametric test and it is more accurate with a smaller sample size. As we perform the tests for each smell, we are splitting out data into smaller chunks. Therefore, the Kendall correlation test is more appropriate.

We find that, except for *External Duplication, Schizophrenic Class*, *SAP Breaker* and *Data Class* all smells are correlated with merge conflicts (Kendall correlation test, $\alpha<0.0031$). We report the statistically significant results in Table VII.

The three strongest correlation to conflicts are with the following smells: *God Class*, *Internal Duplication* and *Distorted Hierarchy*. These smells all relate to cases where object-oriented design principles of encapsulation and structuring is not well used, leading to problems with developers making conflicting parallel changes. We discuss these reasons further in Section V.

TABLE VII. CORRELATION BETWEEN CONFLICT AND SMELL COUNT

Smell	Correlation	p-value
God class	0.18	<0.0001
Internal Duplication	0.17	<0.0001
Distorted Hierarchy	0.13	<0.0001
Refused Parent Bequest	0.10	<0.0001
Message Chain	0.10	<0.0001
Data clump	0.09	<0.0001
Feature Envy	0.09	<0.0001
Tradition Breaker	0.09	<0.0001
Blob Operation	0.08	<0.0001
Shotgun Surgery	0.07	<0.0001
Unnecessary Coupling	0.05	0.00007
Sibling Duplication	0.04	0.00021

Types of conflicts and their classification. Not all conflicts are the same, some involve changes to the actual code structure and require the developer to understand the logic behind the changes before they can be integrated (*semantic* conflicts), whereas others can be formatting or cosmetic changes (*non-semantic*). *Semantic* conflicts are inherently harder to resolve. Therefore, we investigate whether specific types of code smells are more likely to occur with *semantic* conflicts. We use the conflict classification methodology in Section III-D.

Recall, we manually labeled 606 conflicts to classify them into *semantic* or *non-semantic*, which we then use for the automated classification of 6,979 commits. We present the distribution between the manual and automatic classification in Table VIII. The distributions of *semantic* and *non-semantic* conflicts in the automatically classified data match the distribution of our manual labeling (training data), which shows the efficacy of the automated classifier.

TABLE VIII. CONFLICT TYPES BASED ON THEIR FREQUENCY OF OCCURRENCE

Category	# of Conflicts	% of total (classifier)	% of total (training)
Semantic	5,250	75.23%	76.12%
Non-Semantic	1,729	24.77%	23.88%

Semantic conflicts are more common (76.12% in the manually labeled data and 75.23% in the automated classified data), as compared to the *non-semantic* conflicts (23.88% in manually labeled and 24.77% in automated classified data).

Semantic conflicts and code smells: To understand if there is any correlation between *semantic* conflicts and the types of code smells we perform the Kendall correlation test for each smell in the presence of *semantic* merge conflicts (in our total dataset). We use the Kendall correlation test and found significant correlation ($\alpha<0.0031$) only for *Internal Duplication* and *Blob Operation*. Table IX contains all correlations, where the cells marked with ** are significant.

Since the correlation for both *Blob Operation* and *Internal Duplication* are small, we perform an odds-ratio test to understand which of these smells are more likely to be involved in a *Semantic* merge conflict, as compared to entities that do not have these smells, but were involved in a conflict. Since we are performing two comparisons, we have to adjust the significance

value to adjust for multiple hypothesis testing. Like in the previous sections, we performed a Bonferroni correction, which gives us significance value of α=0.0025 to test at.

TABLE IX. SMELL CATEGORIES FOR SEMANTIC CONFLICTS (SIGNIFICANCE LEVEL α=0.0031)

Smell	Correlation	p-value
Blob Operation **	0.05	0.0030
Internal Duplication **	0.07	0.0002
Message Chain	0.01	0.4970
Refused Parent Bequest	0.03	0.0492
SAP Breaker	-0.02	0.2652
Schizophrenic Class	-0.03	0.0832
Shotgun Surgery	-0.008	0.6597
Sibling Duplication	0.031	0.1029
Tradition Breaker	0.018	0.3291
Unnecessary Coupling	-0.01	0.5681
Data Class	-0.02	0.1524
Data Clumps	0.030	0.1103
Distorted Hierarchy	0.034	0.0670
Feature Envy	0.009	0.6206
God Class	0.050	0.0072

We performed an odds ratio test (Fisher's exact test) for the *Blob Operations* and find that they are 1.7 times more likely to be involved in a *Semantic* merge conflict (odds ratio: 1.77, p=0.0024). *Blob Operations* are methods that are very complex and have many responsibilities. Therefore, any change to the method will likely impact multiple lines, which may intersect with logical changes made by another developer to the same method. This explains the high likelihood of their involvement in *Non-Semantic* conflicts.

For *Internal Duplication*, we found that they are 1.55 times more likely to be involved in merge conflict (odds ratio: 1.55, p=0.0001.) We attribute this to the fact that, because of duplication, a change has to be repeated in multiple locations. This increases the chances of developers making overlapping changes.

C. RQ3: Do code smells associated with merge conflicts affect the quality of the resulting code?

We aim to model the effects of code smells on the bugginess of a line of code involved in a merge conflict. As defect prediction literature has already identified several factors (e.g., the size of the module under investigation [25], number of committers [64], centrality of files [13]) that affect bugginess, we include them in our model also. Table X lists the final set of factors that we use, which include metrics that are code-based (F1, F2), change-related (F5-F8), author-related (F3, F4), and code-smells. We compute whether a developer is core or non-core based on our methodology in Section III-G.

To answer our third research question, we build two Generalized Linear Models (GLM). The first contains the number of code smells as a factor, and the second does not. The first (Poisson regression) model is built with a log linking function as explained in Section III-G. After filtering the factors with VIF $\leq$ 5, we had a set of 8 factors out of 43 factors. All eight factors were statistically significant (see Table X). The predicted value is the total number of bug fixes occurring on a line of code that was involved in a merge conflict. Note that smell count was a significant factor in the model ($p<0.05$), with an estimate of 0.427.

The McFadden Adjusted R^2 [35] of this model is 0.47. We calculated McFadden's Adjusted R^2 as a quality indicator of the model because there is no direct equivalent of R^2 metric for Poisson regression. The ordinary least square (OLS) regression approach to goodness-of-fit does not apply for Poisson regression. Moreover, adjusted R^2 values like McFadden's cannot be interpreted as one would interpret OLS R^2 values. McFadden's Adjusted R^2 values tend to be considerably lower than those of the R^2. Values of 0.2 to 0.4 represent an excellent fit [35]

TABLE X. POISSON REGRESSION MODEL PREDICTING BUG-FIX OCCURRENCE ON LINES OF CODE INVOLVED IN A MERGE CONFLICT

Factor#	Factor	Estimate	p-value
F1	In Deps	3.195	<0.0001
F2	Out Deps	-0.053	<0.0001
F3	Noncore author	-3.799	<0.0001
F4	No. Authors	0.129	<0.0001
F5	No. Classes	-0.373	<0.0001
F6	No. Methods	0.244	<0.0001
F7	AST diff	0.001	<0.0001
F8	LOC diff	0.00002571	<0.0001
F9	Number of Smells	0.427	<0.0001

To understand the impact that code smells have, we built the *same* model by removing the total number of smells as a factor. This decreased the adjusted-R^2 from 0.47 to 0.44. We can therefore conclude that code smells have a significant impact on the final quality of the code. Since McFadden's adjusted R^2 penalizes a model for including too many predictors, had the code smells not mattered, removing it could have increased the adjusted-R^2 instead of reducing it.

V. DISCUSSION

To the best of our knowledge, we are the first to investigate the association of code smells with that of merge conflicts, and their impact on the bugginess of the merged results (line of code). We find that program elements that are involved in merge conflicts contain, on average, 3 times more code smells than program elements that are not involved in a merge conflict.

Not all code smells are equally correlated to merge conflicts. 12 out of the 16 code smells that co-occur with conflicts are significant associated with merge conflicts. The top five code smells from this list are: *God class, Message Chain, Internal Duplication, Distorted Hierarchy* and *Refused Parent Bequest*. Interestingly, the only (significant) code smells associated with *Semantic* conflicts are *Blob Operation* and *Internal Duplication.*

All the above code smells arise when developers do not fully exploit the advantages of object-oriented design, leading to high coupling, duplication, or large containers. These factors lay the groundwork for parallel conflicting efforts, where developers step on each other's toes. For example, the *Blob Operation* is a large and complex method that grows over time becoming hard to maintain. In such a situation, multiple developers may need to make changes to the same method and, therefore, collide when merging. Similarly, *Internal Duplication* arises when code is duplicated, which bloats methods and makes it hard to ensure all clones evolve in the same way. In such a situation, developers might have to "touch" multiple parts of the method to ensure all clones are being updated, causing situations of parallel, conflicting edits.

It is interesting to observe that *Semantic* merge conflicts are associated with smells at the method level. For example, the *Blob Operation and Internal Duplication* smells are 1.77 times

and 1.55 times, respectively, more likely to be present in a *semantic* conflict as compared to a *non-semantic* conflict. This indicates that bloated methods or duplicated code in methods increase the spread of the change a developer is likely to make, which in turn increases the likelihood of two or more changes conflicting during a merge. Prior work has associated code duplication with negative consequences such as increased maintenance cost [11,43] and faults [5,38]. Our findings indicate that duplication also negatively impacts the collaborative workflow by making it difficult to merge changes.

It is worth noting that while smells, such as *God Class* have a significant correlation with overall merge conflicts, they do not have a significant correlation with *semantic* merge conflicts. We posit that a large container (class) with cohesive logical units (methods) can lead to multiple developers making parallel changes that are localized to specific areas (methods) and do not intersect. In these cases, when changes are merged conflicts can arise because of the movement of code or formatting changes (*non-semantic* conflicts). The same reasoning is also applicable for *Distorted Hierarchy, Refused Parent Bequest* and *Message Chain*. In contrast, as discussed earlier method-level smells seem are correlated with *semantic* conflicts.

To the best of our knowledge, ours is the first empirical study to investigate the effects of merge conflicts and code smells on the bugginess of code. We found that the presence of code smells on the lines of code involved in a merge conflict has a significant impact on its bugginess (see Table X). Including code smells as a factor increases the McFadden's adjusted R^2 value from 0.44 to 0.47. Since McFadden's adjusted R^2 penalizes a model for including too many predictors, an increase in the value signifies that adding code smells as a factor was valuable. We find that factors such as incoming-dependencies and the number of code smells have the highest correlation estimate, indicating their importance to the model.

We find that some factors, such as non-core author, number of classes, and outward dependencies have a negative effect on bugginess. This is counter intuitive. We had assumed that changes from multiple non-core authors are more likely to be buggy. We believe that the following reasons lead to this surprising outcome. It might be the case that non-core contributors are more thorough and put more effort towards submitting code that is less bug prone. Or it might be the process via which newcomers' contributions are accepted. For example, core developers might pay more attention to changes coming from non-core contributors. Further empirical studies on the differences in review processes for core vs. non-core developers will be interesting. We also found that the number of classes involved in a conflict has a negative correlation to its bugginess. This might be because changes that involve multiple classes are more likely to be refactoring or licensing changes, and therefore, less likely to introduce bugs.

Implications: Our findings have a number of implications for software practitioners, tool builders and researchers.

Code smells have been historically associated with maintenance issues, which are known to be a problem in the long term. However, developers are often unaware of code smells. Yamashita et al. found that a considerable portion (32%) of developers did not know about code smells [66]. Our findings shed a different light on the impact of code smells and on the importance of addressing them. Our results show that code smells are an immediate concern for day-to-day activity such as merging changes.

Merge conflicts delay the project by requiring an examination of the conflict, and disrupting the developers' workflow. Anecdotal evidence shows that developers hate resolving conflicts. Developers are known to follow informal processes (e.g., check in partial code, email the team about impending changes etc.) or rush to commit their work in an effort to avoid having to resolve conflicts [14]. A developer may also choose to delay the incorporation of others' work, fearing that a conflict may be hard to resolve [14]. Such processes can have a detrimental effect on team productivity and morale. This situation can only become worse as the project evolves on two fronts. First, the number of code smells is likely to increase as the project ages [1]. Second, there is a likelihood of increase in merge conflicts as more developers start to contribute. Our results indicate that practitioners should pay more attention to code smells, as it will not only make the code quality better, but will also help them minimize the number of merge conflicts they need to resolve.

Practitioners, when investigating the root cause of a merge conflict can start by looking for smelly program elements in the code. Moreover, since changes that involve entities containing code smells are more likely to lead to semantic merge conflicts, integrators (or code reviewers) should pay particular attention to and attempt to remove code smells when reviewing commits. Practitioners should also pay attention to "good" software engineering processes when they deal with smelly program elements. For example, when changes are being made to smelly parts of the code base developers should merge more frequently and perform more thorough code reviews.

Our results show that code smells are a good predictor of merge conflict and the level of difficulty of that conflict. Therefore, tool builders can use the information of incidence of code smells to support distributed work – either in predicting likelihood of conflicts or their difficulty. Code smells can also be used as a factor to schedule tasks (e.g., program elements that have code smells should not be edited in parallel) or assign tasks (e.g., developers with higher experience should work on smelly program elements).

Our results have implications for researchers. Since code smells together with merge conflicts can predict bugginess, researchers can use this information in bug prediction models to increase their effectiveness. To the best of our knowledge, no merge conflict prediction tool exists. Our results show that code smells have a strong association with merge conflicts, therefore, researchers can use this information to predict impending merge conflicts. Our results also have implications in testing. For example, increasing the test coverage of smelly lines that were involved in a merge conflict can be used as an objective/fitness function in the field of search based software engineering.

VI. THREATS TO VALIDITY

Our research findings may be subject to the concerns that we list below. We have taken all possible steps to neutralize the impacts of these possible threats, but some couldn't be mitigated and it's possible that our mitigation strategies may not have been effective.

Bias due to sampling: Our samples have been from a single source - Github. This may be a source of bias, and our findings

may be limited to open source programs from Github and not generalizable to commercial programs. However, the threat is minimal since we analyze a large number of projects spanning eight different domains.

Bias due to tools used: The smell detection tool we used uses static code analysis to identify smells and research shows that code smells that are "intrinsically historical" such as *Divergent Change*, *Shotgun Surgery* and *Parallel Inheritance* are difficult to detect by just exploiting static source code analysis [55]. So the number occurrence of such "intrinsically historical" smells should be different when historical information based smell detection technique is used.

Secondly, we used the Gumtree algorithm [26] for tracking program elements across commits. However, the algorithm used is unable to track program elements across renames or movement to another folder. Further, refactoring that involves modification of scope, such as moving the code out of the current compilation unit also causes the algorithm to lose track of the program element after refactoring.

Bias Due to using classifiers: We use machine learning to group conflicts into the two categories, and to determine whether a commit was a bug-fix. As with any classifier, we have some mislabeling. While our results do not require those results to be anywhere near perfect, this threat is low as our classifiers have good F1-measure and high precision.

Regarding the bug-fix classifier, our recall and precision measures are on par with past work [7]. Since our analysis relies on relative count of bug fixes, as long as we do not systematically undercount bug fixes, our results are valid.

Finally, we have assumed that all bugs were found and fixed by developers when we use it as a metric of bugginess of merged lines of code. This may not always be true, and hence our results are conservative.

VII. Conclusions

In this paper, we study the history of 143 open source projects, from which we extract 6,979 merge conflicts to see if there is any correlation between code smells and merge conflicts. We found that entities involved in merge conflicts contain almost 3 times more code smells than non-conflicting entities.

To have a better understanding of the effect of code smells on merge conflicts, we categorized conflicts into *semantic* conflicts – changes to the AST and hard to resolve – and *non-semantic* – changes that are cosmetic. We found two method-level code smells (*Blob Operation* and *Internal Duplication*) to be significantly correlated with *semantic* conflicts. More specifically, methods that contained the *Blob Operation* and *Internal Duplication* smells were more likely to be involved in a *semantic* merge conflict, by 1.77 times and 1.55 times respectively. We also found that code smells have a significant impact on the final quality of the code. Count of code smells was a significant factor when we modeled the bugginess of lines of code involved in a merge conflict.

Our results show that code smells, thought to be a maintenance issue and often neglected by practitioners, have an immediate impact in how distributed development is managed. Their presence is not only associated with difficult merge conflicts (*semantic*), but also with the likely-hood of bugs getting introduced in the code base.

Acknowledgments

This work was funded in part by NSF IIS-1559657,CCF-1560526. This work was also funded in part by IBM. We would also like to thank the Oregon State University HCI group for their input and feedback on the research.

References

[1] Ahmed, I., Mannan, U. A., Gopinath, R., & Jensen, C. An empirical study of design degradation: How software projects get worse over time. Empirical Software Engineering & Measurement (ESEM), 2015, pp.1-10.

[2] Apache Software Foundation. Apache maven project. http://maven.apache.org

[3] Apel, S., Leßenich, O., & Lengauer, C. Structured merge with auto-tuning: balancing precision and performance. International Conference on Automated Software Engineering, 2012, (pp. 120-129).

[4] Apel, S., Liebig, J., Brandl, B., Lengauer, C., & Kästner, C. Semistructured merge: rethinking merge in revision control systems. 13th European conference on Foundations of software engineering, 2011, (pp. 190-200).

[5] Bakota, T., Ferenc, R., & Gyimothy, T. Clone smells in software evolution. IEEE International Conference on Software Maintentance, 2007, (pp. 24-33).

[6] Biehl, J. T., Czerwinski, M., Smith, G., & Robertson, G. G. FASTDash: a visual dashboard for fostering awareness in software teams. Human factors in computing systems, 2007, (pp. 1313-1322).

[7] Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., & Devanbu, P. Fair and balanced?: bias in bug-fix datasets. European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2009, (pp. 121-130).

[8] Boehm, B. W., Brown, J. R., & Lipow, M. (Quantitative evaluation of software qualityinternational conference on Software engineering, 1976 (pp. 592-605).

[9] Brun, Y., Holmes, R., Ernst, M. D., & Notkin, D. Proactive detection of collaboration conflicts. European conference on Foundations of software engineering, 2011, (pp. 168-178).

[10] Buckley, J., Mens, T., Zenger, M., Rashid, A., & Kniesel, G. (2005). Towards a taxonomy of software change. Journal of Software Maintenance and Evolution: Research and Practice, 17(5), 309-332.

[11] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's University, Kingston, Canada, Tech. Rep. 2007-541, 2007.

[12] Canfora, G., Cerulo, L., & Di Penta, M. (2007, May). Identifying Changed Source Code Lines from Version Repositories. In MSR (Vol. 7, p. 14).

[13] Cataldo, M., & Herbsleb, J. D. (2013). Coordination breakdowns and their impact on development productivity and software failures. IEEE Transactions on Software Engineering, 39(3), 343-360.

[14] Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. 2003. "Breaking the Code", Moving Between Private and Public Work in Collaborative Software Development. ACM SIGGROUP Conference on Supporting Group Work (GROUP '03) pp. 105–114.

[15] Cohen, J., Cohen, P., West, S. G., & Aiken, L. S. (2013). Applied multiple regression/correlation analysis for the behavioral sciences. Routledge.

[16] Companion Website: https://goo.gl/ORpLkU

[17] Costa, C., Figueiredo, J. J., Ghiotto, G., & Murta, L. (2014). Characterizing the Problem of Developers' Assignment for Merging Branches. International Journal of Software Engineering and Knowledge Engineering, 24(10), 1489-1508.

[18] Cunningham, W. (1993). The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger, 4(2), 29-30.

[19] Da Silva, I. A., Chen, P. H., Van der Westhuizen, C., Ripley, R. M., & Van Der Hoek, A. Lighthouse: coordination through emerging design. 2006 OOPSLA workshop on eclipse technology eXchange (pp. 11-15).

[20] De Souza, L. B. L., & de Almeida Maia, M. Do software categories impact coupling metrics?. In Mining Software Repositories (MSR), 2013, (pp. 217-220).

[21] Deligiannis, I., Shepperd, M., Roumeliotis, M., & Stamelos, I. (2003). An empirical investigation of an object-oriented design heuristic for maintainability. Journal of Systems and Software, 65(2), 127-139.

[22] Deligiannis, I., Stamelos, I., Angelis, L., Roumeliotis, M., & Shepperd, M. (2004). A controlled experiment investigation of an object-oriented design heuristic for maintainability. Journal of Systems and Software, 72(2), 129-143.

[23] Dewan, P., & Hegde, R. (2007). Semi-synchronous conflict detection and resolution in asynchronous software development. ECSCW 2007, 159-178.

[24] Dourish, P., & Bellotti, V. Awareness and coordination in shared workspaces. 1992 ACM conference on Computer-supported cooperative work (pp. 107-114). ACM.

[25] El Emam, K., Benlarbi, S., Goel, N., & Rai, S. N. (2001). The confounding effect of class size on the validity of object-oriented metrics. IEEE Transactions on Software Engineering, 27(7), 630-650.

[26] Falleri, J. R., Morandat, F., Blanc, X., Martinez, M., & Monperrus, M. (2014). Fine-grained and accurate source code differencing. 29th international conference on Automated software engineering (pp. 313-324).

[27] Fontana, F. A., Mäntylä, M. V., Zanoni, M., & Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering, 21(3), 1143-1191.

[28] Fontana, F. A., Mariani, E., Mornioli, A., Sormani, R., & Tonello, A. (2011, March). An experience report on using code smells detection tools. 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshop (ICSTW), pp. 450-457.

[29] Fowler, M., & Beck, K. (1999). Refactoring: improving the design of existing code. Addison-Wesley Professional.

[30] Godfrey, M. W., & Zou, L. (2005). Using origin analysis to detect merging and splitting of source code entities. IEEE Transactions on Software Engineering, 31(2), 166-181.

[31] Gorton, I., & Liu, A. (2002). Software component quality assessment in practice: successes and practical impediments. 24th International Conference on Software Engineering (pp. 555-558). ACM.

[32] Guimarães, M. L., & Silva, A. R. (2012) Improving early detection of software merge conflicts. 34th International Conference on Software Engineering (ICSE), (pp. 342-352).

[33] Hall, T., Zhang, M., Bowes, D., & Sun, Y. (2014). Some code smells have a significant but small effect on faults. ACM Transactions on Software Engineering and Methodology (TOSEM), 23(4), 33.

[34] Hattori, L., & Lanza, M. (2010, May). Syde: A tool for collaborative software development. 32nd International Conference on Software Engineering-Volume 2 (pp. 235-238).

[35] Hensher, D. A., & Stopher, P. R. (Eds.). (1979). Behavioural travel modelling. London: Croom Helm.

[36] InFusion, http://www.intooitus.com/inFusion.html. (accessed at January 2014)

[37] Izurieta, C., & Bieman, J. M. (2007,). How software designs decay: A pilot study of pattern evolution. Empirical Software Engineering and Measurement. (pp. 449-451).

[38] Juergens, E., Deissenboeck, F., Hummel, B., & Wagner, S. (2009). Do code clones matter?. International Conference on Software Engineering. (pp. 485-495).

[39] Kagdi, H., Gethers, M., Poshyvanyk, D., & Collard, M. L. (2010, October). Blending conceptual and evolutionary couplings to support change impact analysis in source code. In Reverse Engineering (WCRE), 2010 17th Working Conference on (pp. 119-128). IEEE.

[40] Kasi, B. K., & Sarma, A. Cassandra: Proactive conflict minimization through optimized task scheduling. 2013 International Conference on Software Engineering (pp. 732-741). IEEE Press.

[41] Khomh, F., Di Penta, M., Guéhéneuc, Y. G., & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Software Engineering, 17(3), 243-275.

[42] Kim, S., Zimmermann, T., Pan, K., & James Jr, E. (2006,). Automatic identification of bug-introducing changes. International Conference on Automated Software Engineering, ASE'06. (pp. 81-90).

[43] Koschke, R. (2007). Survey of research on software clones. In Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[44] Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. Ieee software, 29(6), 18-21.

[45] Lanza, M., & Marinescu, R. (2007). Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer Science & Business Media.

[46] Li, W., & Shatnawi, R. (2007). An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. Journal of systems and software, 80(7), 1120-1128.

[47] Lippe, E., & Van Oosterom, N. (1992, November). Operation-based merging. In ACM SIGSOFT Software Engineering Notes (Vol. 17, No. 5, pp. 78-87). ACM.

[48] Marinescu, R. (2001). Detecting design flaws via metrics in object-oriented systems. International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, (pp. 173-182).

[49] Marinescu, R. (2004, September). Detection strategies: Metrics-based rules for detecting design flaws. In Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on (pp. 350-359). IEEE.

[50] Martin, R. C. (2003). Agile software development: principles, patterns, and practices. Prentice Hall PTR.

[51] Mens, T. (2002). A state-of-the-art survey on software merging. IEEE transactions on software engineering, 28(5), 449-462.

[52] Moha, N., Rezgui, J., Guéhéneuc, Y. G., Valtchev, P., & El Boussaidi, G. (2008). Using FCA to suggest refactorings to correct design defects. In Concept Lattices and Their Applications (pp. 269-275). Springer Berlin Heidelberg.

[53] Nieminen, A. (2012,). Real-time collaborative resolving of merge conflicts. International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), (pp. 540-543).

[54] Olbrich, S., Cruzes, D. S., Basili, V., & Zazworka, N. (2009). The evolution and impact of code smells: A case study of two open source systems. 2009 3rd international symposium on empirical software engineering and measurement (pp. 390-400). IEEE Computer Society.

[55] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. Automated software engineering (ASE), (pp. 268-278).

[56] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12(Oct), 2825-2830.

[57] Sarma, A., & Van Der Hoek, A. (2006, October). Towards awareness in the large. In Global Software Engineering, 2006. ICGSE'06. International Conference on (pp. 127-131).

[58] Sarma, A., Noroozi, Z., & Van Der Hoek, A. (2003). Palantír: raising awareness among configuration management workspaces. International Conference on Software Engineering, (pp. 444-454).

[59] Schumacher, J., Zazworka, N., Shull, F., Seaman, C., & Shaw, M. (2010). Building empirical support for automated code smell detection. International Symposium on Empirical Software Engineering and Measurement (p. 8).

[60] Servant, F., Jones, J. A., & Van Der Hoek, A. (2010). CASI: preventing indirect conflicts through a live visualization. ICSE Workshop on Cooperative and Human Aspects of Software Engineering (pp. 39-46).

[61] Tian, Y., Lawall, J., & Lo, D. (2012, June). Identifying linux bug fixing patches. In Proceedings of the 34th International Conference on Software Engineering (pp. 386-396). IEEE Press.

[62] Tiobe,http://tiobe.com/index.php/content/paperinfo/tpci/index.html

[63] Understand™ Static Code Analysis Tool. (2017).

[64] Weyuker, E. J., Ostrand, T. J., & Bell, R. M. (2008). Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. Empirical Software Engineering, 13(5), 539-559.

[65] Wloka, J., Ryder, B., Tip, F., & Ren, X. (2009, May). Safe-commit analysis to facilitate team software development. In Proceedings of the 31st International Conference on Software Engineering (pp. 507-517). IEEE Computer Society.

[66] Yamashita, A. F., & Moonen, L. Do developers care about code smells? An exploratory survey. WCRE, 2013 (Vol. 13, pp. 242-251).

[67] Zazworka, N., Shaw, M. A., Shull, F., & Seaman, C. Investigating the impact of design debt on software quality. 2nd Workshop on Managing Techinical Debt, 2011

[68] Zimmermann, T., Kim, S., Zeller, A., & Whitehead Jr, E. J. Mining version archives for co-changed lines. 2006 International Workshop on Mining Software Repositories (pp. 72-75).