e-Informatica Software Engineering Journal, Volume 15, Issue 1, 2021, pages: 115–132, DOI 10.37190/e-Inf210106

Software Deterioration Control Based on Issue Reports

Omid Bushehrian*, Mohsen Sayari*, Pirooz Shamsinejad*

**Department of Computer Engineering and Information Technology, Shiraz University of Technology, Shiraz, Iran*

bushehrian@sutech.ac.ir, sayari.mohsen@gmail.com, p.shamsinejad@sutech.ac.ir

Abstract

Introduction: Successive code changes during the maintenance phase may cause the emergence of bad smells and anti-patterns in code and gradually results in deterioration of the code and difficulties in its maintainability. Continuous Quality Control (QC) is essential in this phase to refactor the anti-patterns and bad smells.

Objectives: The objective of this research has been to present a novel component called Code Deterioration Watch (CDW) to be integrated with existing Issue Tracking Systems (ITS) in order to assist the QC team in locating the software modules most vulnerable to deterioration swiftly. The important point regarding the CDW is the fact that its function has to be independent of the code level metrics rather it is totally based on issue level metrics measured from ITS repositories.

Methods: An issue level metric that properly alerts us of bad-smell emergence was identified by mining software repositories. To measure that metric, a Stream Clustering algorithm called ReportChainer was proposed to spot Relatively Long Chains (RLC) of incoming issue reports as they tell the QC team that a concentrated point of successive changes has emerged in the software.

Results: The contribution of this paper is partly creating a huge integrated code and issue repository of twelve medium and large size open-source software products from Apache and Eclipse. By mining this repository it was observed that there is a strong direct correlation (0.73 on average) between the number of issues of type "New Feature" reported on a software package and the number of bad-smells of types "design" and "error prone" emerged in that package. Besides a strong direct correlation (0.97 on average) was observed between the length of a chain and the magnitude of times it caused changes to a software package.

Conclusion: The existence of direct correlation between the number of issues of type "New Feature" reported on a software package and (1) the number of bad-smells of types "design" and "error prone" and (2) the value of "CyclomaticComplexity" metric of the package, justifies the idea of Quality Control merely based on issue-level metrics. A stream clustering algorithm can be effectively applied to alert the emergence of a deteriorated module.

Keywords: Code Smells, Issue report, maintainability, document classification

1. Introduction

Mining open-source software repositories and particularly bug repositories managed by Issue Tracking Systems (ITS) such as BugZilla [1] and Jira [2] has recently attracted much attraction among the software research community. Many researchers have studied the possible correlations among stored knowledge in bug repositories and software quality aspects. By interpreting the number of reported bugs on a specific software version as quality indicator of that version, some works have focused on the correlation among the number of previously reported bugs (and hence changes) and the number of bugs in future release [3, 4]. Maintainability has been another quality aspect of interest and some research works have been dedicated to find meaningful correlations

Submitted: 27 October 2020; Revised: 12 June 2021; Accepted: 17 June 2021; Available online: 25 August 2021

between software maintainability and defect metrics extracted from ITS repositories [5].

The maintainability of a software product is usually defined as its readiness to accept successive modification very easily and with minimum effort [6]. These modifications are due to requested new features, reported bugs, performance difficulties or adapting the software to a new environment and are carried out by the maintenance team. Some studies have argued, particularly in open-source software, neither time lags in fixing bugs nor the distribution of bugs can be considered as the direct representative metrics of maintainability [7]. On the other hand, many research works in the field of software maintainability have proved that the maintainability of software after delivery is significantly dependent to its original design quality [8]. As a result, building the predictive models to assist the designers to better assess the maintainability of their future code based on the current design quality metrics at very first stages of development life cycle has attracted much attention in this research field [8–10]. In addition to the original design of the software, the quality of code modifications during the maintenance phase also matters and affects the maintainability of the software and its future defects number and severity [11]. It is now widely accepted that the maintainability of a software product could be measured by: (1) the number of residual or emerged faults in the maintenance phase (corrective maintainability), (2) the extent to which the code is understandable (adaptive maintainability) which is greatly affected by the degree of exploiting the software design patterns and best practices in development process and (3) the extent to which it is modifiable (perfective maintenance) which is again affected by the amount of anti-patterns and bad-smells [12] in the code [5, 7, 8].

Successive changes made by the developers may cause the emergence of bad smells in code and gradually result in code deterioration and lowering the code maintainability indirectly. Hence continuous Quality Control (QC) is essential in this phase to avoid formation of anti-patterns and bad smells in code. However this QC process could be very tedious or even ineffective task without the help of assisting tools. Since the change requests (and issue reports) are recorded and tracked in ITSs, we believe that these software tools could be equipped with a recommender component that enables the QC team to spot software modules (packages) that are most vulnerable to early deterioration and to put those modules in their priority list for code inspection and refactoring. We call this component CDW (Code Deterioration Watch) and it is capable of reporting the packages that need immediate quality check by QC team to see if the refactoring is necessary. The fundamental requirement in designing CDW is that it should be able to estimate the deterioration level of software modules using issue metrics rather than code metrics. This is the key difference between this research and similar studies where CDW does its function merely by relying on ITS repository data and independent of code repository. We believe that this ability is so important due to the fact that code QC has to be carried out based on a priority list in specific time intervals and this priority list is provided by CDW which continuously analyzes incoming issue streams efficiently. The high priority modules for refactoring are those that have become deteriorated and hence error-prone due to the successive changes performed not in compliance with best-practices and standard design patterns.

The function of CDW is simply based on this general hypothesis: "modules absorbing higher number of issues are those with higher number of (born) code smells and are the hot spots for refactoring". We studied a variety of open-source software repositories from Apache [13] and Eclipse [14] to answer these research questions:

RQ1: Can the QC team evaluate the quality level of the code, which is under successive changes during the maintenance phase, only by observing issue-related metrics such as the number of reported issues on a software package? Is the type of reported issues important in this evaluation?

RQ2: Is there an effective stream clustering method to categorize incoming sequence of issue reports such that a bloated category be truly interpreted as the concentration of frequent

changes on a specific software package and hence be reported as possible point of deterioration?

The rest of this paper is organized as follows: Section 2 explains the related works, Section 3 explains the concept of "code deterioration" and its relation to bad-smells, Section 4 presents the proposed continuous Quality Control model, Section 5 reports the results of experiments, the discussion and justification of the results are presented in section 6, Section 7 provides the threats to the validity of the study and finally Section 8 concludes the paper.

2. Related works

In the field of software maintainability, there are many research works dedicated to the maintainability prediction of software based on the knowledge collected in early stages of SDLC [8]. In [9] the superiority of the dynamic metrics over static metrics for maintainability prediction is studied. They concluded that the dynamic metrics outperform the static ones regardless of the machine learning algorithm used for prediction.

Extracting useful knowledge from bug repositories to estimate quality and maintainability of an open-source software has been the subject of many studies. In [3] authors concluded that the number of bugs, as software quality indicator, in the ith release has no significant correlation with the change size of its previous release. In [4] a class level quality assurance metric named Qi was defined and the correlation of the number of defects and Qi was analyzed but they observed no significant correlation among them. In [7] the maintainability of some open- source software was studied empirically and they reported that neither the time lag of reported bugs nor the distribution of bug reports can represent the maintainability indicator.

In [15] a recommender system to advice developers to avoid bad smells and apply quality practices during the programming is presented. They have built a quality model which is continuously updated based on the reported issues and the (detected) bad smells that have triggered the issue. The SZZ algorithm [16] which identifies the root cause of an issue has been applied by this study to track down the earliest change that has given birth to the exception. Subsequently the bad smells detected in code snippet identified as the root cause of the exception is related to the issue. Machine learning algorithm have been applied to build the quality model. In [17] to improve the software quality metrics and remove bad smells a multi-objective evolutionary algorithm is proposed by which the best sequence of refactoring activities is sought in a large search space of possible solutions. To obtain this a predicting model based on time series is applied to estimate the impact of each sequence of refactoring actions on the future software quality. In [10] thirty different software quality prediction models were studied. Using two standard datasets they concluded that regression and LWL outperformed others.

There are some studies on the relationship between the amount of code-smells and the bug proneness level of the software. In [18] it was shown that adding smell-related features (code-smell intensity) to the bug prediction models could improve the accuracy of the prediction models. The impact of presence of anti-patterns on the change and fault-proneness of the classes has been investigated in [19]. The results confirmed that the classes involving anti-patterns are more change and fault-prone. A Systematic Literature Review has been conducted and reported in [20] on the impact of code-smells on software bugs. The adverse effects of bad architectural decisions (architectural smells) on the maintainability of the software in terms of number of forthcoming issues and increased maintenance efforts have been studied in [21].

The classification or clustering of bug reports has also been the subject of some previous studies. The classification of bug reports are used to predict a variety of factors regarding them. For instance in [23] a two-phased classifier has been proposed to predict the files likely to be fixed using the bug report textual description. In [24] a clustering method based on EM (Expectation Maximization) and X-means has been proposed to categorize bug reports according to their subject similarities. They have used topic modeling to vectorize bug reports and subsequently applied a labeling algorithm to characterize each cluster.

There are also some studies on refactoring prioritization. In [25] a machine learning approach for classification of code smell severity to prioritizing the refactoring effort has been presented. They have reported a relatively high correlation between the predicted and actual severity by modeling the problem as an ordinal classification problem. In [26] a semi-automated refactoring prioritization method to assist developers has been presented. They have applied a combination of three criteria: past modifications history, the relevance of the smell to the architecture and the smell type to rank the refactoring activities.

The contribution of this paper is to propose a novel model of Software Quality Control which enables the QC team to judge about the internal quality of software, constantly changed by developers, without the need of source code analysis and merely by monitoring the incoming issue reports and their sequence. To this end first, a thorough correlation analysis between the code quality metrics (the number of bad-smells) and issue-level metrics (issue absorption rate) has been conducted. Subsequently a new stream clustering method is presented to effectively categorize incoming sequence of issue reports such that a bloated category truly indicates the existence of a software package with high issue absorption rate. It is important to note that in contrast to the previous studies on refactoring prioritization, the proposed method uses the issue-level metrics to find the top-module to refactor without needing to analyze or access the source code.

3. Code deterioration and bad-smells

There are a variety of bad-smell and anti-patterns introduced in the literature that may emerge in the code gradually due to the subsequent changes made by the development team [12]. The PMD static source analyzer [22] has presented a very good categorization of bad-smells in its documentations (Code Style, Design, Error-prone, Doc-

Table 1. Bad-smells according to the categorization presented in [22]

Category	Some Bad smells in this category	Example

Design
(46 bad-smells)
	AbstractClassWithoutAnyMethod,
	ClassWithOnlyPrivateConstructorsShouldBeFinal,
	CouplingBetweenObjects, CyclomaticComplexity
	DataClass, ExceptionAsFlowControl
	ExcessiveClassLength, GodClass,
	ImmutableField, LawOfDemeter,
	LogicInversion, LoosePackageCoupling,
	NPathComplexity,. . .
		DataClass:
		
		public class DataClass {
		public int bar = 0;
		public int na = 0;
		private int bee = 0;
		public void setBee(int n) {
		bee = n;
		}
		}

Error Prone
(98 bad-smells)
	AssignmentInOperand,
	AssignmentToNonFinalStatic,
	AccessibilityAlteration,
	AssertAsIdentifier,
	BranchingStatementAsLastInLoop,
	CallingFinalize,
	CatchingNPE, CatchingThrowable,
	DecimalLiteralsInBigDecimalConstructor,
	DuplicateLiterals, EnumAsIdentifier,
	FieldNameMatchingMethodName,
	FieldNameMatchingTypeName,
	InstanceofChecksInCatchClause,
	LiteralsInIfCondition,
	LosingExceptionInformation,. . .
		AssignmentInOperand:
		
		public void bar() {
		int x = 2;
		if ((x = getX()) == 3) {
		System.out.println("3!");
		}
		}

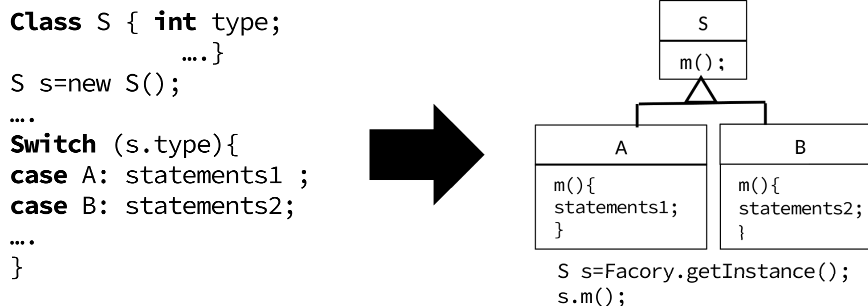


Figure 1. Replacing switch/case statements with polymorphism to reduce the "CyclomaticComplexity" value

Table 2. Issue categories [2]

Issue Type	Example	Issue Ref.	Studied in this paper
Bug	New version of Java 11 seems does not work well	NETBEANS-5636	Yes
New Feature	Have python-archives also take tar.gz	FLINK-22519	Yes
Improvement	Upgrade Kotlin version in Kotlin example to 1.4.x	BEAM-12252	No
Task	Remove landmark directories from web and shim	YUNIKORN-662	No
Sub-Task	Optional removal of fields with UpdateRecord	NIFI-8243	No
Test	Remove some Freon integration tests	HDDS-5160	No
Umbrella	Add SQL Server functions	TRAFODION-3146	No
Documentation	SQL DataFrameReader unescapedQuoteHandling parameter is misdocumented	SPARK-35250	No

umentation, Multithreading, Performance and Security) and among those we have focused on "Design" and "Error-Prone" categories since they refer to much more general forms of anti-patterns compared to other categories that contain more particular subjects. Some bad-smells in either categories along with examples are presented in Table 1.

Apart from these bad-smells we have also analyzed the method-level complexity using two well-known metrics: "Cyclomatic Complexity" and "NpathComplexity" as their high values are very good indicators of missing fundamental design patterns such as strategy, composite, proxy, adapter and many others that are based on polymorphism rather than conditional logics. The former is the number of decision points in the code and the latter is the number of full paths from the beginning to the end of the block of a method [27]. The refactoring practice corresponding to the high method complexity (measured using "CyclomaticComplexity" and "NpathComplexity") is illustrated in Figure 1.

These aforementioned bad-smells are code-level structures (or measures) that alert us of deteriorated code. However the aim of this research is to investigate issue-level metrics that does the same without relying on the code repository and hence make the QC activity possible by merely watching the issue repository. In the Analysis section (Section 5) it will be argued that the number of issues of type "New Feature" reported on a software package is an effective issue-level metric to inform the QC team of the code deterioration extent. See Table 2 for different issue categories and those that are involved in this study.

4. Code deterioration assessment model

The proposed model which incorporates the QC component, called CDW (Code Deterioration Watch) into the ITS, is illustrated in Figure 2. This component keeps track of the reported issues and categorizes them incrementally to detect relatively long sequences of related reports as a sign of possible code deterioration. By analyzing the

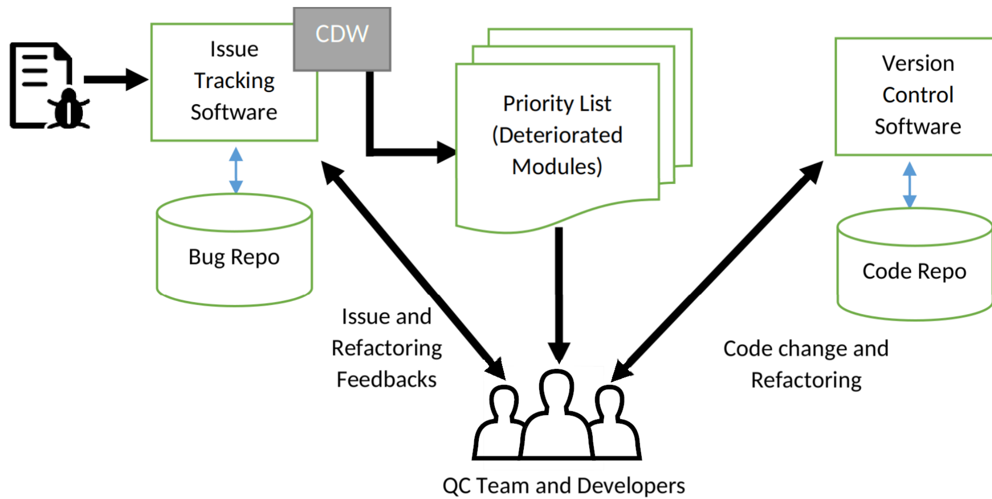


Figure 2. The code quality control based on issue monitoring

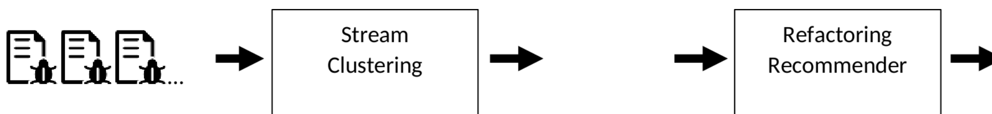


Figure 3. The CDW sub-components

long sequences of reports, CDW prioritizes the software modules to be scrutinized by the QC team for refactoring activities. There are two main sub-components of CDW working together to produce the final refactoring recommendation as shown in Figure 3. The Stream Clustering that splits the incoming sequence of issue reports into a set of chains and the Refactoring Recommender.

Ideally CDW should have the capability of notifying the QC team of the packages that are being changed frequently far more than others as they are code segments very likely to get deteriorated soon and need immediate attention for refactoring (this hypothesis will be verified in Section 6). Even if CDW be able to alert the QC team of existence of such packages without identifying them, it would be very helpful yet. Note that identifying these packages accurately is possible by analyzing the code repository at the later time, however CDW is part of the ITS (and not the version control) and it is supposed to give us insights about the evolution of the software merely by monitoring the incoming sequence of issue reports (issue metrics rather than code metrics).

As the Stream Clustering sub-component clusters the incoming issue reports into chains the idea is to spot Relatively Long Chains (RLC) of incoming issue reports as they tell the QC team that a concentrated point of successive changes has emerged in the software (this correlation will be discussed in Section 6). RLCs are those that their lengths (the number of issue reports in the chain) exceed the average chain length significantly (as much as threshold ß). Note that if all chains are long, none of them is considered as RLC due to the fact that RLC concept is based on significant size difference in a group and not the absolute size itself.

We define the "target" of each chain as the package which is expected to incur majority of the changes as the issue reports in that chain are being resolved. The "target hit number" of a chain is also defined as the expected change frequency of the chain target. For instance in a chain of three reports: R1, R2 and R3 labeled with {P1, P2, P3}, {P1, P3} and {P1}, respectively, as the packages to be changed, the chain target is P1 and its hit number is three.

Obviously a chain with a relatively high target hit number is telling us that a package (the

target) is being changed frequently far more than others. In Section 5 it will be verified that the chain size is a good metric to identify chains with a relatively high target hit number.

4.1. Stream Clustering

A simple stream clustering called "Report-Chainer" is presented here that splits the sequence of issue reports into chains based on their similarity to the previously formed chains. A split threshold controls formation of a new chain. First all documents are vectorized using Tf-Idf method [28] and cosine similarity is applied to compute the similarity of the current issue report with previous ones. If the similarity values are less than the split threshold then it is added to new chain otherwise it is added to the most similar chain (see Algorithm 1).

Vectorizing document d in a collection of N documents using Tf-idf method consists of two steps: first, the frequency of each term t in d is counted (denoted $tf_{t,d}$ and then the value of $tf_{t,d}$ is scaled using the idf_t (inverse document frequency) value:

$$idf_t = \frac{N}{df_t} \tag{1}$$

Where df_t is the number of documents in the collection containing term t. The value of Tf-idf corresponding to term t in document d is calculated using the following formula [28]:

$$Tfidf_{t,d} = tf_{t,d} \times idf_t \tag{2}$$

In fact the Tf-idf method assigns lower weights to the terms with no or very little discriminating power in a document. To compute the similarity of two documents $d1$ and $d2$, vectorized using the Tf-idf method, the cosine similarity has been applied [28]:

$$\mathrm{sim}(d1, d2) = \frac{(V(d1).V(d2))}{(|V(d1)| \times |V(d2)|)} \tag{3}$$

Where $V(d1)$ and $V(d2)$ denote the vector representation of $d1$ and $d2$, respectively, obtained using the Tf-idf method. The advantage of using the cosine similarity method over the ordinary method of computing the vector distances is that the cosine similarity formula is insensitive to the documents' length.

In the next section it will be shown that there is a significant linear correlation between the length of a chain and the magnitude of times that the chain target has been changed. Accordingly the longer chains are good candidates for QC attention. We will also show that, at least for 10 cases studies, not only can a split threshold be found to result a high correlation value but also this value is bounded.

4.2. Refactoring Recommender

The Refactoring Recommender component takes a set of chains C as its input and produces the recommendation by selecting, from the candidate list, those chains whose size differences with the average chain size are greater than threshold β (line 13 in Algorithm 2). These chains are consid-

Algorithm 1. Sequence Clustering Algorithm

```
maxSim = 0
chainNum = −1
thisVec = TF_IDF(r)
for Each vec in C do
    Sim = cosineSim(thisVec, vec)
    if Sim > maxSim then
        maxSim = sim
        chainNum = vec.chainNum
    end if
end for
if maxSim < t then
    chainNum = C.newChain()
end if
C.add(thisVec, chainNum)
```

Algorithm 2. Refactoring recommender algorithm

```
1:  Algorithm RefactoringRecommender(Chains C): List
2:  List r = ∅
3:  Package p
4:  int issueCnt = 0
5:  int chainCnt = 0
6:  for Each chain c in C  do
7:      if c. numberOfNotVisitedIssues()> α then
8:          issueCnt += c.numberOfNotVisitedIssues()
9:          chainCnt += 1
10:     end if
11: end for
12: for Each chain c in C  do
13:     if c.numberOfNotVisitedIssues() − (issueCnt / chainCnt)) > β then
14:         p = targetAnalyze(c)
15:         r.add(p)
16:     end if
17: end for
18: return r
19:
20: Algorithm On_RefactoringCompleted(chain c)
21: for Each Issue s in C  do
22:     s.visited = True
23:     r.add(p)
24: end for
```

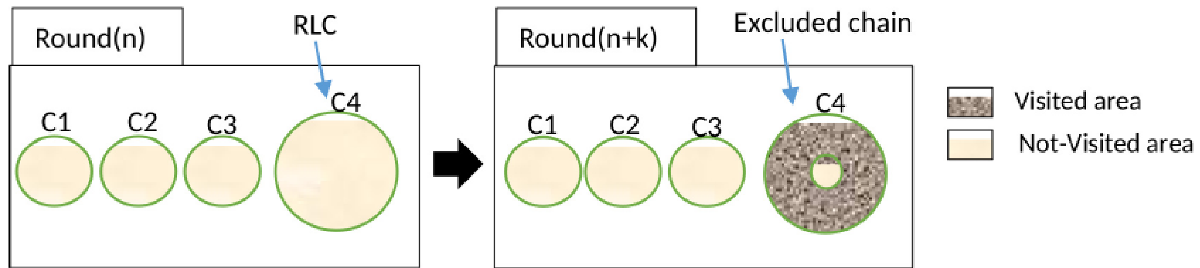


Figure 4. Issues have been categorized in four chains; In round n of recommendation, candidate set is $\{C1, C2, C3, C4\}$ and $C4$ is detected as RLC; during next k rounds $C4$ is excluded from the candidate set and no RLC is reported by the algorithm

ered as RLCs. Subsequently the targetAnalyze() function determines the target package of each RLC (line 14). This function can be carried out using supervised machine learning methods as presented in [23] or be done manually by experts. As soon as a recommended refactoring is performed by the QC team, all issues in the corresponding chain are excluded from the subsequent rounds of process by labeling them as "Visited" (line 20–24). Moreover the chain is excluded from the candidate list in the subsequent rounds of process until the number of "Not-Visited" issues reaches threshold α (line 7). This prevents the algorithm to falsely identify most of the candidate chains as RLCs since recently recommended chain has a few number of "Not Visited" issues and hence moves down the average chain size significantly (see Figure 4). A reasonable value for a could be the average chain size of the current candidate list.

5. Analysis

The objective of our experiments was first to study the correlation between the issue-level metrics and code-level metrics and second to evaluate

the proposed *RepChainer* algorithm. There are two issue-level metrics to study: Bug Absorption Degree (BAD) and Feature Absorption Degree (FAD) defined as the number of bugs and the number of new features reported on a software module in a period of time respectively. The code-level metrics are "CyclomaticComplexity", "NpathComplexity" and the number of detected "Design+ErrorProne" bad-smells as explained in Section 3. The software module granularity was chosen to be the package (a set of related classes). The rationale behind choosing the package granularity is to have more specialized task assignment (i.e. refactoring tasks) to developers. Since packages are often focused on specific subjects they can easily be assigned to developers who are specialized in the respective area to refactor all the classes of them. Moreover another set of experiments were conducted to evaluate the accuracy and usefulness of the stream clustering algorithm used in detecting bloated categories(RLCs) of issue reports.

5.1. Dataset

To do the experiments, a dataset with the schema shown in Figure 5 of medium and large scale open-source software repositories from Apache and Eclipse were created. Table 3 summarizes the products metadata. For each product, its Git [29] repository was cloned and analyzed using a program written with Node.Js. This program extracted all the commits and their associated objects from the repository using the isoMorphic-Git [30] library. During the analysis phase the code metrics: "CyclomaticComplexity", "NpathComplexity" and the number of detected "Design+ErrorProne" bad-smells were also measured using the PMD source code analyzer [22].

```
const cp = require('child_process');
cp.execSync("e:\\pmd\\bin\\pmd.bat
−dir e:\\pmd\\input_cloudstack
−format xml −R e:\\pmd\\ru.xml >
e:\\pmd\\output_cloudstack\\out.xml");
cp.execSync("e:\\pmd\\bin\\pmd.bat
−dir e:\\pmd\\input_cloudstack
−format xml −R category/java/design.xml,
category/java/errorprone.xml,>
e:\\pmd\\output_cloudstack\\out.xml");
```

Listing 1. Javascript code snippet to invoke PMD program using a custom ruleset: ru.xml (top) and the predefined "design" and "error prone" rule sets (bottom)

The Javascript code snippet to invoke PMD program is shown in Listing 1. A custom or built-in rule set has to be passed to PMD to analyze the source code accordingly. For detecting "Design+ErrorProne" bad-smells the built-in rule sets: category/java/design.xml and category/java/errorprone.xml were used. To measure "CyclomaticComplexity" and "NPathComplexity" metrics a custom rule set ru.xml was used as listed in Listing 2.

Due to the size of repositories the analysis took over a week to complete on a cluster of core-i7 PCs during which the analyzer program was running round the clock. To the best of our knowledge such an integrated dataset of issues, measured code metrics and bad-smells and metadata of all commits on Eclipse and Apache products has not been published elsewhere and we are working to make it online soon.

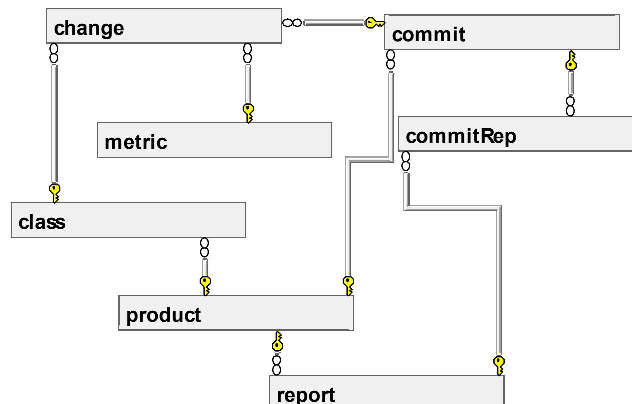


Figure 5. Dataset schema

The issues reported on products were imported into "report" table from the respective ITS. Apache-Jira and Eclipse-Bugzilla ITSs are accessible at [31] and [32] respectively.

5.2. Correlation Analysis

Assume that N_i and B_i are the number of resolved issues of type "New Feature" and "Bug"

Table 3. Products metadata

Product	License	Language	No. of Classes	No. of packages	No. of reported issues	No. of analyzed commits	ITS
Cloudstack	Apache	Java, Python	16100	639	3451	5565	JIRA
Geode	Apache	Java	27277	961	3805	6565	JIRA
Spark	Apache	Java, Python, Scala	8494	490	13289	15044	JIRA
Camel	Apache	Java	37702	2393	9655	18299	JIRA
Geronimo	Apache	Java	13163	846	2127	3115	JIRA
Hadoop	Apache	Java	27045	1168	3671	15902	JIRA
Hbase	Apache	Java	8255	261	9445	11164	JIRA
Myfaces	Apache	Java	2805	219	1594	2640	Bugzilla
4diac.ide	Eclipse	Java	1598	255	1126	504	Bugzilla
Acceleo	Eclipse	Java	896	210	969	520	Bugzilla
Common	Eclipse	Java	1496	177	105	287	Bugzilla
App4mc	Eclipse	Java	1370	68	131	186	Bugzilla

```xml
<?xml version="1.0"?>
<ruleset name="Custom␣Rules"
xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0
␣␣␣https://pmd.sourceforge.io/ruleset_2_0_0.xsd">
 <description>
  My custom rules
 </description>
 <rule ref="category/java/design.xml/CyclomaticComplexity">
  <properties>
   <property name="classReportLevel" value="1" />
   <property name="methodReportLevel" value="1" />
   <property name="cycloOptions" value="" />
  </properties>
  </rule>
 <rule ref="category/java/design.xml/NPathComplexity">
  <properties>
   <property name="reportLevel" value="1" />
  </properties>
 </rule>
</ruleset>
```

Listing 2. PMD custom rule-set applied to measure “CyclomaticComplexity” and “NPathComplexity” metrics

on package P_i, respectively, (issues for which P_i was not modified are excluded) and $I_i = N_i \bigcup B_i$ denotes the number of all resolved issue types on P_i. Moreover $\Delta b_{i,m}$ denotes the variation of code metric m during the successive changes of P_i:

$$\Delta b_{i,m} = m_{\max,i} - m_{\text{init},i} \qquad m \in \left\{ \begin{array}{l} \text{"CyclomaticComplexity"}, \\ \text{"NpathComplexity"}, \\ \text{"Design + ErrorPron"} \end{array} \right\} \tag{4}$$

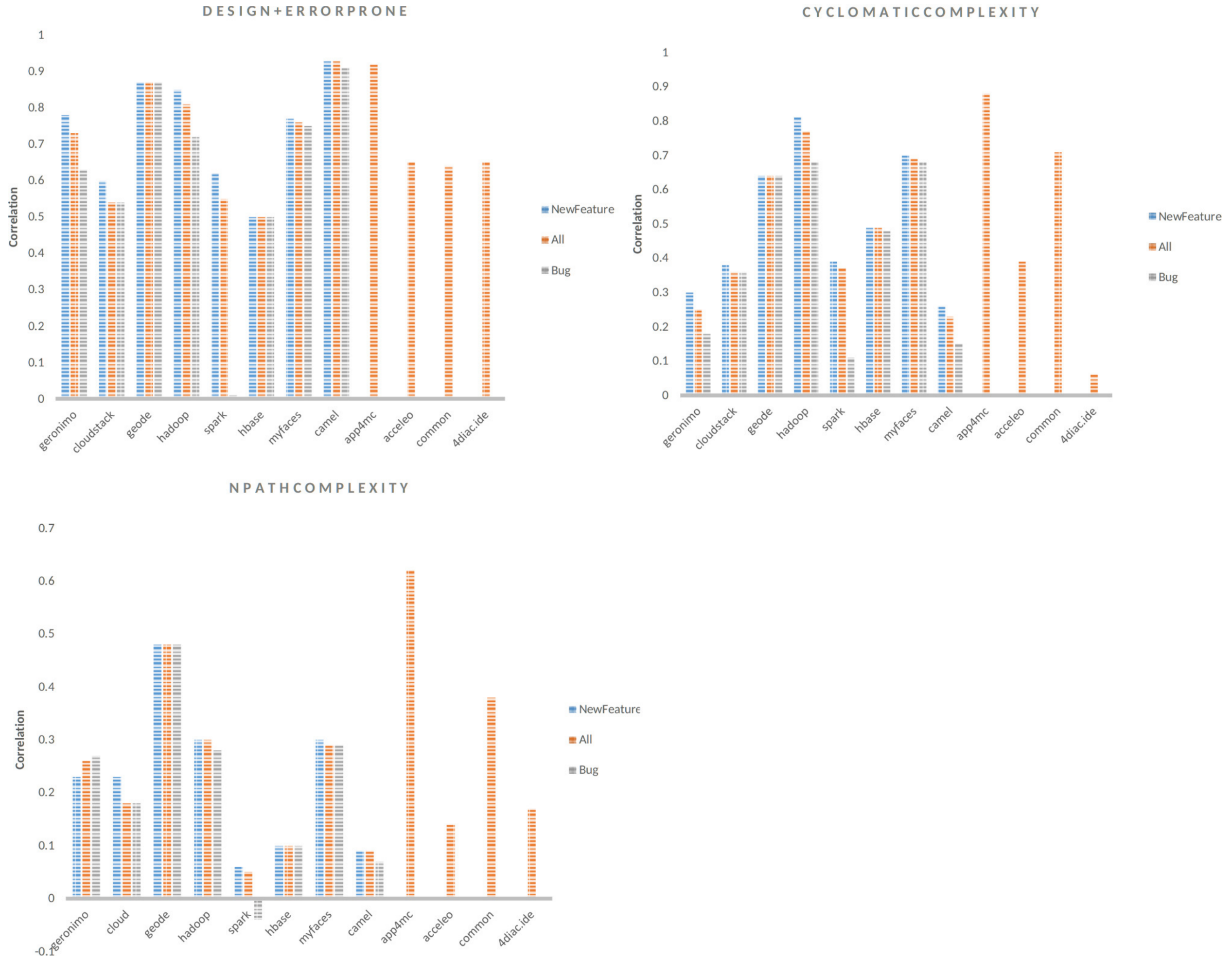


Figure 6. The Pearson correlation analysis on 12 products: (top left): "Design+errorProne" bad smells, (top right) "CyclomaticComplexity" metric and (bottom) "NpathComplexity" metric

Where $m_{\mathrm{max},i}$ and $m_{\mathrm{init},i}$ are the maximum and initial values of code metric m measured over all variations of classes in P_i:

$$m_{max,i} = \sum^{C \in P_i} \mathrm{Max}_{\mathrm{versions}(c)}\{\mathrm{val}(m_C)\} \tag{5}$$

Where versions(c) is the set of all successive versions of class C stored in the Git repository due to subsequent changes made by committers and val(m_C) denotes the measured value of metric m on class C. Likewise:

$$m_{\mathrm{init},i} = \sum^{C \in P_i} \{\mathrm{val}(m_{C0})\} \tag{6}$$

Where C_0 denotes the initial version of class C in Git repository. The objective of the correlation analysis is to examine the random variable pairs: $(\Delta b_i, N_i)$, $(\Delta b_i, I_i)$ and $(\Delta b_i, B_i)$ for existence of significant linear correlation. The correlation was analyzed using the Pearson test and the results are illustrated in Figure 6. The main reason that we considered the difference between the max and the initial metric values has been to obtain the added number of smells to the base value during the successive commits. In other words the increase in the number of smells matters most in our study.

For the Eclipse products the separate correlation analysis for BAD and FAD metrics was not possible as the classification of issue reports into "Bug" and "New Feature" is not provided by the Bugzilla ITS. The highest correlation was

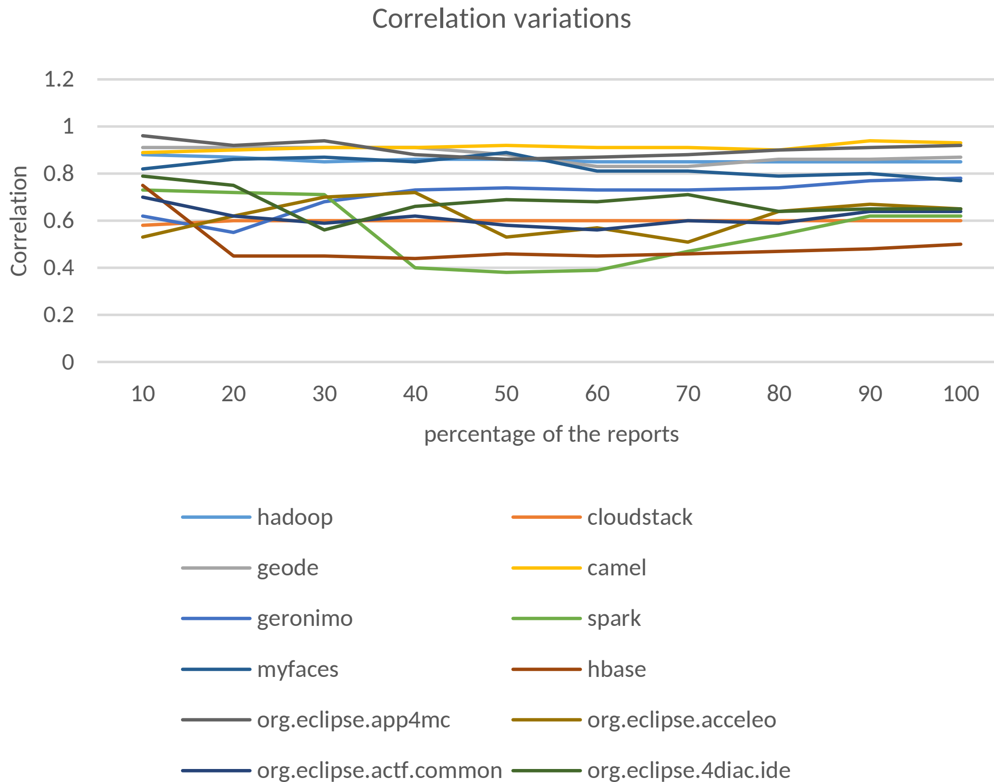


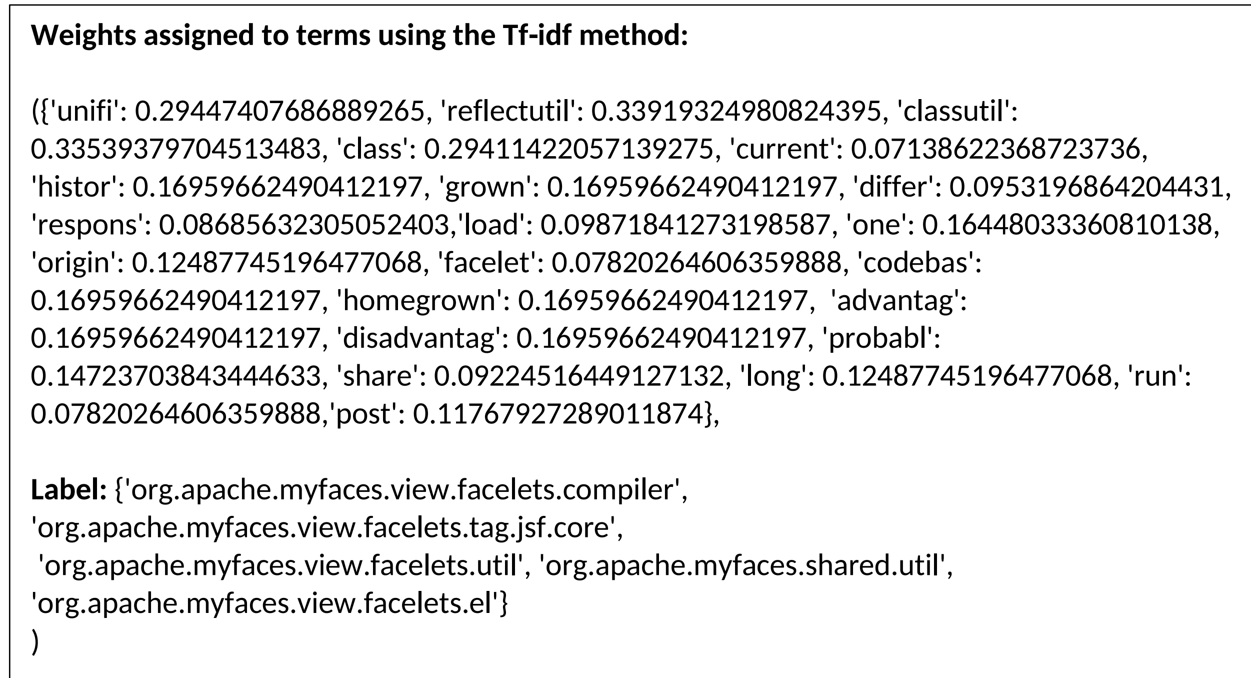
Figure 7. The correction between the number of reports of type "New Feature" and the number of bad-smells of type: "Design+ErrorProne" computed for different dataset sizes

observed between the "Design+ErrorProne" code metric and the FAD issue metric. With respect to the "CyclomaticComplexity" and "NpathComplexity" metrics, as the indicators of the extent to which polymorphism is missing in the design, good correlation is observed between the "CyclomaticComplexity" metric and the FAD metric. The variation of the correlation values (between the FAD and "Design+ErrorProne" metrics) over the maintenance period has also been studied by stepwise correlation analysis using different repository sizes. As shown in Figure 7 the fractional correlation analysis has been performed on data set sizes ranging from 10% to 100% of the issue reports in the repository. The results of the above analysis will be discussed in Section 6.

5.3. Stream Clustering Analysis

To evaluate the proposed RepChainer algorithm, issue reports from 10 Apache products were analyzed. Each report is associated with a set of commits and the set of packages changed by the respective commits is used as the report label. A sample report and its label is shown in Figure 8.

For each product three streams of reports: small, medium and large size were prepared to evaluate the ReportChainer algorithm. The small size stream was used for finding the best value for split threshold (learning set) and the two other streams used for evaluation. A Python program iterates through threshold values in the range [0.1 0.8] and for each threshold the chains are created and the "chain lengths" and their corresponding "target hit numbers" are extracted to compute the correlation value. The "target hit number" of each chain is determined based on the labels of reports in the chain; for instance for the following chain: (R1, {P1, P2, P3}) $\rightarrow$ (R2, {P1, P3}) $\rightarrow$ (R3, {P1}) consisting of three reports R1, R2 and R3 and labels {P1, P2, P3}, {P1, P3} and {P1} the chain target is package P1 and its hit number is three. The objective of this experiment is to show that the longer the chain is the higher the target hit number will be. Note that opposed to many previous studies (for



Weights assigned to terms using the Tf-idf method:

({'unifi': 0.29447407686889265, 'reflectutil': 0.33919324980824395, 'classutil': 0.33539379704513483, 'class': 0.29411422057139275, 'current': 0.07138622368723736, 'histor': 0.16959662490412197, 'grown': 0.16959662490412197, 'differ': 0.0953196864204431, 'respons': 0.08685632305052403,'load': 0.09871841273198587, 'one': 0.16448033360810138, 'origin': 0.12487745196477068, 'facelet': 0.07820264606359888, 'codebas': 0.16959662490412197, 'homegrown': 0.16959662490412197, 'advantag': 0.16959662490412197, 'disadvantag': 0.16959662490412197, 'probabl': 0.14723703843444633, 'share': 0.09224516449127132, 'long': 0.12487745196477068, 'run': 0.07820264606359888,'post': 0.11767927289011874},

Label: {'org.apache.myfaces.view.facelets.compiler', 'org.apache.myfaces.view.facelets.tag.jsf.core', 'org.apache.myfaces.view.facelets.util', 'org.apache.myfaces.shared.util', 'org.apache.myfaces.view.facelets.el'}
)

Figure 8. A sample vectorized issue report and its label

Table 4. Pearson test results for random variables: (chainLen, hitNumber).

Product	Learning set size (number of reports)	Learned Threshold	Pearson coefficient value	Best Threshold	Best Pearson coefficient value
Cloudstack	26	0.35	1	0.35	1
Geode	507	0.1	0.98	0.1	0.98
Spark	1243	0.7	0.84	0.1	0.97
Camel	796	0.1	0.94	0.1	0.94
Geronimo	673	0.7	0.85	0.1	0.94
Hadoop	550	0.25	0.80	0.1	0.97
Myfaces	192	0.1	0.95	0.1	0.95
Hive	3070	0.1	0.99	0.1	0.99
Hbase	783	0.15	0.95	0.1	0.98
Cassandra	70	0.35	0.93	0.1	0.99

example [23]) that aimed at finding the target package which is a more difficult problem to solve even with supervised machine learning methods, the intent of the ReportChainer is merely forming the chains correctly. The correlation value was computed using the Pearson method implemented in NumPy [33]. The results are listed in Table 4. The samples of (chainLen, hitNumber) pairs are plotted in Figure 9.

According to the observed results, for almost all of the products, the threshold value 0.1 resulted in chains with the highest correlation value. It was observed that for fairly long sequences of reports this threshold value worked fine across Apache products. There were products for which the learned threshold value differed from the best value (for instance CloudStack and Cassandra) due to the relatively few number of samples in the learning sequence: 26 and 70 respectively. Generally it can be argued that the shorter the sequence of reports, the higher threshold value will result in the best set of chains (in terms of

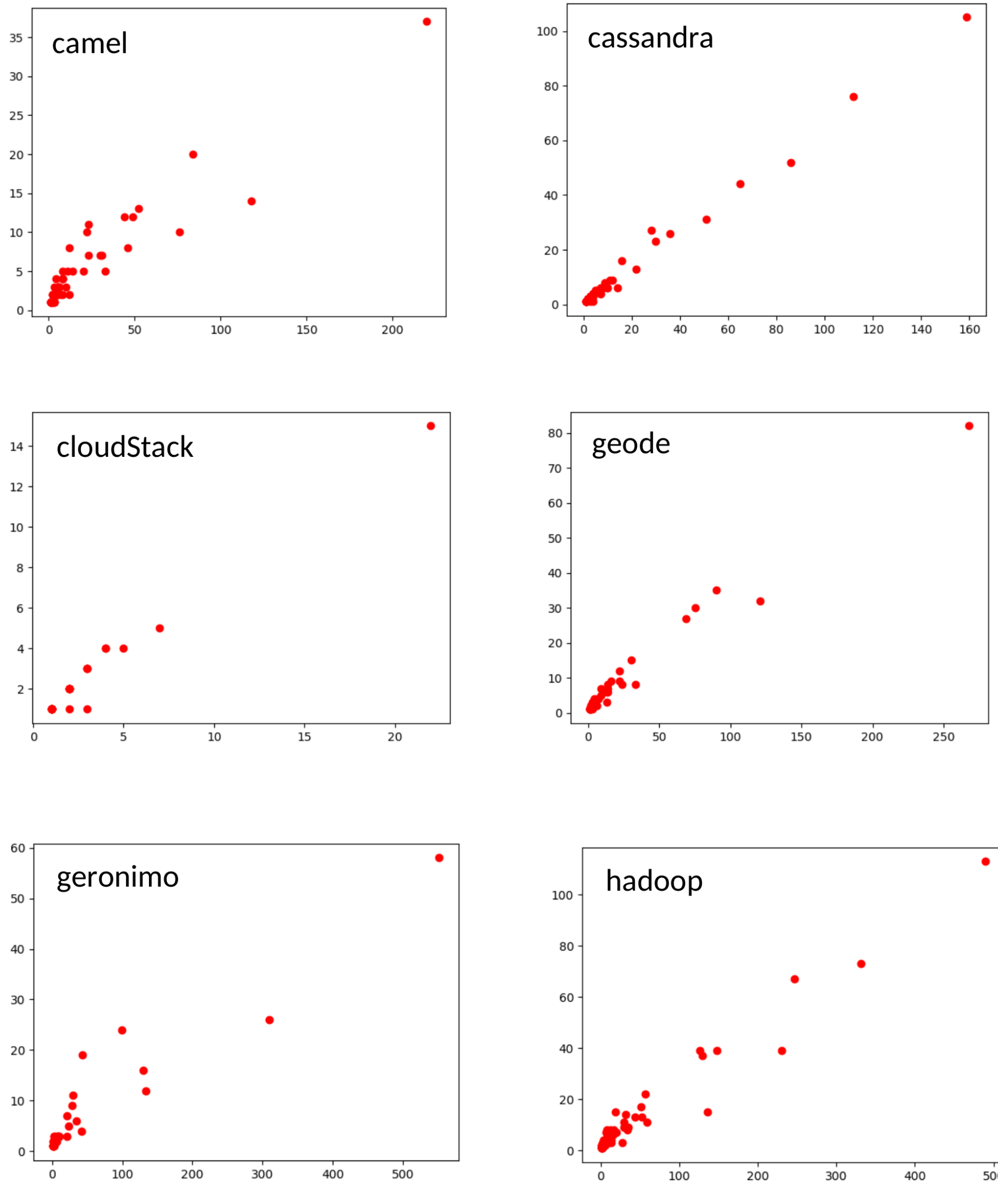


Figure 9. Plot of pairs: (chainLen, hitNumber) for six Apache products: Camel, Cassandra, CloudStack, Geode, Geronimo and Hadoop. The x and y axes are chain length and target hit numbers respectively. The chains were created with split threshold = 0.1

the Pearson correlation metric). On the other hand, a lower threshold value creates fewer equal (short) length of chains due to their less join rejection rate. Many equal short size chains prevent the recommender module from working properly. For instance in CloudStack the best threshold learned 0.35, however by examining the resulted chains, many short equal length chains were observed. By choosing the threshold value less (= 0.1) this problem was overcome.

6. Discussion

The results confirmed strong linear direct correlation (0.73 on average) between the FAD metric and the "Design+ErrorProne" deterioration metric. Moreover the FAD metric is correlated with the "CyclomaticComplexity" deterioration metric well. Hence the FAD metric could be applied as a good indicator of the code deterioration level in terms of the number of emerged bad-smells and the extent to which the polymorphism is re-

placed wrongly by the complex conditional logics in the design. The statistical analysis using the Student"s T distribution showed that the mean value of the correlations (between FAD and "Design+ErrorProne") falls within [0.32, 1.14] with confidence 95%. Moreover as shown in Figure 7, the correlation value is almost independent of the number of received issue-reports on the product and at any time, as the product changes, the FAD metric can be used to measure the relative deterioration level.

To justify the relatively lower correlation values between FAD and "NpathComplexity" metrics, it should be noted that though both "NpathComplexity" and "CyclomaticComplexity" metrics measure the amount of complex conditional logics in the code, the former is very sensitive to the composition of the conditional statements in the code while the latter only counts the number of decision points.

Furthermore to explain why the FAD metric is superior to the BAD metric in terms of the correlation strength to the deterioration code-level metrics, we argue that:

(1) Usually adding a feature involves more (re-)design activities than fixing a bug and it is more likely to incorporate bad- structures into the code compared to fixing bugs.

(2) A good design suppresses the subsequent "New Feature" requests whereas a bad design produces forthcoming related "New Feature" requests: It is easy to see (as we have seen in our industrial experiences) that if a "New Feature" request is designed properly using well-defined design patterns (mostly based on polymorphism), the resulted code will involve the abstract concepts rather than concrete classes and hence the subsequent requests are likely to be variants and special cases of the initially requested feature and could be easily addressed by adding new classes rather than modifying the existing code (Open-Close principle [34]). Hence in this case less forthcoming requests of type "New Feature" is expected to be received. In contrast,

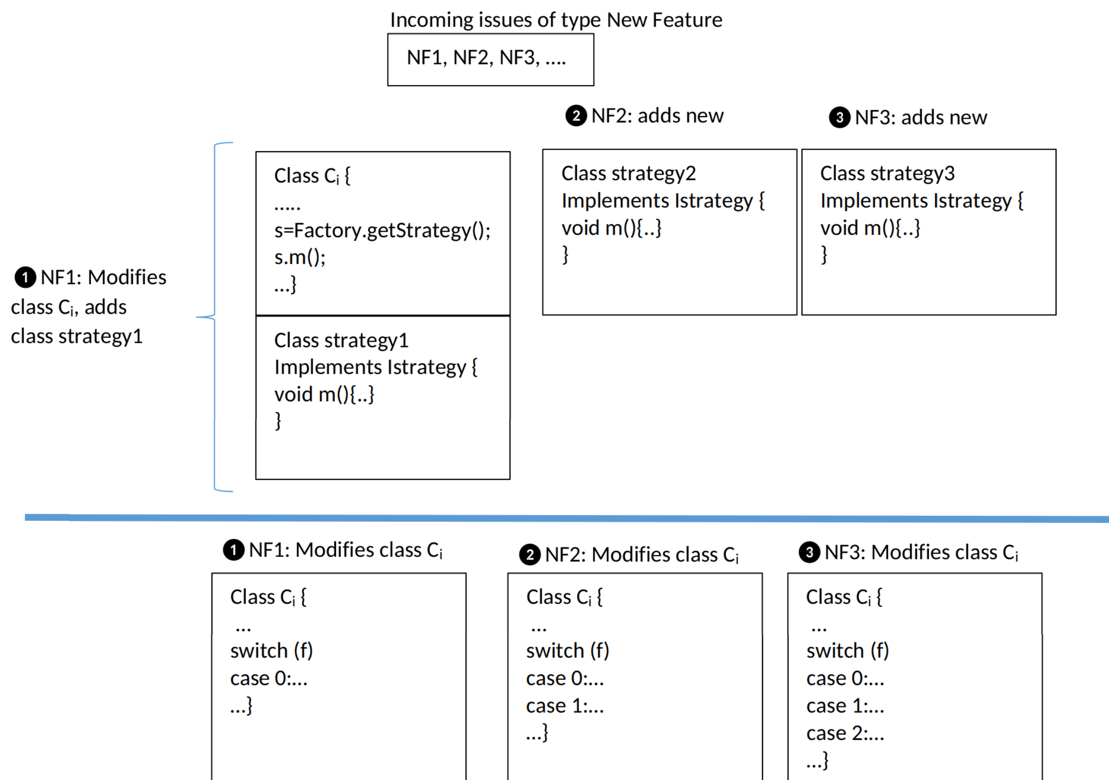


Figure 10. Classes with more anti-pattern constructs absorb more subsequent New Features. Three related New Feature requests: NF1,NF2 and NF3. Using strategy design pattern (top), using conditional constructs (bottom)

the more polymorphism and design pattern constructs are replaced with conditional logics (and anti-patterns) in the code, the higher number of subsequent New Feature requests will be generated and absorbed by the module. This concept is illustrated in Figure 10.

To our best knowledge, there has been no study on the impact of different change types on the deterioration level of the code so far. According to the literature [34, 35]. This is widely accepted in the software engineering community that successive changes in the maintenance phase of software gradually causes the software to rot and the symptoms of deterioration to emerge: rigidity, fragility, needless complexity, opacity, immobility and these are due to bad smells in the code. Hence the more changes to the code the more deteriorated the code becomes. The results obtained in our study also corroborates the literature in the sense that "New Features" are sub types of "Change" and hence we could expect to see a good correlation between the number of "New Features" and the deterioration level. Now we can answer to our mentioned research questions:

RQ1: Can the QC team evaluate the quality level of the code, which is under successive changes during the maintenance phase, only by observing issue-related metrics such as the number of reported issues on a software package? Is the type of reported issues important in this evaluation?

Answer: According to the analysis presented in Section 5, yes. Issue reports are clustered using the stream clustering algorithm and the target package of the longest chain is recommended as the package with a high change rate. Due to the strong correlation between the value of FAD metric and the number of bad-smells, this package is presumed to be a priority for refactoring.

RQ2: Is there an effective stream clustering method to categorize incoming sequence of issue reports such that a bloated category be truly interpreted as the concentration of frequent changes on a specific software package and hence be reported as possible point of deterioration?

Answer: According to the correlation analysis results presented in Table 4, the proposed stream clustering algorithm could successfully create sequences of related issue reports such that the issue reports in each sequence are mostly focused on a specific package called target.

7. Threats to validity

As the most important threat to validity of the proposed QC approach to mention is the accuracy of the presented stream clustering algorithm, the parameters of this algorithm has to be fine-tuned according to the issue-report peculiarities and the product characteristics. Another threat to validity is the accuracy of the PMD bad smell detection tool. In [36] a comparative study of bad smell detection tools has been presented. In this study a detailed comparative study of four tools including PMD showed that this tool was able to reach up to 100% precision and 50% recall on particular test-cases and could outperform other tools in precision while rival in the recall value. Another limitation is that the study is delimited in the Object Oriented programming field though the strength of the proposed approach is independent of the used programming language.

8. Conclusions and future works

The continuous modifications of software modules lead to emergence of bad-smells and it is desirable to equip ITSs with assisting tools to notify the QC team about parts of the code that need immediate refactoring attentions. In this paper by creating a dataset of issue reports and their corresponding change information extracted from Apache and Eclipse open-source software repositories, a thorough study was conducted. The results confirmed a significant linear direct correlation between the FAD issue-level metric measured on a software package and the "Design+ErrorProne" and "CyclomaticComplexity" code-level metrics. Hence the packages with higher measured values for FAD can be considered as good candidates of parts highly exposed to deterioration, they need immediate attention of the QC team. A challenging part of the proposed model was to design a stream clustering to

be able to split the sequence of reports into chains in a way that the length of the chains be a valid indicator of its target hit number. According to the observed results, for most of the products, the threshold value 0.1 resulted in chains with the highest Pearson correlation between chain lengths and their target hit numbers. As the future work we aim at studying other bad-smell types as well as other latent bug level metrics to predict the deterioration trend of the software during the maintenance phase. Another future work could be studying the same dataset with other smell detection tools apart from PMD.

References

[1] *Bugzilla*. [Online]. http://www.bugzilla.org (Accessed on 2018-06-06).

[2] *Atlassian*. [Online]. https://www.atlassian.com/software/jira (Accessed on 2018-06-06).

[3] L. Yu, S. Ramaswamy, and A. Nair, "Using bug reports as a software quality measure," 2013.

[4] M. Badri, N. Drouin, and F. Touré, "On understanding software quality evolution from a defect perspective: A case study on an open source software system," in *International Conference on Computer Systems and Industrial Informatics*. IEEE, 2012, pp. 1–6.

[5] C. Chen, S. Lin, M. Shoga, Q. Wang, and B. Boehm, "How do defects hurt qualities? An empirical study on characterizing a software maintainability ontology in open source software," in *International Conference on Software Quality, Reliability and Security (QRS)*, 2018, pp. 226–237.

[6] *Standard for Software Maintenance*, IEEE Std. 1219-1998, 1998.

[7] L. Yu, S. Schach, and K. Chen, "Measuring the maintainability of open-source software," in *International Symposium on Empirical Software Engineering*, 2005, p. 7.

[8] R. Malhotra and A. Chug, "Software maintainability: Systematic literature review and current trends," *International Journal of Software Engineering and Knowledge Engineering*, Vol. 26, No. 8, 2016, pp. 1221–1253.

[9] H. Sharma and A. Chug, "Dynamic metrics are superior than static metrics in maintainability prediction: An empirical case study," in *4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)*. IEEE, 2015, pp. 1–6.

[10] S. Shafi, S.M. Hassan, A. Arshaq, M.J. Khan, and S. Shamail, "Software quality prediction techniques: A comparative analysis," in *4th International Conference on Emerging Technologies*. IEEE, 2008, pp. 242–246.

[11] P. Piotrowski and L. Madeyski, "Software defect prediction using bad code smells: A systematic literature review," *Data-Centric Business and Applications*, 2020, pp. 77–99.

[12] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[13] *Apache*. [Online]. https://projects.apache.org/projects.html (Accessed on 2018-06-06).

[14] *Eclipse*. [Online]. https://www.eclipse.org/ (Accessed on 2018-06-06).

[15] V. Lenarduzzi, A.C. Stan, D. Taibi, D. Tosi, and G. Venters, "A dynamical quality model to continuously monitor software maintenance," in *The European Conference on Information Systems Management*. Academic Conferences International Limited, 2017, pp. 168–178.

[16] S. Kim, T. Zimmermann, K. Pan, and E.J. Whitehead, Jr., "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*, 2006, pp. 81–90.

[17] H. Wang, M. Kessentini, W. Grosky, and H. Meddeb, "On the use of time series and search based software engineering for refactoring recommendation," in *Proceedings of the 7th International Conference on Management of computational and collective intElligence in Digital EcoSystems*, 2015, pp. 35–42.

[18] F. Palomba, M. Zanoni, F.A. Fontana, A. De Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 244–255.

[19] F. Khomh, M. Di Penta, Y.G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, Vol. 17, No. 3, 2012, pp. 243–275.

[20] A.S. Cairo, G. de F. Carneiro, and M.P. Monteiro, "The impact of code smells on software bugs: A systematic literature review," *Information*, Vol. 9, No. 11, 2018, p. 273.

[21] D.M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *International conference on software architecture (ICSA)*. IEEE, 2018, pp. 176–17 609.

[22] *PMD static code analyzer*. [Online]. https://pmd.github.io/latest/pmd_rules_java.html (Accessed on 2018-06-06).

[23] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? A two-phase recommendation model," *IEEE transactions on software Engineering*, Vol. 39, No. 11, 2013, pp. 1597–1610.

[24] N. Limsettho, H. Hata, A. Monden, and K. Matsumoto, "Unsupervised bug report categorization using clustering and labeling algorithm," *International Journal of Software Engineering and Knowledge Engineering*, Vol. 26, No. 7, 2016, pp. 1027–1053.

[25] F.A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, Vol. 128, 2017, pp. 43–58.

[26] S.A. Vidal, C. Marcos, and J.A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, Vol. 23, No. 3, 2016, pp. 501–532.

[27] T.J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, No. 4, 1976, pp. 308–320.

[28] C.D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[29] *GitHub*. [Online]. https://github.com/ (Accessed on 2018-06-06).

[30] *isomorphic-git*. [Online]. https://isomorphic-git.org/ (Accessed on 2018-06-06).

[31] *Apache's JIRA issue tracker!* [Online]. https://issues.apache.org/jira/secure/Dashboard.jspa (Accessed on 2018-06-06).

[32] *bugs.eclipse.org*. [Online]. https://bugs.eclipse.org/bugs/ (Accessed on 2018-06-06).

[33] *NumPy*. [Online]. https://numpy.org/

[34] R.C. Martin, *Clean code: A handbook of agile software craftsmanship*. Pearson Education, 2009.

[35] R.C. Martin, M. Martin, and M. Martin, *Agile principles, patterns, and practices in C#*. Prentice Hall, 2007.

[36] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16. ACM, 2016.