A Dynamical Quality Model to Continuously Monitor Software Maintenance

Valentina Lenarduzzi[1], Alexandru Cristian Stan[1], Davide Taibi[1], Davide Tosi[2] and Gustavs Venters[1]
[1]Free University of Bozen-Bolzano, Bozen-Bolzano Italy
[2]Università degli Studi dell'Insubria, Varese, Italy
valentina.lenarduzzi@unibz.it; davide.taibi@unibz.it
alexadrucristian.stan@stud-inf.unibz.it; gustavs.venters@stud-inf.unibz.it
davide.tosi@uninsubria.it

Abstract. *Context:* several companies, particularly Small and Medium Sized Enterprises (SMEs), often face software maintenance issues due to the lack of Software Quality Assurance (SQA). SQA is a complex task that requires a lot of effort and expertise, often not available in SMEs. Several SQA models, including maintenance prediction models, have been defined in research papers. However, these models are commonly defined as "one-size-fits-all" and are mainly targeted at the big industry, which can afford software quality experts who undertake the data interpretation tasks.

Objective: in this work, we propose an approach to continuously monitor the software operated by end users, automatically collecting issues and recommending possible fixes to developers. The continuous exception monitoring system will also serve as knowledge base to suggest a set of quality practices to avoid (re)introducing bugs into the code.

Method: first, we identify a set of SQA practices applicable to SMEs, based on the main constraints of these. Then, we identify a set of prediction techniques, including regressions and machine learning, keeping track of bugs and exceptions raised by the released software. Finally, we provide each company with a tailored SQA model, automatically obtained from companies' bug/issue history. Developers are then provided with the quality models through a set of plug-ins for integrated development environments. These suggest a set of SQA actions that should be undertaken, in order to maintain a certain quality level and allowing to remove the most severe issues with the lowest possible effort.

Conclusion: The collected measures will be made available as public dataset, so that researchers can also benefit of the project's results. This work is developed in collaboration with local SMEs and existing Open Source projects and communities.

Keywords: Software Quality, Software Maintenance, Dynamic Software Measurement

1.! Introduction

Software maintenance is an effort-intensive task. One of the main reasons for the high maintenance costs is due to execution errors, which require corrective maintenance to get them fixed. Code exceptions raised in production software are one of the most common reasons of corrective maintenance activities. This happens because exceptions are often not managed nor tracked and often leads to bugs on software running on customers' clients.

From an end-user perspective, exceptions can demotivate them in using an application, hence increasing the risk of losing existing customers. This is common in mobile applications where users can easily find alternative solutions in the blink of an eye, in case the app they are using is not responsive enough or raises exceptions. Nonetheless, only few users report negative feedbacks in case of problems using the applications. However, these feedbacks are not easily traceable and the cause of the feedbacks are not clearly identifiable from user feedbacks reported on the various app stores.

In our previous work (Janes et al, 2017), we confirmed that code smells and anti-patterns, particular structural issues in the source code, as one of the most common drivers for software exceptions.

Code smells, often referred as to bad smells in code are violation to good design principles that can be symptoms of potential problems in the source code or "surface indications that usually correspond to deeper problems in the system" (Fowler and Beck 1999). Fowler and Beck also report negative effects of code smells on software maintenance and recommend refactoring actions to remove them (Fowler and Beck, 1999).

Several works also confirmed that reducing code smells can reduce the risk of errors in the systems also increasing software maintenance effort (Fontana et al 2012)
In this paper, we present an ongoing work on a novel approach to continuously monitor software quality by monitoring code exceptions, identifying root causes, and defining a tailored and self-adapting quality model to predict potential exceptions. To reduce the burden for developers and to understand which quality aspect and which code smells are actually impacting software maintenance, in this paper we propose a continuous SQA monitoring approach that combines continuous learning techniques based on SME's common quality issues, together with a recommendation system, in order to suggest developers which code smell should be removed in order to reduce the probability of exceptions in their code.

The main difference between our approach and existing IDE plug-ins for software quality, lays in the recommendation system at the developer and development-team level, which is based also on the exceptions raised by the operating software.

The remainder of this paper is structured as follows: Section 2 reports on background and related works. Section 3 describes the overall approach of this work. Section 4 describes the approach needed to identify the root cause of the exceptions and to generate the quality model. Section 5 reports on the current status, while Section 6 draws conclusion and highlights future works.

2.! Background and Related Works

In this Section, we describe related works. We first introduce existing quality and defect prediction models, then we report on continuous monitoring, detailing similar approaches for exception monitoring, and finally we describe the algorithms for the defect root cause analysis we adopt in our prediction model.

2.1! Quality Models and Defect Prediction Models

Continuous quality monitoring should be supported by prediction models that predict the quality under control (Janes et al, 2017). As example, a continuous exception monitoring system should be used in conjunction with an exception prediction model, with the intention not only to capture existing exceptions, but also reduce the probability of new exceptions happening in newly developed code.

In the analysis of four decades of research on software maintenance (Lenarduzzi et al, 2017) they highlighted that starting from the 1990s, interesting defect prediction models got introduced. However, at the best of our knowledge, there are no exceptions-specific prediction models in the literature.

Defect prediction models are the closest ones to our work. Several models have been proposed in this domain, based on different information, such as previous defects (Kim et al, 2007 and Hassan et al, 2005), process metrics (Nagappan et al, 2005 and Bernstein et al, 2007) (number of changes, recent activity), and code metrics (Basili et al, 1996 and Nagappan et al, 2006) (lines of code, complexity).

Nagappan and Ball analysed the influence of changes in the code for what concerned the defect density of Windows Server 2003. Some prediction models are based on a wide set of metrics. Moser et al. proposed a prediction model based on several metrics such as code churn, past bugs, refactorings, number of authors (Moser et al, 2008), while Neuhaus et al. used a variety of features of Mozilla (past bugs, package imports, call structure) to detect vulnerabilities (Neuhaus et al, 2007). (Morasca et al 2012), beside confirming that open source software has the same properties and defect level of closed source one, identified a bug prediction model based on the test coverage of open source software, analysing test data, defect data and static code measures.

Other approaches support the defect prediction by analysing changes and defect data (Bernstein et al, 2007, Kim et al, 2007, Hassan et al, 2005, Taber and Port, 2014 and Tian et al, 2008, Janes et al 2017).

Another set of prediction models is based on the usage of the Chidamber and Kemerer (CK) metrics suite (Chidamber et al, 1994). Different works are based on these measures (Basili et al, 1996, Ohlsson et al, 1996, Subramanyam et al, 2003, Gyimothy et al, 2005 and Nagappan 2005). While other works such as (Zimmermann et al, 2007 and Emam et al, 2001) use the CK metric suite in conjunction with other metrics.

As highlighted by (Lenarduzzi et al, 2017) prediction models are commonly defined from scratch and not based on previous ones, therefore they don't benefit of the experience learnt from previous works.
Our work prediction model is based on a similar approach of these adopted in (Lavazza et al. 2010-A, Lavazza et al. 2012, Del Bianco et al. 2010-A, Del Bianco 2010-B, Del Bianco et al. 2011, Tosi et al. 2012), in which they propose to recommend a set of quality practices or a set of measures to be considered during the development, based on correlations between subjective data (perceived quality, issues, perceived trustworthiness) and objective metrics (Lines of Code, Complexity and others).

Our work differs from the previous ones by:

- The definition of a continuous exception monitoring approach
- The definition of a dynamic exception and issue prediction model which:
  - Keep updating based on the real exceptions caught by the software in use
  - Automatically train themselves based on the metrics and code smells that caused exceptions in the software in use.

2.2 Continuous Monitoring

Continuous monitoring is a technique defined in the context of *DevOps* practices or is part of an *application performance management* (APM). The goals of continuous monitoring are to continuously assess if a production software is performing as intended, and to obtain information about the stability of the environment in which the software is executed. Continuous monitoring systems continuously collect metrics of the monitored software, measuring that the software attributes are within the required quality ranges, hence under control. Continuous monitoring tools cover a set of quality practices, including the continuous monitoring of performance, testing, and exceptions. Other quality practices, with tools such as SonarQube (www.sonarqube.org) also include the continuous code inspections.

Taking into account exception monitoring tools, only few exist on the market, being introduced very recently. Moreover, at the best of our knowledge, no research has been carried out in the domain of exception monitoring and prediction. Continuous exception monitoring tools aim at collecting exceptions or errors arising from the execution of production software. As example, when the end user operate on the production software and the software is, unexpectedly, not responding as expected the developers should be able to trace this issue. In some case, these issues are related to exceptions managed by developers in other cases they are considered as general exceptions and not managed. There exist few of these tools that catch, store and classify the issues as soon as they occur, providing an informed insight on their causes and driving developers' attention on the most critical urging to be solved. Some of the currently available tools integrate well with other platforms (e.g. Atlassian's JIRA – issue tracking system – https://www.atlassian.com/software/jira), which allow a prompt management of the issues by the developers.

Table 1 reports the currently most common commercial and open source continuous exception monitor tools while Figure 1 shows an example of *Sentry*®.

Most of these tools are very recent, although they have already a very large user base or practitioners (from 2.000 to 60.000 developers using them).

Table 1: The most common Exception Management Tools

Tool	Website	Supported Programming Languages
OverOps (formerly Takipi)	https://www.overops.com/	Java, Scala, Closure, .NET
Airbrake	https://airbrake.io/	All major programming languages and frameworks
Sentry	https://sentry.io/	All major programming languages and frameworks
Rollbar	https://rollbar.com/	All major programming languages and frameworks
Raygun	https://raygun.com/	All major programming languages and frameworks
Honeybadger	https://www.honeybadger.io/	All major programming languages and frameworks
Stackhunter	https://stackhunter.com/	Java
Bugsnag	https://www.bugsnag.com/	All major programming languages and frameworks
Exceptionless	https://exceptionless.com/	All major programming languages and frameworks

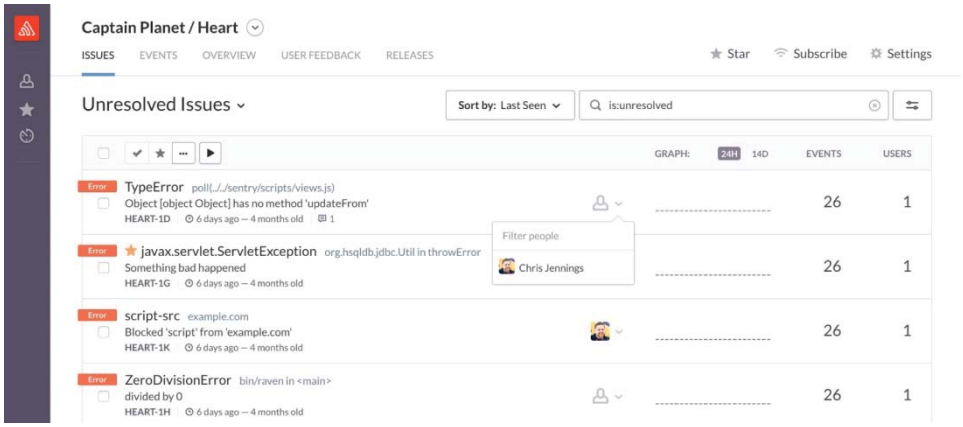

Figure 1: a screenshot of Sentry® - Exception Monitoring tool

2.3! Defect Root Cause Analysis

When an issue is fixed and closed by a developer, we try to locate the root cause of the issue. To do so, we exploit the SZZ algorithm (Kim et el, 2006), which tries to locate in the code's commit history, which code revision did introduce the problem (i.e. when and which lines of code changed, giving birth to the bug).

The SZZ algorithm (Figure 2) works as follows: first, it identifies the bug fixes; second, it locates the fix-inducing changes. To perform the first part, the algorithm searches into the code history for the specific FIXED/CLOSED issue, looking for matches between the bug report number or relevant keywords in the change log text, identifying the transactions (code commits) involving the bug, and giving each of them a confidence value, by applying first a syntactic analysis on the log message, and then a semantic analysis between the log message and the bug report. For the second part, the algorithm takes the locations of the identified bug-fix commit, and searches back for the commit that introduced the bug (i.e. when and which lines of code changed, giving birth to the issue). A *diff tool* is used on the commits to determine what did change in the file revision. SZZ assumes that deleted or modified source code in each hunk is the location of a bug. Afterwards, the algorithm tracks down the origins of deleted or modified source code using the built-in annotate feature of VCS systems (the annotated information only contains triples of: current reversion line, most recent modification revision, developer who made modification).

Finally, for all the detected modified files before the bug-fix revision, the algorithm locates the very first bug-introducing change, discriminating it from false positives and false negatives suspects in the list of matching results.

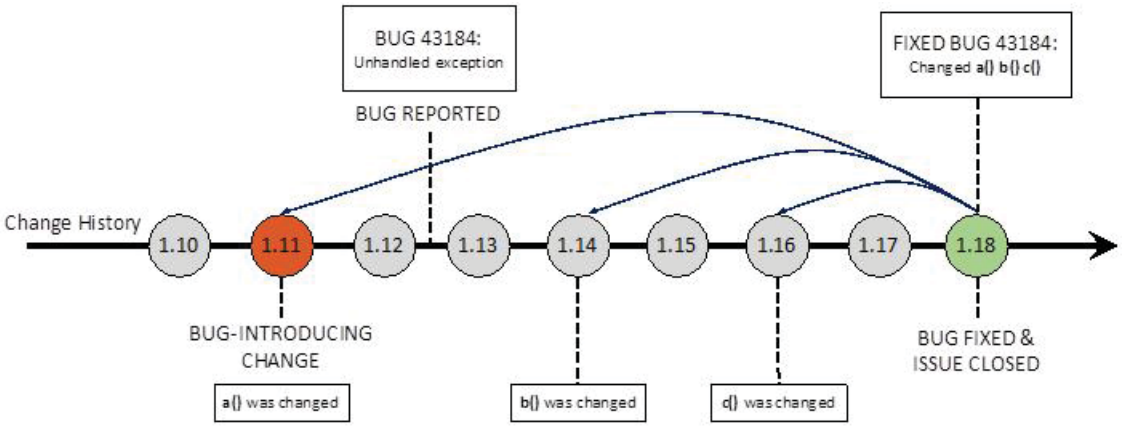

Figure 2: An example of the SZZ algorithm

3.! The Approach

From a bird's-eye view, our approach foresees a set of steps from the operational and development point of view: The operation and Development aspects of this can be seen in figure 3.

1.! Operation
    a.! While the end-user uses the application, an exception monitoring library (developed as "exception listener" on the Java Virtual Machine) captures the unmanaged exceptions and sends them to a central Continuous Exception Monitoring server. The exception listener sends all the information available at runtime about the state of the monitored system to the exception monitoring server (e.g. the stack trace or a screenshot of the current view)
    b.! The monitoring system adds an issue to the issue tracker (Jira, GitHub or GitLab)
2.! Development
    a.! When the developers solve the issue, they commit the changes into the versioning system reporting the issue number in the commit message.
    b.! A continuous building system executes all tests – generally once a day - and, in case of success, launches the SZZ algorithm (Kim et el, 2006) to identify the root cause of the exception.
    c.! The continuous Inspection tool (SonarQube) analyses if some code smells induced the exception, and report it to the recommendation as continuous training data.
    d.! The recommendation system builds a quality model in SonarQube based on the code smells and metrics that commonly generated the same kind of exceptions or issues.

During the development, the SonarQube IDE integration plugin (SonarLint) recommends to developers editing a particular piece of code, whose quality action should be taken to reduce the probability of issues and exceptions that typically arise in the system, based on the quality model build on the historical data collected at the monitoring and the maintenance phase.

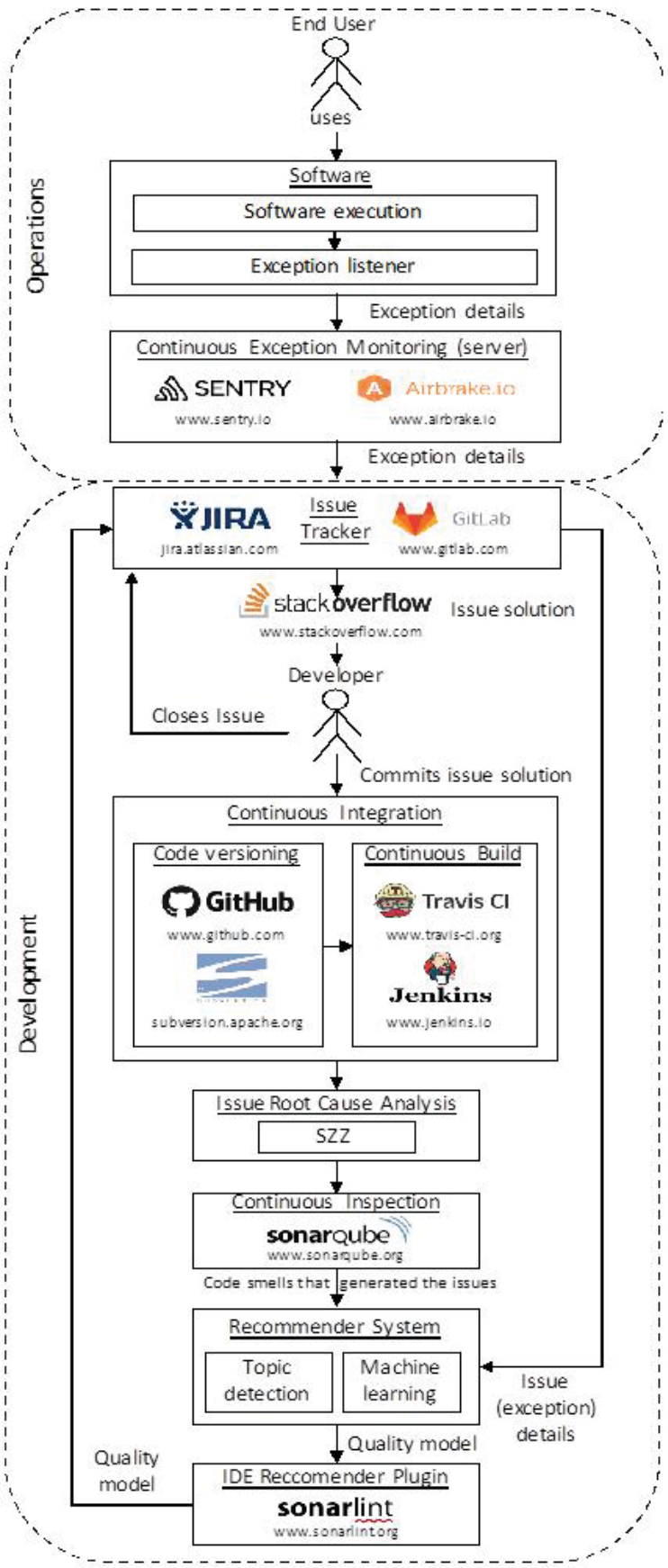

Figure 3: The Proposed Process for Continuous Quality Monitoring

4.! The Exception Monitoring Phase

The exception monitoring is designed to monitor unmanaged exceptions in Java-based applications. Applications can be web-based, mobile Android apps or java desktop applications.
As often happens in production software, developers do not manage all the exceptions, adding a generic try-catch statement at the end of all the exception management. A typical example of unmanaged exception is reported in Figure 4 (lines 8-10) where, at the end of all the catch clauses, developers caught a generic "Exception" and printed the stack trace.

```
1 Try { … }
2 catch(ArithmeticException e){
3      a.manage(a)
4   }
5 catch(ArrayIndexOutOfBoundsException e){
6      b.extendArray(var1, var2)
7   }
8 catch(Exception e){
9      e.printStackTrace();
10  }
```

Figure 4: Example of unmanaged exception

By means of a Java library we developed, we are able to capture all "printStackTrace" without requiring the developers to instrument their code. The only step required by the developer is to reference the library in their class-path and to add it into the application configuration (e.g. in the web.xml configuration file of a Tomcat web application).

We implement this step by extending Sentry exception monitoring and Jira as issue tracking systems.
Sentry is configured to automatically open an issue on Jira, depending on the configuration required, for each exception submitted by the exception listener that runs in the operating application. In case the exception already exists in Jira, Sentry clusters it to only one reference, including the details of the collected exception as a comment to the issue tracker, such that the developers can easily track issues related to the most frequent exceptions.

5.! The Quality Model Generation Phase

In this section, we report on the approach adopted to dynamically generate the quality models as depicted in Figure 5. Quality models are a set of metrics, with related thresholds, that will be used by the SonarLint IDE plug-in to recommend the set of quality actions to the developers (e.g. remove a specific code smell), with the aim of decreasing the probability of generating new exceptions in the running software.

5.1! Exception Root Cause Analysis

In order to understand which are the code smells that generate issues, we need first to identify the root cause of each issue. The goal is to identify where and when a change induced the exception. Whenever developers fix an issue, we identify the location of the fix by identifying the exact lines of code that caused the exception.
Here we apply the SZZ algorithm (Kim et el, 2006). However, to support the location of the changes that induced the fixes, developers must include the *bug report number* in the comment whenever they fix an issue, as also suggested by (Kim et el, 2006).

Compared to (Kim et el, 2006), we also include a new step to ensure that a code smell has been removed only in the issue-fix and not before.

We assume that, if a code smell was detected in the root-cause commit and removed only in the bug fixing commit, this code smell can be considered as driver of the exception. However, one could expect that the code smell could have been removed in a commit before the bug-fix and then re-introduced in other commits. Therefore, in order to assure that the code smell had no other causes or influences, we consider a code smell related to an issue fix, only if the code smell has been created in the root cause of the issue/exception and never removed before the bug fixing.

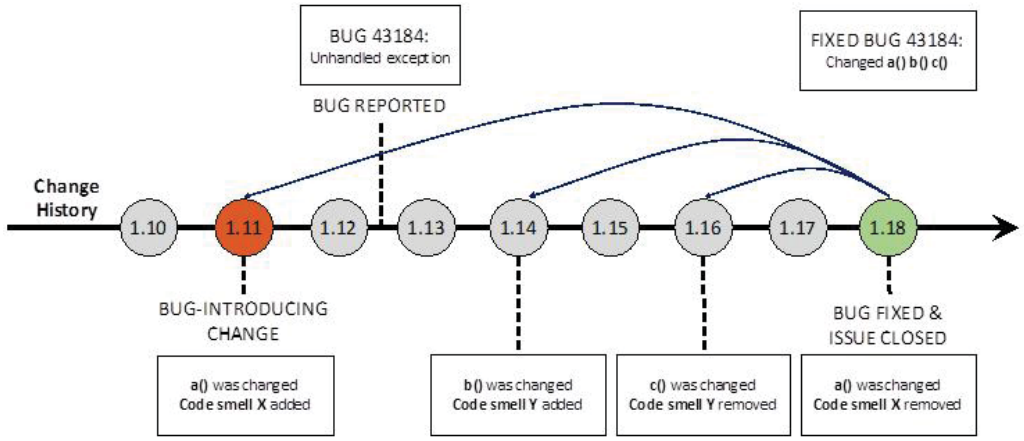

Figure 5: Code smell not related to a bug-fix

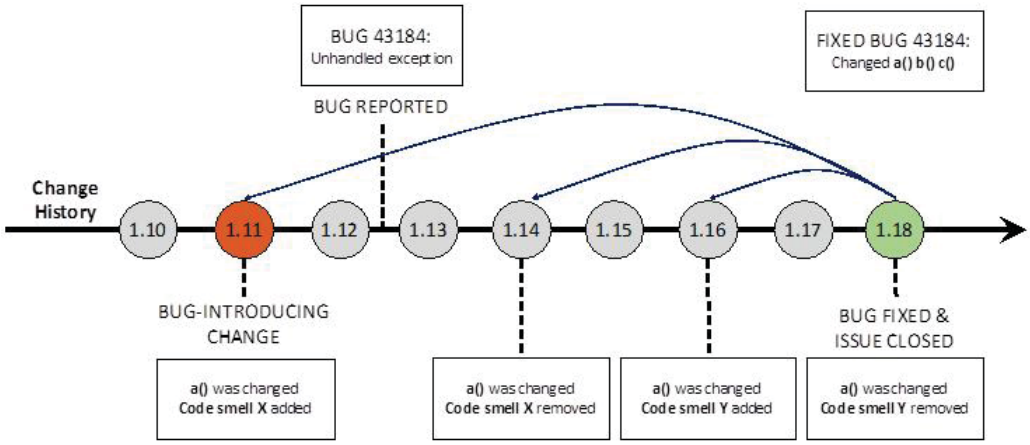

Figure 6: Code smell related to a bug-fix

As example, we report two possible cases. In the first case (Figure 6), we consider a code smell introduced during the root cause of an issue and removed after two weeks for another reason. In this case, we cannot relate the removal of this code smell to the fix of the issue.

In this second case (Figure 6), the code smell was introduced during the root-cause of the issue, removed after two weeks for other reasons, and added again for further reasons two weeks later. Finally, the code smell has been removed during the bug fix. Please, note that “a()”, “b()” and “c()” refer to methods (a, b and c) in a Java class.

5.2! The Recommender System

In this step, we will define the recommender system to analyse the reasons that commonly generated exceptions. Starting from the analysis of metrics that changed in bug fixes, comparing the code smells injected in the root cause and removed in the fix, a machine learning algorithm will continuously update the quality model, so as to provide a set of recommendations for which code smell should be removed, depending on those smells that generated more exceptions or issues in the past.

To achieve this goal, we will first perform a topic detection analysis, classify the topics reported in the commit and issue description, together with the stack trace. This classification will help developers to understand if a specific measure can impact the quality or increase the probability of issues in specific parts of the systems they develop. Then, we will train the recommender system on how to associate existing changes in metrics (e.g. removal of code smells) to issues, based on the classification provided by the topic detection component and on the changed metrics. The most suitable machine learning technique will be identified during the project.

6.! Current status

Currently we are defining the minimum viable product of our application (Lenarduzzi and Taibi 2016), in agreement with a local SME. The goal is to create a start-up that will disseminate the result of the project supporting local companies.

As for the selection of local SMEs that will be part of our pilot project, we are selecting companies with a number of developers that range from 5 to 10. Moreover, we are considering only companies working with an agile development process or companies that are currently migrating to agile, verifying if the agile practices are correctly adopted, including the evaluation of communication practices (Taibi et al. 2017 – C), requirement decomposition into user stories (Taibi et al. 2017 - B) and effort estimation practices (Taibi et al 2015, Taibi et al 2017 – A). However, in case of companies that are currently migrating to agile processes, we will consider them as project partners only if the adoption of agile provided some objective benefits. As for the evaluation of the benefits for the migration to agile processes, we will adopt an approach similar to the one we adopted in (Lavazza et al. 2010 - B, Taibi at al 2017 – B, Taibi at al 2017 - C).

Taking into account the exception monitoring tools, we are evaluating the tools reported in Table 1 along with a set of projects developed by local SMEs. Furthermore, we are providing the initial set of automated recommendations to developers based on the available solutions reported for similar exceptions, based on the analysis of the exception stack trace.

We already have set up the DevOps tool chain including

- -! GitHub as versioning and issue repository
- -! TravisCI as continuous integration and testing, responsible for executing tests and launching SonarQube's execution on every commit
- -! SonarQube for continuos software quality analysis;
- -! SonarLint as IDE plug-in. We are currently working on the Eclipse and IntelliJ versions

We already developed the component to allow SonarQube to correctly analyze the most common code smells, by means of Ptidej (Gueheneuc 2005). Our decision to integrate Ptidej into SonarQube was mainly driven by two reasons: on the one hand simply because, as declared in the previous section, SonarQube is not able to detect these design flaws in the source code, while on the other hand because we found out in the research literature that Ptidej was, in most of the cases, selected as the detection strategy among the encountered studies treating code smells, their detection and their impact on software maintainability.

We are currently validating this approach by analysing 22 Open Source projects from the Apache Software Foundation with a total of more than 5000 source code commits and millions of lines of code. The raw results of the analysis are available. We currently developed a SonarQube plug-in to export the issues in Jira and tested them among the existing issues of the analysed projects. At this stage, the issues were the manually submitted ones, from users and developers. Once the continuous monitoring system will be implemented, we will be able to extract the related issues with the same approach. We are currently implementing a dashboard to provide the developers with the results of the code smells related metrics. At this stage, we have not yet implemented the recommendation component and neither the IDE plug-in.

7.! Conclusion

In this work, we present an ongoing research on the definition of a continuous monitoring system to recommend developers quality actions to remove the likelihood of introducing new exceptions and in general new issues.

We are currently working on the exception monitoring phase, in collaboration with local SMEs and we are developing the extension of SonarQube for continuously building the quality model.
Future works include the application of this approach to four local SMEs developing applications for tourist visiting our region, as well as to validate the project on applications with more than 100K users (with peaks of 1000 concurrent users).

Acknowledgments

This work has been partially supported by the project SQuaSME "recommendation techniques for Software QUAlity improvement in Small Medium Enterprise" funded by the Free University of Bozen-Bolzano.

References

Basili, V. R., Briand, L. C. and Melo, W. L. (1996) "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Software Eng.,* vol. 22, no. 10, pp. 751–761.

Bernstein, A., Ekanayake, J. and Pinzger, M. (2007) "Improving defect prediction using temporal features and non linear models," I*nternational Workshop on Principles of Software Evolution*, pp. 11–18.

Chidamber S. R. and Kemerer, C. F. (1994) "A metrics suite for object oriented design," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 476–493.

Del Bianco, V., Lavazza, L., Morasca, S., Taibi D. and Tosi, D. (2010) "An investigation of the users' perception of OSS quality", International Conference on Open Source Systems, pp. 15-28.

Del Bianco, V., Lavazza, L., Morasca, S. and Taibi D. (2011) "A survey on open source software trustworthiness", IEEE Software, vol.28(5), pp. 67-75.

Del Bianco, V., Lavazza, L., Morasca, S., Taibi D. and Tosi, D. (2010) "The QualiSPo approach to OSS product quality evaluation", Int. Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development, pp. 23-28

Emam, K. E., Melo, W. and Machado, J. C. (2001) "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75.

Fontana, F. A., Braione, P. and Zanoni, M. (2012) "Automatic detection of bad smells in code: An experimental assessment.", *Journal of Object Technology, vol.*11, no. 2, pp 1-38.

Gueheneuc, Y. G. (2005) "Ptidej: Promoting Patterns with Patterns. In 1st ECOOP workshop on Building a System using Paterns". *Springer- Verlag.* !

Fowler, M. and Beck, K. (1999) "Refactoring: Improving the Design of Existing Code", Addison-Wesley.

Gyimothy, T., Ferenc, R. and Siket, I. (2005) "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Software Eng.,* vol. 31, no. 10, pp. 897–910.

Hassan, A. E. and Holt, R. C. (2005) "The top ten list: Dynamic fault prediction," *International Conference on Software Maintennace*, pp. 263–272.

Janes, A., Lenarduzzi, V. and Stan, A.C. (2017) "A Continuous Software Quality Monitoring Approach for Small and Medium Enterprises", Monitoring in Large-scale Software Systems, (MOLS'17).

Kim, K., Zimmermann, T., Whitehead, J., and Zeller, A. (2007) "Predicting faults from cached history", *International Conference on Software Engineering,* pp 489–498.

Kim, S., Zimmermann, T., Pan, K., & James Jr, E. (2006) "Automatic identification of bug-introducing changes", *International Conference on Automated Software Engineering*, pp. 81-90.

Lavazza, L., Morasca, S., Taibi, D., and Tosi, D. (2010) "Predicting OSS Trustworthiness on the Basis of Elementary Code Assessment", ESEM2010, art.36, pp 1-4.

Lavazza, L., Morasca, S., Taibi, D., and Tosi, D. (2010) "Applying SCRUM in an OSS development process: An empirical evaluation", *Agile Processes in Software Engineering and Extreme Programming. XP 2010* pp. 147-159.

Lavazza, L., Morasca, S., Taibi, D., Tosi D. (2012) "An empirical investigation of perceived reliability of open source Java programs", 27th Annual ACM Symposium on Applied Computing (SAC'12), pp. 366-371

Lenarduzzi, V., Sillitti, A., and Taibi, D. (2017) "Analyzing Forty Years of Software Maintenance Models", in 39th International Conference on Software Engineering, Buenos Aires, Argentina, 2017.

Lenarduzzi, V. and Taibi, D. (2016) "MVP Explained A Systematic Mapping Study on the Definitions of Minimal Viable Product", *42th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 112-119.

Morasca, S., Taibi, D., and Tosi, D. (2009) "Towards certifying the testing process of Open-Source Software: New challenges or old methodologies?" ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development, Vancouver, BC, 2009, pp. 25-30

Nagappan, N. and Ball, T. (2005) "Use of relative code churn measures to predict system defect density," *International Conference on Software Engineering*, pp. 284–292.

Nagappan, N., Ball, T. and Zeller, A. (2006) "Mining metrics to predict component failures," *International Conference on Software Engineering*, pp. 452–461.

Neuhaus, S., Zimmermann, T., Holler, C. and Zeller, A. (2007) "Predicting vulnerable software components", *ACM conference on Computer and communications security*, pp. 529–540.

Moser, R., Pedrycz, W. and Succi, G. (2008) "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," *International Conference on Software Engineering*, pp. 181–190.

Ohlsson, N. and Alberg, H. (1996) "Predicting fault-prone software modules in telephone switches," *IEEE Trans. Software Eng*., vol. 22, no. 12, pp. 886–894.

Subramanyam, R. and Krishnan, M. S. (2003) "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Software Eng.,* vol. 29, no. 4, pp. 297–310.

Taibi, D., Lenarduzzi, V., Diebold, P. and Lunesu, I. (2017) "Operationalizing the Experience Factory for Effort Estimation in Agile Processes". *Int.Conference on Evaluation and Assessment in SW Engineering (EASE2017).*

Taibi, D., Lenarduzzi, V., Janes, A., Liukkunen, K. and Ahmad, M.O. (2017) "Comparing Requirements Decomposition within the Scrum, Scrum with Kanban, XP, and Banana Development Processes". *Agile Processes, in Software Engineering, and Extreme Programming (XP2017)*.!

Taibi, D., Lenarduzzi, V., Liukkunen, K. and Ahmad, M.O. (2017) "Comparing Communication Effort within the Scrum, Scrum with Kanban, XP, and Banana Development Processes". *International Conference on Evaluation and Assessment in Software Engineering (EASE2017)*.

Taibi, D., Lenarduzzi, V., Matta, M. and Lunesu, I. (2015) "Functional Size Measures and Effort Estimation in Agile Development: a Replicated Study". *Agile Processes, in Software Engineering, and Extreme Programming (XP2015)*, pp 105-116.

Tosi, D., Lavazza, L., Morasca, S. and Taibi D. (2012) "On the definition of dynamic software measures", International symposium on Empirical software engineering and measurement, pp. 39-48

Taber, W. and Port, D. (2014) "Empirical and face validity of software maintenance defect models used at the jet propulsion laboratory," Int, Symposium on Empirical Software Engineering and Measurement, art.7, pp 1-7.

Tian, Y., Chen, C. and Zhang, C. (2008) "AODE for Source Code Metrics for Improved Software Maintainability," International Conference on Semantics, Knowledge and Grid, pp. 330-335.

Zimmermann, T., Premraj, R. and Zeller, A. (2007) "Predicting defects for eclipse," *International Conference on Predictive Models and Data Analysis in Software Engineering*, p. 76.